The Itanium processor, part 10: Register rotation

devblogs.microsoft.com/oldnewthing/20150807-00

August 7, 2015

Raymond Chen

Last time, we looked at counted loops and then improved a simple loop by explicitly pipelining the loop iterations. This time, we're going to take the pipelining to the next level.

Let's reorder the columns of the chart we had last time so the instructions are grouped not by the register being operated upon but by the operation being performed. Since no instructions within an instruction group are dependent on each other in our example, I can reorder them without affecting the logic.

1	`ld4 r32 = [r29], 4`				`;;`
2	`ld4 r33 = [r29], 4`				`;;`
3	`ld4 r34 = [r29], 4`		`adds r32 = r32, 1`		`;;`
4	`ld4 r35 = [r29], 4`		`adds r33 = r33, 1`	`st4 [r28] = r32, 4`	`;;`
5	`ld4 r32 = [r29], 4`		`adds r34 = r34, 1`	`st4 [r28] = r33, 4`	`;;`
6	`ld4 r33 = [r29], 4`		`adds r35 = r35, 1`	`st4 [r28] = r34, 4`	`;;`
7	`ld4 r34 = [r29], 4`		`adds r32 = r32, 1`	`st4 [r28] = r35, 4`	`;;`
8	`ld4 r35 = [r29], 4`		`adds r33 = r33, 1`	`st4 [r28] = r32, 4`	`;;`
9	`ld4 r32 = [r29], 4`		`adds r34 = r34, 1`	`st4 [r28] = r33, 4`	`;;`
10	`ld4 r33 = [r29], 4`		`adds r35 = r35, 1`	`st4 [r28] = r34, 4`	`;;`

11			`adds r32 = r32, 1`	`st4 [r28] = r35, 4`	`;;`
12			`adds r33 = r33, 1`	`st4 [r28] = r32, 4`	`;;`
13				`st4 [r28] = r33, 4`	`;;`

What an interesting pattern. Each column represents a functional unit, and at each cycle, the unit operates on a different register in a clear pattern: *r32*, *r33*, *r34*, *r35*, then back to *r32*. The units are staggered so that each operates on a register precisely when its result from the previous unit is ready.

Suppose you have to make 2000 sandwiches and you have four employees. You could arrange your sandwich factory with three stations. At the first station, you have the bread and the toaster. At the second station, you have the protein. At the third station, you have the toppings. Each employee goes through the stations in order: First they take two pieces of bread and put them in the toaster. When the toast is finished, they add the protein, then they add the toppings, and then they put the finished sandwich in the box. Once that's done, they go back to the first station. You stagger the starts of the four employees so that at any moment, one is preparing the bread, one is waiting for the toaster, one is adding protein, and one is adding the toppings.

That is how the original code was arranged. Each register is an employee that is at one of the four stages of sandwich construction.

But another way to organize your sandwich factory is as an assembly line. You put one employee in charge of the bread, one in charge of the toaster, one in charge of the protein, and one in charge of the toppings. When a sandwich completes a stage in the process, it gets handed from one employee to the next.

(And since there isn't really anything for the toaster-boy to do, you can eliminate that position and create the same number of sandwiches per second with only three employees. The original version had each employee sitting idle waiting for the toaster 25% of the time. Switching to the assembly line model allowed us to squeeze out that idle time.)

Let's apply the assembly line model to our code. Handing a sandwich from one person to the next is done by moving the value from one register to the next. Let's imagine than there is a *slide* instruction that you can put at the end of an instruction group which copies *r32* to *r33*, *r33* to *r34*, and so on.

1	`ld4 r32 = [r29], 4`				`slide ;;`

2	`ld4 r32 = [r29], 4`				`slide ;;`
3	`ld4 r32 = [r29], 4`		`adds r34 = r34, 1`		`slide ;;`
4	`ld4 r32 = [r29], 4`		`adds r34 = r34, 1`	`st4 [r28] = r35, 4`	`slide ;;`
5	`ld4 r32 = [r29], 4`		`adds r34 = r34, 1`	`st4 [r28] = r35, 4`	`slide ;;`
6	`ld4 r32 = [r29], 4`		`adds r34 = r34, 1`	`st4 [r28] = r35, 4`	`slide ;;`
7	`ld4 r32 = [r29], 4`		`adds r34 = r34, 1`	`st4 [r28] = r35, 4`	`slide ;;`
8	`ld4 r32 = [r29], 4`		`adds r34 = r34, 1`	`st4 [r28] = r35, 4`	`slide ;;`
9	`ld4 r32 = [r29], 4`		`adds r34 = r34, 1`	`st4 [r28] = r35, 4`	`slide ;;`
10	`ld4 r32 = [r29], 4`		`adds r34 = r34, 1`	`st4 [r28] = r35, 4`	`slide ;;`
11			`adds r34 = r34, 1`	`st4 [r28] = r35, 4`	`slide ;;`
12			`adds r34 = r34, 1`	`st4 [r28] = r35, 4`	`slide ;;`
13				`st4 [r28] = r35, 4`	`slide ;;`

During the execution of the first instruction group, the first value is loaded into *r32*, and the *slide* instruction slides it into *r33*.

At the second instruction group, the second value is loaded into *r32*, and the first value sits unchanged in *r33*. (Technically, the value is waiting to be loaded into *r33*.) The *slide* instruction slides the second value into *r33* and the first value into *r34*.

At the third instruction group, the third value is loaded into *r32*, and the first value (now in *r34*) is incremented. Then the *slide* instruction slides the third value into *r33*, the second value into *r34*, and the first value into *r35*.

At the fourth instruction group, the fourth value is loaded into *r32*, the second value (now in *r34*) is incremented, and the first value (now in *r35*) is stored to memory. Then the *slide* instruction slides the fourth value into *r33*, the third value into *r34*, and the second value into *r35*. (The first value slides into *r36*, but we don't really care.)

And so on. At each instruction group, a fresh value is loaded into *r32*, a previously-loaded value is incremented in *r34*, and the incremented value is stored from *r35*. And then the *slide* instruction moves everything down one step for the next instruction group.

When we reach the 11th instruction group, we drain out the last value and don't bother starting up any new ones.

Observe that the above code also falls into a *prologue*/*kernel*/*epilogue* pattern. In the prologue, the assembly line starts up and gradually fills the registers with work. In the kernel, the assembly line is completely busy. And in the epilogue, the work of the final registers drains out.

You can already see how *br.cloop* would come in handy here: The kernel can be written as a single-instruction loop! But wait there's more.

Let's add some predicate registers to the mix. Let's suppose that the `slide` instruction slides not only integer registers but also predicate registers.

1	`(p16) ld4 r32 = [r29], 4`		`(p18) adds r34 = r34, 1`	`(p19) st4 [r28] = r35, 4`	`slide ;;`
2	`(p16) ld4 r32 = [r29], 4`		`(p18) adds r34 = r34, 1`	`(p19) st4 [r28] = r35, 4`	`slide ;;`
3	`(p16) ld4 r32 = [r29], 4`		`(p18) adds r34 = r34, 1`	`(p19) st4 [r28] = r35, 4`	`slide ;;`
4	`(p16) ld4 r32 = [r29], 4`		`(p18) adds r34 = r34, 1`	`(p19) st4 [r28] = r35, 4`	`slide ;;`
5	`(p16) ld4 r32 = [r29], 4`		`(p18) adds r34 = r34, 1`	`(p19) st4 [r28] = r35, 4`	`slide ;;`
6	`(p16) ld4 r32 = [r29], 4`		`(p18) adds r34 = r34, 1`	`(p19) st4 [r28] = r35, 4`	`slide ;;`
7	`(p16) ld4 r32 = [r29], 4`		`(p18) adds r34 = r34, 1`	`(p19) st4 [r28] = r35, 4`	`slide ;;`
8	`(p16) ld4 r32 = [r29], 4`		`(p18) adds r34 = r34, 1`	`(p19) st4 [r28] = r35, 4`	`slide ;;`
9	`(p16) ld4 r32 = [r29], 4`		`(p18) adds r34 = r34, 1`	`(p19) st4 [r28] = r35, 4`	`slide ;;`
10	`(p16) ld4 r32 = [r29], 4`		`(p18) adds r34 = r34, 1`	`(p19) st4 [r28] = r35, 4`	`slide ;;`
11	`(p16) ld4 r32 = [r29], 4`		`(p18) adds r34 = r34, 1`	`(p19) st4 [r28] = r35, 4`	`slide ;;`

12	`(p16) ld4 r32 = [r29], 4`		`(p18) adds r34 = r34, 1`	`(p19) st4 [r28] = r35, 4`	`slide ;;`
13	`(p16) ld4 r32 = [r29], 4`		`(p18) adds r34 = r34, 1`	`(p19) st4 [r28] = r35, 4`	`slide ;;`

We can initally set *p16* = *true*, *p17* = *p18* = *p19* = *false*. That way, only the load executes from the first instruction group. And then the *slide* instruction slides both the integer registers and the predicate registers, which causes *p17* to become *true*.

In the second instruction group, again, only the load executes. And then the *slide* instruction slides *p17* into *p18*, so now *p18* = *true* also.

Since *p18* = *true*, the third instruction group both loads and increments. And then the *slide* instruction slides *p18* into *p19*, so now all of the predicates are true.

With all the predicates true, every step in instruction groups three through 10 execute.

Now, with instruction group 11, we want to slide the predicates, but also set *p16* to *false*. That turns off the *ld4* instruction.

The *p16* = *false* then slides into *p17* for instruction group 12, then into *p18* for instruction group 13, which turns off the increment instruction.

If we can get the *slide* instruction to slide the predicates and set *p16* to *true* for the first 10 instructions, and set it to *false* for the last three, then we can simply execute the same instruction 13 times!

Okay, now I can reveal the true identity of the *slide* instruction: It's called *br.ctop*.

The *br.ctop* instruction works like this:

```
if (ar.lc != 0) { slide; p16 = true; ar.lc = ar.lc - 1; goto branch; }
if (ar.ec != 0) { slide; p16 = false; ar.ec = ar.ec - 1;
                  if (ar.ec != 0) goto branch; }
else { /* unimportant */ }
```

In words, the *br.ctop* instruction first checks the *ar.lc* register (*loop counter*). If it is nonzero, then the registers slide over, the *p16* register is set to *true*, the loop counter is decremented, and the jump is taken.

If *ar.lc* is zero, then the *br.ctop* instruction checks the *ar.ec* register (*epilogue counter*). If it is nonzero, then the register slide over, the *p16* register is set to *false*, and.the epilogue counter is decremented. If the decremented value of the epilogue counter is nonzero, then the jump is taken; otherwise we fall through and the loop ends.

(If both *ar.lc* and *ar.ec* are zero, then the loop is finished before it even started. Some weird edge-case handing happens here which is not important to the discussion.)

Code that takes advantage of the *br.ctop* instruction goes like this:

```
      alloc r36 = ar.pfs, 0, 8, 0, 4 // four rotating registers!
      mov r37 = ar.lc         // preserve lc
      mov r38 = ar.ec         // preserve ec
      mov r39 = preds         // preserve predicates

      addl r31 = r0, 1999 ;;  // r31 = 1999
      mov ar.lc = r31         // ar.lc = 1999
      mov ar.ec = 4
      mov pr.rot = 1 << 16 // p16 = true, all others false
      addl r29 = gp, -205584  // calculate start of array
      addl r28 = r29, 0 ;;     // put it in both r28 and r29

again:
(p16) ld4 r32 = [r29], 4      // execute an entire loop with
(p18) adds r34 = r34, 1       // a single instruction group
(p19) st4 [r28] = r35, 4      // using this one weird trick
      br.ctop again ;;

      mov ar.lc = r37         // restore registers we preserved
      mov ar.ec = r38
      mov preds = r39
      mov ar.pfs = r36
      br.ret.sptk.many rp     // return
```

We are now using the last parameter to the *alloc* instruction. The *4* says that we want four rotating registers. The *ar.lc* and *ar.ec* register must be preserved across calls, so we save them here for restoration at the end. Predicate registers *p16* through *p63* must also be preserved, so we save all the predicate registers by using the *preds* pseudo-register which grabs all 64 predicates into a single 64-bit value.

Next, we set up the loop by setting the loop counter to the number of additional times we want to execute the loop (not counting the one execution we get via fall-through), the epilogue counter to the number of steps we need in order to drain the final iterations, and set the predicates so that *p16* = *true* and everything else is *false*. We also set up *r28* and *r29* to step through the array.

Once that is done, we can execute the entire loop in a single instruction group.

And then we clean up after the loop by restoring all the registers to how we found them, then return.

And there you have register rotation. It lets you compress the prologue, kernel, and epilogue of a pipelined loop into a single instruction group.

I pulled a fast one here: The Itanium requires that the number of rotating registers be a multiple of eight. So our code really should look like this:

```
      alloc r40 = ar.pfs, 0, 12, 0, 8
      mov r41 = ar.lc         // preserve lc
      mov r42 = ar.ec         // preserve ec
      mov r43 = preds         // preserve predicates

      addl r31 = r0, 1999 ;;  // r31 = 1999
      mov ar.lc = r31         // ar.lc = 1999
      mov ar.ec = 4
      mov pr.rot = 1 << 16 // p16 = true, all others false
      addl r29 = gp, -205584  // calculate start of array
      addl r28 = r29, 0 ;;     // put it in both r28 and r29

again:
(p16) ld4 r32 = [r29], 4      // execute an entire loop with
(p18) adds r34 = r34, 1       // a single instruction group
(p19) st4 [r28] = r35, 4      // using this one weird trick
      br.ctop again ;;

      mov ar.lc = r41         // restore registers we preserved
      mov ar.ec = r42
      mov preds = r43
      mov ar.pfs = r40
      br.ret.sptk.many rp     // return
```

Instead of four rotating registers, we use eight. The underlying analysis remains the same. We are just throwing more registers into the pot.

Now, the loop we were studying happens to be very simple, with only one load and one store. For more complex loops, you may need to use things like the unconditional comparison, or pipelining the iterations with a stagger of more than one cycle.

There are other types of instructions for managing loops with register rotation. For example, *br.cexit* is like *br.ctop* except that it jumps when *br.ctop* falls through and vice versa. This is handy to put at the start of your pipelined loop to handle the case where the number of iterations is zero. There are also *br.wtop* and *br.wexit* instructions to handle `while` loops instead of counted loops. The basic idea is the same, so I won't go into the details. You can read the Itanium manual to learn more.

That's the end of the whirlwind tour of the Itanium architecture. There are still parts left unexplored, but I tried to hit the most interesting sights.

Raymond Chen

Follow