

Lecture 2 — A Crash Course in Modern Hardware

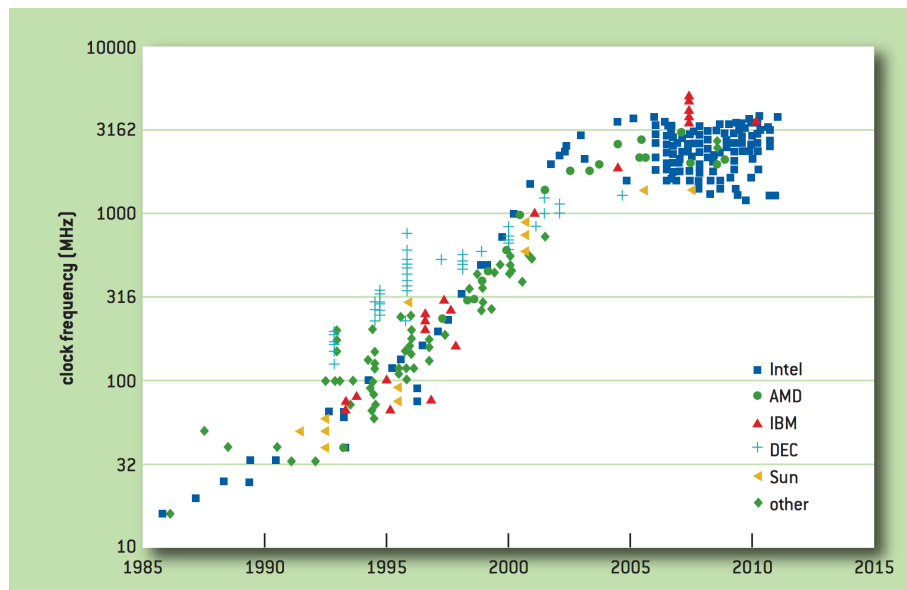
Jeff Zarnett

Modern Processors

It's critical to understand what's going on with the hardware if we want to write good programs. This lecture is based off the talk by Cliff Click [CG10].

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.

Consider this graph of CPU clock speed (frequency) over time from [DKM⁺12]:



Clearly there is an area in which frequency scaling was effective. Next year's CPU would have a higher clock speed, and higher clock speed means more cycles per second, and more cycles per second means more work is done in a given second, and that means better performance. Except, we hit the wall: clock speeds stop getting faster around 2005, stopping at around 3 GHz. Speeding them up beyond this would take, well, more voltage which means more power and more heat, and more heat means higher failure/error rates, and more cooling, and the cooling takes power too, and all that waste heat, well, it will eventually, at the end of this chain, make polar bears sad.

Digression: if we look at the x86 processor, one with which everyone is probably at least *passingly* familiar, it 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: if you were going to write assembly, wouldn't it be nice to have a sine function that takes one argument instead of having to grind out (or copy-paste) the calculation of a sine routine every single time you needed it? 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. Every time, no matter what, it takes the same number of cycles. Program performance was basically the number of page faults (disk accesses) times the amount of time it takes to read from disk, plus the instruction execution time (which is generally small compared to page fault service times)¹. 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; optimize our data access patterns, change how we pack the data, et cetera. 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 really impressive scaling on CPU frequency. This was caused by a few factors. The first is the advent of the Reduced Instruction Set Computing (RISC) CPU: 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. The example in [CG10] is 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. And another thing: memory got cheaper, so we have more of it, so page faults occurred less and less frequently and that’s really something.

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 surrounded². What are these other three seemingly-insurmountable barriers?

The first is 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, no matter how much time and effort and money is invested into improving the branch prediction routine we get maximally a 5% increase in branch prediction accuracy which translates into a very small speedup to the execution when we consider just how often a misprediction is the cause of the problem (5% of 5% is very small... just making up numbers).

The speed of memory advances has not at all kept up with the advances in CPU technology, 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. Adding more cache isn’t a perfect solution though, and 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 (curse you Einstein!). 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 (but we are working on it, and according to Star Trek, should have this sorted out by 2063 or so).

But let’s go back to the subject of ILP. Branch prediction and pipelining have been touched upon but there is so much more to it. 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 this already, especially so if you are taking ECE 429 this term as well. 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. So to do an instruction like ADD R1, R2, we need to fetch the instruction, decode it and figure out what is to be done, read the values from R1 and R2, do the addition, and then write the result to R1. Thus even a simple instruction takes more than one clock cycle, but the good news is that the stages can overlap:

¹For further discussion about this, see the ECE 254 notes about page faults and caching and disk read times.

²“He is intelligent, but not experienced. His pattern indicates two dimensional thinking.” - Spock, *Star Trek II: The Wrath of Khan*


```

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

```

Under normal circumstances, we cannot do instruction 3 until instruction 2 has been completed because we need the value of R2 that was put in there (taken from memory somewhere) to be added to R1. Except, with register renaming, behind the scenes the first two instructions may replace R2 with RK and the second pair of instructions have R2 replaced with RY and these things can take place in parallel, or without a stall, at the very least.

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: the originals if predicted correctly, and the new values otherwise. So we get better recovery if there is a misprediction. Actually, I bet students wish they could do this: write down both answers to a question and let the TA pick the correct one at the end...

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, ultimately. A quick example from the presentation demonstrates this, in x86 assembly [CG10]:

```

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

```

Let’s assume rax (register A) is not available and the first instruction is a cache miss and it will take 300 cycles to get it. So we carry on. Then the instruction next is unrelated so we can go on. It doesn’t matter that rbx is read in the first two instructions, because we can do register renaming so that internally it’s reading from register 4 and writing to register 12. The compare cannot proceed because the first load’s value is not available, but it will be queued up to be done later. Then we have a branch, but we don’t know what the flags are, because we can’t compare, because the load isn’t done. The branch predictor will say in this example that we will not take the branch and go on to the store instruction (st). The store is speculative because the branch may have been taken (so it goes in a store buffer temporarily and not to L1 cache, until we are sure it’s okay to let it go to memory). We have now made it to the second last line and we have another unrelated cache miss, and we have two cache misses running in parallel (the second of which is speculative). Finally, the last instruction has to wait for the first instruction to fill the register rax and we can no longer speculate.

To summarize: 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.

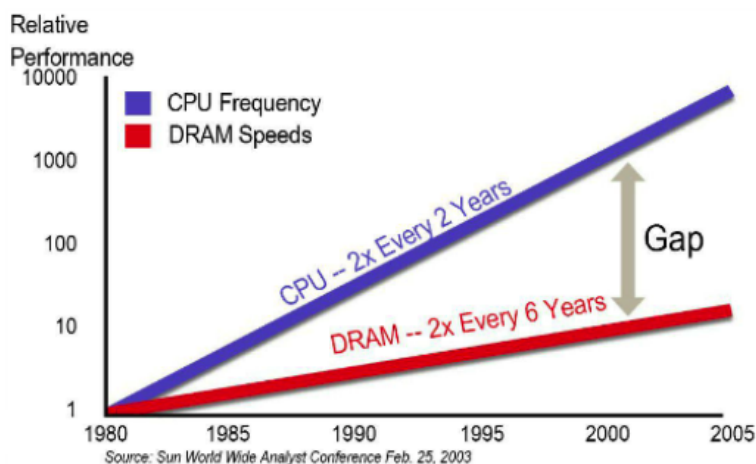
For years Intel was trying to push its Itanium processors (which were so unsuccessful they got the nickname “Itanic”. Ouch). The goal of these was to find static (compile-time) parallelism: if a machine has infinite registers, can speculate infinitely, etc, the program gets sped up. Run all possibilities in parallel and at the end figure out which is right (wasn’t this a Nicolas Cage movie?). 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 we can 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. But the walls are still there: cache miss rates

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

According to [CG10] 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 but this is very instructive as to the trend:



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 (Dual Data Rate... not Dance Dance Revolution) 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 in the system. 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 in some later lectures. If we can track down where our cache misses are occurring, maybe, just maybe, we can do something about it.

References

- [CG10] Cliff Click and Brian Goetz. A crash course in modern hardware, 2010. Online; accessed 27-December-2016. URL: <https://www.infoq.com/presentations/click-crash-course-modern-hardware>.
- [DKM⁺12] Andrew Danowitz, Kyle Kelley, James Mao, John P. Stevenson, and Mark Horowitz. Cpu db: Recording microprocessor history. *Queue*, 10(4):10:10–10:27, April 2012. URL: <http://doi.acm.org/10.1145/2181796.2181798>.