Non-classical processor behavior: How doing something can be faster than not doing it

devblogs.microsoft.com/oldnewthing/20140613-00

June 13, 2014

Raymond Chen

Consider the following program:

```
#include <windows.h>
#include <stdlib.h>
#include <stdlib.h>
#include <stdio.h>
int array[10000];
int countthem(int boundary)
{
 int count = 0;
 for (int i = 0; i < 10000; i++) {
  if (array[i] < boundary) count++;
 }
 return count;
}
int __cdecl wmain(int, wchar_t **)
{
 for (int i = 0; i < 10000; i++) array[i] = rand() % 10;
 for (int boundary = 0; boundary <= 10; boundary++) {
  LARGE_INTEGER liStart, liEnd;
  QueryPerformanceCounter(&liStart);
  int count = 0;
  for (int iterations = 0; iterations < 100; iterations++) {
   count += countthem(boundary);
  }
  QueryPerformanceCounter(&liEnd);
  printf("count=%7d, time = %I64d\n",
         count, liEnd.QuadPart - liStart.QuadPart);
 }
 return 0;
}
```

The program generates a lot of random integers in the range 0..9 and then counts how many are less than 0, less than 1, less than 2, and so on. It also prints how long the operation took in QPC units. We don't really care how big a QPC unit is; we're just interested in the relative values. (We print the number of items found merely to verify that the result is close to the expected value of `boundary * 100000`.)

Here are the results:

boundary	count	time	
0	0	1869	
1	100000	5482	
2	200800	8152	
3	300200	10180	
4	403100	11982	
5	497400	12092	
6	602900	11029	
7	700700	9235	
8	797500	7051	
9	902500	4537	
10	1000000	1864	

To the untrained eye, this chart is strange. Here's the naïve analysis:

When the boundary is zero, there is no incrementing at all, so the entire running time is just loop overhead. You can think of this as our control group. We can subtract 1869 from the running time of every column to remove the loop overhead costs. What remains is the cost of running `count` increment instructions.

The cost of a single increment operation is highly variable. At low boundary values, it is around 0.03 time units per increment. But at high boundary values, the cost drops to one tenth that.

What's even weirder is that once the count crosses 600,000, each addition of another 100,000 increment operations makes the code run *faster*, with the extreme case when the boundary value reaches 10, where we run faster than if we hadn't done any incrementing at all!

How can the running time of an increment instruction be *negative*?

The explanation for all this is that CPUs are more complicated than the naïve analysis realizes. We saw earlier that modern CPUs contain all sorts of hidden variables. Today's hidden variable is the branch predictor.

Executing a single CPU instruction takes multiple steps, and modern CPUs kick off multiple instructions in parallel, with each instruction at a different stage of execution, a technique known as pipelining.

Conditional branch instructions are bad for pipelining. Think about it: When a conditional branch instruction enters the pipeline, the CPU doesn't know whether the condition will be true when the instruction reaches the end of the pipeline. Therefore, it doesn't know what instruction to feed into the pipeline next.

Now, it could just sit there and let the pipeline sit idle until the branch/no-branch decision is made, at which point it now knows which instruction to feed into the pipeline next. But that wastes a lot of pipeline capacity, because it will take time for those new instructions to make it all the way through the pipeline and start doing productive work.

To avoid wasting time, the processor has an internal *branch predictor* which remembers the recent history of which conditional branches were taken and which were not taken. The fanciness of the branch predictor varies. Some processors merely assume that a branch will go the same way that it did the last time it was countered. Others keep complicated branch history and try to infer patterns (such as "the branch is taken every other time").

When a conditional branch is encountered, the branch predictor tells the processor which instructions to feed into the pipeline. If the branch prediction turns out to be correct, then we win! Execution continues without a pipeline stall.

But if the branch prediction turns out to be incorrect, then we lose! All of the instructions that were fed into the pipeline need to be recalled and their effects undone, and the processor has to go find the correct instructions and start feeding them into the pipeline.

Let's look at our little program again. When the boundary is 0, the result of the comparison is always false. Similarly, when the boundary is 10, the result is always true. In those cases, the branch predictor can reach 100% accuracy.

The worst case is when the boundary is 5. In that case, half of the time the comparison is true and half of the time the comparison is false. And since we have random data, fancy historical analysis doesn't help any. The predictor is going to be wrong half the time.

Here's a tweak to the program: Change the line

```
    if (array[i] < boundary) count++;
```

to

```
    count += (array[i] < boundary) ? 1 : 0;
```

This time, the results look like this:

boundary	count	time
0	0	2932
1	100000	2931 </span
2	200800	2941 </span
3	300200	2931 </span
4	403100	2932 </span
5	497400	2932 </span
6	602900	2932 </span
7	700700	2999 </span
8	797500	2931 </span
9	902500	2932 </span
10	1000000	2931 </span

The execution time is now independent of the boundary value. That’s because the optimizer was able to remove the branch from the ternary expression:

```
; on entry to the loop, ebx = boundary
    mov edx, offset array ; start at the beginning of the array
$LL3:
    xor ecx, ecx    ; start with zero
    cmp [edx], ebx  ; compare array[i] with boundary
    setl cl         ; if less than boundary, then set al = 1
    add eax, ecx    ; accumulate result in eax
    add edx, 4      ; loop until end of array
    cmp edx, offset array + 40000
    jl $LL3
```

Since there are no branching decisions in the inner loop aside from the loop counter, there is no need for a branch predictor to decide which way the comparison goes. The same code executes either way.

Exercise: Why are the counts exactly the same for both runs, even though the dataset is random?

Raymond Chen

Follow