Java server benchmarks

by S. J. Baylor
M. Devarakonda
S. J. Fink
E. Gluzberg
M. Kalantar
P. Muttineni
E. Barsness
R. Arora
R. Dimpsey
S. J. Munroe

The Java™ platform has the potential to revolutionize computing, with its promise of "write once, run anywhere"™ development. However, in order to realize this potential, Java applications must demonstrate satisfactory performance. Rapid progress has been made in addressing Java performance, although most of the initial efforts have targeted Java client applications. To make a significant impact in network computing, server applications written in the Java language, or those using Java extensions, frameworks, or components, must exhibit a competitive level of performance. One obstacle to obtaining this goal has been the lack of well-defined, server-specific, Java benchmarks. This paper helps address this shortcoming by defining representative Java server benchmarks. These benchmarks represent server application areas, including Web-based dynamic content delivery (servlets), business object frameworks, and multitier transactional data acquisition. Where applicable, we present benchmarks written using both the Java programming model (i.e., servlets) and the legacy model (i.e., the Common Gateway Interface) for direct comparisons of delivered performance. We also present performance measurements and analysis from multiple IBM server platforms, including both uniprocessor and multiprocessor systems.

With its promise of "write once, run anywhere"** portability, the Java** platform has the potential to revolutionize computing. However, in order to realize this potential, Java solutions must demonstrate satisfactory levels of performance. Java performance has improved rapidly in recent years, but most of the effort targets Java client applications. In order to make a significant impact in network computing, Java server applications must exhibit a competitive level of performance. To achieve this goal, the Java community needs tools and methodologies to (1) identify and eliminate performance bottlenecks, (2) compare competing Java platforms, (3) quantify the performance of Java solutions relative to legacy programming models, and (4) determine the impact of the Java language on systems software and the processor architecture.

This paper reviews our efforts to meet these objectives through server-oriented benchmarks and an associated methodology for data collection and analysis. We have developed several benchmarks that exercise the Java server software environment, including the Java virtual machine (Jvm), core class libraries, the just-in-time (JIT) compiler, and multiple Java server components such as Java Database Connectivity (JDBC**) and the servlet environment.

Although a slew of Java benchmarks have appeared, most focus on performance issues for clients and fail to provide adequate measurement for the server environment. For example, CaffeineMark**[1] and

jBYTEmark[2] are microbenchmark suites that include tests for integer arithmetic, method calls, and graphics. The JMark** suite[3] includes similar processor targeted tests, as well as a set of tests that quantify the Abstract Window Toolkit (AWT) graphics performance. Possibly the most often quoted client benchmark is the SPECjvm98[4] suite. The Standard Performance Evaluation Corporation (SPEC) developed this suite of medium-sized benchmarks to measure the efficiency of the Jvm, JIT compiler, operating system, and hardware working in concert. However, SPECjvm98 does not address server issues such as multithreaded object allocation or concurrent resource access from hundreds or thousands of threads. Furthermore, it does not utilize any of the major Java server extensions such as JDBC or servlet support.

In the server realm, a popular Java benchmark is VolanoMark**,[5] a chat server developed by Volano LLC. This benchmark has been used as a tool to focus on inefficiencies within the Jvm and to provide a basis for comparing Java platforms. It has been especially useful in addressing issues associated with running thousands of threads. While VolanoMark does address some issues associated with the Java platform executing on servers, it does not adequately reflect the diverse types of Java server applications written and available today. In addition, the extreme usage by VolanoMark of the underlying Transmission Control Protocol/Internet Protocol (TCP/IP) limits its utility. SPEC is currently identifying a suite of benchmarks that will address the Java server-specific issues mentioned above. Until it is complete, Java platform developers need a diverse suite of Java server benchmarks that are representative of real applications.

To address this concern, we have developed several Java server benchmarks that represent current programming models and stress key Java server technologies. They include a business object benchmark for Java (jBOB), a portable business object benchmark (pBOB), and a Web application server benchmark (JWeb2). We have also developed several microbenchmarks and an associated framework for running, analyzing, and developing future microbenchmarks. Whereas the macrobenchmarks serve as the primary vehicle for quantifying and communicating Java server performance, the microbenchmarks help isolate performance problems illuminated by the macrobenchmarks.

This paper reviews our Java server benchmarks and presents performance results on four IBM server platforms. One of the key results of the paper indicates that the Java servlet paradigm outperforms the Common Gateway Interface (CGI) paradigm when tested with functionally similar scripts. In addition, it is shown that the DATABASE 2* (DB2*) JDBC implementation is comparable in performance to the call level interface (CLI) for some key functions, but the JDBC application programming interface (API) does not offer as much functional flexibility as the CLI. The paper also shows that the improvements made to the core Jvm (e.g., monitor implementation, object allocation) and JIT compiler by IBM have resulted in the highest-performing Java server platforms. Finally, the results also indicate that significant impediments to the performance of the Java platform remain. Most notable is the performance of remote method invocations (RMIs) when compared to C-level remote procedure calls (RPCs).

The next section presents a detailed discussion of the macrobenchmarks used in this study and benchmark results on several platforms. The subsequent section discusses the microbenchmarks and accompanying results. The conclusion forms the last section and summarizes the work presented, the lessons learned, and suggestions for further work.

Macrobenchmarks

This section presents three Java server macrobenchmarks: the business object benchmark for Java (jBOB), the portable business object benchmark (pBOB), and a Web-server benchmark (JWeb2). These benchmarks model typical commercial server applications currently available in the industry. We discuss the implementation of each benchmark and the workload that it represents. For each benchmark, we present performance results, comparing the Java solution to a comparable solution written with a legacy programming language.

Business object benchmark for Java. The jBOB macrobenchmark is designed to quantify the performance of simple transactional server applications written in the Java language for a typical electronic business scenario. The workload consists of a mixture of read-only and update-intensive transactions that simulate the activities found in complex on-line transaction processing (OLTP) application environments. In particular, the application uses the business model of the Transaction Processing Performance Council's TPC-C** benchmark.[6] In accordance with the TPC's fair use policy, we note that jBOB deviates from the TPC-C specification and is not com-

parable to any official TPC result. In addition, the jBOB application has not been structured to drive the highest possible throughput, but instead to reflect a more typical customer usage of Java. The benchmark is representative of applications exercising a breadth of system components characterized by:

- Concurrent transactions
- On-line and deferred transaction execution modes
- Moderate system and application execution times
- Transaction integrity
- Nonuniform distribution of data access through primary and secondary keys
- Databases consisting of many tables with a wide variety of sizes, attributes, and relationships

The jBOB and TPC-C benchmarks are designed for different computational models. TPC-C is designed to encourage implementers to exploit maximum performance from the hardware and software of the machine within the bounds of the specification. TPC-C is an end-to-end benchmark that quantifies the full transaction response time, including the performance of the client, the network, and the server. jBOB does include application drivers that generate and submit transaction requests from clients to the server; however, its objective is to quantify the performance of the server-side application. As a result, the performance of the client and the network is not intended to be part of the metric. In addition, the jBOB and TPC-C metrics are different. TPC-C only measures the throughput of new order transactions, whereas jBOB measures the throughput of all five transaction types. jBOB was coded using the ACID (atomicity, consistency, isolation, and durability) properties of TPC-C; however, these properties are not tested as part of the benchmark. There is no concept of a specific user transaction request in jBOB, so no transaction monitor is required to handle the routing of requests. Finally, jBOB collects more detailed information on its transactions, facilitating a better understanding of the results.

The jBOB implementation contains three logical tiers. The first tier, the clients, generates data access requests modeled after TPC-C transaction types. The second tier, a thin layer of Java code, acts as a front end to the third tier database server. The benchmark exercises the JDBC or the Structured Query Language (SQL) embedded in the Java (referred to as SQLJ) interface, Java communication using either RMI or Java sockets, core Jvm and class-level synchronization, data conversion, multithreading, object serialization, object creation, and garbage collection. The benchmark reports performance metrics of throughput and transaction response time.

In order to compare jBOB with equivalent benchmarks using established technologies, we have developed a number of benchmarks that use C++ and C with the Open Database Connectivity (ODBC) API, embedded SQL, and stored procedures for accessing data. This collection of Java, C++, and C benchmarks allows us to address a number of issues as outlined above.

Architectural and design issues. jBOB follows the client/server model with the user terminals as clients and the transaction logic as server. jBOB is typically run in one of two modes: (1) a logical two-tier, physical one-tier mode, and (2) a logical three-tier, physical two-tier mode, where the second and third tiers representing the application server and database reside on the same physical tier. The benchmark implementation supports several mechanisms for database connectivity, communication, and a family of application configurations.

Database connectivity. jBOB has the flexibility to use either JDBC or SQLJ to access the third-tier database from Java. To keep jBOB portable across platforms and database products, the benchmark uses ANSI/ISO (American National Standards Institute and International Organization for Standardization) standard SQL statements to access the database. jBOB has been successfully run on a number of IBM platforms and current versions of DB2.

Communication. The communication between the first and the second tiers in jBOB can occur using either an RMI or sockets-based implementation. Sockets allow for more flexible designs than RMI, because sockets can support asynchronous communication. Additionally, the sockets mode allows for reuse of sockets to a greater extent than RMI mode does and, hence, is likely to scale better.

Application configuration. In addition to database connectivity and communication, jBOB parameters control the application configuration, such as two-tier versus three-tier, think times, asynchronous deliveries, and the number of processes and threads in each tier. Wherever possible, we have used default values to correspond to the TPC-C specification. These default values allow for cross-platform result comparisons.

We have developed equivalent benchmarks for jBOB in C++ and C using ODBC and embedded SQL to

Table 1 E-business benchmark options

Language	Java	C++	C
Number of possible tiers	1 2 3	2 3	2
Database connectivity	JDBC (jBOB(JDBC)) SQLJ—no stored procedures (jBOB(SQLJ))	ODBC (CEB(ODBC)) Embedded SQL—no stored procedures (CEB(ESQL))	ODBC with stored procedures (OEB) Embedded SQL with stored procedures (C(TPC-C))
Available communication between 1st and 2nd tiers	RMI Sockets	Sockets	Sockets

facilitate a comparison with Java and static and dynamic SQL. Table 1 shows a summary of the variations of this benchmark, where Java, C++, or C is used as a primary language. Within the database connectivity row, the valid alternatives to run the benchmark are listed, along with the name used to refer to that version shown in parentheses.

jBOB results. Of the variations listed in Table 1, we present results for jBOB (JDBC) and the C Enterprise Benchmark (CEB) (ODBC). The jBOB configuration uses JDBC in the logical three-tier mode on two physical tiers. A single Jvm is used for the second tier, and the physical tiers communicate through sockets. The CEB results presented are from an analogous logical three-tier, two physical tier configuration.

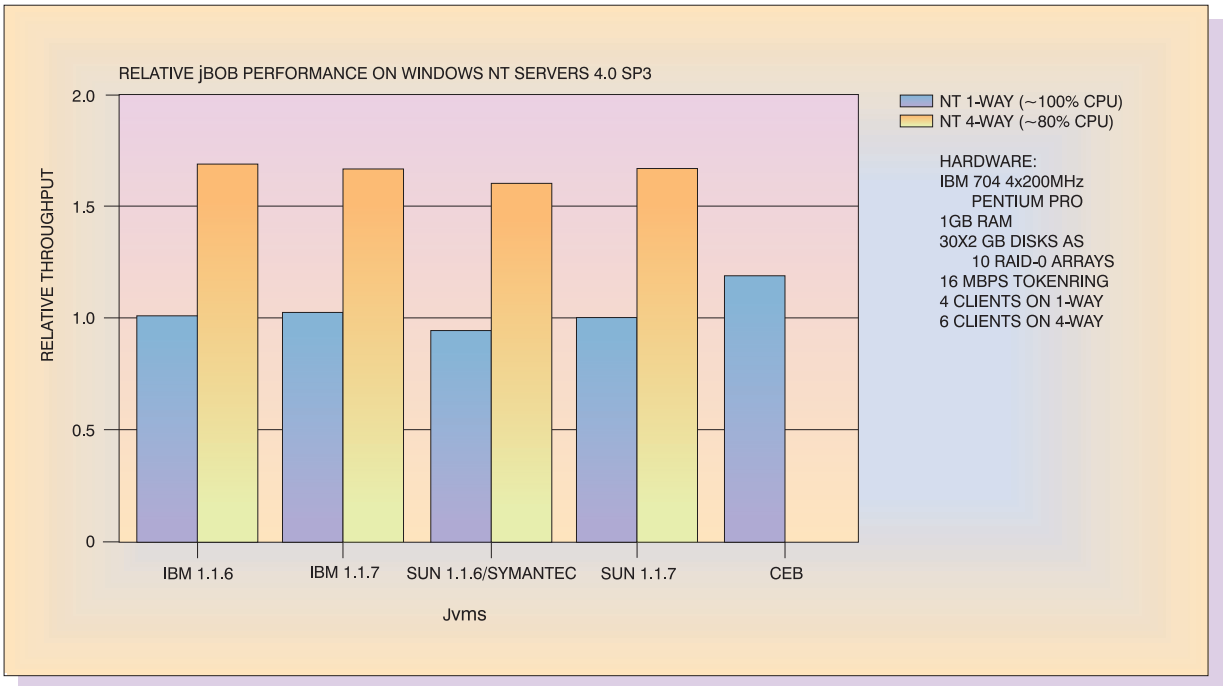
jBOB was run on four different Jvms: the Jvm in the IBM Developer Kit (DK) for Windows NT**, Java Technology Edition, version 1.1.6, the Jvm in the IBM DK for Windows NT, Java Technology Edition, version 1.1.7, the Jvm in Sun's Java Development Kit (JDK**), version 1.1.7, and the Jvm in the Symantec Development Kit, version 1.1.6. Both physical tiers, client and server, ran on Windows NT with Service Pack 3.0. DB2 version 5.0 Fixpack 9014 served as the underlying database. Figure 1 summarizes the data collected on both one-way and four-way 200-MHz Netfinity* systems with 1 GB of memory and 48, 2-GB disks. Four to six client machines were used to drive the server. All data are presented relative to one-way results on Sun JDK 1.1.7 (i.e., this result is shown as one).

The first point of interest that leaps from the chart is that the underlying Jvm employed does not have a major impact on jBOB performance using the DB2 JDBC drivers. All Jvms tested show comparable performance. We have determined that the underlying JDBC implementation dominates the jBOB performance, and the JDBC implementation is held constant across the Jvms measured. To isolate JDBC performance factors, we have developed ODBC/JDBC microbenchmarks, which are presented in the next section.

It is also interesting to note that multiprocessor speedup reaches only 1.7 out of 4. Although this result shows some speedup, it does not approach the 2.5 speedup that one would normally hope for on this type of benchmark on this platform. The most interesting piece of data to be gleaned from the chart is the comparison between jBOB and CEB. This comparison provides a reliable quantification of the performance penalty paid for the benefits of Java. CEB, based on a mature machine-dependent standard, is only 20 percent faster than Java on a one-way system. Note that since the server spends significant time in the third tier, which is common for both workloads, the 20 percent difference can be regarded as a lower bound on Java and JDBC versus C++ and ODBC performance differences. Nevertheless, the end user observes a 20 percent difference on the application performance, which is the most relevant figure for this Java vs C comparison. By analytically isolating the second tier of the benchmark, we found the C++ and ODBC combination to be approximately 35 to 40 percent faster than the Java and JDBC implementation. The difference is attributable to a combination of JDBC architecture, implementation, and Java and C++ differences.

In addition to the results based on Windows NT shown here, AS/400* jBOB results have been published in Reference 7. Also note that, although not available in time to include in this paper, initial results

Figure 1 jBOB results: one-way and four-way database throughput



measured from DB2 Version 6.1 show impressive JDBC performance improvements.

Portable business object benchmark. The pBOB macrobenchmark provides a pure Java implementation of a typical transaction-oriented workload. Like jBOB, pBOB is inspired by the business model of the TPC-C benchmark specification but is not a TPC benchmark. pBOB was designed to analyze the processor-memory subsystem, and associated software, for typical server-based business applications. This use is in contrast to jBOB, which is used to study the complete server configuration.

Each of these three benchmarks, pBOB, jBOB, and TPC-C, focuses on a different component of the computing solution; however, all are associated with transactional business applications. pBOB measures scalability, performance, and throughput of the Java environment. jBOB measures scalability, performance, and throughput of the Java environment when it interacts with the underlying hardware and software stack. TPC-C measures the maximum scalability, performance, and throughput of the entire system, including network and client performance, in addition to the hardware and software stack of the application.

To achieve a pure Java implementation, pBOB replaces third-tier database tables with in-core Java classes and records. There is no persistence or transaction control; instead, all transaction domain instances live only for the duration of the Jvm instance. pBOB does not perform terminal emulation or communication. Instead, Java threads represent terminals in pBOB, where each thread independently generates random input before calling transaction-specific logic. pBOB has no external dependencies and is portable to any compliant Jvm.

The pBOB workload is controlled by varying the number of warehouses, the object population per warehouse, the number of threads (representing terminals) per warehouse, keying and think times, number of order lines per order, and whether initial and transaction result screens are displayed. pBOB also supports a "steady state" mode where the creation of new objects is balanced by the removal of old objects. This mode allows long-duration tests to run in finite heap space.

The pBOB application supports various configurations. Key application parameters include:

- *Object populations*—pBOB supports company and warehouses at 1 percent, 10 percent, and 100 percent of the benchmark-specified populations. Populations of 100 percent (approximately 2.4 million Java objects requiring 200 MB of heap for the company and first warehouse) are impractical for most systems and testing without a persistence framework. Populations of 1 percent and 10 percent are practical for most client PCs and any server platform. Smaller (1 percent) populations and warehouse counts minimize the impact of garbage collection, whereas larger (10 percent) populations and warehouse counts illuminate the behavior of a garbage collection implementation. A 1 percent population contains approximately 27000 objects requiring 2.2 MB of heap for the company and first warehouse. Each additional warehouse generates 20000 objects requiring 1.8 MB of heap. A 10 percent population is approximately 244000 objects requiring 20 MB of heap for the company and first warehouse. Each additional warehouse generates 197000 objects requiring 17.5 MB of heap.
- *Threads per warehouse*—The number of threads (terminals) per warehouse can be varied from 1 to *N*. The total number of threads for a measurement is *Terminals_per_warehouse* ∗ *Warehouse_count* + *1*. pBOB is currently limited to 2048 threads plus the main thread. Running a single terminal per warehouse minimizes contention at the application level and focuses on the Jvm run-time behavior. Running "single terminal" with zero think times and multiple warehouses focuses on hot-locks in the Jvm run time (heap management and class lookup) as well as the overhead for uncontended monitors. Running with "multiple terminals" with think times varying from 0 to 100 percent controls the contention at the application level and stresses the implementation of Java monitors.
- *Keying and think times*—Per-transaction keying and think times can be toggled on (100 percent) or off (0 percent) or set to a specific percentage of the specified values. Zeroing keying and think times maximizes throughput for a given Jvm/OS/system by driving it to saturation. Setting a specific (nonzero) percentage establishes an upper bound on the transactional throughput (*max_tpmBOB* = *1.27* ∗ *Warehouse_count* ∗ *Terminals_per_warehouse* ∗ *100*/*percent_wait_time*).
- *Order lines per order*—Order line processing is the inner loop of the new order, order status, delivery, and stock level transactions. The number of order lines may be set to a random number between 5 and 15 order lines per order, or set to a fixed value between 1 and 20. The setting affects both the per-transaction path length and the heap storage requirement.
- *Transaction screens*—If "screen write" is enabled, formatted initial and result screens are written to System.out, otherwise the screens are formatted, then discarded without display. Most testing is performed with "screens disabled" (the default) to maximize throughput and focus on the performance (not the specific display adapter and driver) of the Jvms.

A pBOB run consists of the following steps: (1) Run finalization and force garbage collection. (2) Create (*Warehouse_count* ∗ *Terminals_per_warehouse*) threads. (3) Run transactions for each thread independently to progress through the following phases: ramp-up, measurement, ramp-down, and accumulate terminal results to warehouse results. (4) Accumulate and display warehouse results.

Figure 2 shows a typical pBOB output for a run with one warehouse and one terminal.

pBOB supports "minimal" and "full" runs, where a minimal run has a 30-second ramp-up and a two-minute measurement period, and a full run has a two-minute ramp-up and 10-minute measurement period. Minimal runs are useful for taking a quick measurement or to make a measurement free of garbage collection by setting the heap large enough to eliminate garbage collection events from the measurement period. Results are fairly repeatable from run to run, at least for small thread counts and 0 percent think times. However, if nonzero think times are used, a two-minute measurement period may be too short for repeatable results. A full run is advisable whenever nonzero think times are used. Full runs are also useful when measuring the impact of garbage collection.

These options create a multitude of possible "measurement profiles." Two profiles are widely used within IBM: *autorun* and *autoserver*. The autorun profile provides a "*single terminal, 1 to 10 warehouse ramp*." This profile runs a single thread per warehouse series from 1 to 10 warehouses with 1 percent population and minimum run time per measurement. Think times and screen displays are disabled to maximize throughput and saturate the CPU(s).

Figure 2 pBOB results: one warehouse and one terminal

TOTALS FOR: COMPANY with 1 warehouse and 1 terminal each
.....BOB1.2 Results (time in seconds).....

New Order>	Count:	204854	Time:	378.85	Min:	0.000	Max:	2.123	Avg: 0.002
Payment>	Count:	204853	Time:	80.17	Min:	0.000	Max:	2.173	Avg: 0.000
OrderStatus>	Count:	20486	Time:	5.50	Min:	0.000	Max:	0.011	Avg: 0.000
Delivery>	Count:	20485	Time:	93.79	Min:	0.000	Max:	2.103	Avg: 0.005
Stock Level>	Count:	20485	Time:	12.77	Min:	0.000	Max:	0.011	Avg: 0.001

Maximum Qualified Throughput (MQTh) 20478.81 tpmBOB

This profile illuminates Java symmetric multiprocessor (SMP) scalability as it provides a direct comparison between single-thread/CPU and multiple-thread/CPU results. The ratio between the first warehouse result and the peak (presumably multiple warehouse) result provides a crude "SMP scale factor" for the system under test. For most systems the performance peaks where the warehouse count matches the number of CPUs, then degrades as contention in the run time and garbage collection load increases. An application server should minimize this degradation. To measure this factor, another figure of merit is the "final SMP scale factor," which is the ratio of one warehouse and the last (tenth) warehouse results. Figure 3 shows a typical profile from a pBOB autorun, comparing three hypothetical Java environments (A, B, and C).

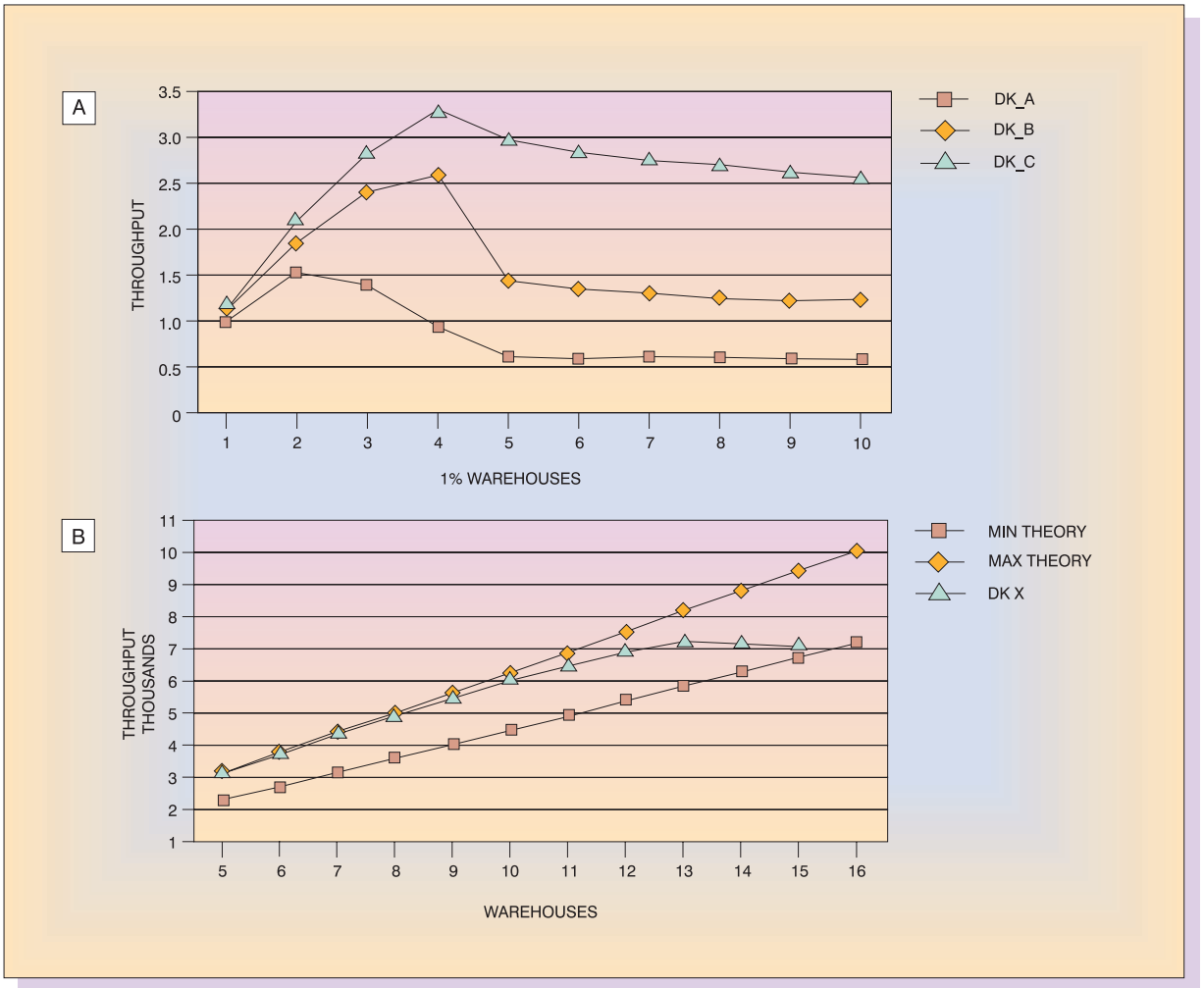
The autoserver profile targets large application servers running on mid- to high-range systems. Although the autorun profile will run on most desktop workstations, large multiuser object-oriented application servers such as the IBM SanFrancisco* product and future Enterprise JavaBeans** (EJB) servers will require larger systems and memory configurations. For the SanFrancisco product, large multiuser workloads require hundreds of threads accessing a shared object cache of multiple 100 MBs. The autoserver profile measures the Jvm performance in this stressful "Java application server" environment. A primary goal is to foster the implementation of more advanced garbage collection technology (e.g., generational and concurrent).

Since the autoserver profile stresses the garbage collection subsystem, the performance results depend strongly on the size of the Java heap and the quality of garbage collection. To further highlight garbage collection performance, the profile uses nonzero keying and think times. This sets an upper limit on throughput for a given warehouse and terminal (thread) count (recall that $max\_tpmBOB = 1.27 * Warehouse\_count * Terminals\_per\_warehouse * 100/percent\_wait\_time$). With keying and think times and fixed thread count per warehouse, the throughput grows linearly with the number of warehouses.

The current autoserver profile runs 25 threads per warehouse and keying or think times set to five percent of that TPC-C specification. We adjust other application parameters to balance CPU loading and memory footprint. Increasing the order lines per order up to 20 brings the CPU loading up nicely but also increases the memory requirement. Full 10-minute measurements are used to force larger memory configurations and to stress the garbage collection subsystem. As various groups within IBM use the autoserver profile, the choice of parameters will continue to evolve as we converge to a configuration that stresses garbage collection and achieves high CPU utilization without an explosion in memory footprint.

Figure 3B shows the result of a typical autoserver run. The result is a heavier warehouse ramp series, which initially follows the *max_tpmBOB* line, until hitting some resource limit (e.g., CPU, total heap space) or the garbage collection overhead starts to dominate. Following TPC-C, there is also a minimum ($min\_tpmBOB = 0.90 * Warehouse\_count * Terminals\_per\_warehouse * 100/percent\_wait\_time$) threshold below which response times are not being met. This min_tpmBOB line can be used as a "stop

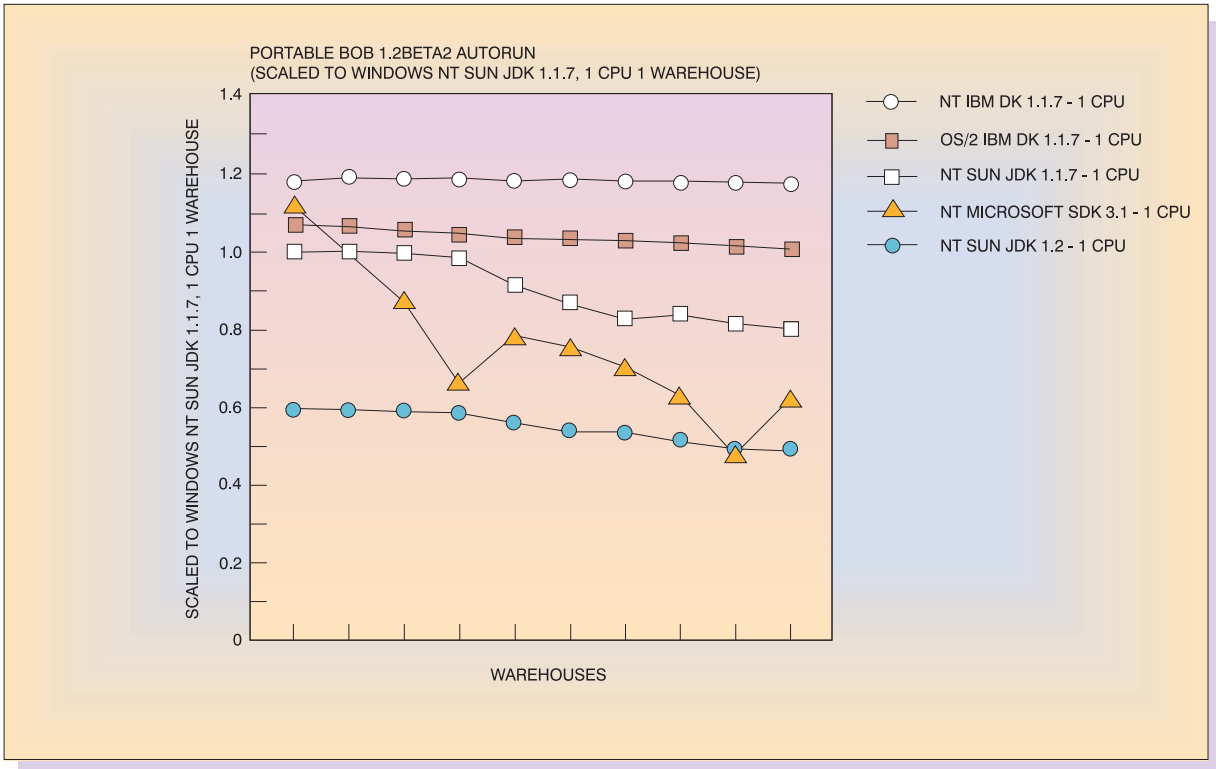
Figure 3 Results for: (A) typical pBOB autorun profile, (B) typical pBOB autoserver profile



criteria" for the profile. Heap space also limits the number of warehouses, since the profile will not run if it cannot fit the required warehouse data in the heap.

pBOB results. The autorun profile was the first and most widely used pBOB profile within IBM. Figures 4 and 5 show pBOB autorun results on five different Jvms from IBM, Sun, and Microsoft. The results are scaled to Sun JDK 1.1.7 running with one warehouse (thread) on one CPU. Results are presented from one to ten warehouses (threads). Figure 4 presents results from a single processor box, whereas Figure 5 plots the four-way results. For example, four threads running on the Jvm in the IBM DK for Windows NT, v 1.1.7 on a four-way system drives approximately 3.6 times more throughput than a single thread on a one-way system using the Sun JDK 1.1.7. For comparison, four threads on Sun JDK 1.2 on a four-way system drives less than 1.2 times more throughput than the Sun one-way configuration. All results ran on a 400-MHz Netfinity system with 4 GB of memory. The Java heaps were set to 1 GB where possible, in order to require a certain amount of garbage collection during the runs. For the Jvms in the DK for Windows NT, NT 4.0 Service Pack 3.0 is employed. For the Jvm in the IBM DK for OS/2, Warp SMP Fixpack 36 is used.

Figure 4 Uniprocessor pBOB results: 1–10 warehouses; one terminal per warehouse



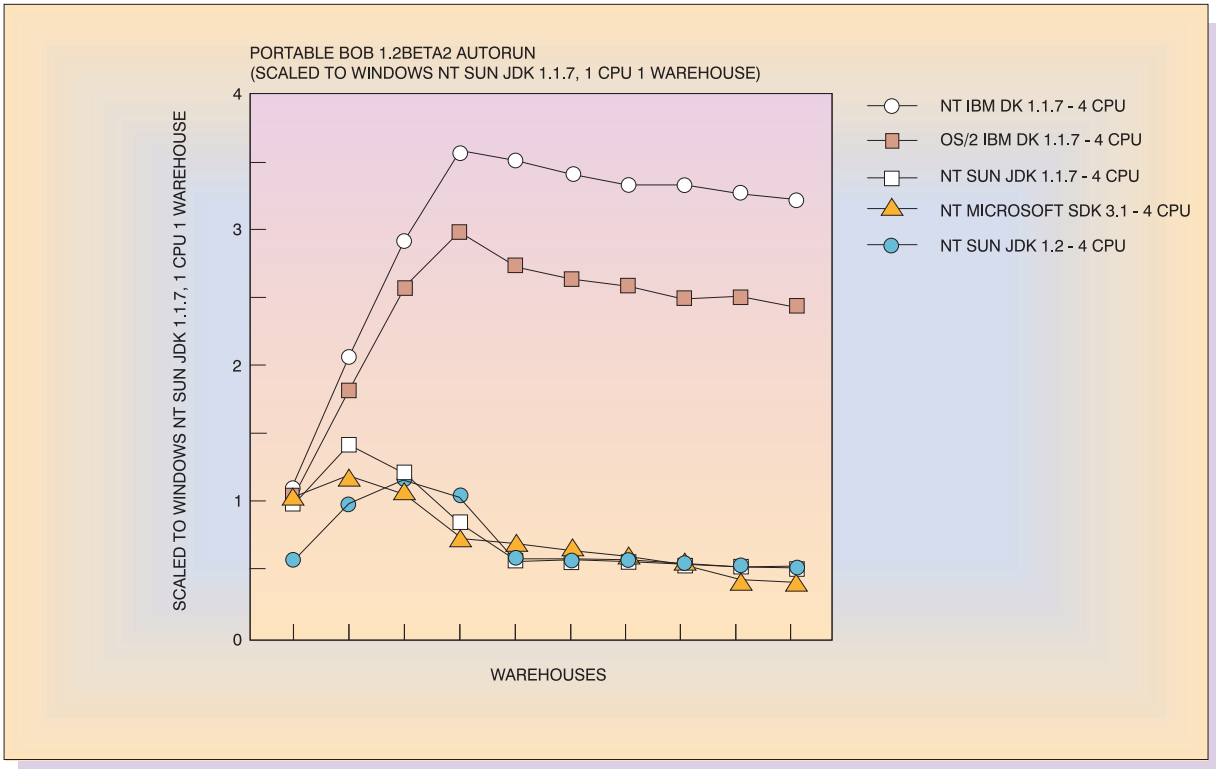
The results show that the IBM Jvms exhibit better performance in highly contended multithreaded workloads. This performance results from IBM's emphasis on core Jvm performance for server workloads (see other papers in this issue). In particular, the IBM Jvms employ efficient monitor and object allocation implementations, which are especially important for good performance on this benchmark. The IBM Jvms allow a high degree of parallelism in object allocation, which results in higher pBOB performance.

The Jvm in the IBM DK for OS/2* (Operating/System 2*), v 1.1.7 closely trails the Jvm in the IBM DK for Windows NT. The OS/2 implementation lags the IBM implementation for Windows NT because it lacks some locking enhancements added to the Jvm in the IBM DK for Windows NT. The Microsoft Jvm does especially poorly on this benchmark because it suffers from poor heap management. An unexpected result is the significant degradation in performance seen from the JVM in Sun's JDK 1.1.7 to Java 2 version 1.2. This degradation most likely results from extra security checks in the *Class.newInstance*( ) method implemented as part of the Java 2 security model. pBOB exercises a significant number of these methods to allow more flexible, dynamic object allocation.

The multiprocessor scaling numbers (ratio of throughput on a four-CPU system compared to a single-CPU system) also show IBM Jvm implementations exhibiting better performance. For instance, the Jvm in the IBM DK for Windows NT is nearly three times faster on four CPUs than on one. This scalability rivals mature operating systems on similar workloads. At ten threads, the IBM implementation is six to seven times faster than the others. For non-IBM Jvms, scaling falls off when many threads are executed. This trait is especially devastating on server workloads where it is expected that hundreds, if not thousands, of threads will run in a single Jvm.

Web server benchmark. At present, the most common technology for dynamic Web page creation is

Figure 5 Four-way pBOB results: 1–10 warehouses; one terminal per warehouse



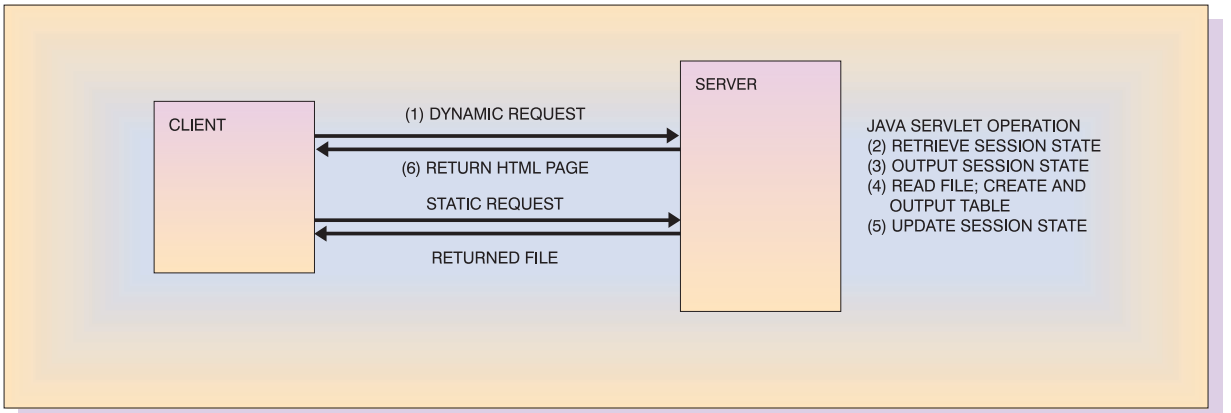
CGI.[8] Other mechanisms include FastCGI,[9] server-specific APIs such as ISAPI,[10] Microsoft Active Server Pages,[11] and Java servlets.[12] Each of these technologies is server- or platform-specific except for Java servlets. Since servlets can execute on any platform, they are an attractive choice for developing portable Web applications that deliver dynamic content.

JWeb2 is a dynamic Web-serving benchmark specification designed to quantify the relative performance of the programming alternatives listed above. The benchmark characterizes the behavior of a simple e-commerce application whose clients are Web browsers based on HyperText Transfer Protocol (HTTP). The JWeb2 specification has been implemented using Java servlets and CGI. In addition to comparing programming models, the benchmark helps compare the relative performance of Jvm implementations.

JWeb2 is derived from the SPECweb99[4] benchmark in which simulated clients access a series of HTML (HyperText Markup Language) pages specified by URL (Uniform Resource Locator). JWeb2 has enhanced this by defining an e-commerce application and supplying implementations of this application as a servlet and a CGI application. JWeb2 exercises Web server-based Java program execution, multiple threads, session management, file I/O, and the capability of a server to handle high HTTP traffic and memory access bandwidth.

Design of the JWeb2 benchmark. Designing a Java Web server benchmark requires balance between two conflicting objectives. First, we want a benchmark that measures how typical e-commerce Web servers behave. Second, we are primarily interested in how Java performs in a server environment. Therefore, it may not make sense to model a Web server too closely. In particular, most Web servers serve a majority of static pages. To study how Java performs, we model a Web server with higher dynamic content.

Figure 6 Diagram of a JWeb2 operation



JWeb2 tries to balance these two objectives in the following ways. First, it uses 50 percent dynamic content and 50 percent static content, based on the assumption that each dynamic page will typically refer to at least one image. Second, the sizes of the static files and the results of related dynamic requests follow a distribution similar to that expected on a typical Web server. In particular, distributions were chosen to match those used by the SPECweb99 benchmark.[4] Last, the activities of a typical e-commerce program are abstracted and developed in the benchmark servlet.

A typical e-commerce Web site uses "cookies" or URL rewriting to maintain session state information about each user. If passwords or cookies are used, the Web server may rely on a database of individual preference. Typical Web server application activities include text manipulation, page development, data lookup (either via a database or file access) and table generation. Communication with clients may or may not be encrypted. Most interactions with an e-commerce Web site involve a number of steps. The user first logs on, creating a new session. Second, the user may search the Web site, access information, or engage in a transaction. Finally, the user logs off. Each of these activities may be carried out by a different dynamic program.

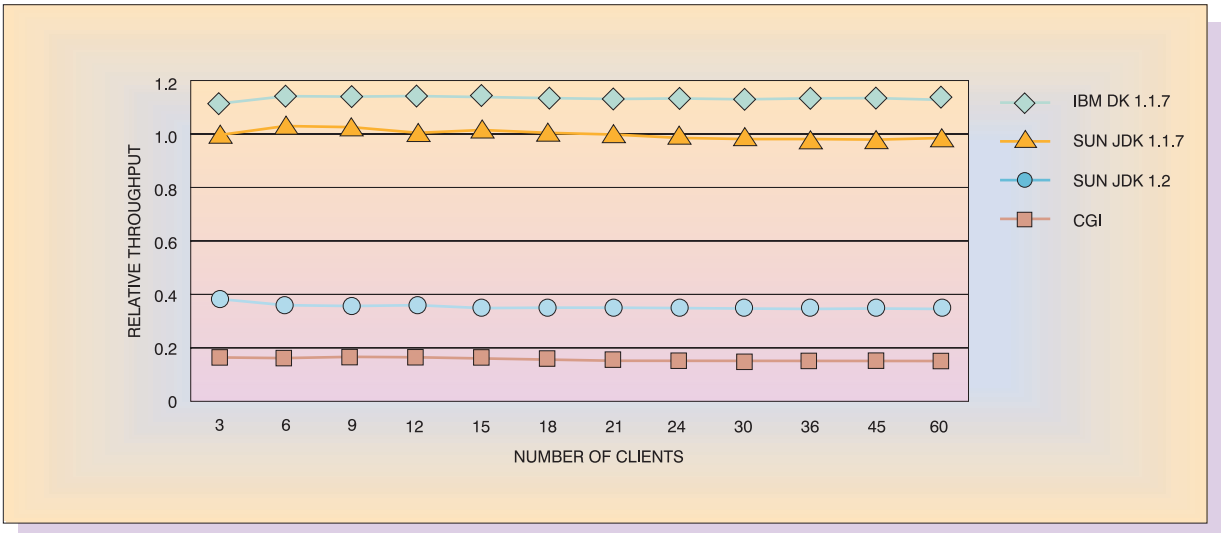
The JWeb2 benchmark abstracts the above activities into a single Java servlet. This servlet first manages session information: it uses a cookie to identify the session, creating a new one if necessary. Second, the session state, a record of all previous accesses during the current session and the number of rows in the reply, is read, and the contents output to a newly created HTML page. Third, the parameters to the dynamic request are used to access a file. The file is read and is used to generate a summary table. Finally, if the session is to be terminated, all session state information is invalidated. Otherwise, information about the current request is added to the session. These steps are illustrated in Figure 6.

Note that information in the response is derived from a file, not a database. Most e-commerce Web servers will probably obtain their data from a database. However, we made the decision to access files because we independently investigated the behavior of database accesses with the jBOB benchmark. In addition, the file-based access is representative of the solution IBM provided at the Winter Olympics held in Nagano in 1998.

CGI program. For comparison purposes, we implemented a CGI program written in C that implements the same functionality as the Java servlet. In the CGI model, each new request results in the initiation of a new process to handle the request. This model differs from the servlet model, in which a set of Java threads is always running and any thread can handle new requests. Thus, the servlet implementation reduces the overhead of process initiation. However, a CGI program has the advantage of expecting a C program to run faster than a Java program.

Since a new process starts for each request, a CGI program cannot, by itself, implement session man-

Figure 7 JWeb2 uniprocessor results



agement. Instead, some form of state needs to be maintained, either through the file system or in a database. We chose to implement a session server to manage session requests and to ensure consistency. This session server maintains an in-core database of the session state. CGI programs communicate with a set of stateless commands over TCP. We believe that this activity closely resembles the implementation of the Java servlet session API. We attempted to minimize the cost of the C program. For example, all fixed text strings are hand-coded in raw ASCII codes to ensure that no conversion will be necessary on any platform; when conversion is necessary, it is done once and cached for future reference. Also, memory is largely preallocated. Additional memory is allocated only when sessions grow very large. In these cases, the additional memory is not released; it can be reused by other sessions.

However, efforts at efficiency were countered by a desire to execute the same program, with minimal changes, on several server platforms. In the interests of portability, we accepted the following implementation choice:

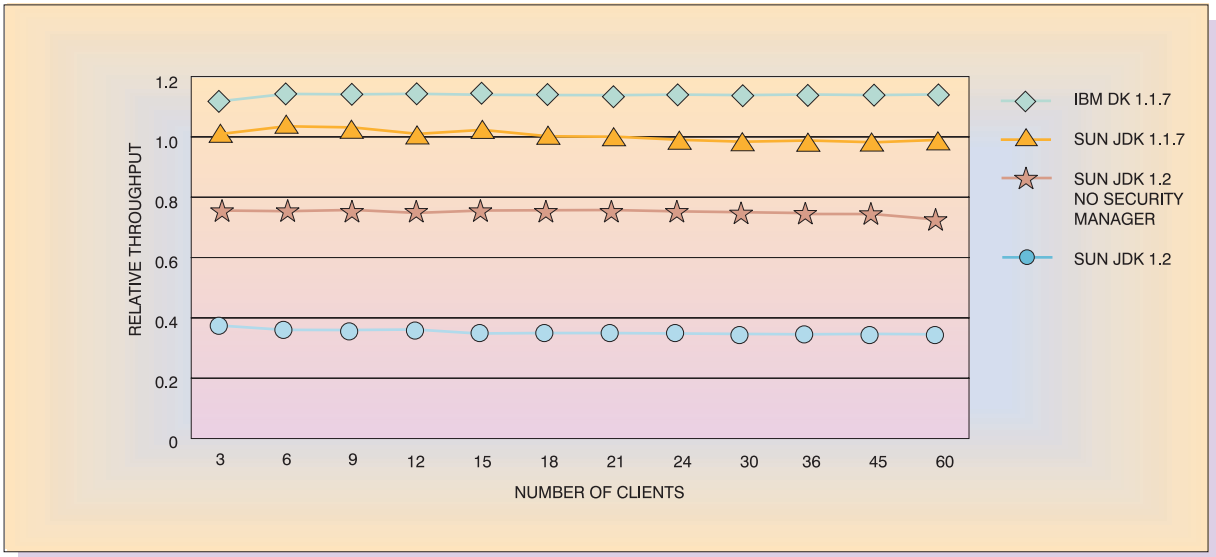
- TCP was used for communication instead of native inter-process communication (IPC) mechanisms.
- Blocking sockets were used as the interface to TCP.
- The session server is single threaded.
- The sprintf( ) and printf( ) commands are used more than might be necessary on any one platform.

JWeb2 results. In order to compare the efficiency of the software stack on the server, we made certain that the benchmark would drive the server CPU to saturation in all runs. We did this by providing a sufficient number of physical client drivers and sufficient network bandwidth. Although the JWeb2 dynamic code performs a significant amount of file read I/O, we made certain that the server was not I/O bound by providing a large file system cache and using a RAID-0 array of four physical disk drives. (A RAID is a redundant array of independent disks.) Finally, we equipped our server with sufficient physical memory to prevent any swapping.

We report results on an IBM PC Server 704, 4 × 200-MHz Pentium Pro**, with 512K L2 cache and 1 GB RAM. The JWeb2 file set data resided on a 4 × 4 GB RAID-0 drive. Two dedicated 100 Mb/s Ethernet segments were used. Three 333 MHz Pentium** II clients were used to drive the server and were distributed across both segments.

The server ran Windows NT 4.0 Server SP4, with Microsoft IIS** (Internet Information Server) 4.0 (MS Option Pack 4.0) as the underlying Web server, and WebSphere* Application Server 2.0 Standard Edition as the servlet engine. The servlet engine was configured to run out of process, i.e., in a process separate from the Web server. The Java heap sizes were set to 200 MB initial, 450 MB maximum. Three Jvms

Figure 8 JWeb2 uniprocessor: effect of disabling WebSphere security manager



were used to run the servlet engine: IBM DK 1.1.7, Sun JDK 1.1.7, and Sun Java JDK 1.2-V.

We present results for the cases where the JWeb2 logic ran as a CGI application launched by IIS 4.0, as well as where it ran as a servlet within the WebSphere Application Server. In both cases, the total offered load was held constant while the total number of simulated clients varied. This arrangement tested the ability of the server to maintain throughput in the presence of an increasing number of clients. Experiments ran both with the server running as a uniprocessor and as a four-way SMP. In all cases, the measured throughput (JWeb2 operations per second) is displayed relative to the throughput of Sun JDK 1.1.7 at three clients.

Figure 7 shows the performance results on a uniprocessor. Several conclusions emerge. In general, Java servlets outperform CGI implementations. This conclusion validates the advantage of Java servlets for dynamic Web serving. Moreover, there are significant differences between Java implementations. Clearly this benchmark exposes differences in the efficiency of Jvm implementations. The superior performance of the IBM DK 1.1.7 is consistent with results on other benchmarks. The poor performance of Sun JDK 1.2 was, however, unexpected.

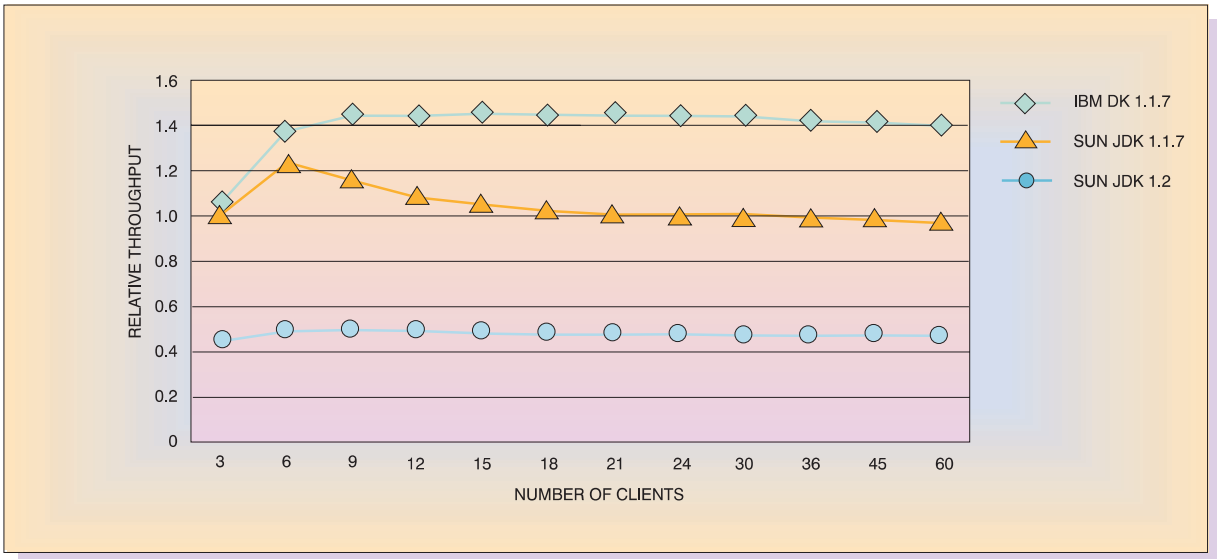
Further investigation revealed an interesting finding. WebSphere implements a security manager that is enabled by default. Disabling this security manager left the 1.1.7 results unaffected, but led to a pronounced improvement in the JDK 1.2 results as shown in Figure 8.

Figure 9 shows the results of running the workload on a four-way SMP configuration. The conclusions from the uniprocessor case still hold. In addition, the effects of lock contention at larger numbers of clients now become apparent. Throughput drops significantly after six clients on the JDK 1.1.7 for the Sun JVM, whereas the Jvm in the IBM DK 1.1.7 displays good scaling. This is consistent with the fact that the Jvm locks and Java monitors in the Jvm in the IBM DK 1.1.7 have been highly optimized. As in the uniprocessor case, the throughput of the JVM for the Sun JDK 1.2 is significantly lower because of the effect of the WebSphere security manager. We did not measure the effect of removing the security manager in this case.

Microbenchmarks

Macrobenchmarks are designed to model realistic application workloads. Although this design offers the best measure of system performance, large pro-

Figure 9 JWeb2 four-way SMP results



grams comprising multiple interacting systems can complicate performance analysis. To simplify performance analysis, microbenchmarks can isolate and identify specific system bottlenecks. However, microbenchmarks cannot capture complex system interactions and offer most value when used in conjunction with more realistic workloads.[13] This section describes several microbenchmarks developed to facilitate a focused analysis of Java server performance. They include *jMocha*, a microbenchmark suite designed to assess the performance of the portable operating system services of Java, *SockPerf*, a networking microbenchmark, and a collection of microbenchmarks designed to evaluate the performance of JDBC and ODBC connectivity.

The jMocha microbenchmark suite. Server applications depend heavily on operating system (OS) services, such as file systems, network protocols, and remote procedure calls. The performance of these services may constrain the performance of the application. To promote portability, the standard Java libraries provide a common API to system services. To understand server application performance, we must understand the performance of the Java interface to system services.

The jMocha microbenchmark suite assesses the cost of the portable operating system services of Java. The following subsections describe previous work on Java and OS microbenchmarks, present our microbenchmark methodology, and discuss a few implementation details.

Previous work. Prior to the introduction of the Java language, a few projects developed microbenchmark suites to evaluate OS performance. Ousterhout[14] developed a set of microbenchmarks to help evaluate the Sprite operating system. His work exposed memory system performance, disk cache structure, and network protocols as crucial factors in OS performance. McVoy and Staelin[15] developed *lmbench*, a portable set of operating system benchmarks focusing on issues, including memory system, IPC, cached I/O, system call, signal handling, network, and process and thread performance. Brown and Seltzer[16] built further on this work with *hbench:OS*, which improved some methodological and practical issues of lmbench.

Saavedra and Smith[17] present microbenchmarks to evaluate memory system performance, exposing performance at the various levels of the memory hierarchy of a computer system. In addition to the above work, the system performance team of the IBM Network Computing Software Division in Austin have used another Java microbenchmark development

Figure 10 Code for file read bandwidth (FileInputStream)

```
FileInputStream f = new FileInputStream(fileName);
byte [] buffer = new byte[fileSize];
int size = 0;
while (size < fileSize)  {
   f.read(buffer,size,blockSize);
   size += blockSize;
}
```

and execution framework called *DecafMark* that was utilized within IBM to evaluate Java performance.

Methodology. Java presents the illusion of a portable operating system, which adds another layer of software to the OS services. To evaluate the performance of OS system services in Java, we build on the line of research culminating in hbench:OS. We have developed jMocha, a set of Java versions of relevant hbench:OS tests, relying on calls to the standard Java APIs where needed. We compare the performance of the native hbench:OS tests, written in C with direct system calls, to the jMocha Java versions.

In order to obtain accurate timing information, we utilize on-chip hardware cycle counters when available. Only timing information is presented in this paper; however, the infrastructure also supports instrumentation of various hardware events such as cache misses, instruction counts, etc. We restrict our attention to fine-grained single-threaded benchmarks and present results on cached file read bandwidth and remote procedure call and remote method invocation (RPC-RMI) performance.

jMocha results. We use jMocha to compare Java performance with native OS services on Windows NT. The Windows NT platform is an IBM IntelliStation* Z-pro machine (6899-120), with an Intel 200-MHz Pentium Pro processor, having 256 MB of RAM, running Windows NT version 4.0, service pack 4. On this platform we compare results using an IBM enhanced port of the Sun JDK 1.1.7 (release 12/07/1998 with IBM just-in-time [JIT] compiler version 3.0), and C tests compiled with Microsoft Visual C++** version 3.1 using the following compiler flags: /O2 /Oa /G6 /GD /GM /MT -DNO_PORTMAPPER -D_WIN32_WINNT=0×0400.

In each trial, we repeated each test for a number of iterations chosen so that the total running time exceeded one second. Each trial included a warm-up stage, not timed, so initial paging and JIT activity do not influence the reported results. The results presented are the 10 percent trimmed mean of ten trials for each data point. Trimmed mean values were collected by running the benchmark ten times, sorting the results in ascending order, removing the maximum and the minimum values, and then taking the mean of the remaining eight. Standard deviations for all results were negligible (less than 5 percent of mean). Of the Java platforms, we could examine the JIT-produced code for the IBM DK, but not for the other JITs.

We present results from a representative subset of the full jMocha suite. The following subsections evaluate Java performance for a cached file system and remote procedure call. The C implementations of these tests, derived from hbench:OS, are described in Reference 16. We describe the corresponding Java implementations in the following subsections.

File read bandwidth. Figure 10 shows a code fragment for a jMocha test that reads from a cached file using a FileInputStream. We chose fileSize = 4 MB, which on Windows NT, ensures that the file fits in the file buffer cache and avoids disk access. For these benchmarks, we read the entire file only once for each trial.

Figure 11 shows the results of this test. Java performance on these benchmarks is competitive with C.

Figure 11 Windows NT results for file read bandwidth (FileInputStream)

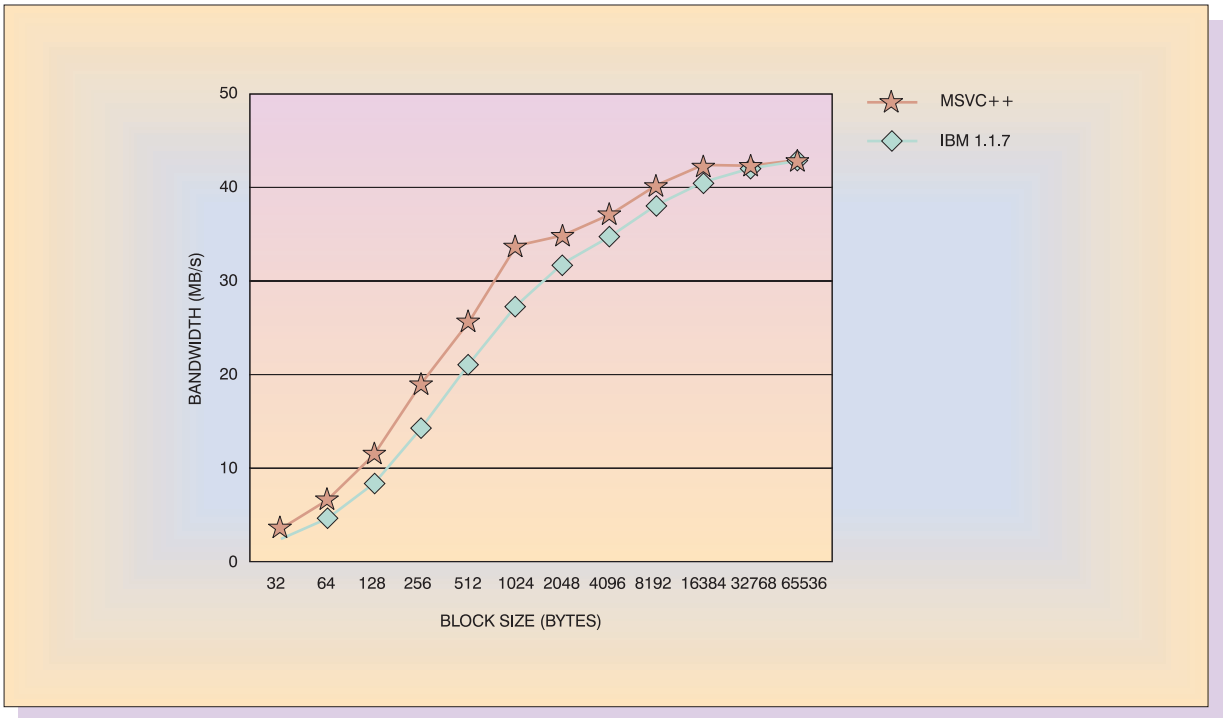


Figure 12 Code for file read bandwidth (FileReader)

```
FileReader f = new FileReader( fileName );
char [] buffer = new char[ fileSize ];
int size = 0;
while (size < fileSize)  {
   f.read(buffer,size,blockSize);
   size += blockSize;
}
```

Java also provides classes to perform formatted file I/O. Figure 12 shows a version of the file read benchmark, using a FileReader class, which reads a file as a sequence of characters. Figure 13 shows the results of this test. The figure shows that Java performance suffers drastically when using the formatted I/O. Every file read (using the Reader classes) must convert the native byte-size characters to the Java two-byte Unicode** representation. The performance of the Java byte-to-Unicode conversion routine dominates this test. For Java formatted I/O to compete with native services, Java developers must address this problem.

The results show that the native implementation significantly outperforms Java.

RMI performance. We compare the performance of a simple Java RMI call to an equivalent RPC. The Java kernel for this microbenchmark appears in Figure 14. The figure shows code for a client that invokes a method, xact, on a server via RMI. The method simply returns the integer value 123.

During the benchmark runs, the RMI call always succeeded, and the client never threw the exception in Figure 14. Figure 15 shows the performance of this program and the corresponding RPC program on Windows NT. The RMI performance lags the TCP RPC performance by a factor of 3.12.

Figure 13 Windows NT results for file read bandwidth (FileReader)

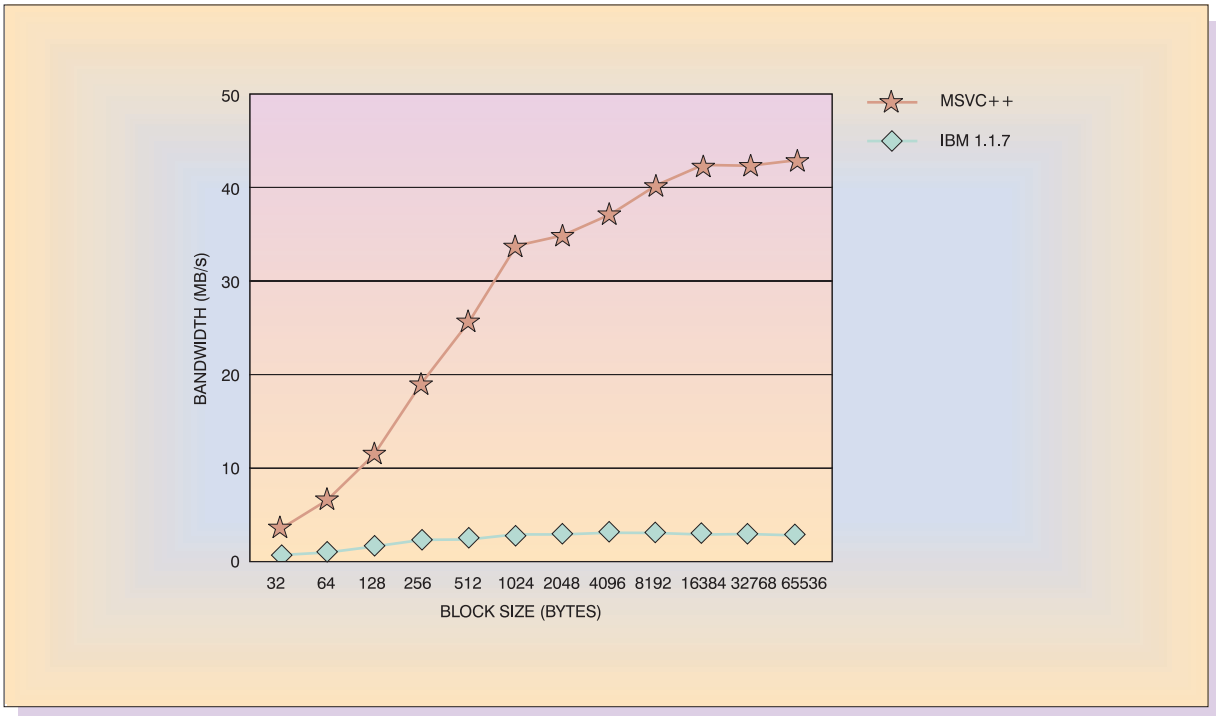


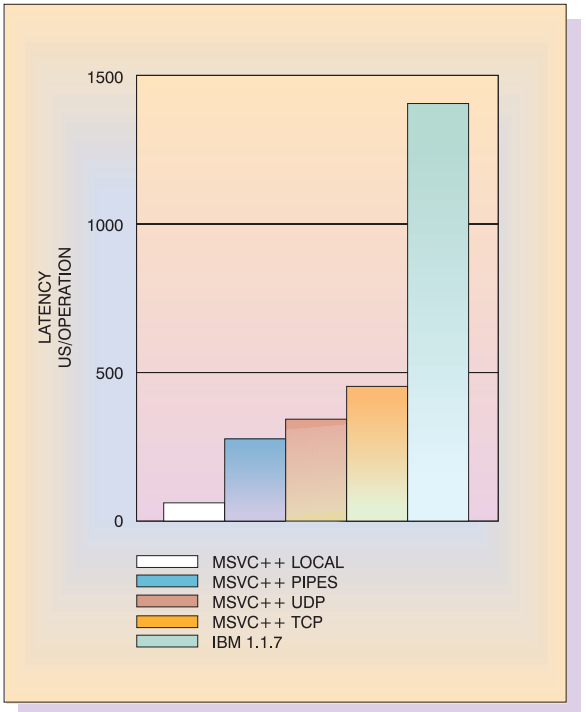
Figure 14 Code for RMI latency

```
for (int i=num_iter;i>0;i–)  {
    result = server.xact();
    if ( result != 123 )
      throw new LatRMIException("Invalid Data: " + result);
}
```

The object-based RMI protocol entails more overhead than the function-based RPC protocol, which helps account for the Java performance penalty. In future work, we will extend the jMocha suite in an attempt to pinpoint bottlenecks in the RMI protocol. Furthermore, we will also compare RMI to CORBA** (Common Object Request Broker Architecture), a more germane object-based communication protocol.

SockPerf microbenchmarks. We used an IBM sockets microbenchmark called SockPerf to measure the cost of the Java sockets API. SockPerf is a peer-to-peer socket benchmark written completely in the Java language. The benchmark is multithreaded, and concurrent sessions can be run over the same network interface or over multiple network interfaces.

Figure 15 Windows NT results for RMI latency



The user may specify characteristics including: (1) the use of a connection-oriented (TCP) or connectionless (UDP, or User Datagram Protocol) protocol, (2) the type of traffic to be measured (request/response, connect/request/response/disconnect or bulk throughput), (3) message size characteristics, and (4) test duration and statistical convergence (confidence interval) characteristics.

For comparison with native sockets, we present results gathered with an internal IBM socket benchmark that runs on OS/2, Windows 95, and Windows NT called XMPT. XMPT is functionally identical to SockPerf and indeed formed the template for SockPerf when it was created.

Methodology. For this study, we used the following four tests:

1. TCP_RR_1: The client sends a one-byte message (request) to the server over a TCP socket, which echoes it back (response). The result of the test is reported as a throughput rate of transactions per second, which is the inverse of the round-trip time for request and response, as well as the CPU utilization of the client and server.
2. UDP_RR_1: The client sends a one-byte message (request) to the server via a UDP datagram, which echoes it back (response). The reported result is transactions per second and client and server CPU utilization.
3. TCP_Stream_8k: The client sends continuous 8-KB messages to the server, which continuously receives them. The reported result is bulk throughput in KB/s and Mb/s and client and server CPU utilization.
4. CRR_64_8k: The client sends a 64-byte message (request) to the server over a TCP socket; the server sends back an 8-KB response. The CRR (connect, request, response) test includes the connection set-up and tear-down costs in the timing loop and is designed to simulate an HTTP 1.0 transaction.

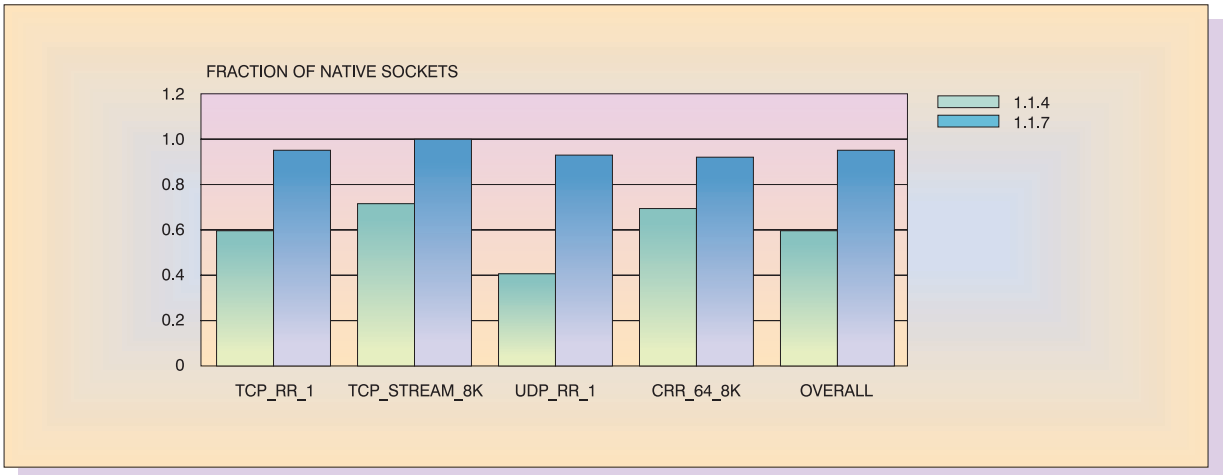
We used internal versions of the SockPerf and XMPT benchmarks that had access to OS/2 kernel instrumentation for CPU utilization. Since CPU utilization could be very accurately collected, we defined a metric called *scaled throughput* as the figure of merit for each test. Scaled throughput was computed by dividing the raw throughput by the average of the client and server CPU utilization. By dividing the Java scaled throughput for a given test by the corresponding native scaled throughput, we finally arrived at a metric of Java performance as a fraction of native performance. In addition to each of the individual tests above, we also computed an overall score for both Java and native performance. This was the geometric mean of the scaled throughputs in each case.

Scaled throughput estimates software efficiency or path length. Thus the ratio of Java to native scaled throughput is a measure of the additional path length imposed by Java on top of the native sockets API.

Results. Figure 16 shows SockPerf and XMPT results gathered between a pair of IBM PC Server 704 systems, each with a 200-MHz Pentium Pro processor, 1 GB RAM running OS/2 Warp Server SMP with TCP 4.1 and the IBM DK 1.1.4 and 1.1.7 releases. The machines were connected to an isolated 100-Mb Ethernet network. Each of the tests listed above is shown as well as an overall geometric mean. The native results are normalized to one, and the Java SockPerf results are shown relative to the native results.

This microbenchmark suite was used to detect and improve the performance of Java sockets in the

Figure 16 SockPerf results



IBM DK for OS/2 following the 1.1.4 release. The performance improvement from an initial low in DK 1.1.4 to DK 1.1.7 is very clear. In DK 1.1.4, Java socket performance was only 60 percent of native. The set of improvements that took performance up to 95 percent in DK 1.1.7 is beyond the scope of this paper. For details, please refer to the paper in Reference 18, which appears elsewhere in this issue. It is worth noting here, based on the OS/2 results, that Java sockets have the potential to be a very thin layer above the native sockets API of the platform. SockPerf is currently being used for similar Java-to-native comparisons and performance enhancements for the Jvm in the IBM Developer Kit for Windows NT, Java Technology Edition, and the Jvm in the IBM Developer Kit for AIX* (Advanced Interactive Executive).

JDBC/ODBC microbenchmarks. The benchmarks jBOB (JDBC) and CEB (ODBC) provide a macro-level view of the efficiency of Java dynamic access to databases as compared to the dynamic access of a C or C++ application (ODBC/CLI). From the standpoint of a user, or the designer of a three-tier solution, this view is crucial. However, from the standpoint of the JDBC architect or developer, a finer-granularity view of performance is also useful. To this end, we created a suite of microbenchmarks focused on the performance of a number of key JDBC methods and their equivalent ODBC/CLI variants.

To determine the important database access methods, we traced the frequency of JDBC calls for each transaction type of jBOB while running on the AS/400. Table 2 provides the results for the new order transaction. Note the high frequency of *PreparedStatement.setInt*, *Resultset.next-ResultSet.getInt*, and *ResultSet.next-ResultSet.getString* calls. The other transaction types, payment, orderline, delivery, and stock, were found to have similar distributions of JDBC method usage. Although the frequency of a call is not always indicative of its role in performance, for the case of jBOB, time-based profiles affirmed the importance of the get and set methods. In addition, anecdotal customer data highlighted *ResultSet* processing as being an especially performance-sensitive area of the JDBC specification. For these reasons, we created a suite of microbenchmarks to focus on the *ResultSet.getXXX* and the *PreparedStatement.setXXX* family of methods.

The microbenchmarks serve several purposes. First, they provide an easy-to-run suite of tests that can magnify and validate JDBC implementation enhancements. Second, they allow performance comparisons between multiple Jvm and JDBC implementations. Third, they allow an easy comparison to similar calls, such as *ParamData*, *Fetch*, and *GetData*, written in the more familiar ODBC/CLI framework. Finally, the benchmarks are useful for performance regression testing as function, and robustness is added to Jvm and JDBC implementations.

Content of JDBC/ODBC benchmarks. The *PreparedStatement.setXXX* method sets the IN parameter in

Table 2 New order JDBC call frequency

Executing	Average Calls per Transaction	Detailed Calls per Transaction
executeQuery( )	23	3 + 2 per orderline
executeUpdate( )	34	4 + 3 per orderline
Prepared Statement Processing	**Average Calls per Transaction**	**Detailed Calls per Transaction**
setBoolean( )	1	
setDouble( )	10	1 per orderline
setInt( )	232	22 + 21 per orderline
setString( )	11	1 + 1 per orderline
ResultSet Processing	**Average Calls per Transaction**	**Detailed Calls per Transaction**
getDouble( )	10	1 per orderline
getFloat( )	3	
getMetaData( )	3	
getInt( )	21	1 + 2 per orderline
getShort( )	20	2 per orderline
getString( )	42	2 + 4 per orderline
next( )	23	3 + 2 per orderline

Table 3 set( )/get( ) methods tested

SET Methods Tested	Underlying Data Type	GET Methods Tested	Data Stored As:
setShort	SMALLINT	getShort	SMALLINT
setInt	INTEGER	getInt	INTEGER
setLong	BIGINT	getLong	BIGINT
setFloat	REAL	getFloat	REAL
SetDouble	DOUBLE	getDouble	DOUBLE
setBoolean	SMALLINT	getBoolean	SMALLINT
setByte	SMALLINT	getByte	SMALLINT
setDate	DATE	getDate	DATE
setString	CHAR(6)	getString	CHAR(6)
setString	VARCHAR(10)	getString	VARCHAR(10)
setTime	TIME	getTime	TIME
setTimeStamp	TIMESTAMP	getTimeStamp	TIMESTAMP
setBigDecimal	NUMERIC	getBigDecimal	NUMERIC
setFloat	NUMERIC	getFloat	NUMERIC

a precompiled statement where *XXX* corresponds to the data type of the variable being set. The microbenchmark suite contains set tests for the datatypes listed in Table 3. The corresponding function is handled in one of two ways for ODBC/CLI applications. Either a user memory location is bound to the PreparedStatement and then assigned, or the *ParamData*/*PutData* calls are used. The former case is the most common and high-performing method. After a location is bound, the user is only required to do a simple variable assignment to perform the equivalent of the JDBC set. The JDBC architecture does not allow this common high-performing alternative because of its aversion to pointers. The set method is most akin to the *ParamData*/*PutData* combination. For the API microbenchmark suite, there are corresponding bind and *ParamData* timed tests for each of the data types used in the *set*( ) suite.

The ResultSet object is a table that contains the results of an executed query. The *ResultSet.getXXX* methods retrieve the specified data from the current

row of the *ResultSet* instance. The *ResultSet.next* method updates the logical cursor to the next row in the ResultSet object. A ResultSet is processed by stepping through the table (next calls) and retrieving data (get calls). The microbenchmark suite contains get tests for the datatypes listed in Table 3. Again, there are two ways to achieve the corresponding function with a dynamic ODBC/CLI application. As before, memory can be bound, and a *Fetch* can be performed, or the *GetData* function can be used. The microbenchmarks contain tests for both types of ODBC access for comparison purposes. It should be noted that in practice the *Fetch* dominates in popularity.

Matching the data type of how data are stored and how they are retrieved is crucial for performance. In most of the microbenchmarks, the type retrieved from the database is the same as the type stored (e.g., *ResultSet.getInt* is done on integer types). The numeric type is an exception. One test retrieves it as a BigDecimal, whereas another retrieves it as a float. This test highlights the expense of retrieving data and casting to an inefficient type. It may be of interest to note that work is currently underway to improve the performance of the BigDecimal class in IBM Java implementations.

Implementation of JDBC/ODBC benchmarks. The microbenchmarks were implemented to ensure repeatability, accuracy, and precision. A small database was created to be queried, and the same database was used for both the ODBC and the JDBC versions of the benchmarks. Each call executed in a loop with many iterations. The time to run all iterations of the call was measured; this run will be referred to as a *trial*. Before the start of the trial, the API or method under test executed a number of times to "warm up" the caches and memory. The intent is to drive the CPU to 100 percent utilization and measure the best case execution time (best memory and cache characteristics) for the method or function. Multiple trials ran until the observed mean time to complete a trial converged to within 5 percent of the theoretical actual mean at a 90 percent confidence level. The time for a single execution was then found by dividing the converged, mean trial time by the number of iterations.

JDBC/ODBC results. Data collected with the JDBC/ODBC microbenchmarks are summarized by Figures 17 and 18. Each bar in the charts corresponds to a Jvm-version/database-version combination. In addition, there are bars representing the corresponding ODBC calls *Fetch*, *GetData*, *ParamData*/*PutData*, and *Bind*. Five Jvms are represented: the Jvms in the IBM DK version 1.1.6 and DK version 1.1.7, and the Jvms in Sun JDK versions 1.1.6, 1.1.7, and Java 2, Version 1.2. Each Jvm (grouping of bars) was measured on five DB2 versions: DB2 5.0 UDB, 5.0 FP9014, 5.2 FP7, and a recent prototype. The enhancements realized in the prototype code are currently targeted for release with DB2 version 6.1. All data were collected on a 333 MHz Pentium II running Windows NT 4.0 with Service Pack 3.

Figure 17 indicates that the core Jvm implementation is not a key factor in the performance of the get access to the *ResultSet*. However, it is interesting to note that the Jvm in the IBM DK v 1.1.7 is the highest performing Jvm, with the JVM in the Sun JDK 1.2 a close second. In all cases the performance difference is not great. IBM obtains its performance advantage through more highly optimized JIT-produced code. The Sun JVM gets a slight edge with a faster Java Native Interface (JNI) implementation. More important to the performance of the get method is the database level and corresponding JDBC implementation. It can be seen that impressive gains were made in the ODBC/CLI level for *GetData* and *Fetch* performance in FixPack 9014. The DB2 JDBC implementation is built on top of the CLI layer; as a result, these performance gains also appear in the JDBC calls.

The large boost in performance with the prototype DB2 code (last bar of each grouping) was realized by avoiding multiple JNI calls between the JDBC and CLI layers of the implementation. In general, transitions from Java to native code and data movement between this barrier are slow. The DB2 performance improvements were achieved by limiting the number of JNI transitions to one per next or get method call, and passing all data as arguments to the native method (instead of using JNI methods to reach back into the Java space). In addition, gains were realized by using the *Get< >ArrayRegion* family of methods as opposed to the *Get< >ArrayElements* whenever possible. The underlying implementation of the *Get< >ArrayElements* call requires objects to be pinned on the heap, whereas the *Get< >ArrayRegion* passes a copy of data by value.

Most interesting from Figure 17 is the difference in performance between the ODBC *Fetch* calls and the get methods. This comparison is useful to a developer moving from the mature ODBC/CLI API to the JDBC API. The performance advantage of the ODBC specification is made clear by these data.

Figure 17 Get DB2 API results

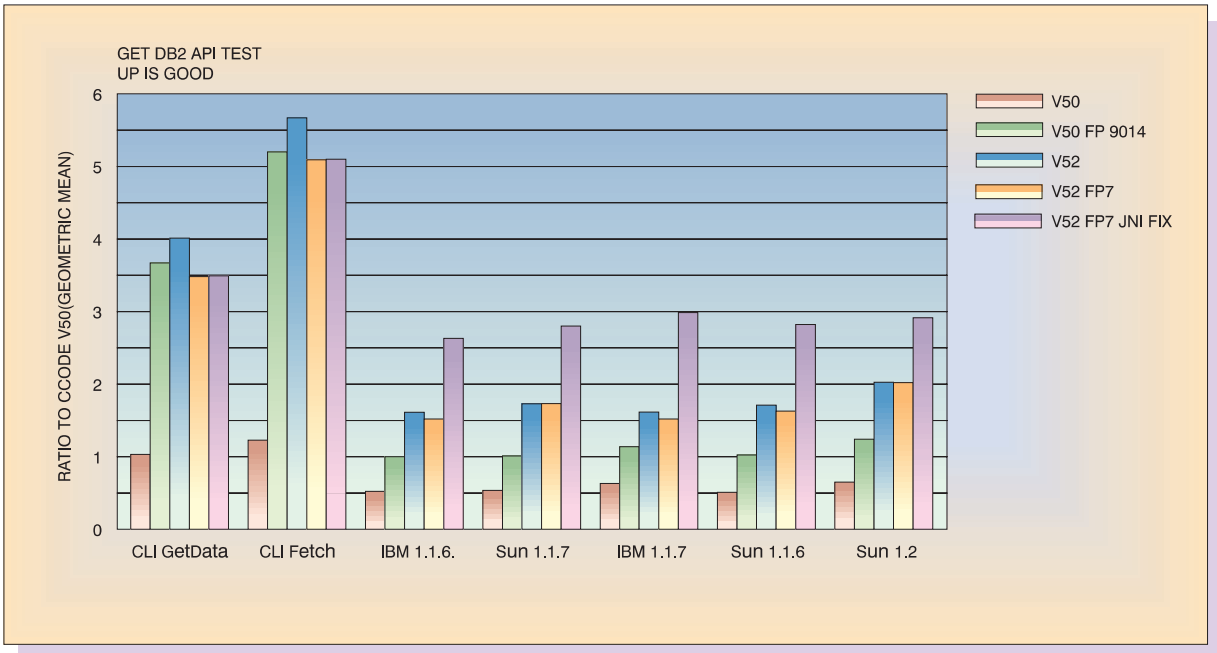
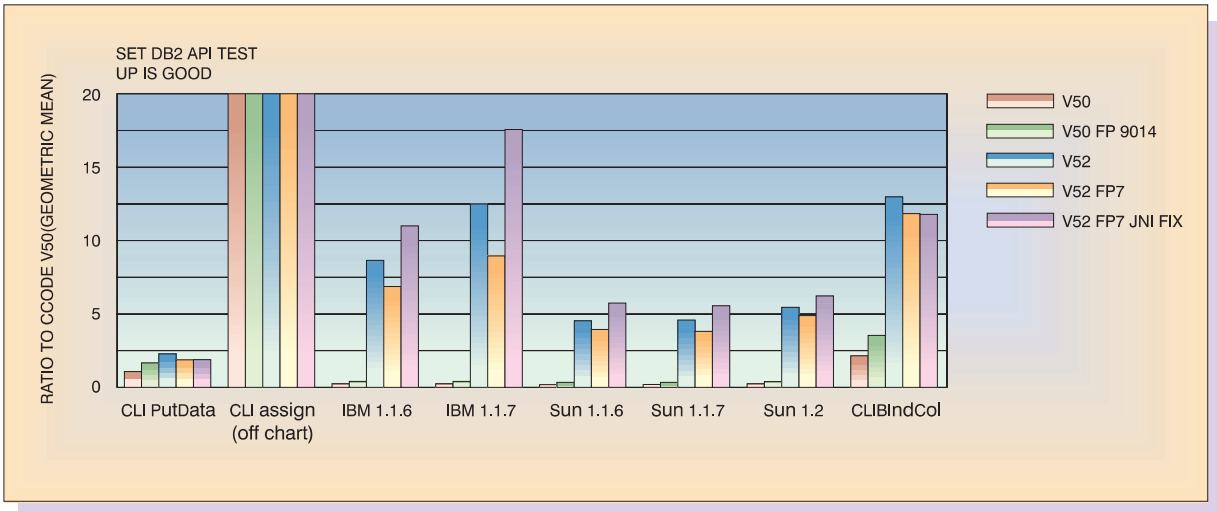


Figure 18 Set DB2 API results



From a performance point of view, it may be advantageous to enhance the JDBC specification to allow *bind* type operations. Currently, there are a number of proposals being considered to address this concern.

Also interesting is the performance advantage the *Fetch* call has over the *GetData* call. For this reason, the *Fetch* is the preferred method of data access for ODBC/CLI applications. However, if an application has been written using the *GetData* call, nearly com-

parable performance can be obtained using the JDBC interface.

The DB2 5.2 implementation of the get methods is in actuality binding memory locations in native space. This allows multiple next or get methods to be done quickly but comes at the cost of an expensive first next call on a ResultSet Object. On the first next call, all the binding is done.

The set comparisons (Figure 18) clearly delineate both Jvm and JDBC implementation performance differences. As in the previous chart, significant performance improvements exist in the CLI layer of DB2 in FP9014. These can be seen in *bind* calls but are not realized in the *ParamData*/*PutData* pair. However, these changes result in only a slight improvement in the corresponding JDBC set calls because the set calls are not built on top of the *bind* interface. Fortunately, improvements in the JDBC implementation have resulted in significant performance improvements in the latest release when compared with the original implementation. As with the get methods, JNI calls (both method invocation and data) were reduced by passing needed data as arguments on the native method call. In addition, the native method return code was overloaded in cases in which more than one return code was required. Finally, the implementation actually binds the memory location to the PreparedStatement object. This location is passed down to the native code, and if it has not changed since the last use, a simple assignment can be made.

As noted in the benchmark description section, the true comparison to the set method in the bound ODBC world is a simple variable assignment. This is illustrated by the bars in the chart that stretch to the top of the graph. It puts JDBC at a decided disadvantage. However, we believe the disadvantage is not serious because the cost of all these calls is small.

The last point to notice in Figure 18 is the superior performance exhibited by the Jvm in IBM DK 1.1.7. It is especially evident on the prototype version of DB2, which has removed a number of JNI calls. The advantage for these tests comes from better optimized code generation from the IBM JIT.

Conclusion

There has been significant activity on Java performance; however, much of the work has focused on Java client performance. IBM has a significant interest in servers and, although there has been some work on Java server performance, there is currently no adequate set of tools to facilitate the analysis of Java server applications. To address this concern, we discussed the design, implementation, and analysis of several Java server benchmarks, including a Web application serving dynamic content, two on-line transaction processing benchmarks, and several families of microbenchmarks.

Overall, the results show that Java server performance is competitive with legacy environments; however, some areas require more attention. Java servlets outperform CGI, and the Java platform is competitive with C or C++ for several areas, including unformatted file I/O, sockets, and certain areas of dynamic database access. IBM's Jvm optimizations such as efficient monitor and object allocation implementations have led to significant performance improvements, including better scalability that is vital for server workloads. In contrast, RMI is not yet competitive with RPC, and formatted I/O requires performance improvements. Furthermore, our experience shows that to understand Java performance for server applications, one must examine more than just *pure Java* performance. Many performance issues arise at the boundaries between programming models, as demonstrated by the jBOB and JWeb2 benchmarks.

Additional benchmarks are needed to provide a more comprehensive assessment of Java server performance. One possible avenue for future work includes extensions to JWeb2. With modifications, the benchmark could stress the Jvm more by increasing the degree of dynamic content or running with a Java Web server such as Java Web Server[19] or Jigsaw.[20] In addition, more work is needed in the area of server-side database access using Java stored procedures.

A second approach would be to use even more realistic Web applications as benchmarks. A set of shopping servlets or banking servlets are good candidates. This approach would provide a more general view of Java server performance, whereas our approach has more successfully abstracted and focused on the typical events of these types of workloads.

Yet another approach would be to alter the dynamic component of the benchmark to separate the logic of the servlet from the creation of the resulting page. This approach, separating content from formatting, can be achieved via JavaServer Pages. We expect

many adopters of Java servlets to also adopt JavaServer Pages because of this cleanness of separation. Hence, we plan to develop an implementation of JWeb2 using JavaServer Pages.

We believe that various network services (such as secure sockets layer, Internet Inter-ORB Protocol [IIOP], remote method invocation over IIOP, and directory services based on the Java Naming and Directory Interface) will be used in developing future Java server applications. Similarly, Enterprise JavaBeans and IBM's Enterprise Java Server will serve as the backbone for a diverse set of applications in the future. In future work, we will consider representative benchmarks incorporating these services.

Acknowledgments

We thank Ben Hoflich, Jimmy Dewitt, and Walter Fraser for helping us to run the benchmarks and graph the results.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Sun Microsystems, Inc., Pendragon Software Corporation, Ziff-Davis, Inc., Volano LLC, Transaction Processing Performance Council, Microsoft Corporation, Unicode, Inc., Intel Corporation, or the Object Management Group.

Cited references

1. CaffeineMark 3.0, Pendragon Software Corporation, Libertyville, IL, http://www.pendragon-software.com/pendragon/cm3/index.html (January 1999).
2. "How to Soup Up Java," *BYTE* magazine online, http://www.byte.com/art/9805/sec5/art4.htm (May 1998).
3. JMark 2.0, ZDNet (Ziff-Davis Labs), New York, http://www.zdnet.com/zdbop/jmark/jmark.html (January 1999).
4. The Standard Performance Evaluation Corporation (SPEC), Manassas, VA, http://www.spec.org (May 1999).
5. VolanoMark 2.0, Volano LLC, San Francisco, http://www.volano.com/benchmarks.html.
6. Benchmark C Standard Specification, Revision 3.3, Transaction Processing Performance Council (TPC), San Jose, CA (April 8, 1997).
7. IBM AS/400 Business Object Benchmark for Java, IBM Corporation, http://www.as400.ibm.com/whpapr/jbob400.htm (May 1999).
8. K. A. Coar and D. R. T. Robinson, *The WWW Common Gateway Interface, Version 1.1*, Internet draft, ftp://ftp.ietf.org/internet-drafts/draft-coar-cgi-v11-02.txt (April 1999).
9. FastCGI specs and documentation, http://www.fastcgi.com/words/ (May 1999).
10. Internet Server API Reference, Microsoft Corp., Redmond, WA, http://www.microsoft.com/win32dev/apiext/isapiref.htm (May 1999).
11. A. Fedorov et al., *Professional Active Server Pages 2.0*, Wrox Press Inc., Chicago (1998).
12. Servlet Specification, Version 2.1, Sun Microsystems, CA, http://java.sun.com/products/servlet/2.1/index.html (May 1999).
13. B. K. Bershad, R. P. Draves, and A. Forin, "Using Microbenchmarks to Evaluate System Performance," *Proceedings of the Third Workshop on Workstation Operating Systems*, IEEE Computer Society Press, Los Alamitos, CA (1992), pp. 148–153.
14. J. K. Ousterhout, "Why Aren't Operating Systems Getting Faster as Fast as Hardware?" *Proceedings USENIX Summer Conference* (June 1990), pp. 247–256.
15. L. McVoy and C. Staelin, "lmbench: Portable Tools for Performance Analysis," *Proceedings of USENIX 1996* (January 1996), pp. 279–294.
16. A. B. Brown and M. I. Seltzer, "Operating System Benchmarking in the Wake of Lmbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture," *Performance Evaluation Review* **25**, No. 1, 214–224 (June 1997).
17. R. H. Saavedra and A. J. Smith, "Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes," *IEEE Transactions on Computers* **44**, No. 10, 1223–1235 (October 1995).
18. R. Dimpsey, R. Arora, and K. Kuiper, "Java Server Performance: A Case Study of Building Efficient, Scalable Jvms," *IBM Systems Journal* **39**, No. 1, 151–174 (2000, this issue).
19. Java Web Server, JavaSoft, San Microsystems, Inc., Palo Alto, CA, http://jserv.javasoft.com/index.html (May 1999).
20. Jigsaw, Jigsaw team, World Wide Wed Consortium, http://www.w3.org/Jigsaw/ (May 1999).

Accepted for publication September 22, 1999.

Sandra Johnson Baylor *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: sandrajb@us.ibm.com).* Dr. Baylor is a manager and a research staff member at the Watson Research Center in Hawthorne, New York. She earned her B.S., M.S., and Ph.D. degrees in electrical engineering from Southern University (*summa cum laude*), Stanford University, and Rice University, respectively. She joined IBM at the Research Center in 1988. She has conducted research on the performance evaluation of cache coherence protocols in shared memory multiprocessors, parallel application workload characterization, parallel I/O design and evaluation, parallel file system design, and thin-client application framework analysis and development. She is currently the manager of the Java Server Performance (JASPER) research group. Her areas of interest include computer architecture, memory systems, parallel processing, performance evaluation, and Java performance.

Murthy Devarakonda *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: mdev@us.ibm.com).* Dr. Devarakonda is a senior manager in IBM Research currently managing groups working in the areas of pervasive and personal computing systems. Earlier, as the manager of the Java server performance project, he developed the strategy for it and led the definition of several Java server benchmarks. Prior to this work, he built a fully recoverable, cluster file system called Calypso, managed the eNetwork Dispatcher research, and led research in Java object frameworks for thin-client applications (TCAF) and script-based mobile agents (NetScript). TCAF and NetScript are available from IBM alphaWorks™. He received a Ph.D. in computer science from the University of Illinois at Urbana-Champaign. He was the program chair

for the USENIX COOTS Conference in 1999, and he is the object-oriented systems area editor for *IEEE Concurrency*.

Stephen J. Fink *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: sjfink@us.ibm.com).* Dr. Fink is a research staff member at the Thomas J. Watson Research Center. He received the B.S. degree from Duke University, Durham, North Carolina, in 1992, and the M.S. and Ph.D. degrees from the University of California, San Diego, in 1994 and 1998, respectively. His research interests include dynamic compilation, run-time systems, object-oriented programming, and parallel scientific computation.

Eugene Gluzberg *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: gluzberg@us.ibm.com).* Mr. Gluzberg received his M.S. and B.S. degrees in computer science from Polytechnic University. He has been with IBM Research since June 1998, working on the jMocha microbenchmark suite along with other Java-related projects in the Java Server Performance group.

Michael Kalantar *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: kalantar@us.ibm.com).* Dr. Kalantar is a visiting scientist at IBM Research. His current interests are performance studies of Java and computer communications. He received his doctorate at Cornell University in 1995 and spent two years teaching computer science in China.

Prakash Muttineni *EC Cubed, Inc., 15 River Road, Suite 310, Wilton, Connecticut 06987.* Dr. Muttineni received both the B.Tech (civil engineering) and M.Tech (computer science) degrees from the Indian Institute of Technology in Madras, India, in 1984 and 1986, respectively. He received a Ph.D. (computer science) degree from the Indian Institute of Sciences in Bangalore, India, in 1996. He was a faculty member at the Regional Engineering College in Warangal, India, for several years. During the last two years, he was at the IBM Thomas J. Watson Research Center with the Java Server Performance team.

Eric Barsness *IBM Software Group, 3605 Highway 52 North, Rochester, Minnesota 55901 (electronic mail: ericbar@us.ibm.com).* Mr. Barsness is a software engineer with the IBM Server Group in Rochester. He joined IBM in 1993 and currently works in the AS/400 systems performance area. His primary responsibility since 1996 has been AS/400 Java performance analysis. Mr. Barsness holds a B.S. in computer science from Iowa State University.

Rajiv Arora *IBM Network Computing Software Division, 11400 Burnet Road, Austin, Texas 78758 (electronic mail: rarora@us.ibm.com).* Dr. Arora is a performance engineer whose current interest is the performance of the Java language on the server. He is working on core performance of the IBM Jvms on the Intel platform. He also represents IBM on the SPEC server-side Java benchmark working group. Prior to his Java work, he worked on Web server performance and the performance of TCP/IP on AIX, the IBM microkernel, and OS/2. Dr. Arora joined IBM in 1992. He holds M.S. and Ph.D. degrees in electrical engineering from the University of Rochester in the area of network protocols.

Robert Dimpsey *IBM Network Computing Software Division, 11400 Burnet Road, Austin, Texas 78758 (electronic mail: dimpsey@us.ibm.com).* Dr. Dimpsey received his Ph.D. degree in 1992 from the University of Illinois where he studied performance measurement, modeling, and analysis of large shared memory multiprocessors. In 1992 he joined IBM to work on symmetric multiprocessing performance of the AIX operating system. In 1994 he began work on the IBM cross-operating system microkernel project. This was followed by work on kernel-level multiprocessor performance for WARPSMP and scalable, journaled file systems. Currently, he is working on server-focused, core Jvm performance for IBM Jvms on Intel-based platforms.

Steven J. Munroe *IBM Software Group, Rochester Laboratory, 3605 Highway 52 North, Rochester, Minnesota 55901 (electronic mail: sjmunroe@us.ibm.com).* Mr. Munroe is a senior software engineer and a member of the IBM SanFrancisco performance group. He has also worked on architecture and design in various system software areas for the IBM System/38 and AS/400 products. He holds nine issued patents. He received a B.S. degree in computer science from Washington State University, Pullman.

Reprint Order No. G321-5716.