**Optimization in GCC**

In this article, we explore the optimization levels provided by the GCC compiler toolchain, including the specific optimizations provided in each. We also identify optimizations that require explicit specifications, including some with architecture dependencies. This discussion focuses on the 3.2.2 version of gcc (released February 2003), but it also applies to the current release, 3.3.2.

Levels of Optimization

Let's first look at how GCC categorizes optimizations and how a developer can control which are used and, sometimes more important, which are not. A large variety of optimizations are provided by GCC. Most are categorized into one of three levels, but some are provided at multiple levels. Some optimizations reduce the size of the resulting machine code, while others try to create code that is faster, potentially increasing its size. For completeness, the default optimization level is zero, which provides no optimization at all. This can be explicitly specified with option -O or -O0.

Level 1 (-O1)

The purpose of the first level of optimization is to produce an optimized image in a short amount of time. These optimizations typically don't require significant amounts of compile time to complete. Level 1 also has two sometimes conflicting goals. These goals are to reduce the size of the compiled code while increasing its performance. The set of optimizations provided in -O1 support these goals, in most cases. These are shown in Table 1 in the column labeled -O1. The first level of optimization is enabled as:

gcc -O1 -o test test.c



Table 1. GCC optimizations and the levels at which they are enabled.

Any optimization can be enabled outside of any level simply by specifying its name with the -f prefix, as:

gcc -fdefer-pop -o test test.c

We also could enable level 1 optimization and then disable any particular optimization using the -fno- prefix, like this:

gcc -O1 -fno-defer-pop -o test test.c

This command would enable the first level of optimization and then specifically disable the defer-pop optimization.

Level 2 (-O2)

The second level of optimization performs all other supported optimizations within the given architecture that do not involve a space-speed trade-off, a balance between the two objectives. For example, loop unrolling and function inlining, which have the effect of increasing code size while also potentially making the code faster, are not performed. The second level is enabled as:

gcc -O2 -o test test.c

Table 1 shows the level -O2 optimizations. The level -O2 optimizations include all of the -O1 optimizations, plus a large number of others.

Level 2.5 (-Os)

The special optimization level (-Os or size) enables all -O2 optimizations that do not increase code size; it puts the emphasis on size over speed. This includes all second-level optimizations, except for the alignment optimizations. The alignment optimizations skip space to align functions, loops, jumps and labels to an address that is a multiple of a power of two, in an architecture-dependent manner. Skipping to these boundaries can increase performance as well as the size of the resulting code and data spaces; therefore, these particular optimizations are disabled. The size optimization level is enabled as:

gcc -Os -o test test.c

In gcc 3.2.2, reorder-blocks is enabled at -Os, but in gcc 3.3.2 reorder-blocks is disabled.

Level 3 (-O3)

The third and highest level enables even more optimizations (Table 1) by putting emphasis on speed over size. This includes optimizations enabled at -O2 and rename-register. The optimization inline-functions also is enabled here, which can increase performance but also can drastically increase the size of the object, depending upon the functions that are inlined. The third level is enabled as:

gcc -O3 -o test test.c

Although -O3 can produce fast code, the increase in the size of the image can have adverse effects on its speed. For example, if the size of the image exceeds the size of the available instruction cache, severe performance penalties can be observed. Therefore, it may be better simply to compile at -O2 to increase the chances that the image fits in the instruction cache.

### Optimization JAVA

Along with dynamic compilation comes the opportunity to insert performance counters. The compiler might, for instance, insert a performance counter to count every time a bytecode block (e.g, corresponding to a specific method) was called. Compilers use data about how "hot" a given bytecode is to determine where in the code optimizations will best impact the running application. Runtime profiling data enables the compiler to make a rich set of code optimization decisions on the fly, further improving code-execution performance. As more refined code-profiling data becomes available it can be used to make additional and better optimization decisions, such as: how to better sequence instructions in the compiled-to language, whether to replace a set of instructions with more efficient sets, or even whether to eliminate redundant operations.

#### Example

Consider the Java code:

static int add7( int x ) {

return x+7;

}

This could be statically compiled by javac to the bytecode:

iload0

bipush 7

iadd

ireturn

When the method is called the bytecode block will be dynamically compiled to machine instructions. When a performance counter (if present for the code block) hits a threshold it might also get optimized. The end result could look like the following machine instruction set for a given execution platform:

lea rax,[rdx+7]

ret

I've so far discussed the value of optimizing code and how and when common JVM compilers optimize code. I'll conclude with some of the actual optimizations available to compilers. JVM optimization actually happens at the bytecode level (or on lower representative language levels), but I'll demonstrate the optimizations using the Java language. I couldn't possibly cover all of the JVM optimizations in this section; rather, I mean to inspire you to explore on your own and learn about the hundreds of advanced optimizations and innovations in compiler technology (see [Resources](http://www.javaworld.com/javaworld/jw-09-2012/120905-jvm-performance-optimization-compilers.html?page=5" \l "resources)).

### Dead code elimination

Dead code elimination is what it sounds like: the elimination of code that has never been called -- i.e., "dead" code. If a compiler discovers during runtime that some instructions are unnecessary, it will simply eliminate them from the execution instruction set. For example, in Listing 1 a certain value assignment for a variable is never used and can be fully ignored at execution time. On a bytecode level this could correspond to never needing to execute the load of the value into a register. Not having to do the load means less CPU time, and hence a quicker code execution, and therefore the application -- especially if the code is hot and called several times per second.

Listing 1 shows Java code exemplifying a variable that is never used, an unnecessary operation.

#### Listing 1. Dead code

int timeToScaleMyApp(boolean endlessOfResources) {

int reArchitect = 24;

int patchByClustering = 15;

int useZing = 2;

if(endlessOfResources)

return reArchitect + useZing;

else

return useZing;

}

On a bytecode level, if a value is loaded but never used, the compiler can detect this and eliminate the dead code, as shown in Listing 2. Never executing the load saves CPU time and thus improves the program's execution speed.

#### Listing 2. The same code following optimization

int timeToScaleMyApp(boolean endlessOfResources) {

int reArchitect = 24;

//unnecessary operation removed here...

int useZing = 2;

if(endlessOfResources)

return reArchitect + useZing;

else

return useZing;

}

Redundancy elimination is a similar optimization that removes duplicate instructions to improve application performance.

### Inlining

Many optimizations try to eliminate machine-level jump instructions (e.g., JMP for x86 architectures). A jump instruction changes the instruction pointer register and thereby transfers the execution flow. This is an expensive operation relative to other ASSEMBLY instructions, which is why it is a common target to reduce or eliminate. A very useful and well-known optimization that targets this is called inlining. Since jumping is expensive, it can be helpful to inline many frequent calls to small methods, with different entry addresses, into the calling function. The Java code in Listings 3 through 5 exemplifies the benefits of inlining.

#### Listing 3. Caller method

int whenToEvaluateZing(int y) {

return daysLeft(y) + daysLeft(0) + daysLeft(y+1);

}

#### Listing 4. Called method

int daysLeft(int x){

if (x == 0)

return 0;

else

return x - 1;

}

#### Listing 5. Inlined method

int whenToEvaluateZing(int y){

int temp = 0;

if(y == 0) temp += 0; else temp += y - 1;

if(0 == 0) temp += 0; else temp += 0 - 1;

if(y+1 == 0) temp += 0; else temp += (y + 1) - 1;

return temp;

}

In Listings 3 through 5 the calling method makes three calls to a small method, which we assume for this example's sake is more beneficial to inline than to jump to three times.

It might not make much difference to inline a method that is called rarely, but inlining a so-called "hot" method that is frequently called could mean a huge difference in performance. Inlining also frequently makes way for further optimizations, as shown in Listing 6.

#### Listing 6. After inlining, more optimizations can be applied

int whenToEvaluateZing(int y){

if(y == 0) return y;

else if (y == -1) return y - 1;

else return y + y - 1;

}

### Loop optimization

Loop optimization plays a big role when it comes to reducing the overhead that comes with executing loops. Overhead in this case means expensive jumps, number of checks of the condition, non-optimal instruction pipeline (i.e., an order of instructions that causes no-operations or extra cycles in the CPU). There are many kinds of loop optimizations, amounting to a vast set of optimizations. Notables include:

* **Combining loops**: When two nearby loops are iterated the same amount of times, the compiler can try to combine the bodies of the loops, to be executed at the same time (in parallel) in the case where nothing in the bodies reference each other, i.e., they are fully independent of each other.
* **Inversion loops**: Basically you replace a regular while loop with a do-while loop. And the do-while loop is set within an if clause. This replacement leads to two less jumps. However, it adds to the condition check and hence increases the code size. This optimization is an excellent example of how using slightly more resources leads to a more efficient code - a cost-gain balance the compiler has to evaluate and decide on dynamically during runtime.
* **Tiling loops**: Reorganizes the loop so that it iterates over blocks of data that are sized to fit in the cache.
* **Unrolling loops**: Reduces the number of times the loop condition has to be evaluated and also the number of jumps. You can think of this as "inlining" several iterations of the body to be executed without crossing the loop condition. Unrolling loops comes with risk, as it might decrease performance by impairing the pipeline and causing multiple redundant instruction fetches. Again, this is a judgment call by the compiler to make at runtime, i.e., if the gain is enough, the cost might be worth it.

This has been an overview of what a compiler does on a bytecode level (and below) to improve an application's execution performance on a target platform. The optimizations discussed are common and popular, but only a brief sampling of the available options. These have been very simple and broad explanations, which hopefully serve to pique your interest for more in-depth exploration. See [Resources](http://www.javaworld.com/javaworld/jw-09-2012/120905-jvm-performance-optimization-compilers.html?page=6" \l "resources) for further reading.

## In conclusion: Reflection points and highlights

Use different compilers for different needs.

* Interpretation is the simplest form of bytecode translation to machine instructions, and works based on an instruction lookup table.
* Compilers allow for optimization based on performance counters, but will require some additional resources (code cache, optimization threads, etc.)
* Client-side compilers improve the performance of execution code by an order of magnitude (5 to 10 times better) when compared to interpreted code.
* Server-side compilers improve application performance by 30 percent to 50 percent over client-side compilers, but utilize more resources.
* Tiered compilation provides the best of two worlds. Enable client compilation to get your code performing well quickly, and server compilation over time, to make frequently called code execute even better.

There are many possible code optimizations. An important task for the compiler is to analyze all possibilities and weigh the cost of using an optimization against the execution speed benefit of the output machine code.