# Optimization is often counter-intuitive

devblogs.microsoft.com/oldnewthing/20041216-00

December 16, 2004

Raymond Chen

Anybody who's done intensive optimization knows that optimization is often counter-intuitive. Things you think would be faster often aren't.

Consider, for example, the exercise of obtaining the current instruction pointer. There's the naïve solution:

```
__declspec(noinline)
void *GetCurrentAddress()
{
  return _ReturnAddress();
}

...
void *currentInstruction = GetCurrentAddress();
```

If you look at the disassembly, you'll get something like this:

```
GetCurrentAddress:
    mov eax, [esp]
    ret

...
    call GetCurrentAddress
    mov [currentInstruction], eax
```

"Pah," you say to yourself, "look at how inefficient that is. I can reduce that to two instructions. Watch:

```
void *currentInstruction;
__asm {
call L1
L1: pop currentInstruction
}
```

That's half the instruction count of your bloated girly-code."

But if you sit down and race the two code sequences, you'll find that the function-call version is faster by a factor of two! How can that be?

The reason is the "hidden variables" inside the processor. All modern processors contain much more state than you can see from the instruction sequence. There are TLBs, L1 and L2 caches, all sorts of stuff that you can't see. The hidden variable that is important here is the return address predictor.

The more recent Pentium (and I believe also Athlon) processors maintain an internal stack that is updated by each `CALL` and `RET` instruction. When a `CALL` is executed, the return address is pushed both onto the real stack (the one that the `ESP` register points to) as well as to the internal return address predictor stack; a `RET` instruction pops the top address of the return address predictor stack as well as the real stack.

The return address predictor stack is used when the processor decodes a `RET` instruction. It looks at the top of the return address predictor stack and says, "I bet that `RET` instruction is going to return to that address." It then speculatively executes the instructions at that address. Since programs rarely fiddle with return addresses on the stack, these predictions tend to be highly accurate.

That's why the "optimization" turns out to be slower. Let's say that at the point of the `CALL L1` instruction, the return address predictor stack looks like this:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Return address predictor stack:** | caller1 | → | caller2 | → | caller3 | → | ⋯ |
| **Actual stack:** | caller1 | → | caller2 | → | caller3 | → | ⋯ |

Here, `caller1` is the function's caller, `caller1` is the function's caller's caller, and so on. So far, the return address predictor stack is right on target. (I've drawn the actual stack below the return address predictor stack so you can see that they match.)

Now you execute the `CALL` instruction. The return address predictor stack and the actual stack now look like this:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Return address predictor stack:** | L1 | → | caller1 | → | caller2 | → | caller3 | → | ⋯ |
| **Actual stack:** | L1 | → | caller1 | → | caller2 | → | caller3 | → | ⋯ |

But instead of executing a `RET` instruction, you pop off the return address. This removes it from the actual stack, but doesn't remove it from the return address predictor stack.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Return address predictor stack:** | L1 | → | caller1 | → | caller2 | → | caller3 | → | ⋯ |
| **Actual stack:** | caller1 | → | caller2 | → | caller3 | → | caller4 | → | ⋯ |

I think you can see where this is going.

Eventually your function returns. The processor decodes your `RET` instruction and looks at the return address predictor stack and says, "My predictor stack says that this `RET` is going to return to `L1`. I will begin speculatively executing there."

But oh no, the value on the top of the real stack isn't `L1` at all. It's `caller1`. The processor's return address predictor predicted incorrectly, and it ended up wasting its time studying the wrong code!

The effects of this bad guess don't end there. After the `RET` instruction, the return address predictor stack looks like this:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **Return address predictor stack:** | caller1 | → | caller2 | → | caller3 | → | ⋯ |
| **Actual stack:** | caller2 | → | caller3 | → | caller4 | → | ⋯ |

Eventually your caller returns. Again, the processor consults its return address predictor stack and speculatively executes at `caller1`. But that's not where you're returning to. You're really returning to `caller2`.

And so on. By mismatching the `CALL` and `RET` instructions, you managed to cause every single return address prediction on the stack to be wrong. Notice in the diagram that, in the absence of somebody playing games with the return address predictor stack of the type that created the problem initially, **not a single prediction on the return address predictor stack will be correct**. None of the predicted return addresses match up with actual return addresses.

Your peephole optimization has proven to be shortsighted.

Some processors expose this predictor more explicitly. The Alpha AXP, for example, has several types of control flow instructions, all of which have the same logical effect, but which hint to the processor how it should maintain its internal predictor stack. For example, the `BR` instruction says, "Jump to this address, but do not push the old address onto the predictor stack." On the other hand, the `JSR` instruction says, "Jump to this address, and push the old address onto the predictor stack." There is also a `RET` instruction that says, "Jump to this address, and pop an address from the predictor stack." (There's also a fourth type that isn't used much.)

Moral of the story: Just because something looks better doesn't mean that it necessarily **is** better.

Raymond Chen

**Follow**