ECE459: Programming for Performance	Winter 2013

Lecture 2 — January 10, 2013

Patrick Lam — *version 2*

Modern Processors

I asked you to watch the video by Cliff Click on modern hardware:

`http://www.infoq.com/presentations/click-crash-course-modern-hardware`

Cliff Click said that 5% miss rates dominate performance. Let's look at why. I looked up a characterization of the SPEC CPU2000 and CPU2006 benchmarks[1].

Here are the reported cache miss rates[2] for SPEC CPU2006.

L1D	40‰
L2	4 ‰

Let's assume that the L1D cache miss penalty is 5 cycles and the L2 miss penalty is 300 cycles, as in the video. Then, for every instruction, you would expect a running time of, on average:

$$1 + 0.04 \times 5 + 0.004 \times 300 = 2.4.$$

Misses are expensive!

Forcing branch mispredicts. It takes a bit of trickery to force branch mispredicts. `gcc` extensions allow hinting, but usually `gcc` or the processor is smart enough to ignore bad hints. This[3] works, though:

```
#include <stdlib.h>
#include <stdio.h>

static __attribute__ ((noinline)) int f(int a) { return a; }

#define BSIZE 1000000
int main(int argc, char* argv[])
{
  int *p = calloc(BSIZE, sizeof(int));
  int j, k, m1 = 0, m2 = 0;
  for (j = 0; j < 1000; j++) {
    for (k = 0; k < BSIZE; k++) {
      if (__builtin_expect(p[k], EXPECT_RESULT)) {
        m1 = f(++m1);
      } else {
        m2 = f(++m2);
      }
    }
  }

  printf("%d, %d\n", m1, m2);
}
```

[1] A. Kejariwal et al. "Comparative architectural characterization of SPEC CPU2000 and CPU2006 benchmarks on the Intel Core 2 Duo processor", SAMOS 2008.

[2] ‰ is "permil", or per-1000.

[3] Source: `blog.man7.org/2012/10/how-much-do-builtinexpect-likely-and.html`.

Running it yields:

```
plam@plym:~/459$ gcc -O2 likely-simplified.c -DEXPECT_RESULT=0 -o likely-simplified
plam@plym:~/459$ time ./likely-simplified
0, 1000000000

real 0m2.521s
user 0m2.496s
sys 0m0.000s
plam@plym:~/459$ gcc -O2 likely-simplified.c -DEXPECT_RESULT=1 -o likely-simplified
plam@plym:~/459$ time ./likely-simplified
0, 1000000000

real 0m3.938s
user 0m3.868s
sys 0m0.000s
```

Limits to parallelization

I mentioned briefly in Lecture 1 that programs often have a sequential part and a parallel part. We'll quantify this observation today and discuss its consequences.

Amdahl's Law. One classic model of parallel execution is Amdahl's Law. In 1967, Gene Amdahl argued that improvements in processor design for single processors would be more effective than designing multi-processor systems. Here's the argument. Let's say that you are trying to run a task which has a serial part, taking fraction S, and a parallelizable part, taking fraction P. Define T_s to be the total amount of time needed on a single-processor system. Now, moving to a parallel system with N processors, the parallel time T_p is instead:

$$T_p = T_s \cdot (S + \frac{P}{N}).$$

As N increases, T_p is dominated by S, limiting the potential speedup for a fixed problem size.

We can restate this law in terms of speedup, which is generally the original time T_s divided by the sped-up time T_p:

$$S_p = \frac{T_s}{T_p}.$$

If we let f be the parallelizable fraction of the computation, i.e. set f to P/T_s, and we let S_f be the speedup we can achieve on f, then we get[4]:

$$S_p(f, S_f) = \frac{1}{(1-f) + \frac{f}{S_f}}.$$

[4] http://www.cs.wisc.edu/multifacet/amdahl/

Plugging in numbers. If $f = 1$, then we can indeed get good scaling, since $S_p = S_f$ in that case; running on an N-processor machine will give you a speedup of N. Unfortunately, usually $f < 1$. Let's see what happens.

f	$S_p(f, 18)$
1	18
0.99	~ 15
0.95	~ 10
0.5	~ 2

To get close to a 2× speedup for $f = 0.5$, you'd need to use around 18 cores.

Amdahl's Law tells you how many cores you can hope to leverage in an execution given a fixed problem size, if you can estimate f.

Consequences of Amdahl's Law. For over 30 years, most performance gains did indeed come from increasing single-processor performance. The main reason that we're here today is that, as we saw in the video in Lecture 2, single-processor performance gains have hit the wall.

By the way, note that we didn't talk about the cost of synchronization between threads here. That can drag the performance down even more.

Amdahl's Assumptions. Despite Amdahl's pessimism, we still all have multicore computers today. Why is that? Amdahl's Law assumes that:

- problem size is fixed (read on);
- the program, or the underlying implementation, behaves the same on 1 processor as on N processors; and
- that we can accurately measure runtimes—i.e. that overheads don't matter.

A more optimistic point of view

In 1988, John Gustafson pointed out[5] that Amdahl's Law only applies to fixed-size problems, but that the point of computers is to deal with bigger and bigger problems.

In particular, you might vary the input size, or the grid resolution, number of timesteps, etc. When running the software, then, you might need to hold the running time constant, not the problem size: you're willing to wait, say, 10 hours for your task to finish, but not 500 hours. So you can change the question to: how big a problem can you run in 10 hours?

According to Gustafson, scaling up the problem tends to increase the amount of work in the parallel part of the code, while leaving the serial part alone. As long as the algorithm is linear, it is possible to handle linearly larger problems with a linearly larger number of processors.

Of course, Gustafson's Law works when there is some "problem-size" knob you can crank up. As a practical example, observe Google, which deals with huge datasets.

[5] `http://www.scl.ameslab.gov/Publications/Gus/AmdahlsLaw/Amdahls.html`