ECE459: Programming for Performance Winter 2024

Lecture 7 — CPU Hardware, Branch Prediction

Patrick Lam & Jeff Zarnett *2024-09-10*

Multicore Processors

As I've alluded to earlier, multicore processors came about because clock speeds just aren't going up anymore. We'll discuss technical details today. Each processor *core* executes instructions; a processor with more than one core can therefore simultaneously execute multiple (unrelated) instructions.

Speed and Heat. The whole reason we're here with multicores is that it's impossible to continue to crank up clock rates, because the chips get too hot. Back in the 2000s, before they figured to put thermal sensors, I would occasionally hear about CPUs catching fire. These days, Intel and AMD chips support burst speeds which are faster than normal clock rates, and which can be used until the chips get too hot. So the base rate hasn't gotten much above 3GHz in recent decades, but burst rates are higher.

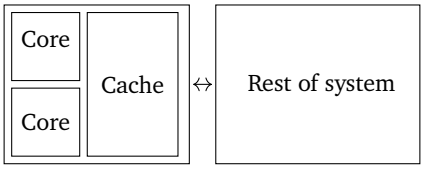
Chips and cores. Multiprocessor (usually SMP, or symmetric multiprocessor) systems have been around for a while. Such systems contain more than one CPU. We can count the number of CPUs by physically looking at the board; each CPU is a discrete physical thing.

Cores, on the other hand, are harder to count. In fact, they look just like distinct CPUs to the operating system:

```
plam@plym:~/courses/p4p/lectures$ cat /proc/cpuinfo
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model  : 23
model name : Pentium(R) Dual-Core  CPU      E6300  @ 2.80GHz
...
processor : 1
vendor_id : GenuineIntel
cpu family : 6
model  : 23
model name : Pentium(R) Dual-Core  CPU      E6300  @ 2.80GHz
```

If you actually opened my computer, though, you'd only find one chip. The chip is pretending to have two *virtual CPUs*, and the operating system can schedule work on each of these CPUs. In general, you can't look at the chip and figure out how many cores it contains.

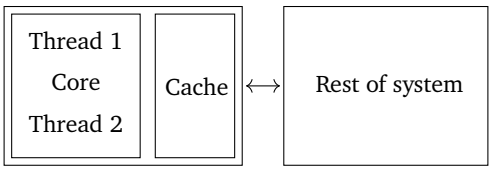
Hardware Designs for Multicores. In terms of the hardware design, cores might share a cache, as in this picture:



(credit: *Multicore Application Programming*, p. 5)

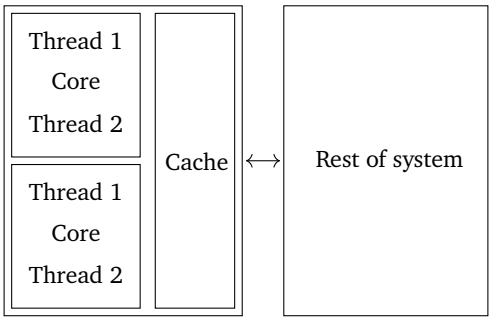
This above Symmetric Multithreading (SMP) design is especially good for the 1:1 threading model. In this case, the design of the cores don't need to change much, but they still need to communicate with each other and the rest of the system.

Or, we can have a design that works well for the N:1 model (typically called hyperthreading):



One would expect that executing two threads on one core might mean that each thread would run more slowly. It depends on the instruction mix. If the threads are trying to access the same resource (e.g. Arithmetic Logic Units), then each thread would run more slowly. If they're doing different things, there's potential for speedup.

Finally, one can both use many cores and many threads on one core, as in the M:N model (also hyperthreading):



Here we have four hardware threads; pairs of threads share hardware resources. Looking at the specs of the AMD Ryzen 5 7600 (Zen 4), I see that it has 6 cores and 12 threads. Everyone does this nowadays. On the Zen 4 chips, each core has its own L1 and L2 caches, while the L3 caches are shared among the cores on the same so-called "Core Complex Die".

Non-SMP systems. The designs we've seen above have been more or less SMP designs; all of the cores are mostly alike. A very non-SMP system is the Cell, which contains a PowerPC main core (the PPE) and 7 Synergistic Processing Elements (SPEs), which are small vector computers. Recent Intel and AMD processors do have cores that provide the same programming interface but have different performance characteristics. Intel chips specifically have P (performance) and E (efficient) cores. Thinking back to bandwidth vs latency, E cores are higher-latency, but they're smaller and so Intel can put more of them on the chip. P cores are good for low-latency. AMD has c-cores which seem to do the same thing as Intel's E cores.

Non-Uniform Memory Access. In SMP systems, all CPUs have approximately the same access time for resources (subject to cache misses). There are also NUMA, or Non-Uniform Memory Access, systems out there. In that case, CPUs can access different resources at different speeds. (Resources goes beyond just memory).

In this case, the operating system should schedule tasks on CPUs which can access resources faster. Since memory is commonly the bottleneck, each CPU has its own memory bank.

Using CMT effectively. Typically, a CPU will expose its hardware threads using virtual CPUs. It is also the responsibility of the scheduler to make sure that tasks are scheduled on the best cores/virtual CPUs.

However, performance varies depending on context. In the above example, two threads running on the same core will most probably run more slowly than two threads running on separate cores, since they'd contend for the same core's resources. Task switches between cores (or CPUs!) are also slow, as they may involve reloading caches.

Solaris "processor sets" enable the operating system to assign processes to specific virtual CPUs, while Linux's "affinity" keeps a process running on the same virtual CPU. Both of these features reduce the number of task switches, and processor sets can help reduce resource contention, along with Solaris's locality groups.[1]

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`[2], 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. It's possible to give `gcc` some hints: we say either something is likely or 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.

From what I can tell, the core Rust team isn't super comfortable with the idea of exposing these kinds of internal-compiler things, but there is an implementation of the likely/unlikely concept. You can sort of use it, but it could break in the future, as an experimental feature. If you want the experimental features enabled, you have to be using nightly build of Rust and to specify the feature at the top of your source file (e.g., `#![feature(core_intrinsics)]`)

Do they work? Here's a sample program to find out. I'll first test it with no hint, then putting `likely()` around the if condition, and then `unlikely()`, and show you the results.

```
fn f(a: i32) -> i32 {
    a
}

fn main() {
    let size = 100000;
    let large_vector = vec![0; size];
    let mut m1 = 0;
    let mut m2 = 0;

    for _j in 0..1000 {
        for k in 0..size {
            if *large_vector.get(k).unwrap() == 0 {
                m1 = f(m1 + 1)
            } else {
                m2 = f(m2 + 1)
            }
        }
    }
```

[1] Gove suggests that locality groups help reduce contention for core resources, but they seem to help more with memory.
[2] Which I still maintain is a swear word in C.

```
    println!("m1 = {}; m2 = {}", m1, m2);
}
```

And the results:

```
No hint at all:
  Time (mean +/- ?):      6.657 s +/-  0.144 s    [User: 6.614 s, System: 0.029 s]
  Range (min ... max):    6.413 s ...  6.905 s    10 runs

Likely:
  Time (mean +/- ?):      6.762 s +/-  0.175 s    [User: 6.729 s, System: 0.028 s]
  Range (min ... max):    6.590 s ...  7.200 s    10 runs

Unlikely:
  Time (mean +/- ?):      6.943 s +/-  0.200 s    [User: 6.893 s, System: 0.033 s]
  Range (min ... max):    6.732 s ...  7.309 s    10 runs
```

Looks like hints don't help very much in this program at all. They made it marginally worse, not better. And getting it wrong comes with a penalty, too. This program might not be the ideal test case for hints, in that there might be a different scenario where the hints have a positive impact. However, we have at least established that hints aren't always a benefit, even if we know we're right. Under a lot of circumstances then, it's probably best just to leave it alone, unless we're really, really, really sure.

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:

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

Let's predict that "things" gets taken. 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

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):

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

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× 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 maps 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 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.

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.

Side-channel attacks

A few years ago, a lot happened in terms of exploiting the hardware of CPU architectures to get access to privileged data, and unfortunately these things have performance implications!

Cache Attacks

In early 2018, the Spectre [KGG+18] and Meltdown [LSG+18] attacks were disclosed. These attacks leverage performance features of modern CPUs to break process isolation guarantees—in principle, a process shouldn't be able to read memory that belongs to the kernel or to other processes.

The concept of cache side-channel attacks has been known for a while. If an attacker can get some memory loaded into the cache, then it can extract that memory using a cache side-channel attack.

Spectre and Meltdown can cause privileged memory to be loaded into the cache, and then extracted using a cache side-channel attack. We'll talk about Spectre (variant 2), since it attacks branch prediction in particular. My explanation follows [Mas18] by Jon Masters of RedHat. However, you should now have enough background to follow the Google Project Zero description at [Hor18].

We know that at a branch, the CPU will start speculatively executing code at the inferred target of the branch. To exploit this vulnerability, the attack code convinces the CPU to speculatively execute code of its choice. The speculatively-executed attack code reads secret data (e.g. from the hypervisor kernel) and puts it in the cache. Once the data is in the cache, the attack proceeds by extracting the data from the cache.

Unfortunately, the only mitigation thus far involved additional security measures (in hardware and software) that unfortunately result in lower performance in program execution.

Hyperthreading attacks

Multiprocessor (multicore) processors have some hardware that tries to keep the data consistent between different pipelines and caches (as we saw in the video). More processors, more threads means more work is necessary to keep these things in order. We will discuss cache coherence soon, but about hyperthreading... it turns out this is vulnerable too.

Remember that in hyperthreading, two threads are sharing the same execution core. That means they have hardware in common. Because of this, a thread can figure out what the other thread is doing by noticing its cache accesses and by timing how long it takes to complete operations. This is like sitting next to someone who's taking a multiple choice exam and noticing what answers they are choosing by how long it takes them to move their pencil down the page to fill in the correct circle. Yes, you have to be running on the same CPU as the victim, but still... Yikes!

Researchers discovered and published a paper [ABuH+18] detailing the attack and showing a practical implementation of it. In the practical example, a 384-bit secret key is (over time) completely stolen by another process. It seems likely that this will lead in the long term to slowdowns of existing hardware as Operating System patches will need to prevent threads from different processes from using the same core... And possibly the only long term solution is to not use hyperthreading at all... the performance implications of which are both obvious and significant.

References

[ABuH+18] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. Cryptology ePrint Archive, Report 2018/1060, 2018. URL: https://eprint.iacr.org/2018/1060.

[Hor18] Jann Horn. Reading privileged memory with a side-channel, January 2018. Online; accessed 10-January-2018. URL: https://googleprojectzero.blogspot.ca/2018/01/reading-privileged-memory-with-side.html.

[KGG+18] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, January 2018. arXiv:1801.01203.

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *ArXiv e-prints*, January 2018. arXiv:1801.01207.

[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/.

[Mas18] Jon Masters. What are Meltdown and Spectre? here's what you need to know, January 2018. Online; accessed 10-January-2018. URL: https://www.redhat.com/en/blog/what-are-meltdown-and-spectre-here%E2%80%99s-what-you-need-know.