ECE459: Programming for Performance Winter 2018

Lecture 3 — Branch Prediction & Amdahl's Law

Patrick Lam & Jeff Zarnett

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`[1], 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.

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++) {
```

[1] Which I still maintain is a swear word in C.

```
            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.

How does branch prediction work, anyway?

We can write software. The hardware will make it fast. If we understand the hardware, we can understand when it has trouble making our software fast.

You've seen how branch prediction works in ECE 222. However, we'll talk about it today in the context of performance. Notes based on a transcript of a talk by Dan Luu [Luu17].

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

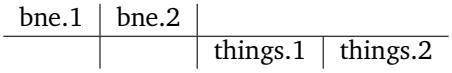
- how branch predictors work—this helps you understand some of the apparent randomness in your execution times, and possibly helps you make your code more predictable; and,
- applying a (straightforward) expected value computation to predict performance.

Let's consider the following assembly code:

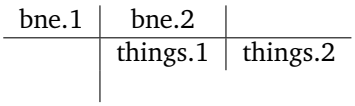
```
branch_if_not_equal x, 0, else_label
// Do stuff
goto end_label
else_label:
// Do things
end_label:
// whatever happens later
```

The `branch` instruction may be followed by either "stuff" or "things". The pipeline needs to know what the next instruction is, for instance to fetch it. But it can't know the next instruction until it almost finishes executing the branch. Let's look at some pictures, assuming a 2-stage pipeline.

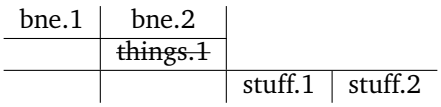
With no prediction, we need to serialize:



Let's predict that "things" gets taken. If our prediction is correct, we save time.



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



Cartoon model. We need to quantify the performance. For the purpose of this lecture, 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):

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

With perfect branch prediction:

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

So we can make our code run 4.8× faster with branch prediction!

Predict taken. What's the simplest possible thing? We can predict that a branch is always taken. (Loop branches, for instance, account for many of the branches in an execution, and are often 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.

Backwards taken, forwards not taken (BTFNT). Let's leverage that observation about loop branches to do better. Loop branches are, by definition, backwards (go back to previous code). So we can design a branch predictor which predicts "taken" for backwards and "not taken" for forwards. The compiler can then use this

information to encode what it thinks about forwards branches (that is, making the not-taken branch the one it thinks is more likely). 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.

Going dynamic: using history for branch prediction. 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.

Once again, let's start with the simplest possible thing. 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.

Two-bit schemes. 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. Saturating means that we don't overflow or underflow. We instead stay at 11 or 00, respectively.

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).

Two-level adaptive, global. 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.

After we take a branch, we add its direction to the history, so that the next lookup gets sent to a different table entry.

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.

Two-level adaptive, local. 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 dminishing 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.

gshare. 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.

Other predictors. 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. But we'll stop here.

Summary of branch prediction. We can summarize as follows. Branch prediction enables pipelining and hence increased performance. We can create a model to estimate just how critical branch prediction is for modern processors. Fortunately, most branches are predictable now. Aliasing (multiple branches mapping to the same entry in a prediction table) can be a problem, but processors are pretty good at dealing with that too.

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

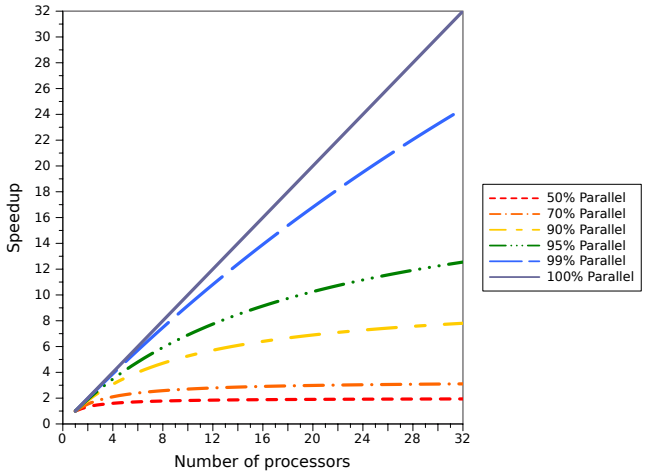
$$\text{max speedup} = \frac{1}{(1-P)},$$

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

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:

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

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.

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 last time, 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[2] 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. URL: `http://blog.man7.org/2012/10/how-much-do-builtinexpect-likely-and.html`.

[Luu17] Dan Luu. A history of branch prediction from 1500000 bc to 1995, 2017. Online; accessed 5-December-2017. URL: `http://danluu.com/branch-prediction/`.

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