# Performance considerations

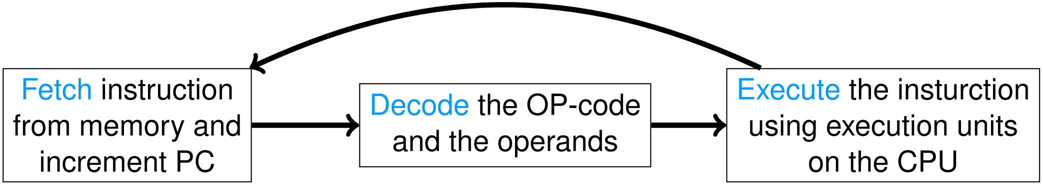
Saska Dönges

# AGENDA

1. Instruction cycle
2. Memory hierarchy
3. Division
4. Loop unrolling
5. Are trees bad actually?
6. Templates
7. How to know what to do

Interrupt to comment or ask questions! You will get points!

# FETCH–DECODE–EXECUTE -LOOP[†]



Fetch instruction from memory and increment PC

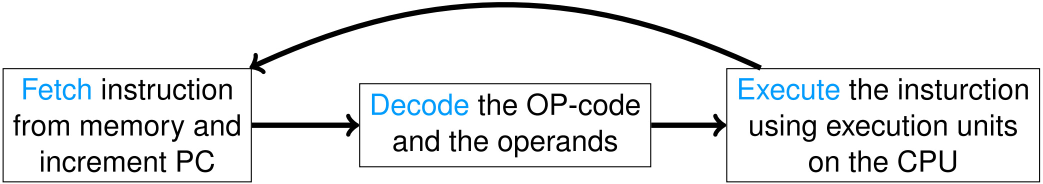
Decode the OP-code and the operands

Execute the insturction using execution units on the CPU

Why is this something you should know as a systems programmer?

[†] `https://en.wikipedia.org/wiki/Instruction_cycle`

# FETCH–DECODE–EXECUTE -LOOP[†]


Fetch instruction from memory and increment PC
Decode the OP-code and the operands
Execute the insturction using execution units on the CPU

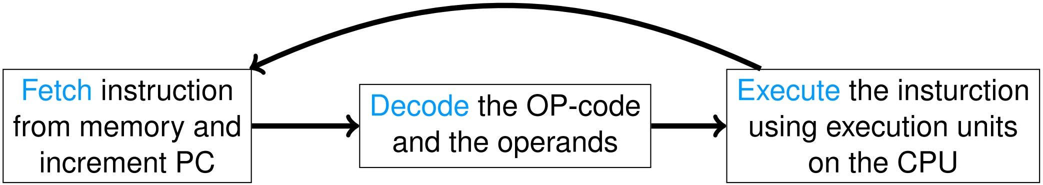
Why is this something you should know as a systems programmer?

Because all (single thread) computation “behaves” like this. (Unless there are hardware bugs [*])

[*] https://en.wikipedia.org/wiki/Spectre_(security_vulnerability)

[†] https://en.wikipedia.org/wiki/Instruction_cycle

# FETCH–DECODE–EXECUTE -LOOP



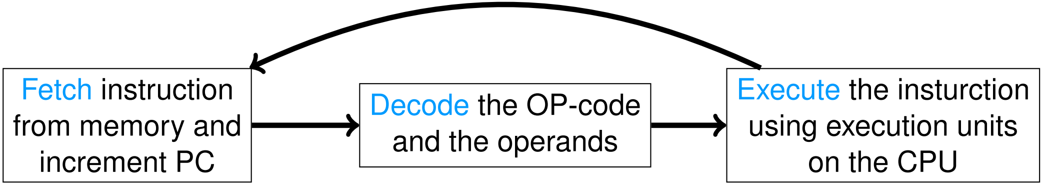
Fetch instruction from memory and increment PC

Decode the OP-code and the operands

Execute the insturction using execution units on the CPU

Why is not something you should concern yourself with?

# Fetch–Decode–Execute -loop



Fetch instruction from memory and increment PC

Decode the OP-code and the operands

Execute the insturction using execution units on the CPU

Why is not something you should concern yourself with?

Because no modern systems actually work like this.

# WHY THE INSTRUCTION LOOP IS A LIE

# Why the instruction loop is a lie

Out-of-order execution
`https://en.wikipedia.org/wiki/Out-of-order_execution`

Pipelining `https://en.wikipedia.org/wiki/Instruction_pipelining`

Speculative execution
`https://en.wikipedia.org/wiki/Speculative_execution`

Data dependence `https://en.wikipedia.org/wiki/Data_dependency`

Cache misses `https://en.wikipedia.org/wiki/CPU_cache`

Dark silicon `https://en.wikipedia.org/wiki/Dark_silicon`

Variable CPI `https://en.wikipedia.org/wiki/Cycles_per_instruction`

# MEMORY HIERARCHY

What is it?

What are the parts?

How fast are the parts?

Why should you care?

# TEEMU'S CHEESECAKE[‡] [§]

Relative speed differences within the memory hierarchy, compared to the differences in speed of adding the cheese to you mixing bowl when making cheesecake.

Data in register: Cheese already in mixing bowl ⇔ Instant.

Data in L1 cache: Cheese on spatula, ready to drop in ⇔ Imperceptible slowdown.

Data in L2 cache: Cheese on table next to bowl ⇔ Often irrelevant an unavoidable.

Data in LL cache: Chesee in fridge ⇔ Slightly annoying but you have to get it at some point.

[‡] `https://www.cs.helsinki.fi/u/kerola/tikra/s2001/luennot/ch4.3_p2.pdf`

[§] `https://en.wikipedia.org/wiki/Memory_hierarchy`

# TEEMU'S CHEESECAKE

Data in RAM: You forgot to buy cheese ⇔ Put stuff away and continue when you get back from the store.

Data on SSD: Global supply chain shortage ⇔ Cheese will be available in some days.

Data on Spinning disk: Cheese on the ISS ⇔ Talk to NASA, perhaps available in some months.

Access requires human intervention: Cheese on the moon ⇔ No planned missions, technically possible in a couple of years.

Fortunately computers are very patient.

# WHY CARE ABOUT THE CHEESECAKE?

Unless your computation happens almost entirely in registers and low level cache, memory access latencies will probably have a massive effect on practical performance!

Fortunately, modern hardware is amazing at hiding memory latency, if your memory access patterns are “predictable”.

Unfortunately only the engineers at Intel and AMD, know exactly what “predictable” means.

This should be a central topic at the case study session next week.

# DIVISION

Is division bad?

Should you care?

When?

Why? / Why not?

# DIVISION BY CONSTANT IS OK

If you use an optimizing compiler, any division by a constant will be fine.

```
uint32_t div_by(uint32_t x) {
    return x / 310;
}
```

```
mov     eax, 3546811703
mov     ecx, edi
imul    rax, rcx
shr     rax, 40
ret
```

# DIVISION BY SOMETHING UNKNOWN SUCKS

If you just have to divide, you have to pay the price.

```
std::pair<uint64_t, uint64_t> div_by(uint64_t x, uint64_t a) {
    return {x / a, x % a};
}
```

Upside is that `mod` is free if you `div`.

```
mov     rax, rdi
xor     edx, edx
div     rsi
ret
```

# PRACTICAL TAKE-AWAYS

Use `const` when you can, or otherwise make sure that the compiler knows the divisor is a constant.

More generally: Only optimize manually when it actually matters.

For example: Don’t work a lot to eliminate branching that the compiler would eliminate anyway.

`if (a) x = 1; else x = 2;` VS. `x = a ? 1 : 2;`.

# SHOULD YOU DO LOOP UNROLLING MANUALLY

Probably not...

Why?

What should you do then?

# COMPILERS DO VERY AGGRESSIVE LOOP UNROLLING

Check if your loop gets unrolled. Unrolling has high overhead but high througput.

If it does and it shouldn't. Try to stop it. `[[assume]]` and `[[unreachable]]` perhaps?

If it should and it doesn't. Figure out why and fix it. Perhaps eliminate data dependence or manually split loops.

Typically works very well for common programming patterns, and is immensely difficult when it doesn't.

# TREES?

Are trees bad?

Why?

Why not?

# REFERENCE-BASED DATA STRUCTURES

Reference-based data structures have a risk of spreading out over application memory, in a way that may as well be random.

Unless your use case specifically can't avoid random memory access, a purely reference-based structure is probably not for you.

Luckily, a lot of tree structures can be made more blocky, and thus more memory efficient[¶].

[¶] https://en.wikipedia.org/wiki/B-tree

# TEMPLATES, NOT JUST GENERICS!

Are they amazing?

Are they a 0-cost abstration[‖]?

Should you do everything you possibly can with templates?

[‖] https://youtu.be/rHIkrotSwcc

# TEMPLATE EVERYTHING?

From a purely PfP point of view... Kinda yes... If you don’t mind hard-to-maintain code... And as long as you don’t run into insurmountable issue w.r.t compile time or binary size.

Templating kinda either eliminates branching or moves it lower on the call stack.

So the main benefit is less issues with branch missprediction? Right?

# TEMPLATE EVERYTHING?

From a purely PfP point of view... Kinda yes... If you don't mind hard-to-maintain code... And as long as you don't run into insurmountable issue w.r.t compile time or binary size.

Templating kinda either eliminates branching or moves it lower on the call stack.

So the main benefit is less issues with branch missprediction? Right?

Not really. Well written code will have very few unavoidable branch misspredictions anyway.

But templating can improve the working of the instruction cache massively, which is often a more important thing, than a simple missed branch.

# How should I know what is or isn't fast in practice?

# HOW SHOULD I KNOW WHAT IS OR ISN’T FAST IN PRACTICE?

Figure it out!

Look it up online.

See what compiler produces.

Run profiling (cachegrind, grpof...).

Run actual michrobenchmarks.

Remain skeptical of everything.