Why does the compiler generate memory operations on the full variable even though only one byte is involved?

devblogs.microsoft.com/oldnewthing/20170428-00

April 28, 2017

Raymond Chen

Some time ago, I was helping out with code generation in a just-in-time compiler, and one thing I noticed was that when the compiler needed to, say, set the top bit in a four-byte variable, it did this:

```
    or dword ptr [variable], 80000000h
```

instead of the more compact

```
    or byte ptr [variable+3], 80h
```

The two operations are functionally equivalent: Setting the top bit in a four-byte value is the same as setting the top bit in a one-byte value, because the lower bits are unaffected by the operation.

I knew there was a good reason for this because the person who originally wrote the compiler has decades of experience in this sort of thing, and this type of obvious optimization would not have been passed up.

The answer is another of the hidden variables inside the CPU, this one called the *store buffer*, which is used in a process called *store-to-load forwarding*. You can read more about the topic here, but the short version is that when speculative execution encounters a write to memory (a *store* operation), it cannot write the memory immediately because it is merely speculating. Instead, it writes to a *store buffer* which remembers, “If we ultimately end up realizing this speculation, we need to write the value *V* to the address *A*.”

When a memory read operation (a *load*) is speculated, it first checks the store buffer to see whether there is any speculated write to to that address, and if so, it uses that speculated value instead the actual value in memory. This step in the speculation process is known as *store-to-load forwarding*.

Of course, life is not as easy as it appears because there are many ways you could have modified the memory at the address *A*, thanks to the fact that the x86 permits both sub-word memory access as well as misaligned memory access. Misaligned memory access means that if you want to read a four-byte value from *A*, you have to look not only for four-byte writes to *A*, but also four-byte writes in the range *A* − 3 through *A* + 3, because those overlap the memory you are about to read. And sub-word memory access means that you also have to look for one-byte writes in the range *A* through *A* + 3, as well as two-byte writes in the range *A* − 1 through *A* + 3. (And even more combinations once you add SIMD registers.)

And just detecting the conflicting write is the easy part. The hard part is finding all the little pieces that wrote to the memory you want to read and combine them in the right order to reconstruct the final value. (And this might involve going back out to memory if the little pieces do not completely cover the range of memory addresses you want to read.)

In practice, the x86 doesn't bother with the complex reconstruction. When it discovers that there is a complicated interaction between the store buffer and the speculated load, it triggers a *store-to-load forwarding stall*.

I don't know how severe this stall is, but it stands to reason that you don't want it to happen, so the just-in-time compiler I was working on tries to access each variable in exactly the same way (four-byte variables with four-byte instructions, and so on), so that these stalls do not occur.

Raymond Chen

Follow