Lecture 3 — Performance Killers & Amdahl's Law

Patrick Lam & Jeff Zarnett
p.lam@ece.uwaterloo.ca jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

December 3, 2017

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.

- If it is, we call it a cache hit;
- Otherwise, it is a cache miss.

Missed Me by THIS Much

In case of a miss, we must load the page from memory, a comparatively slow operation.

The percentage of the time that a page is found in the cache is called the hit ratio.

Effective Access Time

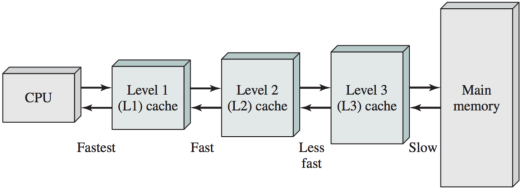
The effective access time is therefore computed as:

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

Cache Size - How Much can you Afford?

Intel 64-bit CPUs tend to have L1, L2, and L3 caches.

L1 is the smallest and L3 is the largest.



Three levels of cache between the CPU and main memory.

Pay Up

Cliff Click said that 5% miss rates dominate performance. Let's look at why.

Here are the reported cache miss rates for SPEC CPU2006.

L1D	40 per 1000
L2	4 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:

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

Bring in the Disk

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.

Effective access time in virtual memory:

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

Put It Together

We can combine the caching and disk read formulae to get the true effective access time for a system where there is only 1 level of cache:

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

We can measure t_d if we're so inclined.

Slow as A Snail Chained to an Anvil

The slow step is the amount of time it takes to load the page from disk.

A typical hard drive may have a latency of 3 ms, seek time is around 5 ms, and a transfer time of 0.05 ms.

This is several orders of magnitude larger than any of the other costs.

Several requests may be queued, making the time even longer.

Dominating Flurry

Thus the disk read term t_d dominates the effective access time equation.

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 so much.

Predict and Mispredict

The compiler (& CPU) take a look at code that results in branch instructions.

Examples: loops, conditionals, or the dreaded `goto`.

It will take an assessment of what it thinks is likely to happen.

Let’s Not Predict

In the beginning the CPUs/compilers didn’t really think about this sort of thing.

They 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...

Not Just for Oil

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...

Take a Guess

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 flush the pipeline and take the performance penalty.

Take a Hint

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.

Check the Header

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)
```

Compile with at least optimization level 2 (`-O2`) to get the compiler to take these hints at all.

Force a Mispredict

```
#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);
}
```

Mispredict Results

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
```

Hint Results

In the original source 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`

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.

Limitations of Speedups

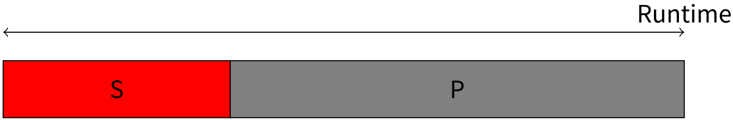
Our main focus is parallelization.

- Most programs have a sequential part and a parallel part; and,
- Amdahl’s Law answers, “what are the limits to parallelization?”

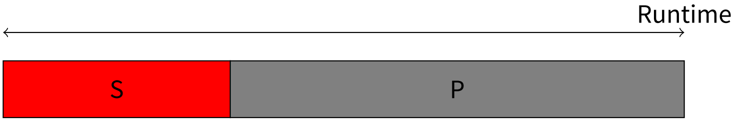
Visualizing Amdahl's Law

S: fraction of serial runtime in a serial execution.
P: fraction of parallel runtime in a serial execution.
Therefore, $S + P = 1$.

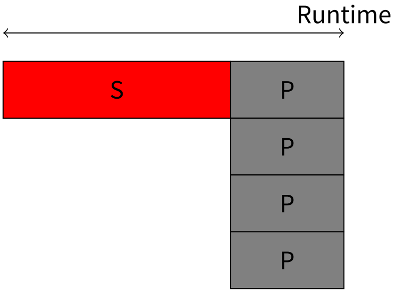
With 4 processors, best case, what can happen to the following runtime?



Visualizing Amdahl's Law



We want to split up the parallel part over 4 processors



Obey the Law

T_s: time for the program to run in serial

N: number of processors/parallel executions

T_p: time for the program to run in parallel

- Under perfect conditions, get N speedup for P

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

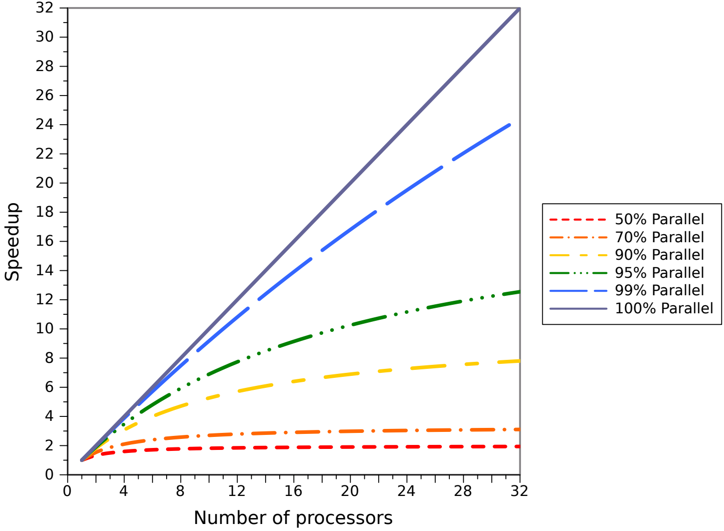
Obey the Law

How much faster can we make the program?

$$\begin{aligned} speedup &= \frac{T_s}{T_p} \\ &= \frac{T_s}{T_S \cdot (S + \frac{P}{N})} \\ &= \frac{1}{S + \frac{P}{N}} \end{aligned}$$

(assuming no overhead for parallelizing; or costs near zero)

Fixed-Size Problem Scaling, Varying Fraction of Parallel Code



Amdahl's Law

Replace S with $(1 - P)$:

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

$$maximum\ speedup = \frac{1}{(1-P)}, \text{since } \frac{P}{N} \rightarrow 0$$

As you might imagine, the asymptotes in the previous graph are bounded by the maximum speedup.

Speedup Example

Suppose: a task that can be executed in 5 s, containing a parallelizable loop.

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.

Split it up and execute on two processors: 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%.

Amdahl’s Law on the 5 s Task

Applying this formula to the example:

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

Observations on the 5 s Task

1. Diminishing returns as we add more processors.

2. Converges on 0.4 s.

The most we could speed up this code is by a factor of $\frac{5}{0.4} \approx 12.5$.

But that would require infinite processors (and therefore infinite money).

Assumptions behind Amdahl's Law

We assume:

- problem size is fixed (we'll see this soon);
- program/algorithm behaves the same on 1 processor and on *N* processors;
- that we can accurately measure runtimes—
  i.e. that overheads don't matter.

Amdahl's Law Generalization

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

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

Application (1)

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

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

Application (2)

“Plug and chug” the numbers:

Option 1

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

Option 2

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

Empirically estimating parallel speedup P

Useful to know, don't have to commit to memory:

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

- Quick way to guess the fraction of parallel code
- Use $P_{\text{estimated}}$ to predict speedup for a different number of processors

Summary of Amdahl’s Law

Important to focus on the part of the program with most impact.

Amdahl’s Law:

- estimates perfect performance gains from parallelization (under assumptions); but,
- only applies to solving a fixed problem size in the shortest possible period of time

Gustafson's Law: Formulation

n: problem size

$S(n)$: fraction of serial runtime for a parallel execution

$P(n)$: fraction of parallel runtime for a parallel execution

$$\begin{aligned} T_p &= S(n) + P(n) = 1 \\ T_s &= S(n) + N \cdot P(n) \end{aligned}$$

$$speedup = \frac{T_s}{T_p}$$

Gustafson's Law

$$speedup = S(n) + N \cdot P(n)$$

Assuming the fraction of runtime in serial part decreases as *n* increases, the speedup approaches *N*.

Yes! Large problems can be efficiently parallelized. (Ask Google.)

Driving Metaphor

Amdahl's Law

Suppose you're travelling between 2 cities 90 km apart. If you travel for an hour at a constant speed less than 90 km/h, your average will never equal 90 km/h, even if you energize after that hour.

Gustafson's Law

Suppose you've been travelling at a constant speed less than 90 km/h. Given enough distance, you can bring your average up to 90 km/h.