The Itanium processor, part 7: Speculative loads

devblogs.microsoft.com/oldnewthing/20150804-00

August 4, 2015

Raymond Chen

Today we'll look at speculative loads, which is when you load a value before you even know whether the value should be loaded. (A related but distinct concept is advanced loading, which we'll look at next time.)

Consider the following code:

```
int32_t SomeClass::calculateSomething(int32_t *a, int32_t *b)
{
 int32_t result;
 if (m_p > m_q + 1) {
  result = a[*b];
 } else {
  result = defaultValue;
 }
 return result;
}
```

The naïve way to compile this function would go something like this:

```
        // we are a leaf function, so no need to use "alloc" or to save rp.
        // on entry: r32 = this, r33 = a, r34 = b

        // calculate complex condition, putting the result in p6
        // and the opposite of the result in p7.

        addl    r31 = 08h, r32          // r31 = &this->m_p
        addl    r30 = 10h, r32 ;;       // r30 = &this->m_q
        ld8     r31 = [r31]             // r31 = this->m_p
        ld8     r30 = [r30] ;;          // r30 = this->m_q
        addl    r30 = 08h, r30 ;;       // r30 = this->m_q + 1
        cmp.gt  p6, p7 = r30, r31 ;;    // p6 = m_p > m_q + 1; p7 = !p6

        // Now take action based on the result.

(p6)    ld4     r29 = [r34] ;;          // if true: load *b
(p6)    shladd  r29 = r29, 2, r33 ;;    // if true: calculate &a[*b]
(p6)    ld4     ret0 = [r29]            // if true: load a[*b]
(p7)    or      ret0 = 2ah, r0          // if false: return default value
        br.ret.sptk.many rp             // return
```

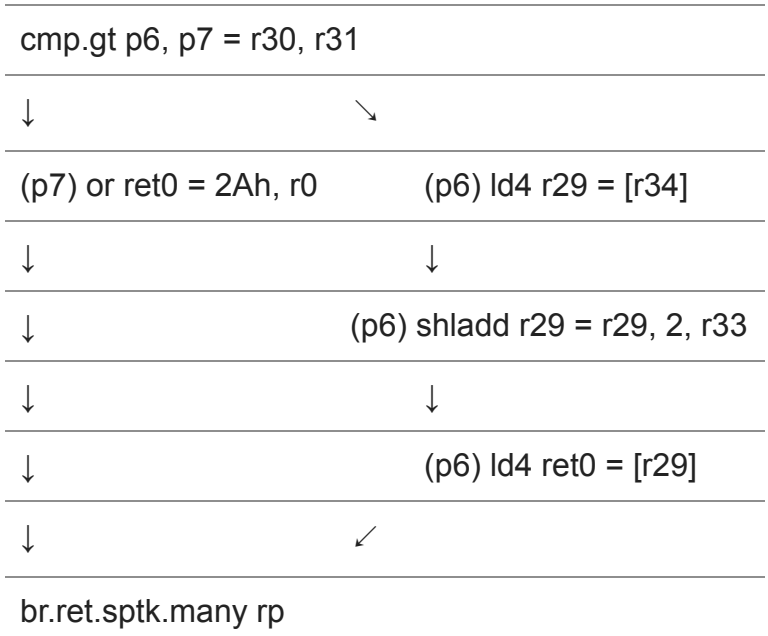
First we decide whether the condition is true or not. Once we know whether the branch is taken, we execute the appropriate branch. If the condition is true, then we load `*b`, then use it to index the array `a`. If the condition is false, then we just set the result to the default value.

Now, in reality, the encoded instructions aren't that neat due to template restrictions. For example, we cannot put the first three instructions into a bundle because they consist of two arithmetic instructions, a stop, and a memory instruction, but there is no II|M template. The actual encoding would be more like this:

```
        // we are a leaf function, so no need to use "alloc" or to save rp.
        // on entry: r32 = this, r33 = a, r34 = b

        // calculate complex condition, putting the result in p6
        // and the opposite of the result in p7.

{      // MII| template
        nop.m
        addl    r31 = 08h, r32          // r31 = &this->m_p
        addl    r30 = 10h, r32 ;;       // r30 = &this->m_q
}
{       // MM|I| template (M port can also execute integer operations)
        ld8     r31 = [r31]             // r31 = this->m_p
        ld8     r30 = [r30] ;;          // r30 = this->m_q
        addl    r30 = 08h, r30 ;;       // r30 = this->m_q + 1
}
{       // M|MI| template
        cmp.gt  p6, p7 = r30, r31 ;;    // p6 = m_p > m_q + 1; p7 = !p6

        // Now take action based on the result.

(p6)    ld4     r29 = [r34]             // if true: load *b
        nop.i ;;                        // move the stop here
}
{       //  M|MI template
(p6)    shladd  r29 = r29, 2, r33 ;;    // if true: calculate &a[*b]
(p6)    ld4     ret0 = [r29]            // if true: load a[*b]
(p7)    or      ret0 = 2ah, r0          // if false: return default value
}
{       // BBB template
        br.ret.sptk.many rp             // return
        nop.b
        nop.b
}
```

Anyway, let's go back to the original version. What you might notice is that there is a long dependency chain:

addl r31 = 08h, r32	addl r30 = 10h, r32
↓	↓
ld8 r31 = [r31]	ld8 r30 = [r30]
↓	↓
↓	addl r30 = 08h, r30
↓ ↙	



It would be great if we could parallelize more of this computation. For example, we could precalculate the *true* branch:

```
        // we are a leaf function, so no need to use "alloc" or to save rp.
        // on entry: r32 = this, r33 = a, r34 = b

        ld4     r29 = [r34]             // assume true: load *b
        addl    r31 = 08h, r32
        addl    r30 = 10h, r32 ;;
        shladd  r29 = r29, 2, r33       // assume true: calculate &a[*b]
        ld8     r31 = [r31]
        ld8     r30 = [r30] ;;
        ld4     r29 = [r29]             // assume true: load a[*b]
        addl    r30 = 08h, r30 ;;       // r30 = this->m_q + 1
        cmp.gt  p6, p7 = r30, r31 ;;    // p6 = m_p > m_q + 1; p7 = !p6

        // Now take action based on the result.

(p6)    or      ret0 = r0, r29          // if true: use the value we precalculated
(p7)    or      ret0 = 2ah, r0          // if false: return the default value
        br.ret.sptk.many rp             // return
```

After applying template restrictions, the code looks more like this:

```
{       // MII| template
        ld4     r29 = [r34]             // assume true: load *b
        addl    r31 = 08h, r32
        addl    r30 = 10h, r32 ;;
}
{       // MMI| template
        ld8     r31 = [r31]
        ld8     r30 = [r30]
        shladd  r29 = r29, 2, r33 ;;    // assume true: calculate &a[*b]
}
{       // MI|I| template
        ld4     r29 = [r29]             // assume true: load a[*b]
        addl    r30 = 08h, r30 ;;       // r30 = this->m_q + 1
        cmp.gt  p6, p7 = r30, r31 ;;    // p6 = m_p > m_q + 1; p7 = !p6
}
{       // MIB template (recalling that the M port can also execute integer
instructions)
(p6)    or      ret0 = r0, r29          // if true: use the value we precalculated
(p7)    or      ret0 = 2ah, r0          // if false: return the default value
        br.ret.sptk.many rp             // return
}
```

Note that we managed to shrink the code because we were able to use the speculative instructions to "fill in the holes" left by the template-mandated `nop` instructions in the original. We managed to squeeze out all the `nop` s!

Okay, enough with the template restrictions digressions.

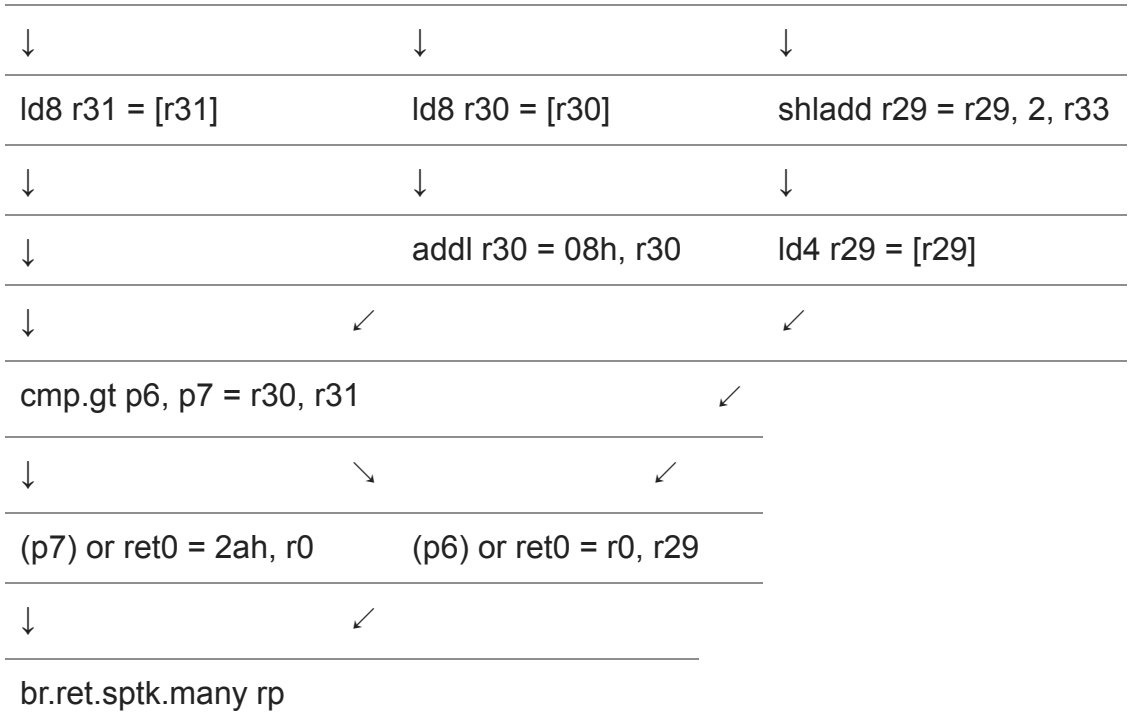
This alternate version assumes that the complex condition is true and speculatively evaluates the *true* branch. If the test turns out to be true, then it just uses the precalculated result. if it turns out to be false, then the precalculated result is thrown away.

In other words, we rewrote the code like this:

```
int32_t SomeClass::calculateSomething(int32_t *a, int32_t *b)
{
 int32_t speculativeResult = a[*b];
 int32_t result;
 if (m_p > m_q + 1) {
  result = speculativeResult;
 } else {
  result = defaultValue;
 }
 return result;
}
```

The dependency chain is now much shorter.

addl r31 = 08h, r32 addl r30 = 10h, r32 ld4 r29 = [r34]



This rewrite is generally beneficial if profiling feedback suggests that the conditional is normally true, since it hides the memory access latency inside the calculation of the conditional.

Until somebody does this:

```
// Get the default value. We know that m_p == m_q.
p->calculateSomething(nullptr, nullptr);
```

In this case, the speculative execution will take an access violation because it tries to deference the null pointer as part of calculating the speculative result.

Well, that sucks.

To solve this problem, the Itanium lets you explicitly tag memory read operations as speculative. If you try to load a value speculatively, the instruction will read the value if possible, but if doing so would normally raise an exception, the processor says, "Sorry, I couldn't read it. But since this is part of speculative execution, I shouldn't raise an exception. Instead, I will set the NaT bit on the value so that you will know that the speculative load failed."

The NaT bit (short for *Not a Thing*) is a 65th bit associated with each 64-bit general-purpose integer register that says whether the register holds a valid value. (Floating point registers do not have a NaT bit; instead there is a special value called NaTVal which serves the same purpose.) Arithmetic operations on NaT simply result in another NaT, but if you try to do something interesting with a NaT, you incur a `STATUS_REG_NAT_CONSUMPTION` exception.

So let's take advantage of explicit speculative execution:

```
        // we are a leaf function, so no need to use "alloc" or to save rp.
        // on entry: r32 = this, r33 = a, r34 = b

        ld4.s   r29 = [r34]             // speculatively load *b
        addl    r31 = 08h, r32
        addl    r30 = 10h, r32 ;;
        ld8     r31 = [r31]
        ld8     r30 = [r30]
        shladd  r29 = r29, 2, r33 ;;    // speculatively calculate &a[*b]
        ld4.s   r29 = [r29]             // speculatively load a[*b]
        addl    r30 = 08h, r30 ;;       // r30 = this->m_q + 1
        cmp.gt  p6, p7 = r30, r31 ;;    // p6 = m_p > m_q + 1; p7 = !p6

        // Now take action based on the result.

(p6)    or      ret0 = r0, r29          // if true: use the value we precalculated
(p7)    or      ret0 = 2ah, r0          // if false: return the default value
        br.ret.sptk.many rp             // return
```

We changed the two load operations from `ld4` to `ld4.s`. The trailing `.s` means that the load is being performed speculatively, and that if an error occurs or if the address is itself NaT, then set the result to NaT rather than raising an exception.

Okay, so this prevents the exception from being raised during the speculative execution, but what if an exception really occurred? How do we turn the NaT back into the original exception? As written, we return NaT back to our caller, which is definitely not what the caller expects!

You might put on your language lawyer hat at this point and say that dereferencing a null pointer invokes *undefined behavior*, so returning NaT is standard-conforming (because undefined behavior allows *anything* to be standard-conforming). That's true, if the exception was due to an access violation. But the exception might have been a *page not present* exception because the memory was paged out. In that case, we really do want to raise the exception so that the kernel can handle it by paging the memory back in, and then we want to read the value and resume our calculations. The caller definitely does not expect that passing valid parameters will result in a NaT just because the memory happens to be paged out.

What we need to do is convert the deferred exception back into the original exception so that it can be raised as if no speculation had occurred. The instruction that lets us know that an exception got converted to a NaT is `chk.s`. This means "Check if the register contains NaT. If so, then jump to recovery code." The recovery code re-executes the instructions non-speculatively so that all the exceptions can be raised in the standard way, and any exception handlers can do their work in the standard way. Since NaT infects future computations, we don't need to check every speculative step; we need only check the final speculated result.

```
        // we are a leaf function, so no need to use "alloc" or to save rp.
        // on entry: r32 = this, r33 = a, r34 = b

        ld4.s   r29 = [r34]             // speculatively load *b
        addl    r31 = 08h, r32
        addl    r30 = 10h, r32 ;;
        ld8     r31 = [r31]
        ld8     r30 = [r30]
        shladd  r29 = r29, 2, r33 ;;    // speculatively calculate &a[*b]
        ld4.s   r29 = [r29]             // speculatively load a[*b]
        addl    r30 = 08h, r30 ;;       // r30 = this->m_q + 1
        cmp.gt  p6, p7 = r30, r31 ;;    // p6 = m_p > m_q + 1; p7 = !p6

        // Now take action based on the result.

(p6)    chk.s   r29, recovery           // if true: recover r29 if not valid
recovered:
(p6)    or      ret0 = r0, r29          // if true: use the value we precalculated
(p7)    or      ret0 = 2ah, r0          // if false: return the default value
        br.ret.sptk.many rp             // return

recovery:
        ld4     r29 = [r34] ;;          // load *b
        shladd  r29 = r29, 2, r33 ;;    // calculate &a[*b]
        ld4     r29 = [r29]             // load a[*b]
        br      recovered               // resume with recovered value
```

The `chk.s` instruction checks the specified register to see if the NaT bit is set. If not, the instruction allows execution to continue normally. But if the register is invalid, then control transfers to the specified label. Our recovery code re-executes the instructions that led to the invalid value, but this time we execute them non-speculatively so that exceptions can be raised and handled. Once the value has been recovered, we jump back to the instruction after the `chk.s` so that normal execution can resume.

In this case, we can make an additional optimization. Since the only things happening after recovery are copying `r29` to `ret0` and returning, we can inline those two instructions then perform peephole optimization to combine the `ld4 r29 = [r29]` and `or ret0 = r0, r29` into `ld4 ret0 = [r29]`.

Note that optimizing the recovery code is not really that important from an execution speed standpoint, since the recovery code runs only if an exception occurred, and the cost of raising and handling the exception will drown out any cycle squeezing effects. The real benefit of optimizing the recovery code is to avoid the jump back into the mainline code, because that allows the mainline code to be more compact: Recall that all jump targets must be the start of a bundle. If we had the recovery code jump back to the mainline code, we would have to insert some `nop` s so that the `recovered` label is at the start of a bundle. (In practice, what the compiler will do is repeat the trailing instructions in the bundle containing the `chk.s` then jump to the start of the next bundle.)

The final compiled function now looks like this:

```
        // we are a leaf function, so no need to use "alloc" or to save rp.
        // on entry: r32 = this, r33 = a, r34 = b

        ld4.s   r29 = [r34]             // speculatively load *b
        addl    r31 = 08h, r32
        addl    r30 = 10h, r32 ;;
        ld8     r31 = [r31]
        ld8     r30 = [r30]
        shladd  r29 = r29, 2, r33 ;;    // speculatively calculate &a[*b]
        ld4.s   r29 = [r29]             // speculatively load a[*b]
        addl    r30 = 08h, r30 ;;       // r30 = this->m_q + 1
        cmp.gt  p6, p7 = r30, r31 ;;    // p6 = m_p > m_q + 1; p7 = !p6

        // Now take action based on the result.

(p6)    chk.s   r29, recovery           // if true: recover r29 if not valid
(p6)    or      ret0 = r0, r29          // if true: use the value we precalculated
(p7)    or      ret0 = 2ah, r0          // if false: return the default value
        br.ret.sptk.many rp             // return

recovery:
        ld4     r29 = [r34] ;;          // load *b
        shladd  r29 = r29, 2, r33 ;;    // calculate &a[*b]
        ld4     ret0 = [r29]            // load a[*b] as return value
        br.ret.sptk.many rp             // return
```

Next time, we'll look at advanced loading, which is a different type of speculative execution.

Raymond Chen

Follow