

Optimizing Program Performance

This chapter describes techniques to optimize program performance. It contains the following sections:

§6.1: *Overview*

§6.2: *Optimization Levels*

§6.3: *Execution Speed Versus Code Size*

§6.4: *Debugging Optimized Code*

§6.5: *Optimization Reference*

6.1 Overview

This chapter describes how to optimize a program, that is, compile it to run faster or require less memory. A speed improvement might come at the expense of code space, or vice versa. Some optimizations save both time and space. Many increase compile time.

You can activate some optimizations by specifying the appropriate *optimization level* with command-line option `-On` (where *n* represents the optimization level). You activate other optimizations with separate compiler options, pragmas, or toggles, instead of (or in addition to) specifying an optimization level.

This chapter describes the various optimization levels and how you specify them. It also explains how the optimizations affect program size and speed, as well as program debugging. §6.5: *Optimization Reference* provides detailed information about each of the available optimizations.

6.2 Optimization Levels

High C/C++ provides the following optimization levels:

- Level 0 produces faster compilation.
- Levels 1 — 6 generate faster code.
- Level *s* generates smaller code.

In general, these optimization levels are progressive, in that they successively add further optimization phases. For example:

- Level 1 includes the optimizations performed at level 0.
- Level 2 includes cross jumping, only, from level 1.
- Level 3 includes the optimizations performed at level 2.
- From level 3 on, each higher-numbered level includes the optimizations performed at lower levels and specifies additional optimizations.

This last statement has a few exceptions. For example, preloading arguments from memory is performed only at level 1 and level *s*.

Some of the High C/C++ optimizations are defined as “default” or “level-independent.” Default optimizations are always on and cannot be disabled. Level-independent optimizations are enabled and disabled by compiler options, toggles, or pragmas. They are not turned on automatically by specifying a numeric optimization level.

6.2.1 Specifying an Optimization Level

To specify an optimization level, use command-line option `-On`, where *n* is the number of the level (0 through 6) or the letter *s*.

- If you do not specify the `-O` option, level 2 is the default.
- If you specify `-O` without an argument, level 6 is the default, except that toggle `Behaved` is turned `Off`.

In general, the higher the optimization level, the faster the generated code runs, usually at the expense of compile time (or, in some cases, code size).

<i>Level 0:</i>	Optimization level 0, which produces the fastest compile times, includes
<i>Default</i>	default optimizations that you cannot disable. Level 0 also produces code in
<i>optimizations,</i>	source order to make code easier to debug at the assembly-language level. If
<i>fast compile time</i>	compilation speed or ease of debugging are more important to you than the
	speed of generated code, specify option <code>-O0</code> to compile at optimization
	level 0.
<i>Level s:</i>	Level <i>s</i> includes time- and space-saving optimizations that also appear at
<i>Small code size</i>	other levels. To compile at optimization level <i>s</i> , specify option <code>-Os</code> .
<i>Level-independent</i>	To compile with any of the level-independent optimizations, you must
<i>optimizations</i>	specify the appropriate control (option, toggle, or pragma). See the
	explanation of each optimization in this chapter for the relevant control(s).

6.3 Execution Speed Versus Code Size

Most High C/C++ optimizations are intended to improve execution speed; some do so at the expense of code size. Some save code space, at the possible expense of execution time; a few save both time and space. You must decide which is more important to your application: faster execution or smaller code.

In a few cases (for example, cross jumping and inlining) benefits and disadvantages might seem contradictory. Table 6.1 summarizes these considerations.

Cross jumping Cross jumping can speed up execution time by decreasing the size of a function and therefore increasing its chances of being held in an instruction cache. On the other hand, cross jumping can slow down execution time by introducing extra branch instructions.

For information on cross jumping, see §6.5.14: *Cross Jumping (Tail Merging)*.

Inlining Inlining might make function definitions longer, resulting in increased compile time. Also, space might be increased, unless the inlined function contains no more code than would be generated for the calling sequence that is replaced. Too much inlining can increase the size of a function beyond the ability to fit in a cache.

For information on inlining, see §6.5.18: *Inlining Functions*.

Constants in code Placing constants in code can reduce page writes on systems that implement virtual memory management.

Smaller code If smaller code is more important to you than execution speed, specify `-Os`.

6.3.1 How Optimizations Can Affect Speed and Size

Table 6.1 lists optimizations turned on at each of the optimization levels, shows how optimizations can affect execution time and code space, and notes possible disadvantages of each optimization.

- A *primary benefit* is one that *almost* always happens (optimizations sometimes backfire).
- A *secondary benefit* is one that less frequently makes a difference.

Note: The terms *larger* and *smaller* in Table 6.1 refer to the size of the final executable image. The terms *faster* and *slower* refer to the run-time execution speed of the program, unless otherwise noted.

Table 6.1 Optimization Levels and Benefits

Level, Optimization, and Page	Benefits		Possible Disadvantages
	Primary	Secondary	
Level 0 Optimizations:			
Default optimizations that cannot be disabled:			
Common Subexpression Elimination (Local) (page 127)	Faster, smaller		Larger
Constant Expression Folding (page 129)	Faster		
Constant Propagation (page 129)	Faster, smaller		
Dead (Unreachable) Code Elimination (page 130)	Smaller		
Live/Dead Analysis (page 135)	Faster, smaller		
Operator Strength Reduction (page 138)	Faster		
Register Allocation (Global) (page 139)	Faster		
Default optimization that can be disabled:			
Code in Source-Code Order (page 127)	Faster		

Table 6.1 Optimization Levels and Benefits (Continued)

Level, Optimization, and Page	Benefits		Possible Disadvantages
	Primary	Secondary	
Level 1 Optimizations include all Level 0 optimizations, and also the following:			
Cross Jumping (Tail Merging) (page 130)	Smaller	Faster	Slower
Preloading Arguments from Memory (page 138)	Smaller	Faster	
Level 2 Optimizations include Cross Jumping (Tail Merging) from Level 1, and also the following:			
Common Subexpression Elimination (Global) (page 127)	Faster, smaller		Slower compile
Common Subexpression Elimination (Local, Iterative) (page 128)	Faster, smaller		
Instruction Scheduling (High-Level) (page 134)	Faster		
Live/Dead Analysis (Iterative) (page 135)	Faster, smaller		
Loop Memory-Reference Elimination (page 136)	Faster, smaller		
Register Allocation (Enhanced) (page 138)	Faster, smaller		
Register Lifetime Analysis (page 139)	Faster, smaller		
Spill-Code Reduction (page 140)	Faster, smaller		
Level 3 Optimizations include all Level 2 optimizations, and also the following:			
Instruction Scheduling (Low-Level) (page 135)	Faster		Slower compile
Loop Strength Reduction (page 136)	Faster		
Spill-Code Clean-Up (page 140)	Faster, smaller		
Level 4 Optimizations include all Level 3 optimizations, and also the following:			
Back-Substitution of Nodes (Jump Reduction Inside Nodes) (page 126)	Faster		Slower compile
Code Assumed “Well Behaved” (page 126)	Faster		Larger, slower execution and compile
Inlining Functions (page 131)	Faster		
Loop Induction-Variable Elimination (page 136)	Faster		Larger
Level 5 Optimizations include all Level 4 optimizations, and also the following:			
Single Static Assignment (page 139)	Faster		Slower compile
Level 6 Optimizations include all Level 5 optimizations, and also the following:			
Back-Substitution of Epilog Code (page 126)	Faster		Larger
Loop Unrolling (page 137)	Faster		Larger
Spill Analysis (Improved) (page 140)	Faster		Slower compile

Table 6.1 Optimization Levels and Benefits (Continued)

Level, Optimization, and Page	Benefits		Possible Disadvantages
	Primary	Secondary	
Level s Optimizations: (Space-Saving)			
One iteration of each of these optimizations:			
Code Space Minimizing (Optimizing for Space) (page 127)	Smaller		Slower
Cross Jumping (Tail Merging) (page 130)	Smaller	Faster	Slower
Loop Memory-Reference Elimination (page 136)	Faster, smaller		
Preloading Arguments from Memory (page 138)	Smaller	Faster	
Register Allocation (Enhanced) (page 138)	Faster, smaller		
Small-Data Sections Allocation (page 139)	Faster	Smaller	(See reference section)
Multiple iterations of these optimizations:			
Common Subexpression Elimination (Local, Iterative) (page 128)	Faster, smaller		
Live/Dead Analysis (Iterative) (page 135)	Faster, smaller		
Level-Independent Optimizations:			
Aligning Code and Data (page 125)	Faster		Larger
Compiler Intrinsics (page 128)	Faster		Larger
Constants in Code (page 129)	Faster		
Expression Simplification (page 130)	Smaller		
Inlining ANSI Standard C Functions (page 131)	Faster		Larger
In-Order Execution of I/O (Disable) (page 133)	Faster	Smaller	
No Aliasing (page 137)	Faster		Corruption
Tail Recursion (page 141)	Faster	Smaller	

6.4 Debugging Optimized Code

When you specify option `-O0` (zero) or `-O1` (one), the compiler tries to generate code in source order. This makes your code easier to debug at the assembly-language level.

Limit optimizations when you plan to debug a program In general, optimizations make code harder to debug. Debuggers cannot usually determine when a variable’s current value is in a register rather than in memory. The following are some of the features that impact debugging:

- Inlining often prevents debuggers from stepping through a program correctly.
- During debugging, cross jumping can result in surprising jumps that are not reflected in the source code.

To disable all optimizations that could “fool” a debugger, use command-line option `-g` and do not set an optimization level higher than level 1.

6.5 Optimization Reference

This section provides an alphabetical reference of all compiler optimizations (see Table 6.1 for a high-level summary). Each entry for an optimization provides the following information:

- level(s) at which the optimization occurs
- compiler control(s), if any, that enable or disable the optimization
- details of implementation or usage

6.5.1 Aligning Code and Data

Optimization levels Level-independent

Pragmas `Align_members`
`Push_align_members`
`Pop_align_members`

References to structure members are faster when the members are aligned on appropriate byte boundaries. Whether there is much of a space trade-off depends on how many structures your application contains and how they are organized.

6.5.2 Back-Substitution of Epilog Code

Optimization levels 6

Toggle `Back_substitute_epilog`

Whenever there is an unconditional jump to the epilog of a function, the compiler replaces the jump with the epilog sequence.

This optimization increases execution speed at the expense of code space.

6.5.3 Back-Substitution of Nodes (Jump Reduction Inside Nodes)

Optimization levels 4 — 6

Toggle `Back_substitute_nodes`

This optimization reduces the number of jumps, particularly those in conditional blocks, within a loop.

This optimization increases execution speed, but can lead to a longer compile time.

6.5.4 Code Assumed “Well Behaved”

Optimization levels 4 — 6

Toggle `Behaved` (works at any level; default is On at level 4)

The compiler can be less conservative in generating code for pointer-based objects in a program that follows these rules:

- The address of a union member is never assigned to a pointer.
- A value of a pointer type is never cast to an incompatible pointer type.

Given these assumptions, the compiler might be able to generate substantially faster code in referencing pointer-based variables.

6.5.5 Code in Source-Code Order

Optimization levels 0 — 1

Toggle `Source_code_order`

The compiler attempts to generate code in source order. This makes code easier to debug at the assembly-language level.

6.5.6 Code Space Minimizing (Optimizing for Space)

Optimization level `s`

Toggle `Optimize_for_space`

When this toggle is On, the code generator produces code sequences that minimize code space, sometimes at the expense of execution speed.

6.5.7 Common Subexpression Elimination (Global)

Optimization levels 2 — 6

Toggle `Global_CSE`

The compiler eliminates common subexpressions across all basic blocks of a function. Also, any register computation that is invariant within a loop is moved above the loop.

Note: If a function is too large, the compiler issues a warning and does *not* perform common subexpression elimination, even though you have explicitly turned On toggle `Global_CSE`.

6.5.8 Common Subexpression Elimination (Local)

Optimization levels Default optimization, one iteration always performed

The compiler eliminates common subexpressions within extended blocks of code. An *extended block* is a section of self-contained code with one entry and one or more exits.

6.5.9 Common Subexpression Elimination (Local, Iterative)

Optimization levels 2 — 6, *s*

Toggle Local_CSE_iterate

The compiler always performs local common subexpression elimination (CSE) once. With this optimization, local CSE is performed iteratively to further improve the code.

See §6.5.8: *Common Subexpression Elimination (Local)* for a description.

6.5.10 Compiler Ininsics

Optimization levels Level-independent, compiler intrinsics are always used unless explicitly disabled (see the following subsection for details).

High C/C++ supports the compiler intrinsics shown in Table 6.2.

Table 6.2 *Compiler Intrinsics*

<code>_abs()</code>	<code>_memcpy()</code>
<code>_alloca()</code>	<code>_memset()</code>
<code>_fabs()</code>	<code>_strcmp()</code>
<code>_labs()</code>	<code>_strcpy()</code>
<code>_memcmp()</code>	<code>_strlen()</code>

Disabling Compiler Intrinsics

If you want to call an equivalent library function instead of an intrinsic, you must provide an external declaration. For example, this declaration:

```
extern int _abs(int);
```

disables the `_abs` intrinsic.

6.5.11 Constant Expression Folding

Optimization levels Default optimization, always performed

Arithmetic expressions whose operands are constants are evaluated at compile time. Constant propagation often exposes such expressions that are not otherwise apparent.

6.5.12 Constant Propagation

Optimization levels Default optimization, always performed

When the compiler sees a reference to a variable whose last assigned value was a constant, the reference is replaced by the constant.

6.5.13 Constants in Code

Optimization levels Level-independent

Toggles `Read_only_strings`
`Const_in_code`
`Literals_in_code`

Static variables (exported or local) declared with the **const** qualifier can be mapped to the code area. Lengthy literals can also be placed in code space rather than data space. This can be beneficial where the operating system performs dynamic code loading.

In such circumstances code is most often read-only, so literals can be swapped out of memory without the need to write them to the paging medium used by the operating system. Also, when data space is at a premium, this is a useful way to shift potentially large amounts of program from data to code.

When toggle `Read_only_strings` is On, all string constants are assumed to be read-only. This permits the compiler to place them in read-only sections, thus reducing system overhead when more than one process is executing the same code.

Note: String constants qualified with type **const** are unconditionally placed in the read-only data section.

6.5.14 Cross Jumping (Tail Merging)

Optimization levels 1 — 6, *s*

Toggle `Cross_jump`

Two or more basic blocks of code that end in the same sequence of code, and that have a common successor, can be rewritten to eliminate the redundant code. This is an effective space-saving optimization.

The effect of cross jumping on execution speed depends on machine characteristics and the nature of your application: it could slow down code, or speed it up, or it might have no effect on speed. You can disable cross jumping by turning `Off` toggle `Cross_jump`.

6.5.15 Dead (Unreachable) Code Elimination

Optimization levels Default optimization, always performed

The compiler eliminates code that cannot be reached by any execution path. Such “dead” code might be created as a result of another optimization. For example, constant propagation might have converted a conditional jump into an unconditional jump, eliminating a path out of the associated block.

6.5.16 Expression Simplification

Optimization levels Level-independent

Toggle `Use_reciprocal_for_divide`

A floating-point divide operation whose divisor is a constant can be converted to a multiplication by the reciprocal of the constant. Floating-point multiplication is significantly faster than division in most cases.

However, if the reciprocal of the constant is not exactly representable, multiplication might introduce an inaccuracy in the last bit of the resulting mantissa. In most programs this does not matter; you must decide whether this level of accuracy is critical in your application.

Toggle `Use_reciprocal_for_divide` must be turned On to force multiplication by reciprocal; it works at all optimization levels.

6.5.17 Inlining ANSI Standard C Functions

Optimization levels Level-independent

Toggle `Recognize_library`

When toggle `Recognize_library` is On, calls to several library functions are replaced by fast **inline** code. The High C/C++ compiler automatically recognizes ANSI Standard C functions such as `abs()` and `memcpy()` and inlines them. To disable inlining of these functions, turn Off toggle `Recognize_library`.

6.5.18 Inlining Functions

Optimization levels 4, 5 — 6

Option `-Hi[argument]`

When a function is *inlined*, calls to it are replaced with the logic contained in the function. This substitution speeds up execution, but code size is potentially larger.

The substitution saves prolog, epilog, and parameter-passing overhead, and exposes opportunities for additional optimizations such as copy propagation and constant folding.

Level 4 Function inlining at optimization level 4 inlines functions with less than 40 tree nodes that are called less than 4 times.

Levels 5 — 6 Function inlining at optimization levels 5 and 6 inlines functions with less than 40 tree nodes that are called less than 10 times.

You can achieve the same effect with the inliner by specifying the following option: **-HiC=40,10**.

Manual Versus Automatic Inlining

You can manually select individual functions to be inlined, or allow automatic selection of functions according to criteria you provide.

Manually selecting a function for inlining For manual selection, use type qualifier **_Inline** to cause a particular function to be inlined (you must also specify **-Hi** on the command line). For example:

```
_Inline void func() {
    . . .
}
```

Automatically selecting functions for inlining For automatic selection, use these options to specify selection criteria:

-Hia	Do not inline functions that have their address taken
-Hib=<i>n</i>	Specify temporary-file buffer size
-Hic=<i>n</i>	Inline functions called fewer than <i>n</i> times
-HiC=<i>s, n</i>	Inline functions smaller than <i>s</i> and called fewer than <i>n</i> times
-Hih	Show which functions have been inlined and/or defined
-Hir	Do not inline recursive functions
-His=<i>n</i>	Inline functions with stack size less than <i>n</i> bytes
-Hit=<i>n</i>	Inline functions with fewer than <i>n</i> tree nodes
-Hiw	Invoke the inliner in multi-module (wide inlining) mode
-Hix	Do not inline exported functions

These options and others available for inlining are described in greater detail in Appendix C: *Using the Optional Inliner*.

Wide (Multi-Module) Inlining

You can specify option **-Hiw** to make the inliner operate on a group of files being compiled as a collection, rather than on each file individually. This is known as *multi-module inlining* or *wide inlining*.

When you use separate source files for data-hiding purposes or because more than one programmer is working on a project, inlining a function from one module into another can markedly improve performance. Also, benchmark results can be much improved where a benchmark is split across multiple files.

To achieve multi-module inlining you must also specify **-Hwide**, for “wide” compilation. See Chapter 3: *Using Compiler Options* for more information about option **-Hwide**. See Appendix C: *Using the Optional Inliner* for more information about multi-module inlining.

The optional inliner constructs unique public names for variables and functions that can easily be traced back to the defining module. See §C.4: *How the Inliner Constructs Unique Names* for more information.

Effect of Inlining on Compile Time and Code Size

Inlining can make function definitions longer, resulting in increased compile time. Too much inlining can increase the size of a function beyond the ability to fit in a cache.

Also, code size might be increased unless the inlined function contains no more code than would be generated for the calling sequence that is replaced.

6.5.19 In-Order Execution of I/O (Disable)

Optimization levels Level-independent

Toggle Use_eieio

By default, toggle Use_eieio is Off.

The PowerPC processor normally does not perform write transactions immediately after execution of a store instruction. Instead, the processor queues requests for the writes. As a result, writes are not guaranteed to be performed in the order in which the requests are queued.

With **volatile** non-stack variables, when you turn On toggle `Use_eieio`, the compiler generates an **eieio** (enforce in-order execution of I/O) instruction after each store instruction, in order to maintain data integrity during writes. (With such variables, data transfers might involve device drivers and controllers, and the order in which data is written might be significant.)

The **eieio** instruction ensures that all writes posted prior to the execution of the **eieio** instruction are performed before any writes posted after the execution of the **eieio** instruction. This incurs the overhead of one **eieio** instruction for every store instruction.

There are times when you can predict that the out-of-order dispatch of write requests does not affect the program's consistency. For example, the writes might involve updating only primary memory and not involve any device controllers. Under these circumstances, the processor can keep track of all of the writes it has performed, as well as those that are still queued.

6.5.20 Instruction Scheduling (High-Level)

Optimization levels 2 — 6

Toggle `High_level_scheduling`

When this optimization is on, the compiler re-orders machine instructions to avoid any *processor idle time*: when an instruction waits for one of its operands to be computed by a prior instruction. Unrelated instructions are inserted in between two such instructions so that there will be (hopefully) no such idle time.

Floating-point instructions usually take the most execution time. Therefore, floating-point instructions are started at the earliest possible time, and use of their results is delayed until the results become available.

Instruction scheduling also improves the performance of a target machine with multiple functional units. Instructions that can execute in the same cycle are scheduled together.

Debugging fully scheduled code at the assembly level can be difficult, and scheduling also impairs symbolic debugging (option `-g`). We do not recommend instruction scheduling in a debugging context.

High-level scheduling reorders instructions prior to register allocation, to reduce instruction latency. This reordering can increase contention in the register allocator, forcing more temporary variables to be placed in memory instead of in machine registers. Thus there must be a trade-off between the high-level scheduler and the register allocator.

6.5.21 Instruction Scheduling (Low-Level)

Optimization levels 3 — 6

Toggle `Low_level_scheduling`

Low-level instruction scheduling attempts to reorder instructions after register allocation. Because register allocation is already performed, the scheduling is somewhat constrained. Nevertheless, opportunities arise in which the low-level scheduler is able to reduce overall instruction cycles by moving instructions up or down.

6.5.22 Live/Dead Analysis

Optimization levels Default optimization, one iteration always performed

Computations whose results are never used are eliminated. Assignments of values that are never used (“dead stores”) are also eliminated.

6.5.23 Live/Dead Analysis (Iterative)

Optimization levels 2 — 6, *s*

Toggle `Live_dead_iterate`

The compiler always performs live/dead analysis once. With this optimization, the compiler performs live/dead analysis iteratively to remove dead code transitively.

See §6.5.22: *Live/Dead Analysis* for a description.

6.5.24 Loop Induction-Variable Elimination

Optimization levels 4 — 6

Toggles Induction_analysis
Strength_reduce

A loop counter that varies by a constant and whose value is tested to determine when to exit the loop is called an *induction variable*.

If the induction variable is used *only* to test for loop termination, and not in any other context, it is readjusted to terminate the loop when its value is 0 (zero). Your program's speed is increased because the compare instruction is eliminated.

Toggle Induction_analysis automatically turns On toggle Strength_reduce.

6.5.25 Loop Memory-Reference Elimination

Optimization levels 2 — 6, *s*

Toggle Mem_refs_from_loop

The compiler attempts to replace memory references to variables in loops with references to registers.

6.5.26 Loop Strength Reduction

Optimization levels 3 — 6

Toggle Strength_reduce

In the body of a loop, multiplications and array-indexing operations involving the loop counter can be replaced by additions. This optimization usually speeds up operations that are performed on every iteration of the loop.

However, this optimization could actually slow down execution if rarely-executed multiplications (as in a **switch** statement) are replaced by additions that are always executed.

6.5.27 Loop Unrolling

Optimization levels 6

Pragma `Loop_factor`

Option **-Hunroll** (works at any level; default is On at level 6)

Loop unrolling (or loop unwinding) involves replacing a loop with the code that makes up the loop body, replicated for some number of loop iterations.

This optimization usually increases code size. Loop unrolling is discussed in detail in Appendix G: *Loop Unrolling*.

6.5.28 No Aliasing

Optimization level Level-independent

Toggle `Noalias`

The compiler assumes that no aliasing is used in the code (for example, no more than one pointer points to the same memory location).

Caution: Turning On toggle `Noalias` for a program that has aliased pointers might cause the generated code to run incorrectly.

Caution: Pragma `Noalias` is used for certain classes of benchmarks. Exercise caution when using pragma `Noalias` with applications.

When you use `Noalias`, the optimizer assumes that no two pointers reference the same item at the same time, and that no variable address is ever assigned to a pointer.

Because many C and C++ programs violate these assumptions, they may not run correctly when compiled with `Noalias`.

For more information, see the discussion of type qualifiers `_Alias` and `_Noalias` in §7.2.2: *Access-Related Type Qualifiers* and the entry for toggle `Noalias` in Chapter 4: *Using Compiler Toggles*.

6.5.29 Operator Strength Reduction

Optimization levels Default optimization, always performed

Sometimes an operation can be replaced by a more efficient operation (or series of operations) that yield the same result in less time.

For example, multiplication by a constant can be converted to a series of additions, subtractions, and shifts. Division or modulo by an integer power of 2 can be converted to a shift or masking operation.

Note: Some strength reductions, such as those that replace multiplication with additions and shifts, can result in larger code size.

6.5.30 Preloading Arguments from Memory

Optimization levels 1, s

Toggle `Preload_args_from_memory`

When possible, the compiler copies function arguments to machine registers in the function prolog to reduce memory fetches.

Do not turn On toggle `Preload_args_from_memory` when global common subexpression optimization is being performed. Although the compiler still generates correct code, it might tie up registers unnecessarily. This toggle is Off by default at higher optimization levels.

6.5.31 Register Allocation (Enhanced)

Optimization levels 2 — 6, s

Toggles `Reg_alloc_enhance`
`Reduce_reg_contention`

The compiler has a number of heuristics available to determine how best to allocate registers. By default, the compiler always uses one heuristic to allocate registers. With this optimization, potentially all the heuristics are analyzed and the best one is used.

6.5.32 Register Allocation (Global)

Optimization levels Default optimization, always performed

Operations using registers are often much faster than equivalent operations that reference memory locations. The compiler allocates registers to variables according to how frequently the variables are used.

With global register allocation, the compiler can override explicit **register** declarations to achieve the most efficient usage.

6.5.33 Register Lifetime Analysis

Optimization levels 2 — 6

Toggle `Loop_reg_rename`

Some disjointed loops are referenced in more than one loop but are “dead” upon exiting a loop. The compiler tries to allocate such variables to different registers in different loops.

6.5.34 Single Static Assignment

Optimization levels 5 — 6

Toggle `Single_static_assignment`

When the intermediate representation of a program is put in single static assignment form, the optimizer can perform additional global copy, constant propagation, and other optimizations that are not otherwise available.

6.5.35 Small-Data Sections Allocation

Optimization level `s`

Option `-Os`

Using option `-Os` sets the following pragma at compile time:

```
Push_small_data(sdata, 8);
```

For finer control of data assignments to small-data sections, see the entry for toggle `Push_small_data` in §4.2: *Toggle Reference* and the discussion of small-data sections in §11.5.4: *Small-Data Sections*.

Possible disadvantage Modules containing the definition and use of small-data variables must be compiled with the same small-data specifications.

6.5.36 Spill Analysis (Improved)

Optimization level 6

Toggle `Improved_spill_analysis`

This optimization results in more efficient spill code and a faster executable, at the expense of a longer compile time.

6.5.37 Spill-Code Clean-Up

Optimization levels 3 — 6

Toggle `Cleanup_spills`

If the register allocator produces spill code, the compiler performs local common subexpression elimination again to eliminate any unnecessary spills.

See §6.5.8: *Common Subexpression Elimination (Local)* for more information.

6.5.38 Spill-Code Reduction

Optimization levels 2 — 6

Toggle `Reduce_reg_contention`

The register allocator attempts to limit the lifetime of registers to reduce spill code.

6.5.39 Tail Recursion

Optimization level Level-independent

Toggle Tail_recursion

If a function calls itself recursively, and the call is the last action the function performs, the code can be modified to branch to the beginning of the function instead of calling it again, with appropriate adjustments for parameter passing and return.

When you turn On toggle Tail_recursion, the compiler optimizes any recursively called function that can be modified this way.

