Loop Optimizations in Modern C Compilers

Chae Jubb
ecj2122@columbia.edu

10 December 2014

Abstract

Many programs spend a significant portion of execution time in loops. Because of this, loop optimizations are increasingly important. We see two major types of optimizations affecting loop performance: general and loop-specific. General optimizations help loop performance simply because the loop is repeated many times. Loop-specific optimizations better performance because they alter the structure of the loop or make improvements that require many iterations to be efficient due to a large overhead. We discuss loop optimization strategies and then, using directed test cases, analyze how `gcc` and `clang` use those techniques to optimize at different levels. We find `clang` to be much more aggressive in optimizations at a lower level. At the highest optimization levels, these compilers produce executables that perform similarly. Importantly, at the most common optimization level (`-O2`), `clang` equals or exceeds `gcc` performance.

1 Introduction

Loop performance is a large contributor to the overall performance of many applications. Programs, especially mathematical and scientific, spend a large majority of their execution in loops. This makes the performance especially critical. Small inefficiencies are magnified with hundreds or thousands of iterations. Because it's in the compiler's best interests to emit high-performing code, many special techniques are used specifically to increase loop performance. We examine the use of a subset of these technique and compare both emitted code and runtime performance across multiple optimization levels for multiple C compilers.

2 Method

To examine loop optimization techniques, we first prepare some sample programs. These programs are then compiled using `clang`[1] and `gcc`[2]. For each compiler, various optimization levels are examined: `-O0`, `-O1`, `-O2`, `-O3`. The binaries are generated for Intel `x86` targets: an i7 quad-core, a Xeon 16-core, and a Pentium 4.

We then analyze each output binary in two ways. First, we examine the emitted `x86` assembly code. This allows us to directly verify techniques used by the compiler in transforming the source code to the target `x86`. Second, the binary is performance-tested. Because the compiler strives to create binaries that perform well—not binaries that look as if they perform well—this portion is critical.

The disparities in runtimes of each test are drastic; thus, performance characteristics are used to evaluate the effectiveness of a different optimization level on a certain program compiled with a certain compiler. The times ought not be compared across tests.

3 Loop Optimizations

Before beginning discussion of results, we first introduce common loop optimizations. The techniques described include both machine-independent and machine-dependent optimizations. As the names suggest, the former category is used to make gen-

[1] `clang` version 3.5
[2] `gcc` version 4.8.2

```
void bad_invariant() {
  int i, a = 4;
  for(i = 0; i < 5; ++i) {
    int a2 = a * a;
    /* loop body; `a` not touched */
  }
}
```

Figure 1: Re-calculating a loop invariant each iteration

```
void good_invariant() {
  int i, a = 4;
  int a2 = a * a;
  for(i = 0; i < 5; ++i) {
    /* loop body; `a` not touched */
  }
}
```

Figure 2: Calculating a loop invariant *outside* the loop.

eral loop optimizations more concerned with the algorithm used; whereas, the latter is more concerned with the implementation and takes into account specifics of the target device.

Before beginning our examination of loop optimizations, we note that many, many optimizations will help loop performance. While not loop-specific, optimizations such as moving variables to registers from the stack will help performance, simply because of the gains of the optimization will be realized in each iteration.

3.1 Machine-Independent

We first consider optimizations made independent of the `x86` architecture. These include strategies such as identifying loop invariants, inverting the loop, and removing induction variables.

3.1.1 Loop Invariants

One simple technique used to improve the performance of loops is moving invariant calculations outside the loop. We see a sample program in Figure 1 that unnecessarily re-calculates a loop invariant on each iterations. This will cause wasted cycles, hurting performance. Luckily, many compilers will recognize that program as equivalent to the one in Figure 2, and, as such, produce the optimized code. (It will, of course, take into account the differences in scope of that loop-invariant variable.)

3.1.2 Induction Variables

Nearly all `for` loops and some `while` loops will have variables that function as a sort of loop counter. We

```
void bad_induction() {
  int i, j, *array;
  for(i = 0; i < 32; ++i) {
    int j = 4 * i;
    /* loop body, uses `i` */
    array[j] = rand();
  }
}
```

Figure 3: Inefficient redundant induction variables

label variables such as these as "induction variables". More formally, any variable whose value is altered by a fixed amount each loop iteration is an induction variable.

We can generally apply two types of optimizations to induction variables: reduction of strength and elimination. Generally, reduction of strength involves replacing an expensive operation (like multiplication) with a less expensive one (such as addition). Sometimes, however, a compiler may realize an induction variable is redundant and completely eliminate it.

Figure 3 shows an example of an inefficient use of two redundant induction variables. We improve this slightly in Figure 4 when we invoke a reduction of strength. Finally, Figure 5 shows the redundant variable completely optimized away.

3.1.3 Loop Unrolling

We next turn our attention to a simple trick sometimes employed: loop unrolling. This optimization is extremely straightforward and can only be applied to loops with a known length. Rather than having a loop with n iterations, the compiler will produce target code that simply repeats n times. This opti-

```
void ros_induction() {
  int i, j, *array;
  for(i = 0, j = -4; i < 32; ++i) {
    int j += 4;
    /* loop body, uses `i` */
    array[j] = rand();
  }
}
```

Figure 4: Reduction of strength with redundant induction variables

```
void elim_induction() {
  int i, *array;
  for(i = 0; i < 32; ++i) {
    /* loop body, uses `i` */
    array[i*4] = rand();
  }
}
```

Figure 5: Elimination of redundant induction variables

mization may increase performance on some processors because it eliminates any jump instructions. In fact, minimizing jump instructions is often the goal of many optimizations (even those not loop-specific) because that type of instruction presents the possibility of a costly branch-misprediction. We do not always use loops, though, because of a major drawback: increased binary size.

3.1.4 Loop Inversion

We now turn our attention to another optimization designed to reduce the number of branch instructions. Loop inversion is a fairly simple transformation: a `while` loop is converted to a `do-while` loop wrapped by an `if` statement as shown in Figure 6 and Figure 7.

To fully analyze the effectiveness of this optimization, we consider three cases: first iteration, any middle iteration, final iteration.

First Iteration We consider the two cases of the first iteration. Either the loop is entered or it is not. When the condition is *not* true before entering the loop, each version produces a single jump instruction.

```
void pre_inversion() {
  while(/* condition */) {
    /* loop body */
  }
}
```

Figure 6: Example `while` loop before inversion

```
void post_inversion() {
  if (/* condition */) {
    do {
      /* loop body */
    } while (/* condition */);
  }
}
```

Figure 7: Example `while` loop after inversion

We see no gain, but, more importantly, no loss in performance in this case.

When the condition is true, both execution flows behave as they would for any middle, non-final iteration.

Middle Iteration With a non-final iteration, we obviously see the same behavior between the two version. This is due to the well-known behavior of `while` and `do-while` loops.

Final Iteration By having the comparison after the loop rather than at the beginning, we can save cycles. The unoptimized version will run the last iteration, jump to the beginning to check the condition. Seeing the condition is no longer satisfied, we jump back to outside the loop.

Now we consider the optimized version. We run the last iteration and then check the loop condition. It will not be satisfied, and thus, we do not jump to the beginning and instead fall through.

This optimization results in a savings of 2 jumps. While this savings may seem trivial, consider nested loops. If this optimization were applied to an inner loop, the savings could be quite noticeable.

3.2 x86 Optimizations

We now turn our attention to the more specialized optimizations that directly target the x86 ISA's specific features. These optimizations exploit things such as memory addressing, flags, and register number and width, to name a few.

3.2.1 Use of Flags

Often a `for` loop is used to repeat an exact execution sequence a pre-determined number of times. The index variable might not be used in any way other than managing the number of iterations.

In a scenario such as this, we can take advantage of the x86 Zero Flag (ZF). This flag is a special bit that is set according to a complex set of rules that essentially amount to checking if the most recent value was equal to zero.

By taking advantage of this flag, we can eliminate an instruction in this type of loop. We replace an explicit `cmp` and a check flag instruction with the corresponding implicit `cmp` and flag-checking during the `jne` instruction.

We see an unoptimized

```
mov ecx 0x0
 <body of loop>
add ecx 0x1
cmp ecx 0x20
jb <body>
```

transformed into an optimized

```
mov ecx 0x20
 <body of loop>
sub ecx 0x1
jne <body>
```

3.2.2 SSE2 / XMM

Since the Pentium III processor, Intel has included support for Streaming SIMD Extensions. In conjunction with this, 8 128-bit processors `xmm0` through `xmm7` were added. Originally permitted to hold only 4 single-precision floating point numbers, SSE2 expanded this to support two 64-bit integers, four 32-bit integers, eight 16-bit integers, or sixteen 8-bit integers. Using these `xmm*` registers, we can now manipulate four integers at a time! This means we have the potential to cut the number of writes four-fold.

We consider this here in the loop-specific optimizations because of the overhead to set-up these registers, they would likely not be used for a one-time write to memory.

```c
#include <stdlib.h>
#include <stdint.h>

int loop_inv(uint8_t len) {
  int a[256];
  int i = 0;

  while (len < 255) {
    a[len] = 0;
    ++len;
    ++i;
  }

  return i;
}
```

Figure 8: Directed test for loop inversion

4 Test Cases

Now that we have discussed loop optimizations we wish to target, we examine the directed test cases to evaluate `gcc` and `clang` performance.

The first, simplest test case is one to explicitly check for loop inversion. The source for this can be found in Figure 8.

We also test explicitly for the handling of induction variables and loop constants using the source found in Figure 9.

The final test base test case is a program used to count the number of zeros in the binary representation of a 32-bit integer. To more fully explore how these compilers optimize, we analyze the code emitted using 5 different input programs. Source for three of these programs can be found in Figure 10 (branching version), Figure 11 (non-branching version), Figure 12 (nested for loop). The final two programs are similar except they use an infinite `for` and `while`, respectively, with an explicit `break` statement.

```c
#include <stdlib.h>
#include <stdint.h>

int *ind_var(uint16_t in) {
  int *a = malloc(sizeof *a * in * 4);
  int i, marker;
  for (i = 0; i < in; ++i) {
    marker = 4 * i;
    a[marker + 0] = in % 2;
    a[marker + 1] = in % 3;
    a[marker + 2] = in % 5;
    a[marker + 3] = in % 7;
  }

  return a;
}
```

Figure 9: Directed test for loop inversion

```c
#include <stdint.h>

int count_zeros(uint32_t a) {
  unsigned int counter = 0;
  unsigned int i = 0;
  for (; i < 32; ++i, a >>= 1) {
    if ((a & 0x00000001) == 0)
      counter++;
  }

  return counter;
}
```

Figure 10: Branching version of count zeros

```c
#include <stdint.h>

int count_zeros(uint32_t a) {
  unsigned int counter = 0;
  unsigned int i = 0;
  for (; i < 32; ++i, a >>= 1) {
    counter += !(a & 0x00000001);
  }

  return counter;
}
```

Figure 11: Non-branching version of count zeros

```c
#include <stdint.h>

int count_zeros(uint32_t a) {
  unsigned int counter = 0;
  unsigned int i, j = 0;
  for (; j < 4; ++j) {
    for (i = 0; i < 8; ++i, a >>= 1) {
      counter += !(a & 0x00000001);
    }
  }

  return counter;
}
```

Figure 12: Nested loop version of count zeros

5 Analysis

The above sample programs were compiled using each `clang` and `gcc` and each of the the previously mentioned optimization levels. We can comment on the emitted `x86` as well as the performance. Notable performance measures will be discussed below, with the full performance data available in Appendix A.

5.0.1 General Loop Format

Before continuing, we introduce the general layout that each compiler uses when it generates loops. There is not much performance effect; mostly a preference for where the jump statements ought to occur.

We see these general layouts for `gcc` and `clang` in Figure 13 and Figure 14, respectively.

Unless otherwise noted, all loops have the basic structure corresponding to the compiler that produced that loop. Most of the optimizations are simply dictating the appearance of the "body" section of the loop and the placement of the loop itself.

5.1 Loop Inversion

We begin with one of our simpler examples, the loop inversion described in Figure 8. The primary optimizations we expect to be made are loop inversion, elimination of unnecessary variables, and, finally, elimination of the loop itself.

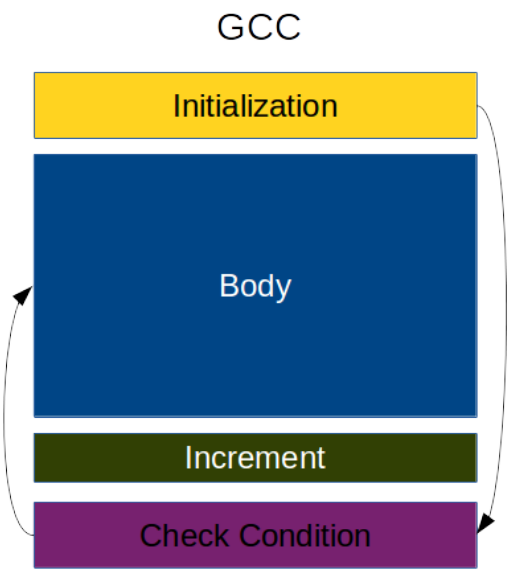

Figure 13: `gcc` loop layout

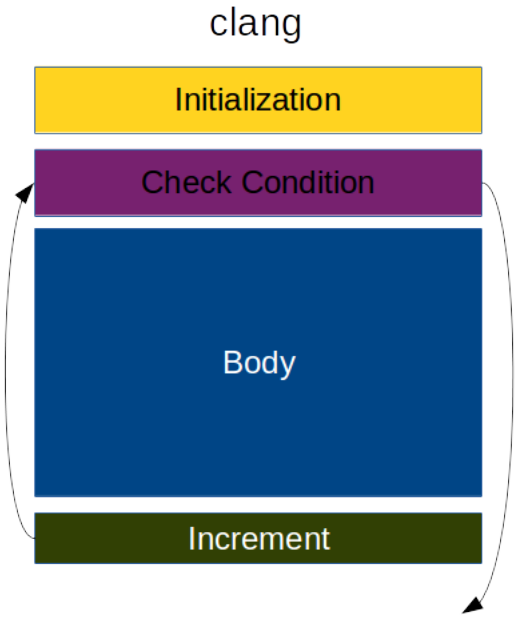

Figure 14: `clang` loop layout

```
xor     eax,eax
cmp     edi,0xff
je      4005d4 <return>
mov     al,0xfe
sub     al,dil
movzx   eax,al
inc     eax
ret
```

Figure 15: Loop Inversion: `clang` at `-O1`. Compiled from Figure 8

5.1.1 No Optimization

Without optimization, the code emitted by both compilers is as expected. We see a clear transformation of each statement into corresponding `x86`. Each compiler uses it preferred loop structure as described above.

5.1.2 Some Optimization `-O1`

Once we apply an optimization we immediately see loop inversion from `gcc`. The first two instructions are a check as to whether we should do anything or simply return 0.

```
cmp     dil,0xff
je      4005f8 <store 0 in eax and return>
```

We also see the array and the `i` variable optimized away as unnecessary. The return value is calculated indirectly using `len`.

`clang` is much more aggressive in its initial optimizations than `gcc`. All local variables as eliminated, as is the loop itself. All that remains is a few instructions to calculate `i` based on the `len` parameter. We see how compact the entire optimized output is in Figure 15.

At this point, `clang` is finished optimizing. Higher levels have no effect on the function.

5.1.3 More Optimization `-O2`

Increasing the optimization level for `gcc` eliminates the loop and produces output with only trivial differences from `clang -O1`.

5.1.4 Summary

At high optimization levels, the compilers emit code with nearly identical performance across machines. Optimizing produces up to a 30x boost in performance. Because it optimizes more aggressively, `clang` performance hits the peak sooner. However, as `-O1` is not a common optimization level, the effect of this boost may not have a large real-world effect.

5.2 Induction Variables and Loop Constants

Our attention now turns to a program designed to identify the handling of induction variables and loop constants. This program is described in Figure 9.

5.2.1 No Optimization

Compiling with `-O0` provides insight into what is guaranteed by compilers when using a flag indicating that "no optimization" ought to be performed. Statements in the C code must have a correspondence to an instruction or sequence of instructions in the emitted code. That is, statements cannot be intermixed and all statements must have some corresponding instructions that perform the statement.

While this may seem straightforward, we present a portion of the output assembly by each compiler for computing the result of the modulus operator. We will consider the line `a[marker + 3] = in % 7`.

The `clang` assembly is more transparent and expected, though it does use the infamously slow `idiv` instruction (Figure 16), retrieving the modulus computed during this instruction from `edx`. However, `gcc` uses a faster[3], more obscure algorithm. However, because there is a clear correspondence between the C statement and a block of `x86`, this is acceptable at `-O0`.

5.2.2 Some Optimization `-O1`

Adding a touch of optimization, we see a few changes in the code emitted from `gcc`. Most noticeably, we see

[3] `gcc` runs this test nearly twice as fast as `clang` without optimization

```
movzx   eax,WORD PTR [rbp-0x2]
cdq
mov     ecx, 0x7
idiv    ecx ; result: eax = eax' * ecx + edx
mov     eax,DWORD PTR [rbp-0x18] ; marker
add     eax,0x3
movsxd  r8,eax
mov     r9,QWORD PTR [rbp-0x10] ; a
mov     DWORD PTR [r9+r8*4],edx
```

Figure 16: Implementation of Modulus Operator: `clang` at `-O0`. Computing modulus by 7. Simplified from original (preserves structure)

```
mov     eax,DWORD PTR [rbp-0xc] ; marker
cdqe
add     rax,0x3
lea     rdx,[rax*4+0x0]
mov     rax,QWORD PTR [rbp-0x8] ; a
lea     rsi,[rdx+rax*1]
        ; rsi = addr of a[marker + 3]
movzx   ecx,WORD PTR [rbp-0x14] ; in
movzx   eax,cx
imul    eax,eax,0x2493 ; begin magic
shr     eax,0x10
mov     edx,ecx
sub     edx,eax
shr     dx,1
add     eax,edx
shr     ax,0x2
mov     edx,eax
mov     eax,edx
shl     eax,0x3
sub     eax,edx
sub     ecx,eax
mov     edx,ecx
movzx   eax,dx ; end magic
mov     DWORD PTR [rsi],eax ; result of % 7
```

Figure 17: Implementation of Modulus Operator: `gcc` at `-O0`. Computing modulus by 7.

```
vpinsrw xmm0,xmm0,r8d,0x0 ; in % 2
vpinsrw xmm0,xmm0,edi,0x2 ; in % 3
vpinsrw xmm0,xmm0,ecx,0x4 ; in % 5
vpinsrw xmm0,xmm0,esi,0x6 ; in % 7
xor     ecx,ecx
mov     rdx,rax
; excluding NOPs for alignment of loop
vmovdqu XMMWORD PTR [rdx],xmm0
add     rdx,0x10
inc     ecx
cmp     ecx,ebx
jl      400740 ; vmovdqu command
```

Figure 18: Implementation of Modulus Operator: `clang` at `-O0`. Computing modulus by 7. Simplified from original (preserves structure)

loop inversion. Additionally, the redundant induction variable `marker` is removed. This is replaced with direct 4x multiplication in the memory addressing. The implementation of this optimization is machine-dependent and is permitted here because that multiplication by constant in this way is a permitted `x86` addressing mode.[4] However, the loop constants are still computed each time, greatly hindering performance gains.

With our other compiler, `clang`, we see another aggressive optimization. Along with loop inversion and the elimination of an induction variable, we see use of the architecture-specific `xmm*` registers. As mentioned previously, these registers permit us to write a block of 4 integers in one write. Because of the novelty and efficiency of this algorithm, we include a portion of the emitted `x86` in Figure 18.

Using these specialized registers show a huge performance bonus: about 2x over usual registers. All optimizations considered, `O1` gives about a 25x speedup over `O0`.

5.2.3 Use of `xmm` registers by `clang`

We saw above that `xmm` registers were used easily because we were writing 4 entries at a time. We naturally wonder the effects of writing some non-multiple of 4 in each loop iteration.

Less than Eight Entries When we write five entries in each loop iteration, we see `clang` using the `xmm` registers along with one extended register `ecx`. We do not revert to solely using extended width registers because we do not have a multiple of 4.

Eight Entries Expectations are not met when loading eight entries simultaneously. Only one `xmm` register is used in conjunction with four extended width registers. In an attempt to determine if this was a conscious decision due to some performance hit caused by using multiple `xmm` interchangeably, we hand-tune the assembly to use both `xmm0` and `xmm1` in conjunction with two `vmovdqu` commands.[5] By using two `xmm` registers rather than one, we notice a nearly 30 percent performance improvement. We are left only to wonder why does not use multiple specialized registers.

5.2.4 More Optimization `-O2` and `-O3`

`clang` makes no optimizations after the `-O1` level.

On the other hand, `gcc` continues making optimizations. `O2` sees loop invariants calculated outside the loop. That is, all moduli operations are done before entering the loop and repeated assigned during iteration.

Finally at `O3`, we see `gcc` also using the `xmm` registers. It is perhaps worth noting that with 8 writes per loop, `gcc` does in fact use multiple `xmm` registers to achieve the same performance as the above hand-tuned `x86` (though in a more complex way).

5.2.5 Architecture Differences

If an architecture does not have support for `xmm` registers holding integers, obviously they cannot be used. The tests run on a Pentium 4 processor do not compile to `xmm` registers. For machines such as these, we do not see speedups comparable to those with `xmm` registers. We see only a 5x speedup compared to a 25x speedup (`clang` `O0` to `O1`).

[4] Other ISAs such as MIPS do not permit this. On a MIPS architecture, we would likely see reduction of strength: the multiplication converted to a constant addition during each iteration.

[5] The relevant code snippets are available in Appendix B

This difference underscores the importance of the target architecture in performance. Without advanced hardware, compilers cannot use advanced optimization techniques.

5.2.6 Summary

Again we see `clang` making more aggressive optimizations than `gcc`. Interestingly, `clang` performs much more poorly with no optimization: nearly twice as slow as its `gcc` counterpart, because of the choice of modulus implementation.

At maximum optimization, we see nearly identically performing code, though at the commonly used `O2`, we see `clang` as a clear winner, performing approximately twice as fast as the corresponding `gcc` binary.

5.3 Constant-Length Loops

Our final analysis considers the programs designed to count the number of zeros in the binary representation of a 32-bit integer as seen in Figure 10, Figure 11, and Figure 12. In addition to these samples, versions with an infinite `for` and `while` loop in conjunction with an explicit break statement were examined. By analyzing these different versions of the program, we can gather some sense of how much "error-correcting" the compiler can do on non-optimal code.

5.3.1 No Optimization

All versions of the programs emit assembly as expected according to the compiler's preferred loop structure. The performance of the samples was generally homogeneous—with the strong exception of the branching sample. The branching version took 2–3 times as long as the others. This serves as a clear indicator that branching in tight loops ought to be avoided as much as possible.

5.3.2 Some Optimization `-O1`

We first note that both compilers immediately recognize the infinite `for` and infinite `while` loops as being equivalent to the non-branching version with a finite `for` loop.

The most obvious optimization applied to all versions with both compilers is the conversion of the incrementing `for` loop into a decrementing one. The compiler identifies the static number of loop iterations and takes advantage of the Zero Flag to save an instruction, as described previously. We note a slight difference in the handling of the nested `for` loop between compilers. `gcc` converts *both* loops to the decrementing style, while `clang` is only able to optimize the outer loop in this way. The inner loop remains an incrementing loop.

While not a loop-specific optimization, we see the use of the stack is completely eliminated. All temporary values are stored in registers.

Branching Case The optimization of the branching case (again Figure 10) is much more compiler-dependent. We see `gcc` use an add with carry instruction as follows:

```
and     ecx,0x1
cmp     ecx,0x1
adc     eax,0x0
```

While this version is certainly an improvement over the `-O0` version, it's performance relative to the optimized version of the non-branching sample greatly varies across architectures.

We see `clang` perform comparatively better than `gcc`. `clang` recognizes the branching sample as functionally identical to the non-branching algorithm and emits the same exact target code for the two samples! In fact all but the nested `for` samples have the same exactly target code when compiled with `clang`.

5.3.3 More Optimization with `gcc`: `-O2` `-O3`

The only further loop-based optimization we see by advancing to `-O2` is a slight loop reorganization that has little performance effect. In fact, performance is actually reduced on some architectures.[6] Because we see the difference across all `gcc` samples, it is likely due to some change in the consistent loop body.

Further optimizing at the `-O3` level produces two changes. First, the nested `for` example now has a

[6] We recall that compilers will optimize for the "average" machine.

completely unrolled inner loop. This eliminates a few jump instructions, which is good for pipelined performance. Second, the branching version now uses a conditional move:

```
lea    ecx,[rax+0x1]
test   dil,0x1
cmove  eax,ecx
```

This has little effect on two of three architectures tested; the program slows by approximately 45 percent on the third (Pentium 4).

5.3.4 More Optimization with `clang`: `-O2`

The only further optimization made at the `-O2` level is loop unrolling. `clang` completely unrolls the nested `for` loop program. We now have 32 repetitions of a `shr`, `not`, `and`, `add` cycle.

5.3.5 Summary

For this third example, we see `clang` making the clever realization that the branching and non-branching samples are *functionally* equivalent. This allows us to remove conditional statements, which allow the pipeline to flow more naturally. A more naturally flowing pipeline—that is, one will fewer stalls mis-predictions—will give better performance.

Loop unrolling takes advantage of this fact: by eliminating conditional jump statements, we cannot have mis-predictions! It is not surprising, then, that the samples which were ultimately unrolled have the best performance across architectures. It is surprising, however, that this comes from the (arguably) worst-written sample. Both branching and non-branching samples are reasonable implementations of the algorithm. Ostensibly, an unnecessary nested `for` loop is exactly that: unnecessary. However, we see 30–40 percent performance boosts in comparison to non-unrolled loops.

We should not, however, take this result and use it as an endorsement of unnecessary and cluttering control constructs. They serve only to obfuscate the purpose of the code and when compilers begin to recognize the ability to optimize, there will be no advantage. If performance is that high a consideration, inline assembly is likely the best solution.

6 Conclusions

In line with Amdahl's Law, improving the performance of loops will likely improve the performance of the entire program as most of the execution time of many programs is spent inside loops. Modern C compilers such as `gcc` and `clang` provide many loop-specific optimizations that can better performance metrics.

With these machine-independent and machine-dependent optimizations available, the compiler is able to take quite inefficient programs and increase performance by orders or magnitude.

Both `gcc` and `clang` use standard loop optimizations such as loop inversion, reduction of induction variables, extracting loop constants, and loop unrolling. They seem to also have a nearly identical set of `x86` optimizationsAs these two compilers are dominating the market and competing with each other, it is only natural that they mirror each other's progress.

However, we see obvious potential optimizations that neither compiler is able to make. Nested `for` loops are sometimes converted to a single loop via loop unrolling. Neither compiler, though, was able to recognize that the nesting was functionally unnecessary and could be reduced to an equivalent, still-rolled loop. Unused variables are optimized away, why not superfluous control structures?

If generalizations about the loop performance of compilers could be made, it would be this: `clang` seems to produce better target code at lower optimization levels, but the two compilers produce very similarly performing code at the highest optimization levels. Considering pure, unoptimized code, in cases with a clear performance differential, `gcc` seems to outperform `clang`.

We note, though, that many shun the highest levels of optimization because it often creates large binaries. Similarly, many avoid using low levels of optimization because it doesn't do enough. Perhaps the most common optimization level is `-O2`, where compilers often strike a pleasant balance between speed and size. With this level of optimization on the modern systems tested with loop-dominant programs, `clang` executable performance is equal to if not better than `gcc` binary performance. ■

A Performance Evaluation

Below we see the median execution times over 20 tests. The magnitudes of the times themselves are largely irrelevant *between different tests* but are help equivalent across all of the *same* tests over different optimization levels, compilers, and machines. The number of iterations was chosen to provide a reasonable runtime. We see a table for each compiler on each machine.

Performance for i7 Quad core can be found in Table 1 and Table 2 for `gcc` and `clang`, respectively. Performance for Xeon 16-core can be found in Table 3 and Table 4 for `gcc` and `clang`, respectively. Performance for Pentium 4 can be found in Table 5 and Table 6 for `gcc` and `clang`, respectively.

`gcc`	O0	O1	O2	O3
Basic	1.535250	0.740309	0.750498	0.744492
Branch	4.331550	0.745291	0.738352	0.745730
Infinite for	1.756510	0.743746	0.752588	0.741769
Infinite while	1.743800	0.740280	0.751422	0.739939
Nested for	1.595010	0.623453	0.440711	0.437040
Induction variable	5.004260	0.203298	0.203790	0.192637
Loop inversion	6.033830	0.182433	0.187445	0.187577

Table 1: `gcc`, Intel i7 Quad Core, times in seconds

`clang`	O0	O1	O2	O3
Basic	1.620	0.640	0.640	0.640
Branch	3.640	0.640	0.640	0.640
Infinite for	1.670	0.640	0.640	0.640
Infinite while	1.670	0.640	0.640	0.640
Nested for	1.670	0.720	0.460	0.460
Induction variable	2.320	0.410	0.410	0.410
Loop inversion	4.050	0.180	0.180	0.180

Table 2: `clang`, Intel i7 Quad Core, times in seconds

gcc	O0	O1	O2	O3
Basic	1.565	0.890	0.740	0.740
Branch	3.460	0.720	0.750	0.740
Infinite for	1.580	0.890	0.740	0.740
Infinite while	1.580	0.890	0.740	0.740
Nested for	1.630	0.820	0.800	0.460
Induction variable	3.380	0.410	0.410	0.220
Loop inversion	4.590	1.760	0.170	0.170

Table 3: `gcc`, Intel Xeon 16 core, times in seconds

clang	O0	O1	O2	O3
Basic	1.620	0.640	0.640	0.640
Branch	3.640	0.640	0.640	0.640
Infinite for	1.670	0.640	0.640	0.640
Infinite while	1.670	0.640	0.640	0.640
Nested for	1.670	0.720	0.460	0.460
Induction variable	2.320	0.410	0.410	0.410
Loop inversion	4.050	0.180	0.180	0.180

Table 4: `clang`, Intel Xeon 16 core, times in seconds

gcc	O0	O1	O2	O3
Basic	4.015420	1.484550	1.490820	1.491320
Branch	7.169850	1.814280	1.831730	2.643850
Infinite for	4.386650	1.481490	1.489430	1.491940
Infinite while	4.376770	1.479890	1.487900	1.490690
Nested for	4.804280	1.605150	1.717000	1.188230
Induction variable	9.997740	3.948260	4.112130	4.387550
Loop inversion	11.033000	2.070470	0.688073	0.688639

Table 5: `gcc`, Intel Pentium 4, times in seconds

clang	O0	O1	O2	O3
Basic	3.377270	1.654560	1.654590	1.654020
Branch	6.779970	1.651640	1.652980	1.654070
Infinite for	3.882610	1.652520	1.654050	1.652970
Infinite while	3.876050	1.651210	1.653400	1.652260
Nested for	3.626910	1.719660	0.993197	0.997602
Induction variable	16.628900	3.65794	3.833720	4.08955
Loop inversion	12.505200	0.679849	0.676784	0.680679

Table 6: `clang`, Intel Pentium 4, times in seconds

B SSE2 Usage

Below we show excerpts from the `x86` using `xmm` registers. Figure 20 shows what `clang` emits. Figure 22 shows the hand-tuned `x86` based on `clang` output. Figured 21 shows what `gcc` emits. We also present the modified source code in Figure 19 (modified from Figure 9).

```
#include <stdlib.h>
#include <stdint.h>

int *ind_var(uint16_t in) {
  int *a = malloc(sizeof *a * in * 8);
  int i, marker;
  for (i = 0; i < in; ++i) {
    marker = 8 * i;
    a[marker + 0] = in % 2;
    a[marker + 1] = in % 3;
    a[marker + 2] = in % 5;
    a[marker + 3] = in % 7;
    a[marker + 4] = in % 11;
    a[marker + 5] = in % 13;
    a[marker + 6] = in % 17;
    a[marker + 7] = in % 19;
  }

  return a;
}
```

Figure 19: Alteration of Figure 9 with 8 sequential writes per loop.

```
; prepare r8d, esi, ebx, edx
vpinsrw xmm0,xmm0,r8d,0x0 ; in % 2
vpinsrw xmm0,xmm0,esi,0x2 ; in % 3
vpinsrw xmm0,xmm0,ebx,0x4 ; in % 5
vpinsrw xmm0,xmm0,edx,0x6 ; in % 7
; prepare r8d, edx, esi, edi
vmovdqu XMMWORD PTR [rbx-0x1c],xmm0
mov     DWORD PTR [rbx-0xc],r8d
mov     DWORD PTR [rbx-0x8],edx
mov     DWORD PTR [rbx-0x4],esi
mov     DWORD PTR [rbx],edi
add     rbx,0x20
inc     ecx
cmp     ecx,r14d
jl      400740 ; vmovdqu
```

Figure 20: Selections from `clang` version of induction variable C code from Figure 19.

```
; prepare edx
vmovd  xmm1,edx ; in % 2
; prepare esi
vmovd  xmm0,esi
; prepare esi, edi
vpinsrd xmm0,xmm0,esi,0x1
xor    esi,esi
vpinsrd xmm1,xmm1,edi,0x1
vpunpcklqdq xmm2,xmm1,xmm0
vinsertf128 ymm2,ymm2,xmm2,0x1
; beginning of main loop
mov    rdi,rsi
add    rsi,0x1
shl    rdi,0x5
cmp    ecx,esi
vmovdqu XMMWORD PTR [rax+rdi*1],xmm2
vextractf128 XMMWORD PTR [rax+rdi*1+0x10],ymm2,0x1
ja     400776 ; beginning of main loop
cmp    edx,r8d
mov    ecx,r8d
je     4007d0 ; exit_final
vzeroupper
; shift_unpack_leave:
shl    ecx,0x2
vpunpcklqdq xmm0,xmm1,xmm0
movsxd rcx,ecx
vmovdqu XMMWORD PTR [rax+rcx*4],xmm0
mov    rbx,QWORD PTR [rbp-0x8]
leave
ret
; NOPs for alignment
xor    ecx,ecx
vpinsrd xmm0,xmm0,esi,0x1
vpinsrd xmm1,xmm1,edi,0x1
jmp    40079d ; shift_unpack_leave
; NOPs for alignment
; exit_final:
vzeroupper
mov    rbx,QWORD PTR [rbp-0x8]
leave
ret
```

Figure 21: Selections from gcc version of induction variable C code from Figure 19.

```
; prepare r8d, esi, ebx, edx
vpinsrw xmm0,r8d,0x0 ; in % 2
vpinsrw xmm0,esi,0x2 ; in % 3
vpinsrw xmm0,ebx,0x4 ; in % 5
vpinsrw xmm0,edx,0x6 ; in % 7
; prepare r8d, edx, esi, sdi
vpinsrw xmm1,r8d,0x0 ; in % 11
vpinsrw xmm1,edx,0x2 ; in % 13
vpinsrw xmm1,esi,0x4 ; in % 17
vpinsrw xmm1,edi,0x6 ; in % 19
; NOPs for alignment
vmovdqu XMMWORD PTR [rbx-0x1c],xmm0
vmovdqu XMMWORD PTR [rbx-0xc],xmm1
add     rbx,0x20
inc     ecx
cmp     ecx,r14d
jl      4007b0 ; first movdqu
```

Figure 22: Selections from Hand-optimized version of induction variable C code from Figure 19.