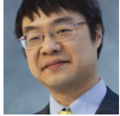
The Itanium processor, part 8: Advanced loads

devblogs.microsoft.com/oldnewthing/20150805-00

August 5, 2015



Raymond Chen

Today we'll look at advanced loads, which is when you load a value before you're supposed to, in the hope that the value won't change in the meantime.

Consider the following code:

```
int32_t SomeClass::tryGetValue(int32_t *value)
{
 if (!m_errno) {
  *value = m_value;
  m_readCount++;
 }
 return m_errno;
}
```

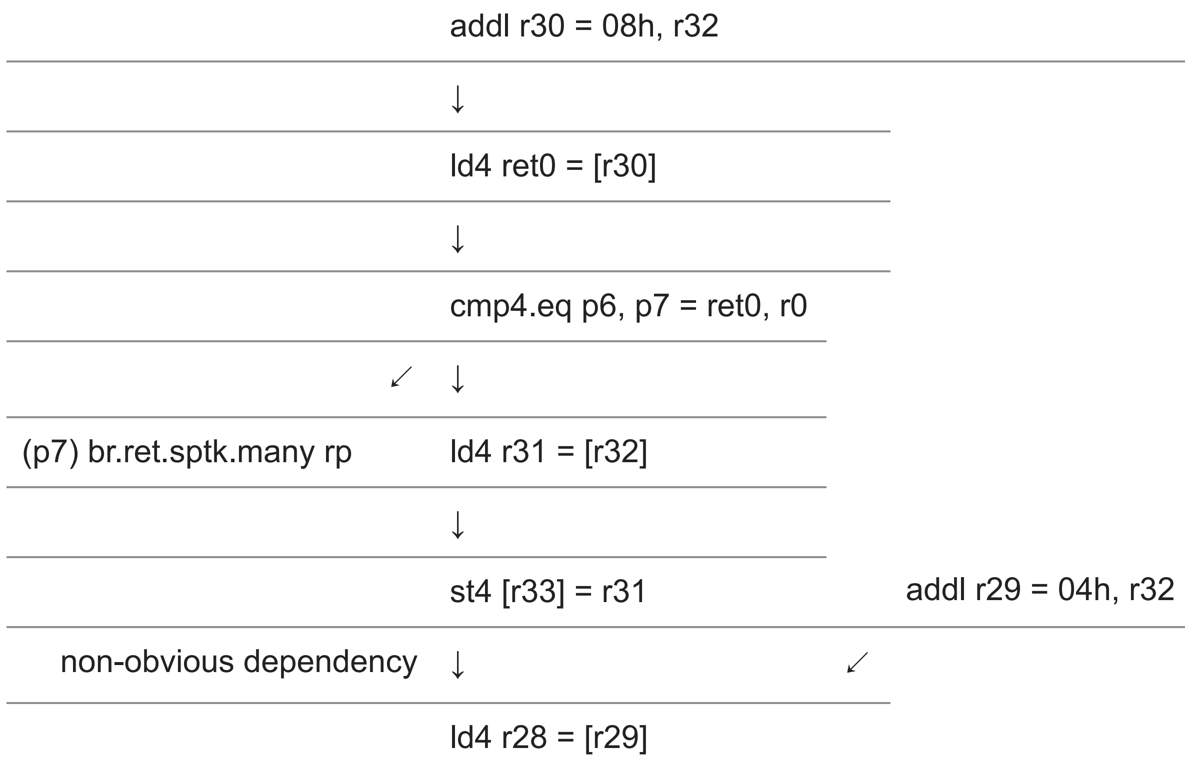
Let's say that the `SomeClass` has `m_value` at offset zero, `m_errno` at offset 4, and `m_readCount` at offset 8.

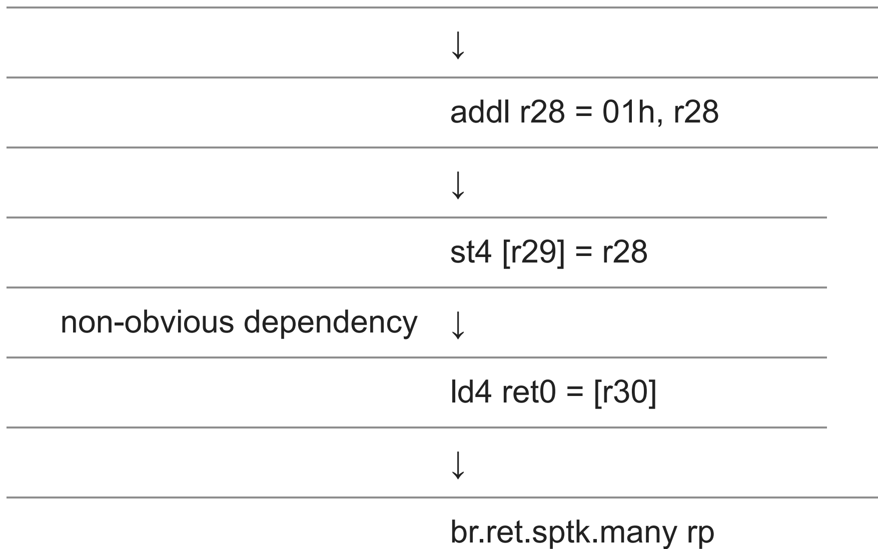
The naïve way of compiling this function would go something like this:

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

        addl    r30 = 08h, r32          // calculate &m_errno
        addl    r29 = 04h, r32 ;;       // calculate &m_readCount

        ld4     ret0 = [r30] ;;         // load m_errno

        cmp4.eq p6, p7 = ret0, r0       // p6 = m_errno == 0, p7 = !p6

(p7)    br.ret.sptk.many rp             // return m_errno if there was an error¹

        ld4     r31 = [r32] ;;          // load m_value (at offset 0)
        st4     [r33] = r31 ;;          // store m_value to *value

        ld4     r28 = [r29] ;;          // load m_readCount
        addl    r28 = 01h, r28 ;;       // calculate m_readCount + 1
        st4     [r29] = r28 ;;          // store updated m_readCount

        ld4     ret0 = [r30]            // reload m_errno for return value

        br.ret.sptk.many rp             // return
```

First, we calculate the addresses of our member variables. Then we load `m_errno`, and if there is an error, then we return it immediately. Otherwise, we copy the current value to `*value`, load `m_readCount`, increment it, and finally, we return `m_errno`.

The problem here is that we have a deep dependency chain.





	↓
	addl r28 = 01h, r28
	↓
	st4 [r29] = r28
non-obvious dependency	↓
	ld4 ret0 = [r30]
	↓
	br.ret.sptk.many rp

Pretty much every instruction depends on the result of the previous instruction. Some of these dependencies are obvious. You have to calculate the address of a member variable before you can read it, and you have to get the result of a memory access befure you can perform arithmetic on it. Some of the dependencies are not obvious. For example, we cannot access `m_value` or `m_readCount` until after we confirm that `m_errno` is zero to avoid a potential access violation if the object straddles a page boundary with `m_errno` on one page and `m_value` on the other (invalid) page. (We saw last time how this can be solved with speculative loads, but let's not add that to the mix yet.)

Returning `m_errno` is a non-obvious dependency. We'll see why later. For now, note that the return value came from a memory access, which means that if the caller of the function tries to use the return value, it may stall waiting for the result to arrive from the memory controller.

When you issue a read on Itanium, the processor merely initiates the operation and proceeds to the next instruction before the read completes. If you try to use the result of the read too soon, the processor stalls until the value is received from the memory controller. Therefore, you want to put as much distance as possible between the load of a value from memory and the attempt to use the result.

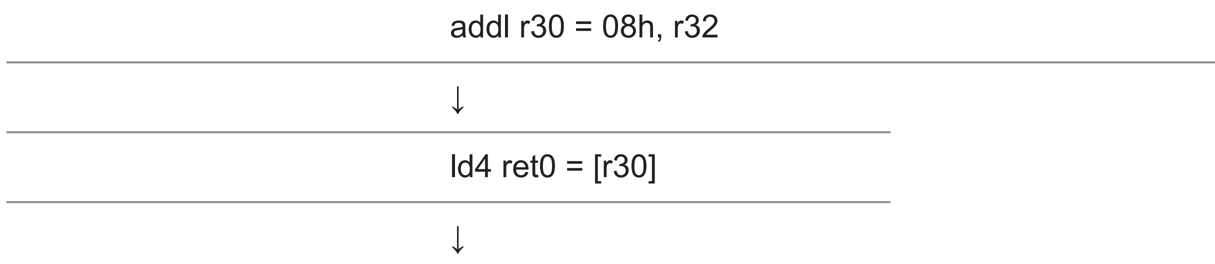
Let's see what we can do to parallelize this function. We'll perform the increment of `m_readCount` and the fetch of `m_value` simultaneously.

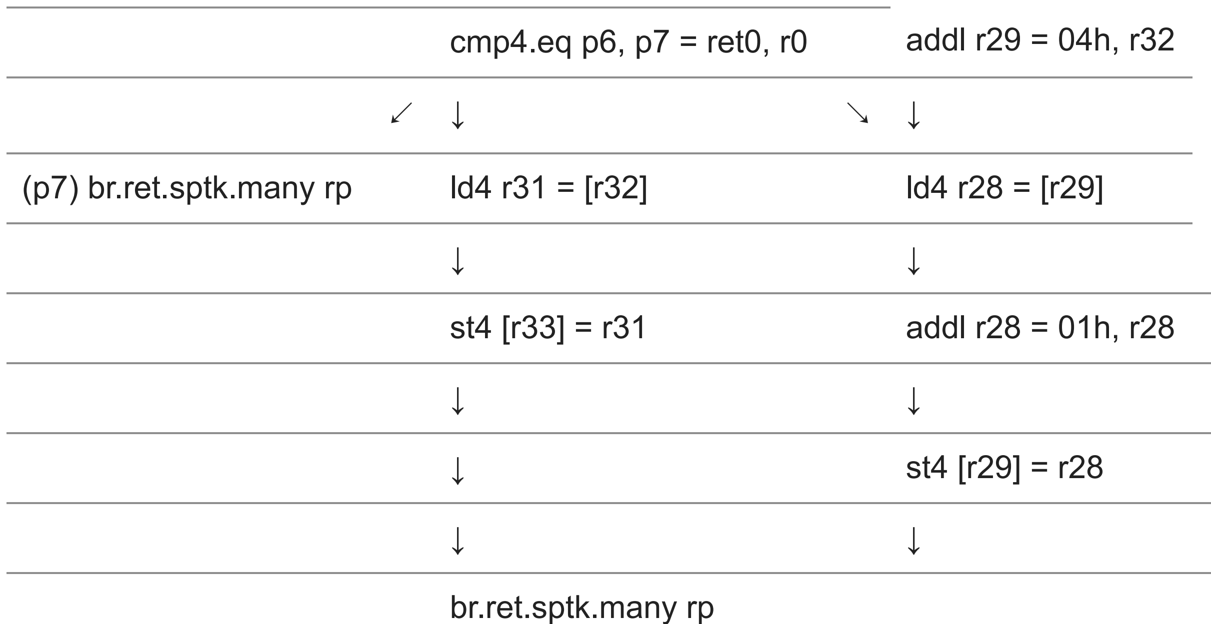
```
        // we are a leaf function, so no need to use "alloc" or to save rp.
        // on entry: r32 = this, r33 = value

        addl    r30 = 08h, r32          // calculate &m_errno
        addl    r29 = 04h, r32 ;;       // calculate &m_readCount

        ld4     ret0 = [r30] ;;         // load m_errno

        cmp4.eq p6, p7 = ret0, r0       // p6 = m_errno == 0, p7 = !p6

(p7)    br.ret.sptk.many rp             // return m_errno if there was an error

        ld4     r31 = [r32]             // load m_value (at offset 0)
        ld4     r28 = [r29] ;;          // preload m_readCount

        addl    r28 = 01h, r28          // calculate m_readCount + 1
        st4     [r33] = r31 ;;          // store m_value to *value

        st4     [r29] = r28             // store updated m_readCount

        br.ret.sptk.many rp             // return (answer already in ret0)
```

We’ve basically rewritten the function as

```
int32_t SomeClass::getValue(int32_t *value)
{
 int32_t local_errno = m_errno;
 if (!local_errno) {
  int32_t local_readCount = m_readCount;
  int32_t local_value = m_value;
  local_readCount = local_readCount + 1;
  *value = local_value;
  m_readCount = local_readCount;
 }
 return local_errno;
}
```

This time we loaded the return value from `m_errno` long before the function ends, so when the caller tries to use the return value, it will definitely be ready and not incur a memory stall. (If a stall were needed, it would have occurred at the `cmp4`.) And we’ve also shortened the dependency chain significantly in the second half of the function.





	cmp4.eq p6, p7 = ret0, r0	addl r29 = 04h, r32
↙	↓ ↘	↓
(p7) br.ret.sptk.many rp	ld4 r31 = [r32]	ld4 r28 = [r29]
	↓	↓
	st4 [r33] = r31	addl r28 = 01h, r28
	↓	↓
	↓	st4 [r29] = r28
	↓	↓
	br.ret.sptk.many rp	

This works great until somebody does this:

```
int32_t SomeClass::Haha()
{
  return this->tryGetValue(&m_readCount);
}
```

or even this:

```
int32_t SomeClass::Hoho()
{
  return this->tryGetValue(&m_errno);
}
```

Oops.

Let's look at `Haha`. Suppose that our initial conditions are `m_errno = 0`, `m_value = 42`, and `m_readCount = 0`.

Original		**Optimized**	
		local_errno = m_errno;	// true
if (!m_errno)	// true	if (!m_errno)	// true
		readCount = m_readCount;	// 0
*value = m_value;	// m_readCount = 42	*value = m_value;	// m_readCount = 42

m_readCount++;	// m_readCount = 43	m_readCount = readCount + 1;	// m_readCount = 1
return m_errno;	// 0	return errno;	// 0

The original code copies the `value` before incrementing the read count. This means that if the caller says that `m_readCount` is the output variable, the act of copying the value *modifies* `m_readCount`. This modified value is then incremented. Our optimized version does not take this case into account and sets `m_readCount` to the old value incremented by 1.

We were faked out by pointer aliasing!

(A similar disaster occurs in `Hoho`.)

Now, whether the behavior described above is intentional or desirable is not at issue here. The C++ language specification requires that the original code result in the specified behavior, so the compiler is required to honor it. Optimizations cannot alter the behavior of standard-conforming code, even if that behavior seems strange to a human being reading it.

But we can still salvage this optimization by handling the aliasing case. The processor contains support for aliasing detection via the `ld.a` instruction.

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

        addl    r30 = 08h, r32          // calculate &m_errno
        addl    r29 = 04h, r32 ;;       // calculate &m_readCount

        ld4     ret0 = [r30] ;;         // load m_errno

        cmp4.eq p6, p7 = ret0, r0       // p6 = m_errno == 0, p7 = !p6

(p7)    br.ret.sptk.many rp             // return m_errno if there was an error

        ld4     r31 = [r32]             // load m_value (at offset 0)
        ld4.a   r28 = [r29] ;;          // preload m_readCount

        addl    r28 = 01h, r28          // calculate m_readCount + 1
        st4     [r33] = r31             // store m_value to *value

        chk.a.clr r28, recover ;;       // recover from pointer aliasing
recovered:
        st4     [r29] = r28 ;;          // store updated m_readCount

        br.ret.sptk.many rp             // return

recover:
        ld4     r28 = [r29] ;;          // reload m_readCount
        addl    r28 = 01h, r28          // recalculate m_readCount + 1
        br      recovered               // recovery complete, resume mainline code
```

The `ld.a` instruction is the same as an `ld` instruction, but it also tells the processor that this is an *advanced load*, and that the processor should stay on the lookout for any instructions that write to any bytes accessed by the load instruction. When the value is finally consumed, you perform a `chk.a.clr` to check whether the value you loaded is still valid. If no instructions have written to the memory in the meantime, then great. But if the address was written to, the processor will jump to the recovery code you provided. The recovery code re-executes the load and any other follow-up calculations, then returns to the original mainline code path.

The `.clr` completer tells the processor to stop monitoring that address. It clears the entry from the Advanced Load Address Table, freeing it up for somebody else to use.

There is also a `ld.c` instruction which is equivalent to a `chk.a` that jumps to a reload and then jumps back. In other words,

```
ld.c.clr r1 = [r2]
```

is equivalent to

```
    chk.a.clr r1, recover
recovered:
    ...

recover:
    ld     r1 = [r2]
    br     recovered
```

but is much more compact and doesn't take branch penalties. This is used if there is no follow-up computation; you merely want to reload the value if it changed.

As with recovery from speculative loads, we can inline some of the mainline code into the recovery code so that we don't have to pad out the mainline code to get `recovered` to sit on a bundle boundary. I didn't bother doing it here; you can do it as an exercise.

The nice thing about processor support for pointer aliasing detection is that it can be done across functions, something that cannot easily be done statically. Consider this function:

```
void accumulateTenTimes(void (*something)(int32_t), int32_t *victim)
{
 int32_t total = 0;
 for (int32_t i = 0; i < 10; i++) {
  total += something(*victim);
 }
 *victim = total;
}

int32_t negate(int32_t a) { return -a; }

int32_t value = 2;
accumulateTenTimes(negate, &value);
// result: value = -2 + -2 + -2 + ... + -2 = -20

int32_t sneaky_negate(int32_t a) { value2 /= 2; return -a; }
int32_t value2 = 2;
accumulateTenTimes(sneaky_negate, &value2);
// result: value2 = -2 + -1 + -0 + -0 + ... + -0 = -3
```

When compiling the `accumulateTenTimes` function, the compiler has no way of knowing whether the `something` function will modify `victim`, so it must be conservative and assume that it might, just in case we are in the `sneaky_negate` case.

Let's assume that the compiler has done flow analysis and determined that the function pointer passed to `accumulateTenTimes` is always within the same module, so it doesn't need to deal with `gp`. Since function descriptors are immutable, it can also enregister the function address.

```
        // 2 input registers, 6 local registers, 1 output register
        alloc   r34 = ar.pfs, 2, 6, 1, 0
        mov     r35 = rp                // save return address
        mov     r36 = ar.lc             // save loop counter
        or      r37 = r0, r0            // total = 0
        ld8     r38 = [r32]             // get the function address
        or      r31 = 09h, r0 ;;        // r31 = 9
        mov     ar.lc = r31             // loop nine more times (ten total)
again:
        ld4     r39 = [r33]             // load *victim for output
        mov     b6 = r38                // move to branch register
        br.call.dptk.many rp = b6 ;;    // call function in b6
        addl    r37 = ret0, r37         // accumulate total
        br.cloop.sptk.few again ;;      // loop 9 more times

        st4     [r33] = r37             // save the total

        mov     ar.lc = r36             // restore loop counter
        mov     rp = r35                // restore return address
        mov     ar.pfs = r34            // restore stack frame
        br.ret.sptk.many rp             // return
```

Note that at each iteration, we read `*victim` from memory because we aren't sure whether the `something` function modifies it. But with advanced loads, we can remove the memory access from the loop.

```
        // 2 input registers, 7 local registers, 1 output register
        alloc   r34 = ar.pfs, 2, 7, 1, 0
        mov     r35 = rp                // save return address
        mov     r36 = ar.lc             // save loop counter
        or      r37 = r0, r0            // total = 0
        ld8     r38 = [r32]             // get the function address
        or      r31 = 09h, r0 ;;        // r31 = 9
        mov     ar.lc = r31             // loop nine more times (ten total)
        ld4.a   r39 = [r33]             // get the value of *victim
again:
        ld4.c.nc r39 = [r33]            // reload *victim if necessary
        or      r40 = r39, r0           // set *victim as the output parameter
        mov     b6 = r38                // move to branch register
        br.call.dptk.many rp = b6 ;;    // call function in b6
        addl    r37 = ret0, r37         // accumulate total
        br.cloop.sptk.few again ;;      // loop 9 more times

        invala.e r39                    // stop tracking r39

        st4     [r33] = r37             // save the total

        mov     ar.lc = r36             // restore loop counter
        mov     rp = r35                // restore return address
        mov     ar.pfs = r34            // restore stack frame
        br.ret.sptk.many rp             // return
```

We perform an advanced load of `*value` in the hope that the callback function will not modify it. This is true if the callback function is `negate`, but it will trigger reloads if the accumulator function is `sneaky_negate`.

Note here that we use the `.nc` completer on the `ld.c` instruction. This stands for *no clear* and tells the processor to keep tracking the address because we will be checking it again. When the loop is over, we use `invala.e` to tell the processor, "Okay, you can stop tracking it now." This also shows how handy the `ld.c` instruction is. We can do the reload inline rather than have to write separate recovery code and jumping out and back.

(Processor trivia: We do not need a stop after the `ld4.c.nc`. You are allowed to consume the result of a check load in the same instruction group.)

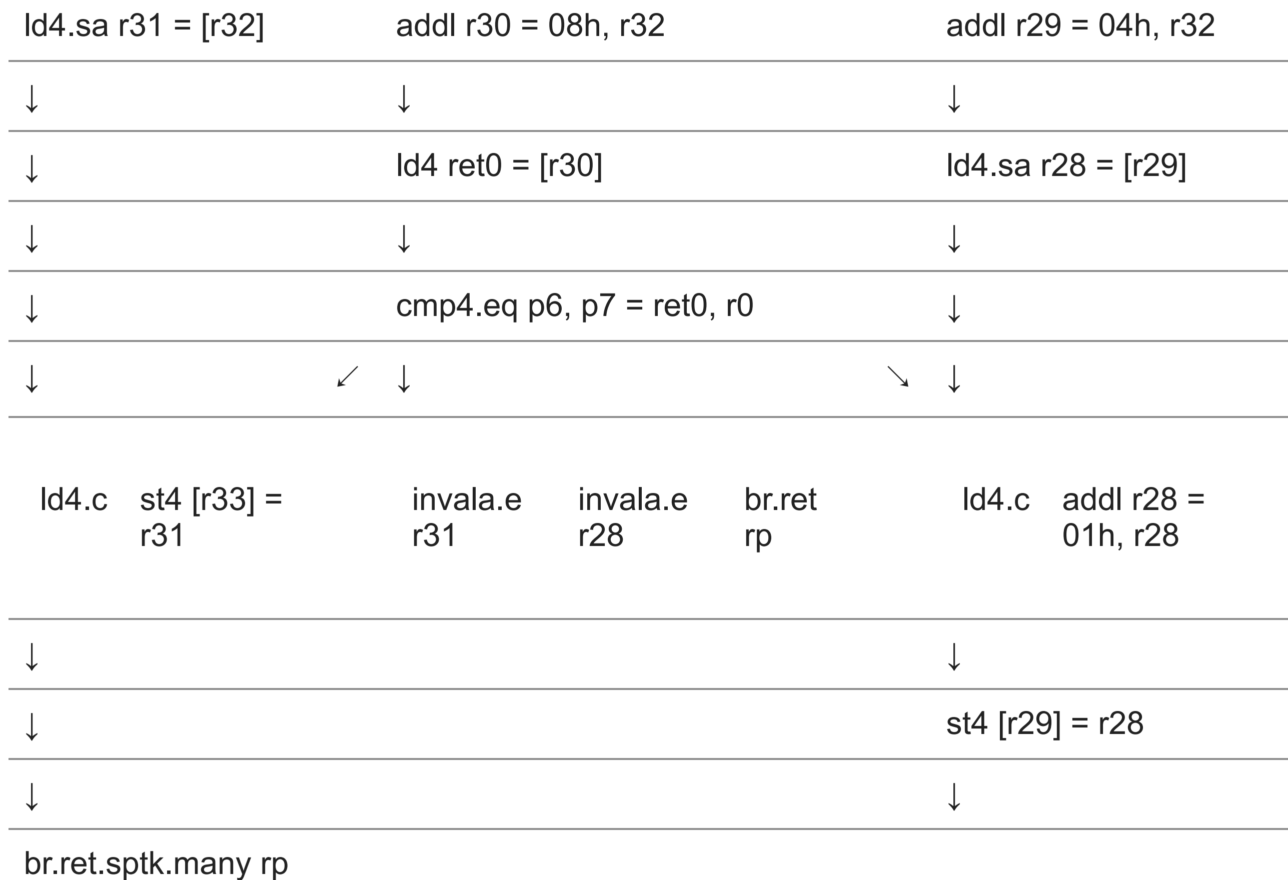
In the case where the callback function does not modify `value`, the only memory accesses performed by this function and the callback are loading the function address, loading the initial value from `*value`, and storing the final value to `*value`. The loop body itself runs without any memory access at all!

Going back to our original function, I noted that we could also add speculation to the mix. So let's do that. We're going to speculate an advanced load!

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

        ld4.sa  r31 = [r32]             // speculatively preload m_value (at offset
0)
        addl    r30 = 08h, r32          // calculate &m_errno
        addl    r29 = 04h, r32 ;;       // calculate &m_readCount

        ld4.sa  r28 = [r29]             // speculatively preload m_readCount
        ld4     ret0 = [r30] ;;         // load m_errno

        cmp4.eq p6, p7 = ret0, r0       // p6 = m_errno == 0, p7 = !p6

(p7)    invala.e r31                    // abandon the advanced load
(p7)    invala.e r28                    // abandon the advanced load
(p7)    br.ret.sptk.many rp             // return false if value not set

        ld4.c.clr r31 = [r32]           // validate speculation and advanced load of
m_value
        st4     [r33] = r31             // store m_value to *value

        ld4.c.clr r28 = [r29]           // validate speculation and advanced load of
m_readCount
        addl    r28 = 01h, r28 ;;       // calculate m_readCount + 1
        st4     [r29] = r28             // store updated m_readCount

        br.ret.sptk.many rp             // return
```

To validate a speculative advanced load, you just need to do a `ld.c`. If the speculation failed, then the advanced load also fails, so all we need to do is check the advanced load. and the reload will raise the exception.

The dependency chain for this function is even shorter now that we were able to speculate the case where there is no error. (Since you are allowed to consume an `ld4.c` in the same instruction group, I combined the `ld4.c` and its consumption in a single box since they occur within the same cycle.)



Aw, look at that pretty diagram. Control speculation and data speculation allowed us to run three different operations in parallel even though they might have dependencies on each other. The idea here is that if profiling suggests that the dependencies are rarely realized (pointers are usually not aliased), you can use speculation to run the operations as if they had no dependencies, and then use the check instructions to convert the speculated results to real ones.

[1] Note the absence of a stop between the `cmp4` and the `br.ret`. That's because of a special Itanium rule that says that a conditional branch is permitted to use a predicate register calculated earlier within the same instruction group. (Normally, instructions within an instruction group are not allowed to have dependencies among each other.) This allows a test and jump to occur within the same cycle.

Raymond Chen

Follow