Lecture 2 — Modern Processors

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

December 20, 2018

Remember the classic von Neumann machine architecture.

A program is comprised of both instructions and data, both of which are stored in the same memory.

A program executes sequentially, one statement at a time, one after another.

That is not really how computers work, at least not anymore, but it is an abstraction we still maybe find useful when it comes to algorithm analysis.
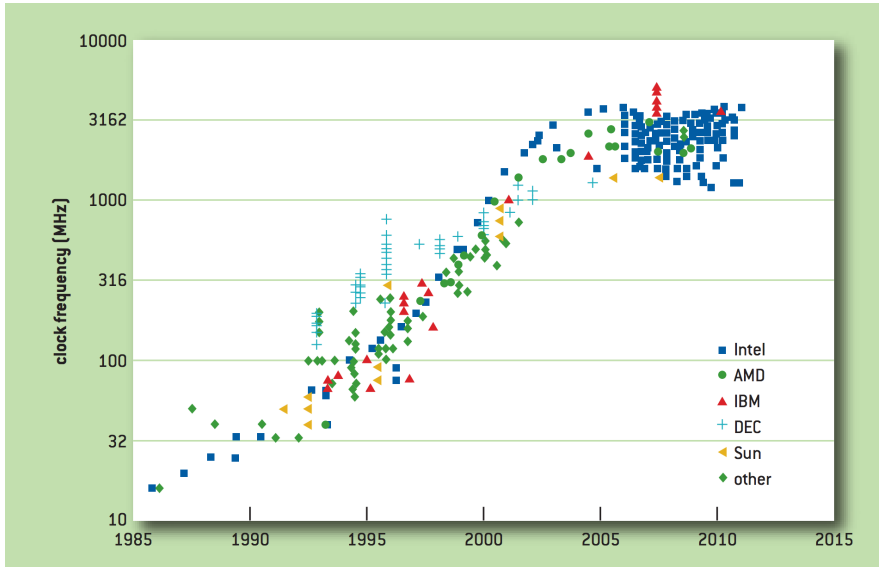
The x86 processor is a Complex Instruction Set Computing (CISC) processor. In other words, there are a lot of assembly instructions.

But why? This was intended for your convenience as a programmer!

So the hardware people thought they were doing everyone a favour.

These are easy to program in, from the way the assembly programmer thinks, but hard to implement and hard to pipeline.

For a lot of CISC machines, the Cycles Per Instruction (CPI) varied, something like 4-10 cycles to complete any instruction, but at least it was predictable.

Program performance was basically the number of page faults times the amount of time it takes to read from disk, plus the instruction execution time.

Thus the optimization goal is: minimize page faults.

Page fault count is relatively easy to measure and there are some things we can do to reduce the number of page faults.

If you were working with an embedded system with no disk (or at least no page faults) then the optimization goal is minimize instruction count.

Between 1990 and 2005 we got some impressive scaling on CPU frequency.

We got Reduced Instruction Set Computing (RISC) CPUs.

Simpler processors are easier to scale than complex ones, and simpler instructions mean fewer cycles per instruction.

That also means we can have more pipelining.

The tradeoff is that RISC CPUs are much harder to program in assembly directly, so compilers had to do the work.

Consider delay slots: an instruction after a branch is always executed or worse, the result of a computation is not available to the next instruction.

In these cases the "simple" solution is to put a NOP (do-nothing) instruction in.

Good compilers (and programmers) can rearrange instructions, hopefully, to make this work without wasting time.

Also: memory got cheaper, so we have more of it, so page faults occurred less and less frequently.

But then, as we have seen, we hit the (power) wall.

And you might think, well, if I run into a wall, I can just go around it.
There must be other ways to advance!

And there are, except, we hit three other walls too, and now we are surrounded.

What are these other three seemingly-insurmountable barriers?

Mr. Gorbachev, Tear Down This Wall!

Instruction level parallelism (ILP) is getting close to the limit of what we can do.

We can predict branches with a certain accuracy but if we have already got 95% efficiency, our maximum improvement is 5%.

The speed of memory advances has not kept up with the advances in CPUs.

So now we have moved from the era of runtime being dominated by page faults to the era of runtime being dominated by cache misses.

Doubling, say, level one cache (at great expense) does not double the speed of the program; it may speed it up by a small amount at most.

The final wall is the universal speed limit: the speed of light.

The more complex the CPU is, the longer the path any signal may have to travel to get from A to B.

This is limited, most practically, by the speed of light, and thus far, nobody has invented a way to get around this universal speed limit.

The idea with ILP is not having more cycles to work with, but instead, doing more in each clock cycle. And there's a lot of clever ideas.

Pipelining: you may have heard a bit about his already, especially so if you have (or are currently) taking a CPU architecture course.

To complete an instruction there are five basic steps:

1. Fetch the instruction from memory
2. Decode the instruction
3. Fetch needed operands
4. Perform the operation, and
5. Write the result

Example: ADD R1, R2

Thus even a simple instruction takes more than one clock cycle, but the good news is that the stages can overlap:

Clock Cycle	1	2	3	4	5	6	7	8	9	10
	IF 1									
		DC 1								
			OP 1							
				EX 1						
					WB 1					
						IF 2				
							DC 2			
								OP 2		
									EX 2	
										WB 2
	IF 1									
		DC 1								
		IF 2	OP 1							
			DC 2	EX 1						
				OP 2	WB 1					
					EX 2					
						WB 2				

There are pipeline hazards: sometimes you need the result of a previous step before you can go on.

These prevent us from reaching the theoretical maximum of one instruction completed per clock cycle.

There are other hazards; we may have conflicts over certain CPU resources or fetch may be unable to identify the next instruction because of a branch.

In the worst case, if we've mispredicted a branch, we have to flush the pipeline.

In that case, some extra work was done that was not necessary...

If we do a load from memory, and we are lucky, it is found in the fastest cache and is available in perhaps 2-3 clock cycles.

If we must go to memory, it may be 200-300 cycles.

If it is in level 2 or level 3 cache it will be somewhere in between.

The key idea, though, is if we are trying to put something in register R7, it will take time until that value is actually present in that register.

The simplest approach is to just wait around until that value has arrived, and then go on to the next instruction.

That works, but we can do better.

That better idea is: continue executing until R7 is used somewhere.

This allows us to get some useful work done in the meantime.

Hardware keeps track of the fact that the register is not quite ready yet, but the work can get done in what is called the "miss shadow".

It's possible to have more than one load in flight at a time, two or more can be done in various CPU architectures, but it is of course hardware dependant.

Branch prediction has come up already, but if we have a load followed by a compare used as a branch, we can then, well, guess.

If we are wrong, there is the need to cleanup. But the good news is that branch prediction is usually right most of the time, perhaps 95% or more.

Dual Issue

Another nice thing that the hardware can do for us is "dual issue" instructions.

If we have two consecutive instructions that both take the same amount of time, use unrelated registers, and don't consume two of the same resource.

Example: `ADD R1, 16` and `CMP R2, 0`.

In an embedded system, you may be interested in ensuring that this happens during a computationally intensive loop, such as encoding/decoding of media.

If programmed correctly, you can be sure you get dual issue on every cycle.

Then a group of things that somewhat go together: register renaming, branch prediction, speculation, and Out-of-Order (O-O-O) Execution.

Register renaming works on a fairly simple principle: an assembly instruction says to read from register R4.

Behind the scenes inside the processor, it is mapped to a physical register (let's say RA for the purpose of the example).

Consider the following assembly instructions:

```
MOV R2, R7 + 32
ADD R1, R2
MOV R2, R9 + 64
ADD R3, R2
```

This has a certain synergy with the branch prediction.

If we predict a branch, we can do speculative changes into one set of registers while we keep the "old" register values around too.

When we figure out whether the branch prediction is correct, we can then get rid of the ones we don't need.

So we get better recovery if there is a misprediction.

Most importantly, it allows us to get past a cache miss and keep going.

The goal here is to run until we can start the next cache miss, because the sooner that starts the sooner it's over, and the faster the program executes.

```
ld rax, rbx+16
add rbx, 16
cmp rax, 0
jeq null_chk
st rbx-16, rcx
ld rcx, rdx
ld rax, rax+8
```

It's about how hard you can get hit & keep moving forward

There are seven operations we were trying to do here with two cache misses.

The cache misses complete in cycles 300 and 304 (maybe 302 if we have dual issue), so in total we complete 7 operations in about 305 cycles.

All the trickery and cleverness got us to that second miss which means we complete in 305.

If we did not manage that, it would take about 600 cycles to complete it all.

So we did double performance, even though in this example our overall performance was very, very low.

Intel Itanium

For years Intel was trying to push its Itanium processors.

The goal of these was to find static (compile-time) parallelism: if a machine has infinite registers, can speculate infinitely, etc, the program gets sped up.

Run all possibilities in parallel and at the end figure out which is right.

Unfortunately it didn't work out very well because this requires the right kind of program and an super smart compiler.

Oh yes, and infinite registers requires infinite space as well as infinite money.

So instead the quest has turned to how to get better performance out of x86...

The x86 approach tries to maximize dynamic (run-time) parallelism.

This has been done incrementally, with more pipelining, re-order buffers, adding more functional units, and so on.

Cache miss rates and branch mispredicts continue to dominate performance, even though the rates are very low, because a miss costs so much.

Something like 90-99% of the transistors on a modern x86 chip are spent in cache.

In spite of the extreme complexity of the decode logic that allows multiple parallel decodes of all the weird and wacky instructions of the x86...

Pushing cache to the biggest size it can be is so important because it prevents the performance hit of going to memory.

The image below (from Sun World Wide Analyst Conference in 2003) is obviously a bit dated and rather insulting:
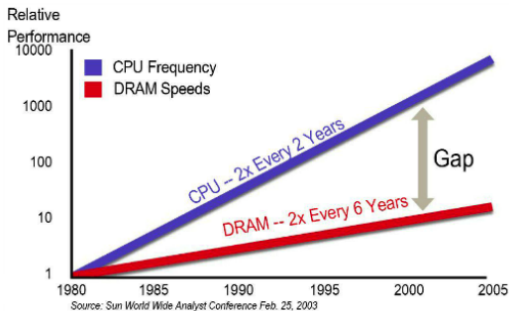
DRAM is, however, not the only kind of memory.

There is SRAM (Static RAM) which is fast but expensive, the kind of stuff that goes on the CPU die, and it is six transistors per bit.

Compare against DRAM which is much cheaper, but slow: one transistor and one capacitor per bit.

Improvements in DRAM have not really improved latency but have improved bandwidth.

DDR means there are two transfers per cycle, but it still takes significant time to get any data out.

And DRAM needs occasional refreshes (capacitors...) so sometimes we have to wait for that.

In the Operating Systems course you probably learned that disk is the slowest thing and the limiting factor.

That's true, as Obi-Wan Kenobi would say, from a certain point of view.

Now that we live in the world of Solid State Drives (SSDs), "disk" reads are about as fast as memory reads and memory reads are the rate-limiting step.

More is the new more, orange is the new black, and memory is the new disk.

To get memory access speed up there are things we can do, like relax coherency constraints, more synchronization through locks...

All of which we will come back to in some upcoming lectures.

If we want to get better performance, we need to figure out where time is going.

For that we will have the subject of profiling, which comes up later.

If we can track down where our cache misses are occurring, maybe, just maybe, we can do something about it.

If you'd much prefer to hear this material from someone else, the source for this lecture is:

```
http://www.infoq.com/presentations/
click-crash-course-modern-hardware
```

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.

Miss: we must load the page from memory, a comparatively slow operation.

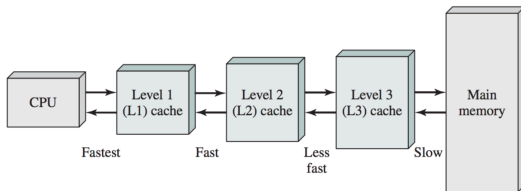The %age of the time that a page is found in the cache is called the hit ratio.

The effective access time is therefore computed as:

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

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.

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:

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

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:

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

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.

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.

Summary: misses are not just expensive, they hurt performance the most.