ECE459: Programming for Performance Winter 2016

Lecture 3 — Performance Killers & Amdahl's Law

Patrick Lam & Jeff Zarnett

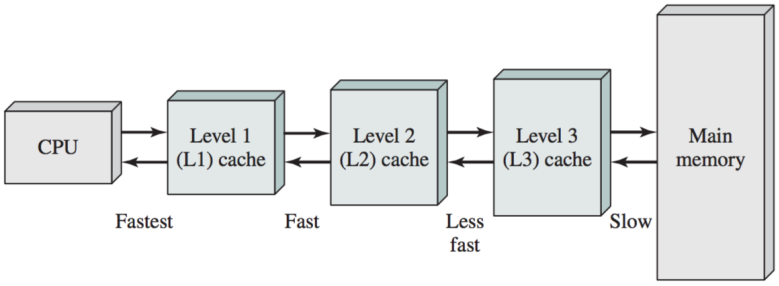
Cache Misses

As discussed, the CPU generates a memory address for a read or write operation. The address will be mapped to a page. Ideally, the page is found in the cache, because that would be faster. If the requested page is, in fact, in the cache, we call that a cache *hit*. If the page is not found in the cache, it is considered a cache *miss*. In case of a miss, we must load the page from memory, a comparatively slow operation. A page miss is also called a *page fault*. The percentage of the time that a page is found in the cache is called the *hit ratio*, because it is how often we have a cache hit. We can calculate the effective access time if we have a good estimate of the hit ratio (which is not overly difficult to obtain) and some measurements of how long it takes to load data from the cache and how long from memory. The effective access time is therefore computed as:

$$\text{Effective Access Time} = h \times t_c + (1 - h) \times t_m$$

Where h is the hit ratio, t_c is the time required to load a page from cache, and t_m is the time to load a page from memory. Of course, we would like the hit ratio to be as high as possible.

Caches have limited size, because faster caches are more expensive. With infinite money we might put everything in registers, but that is rather unrealistic. Caches for memory are very often multileveled; Intel 64-bit CPUs tend to have L1, L2, and L3 caches. L1 is the smallest and L3 is the largest. Obviously, the effective access time formula needs to be updated and expanded, when we have multiple levels of cache with different access times and hit rates. See the diagram below:



Three levels of cache between the CPU and main memory [Sta14].

If we have a miss in the L1 cache, the L2 cache is checked. If the L2 cache contains the desired page, it will be copied to the L1 cache and sent to the CPU. If it is not in L2, then L3 is checked. If it is not there either, it is in main memory and will be retrieved from there and copied to the in-between levels on its way to the CPU.

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 [KVN+08].

Here are the reported cache miss rates[1] for SPEC CPU2006.

L1D	40‰
L2	4 ‰

[1] ‰ is "permil", or per-1000.

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!

If we replace the terms t_c and t_m with t_m and t_d (time to retrieve it from disk) respectively, and redefine h as p, the chance that a page is in memory, we can get an idea of the effective access time in virtual memory:

$$\text{Effective Access Time} = p \times t_m + (1 - p) \times t_d$$

And just while we're at it, we can combine the caching and disk read formulae to get the true effective access time for a system where there is only one level of cache:

$$\text{Effective Access Time} = h \times t_c + (1 - h)(p \times t_m + (1 - p) \times t_d)$$

This is good, but what is t_d? This is a measurable quantity so it is possible, of course, to just measure it[2].

The slow step in all of this, is obviously, the amount of time it takes to load the page from disk. According to [SGG13], restarting the process and managing memory and such take something like 1 to 100 μs. A typical hard drive in their example has a latency of 3 ms, seek time (moving the read head of the disk to the location of the page) is around 5 ms, and a transfer time of 0.05 ms. So the latency plus seek time is the limiting component, and it's several orders of magnitude larger than any of the other costs in the system. And this is for servicing a request; don't forget that several requests may be queued, making the time even longer.

Thus the disk read term t_d dominates the effective access time equation. If memory access takes 200 ns and a disk read 8 ms, we can roughly estimate the access time in nanoseconds as $(1 - p) \times 8\ 000\ 000$.

If the page fault rate is high, performance is awful. If performance of the computer is to be reasonable, the page fault rate has to be very, very low. On the order of 10^{-6}.

Summary: misses are not just expensive, they hurt performance more than anything.

Branch Prediction and Misprediction

The compiler (and the CPU) take a look at code that results in branch instructions such as loops, conditionals, or the dreaded `goto`[3], and it will take an assessment of what it thinks is likely to happen. By default I think it's assumed that backward branches are taken and forward branches are not taken (but that may be wrong). Well, how did we get here anyway?

In the beginning the CPUs and compilers didn't really think about this sort of thing, they would just come across instructions one at a time and do them and that was that. If one of them required a branch, it was no real issue. Then we had pipelining: the CPU would fetch the next instruction while decoding the previous one, and while executing the instruction before. That means if evaluation of an instruction results in a branch, we might go somewhere else and therefore throw away the contents of the pipeline. Thus we'd have wasted some time and effort. If the pipeline is short, this is not very expensive. But pipelines keep getting longer...

So then we got to the subject of branch prediction. The compiler and CPU look at instructions on their way to be executed and analyze whether it thinks it's likely the branch is taken. This can be based on several things, including the recent execution history. If we guess correctly, this is great, because it minimizes the cost of the branch. If we guess wrong, we have to flush the pipeline and take the performance penalty.

[2] One of my favourite engineering sayings is "Don't guess; measure." You may be sick of hearing me say that one by now.
[3] Which I still maintain is a swear word in C.

The compiler and CPU's branch prediction routines are pretty smart. Trying to outsmart them isn't necessarily a good idea. But we can give the compiler (`gcc` at least) some hints about what we think is likely to happen. Our tool for this is the `__builtin_expect()` function, which takes two arguments, the value to be tested and the expected result.

In the linux `compiler.h` header there are two neat little shortcuts defined:

```
# define likely(x)      __builtin_expect(!!(x), 1)
# define unlikely(x)    __builtin_expect(!!(x), 0)
```

These are nice ways of saying that we expect `x` to be true (likely) or false (unlikely). These hints tell the compiler some information about how it should predict. It will then arrange the instructions in such a way that, if the prediction is right, the instructions in the pipeline will be executed. But if we're wrong, then the instructions will have to be flushed.

It's noteworthy that we have to compile with at least optimization level 2 (`-O2`) to get the compiler to take these hints at all. Otherwise they won't do anything.

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 code from [Ker] worked in 2013, 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);
}
```

Running it yielded:

```
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 0m3.868ss
sys 0m0.000s
```

`gcc` seems to have gotten smart enough to reject bogus hints in the interim.

In the original source [Ker] the author reports the following results: scanning a one million element array, with all elements initially zero, the results are:

- No use of hints: `0:02.68 real, 2.67 user, 0.00 sys`
- Good prediction: `0:02.28 real, 2.28 user, 0.00 sys`
- Bad prediction: `0:04.19 real, 4.18 user, 0.00 sys`

Using the hints. From these results we can see pretty clearly that if we're wrong, the penalty is pretty large (assuming the compiler does not look at your hint and think "stupid human, I know better"). Under a lot of circumstances then, it's probably best just to leave it alone, unless we're really, really, really sure.

How sure do we have to be? The answer depends dramatically on the code, the CPU you're using, the compiler, and all those little details. But [Ker] to the rescue here again, because there are some tests here. To cut to the chase, when about one in ten thousand values in the array is nonzero, then it's roughly the "break-even" point for the setup as described.

Conclusion: it's hard to outsmart the compiler. Maybe it's better not to try.

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 $P = 1 - S$. 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 potential speedup.

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

$$\text{speedup} = \frac{T_s}{T_p} = \frac{1}{S + P/N}.$$

Replacing S with $(1 - P)$, we get:

$$\text{speedup} = \frac{1}{(1 - P) + P/N},$$

and

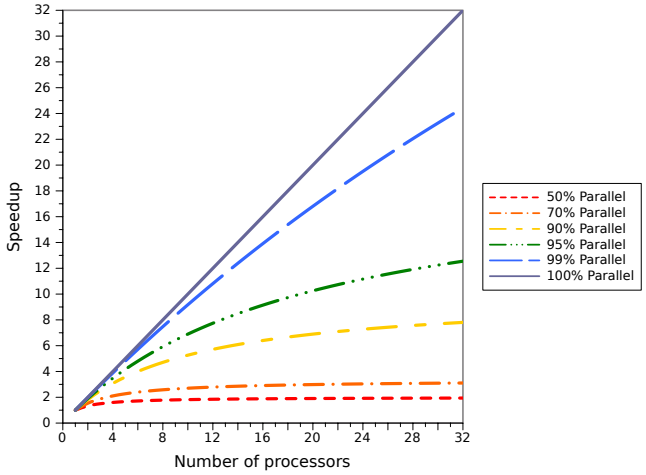
$$\text{max speedup} = \frac{1}{(1 - P)},$$

since $\frac{P}{N} \to 0$.

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

P	speedup ($N = 18$)
1	18
0.99	~ 15
0.95	~ 10
0.5	~ 2

Graphically, we have something like this:



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 P.

Let us consider an example from [HZM14]: Suppose we have a task that can be executed in 5 s and this task contains a loop that can be parallelized. Let us also say initialization and recombination code in this routine requires 400 ms. So with one processor executing, it would take about 4.6 s to execute the loop. If we split it up and execute on two processors it will take about 2.3 s to execute the loop. Add to that the setup and cleanup time of 0.4 s and we get a total time of 2.7 s. Completing the task in 2.7 s rather than 5 s represents a speedup of about 46%. Applying the formula, we get the following run times:

Processors	**Run Time (s)**
1	5
2	2.7
4	1.55
8	0.975
16	0.6875
32	0.54375
64	0.471875
128	0.4359375

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, 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.

Generalizing Amdahl's Law. We made a simplification, which was that programs only have one parallel part and one serial part. Of course, this is not true. The program may have many parts, each of which we can tune to a different degree.

Let's generalize Amdahl's Law:

- $f_1, f_2, \ldots, f_n$: fraction of time in part n
- $S_{f_1}, S_{f_n}, \ldots, S_{f_n}$: speedup for part n

Then,

$$\textit{speedup} = \frac{1}{\frac{f_1}{S_{f_1}} + \frac{f_2}{S_{f_2}} + \ldots + \frac{f_n}{S_{f_n}}}.$$

Example. Consider a program with 4 parts in the following scenario:

		Speedup	
Part	Fraction of Runtime	Option 1	Option 2
1	0.55	1	2
2	0.25	5	1
3	0.15	3	1
4	0.05	10	1

(Note: these speedups don't have to be speedups from parallelization.)

We can implement either Option 1 or Option 2. Which option is better?

"Plug and chug" the numbers:

- **Option 1.**

$$\textit{speedup} = \frac{1}{0.55 + \frac{0.25}{5} + \frac{0.15}{3} + \frac{0.05}{5}} = 1.53$$

- **Option 2.**

$$\textit{speedup} = \frac{1}{\frac{0.55}{2} + 0.45} = 1.38$$

Empirically estimating parallel speedup P. Assuming that you know things that are actually really hard to know, here's a formula for estimating speedup. You don't have to commit it to memory:

$$P_{\text{estimated}} = \frac{\frac{1}{speedup} - 1}{\frac{1}{N} - 1}.$$

It's just an estimation, but you can use it to guess the fraction of parallel code, given N and the speedup. You can then use $P_{\text{estimated}}$ to predict speedup for a different number of processors.

A more optimistic point of view

In 1988, John Gustafson pointed out[4] 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.

References

[HZM14] Douglas Wilhelm Harder, Jeff Zarnett, and Vajih Montaghami. *A Practical Introduction to Real-Time Systems for Undergraduate Engineering*. 2014. Online; version 0.14.12.22.

[Ker] Michael Kerrisk. How much do __builtin_expect(), likely(), and unlikely() improve performance? Online; accessed 12-November-2015.

[KVN+08] A. Kejariwal, A.V. Veidenbaum, A. Nicolau, X. Tian, M. Girkar, H. Saito, and U. Banerjee. Comparative architectural characterization of spec cpu2000 and cpu2006 benchmarks on the intelÂő core 2 duo processor. In *Proceedings, International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation; SAMOS*, 2008.

[SGG13] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts (9th Edition)*. John Wiley & Sons, 2013.

[Sta14] William Stallings. *Operating Systems Internals and Design Principles (8th Edition)*. Prentice Hall, 2014.

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