Lecture 3 — CPU Hardware, Branch Prediction

Patrick Lam & Jeff Zarnett
patrick.lam@uwaterloo.ca jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

December 18, 2019

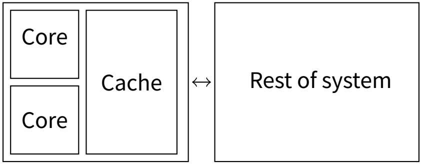
SMP (Symmetric Multiprocessing)

Identical processors or cores, which:

- are interconnected, usually using buses; and
- share main memory.

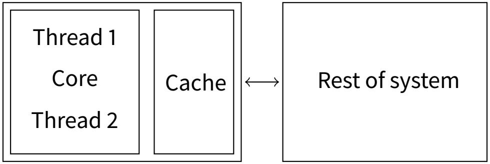
- SMP is most common type of multiprocessing system.

Example of an SMP System



- Each core can execute a different thread
- Shared memory quickly becomes the bottleneck

Executing 2 Threads on a Single Core



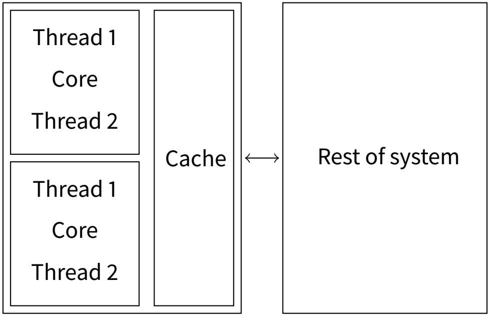
On a single core, must context switch between threads:

- every *N* cycles; or
- wait until cache miss, or another long event

Resources may be unused during execution.

Why not take advantage of this?

Executing M Threads on a N Cores



Here’s a Chip Multithreading example.

UltraSPARC T2 has 8 cores, each of which supports 8 threads. All of the cores share a common level 2 cache.

SMT (Simultaneous Multithreading)

Use idle CPU resources (may be calculating or waiting for memory) to execute another task.

Cannot improve performance if shared resources are the bottleneck.

Issue instructions for each thread per cycle.

To the OS, it looks a lot like SMP, but gives only up to 30% performance improvement.

Intel implementation: Hyper-Threading.

Example: Non-SMP system

PlayStation 3 contains a Cell processor:

- PowerPC main core (Power Processing Element, or “PPE”)
- 7 Synergistic Processing Elements (“SPE”s): small vector computers.

NUMA (Non-Uniform Memory Access)

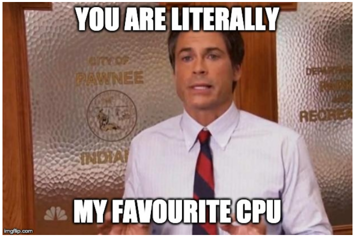
In SMP, all CPUs have uniform (the same) access time for resources.

For NUMA, CPUs can access different resources faster (resources: not just memory).

Schedule tasks on CPUs which access resources faster.

Since memory is commonly the bottleneck, each CPU has its own memory bank.

Processor Affinity



Each task (process/thread) can be associated with a set of processors.

Useful to take advantage of existing caches (either from the last time the task ran or task uses the same data).

Hyper-Threading is an example of complete affinity for both threads on the same core.

Often better to use a different processor if current set busy.

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.

Branch Prediction Hints

`gcc` allows you to give branch prediction hints by calling this builtin function:

```
long __builtin_expect (long exp, long c)
```

The expected result is that exp equals c.

Compiler reorders code & tells CPU the prediction.

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.

More About Branch Prediction

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
```

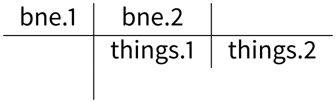
No Prediction

With no prediction, we need to serialize:

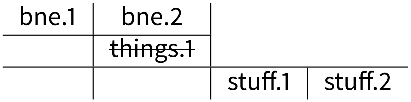
bne.1	bne.2		
		things.1	things.2

Predict Things

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.

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.

Prediction Quantification

$V = \frac{1}{3}\pi r^2 \cdot h$
h
r
30° 45° 60°
sin
cos
tan
$y = ax^2 + bx + c$
$(x_1, x_2) = \frac{-b \pm \Delta}{2a}$

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!

Strategy 1: Predict Taken

We can predict that a branch is always taken.

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

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

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

BTFNT - Dynamite

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.

Dynamic

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.

Remember Your History

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 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.

Two-Bit Scheme

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.

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

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

Two-Bit 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.

Two-Bit Adaptive, Global

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.

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

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.

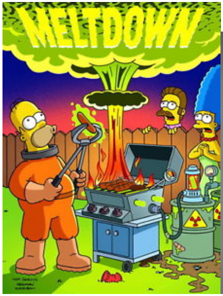
Hacking Time... Hacking Too Much Time

We Are Under Attack!

There's been a lot happening lately in terms of exploiting the hardware of CPU architectures to get access to privileged data.

Unfortunately these things have performance implications!

Cache side-channel attacks



Cache side-channel attacks

Cache Side-Channel Attacks

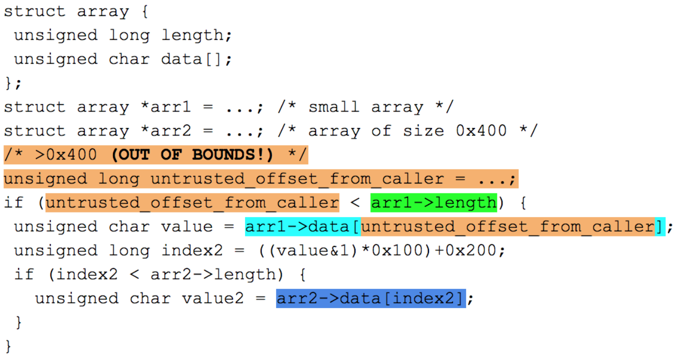
These attacks leverage performance features of modern CPUs to break process isolation guarantees.

In principle, a process shouldn’t be able to read memory that belongs to the kernel or to other processes.

Spectre and Meltdown can cause privileged memory to be loaded into the cache, and then extracted using a cache side-channel attack.

Mitigation: more isolation, lower performance.

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];
 }
}
```

(from `https://hk.saowen.com/a/81873c7b149c0993836e8a4fa4f879b4178085a3ad14c09a5f22f5a9c76373ca`)

Hyperthreading Attacks

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.

Attack name: PortSmash

Hyperthreading Attacks

In the practical example, a 384-bit secret key is (over time) completely stolen by another process.

Mitigation: prevent threads from different processes from using the same core...

Possibly the only long term solution is to not use hyperthreading at all...

The performance implications of which are obvious & significant!