

# Lecture 3 — Branch Prediction & Amdahl's Law

Patrick Lam & Jeff Zarnett

`patrick.lam@uwaterloo.ca jzarnett@uwaterloo.ca`

Department of Electrical and Computer Engineering  
University of Waterloo

January 12, 2018

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.

In the beginning the CPUs/compiler 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...

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...

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.

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)
```

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

```
#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 0m3.868s
```

```
sys 0m0.000s
```

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.

I want you to pick up two points from this discussion:

- How branch predictors work
- Applying a (straightforward) expected value computation to predict performance.

# Branch Prediction & Misprediction

---

```
branch_if_not_equal x, 0,  
    else_label  
// Do stuff  
goto end_label  
else_label:  
// Do things  
end_label:  
// whatever happens later
```

---

With no prediction, we need to serialize:

bne.1	bne.2		
		things.1	things.2

If our prediction is correct, we save time.

bne.1	bne.2	
	things.1	things.2

But we might be wrong and need to throw out the bad prediction.

bne.1	bne.2		
	things.1		
		stuff.1	stuff.2

We need to quantify the performance.

Let's pretend that our pipelined CPU executes, on average, one instruction per clock.

Mispredicted branches cost 20 cycles, while correctly-predicted branches cost 1 cycle.

We'll also assume that the instruction mix contains 80% non-branches and 20% branches.

So we can predict average cycles per instruction.

With no prediction (or always-wrong prediction):

$$\text{non\_branch\_}\% \times 1 \text{ cycle} + \text{branch\_}\% \times 20 \text{ cycles} = 4.8 \text{ cycles.}$$

With perfect branch prediction:

$$\text{non\_branch\_}\% \times 1 \text{ cycle} + \text{branch\_}\% \times 1 \text{ cycle} = 1 \text{ cycle.}$$

So we can make our code run  $4.8\times$  faster with branch prediction!



We can predict that a branch is always taken.

If we got 70% accuracy, then our cycles per instruction would be:

$$(0.8 + 0.7 \times 0.2) \times 1 \text{ cycle} + (0.3 \times 0.2) \times 20 \text{ cycles} = 2.14 \text{ cycles.}$$

The simplest possible thing already greatly improves the CPU's average throughput.

Let's leverage that observation about loop branches to do better. Loop branches are, by definition, backwards.

So we can design a branch predictor which predicts “taken” for backwards and “not taken” for forwards.

Let's say that this might get us to 80% accuracy.

$$(0.8 + 0.8 \times 0.2) \times 1 \text{ cycle} + (0.2 \times 0.2) \times 20 \text{ cycles} = 1.76 \text{ cycles.}$$

The PPC 601 (1993) and 603 used this scheme.

So far, we will always make the same prediction at each branch—known as a **static** scheme.

But we can do better by using what recently happened to improve our predictions.

This is particularly important when program execution contains distinct phases, with distinct behaviours.

We therefore move to **dynamic** schemes.

For every branch, we record whether it was taken or not last time it executed (a 1-bit scheme).

Of course, we can't store all branches.

So let's use the low 6 bits of the address to identify branches.

Doing so raises the prospect of *aliasing*: different branches (with different behaviour) map to the same spot in the table.

We might get 85% accuracy with such a scheme.

$$(0.8 + 0.85 \times 0.2) \times 1 \text{ cycle} + (0.15 \times 0.2) \times 20 \text{ cycles} = 1.57 \text{ cycles.}$$

At the cost of more hardware, we get noticeable performance improvements. The DEC EV4 (1992) and MIPS R8000 (1994) used this one-bit scheme.

What if a branch is almost always taken but occasionally not taken (e.g. TTTTTTNTTTT)?

We get penalized twice for that misprediction: once when we mispredict the not taken, and once when we mispredict the next taken.

So, let's store whether a branch is “usually” taken, using a so-called 2-bit saturating counter.

Every time we see a taken branch, we increment the counter for that branch; every time we see a not-taken branch, we decrement.

If the counter is 00 or 01, we predict “not taken”; if it is 10 or 11, we predict “taken”.

With a two-bit counter, we can have fewer entries at the same size, but they'll do better. It would be reasonable to expect 90% accuracy.

$$(0.8 + 0.9 \times 0.2) \times 1 \text{ cycle} + (0.1 \times 0.2) \times 20 \text{ cycles} = 1.38 \text{ cycles.}$$

This was used in a number of chips, from the LLNL S-1 (1977) through the Intel Pentium (1993).

We're still not taking patterns into account. Consider the following for loop.

---

```
for (int i = 0; i < 3; ++i) {  
    // code  
}
```

---

The last three executions of the branch determine the next direction:

TTT => N

TTN => T

TNT => T

NTT => T

Let's store what happened the last few times we were at a particular address—the **branch history**.

From a branch address and history, we derive an index, which points to a table of 2-bit saturating counters.

What's changed from the two-bit scheme is that the history helps determine the index and hence the prediction.

This scheme might give something like 93% accuracy.

$$(0.8 + 0.93 \times 0.2) \times 1 \text{ cycle} + (0.07 \times 0.2) \times 20 \text{ cycles} = 1.27 \text{ cycles.}$$

The Pentium MMX (1996) used a 4-bit global branch history.



The change here is that the CPU keeps a separate history for each branch.

So the branch address determines which branch history gets used.

We concatenate the address and history to get the index, which then points to a 2-bit counter again.

We are starting to encounter diminishing returns, but we might get 94% accuracy:

$$(0.8 + 0.94 \times 0.2) \times 1 \text{ cycle} + (0.06 \times 0.2) \times 20 \text{ cycles} = 1.23 \text{ cycles.}$$

The Pentium Pro (1996), Pentium II (1997) and Pentium III (1999) use this.

Instead of concatenating the address and history, we can xor them.

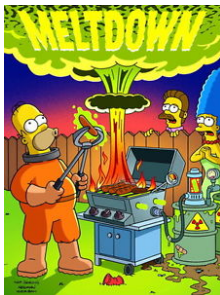
This allows us to use more bits for both the history and address.

This keeps the accuracy the same, but simplifies the design.

We can build (and people have built) more sophisticated predictors.

These predictors could, for instance, better handle aliasing, where different branches/histories map to the same index in the table.

# Cache side-channel attacks





# Vulnerability Example

```
struct array {
    unsigned long length;
    unsigned char data[];
};

struct array *arr1 = ...; /* small array */
struct array *arr2 = ...; /* array of size 0x400 */
/* >0x400 (OUT OF BOUNDS!) */
unsigned long untrusted_offset_from_caller = ...;
if (untrusted_offset_from_caller < arr1->length) {
    unsigned char value = arr1->data[untrusted_offset_from_caller];
    unsigned long index2 = ((value&1)*0x100)+0x200;
    if (index2 < arr2->length) {
        unsigned char value2 = arr2->data[index2];
    }
}
```

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?”

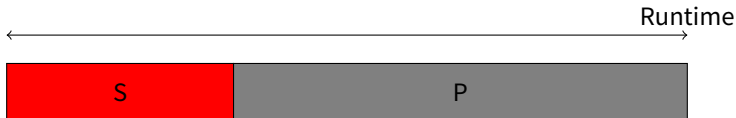
# Visualizing Amdahl's Law

$S$ : fraction of serial runtime in a serial execution.

$P$ : fraction of parallel runtime in a serial execution.

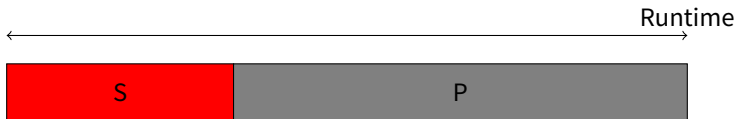
Therefore,  $S + P = 1$ .

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

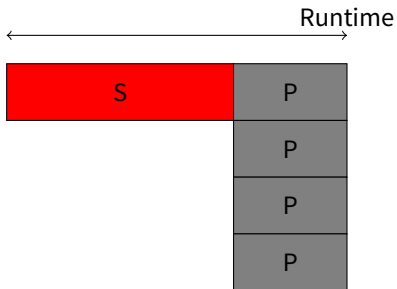




# Visualizing Amdahl's Law



We want to split up the parallel part over 4 processors



$T_s$ : 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  $P$

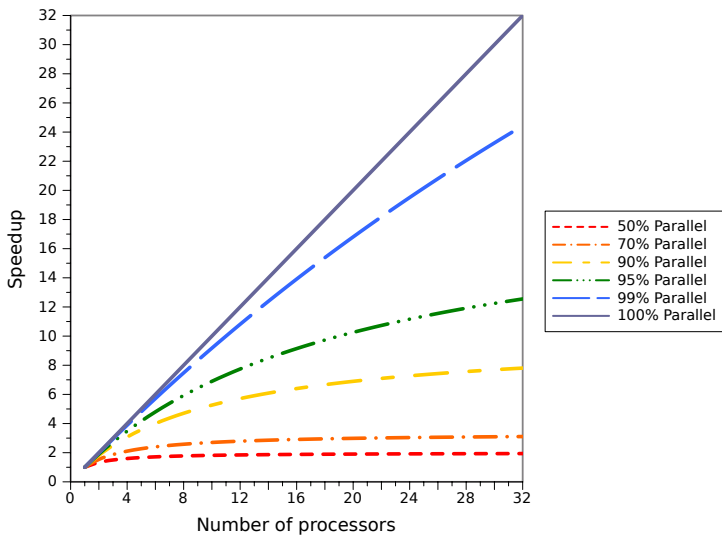
$$T_p = T_s \cdot \left( S + \frac{P}{N} \right)$$

How much faster can we make the program?

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

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

# Fixed-Size Problem Scaling, Varying Fraction of Parallel Code



Replace  $S$  with  $(1 - P)$ :

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

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

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

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%.

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

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).



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.

$f_1, f_2, \dots, f_n$ : fraction of time in part  $n$

$S_{f_1}, S_{f_2}, \dots, S_{f_n}$ : speedup for part  $n$

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

Consider a program with 4 parts in the following scenario:

Part	Fraction of Runtime	Speedup	
		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?

“Plug and chug” the numbers:

### Option 1

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

### Option 2

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

# Empirically estimating parallel speedup P

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

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

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

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

$$T_p = S(n) + P(n) = 1$$

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

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

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

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.)



## 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.