# The Definitive Guide to GCC

## Second Edition

William von Hagen

Apress®

**The Definitive Guide to GCC, Second Edition**



ISBN-13 (pbk): 978-1-59059-585-5

ISBN-10 (pbk): 1-59059-585-8

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1



Lead Editors: Jason Gilmore, Keir Thomas
Technical Reviewer: Gene Sally
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Richard Dal Porto
Copy Edit Manager: Nicole LeClerc
Copy Editor: Jennifer Whipple
Assistant Production Director: Kari Brooks-Copony
Production Editor: Katie Stence
Compositor: Susan Glinert
Proofreader: Elizabeth Berry
Indexer: Toma Mulligan
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.



The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 5

# Optimizing Code with GCC

These days, compilers are pretty smart. They can perform all sorts of code transformations—from simple inlining to sophisticated register analysis—that make compiled code run faster. In most situations, faster is better than smaller, because disk space and memory are quite cheap for desktop users. However, for embedded systems small is often at least as important as fast because of a commonplace environment consisting of extreme memory constraints and no disk space, making code optimization a very important task.

By this point in the book, you have a pretty good grasp of how to compile your code and how to make gcc, the GCC C compiler, do as you please. The next step, accordingly, is to make your code faster or smaller, which is the topic of this chapter. Based on what you learn, you might even be able to make your next program faster *and* smaller. After a quick, high-level overview of compiler optimization theory, we'll discuss GCC's command-line options for code optimization, starting with general, architecture-independent optimizations and concluding with architecture-specific optimizations.

While the examples in this chapter are given in the C programming language, the optimization options discussed in this chapter are independent of the programming language in which your code is written. Being able to share optimization flags across compilers for different languages is a significant advantage of using a suite of compilers, such as those provided by GCC (GNU Compiler Collection).

### OPTIMIZATION AND DEBUGGING

In the absence of optimization, GCC's goal, besides compiling code that works, is to keep compilation time to a minimum and generate code that runs predictably in a debugging environment. For example, in optimized code, a variable whose value is repeatedly computed inside a loop might be moved above the loop if the optimizer determines that its value does not change during the loop sequence. Although this is acceptable (if, of course, it does not alter the program's results), this optimization makes it impossible to debug the loop as expressed in your source code because you cannot set a breakpoint on that variable to halt execution of the loop. Without optimization, on the other hand, you can set a breakpoint on the statement computing the value of this variable, examine variables, and then continue execution. This is what is meant by "code that runs predictably in a debugging environment."

As is discussed in this chapter, optimization can change the flow, but not the results, of your code. For this reason, optimization is a phase of development generally best left until after you have completely written and debugged your application. Your mileage may vary, but it can be tricky to debug code that has been optimized by any of the GCC compilers.

# A Whirlwind Tour of Compiler Optimization Theory

*Optimization* refers to analyzing a straightforward compilation's resulting code to determine how to transform it to run faster, consume fewer resources, or both. Compilers that do this are known as *optimizing compilers,* and the resulting code is known as *optimized code*. To produce optimized code, an optimizing compiler performs one or more transformations on the source code provided as input. The purpose is to replace less efficient code with more efficient code while at the same time preserving the meaning and ultimate results of the code. Throughout this section, I will use the term *transformation* to refer to the code modifications performed by an optimizing compiler because I want to distinguish between optimization techniques, such as loop unrolling and common subexpression elimination, and the way in which these techniques are implemented using specific code transformations.

Optimizing compilers use several methods to determine where generated code can be improved. One method is *control flow analysis*, which examines loops and other control constructs, such as `if-then` and `case` statements, to identify the execution paths a program might take and, based on this analysis, determine how the execution path can be streamlined. Another typical optimization technique examines how data is used in a program, a procedure known as *data flow analysis*. Data flow analysis examines how and when variables are used (or not) in a program and then applies various set equations to these usage patterns to derive optimization opportunities.

Optimization properly includes improving the algorithms that make up a program as well as the mechanical transformations described in this chapter. The classic illustration of this is the performance of bubble sorts compared to, say, quick sorts or shell sorts. While code transformations might produce marginally better runtimes for a program that sorts 10,000 items using a bubble sort, replacing the naive bubble sort algorithm with a more sophisticated sorting algorithm will produce dramatically better results. The point is simply that optimization requires expending both CPU cycles and human intellectual energy.

For the purposes of this section, a *basic block* is a sequence of consecutive statements entered and exited without stopping or branching elsewhere in the program. Transformations that occur in the context of a basic block, that is, within a basic block, are known as *local transformations*. Similarly, transformations that do not occur solely within a basic block are known as *global transformations*. As it happens, many transformations can be performed both locally and globally, but the usual procedure is to perform local transformations first.

Although the examples in this section use C, all GCC compilers actually perform transformations on intermediate representations of your program. These intermediate representations are better suited for compiler manipulations than unmodified language-specific source code. GCC uses a sequence of different intermediate representations of your code before generating binaries, as discussed later in this chapter in "GCC Optimization Basics." I use C source code in the examples in this chapter rather than any intermediate representation because, well, people write code in C and other high-level languages, and not directly in any intermediate representation.

GCC compilers perform many other types of optimization, some extremely granular and best left to dedicated discussions of compiler theory. The optimizations listed in this section are some of the more common optimizations that GCC compilers can perform based on the command-line switches that are discussed later in this chapter.

---

■ **Note** The classic compiler reference, affectionately known as "the dragon book" because of the dragon on its cover, is *Compilers: Principles, Techniques, and Tools*, Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman (Addison Wesley Longman, 1986. ISBN: 0-201-10088-6). This seminal text goes into much more detail than this chapter on various optimization techniques; and I'm indebted to it for teaching me this stuff in the first place.

---

## Code Motion

Code motion is an optimization technique that is related to eliminating redundant code through common subexpression elimination (discussed later in this chapter). Code motion does not entirely remove common subexpressions, but attempts to reduce the number of times they occur by relocating them to more optimal locations in an intermediate representation of the code. For example, in nested loops or other control constructs, intermediate value calculations may be performed more times than necessary. In order to optimize such programs, compilers can move these calculations to locations where they will be executed less frequently, but will still produce the same result in the compiled code. Code motion that specifically moves redundant calculations outside the scope of a loop is referred to as *loop-invariant code motion*. Code motion is also used in an optimization technique that is related to common subexpression elimination known as *partial redundancy elimination*.

## Common Subexpression Elimination

Eliminating redundant computations is a standard optimization mechanism because it reduces the number of instructions that a program has to execute in order to arrive at the same result. For example, given an expression whose value is computed only once, each subsequent occurrence of that expression can be eliminated and its value used instead, if the values of the variables in the expression have not changed since first computed. The subsequent occurrences of the expression are called *common subexpressions*. Consider the code fragment in Listing 5-1.

**Listing 5-1.** *An Example of a Common Subexpression*

```
#define STEP 3
#define SIZE 100

int i, j, k;
int p[SIZE];

for (i = 0; i < SIZE; ++i) {
        j = 2 * STEP;
        k = p[i] * j;
}
```

The expression `j = 2 * STEP` in the `for` loop is a common subexpression because its value is computed before entering the loop and its constituent variable, `STEP` (actually a predefined value), has not changed. Common subexpression elimination (CSE) removes the repeated computation of `j` in the `for` loop. After CSE, the `for` loop might look like the following:

```
j = 2 * STEP;
for (i = 0; i < 100; ++i) {
        k = p[i] * j;
}
```

Admittedly, the example is contrived, but it makes the point: CSE has eliminated 100 redundant computations of `j`. CSE improves performance by eliminating unnecessary computations and typically also reduces the size of the resulting binary by eliminating unnecessary instructions.

## Constant Folding

Constant folding is an optimization technique that eliminates expressions that calculate a value that can already be determined when the program is compiled. These are typically calculations that only

reference constant values or expressions that reference variables whose values are constant. For example, both of the calculations in the following code fragment could be replaced with a simple assignment statement:

```
n = 10 * 20 * 400;

i = 10;
j = 20;
ij = i * j;
```

In the latter case, the assignments to the variables `i` and `j` could also be eliminated if these variables were not used elsewhere in the code.

## Copy Propagation Transformations

Another way to reduce or eliminate redundant computations is to perform copy propagation transformations, which eliminate cases in which values are copied from one location or variable to another in order to simply assign their value to another variable. Given an assignment of the form `f = g`, subsequent uses of `f`, called *copies*, use `g` instead. Consider a code fragment such as the following:

```
i = 10;
x[j] = i;
y[MIN] = b;
b = i;
```

Copy propagation transformation might result in a fragment that looks like the following:

```
i = 10;
x[j] = 10;
y[MIN] = b;
b = 10;
```

In this example, copy propagation enables the code to assign values from a static location, which is faster than looking up and copying the value of a variable, and also saves time by eliminating assigning a value to a variable that is itself subsequently used only to propagate that value throughout the code. In some cases, copy propagation itself may not provide direct optimizations, but simply facilitates other transformations, such as code motion and dead code elimination.

*Constant propagation* is a related optimization to copy propagation transformation that optimizes code by replacing variables with constant values wherever possible. Copy propagation focuses on eliminating needless copies between different variables, while constant propagation focuses on eliminating needless copies between predefined values and variables.

## Dead Code Elimination

*Dead code elimination* (DCE) is the term used for optimizations that remove code that doesn't actually do anything permanent or useful. You might wonder why you'd write this type of source code, but it can easily creep into large, long-lived programs even at the source code level. Much dead code elimination is actually done on intermediate representations of your code, which may contain unnecessary intermediate calculations, simply because they are more generic representations of your source code.

*Unreachable code elimination* is a related optimization that removes code that the compiler can identify as impossible to reach. For example, consider the following sample block:

```
if ( i == 10 ) {
    . . .
} else {
    . . .
    if ( i == 10) {
       . . .
    }
}
```

In this example, the nested test for whether `i` is equal to 10 can be eliminated because it can never be reached, since that case would be caught by the first clause of the enclosing `if` construct. Unreachable code elimination is much more likely to occur in the intermediate forms of your code that are generated as part of the compilation process than on anything that is directly visible in your source code.

## If-Conversion

If-conversion is a technique where branch constructs, such as large if-then-elseif-else constructs, are broken into separate `if` statements to simplify generated code, provide opportunities for further optimization, and eliminate jumps and branches wherever possible.

## Inlining

Inlining is an optimization technique where code performance can improve by replacing complex constructs and even function calls with inline representations of the construct or function call. *Code inlining*, or *loop unrolling*, are terms for replacing all or portions of a loop with an explicit series of instructions. *Function inlining* is the term for replacing function calls with the explicit set of instructions that are performed by the function. In general, inlining can reduce code complexity and provide higher performance than would be required by branching that would be done otherwise. It often also provides opportunities for common subexpression elimination and code motion optimization. The classic example of optimization through inlining and unrolling is Duff's Device, which is explained at `http://en.wikipedia.org/wiki/Duff's_device` in more detail than would be useful here.

# GCC Optimization Basics

GCC uses a sequence of different intermediate representations of your code before generating binaries. Converting language-specific code into simpler forms provides several primary advantages:

- Breaking your code into simpler, more low-level constructs exposes opportunities for optimization that may not be directly visible in your source code.
- Using intermediate representations enables the simpler form to more easily and readably represent parse trees that are otherwise at varying levels of complexity.
- Using a simpler form enables the GCC compilers to share optimization mechanisms, regardless of the language in which your code was originally written.

Traditionally, GCC has always internally used an intermediate form known as *Register Transfer Language* (RTL), which is a very low-level language into which all code compiled with a GCC compiler (in any high-level language) is transformed before object code generation begins. Transforming high-level input languages into an intermediate, internal form is a time-tested mechanism for exposing opportunities for optimization. As a very low-level representation of your code, GCC's RTL lends itself well to optimizations that are similarly low-level, such as register allocation and stack and data

optimizations. Because it is relatively low-level, RTL is not as good as one would like in terms of supporting higher-level optimizations related to data types, array and variable references, and overall data and control flow within your code.

With GCC 4.0, the fine makers of the GCC compilers introduced a new intermediate form, static single assignment (SSA), which operates on the parse trees that are produced from your code by GCC compilers, and is thus known as Tree SSA. Without clubbing you to death with algorithms, it is interesting to note that 4.0 and later GCC compilers use two intermediate forms before arriving at a Tree SSA representation, known as GENERIC and GIMPLE. A GENERIC representation is produced by eliminating language-specific constructs from the parse tree that is generated from your code, and a GIMPLE representation is then produced from the GENERIC representation by simplifying address references within the code. Even semantically, you can see that introducing these two intermediate forms and the Tree SSA representation provides several additional points at which optimizations can occur at as high a level as possible before zooming into the depths of the RTL representation of your code.

The best source of information on Tree SSA and the actual steps of the optimization process can be found in various papers and presentations by its authors. One particularly interesting presentation was made at the 2003 GCC Developers' Summit, and is part of the proceedings for that conference. You can find these online at `http://www.linux.org.uk/~ajh/gcc/gccsummit-2003-proceedings.pdf`.

# What's New in GCC 4.x Optimization

The most significant changes to GCC optimization in the GCC 4.x family are related to the introduction of the intermediate Tree SSA format discussed in the previous section. This has provided many new opportunities for optimization and therefore introduced many new options, including `-ftree-ccp`, `-ftree-ch`, `-ftree-copyrename`, `-ftree-dce`, `-ftree-dominator-opts`, `-ftree-dse`, `-ftree-fre`, `-ftree-loop-im`, `-ftree-loop-ivcanon`, `-ftree-loop-linear`, `-ftree-loop-optimize`, `-ftree-lrs`, `-ftree-pre`, `-ftree-sra`, `-ftree-ter`, and `-ftree-vectorize`, which are explained later in this chapter. Due largely to the introduction of these new options, the default optimizations that are invoked by the generic `-O1`, `-O2`, `-O3`, and `-Os` optimization levels have also changed. Another side effect of introducing the intermediate Tree SSA form is that optimization is more consistent, regardless of the input language and GCC compiler you are using.

GCC 4 also introduces substantial improvements in vectorization, thanks largely to contributions from IBM. Vectorization increases the efficiency of processor operations by finding code where a single instruction can be applied to multiple data elements. GCC 4 enables up to 16 scalar operations to be mapped to a single vector operation. This optimization can be especially useful in gaming, graphics, and multimedia applications where instructions are repetitively applied to arrays of values.

GCC 4 also introduces improvements in checking array boundaries and validating the contents and structure of the stack, both of which help automate protections against popular application attacks that attempt to induce buffer or stack overflows.

# Architecture-Independent Optimizations

GCC's optimization knobs and switches fall into two broad categories: architecture-independent and architecture-specific optimizations. This section covers the architecture-independent optimizations. These optimizations do not depend on features specific to a given architecture, such as x86; class of processors, such as Intel IA-32 CPUs; and characteristics of a given instance of a processor family, such as a Pentium IV (Xeon).

GCC's most well-known optimization switches are `-O`; its variant `-On`, where `n` is an integer between 0 and 3; and `-Os`. `-O0` turns off optimization. `-O` and `-O1` (which I will call *level 1 optimizations*) are equivalent, telling GCC to optimize code. With `-O` or `-O1`, the compiler attempts to minimize both

code size and execution time without dramatically increasing compilation time. Using `-O2` and `-O3` will increase the optimization level from that requested by `-O1`, while still invoking the optimization options requested by `-O1`. To minimize code size, use option `-Os`.

The tables in this section show the optimization options associated with various GCC optimization levels. To disable one of them while leaving the others enabled, negate the option using `no-` between `-f` and the optimization name. For example, to disable deferred stack pops, the command line might resemble this:

```
$ gcc myprog.c -o myprog -O1 -fno-defer-pop
```

**Note** `-f` denotes a flag whose operation is machine independent, that is, it requests an optimization that can be applied regardless of the architecture (in most cases). Flags, or flag options, modify GCC's default behavior at a given optimization level but do not require specific hardware support to implement the optimization. As usual, you can specify multiple flags as needed.

## Level 1 GCC Optimizations

The optimizations listed in Table 5-1 are enabled by default when you specify the `-O` or `-O1` optimization options.

**Table 5-1.** *Optimizations Enabled with -O and -O1*

| Optimization | Description |
|---|---|
| `-fcprop-registers` | Attempts to reduce the number of register copy operations performed. |
| `-fdefer-pop` | Accumulates function arguments on the stack. |
| `-fdelayed-branch` | Utilizes instruction slots available after delayed branch instructions. |
| `-fguess-branch-probability` | Uses a randomized predictor to guess branch possibilities. |
| `-fif-conversion` | Converts conditional jumps into nonbranching code. |
| `-fif-conversion2` | Performs if-conversion using conditional execution (on CPUs that support it). |
| `-floop-optimize` | Applies several loop-specific optimizations. |
| `-fmerge-constants` | Merges identical constants used in multiple modules. |
| `-fomit-frame-pointer` | Omits storing function frame pointers in a register. Only activated on systems where this does not interfere with debugging. |
| `-ftree-ccp` | Performs sparse conditional constant propagation (CCP) on SSA trees (GCC 4.x only). |
| `-ftree-ch` | Performs loop header copying on SSA trees, which eliminates a jump and provides opportunities for subsequent code motion optimization (GCC 4.x only). |
| `-ftree-copyrename` | Performs copy renaming on SSA trees, which attempts to rename internal compiler temporary names at copy location to names that more closely resemble the original variable names (GCC 4.x only). |

**Table 5-1.** *Optimizations Enabled with -O and -O1 (Continued)*

| Optimization | Description |
|---|---|
| `-ftree-dce` | Performs dead code elimination (DCE) on SSA trees (GCC 4.x only). |
| `-ftree-dominator-opts` | Performs a variety of optimizations using a dominator tree traversal. A dominator tree is a tree where each node's children are the nodes that it immediately dominates. These cleanups include constant/copy propagation, redundancy elimination, range propagation, expression simplification, and jump threading (reducing jumps to other jumps) (GCC 4.x only). |
| `-ftree-dse` | Performs dead store elimination (DSE) on SSA trees (GCC 4.x only). |
| `-ftree-fre` | Performs full redundancy elimination (FRE) on SSA trees, which only considers expressions that are computed on full paths leading to the redundant compilation. This is similar to and faster than a full partial redundancy elimination (PRE) pass, but discovers fewer redundancies than PRE (GCC 4.x only). |
| `-ftree-lrs` | Performs live range splitting when converting SSA trees back to normal form prior to RTL generation. This creates unique variables in distinct live ranges where a variable is used, providing subsequent opportunities for optimization (GCC 4.x only). |
| `-ftree-sra` | Performs scalar replacement of aggregates, which replaces structure references with scalar values to avoid committing structures to memory earlier than necessary (GCC 4.x only). |
| `-ftree-ter` | Performs temporary expression replacement (TER) when converting SSA trees back to normal form prior to RTL generation. This replaces single-use temporary expressions with the expressions that defined them, making it easier to generate RTL code and provide opportunities for better subsequent optimization in the RTL code (GCC 4.x only). |

Level 1 optimizations comprise a reasonable set of optimizations that include both size reduction and speed enhancement. For example, `-tree-dce` eliminates dead code in applications compiled with GCC 4, thereby reducing the overall code size. Fewer jump instructions mean that a program's overall stack consumption is smaller. `-fcprop-registers`, on the other hand, is a performance optimization that works by minimizing the number of times register values are copied around, saving the overhead associated with register copies.

`-fdelayed-branch` and `-fguess-branch-probability` are instruction scheduler enhancements. If the underlying CPU supports instruction scheduling, these optimization flags attempt to utilize the instruction scheduler to minimize CPU delays while the CPU waits for the next instruction.

The loop optimizations applied when you specify `-floop-optimize` include moving constant expressions above loops and simplifying test conditions for exiting loops. At level 2 and higher optimization levels, this flag also performs strength reduction and unrolls loops.

`-fomit-frame-pointer` is a valuable and popular optimization for two reasons: it avoids the instructions required to set up, save, and restore frame pointers and, in some cases, makes an additional CPU register available for other uses. On the downside, in the absence of frame pointers, debugging (such as generating stack traces, especially from deeply nested functions) can be difficult if not impossible.

-O2 optimization (level 2 optimization) includes all level 1 optimizations plus the additional optimizations listed in Table 5-2. Applying these optimizations will lengthen the compile time, but as a result you should also see a measurable increase in the resulting code's performance, or, rather, a measurable decrease in execution time.

## Level 2 GCC Optimizations

The optimizations listed in Table 5-2 are enabled by default when you specify the -O2 optimization option.

**Table 5-2.** *Optimizations Enabled with -O2*

| Optimization | Description |
|---|---|
| -falign-functions | Aligns functions on powers-of-2 byte boundaries. |
| -falign-jumps | Aligns jumps on powers-of-2 byte boundaries. |
| -falign-labels | Aligns labels on powers-of-2 byte boundaries. |
| -falign-loops | Aligns loops on powers-of-2 byte boundaries. |
| -fcaller-saves | Saves and restores register values overwritten by function calls. |
| -fcrossjumping | Collapses equivalent code to reduce code size. |
| -fcse-follow-jumps | Follows jumps whose targets are not otherwise reached. |
| -fcse-skip-blocks | Follows jumps that conditionally skip code blocks. |
| -fdelete-null-pointer-checks | Eliminates unnecessary checks for null pointers. |
| -fexpensive-optimizations | Performs "relatively expensive" optimizations. |
| -fforce-mem | Stores memory operands in registers (obsolete in GCC 4.1). |
| -fgcse | Executes a global CSE pass. |
| -fgcse-lm | Moves loads outside of loops during global CSE. |
| -fgcse-sm | Moves stores outside of loops during global CSE. |
| -foptimize-sibling-calls | Optimizes sibling and tail recursive function calls. |
| -fpeephole2 | Performs machine-specific peephole optimizations. |
| -fregmove | Reassigns register numbers for maximum register tying. |
| -freorder-blocks | Reorders basic blocks in the compiled function in order to reduce branches and improve code locality. |
| -freorder-functions | Reorders basic blocks in the compiled function in order to improve code locality by using special text segments for frequently and rarely executed functions. |
| -frerun-cse-after-loop | Executes the CSE pass after running the loop optimizer. |
| -frerun-loop-opt | Executes the loop optimizer twice. |
| -fsched-interblock | Schedules instructions across basic blocks. |
| -fsched-spec | Schedules speculative execution of nonload instructions. |

**Table 5-2.** *Optimizations Enabled with -O2 (Continued)*

| Optimization | Description |
|---|---|
| `-fschedule-insns` | Reorders instructions to minimize execution stalls. |
| `-fschedule-insns2` | Performs a second `schedule-insns` pass. |
| `-fstrength-reduce` | Replaces expensive operations with cheaper instructions. |
| `-fstrict-aliasing` | Instructs the compiler to assume the strictest possible aliasing rules. |
| `-fthread-jumps` | Attempts to reorder jumps so they are arranged in execution order. |
| `-ftree-pre` | Performs partial redundancy elimination (PRE) on SSA Trees. |
| `-funit-at-a-time` | Parses the entire file being compiled before beginning code generation, enabling extra optimizations such as reordering code and declarations, and removing unreferenced static variables and functions. |
| `-fweb` | Assigns each web (the live range for a variable) to its own pseudo-register, which can improve subsequent optimizations such as CSE, dead code elimination, and loop optimization. |

The four `-falign-` optimizations force functions, jumps, labels, and loops, respectively, to be aligned on boundaries of powers of 2. The rationale is to align data and structures on the machine's natural memory size boundaries, which should make accessing them faster. The assumption is that code so aligned will be executed often enough to make up for the delays caused by the no-op instructions necessary to obtain the desired alignment.

`-fcse-follow-jumps` and `-fcse-skip-blocks`, as their names suggest, are optimizations performed during the CSE optimization pass described in the first section of this chapter. With `-fcse-follow-jumps`, the optimizer follows jump instructions whose targets are otherwise unreached. For example, consider the following conditional:

```
if (i < 10) {
        foo();
} else {
        bar();
}
```

Ordinarily, if the condition (`i < 10`) is false, CSE will follow the code path and jump to `foo()` to perform the CSE pass. If you specify `-fcse-follow-jumps`, though, the optimizer will not jump to `foo()` but to the jump in the `else` clause (`bar()`).

`-fcse-skip-blocks` causes CSE to skip blocks conditionally. Suppose you have an `if` clause with no `else` statement, such as the following:

```
if (i >= 0) {
        j = foo(i);
}
bar(j);
```

If you specify `-fcse-skip-blocks`, and if, in fact, i is negative, CSE will jump to `bar()`, bypassing the interior of the `if` statement. Ordinarily, the optimizer would process the body of the `if` statement, even if `i` tests false.

■ **Tip** If you use computed `goto`s, GCC extensions discussed in Chapter 2, you might want to disable global CSE optimization using the `-fno-gcse` flag. Disabling global CSE in code using computed `goto`s often results in better performance in the resulting binary.

`-fpeephole2` performs CPU-specific peephole optimizations. During peephole optimization, the compiler attempts to replace longer sets of instructions with shorter, more concise instructions. For example, given the following code:

```
a = 2;
for (i = 1; i < 10; ++i)
        a += 2;
```

GCC might replace the loop with the simple assignment `a = 20`. With `-fpeephole2`, GCC performs peephole optimizations using features specific to the target CPU instead of, or in addition to, standard peephole optimization tricks such as, in C, replacing arithmetic operations with bit operations where this improves the resulting code.

`-fforce-mem` copies memory operands and constants into registers before performing pointer arithmetic on them. The idea behind these optimizations is to make memory references common subexpressions, which can be optimized using CSE. As explained in the first section of this chapter, CSE can often eliminate multiple redundant register loads, which incur additional CPU delays due to the load-store operation.

`-foptimize-sibling-calls` attempts to optimize away tail recursive or sibling call functions. A *tail recursive call* is a recursive function call made in the tail of a function. Consider the following code snippet:

```
int inc(int i)
{
    printf("%d\n" i);
    if(i < 10)
        inc(i + 1);
}
```

This defines a function named `inc()` that displays the value of its argument, `i`, and then calls itself with 1 + its argument, `i + 1`, as long as the argument is less than 10. The tail call is the recursive call to `inc()` in the tail of the function. Clearly, though, the recursive sequence can be eliminated and converted to a simple series of iterations because the depth of recursion is fixed. `-foptimize-sibling-calls` attempts to perform this optimization. A *sibling call* refers to function calls made in a tail context that can also be optimized away.

## GCC Optimizations for Code Size

The `-Os` option is becoming increasingly popular because it applies all of the level 2 optimizations except those known to increase the code size. `-Os` also applies additional techniques to attempt to reduce the code size. Code size, in this context, refers to a program's memory footprint at runtime rather than its on-disk storage requirement. In particular, `-Os` disables the following optimization flags (meaning they will be ignored if specified in conjunction with `-Os`):

- `-falign-functions`
- `-falign-jumps`
- `-falign-labels`
- `-falign-loops`

- `-fprefetch-loop-arrays`
- `-freorder-blocks`
- `-freorder-blocks-and-partition`
- `-ftree-ch`

You might find it instructive to compile a program with `-O2` and `-Os` and compare runtime performance and memory footprint. For example, I have found that recent versions of the Linux kernel have nearly the same runtime performance when compiled with `-O2` and `-Os`, but the runtime memory footprint is 15 percent smaller when the kernel is compiled with `-Os`. Naturally, your mileage may vary and, as always, if it breaks, you get to keep all of the pieces.

## Level 3 GCC Optimizations

Specifying the `-O3` optimization option enables all level 1 and level 2 optimizations plus the following:

- `-fgcse-after-reload`: Performs an extra load elimination pass after reloading
- `-finline-functions`: Integrates all "simple" functions into their callers
- `-funswitch-loops`: Moves branches with loop invariant conditions out of loops

---

■ **Note** If you specify multiple `-O` options, the last one encountered takes precedence. Thus, given the command `gcc -O3 foo.c bar.c -O0 -o baz`, no optimization will be performed because `-O0` overrides the earlier `-O3`.

---

## Manual GCC Optimization Flags

In addition to the optimizations enabled using one of the `-O` options, GCC has a number of specialized optimizations that can only be enabled by specifically requesting them using `-f`. Table 5-3 lists these options.

**Table 5-3.** *Specific GCC Optimizations*

| Flag | Description |
|---|---|
| `-fbounds-check` | Generates code to validate indices used for array access. |
| `-fdefault-inline` | Compiles C++ member functions inline by default. |
| `-ffast-math` | Sets `-fno-math-errno`, `-funsafe-math-optimizations`, and `-fno-trapping-math`. |
| `-ffinite-math-only` | Disables checks for NaNs and infinite arguments and results. |
| `-ffloat-store` | Disables storing floating-point values in registers. |
| `-fforce-addr` | Stores memory constants in registers. |
| `-ffunction-cse` | Stores function addresses in registers. |
| `-finline` | Expands functions inline using the `inline` keyword. |
| `-finline-functions` | Expands simple functions in the calling function. |

**Table 5-3.** *Specific GCC Optimizations*

| Flag | Description |
|---|---|
| `-finline-limit=n` | Limits inlining to functions no greater than `n` pseudo-instructions. |
| `-fkeep-inline-functions` | Keeps inline functions available as callable functions. |
| `-fkeep-static-consts` | Preserves unreferenced variables declared `static const`. |
| `-fmath-errno` | Sets `errno` for math functions executed as a single instruction. |
| `-fmerge-all-constants` | Merges identical variables used in multiple modules. |
| `-ftrapping-math` | Emits code that generates user-visible traps for FP operations. |
| `-ftrapv` | Generates code to trap overflow operations on signed values. |
| `-funsafe-math-optimizations` | Disables certain error checking and conformance tests on floating-point operations. |

Many of the options listed in Table 5-3 involve floating-point operations. In order to apply the optimizations in question, the optimizer deviates from strict adherence to ISO and/or IEEE specifications for math functions in general and floating-point math in particular. In floating-point heavy applications, you might see significant performance improvements, but the trade-off is that you give up compatibility with established standards. In some situations, noncompliant math operations might be acceptable, but you are the only one who can make that determination.

---

■ **Note** Not all of GCC's optimizations can be controlled using a flag. GCC performs some optimizations automatically and, short of modifying the source code, you cannot disable these optimizations when you request optimization using `-O`.

---

# Processor-Specific Optimizations

Due to the wide variety of processor architectures GCC supports, I cannot begin to cover the processor-specific optimizations in this chapter. Appendix A covers, in detail, all of the processor-specific optimizations you can apply. So curious readers who know the details of their CPUs are encouraged to review the material there. In reality, though, the target-specific switches discussed in Appendix A are not properly optimizations in the traditional sense. Rather, they are options that give GCC more information about the type of system on which the code being compiled will run. GCC can use this additional information to generate code that takes advantage of specific features or that works around known misfeatures of the given processor.

Before starting work on this book, I usually used the (architecture-independent) `-O2` option exclusively, and left it to the compiler to do the right thing otherwise. After writing this book, I have expanded my repertoire, adding some additional options that I have found to be useful in specific cases. My goal is to provide some guidelines and tips to help you select the right optimization level and, in some situations, the particular optimization you want GCC to apply. These can only be guidelines, though, because you know your code better than I do.

# Automating Optimization with Acovea

If this book teaches you nothing else, you've learned that together the GCC compilers offer approximately 1.3 zillion options. Trying to find the absolute best set of GCC options to use for a specific application and architecture can be tedious at best; so most people stick with the standard GCC optimization options, invoking selected others they have found to be useful over time. This is something of a shame because it overlooks additional optimization possibilities; but your development time has to be optimized, too.

Scott Ladd's Acovea application (`http://www.coyotegulch.com/products/acovea/index.html`) provides an interesting and useful mechanism for deriving optimal sets of optimization switches, using an evolutionary algorithm that emulates natural selection. Lest this sound like voodoo, let's think about how optimization works in the first place. Optimization applies algorithms that potentially improve various code segments; checks the results; and retains those that result in code improvements. This is conceptually akin to the first step of the natural selection process. Acovea simply automates the propagation of satisfactory optimizations to subsequent optimization passes, which apply additional optimizations and measure the results. Optimizations that do not indeed optimize or improve the code are therefore treated as evolutionary dead ends that are not explored.

GCC experts, random posts on the Internet, and books such as this one all offer a variety of general suggestions for trying to derive "the best" optimizations. Unfortunately, these can be contradictory and can never take into account the content and structure of your specific application. Acovea attempts to address this by enabling the exhaustive, automatic analysis of the performance of applications compiled with an iterative set of optimization options. Its exhaustive analysis also enables you to empirically derive information about the most elusive aspect of GCC's optimization options—their interaction. Invoking one promising optimization option may have significant impact on the performance of other optimization options. Acovea enables you to automatically test all configured GCC options in combination with each other, helping you locate the holy grails of application development, the smallest or fastest compiled version of your application.

## Building Acovea

You can download the source code for the latest version of Acovea from links located at `http://www.coyotegulch.com/products/acovea/index.html`. Acovea requires that you build and install two additional libraries before actually building Acovea:

- `coyotl`: A library of various routines used by Acovea, including a specialized random number generator, low-level floating point utilities, a generalized command-line parser, and improved sorting and validation tools.
- `evocosm`: A library that provides a framework for developing evolutionary algorithms.

After building and installing these libraries (in order), using the traditional "unpack, configure, make install" sequence, you can build and install Acovea itself. Downloading and building the libacovea package compiles and installs an application called runacovea in `/usr/local/bin` (by default), which is the master control program for GCC optimization testing using Acovea.

---

■ **Note** Acovea is only supported on actual Linux and Unix-like systems—some work will be required to get it working under Cygwin, but I'm sure Scott Ladd will appreciate any contributions you'd like to make.

---

## Configuring and Running Acovea

The options that Acovea will test for you are defined in an XML-format configuration file. Template configuration files are available from Scott's site at `http://www.coyotegulch.com/products/acovea/acovea-config.html`. These configuration files are highly dependent on the version of GCC you're using, so make sure you get the configuration files for that version of GCC, and also for your specific class of processor. The following is a sample section of an Acovea configuration file:

```
<?xml version="1.0"?>
<acovea_config>
  <acovea version="5.2.0" />
  <description value="gcc 4.1 Opteron (AMD64/x86_64)" />
  <quoted_options value="false" />
  <prime command="gcc"
        flags="-lrt -lm -std=gnu99 -O1 -march=opteron ACOVEA_OPTIONS
            -o ACOVEA_OUTPUT ACOVEA_INPUT" />
  <baseline description="-O1"
        command="gcc"
        flags="-lrt -lm -std=gnu99 -O1 -march=opteron -o ACOVEA_OUTPUT
            ACOVEA_INPUT" />
  <baseline description="-O2"
        command="gcc"
        flags="-lrt -lm -std=gnu99 -O2 -march=opteron -o ACOVEA_OUTPUT
            ACOVEA_INPUT" />
  <baseline description="-O3"
        command="gcc"
        flags="-lrt -lm -std=gnu99 -O3 -march=opteron -o ACOVEA_OUTPUT
            ACOVEA_INPUT" />
  <baseline description="-O3 -ffast-math"
        command="gcc"
        flags="-lrt -lm -std=gnu99 -O3 -march=opteron -ffast-math
            -o ACOVEA_OUTPUT ACOVEA_INPUT" />
  <baseline description="-Os"
        command="gcc"
        flags="-lrt -lm -std=gnu99 -Os -march=opteron -o ACOVEA_OUTPUT
            ACOVEA_INPUT" />

  <!-- A list of flags that will be "evolved" by ACOVEA (85 for GCC 4.1!) -->
  <flags>
    <!-- O1 options (these turn off options implied by -O1) -->
    <flag type="simple" value="-fno-merge-constants" />
    <flag type="simple" value="-fno-defer-pop" />
    <flag type="simple" value="-fno-thread-jumps" />
    <flag type="enum"
        value="-fno-omit-frame-pointer|-momit-leaf-frame-pointer" />
    <flag type="simple" value="-fno-guess-branch-probability" />
    <flag type="simple" value="-fno-cprop-registers" />
    <flag type="simple" value="-fno-if-conversion" />
     . . ..
      <!-- O2 options -->
      <flag type="simple" value="-fcrossjumping" />
      <flag type="simple" value="-foptimize-sibling-calls" />
      <flag type="simple" value="-fcse-follow-jumps" />
      <flag type="simple" value="-fcse-skip-blocks" />
```

```
      <flag type="simple" value="-fgcse" />
      <flag type="simple" value="-fexpensive-optimizations" />
      <flag type="simple" value="-fstrength-reduce" />
      <flag type="simple" value="-frerun-cse-after-loop" />
      <flag type="simple" value="-frerun-loop-opt" />
     …
      <!-- O3 options -->
     <flag type="simple" value="-fgcse-after-reload" />
     <flag type="simple" value="-finline-functions" />
     <flag type="simple" value="-funswitch-loops" />
     …
     <!-- Additional options -->
     <flag type="simple" value="-ffloat-store" />
      <flag type="simple" value="-fprefetch-loop-arrays" />
      <flag type="simple" value="-fno-inline" />
      <flag type="simple" value="-fpeel-loops" />
      …
      <!-- Tuning options that have a numeric value -->
      <flag type="tuning" value="-finline-limit" default="600" min="100"
            max="10000" step="100" separator="=" />
   </flags>
</acovea_config>
```

---

■ **Note** Acovea can be used with any of the GCC compilers by specifying the compiler that you want to run in the baseline element's `command` attribute in your Acovea configuration file.

---

Once you have downloaded and optionally customized your configuration file, use the runacovea application to test GCC with the selected options, as in the following example:

```
runacovea –config config-file-name –input source-file-name
```

---

■ **Note** By default, Acovea selects optimization options that produce the fastest, highest-performance code, but it can also be instructed to optimize for size by specifying the `–size` option on the runacovea command line.

---

Executing the runacovea application produces a variety of output as Acovea tests your compiler with permutations of the specified options, culminating in output such as the following:

---

```
Acovea completed its analysis at 2005 Nov 24 08:45:34

Optimistic options:

                        -fno-defer-pop  (2.551)
                     -fmerge-constants  (1.774)
                    -fcse-follow-jumps  (1.725)
                        -fthread-jumps  (1.822)

Pessimistic options:

                        -fcaller-saves  (-1.824)
                      -funswitch-loops  (-1.581)
                        -funroll-loops  (-2.262)
```

```
          -fbranch-target-load-optimize2  (-2.31)
                              -fgcse-sm  (-1.533)
                    -ftree-loop-ivcanon  (-1.824)
                           -mfpmath=387  (-2.31)
                           -mfpmath=sse  (-1.581)

Acovea's Best-of-the-Best:
gcc -lrt -lm -std=gnu99 -O1 -march=opteron -fno-merge-constants
    -fno-defer-pop -momit-leaf-frame-pointer -fno-if-conversion
    -fno-loop-optimize -ftree-ccp -ftree-dce -ftree-dominator-opts
    -ftree-dse -ftree-copyrename -ftree-fre -ftree-ch -fmerge-constants
    -fcrossjumping -fcse-follow-jumps -fpeephole2 -fschedule-insns2
    -fstrict-aliasing -fthread-jumps -fgcse-lm -fsched-interblock -fsched-spec
    -freorder-functions -funit-at-a-time -falign-functions -falign-jumps
    -falign-loops -falign-labels -ftree-pre -finline-functions -fgcse-after-reload
    -fno-inline -fpeel-loops -funswitch-loops -funroll-all-loops -fno-function-cse
    -fgcse-las -ftree-vectorize -mno-push-args -mno-align-stringops
    -minline-all-stringops -mfpmath=sse,387 -funsafe-math-optimizations
    -finline-limit=600 -o /tmp/ACOVEAA7069796 fibonacci_all.c

Acovea's Common Options:
gcc -lrt -lm -std=gnu99 -O1 -march=opteron -fno-merge-constants
    -fno-defer-pop -momit-leaf-frame-pointer -fcse-follow-jumps -fthread-jumps
    -ftree-pre -o /tmp/ACOVEAAA635117 fibonacci_all.c

-O1:
gcc -lrt -lm -std=gnu99 -O1 -march=opteron -o /tmp/ACOVEA58D74660 fibonacci_all.c

-O2:
gcc -lrt -lm -std=gnu99 -O2 -march=opteron -o /tmp/ACOVEA065F6A10 fibonacci_all.c

-O3:
gcc -lrt -lm -std=gnu99 -O3 -march=opteron -o /tmp/ACOVEA934D7357 fibonacci_all.c

-O3 -ffast-math:
gcc -lrt -lm -std=gnu99 -O3 -march=opteron -ffast-math -o /tmp/ACOVEA408E67B6
fibonacci_all.c

-Os:
gcc -lrt -lm -std=gnu99 -Os -march=opteron -o /tmp/ACOVEAAB2E22A4 fibonacci_all.c
```

As you can see, Acovea produces a listing of the combination of options that produce the best results, as well as information about optimization options that it suggests should be added to each of the standard GCC optimization options.

As mentioned previously, exhaustively testing all GCC optimization options and determining the interaction between them takes a prohibitive amount of time for mere mortals, even graduate students and interns. Acovea is an impressive application that can do this for you, and can help you automagically produce an optimal executable for your application. For more detailed information about using Acovea and integrating it into the make process for more complex applications than the simple example we've used, see `http://www.coyotegulch.com/products/acovea`.