The Itanium processor, part 9: Counted loops and loop pipelining

devblogs.microsoft.com/oldnewthing/20150806-00

August 6, 2015

Raymond Chen

There is a dedicated register named *ar.lc* for counted loops. The *br.cloop* instruction acts like this:

```
    if (ar.lc != 0) { ar.lc = ar.lc - 1; goto branch; }
```

Consider this loop to increment every 32-bit integer in an array.

```
extern int array[2000];

void IncrementEachElement()
{
 for (int i = 0; i < 2000; i++) {
  array[i]++;
 }
}
```

This could be compiled as

```
    mov r30 = ar.lc         // save original value of ar.lc
    addl r29 = gp, -205584  // calculate start of array
    addl r31 = r0, 1999 ;;  // r31 = 1999
    mov ar.lc = r31         // loop 1999 times
again:
    ld4 r31 = [r29] ;;      // load the next integer
    adds r31 = r31, 1 ;;    // increment the value
    st4 [r29] = r31, 4      // store it and autoincrement
    br.cloop again ;;       // do it 1999 more times
    mov ar.lc = r30         // restore ar.lc
    br.ret.sptk.many rp     // return
```

Note that the *ar.lc* register is initialized to one fewer than the number of iterations desired. That's because it counts the number of times the *br.cloop* instruction will branch. Since we used fall-through to initiate the loop, one of the times through the loop was already performed, and we want *br.cloop* to branch only 1999 times.

The *ar.lc* register must be preserved across calls, so if you intend to use it in your function, you need to save its original value and restore it when done. (You also need to record in the unwind table where you saved it, so it's easier to do it up front; otherwise, you have to go to the extra work of encoding how you shrink-wrapped the function.)

For the sake of illustration, let's say that the CPU can fetch memory from cache in two cycles, and each cycle it can issue one load and one store. (If the memory access is not in cache, it takes basically forever, in which case it doesn't really matter how we optimize the rest of the code, so we may as well assume that all memory accesses are cache hits.) Each iteration of the loop performs a fetch (two cycles), an addition (one cycle), then a store in parallel with a conditional jump (one cycle), for a total of four cycles per iteration.

Let's try to do better.

First, let's simplify to the case where the array has only four elements. We could do it like this:

```
alloc r36 = ar.pfs, 0, 5, 0, 0 // set up frame
addl r29 = gp, -205584  // calculate start of array
addl r28 = r29, 0 ;;    // put it in both r28 and r29

ld4 r32 = [r29], 4 ;;   // crazy stuff
ld4 r33 = [r29], 4 ;;
adds r32 = r32, 1
ld4 r34 = [r29], 4 ;;
st4 [r28] = r32, 4
adds r33 = r33, 1
ld4 r35 = [r29], 4 ;;
st4 [r28] = r33, 4
adds r34 = r34, 1 ;;
st4 [r28] = r34, 4
adds r35 = r35, 1 ;;
st4 [r28] = r35, 4 ;;

mov ar.pfs = r36
br.ret.sptk.many rp     // return
```

(In reality, we would reorder the instructions in order to match the templates better, but I'll leave them in this order for now.)

That is kind of hard to understand, so let me rewrite the crazy middle part like this, putting all the instructions from an instruction group on one line, adding some separator lines, and putting instructions into columns carefully chosen to highlight the structure of the code.

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

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

The first thing to observe is that this sequence completes in just seven cycles, as opposed to the 16 cycles of the original version. That's over double the performance!

Notice that each column performs one iteration of the loop. Each column uses a different register to do the calculation, and they share register `r29` to hold the address of the next value to read and `r28` to hold the address of the next value to write. Each column also waits two cycles after each read before consuming the result, thereby avoiding memory stalls.

The idea here is to run multiple iterations of the loop in parallel, but setting each one to begin one cycle after the start of the previous iteration. Staggering the starts keeps us from overloading the memory controller. (Otherwise, everybody would issue load requests in cycle 1, and the memory controller would stall.)

Now, the Itanium has a lot of registers, but it doesn't have 2000 of them. Fortunately, we don't need 2000 of them. Observe that starting at cycle 5, we can reuse register `r32` because the previous iteration doesn't need it any more. So if we need to increment ten elements, we can do it this way:

1	`ld4 r32 = [r29], 4`				`;;`
2		`ld4 r33 = [r29], 4`			`;;`
3	`adds r32 = r32, 1`		`ld4 r34 = [r29], 4`		`;;`
4	`st4 [r28] = r32, 4`	`adds r33 = r33, 1`		`ld4 r35 = [r29], 4`	`;;`

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

We incremented ten elements in 13 cycles instead of 40. In general, we can increment n elements in $n + 3$ cycles instead of $4n$. For large values of n this is a four-fold speed-up over the original version.

The pattern above breaks down into three natural sections.

1	`ld4 r32 = [r29], 4`				`;;`	Warm-up
2		`ld4 r33 = [r29], 4`			`;;`	
3	`adds r32 = r32, 1`		`ld4 r34 = [r29], 4`		`;;`	
4	`st4 [r28] = r32, 4`	`adds r33 = r33, 1`		`ld4 r35 = [r29], 4`	`;;`	Cruise
5	`ld4 r32 = [r29], 4`	`st4 [r28] = r33, 4`	`adds r34 = r34, 1`		`;;`	
6		`ld4 r33 = [r29], 4`	`st4 [r28] = r34, 4`	`adds r35 = r35, 1`	`;;`	
7	`adds r32 = r32, 1`		`ld4 r34 = [r29], 4`	`st4 [r28] = r35, 4`	`;;`	

8	`st4 [r28] = r32, 4`	`adds r33 = r33, 1`		`ld4 r35 = [r29], 4`	`;;`	
9	`ld4 r32 = [r29], 4`	`st4 [r28] = r33, 4`	`adds r34 = r34, 1`		`;;`	
10		`ld4 r33 = [r29], 4`	`st4 [r28] = r34, 4`	`adds r35 = r35, 1`	`;;`	
11	`adds r32 = r32, 1`			`st4 [r28] = r35, 4`	`;;`	Cool-down
12	`st4 [r28] = r32, 4`	`adds r33 = r33, 1`			`;;`	
13		`st4 [r28] = r33, 4`			`;;`	

The first three cycles comprise the warm-up phase (formally known as the *prologue*). At the start, no registers are doing any work, but during the course of the warm-up phase, they get into the act one at a time. At the end of the warm-up phase, all the registers are busy doing work.

Most of the time is spent in the middle cruise phase (formally known as the *kernel*), wherein all four registers are busy carrying out one of the iterations. Note that during every cycle of the cruise phase, there is a load, an increment, and a store, with the registers taking turns performing each of the operations.

The last three cycles are the cool-down phase (formally known as the *epilogue*), where the registers start draining their last bits of work and no new work is started.

Okay, now that we understand how the above code works, we're going to turn it on its side next time. Stay tuned for the thrilling conclusion!

Raymond Chen

Follow