GCC as Optimising Compiler

Ivan Voras

Feb 17, 2004.

Abstract

This paper aims to present and analyse some of the optimisations that are conducted by the GNU C compiler, in version 3.3.4 (on a i386 FreeBSD system). This is only an informal and introductory paper and is not meant to be comprehensive nor complete, only to give insight at things that are happening "behind the scenes".

1 Importance of Compiler Optimisations

There was a time when high-level languages were used only in cases when performance was not critical – primarily because the compilers produced very unoptimised and naive machine code. The reason behind it was that the computers themselves did not have the capacity to run complex programs such as optimising compilers. Optimising machine code is expensive in both the complexity of operations and in the memory space required to keep track of data structures involved in the process, and the produced code was still far from that made by a skilled assembler programmer. Since then, several things have changed: computers have become faster, and equally more complex, by several orders of magnitude in a very short time, memory became cheap and huge (both as compared to ones deployed about 20 years ago) and compilers could execute algorithms that were once deemed unthinkably expensive. As a consequence, really skilled programmers (both assembler and high-level specialists) have become sort of a rarity. That condition lead to more research in compiler optimisations, and now there exists a paradox that the code produced by a compiler is often faster than the one hand-optimised by a programmer. Today's programmers don't have to be (and often aren't) as skilled as they used to be, and the responsibility to generate good code has fallen on to the compiler itself.

Although this author greatly prefers the "Intel" notation of assembler programmes, since this paper concentrates on the gcc compiler, the assembler output generated by gcc, in "AT&T" form, is used.

2 Some of the Techniques of Optimisation

Today's CPUs are extremely complex. In order to achieve higher and higher execution speeds, they employ pipelining, branch prediction, out-of-order instruction execution, delayed branching etc. Human mind cannot keep track of all consequences of code execution on such a complex machine, and besides that - different processor models have different execution optimisations and often slightly different instruction sets (e.g. Intel SSE vs AMD 3DNow!).

Using such model-specific processor instructions is the next-to-simplest form of optimisation (the simplest form is padding memory structures and code to start on predetermined memory boundaries that are faster to access). The problem with such code is that it doesn't run on all common CPUs, even if they are nominally of the same family (e.g. Pentium 2 vs Pentium 3). Some operating environments are lucky because most of the applications are provided in the form of the source code (most Unix-like environments), and by compiling for a specific processor model the user can be sure that the application is using all the available processor features and runs at its optimum. Other environments, where distribution of already-compiled applications is much more common need "blended code", which contains several execution paths that are chosen depending on the CPU features (e.g. a performance-critical function could be compiled (and included in the executable) several times: once containing only 80386 code, once with MMX instructions, once optimised for Pentium IV with SSE2). Blended code is an important feature of compilers targeted for MS Windows and other commercial environments. That's about it for using model-specific instructions – they normally *are* faster, and if they can be used, the improvements can be great, but they are not a panacea - only using the extended instruction set instead of the usual one does not guarantee optimal, or even faster execution.

Next methods are ones that involve redistribution and reordering of the instructions to take advantage of some specific processor feature, such as the length of its pipeline, how it predicts the execution, the size its caches, etc.

Much more interesting are the methods that eliminate code redundancies in the code made by the programmer, such as double-assigning and algorithmical redundancies like "empty" loops. These are actually the most difficult optimisations, because they have to track the behaviour of the program and its internal state.

Compilers employing all these optimisations can significantly alter the structure of the program, making the end-result machine code seemingly unrelated to the original high-level source. For example, order of code blocks can be changed, loops can be combined or expanded, some code can be missing or duplicated in places that have little or no apparent connection. It all depends on how well the compiler "understands" the code, and how well it can adapt it to suit a particular processor.

3 GCC Examples

The -S switch of gcc is used to generate assembler-source output instead of executable machine code. Such code is presented and examined here. Here is the C source that is used to generate code:

```
int constant() {
    return 3;
}

void main() {
    int a = 2;

    a += constant();

    while (!a) {
        puts ("first");
    }

    do {
        puts ("second");
    } while (!a);
}
```

This is a simple program, with several obvious chances for optimisation. Let's see how the compiler copes with the code.

3.1 No optimisations (-O0)

The compiler is called as "GCC33 -O TEST_0.S -O0 -S TEST.C".

```
        .file   "test1.c"
        .text
.globl constant
        .type   constant, @function
```

This is the preamble. It servers to keep some information internal to the compiler.

```
constant:
        pushl   %ebp
        movl    %esp, %ebp
        movl    $3, %eax
        leave
        ret
        .size   constant, .-constant
```

Here is the compiled code of the function CONSTANT(). The function is quite literally translated into machine code, starting with setting up the stack, moving the literal constant "3" into the return-register (%eax), cleaning up the stack and returning from the function.

```
        .section        .rodata
.LC0:
        .string "first"
.LC1:
        .string "second"
```

This part is the storage for the string constants used.

```
        .text
.globl main
        .type   main, @function
main:
```

The main() function starts here.

```
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp
        movl    $2, -4(%ebp)
```

The above code is generated by the "int a=2;" line. In effect, this reserves a variable on the stack (located at -4(%ebp)), and writes integer "2" to it (the last line).

```
        call    constant
        movl    %eax, %edx
        leal    -4(%ebp), %eax
        addl    %edx, (%eax)
```

This code calls the function, and adds it's result (located in %eax, then transfered to the %edx) to the variable.

```
.L3:
        cmpl    $0, -4(%ebp)
        je      .L5
        jmp     .L4
```

The variable is tested. If it is equal to zero (meaning it's evaluated as false), the execution jumps to the label "L5". Else, the execution jumps to L4. This is bad code, because a code jump is unavoidable, even if it is only one instruction away (jumps are generally the slowest instructions available).

```
.L5:
        subl    $12, %esp
        pushl   $.LC0
        call    puts
        addl    $16, %esp
        jmp     .L3
```

The address of the LC0 (literal-constant 0) is pushed to the stack for the PUTS() function and the function is called. A jump is then made to the beginning of the loop (the testing of the loop condition).

```
.L4:
        nop
```

I really don't know why the no-operation code is inserted here. Maybe its purpose is to align the following code on some boundary. More likely, it is a residual of some internal placeholder.

```
.L6:
        subl    $12, %esp
        pushl   $.LC1
        call    puts
        addl    $16, %esp
        cmpl    $0, -4(%ebp)
        je      .L6
```

This loop (do...while) is visibly more compact than the above one. Such behaviour has resulted in "anecdotal advice" between programmers to use this form of the loop instead of "while...{}" in order to achieve performance gains. However, see the results of more advanced optimisations.

```
        leave
        ret
```

Function ends.

```
        .size   main, .-main
        .ident  "GCC: (GNU) 3.3.4 20040216 (prere-
lease) [FreeBSD]"
```

Standard epilogue.

3.2 Simple optimisations (-O1, or just -O)

The compiler is called as "GCC33 -O TEST_1.S -O1 -S TEST.C".

```
        .file   "test1.c"
        .text
.globl constant
        .type   constant, @function
constant:
        pushl   %ebp
        movl    %esp, %ebp
        movl    $3, %eax
        leave
        ret
        .size   constant, .-constant
        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0:
        .string "first"
.LC1:
        .string "second"
        .text
.globl main
        .type   main, @function
```

This part is virtually identical to the one without optimisations. The only difference is use of additional linker flags in the .section directive above.

```
main:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        subl    $4, %esp
        andl    $-16, %esp
        call    constant
        movl    %eax, %ebx
        addl    $2, %ebx
        jne     .L7
```

Here we see that the initialisation of the variable "a", the call to the CONSTANT() function, and the initial testing of the loop condition are combined. The order of operations is no longer "initialise a; call function; add its result to the variable", rather it is all mixed together, and what's more, the variable isn't even stored in memory! During the whole execution of the MAIN() function the variable exists in the %ebx register, and that is a big speedup.

The last line of the above code is a conditional jump (jump-if-not-equal), and it depends on the result of the previous instruction (addl), almost like a side-effect. The compiler now "understands" the effects various commands have on each other and doesn't just translate blocks of code into other blocks of code. Also, there is no obligatory jump here: if the condition is not met, the execution naturaly flows to the next instruction.

```
.L6:
        subl    $12, %esp
        pushl   $.LC0
        call    puts
        addl    $16, %esp
        testl   %ebx, %ebx
        je      .L6
```

Although the original form of the loop is "while(condition) do {...}", here we see that the condition is tested at the end of the loop. This is because of 2 reasons: first, because the condition at the beginning of the loop is tested "implicitly", and second, because branch-predicting logic in the processor usually "predicts" that the conditional forward-jumps are not taken and the conditional back-jumps are taken (thus optimising the case where the conditional jumps form loops).

```
.L7:
        subl    $12, %esp
        pushl   $.LC1
        call    puts
        addl    $16, %esp
        testl   %ebx, %ebx
        je      .L7
```

The second loop is very similar to the unoptimised case, because condition testing must appear at the end of the loop, and the loop code block must be executed at least once. The only difference is the choice of condition-testing instruction, which relies more on the side-effects of instructions. Notice that there are only 2 jump-labels in the above code, which is a vast improvement over the 4 that exist in the unoptimised case.

```
        movl    -4(%ebp), %ebx
        leave
        ret
```

The compiler loads the variable into the register "just in case" it is needed. The movl instruction has no apparent purpose in the above code and could be left out by further optimisations.

```
        .size   main, .-main
        .ident  "GCC: (GNU) 3.3.4 20040216 (prere-
lease) [FreeBSD]"
```

The epilogue is standard.

3.3 More optimisations (-O2)

The compiler is called as "GCC -O TEST_2.S -O2 -S TEST.C".

```
        .file   "test1.c"
        .text
        .p2align 2, ,3
```

Here, for the first time we see that padding is used to squeeze performance from the code. As modern CPUs access aligned memory data much faster than unaligned, this can give a significant performance boost in loops. The .p2align directive above states that code should be aligned on addresses which are multiple of 4 bytes ($=2^2$), but only if it doesn't mean wasting more than 3 bytes of space.

```
.globl constant
        .type   constant, @function
constant:
        pushl   %ebp
        movl    %esp, %ebp
        movl    $3, %eax
        leave
        ret
        .size   constant, .-constant
        .section        .rodata.str1.1,"aMS",@progbits,1
.LC1:
        .string "second"
.LC0:
        .string "first"
        .text
        .p2align 2, ,3
.globl main
        .type   main, @function
```

This code is same as before, except for the alignment directive.

```
main:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        pushl   %eax
        andl    $-16, %esp
        call    constant
        movl    %eax, %ebx
        addl    $2, %ebx
        je      .L6
        .p2align 2, ,3
```

This code is same as before, except for the alignment directive.

```
.L7:
```

```
        subl    $12, %esp
        pushl   $.LC1
        call    puts
        addl    $16, %esp
        testl   %ebx, %ebx
        je      .L7
        movl    -4(%ebp), %ebx
        leave
        ret
        .p2align 2, ,3
```

Now we see something strange - the order of the loops has changed! The compiler has, for some reason, changed the order in which the two loops appear in the generated code, although the execution path remains the same.

```
.L6:
        subl    $12, %esp
        pushl   $.LC0
        call    puts
        addl    $16, %esp
        testl   %ebx, %ebx
        je      .L6
        jmp     .L7
```

Here is the possible reason for this change of order: a back-jump to L7 is made, and thus the code for the loop might well be cached in memory.

```
        .size   main, .-main
        .ident  "GCC: (GNU) 3.3.4 20040216 (prere-
lease) [FreeBSD]"
```

Standard epilogue.

3.4 Maximum optimisations (-O3)

The compiler is called as "GCC -O TEST_2.S -O3 -S TEST.C".

```
        .file   "test1.c"
        .section        .rodata.str1.1,"aMS",@progbits,1
.LC1:
        .string "second"
        .text
```

Now we see some real effort from the compiler! The constant function was recognised as such, but was for some reason kept in the code below "just in case" (probably because some other module could have called it). Also, a literal constant is missing, for reasons described below.

```
        .p2align 2, ,3
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        subl    $12, %esp
        pushl   $.LC1
        call    puts
        addl    $16, %esp
        leave
        ret
```

Wow! The compiler saw right through us! There is no mention of the original loops, only one single call to the PUTS() function, which is all that the original code really does. To come to this conclusion, the compiler must have figured out:

- That the function CONSTANT() is really a constant - returning the same value every time. Since it uses no parameters and no variables, this one was easy to figure out.
- The variable "a" is always non-zero. This conclusion must have been made after the first one. When the CONSTANT() function was folded into a constant, all the calculations and condition-checking could have been made at compile-time rather than at run-time, and so this conclusion follows.

```
        .size   main, .-main
        .p2align 2, ,3
.globl constant
        .type   constant, @function
constant:
        pushl   %ebp
        movl    %esp, %ebp
        movl    $3, %eax
        leave
        ret
        .size   constant, .-constant
        .ident  "GCC: (GNU) 3.3.4 20040216 (prere-
lease) [FreeBSD]"
```

The ending of the MAIN() function, the actual CONSTANT() function, and the epilogue are unchanged from the previous cases.

3.5 Maximum optimisations (-O3) with CPU-specific optimisations (-march=pentium3)

The compiler is called as "GCC -O TEST_2.S -O3 -MARCH=PENTIUM3 -S TEST.C".

```
        .file   "test1.c"
        .section        .rodata.str1.1,"aMS",@progbits,1
.LC1:
        .string "second"
        .text
        .p2align 4, ,15
```

The first big difference is alignment of the code, which has changed to 16 bytes. Obviously, Pentium 3 processor are even faster when fetching from memory addresses that lie on the multiples of 16 bytes.

```
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $.LC1, (%esp)
        call    puts
        movl    %ebp, %esp
        popl    %ebp
        ret
```

Although the "payload" of the code itself hasn't changed much, the manipulation of the stack has changed significantly. The PUSH and POP instructions are used much less often and RISC-like accessing of the stack via %esp is used instead. Also, the leave instruction is RISC-ified into 2 instructions.

```
        .size   main, .-main
        .p2align 4, ,15
.globl constant
        .type   constant, @function
constant:
        pushl   %ebp
        movl    $3, %eax
        movl    %esp, %ebp
        popl    %ebp
        ret
        .size   constant, .-constant
        .ident  "GCC: (GNU) 3.3.4 20040216 (prere-
lease) [FreeBSD]"
```

The transformations in the above code snippet are similar to those in the preceding one: alignment has changed, and LEAVE instruction is resolved into 2 instructions.

4 Conclusion

Having the compiler do low-level optimisations is very handy, and today's compilers have become very smart at it. However, there are some caveats ("there's no free lunch"):

- Turning on optimisations significantly increases compilation times. Using -O2 lengthens the compilation about two times as compared to -O1, and with -O3 the compilation time sky-rockets.
- Not all compilers perform the optimisations in the same way, and some compilers are much more successful at it then others. For example, Intel's C compiler (icc) produces code that is much more optimised than the one gcc produces. A good algorithm is always much faster than a highly-optimised bad algorithm.
- As seen above, a large portion of optimisations can only be performed inside a single module. Other modules linked with the above module cannot take advantage of the knowledge that the call to the CONSTANT() function can be replaced by a numerical constant.