The wrong way of benchmarking the most efficient integer comparison function

devblogs.microsoft.com/oldnewthing/20171117-00

November 17, 2017

Raymond Chen

On StackOverflow, there's a question about the most efficient way to compare two integers and produce a result suitable for a comparison function, where a negative value means that the first value is smaller than the second, a positive value means that the first value is greater than the second, and zero means that they are equal.

There was much microbenchmarking of various options, ranging from the straightforward

```
int compare1(int a, int b)
{
    if (a < b) return -1;
    if (a > b) return 1;
    return 0;
}
```

to the clever

```
int compare2(int a, int b)
{
    return (a > b) - (a < b);
}
```

to the hybrid

```
int compare3(int a, int b)
{
    return (a < b) ? -1 : (a > b);
}
```

to inline assembly

```
int compare4(int a, int b)
{
    __asm__ __volatile__ (
        "sub %1, %0 \n\t"
        "jno 1f \n\t"
        "cmc \n\t"
        "rcr %0 \n\t"
        "1: "
    : "+r"(a)
    : "r"(b)
    : "cc");
    return a;
}
```

The benchmark pitted the comparison functions against each other by comparing random pairs of numbers and adding up the results to prevent the code from being optimized out.

But here's the thing: Adding up the results is completely unrealistic.

There are no meaningful semantics that could be applied to a sum of numbers for which only the sign is significant. No program that uses a comparison function will add the results. The only thing you can do with the result is compare it against zero and take one of three actions based on the sign.

Adding up all the results means that you're not using the function in a realistic way, which means that your benchmark isn't realistic.

Let's try to fix that. Here's my alternative test:

```
// Looks for "key" in sorted range [first, last) using the
// specified comparison function. Returns iterator to found item,
// or last if not found.

template<typename It, typename T, typename Comp>
It binarySearch(It first, It last, const T& key, Comp compare)
{
 // invariant: if key exists, it is in the range [first, first+length)
 // This binary search avoids the integer overflow problem
 // by operating on lengths rather than ranges.
 auto length = last - first;
 while (length > 0) {
  auto step = length / 2;
  It it = first + step;
  auto result = compare(*it, key);
  if (result < 0) {
   first = it + 1;
   length -= step + 1;
  } else if (result == 0) {
   return it;
  } else {
   length = step;
  }
 }
 return last;
}

int main(int argc, char **argv)
{
 // initialize the array with sorted even numbers
 int a[8192];
 for (int i = 0; i < 8192; i++) a[i] = i * 2;

 for (int iterations = 0; iterations < 1000; iterations++) {
  int correct = 0;
  for (int j = -1; j < 16383; j++) {
   auto it = binarySearch(a, a+8192, j, COMPARE);
   if (j < 0 || j > 16382 || j % 2) correct += it == a+8192;
   else correct += it == a + (j / 2);
  }
  // if correct != 16384, then we have a bug somewhere
  if (correct != 16384) return 1;
 }
 return 0;
}
```

Let's look at the code generation for the various comparison functions. I used gcc.godbolt.org with x86-64 gcc 7.2 and optimization `-O3`.

If we try `compare1`, then the binary search looks like this:

```
    ; on entry, esi is the value to search for

    lea rdi, [rsp-120]          ; rdi = first
    mov edx, 8192               ; edx = length
    jmp .L9
.L25:                           ; was greater than
    mov rdx, rax                ; length = step
    test rdx, rdx               ; while (length > 0)
    jle .L19
.L9:
    mov rax, rdx                ;
    sar rax, 1                  ; eax = step = length / 2
    lea rcx, [rdi+rax*4]        ; it = first + step

    ; result = compare(*it, key), and then test the result
    cmp dword ptr [rcx], esi    ; compare(*it, key)
    jl .L11                     ; if less than
    jne .L25                    ; if not equal (therefore if greater than)
    ... return value in rcx     ; if equal, answer is in rcx

.L11:                           ; was less than
    add rax, 1                  ; step + 1
    lea rdi, [rcx+4]            ; first = it + 1
    sub rdx, rax                ; length -= step + 1
    test rdx, rdx               ; while (length > 0)
    jg .L9
.L19:
    lea rcx, [rsp+32648]        ; rcx = last
    ... return value in rcx
```

Exercise: Why is `rsp - 120` the start of the array?

Observe that despite using the lamest, least-optimized comparison function, we got the comparison-and-test code that is much what we would have written if we had done it in assembly language ourselves: We compare the two values, and then follow up with two branches based on the same shared flags. The comparison is still there, but the calculation and testing of the return value are gone.

In other words, not only was `compare1` optimized down to one `cmp` instruction, but it also managed to delete instructions from the `binarySearch` function too. It had a net cost of negative instructions!

What happened here? How did the compiler manage to optimize out all our code and leave us with the shortest possible assembly language equivalent?

Simple: First, the compiler did some constant propagation. After inlining the `compare1` function, the compiler saw this:

```
    int result;
    if (*it < key) result = -1;
    else if (*it > key) result = 1;
    else result = 0;
    if (result < 0) {
      ... less than ...
    } else if (result == 0) {
      ... equal to ...
    } else {
      ... greater than ...
    }
```

The compiler realized that it already knew whether constants were greater than, less than, or equal to zero, so it could remove the test against `result` and jump straight to the answer:

```
    int result;
    if (*it < key) { result = -1; goto less_than; }
    else if (*it > key) { result = 1; goto greater_than; }
    else { result = 0; goto equal_to; }
    if (result < 0) {
less_than:
      ... less than ...
    } else if (result == 0) {
equal_to:
      ... equal to ...
    } else {
greater_than:
      ... greater than ...
    }
```

And then it saw that all of the tests against `result` were unreachable code, so it deleted them.

```
    int result;
    if (*it < key) { result = -1; goto less_than; }
    else if (*it > key) { result = 1; goto greater_than; }
    else { result = 0; goto equal_to; }

less_than:
      ... less than ...
      goto done;

equal_to:
      ... equal to ...
      goto done;

greater_than:
      ... greater than ...
done:
```

That then left `result` as a write-only variable, so it too could be deleted:

```
    if (*it < key) { goto less_than; }
    else if (*it > key) { goto greater_than; }
    else { goto equal_to; }

less_than:
      ... less than ...
      goto done;

equal_to:
      ... equal to ...
      goto done;

greater_than:
      ... greater than ...
done:
```

Which is equivalent to the code we wanted all along:

```
    if (*it < key) {
      ... less than ...
    } else if (*it > key) {
      ... greater than ...
    } else {
      ... equal to ...
    }
```

The last optimization is realizing that the test in the `else if` could use the flags left over by the `if`, so all that was left was the conditional jump.

Some very straightforward optimizations took our very unoptimized (but easy-to-analyze) code and turned it into something much more efficient.

On the other hand, let's look at what happens with, say, the second comparison function:

```
    ; on entry, edi is the value to search for

    lea r9, [rsp-120]           ; r9 = first
    mov ecx, 8192               ; ecx = length
    jmp .L9
.L11:                           ;
    test eax, eax               ; result == 0?
    je .L10                     ; Y: found it
                                ; was greater than
    mov rcx, rdx                ; length = step
    test rcx, rcx               ; while (length > 0)
    jle .L19
.L9:
    mov rdx, rcx
    xor eax, eax                ; return value of compare2
    sar rdx, 1                  ; rdx = step = length / 2
    lea r8, [r9+rdx*4]          ; it = first + step

    ; result = compare(*it, key), and then test the result
    mov esi, dword ptr [r8]     ; esi = *it
    cmp esi, edi                ; compare *it with key
    setl sil                    ; sil = 1 if less than
    setg al                     ; al  = 1 if greater than
                                ; eax = 1 if greater than
    movzx esi, sil              ; esi = 1 if less than
    sub eax, esi                ; result = (greater than) - (less than)
    cmp eax, -1                 ; less than zero?
    jne .L11                    ; N: Try zero or positive

                                ; was less than
    add rdx, 1                  ; step + 1
    lea r9, [r8+4]              ; first = it + 1
    sub rcx, rdx                ; length -= step + 1
    test rcx, rcx               ; while (length > 0)
    jg .L9
.L19:
    lea r8, [rsp+32648]         ; r8 = last
.L10:
    ... return value in r8
```

The second comparison function `compare2` uses the relational comparison operators to generate exactly 0 or 1. This is a clever way of generating `-1`, `0`, or `+1`, but unfortunately, that was not our goal in the grand scheme of things. It was merely a step toward that goal. The way that `compare2` calculates the result is too complicated for the optimizer to understand, so it just does its best at calculating the formal return value from `compare2` and testing its sign. (The compiler does realize that the only possible negative value is `-1`, but that's not enough insight to let it optimize the entire expression away.)

If we try `compare3`, we get this:

```
    ; on entry, esi is the value to search for

    lea rdi, [rsp-120]          ; rdi = first
    mov ecx, 8192               ; ecx = length
    jmp .L12
.L28:                           ; was greater than
    mov rcx, rax                ; length = step
.L12:
    mov rax, rcx
    sar rax, 1                  ; rax = step = length / 2
    lea rdx, [rdi+rax*4]        ; it = first + step

    ; result = compare(*it, key), and then test the result
    cmp dword ptr [rdx], esi    ; compare(*it, key)
    jl .L14                     ; if less than
    jle .L13                    ; if less than or equal (therefore equal)

    ; "length" is in eax now
.L15:                           ; was greater than
    test eax, eax               ; length == 0?
    jg .L28                     ; N: continue looping
    lea rdx, [rsp+32648]        ; rdx = last
.L13:
    ... return value in rdx

.L14:                           ; was less than
    add rax, 1                  ; step + 1
    lea rdi, [rdx+4]            ; first = it + 1
    sub rcx, rax                ; length -= step + 1
    mov rax, rcx                ; rax = length
    jmp .L15
```

The compiler was able to understand this version of the comparison function: It observed that if `a < b`, then the result of `compare3` is always negative, so it jumped straight to the less-than case. Otherwise, it observed that the result was zero if `a` is not greater than `b` and jumped straight to that case too. The compiler did have some room for improvement with the placement of the basic blocks, since there is an unconditional jump in the inner loop, but overall it did a pretty good job.

The last case is the inline assembly with `compare4`. As you might expect, the compiler had the most trouble with this.

```
    ; on entry, edi is the value to search for

    lea r8, [rsp-120]           ; r8 = first
    mov ecx, 8192               ; ecx = length
    jmp .L12
.L14:                           ; zero or positive
    je .L13                     ; zero - done
                                ; was greater than
    mov rcx, rdx                ; length = step
    test rcx, rcx               ; while (length > 0)
    jle .L22
.L12:
    mov rdx, rcx
    sar rdx, 1                  ; rdx = step = length / 2
    lea rsi, [r8+rdx*4]         ; it = first + step

    ; result = compare(*it, key), and then test the result
    mov eax, dword ptr [rsi]    ; eax = *it
    sub eax, edi
    jno 1f
    cmc
    rcr eax, 1
1:
    test eax, eax               ; less than zero?
    jne .L14                    ; N: Try zero or positive

                                ; was less than
    add rdx, 1                  ; step + 1
    lea r8, [rsi+4]             ; first = it + 1
    sub rcx, rdx                ; length -= step + 1
    test rcx, rcx               ; while (length > 0)
    jg .L12
.L22:
    lea rsi, [rsp+32648]        ; rsi = last
.L13:
    ... return value in rsi
```

This is pretty much the same as `compare2`: The compiler has no insight at all into the inline assembly, so it just dumps it into the code like a black box, and then once control exits the black box, it checks the sign in a fairly efficient way. But it had no real optimization opportunities because you can't really optimize inline assembly.

The conclusion of all this is that optimizing the instruction count in your finely-tuned comparison function is a fun little exercise, but it doesn't necessarily translate into real-world improvements. In our case, we focused on optimizing the code that encodes the result of the comparison without regard for how the caller is going to decode that result. The contract between the two functions is that one function needs to package some result, and the other function needs to unpack it. But we discovered that the more obtusely we wrote the code for the packing side, the less likely the compiler would be able to see how to optimize out the

entire hassle of packing and unpacking in the first place. In the specific case of comparison functions, it means that you may want to return `+1`, `0`, and `-1` explicitly rather than calculating those values in a fancy way, because it turns out compilers are really good at optimizing "compare a constant with zero".

You have to see how your attempted optimizations fit into the bigger picture because you may have hyper-optimized one part of the solution to the point that it prevents deeper optimizations in other parts of the solution.

Bonus chatter: If the comparison function is not inlined, then all of these optimization opportunities disappear. But I personally wouldn't worry about it too much, because if the comparison function is not inlined, then the entire operation is going to be dominated by the function call overhead: Setting up the registers for the call, making the call, returning from the call, testing the result, and most importantly, the lost register optimization opportunities not only because the compiler loses opportunities to enregister values across the call, but also because the compiler has to protect against the possibility that the comparison function will mutate global state and consequently create aliasing issues.

Raymond Chen

Follow