Effizientes Programmieren in C, C++ und Rust

Intro and C Basics

Colin Bretl, Nikolai Maas, Peter Maucher | 24.10.2024

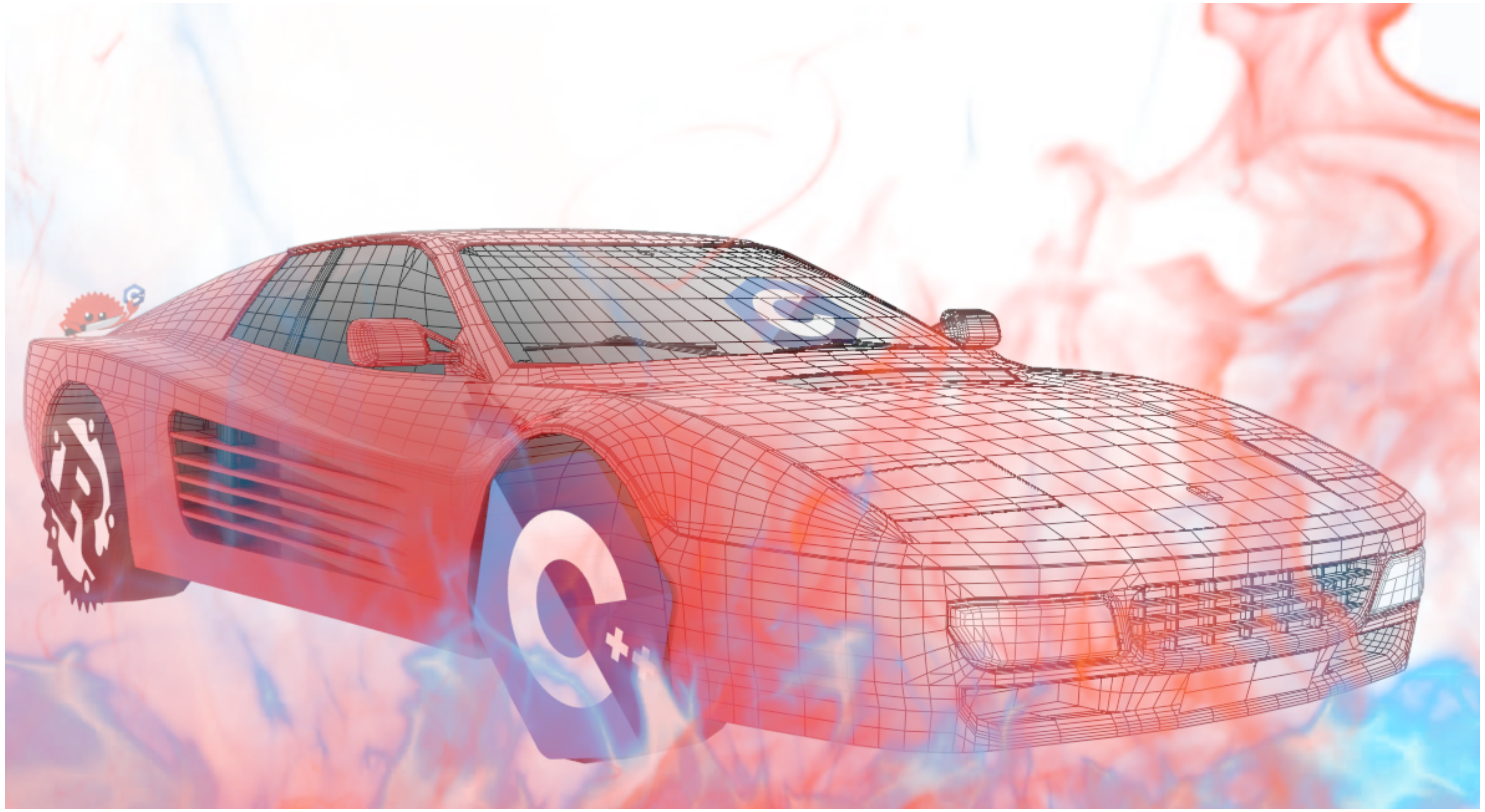


Table of Contents

Program Efficiently?

Why program efficiently?

- Be *FAST*
- But why is fast important?

Why fast?

Discuss!

ENABLE SOLUTIONS

- Solve large datasets (e.g., CERN)
- Overcome hardware limits (your laptop)

SAVE MONEY

- Less hardware
- Less CPU time (AWS, ...)
- Less power (environmental impact)
- Time of employees

IMPROVE (CUSTOMER) EXPERIENCE

- Shorter response time
- Less wait time
- All of the above

How to be fast?

OTHER LECTURES

- Algorithms ⇒ (mostly) other lectures
- Data structures ⇒ (somewhat) other lectures
- Design for Performance ⇒ (somewhat) other lectures
- Parallelization ⇒ (mostly) other lectures

THIS LECTURE

- Data layout
- Optimizations
  - Machine characteristics
    - Caches
    - Prefetchers
    - ...
  - Code structure
    - Loop unrolling
    - Dynamic Dispatch vs Generic Code
    - ...
- Profiling
- Data modelling

Prior Knowledge

We assume knowledge in:

- *Operating system* ↑
- *Basic algorithms* ↑
- *Computer architecture* ↑

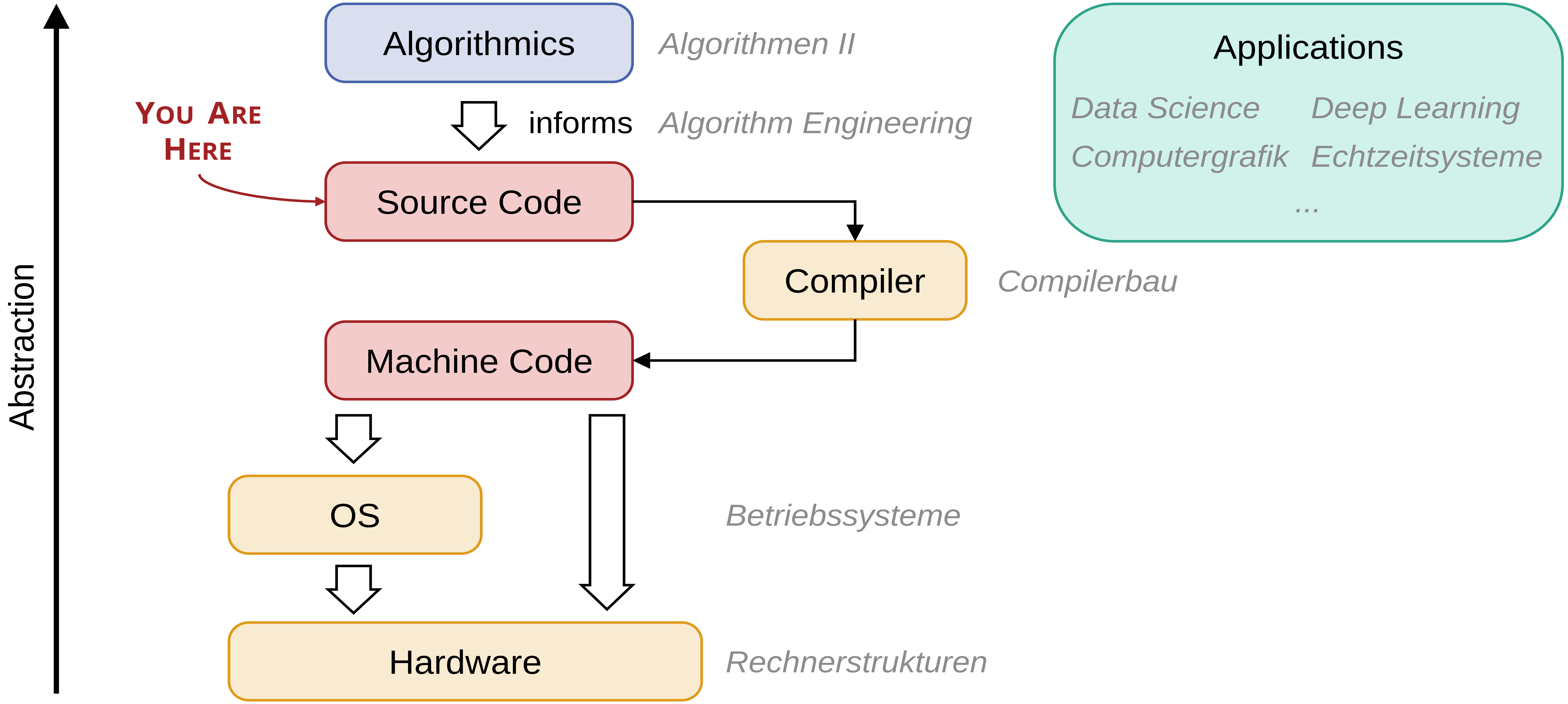
We do not assume knowledge in:

- Advanced computer architecture ↗
  - See next slides...
- Advanced algorithms ↗
  - Introduced as needed

Assumed Knowledge

- *Operating system* ↑
  - We (mostly) do not inspect OS
  - Programs start at `main()` and end at `exit()`
  - We ignore implementation details, like drivers
  - System libraries exist
  - Historical baggage is just that
  - we *DO* model OS performance ↗
- *Basic algorithms* ↑
  - Big O notation
  - Some basic algorithms as presented in GBI, Algo 1
- *Computer architecture* ↑
  - No lecture referencable
  - Need more than "Rechnerorganisation"
  - Assumed: 5-Stage RISC Pipeline
  - Caches
  - Some assembly knowledge (x86, RISC-V, Pseudoops)

The Context of this Lecture



There will be frequent backward- ↖, forward- ↗, and external- ↑ references

Our Performance Model

Or How I Learned to Stop Worrying and Love the Prefetcher

RO: 5 STAGE DLX ("DELUXE") PIPELINE

- In order ⇒ one slow instruction slows down all following instructions
- Stall on memory access
- (Relatively) low clock
- Weird hacks (branch delay slot ⇒ nops everywhere)
- Uncore (TLB, Caches, ...) not discussed

the DLX [DeLuXe] 5-Stage RISC pipeline

Reminder: The 5-Stage RISC pipeline

Our Performance Model

Or How I Learned to Stop Worrying and Love the Prefetcher

ACTUAL CONTEMPORARY HW: AMD ZEN 4

- Out of Order (OoO)
- A lot of functional blocks
- Extremely fast
- Multiple layers of optimizations
- Difficult to understand
- Sometimes weird performance characteristics
- Not discussed in detail in this lecture

Zen 4 block diagram

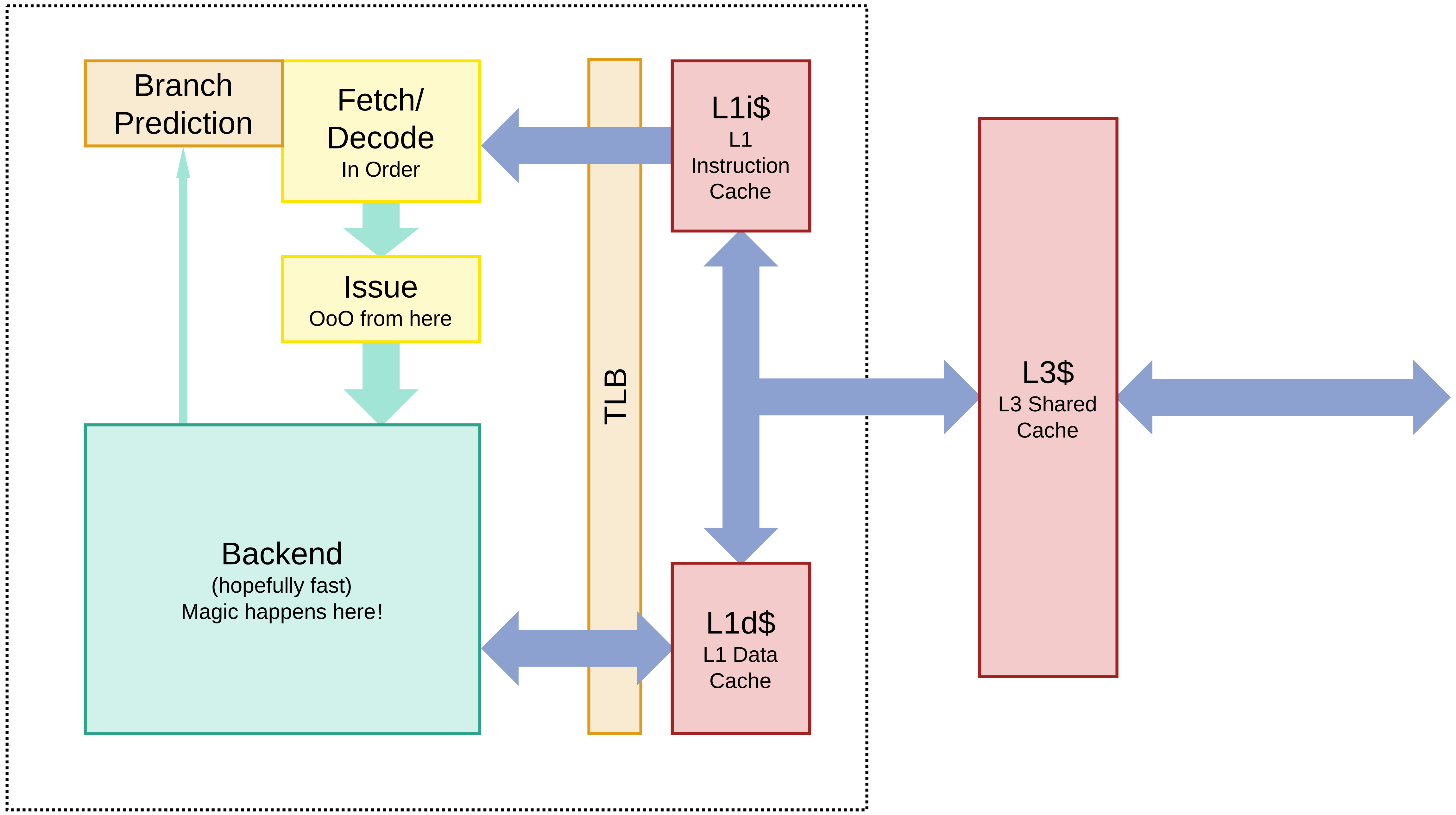
Real world: AMD Zen 4

Our Performance Model

Or How I Learned to Stop Worrying and Love the Prefetcher

THIS LECTURE:
IN-BETWEEN & BLACK BOXED ⇒ INTUITION

- Simple OoO intuition, OoO on instruction level
- simple Branch predictor
- Only L1 and L3 cache
- Single-level TLB
- Linear prefetcher
- No forwarding ⇒ memory latency either L1 or L3 or DRAM



This lecture: μEffCpp

(Instruction) Caching

Money, Money, Money

```c
int sum = 0;
for(int i = 0; i < 3; ++i) sum += i;
```

```asm
  addi a0, zero, 0 # sum
  addi t0, zero, 0 # i
  addi t1, zero, 3 # limit
loop:
  beq t0, t1, end
  add a0, a0, t0
  addi t0, t0, 1
  jal zero, loop
end:
```

How long does it take? (rough estimation)

- Completely executed ("retired") Instructions: 3 Preamble + 3 $\times$ 4 Loop Body + 1 Postamble = 16 Instructions
- Instruction execution:
  about 1 cycle (4 GHz: .25 ns) $\Rightarrow$ 4 ns
- But wait, there's more!

- Issue: instruction fetch: about 75 ns (300$\times$ slower!) ($\sum$ 1204 ns)
- Idea: cache loaded instruction $\Rightarrow$
  7 Fetches, 16 Instructions = 529 ns (2.28$\times$ speedup!)
- Best case: Fetch all at once, 75 + 4 = 79 ns, 15,25x speedup

(Instruction) Caching

Cache Lines

```
int sum = 0;
for(int i = 0; i < 3; ++i) sum += i;
```

```
  addi a0, zero, 0 # sum
  addi t0, zero, 0 # i
  addi t1, zero, 3 # limit
loop:
  beq t0, t1, end
  add a0, a0, t0
  addi t0, t0, 1
  jal zero, loop
end:
```

Best case last slide:

Fetch all at once, 75 + 4 = 79 ns, 15,25x speedup

- Number of bytes to fetch?
  RISC-V: 4 Byte per Instruction: 7×4 Bytes = 28 Bytes
- Number of bytes per memory transaction?
  (usually) 64 bytes
- Number of fetches required? sometimes 2! (often 1)

What does number of fetches depend on?

- Memory organized as cache lines
- Cache line (usually) size of memory transaction
- Cache lines are aligned:
  $[n \times 64, n \times 64 + 63], n \in \mathbb{N}$
- Crossing boundaries: Both lines need to be fetched

(Instruction) Caching

Speedup to the moon! 🚴

```c
int sum = 0;
for(int i = 0; i < 1000; ++i) sum += i;
```

```asm
  addi a0, zero, 0 # sum
  addi t0, zero, 0 # i
  addi t1, zero, 1000 # limit
loop:
  beq t0, t1, end
  add a0, a0, t0
  addi t0, t0, 1
  jal zero, loop
end:
```

- Retired Instructions: 3 Preamble + 1000 $\times$ 4 Loop Body + 1 Postamble = 4004 Instructions
- Cacheless execution: $\Rightarrow$ 4004 * (0.25 + 75) = 1001 ns + 300300 ns = 301301 ns
- Cached Execution: 1001 ns + 75 ns = 1076 ns
- Speedup: 280 $\times$
- Limit? 75.25 ns / .25 ns = 301 $\times$

(Instruction) Caching

A (Real) Cache Hierarchy

Zen 4

Level	Name	Latency	Factor
"0"	Registers	1 cycle (often even lower)	*n/a*
L1	L1 Cache	4 cycles (< 1 ns)	$\geq 4 \times$
L2	L2 Cache	14 cycles ($\approx$ 2.5 ns)	$3.5 \times$
L3	L3 Cache	$\approx$ 8-9 ns	$\approx 3.5 \times$
"4"	(D) RAM	$\approx$ 75 ns	$\approx 8.8 \times$

- Model:
  Registers $\times$ 4 $\Rightarrow$
  L1 $\times$ 12.25 $\Rightarrow$
  L3 $\times$ 8.9 $\Rightarrow$ RAM
- L1 Latency: 4 cycles...
- Assumption before: Instruction fetch in "0" cycles?
- Solution: Prefetching

(Instruction) Prefetching

Gimme, Gimme, Gimme

```
addi a0, zero, 0
addi a0, a0, 1
addi a0, a0, 1
addi a0, a0, 1
addi a0, a0, 1
addi a0, a0, 1
addi a0, a0, 1
addi a0, a0, 1
addi a0, a0, 1
addi a0, a0, 1
addi a0, a0, 1
addi a0, a0, 1
addi a0, a0, 1
addi a0, a0, 1
addi a0, a0, 1
addi a0, a0, 1
addi a0, a0, 1
addi a0, a0, 1
addi a0, a0, 1
addi a0, a0, 1
addi a0, a0, 1
addi a0, a0, 1
addi a0, a0, 1
addi a0, a0, 1
addi a0, a0, 1
addi a0, a0, 1
addi a0, a0, 1
addi a0, a0, 1
addi a0, a0, 1
addi a0, a0, 1
addi a0, a0, 1
addi a0, a0, 1
```

- Is that efficient assembly? No
- But does it show my point? Yes
- What is the next instruction? The one below

- **Idea prefetching**: Preload data that is expected to be used into cache / core
- **Our Model**: Simple prefetcher that only guesses linear patterns
- **Linear pattern**: Prefetch the data at same offset of previous two loads, if "enough" such loads have happened

DISCLAIMERS AND PRELIMINARIES

This lecture is opinionated :)

Why C?

Language to implement Unix in

- 👍: No assembly!
- 👍: Generic memory management!
- 👍: Platform-independent code!
- 👍: Simple compiler!
- 👍: Fast!
- 👍: Library writeable in C!

TL;DR: It was a great tool in 1972...

Why C today?

- Mostly for supporting existing projects
- Lingua franca: Interface language for a lot of languages
- Quite influential

- Performance reference
- Ship precompiled? ⇒ C an option
- Memory management still important topic

A Simple C Program

DI*why*?

Get out your laptops, it's practice time!

Hello World *(it had to be done)*

```
#include <stdio.h>
void main(int argc, char **argv) {
    printf("%s %s\n", "Hello", "EffCpp");
}
```

And now?

Compile and execute

```
vesemir@kaermorhen ~/effcpp/C1$ gcc hello.c -o hello
vesemir@kaermorhen ~/effcpp/C1$ ./hello
Hello EffCpp
vesemir@kaermorhen ~/effcpp/C1$
```

Wait...

```
vesemir@kaermorhen ~/effcpp/C1$ gcc hello.c -O3 -o hello
vesemir@kaermorhen ~/effcpp/C1$ ./hello
Hello EffCpp, but faster!
vesemir@kaermorhen ~/effcpp/C1$
```

En Détail

```
#include <stdio.h>
void main(int argc, char **argv) {
    printf("%s %s\n", "Hello", "EffCpp");
}
```

```
#include <stdio.h>
```

- Used to include other files ↗
- Here: what is this `printf()`?

```
void main(int argc, char **argv) {
```

- The program begins here
- `argc, argv` command line parameters

```
    printf("%s %s\n", "Hello", "EffCpp");
```

- print to stdout
- Format string: see manpage ↑

```
}
```

- This is the end my only friend, the end
- Closes function ↗, block ↗, and scope ↗

A Simple C Program

Control Flow

Say Hello to Java

`int friends() {`

It was raining `while` you waited `for` the plane to arrive. But what `else` could you `do`? In any `case`, it was no question `if` they arrived, the pilots aren't taking a `break`, they'd `continue` the approach. You already planned where y'all would `goto`↗ after the airport, and then `switch` to another club. You're very happy they all did `return EXIT_SUCCESS;`

`}`

Oh Look, a Performance Discussion!🚴

WHICH ALTERNATIVE IS BETTER?

```
int sum = 0;
for(int i = 0; i < 3; ++i) sum += i;
```

```
int sum = 0;
sum += 1;
sum += 2;
```

AND NOW?

```
int sum = 0;
for(int i = 0; i < 4096; ++i) sum += i;
```

```
int sum = 0;
sum += 1;
...
sum += 4095;
```

- Few instructions ⇒ good for i-cache
- Bad: branches
- Good to predict
- Easy to understand

- Good: No branches: good for GPU (?)
- Bad: cache size
- Bad: hard to understand, easy to mess up

And what is better? ⇒ Depends, go benchmark! ↖

PS: Why not Gauss? Example also valid for other cases, also compiler will optimize.

Types

NUMBERS

- Unsized signed integers
  - `signed char` ↗
  - `short`
  - `int`
  - `long`
  - `long long`
- Unsized floating point numbers
  - `float`
  - `double`
  - `long double`

NOT NUMBERS

- `void` ↗
- `char` ↗
- `_Bool` *Ab C23:* `bool`

- Functions ↗
- `struct`ures ↗
- Arrays ↗
- Pointers ↗
- `union`s ↗
- `enum`erations ↗

Types

Numbers

Remember platform independency? Data width: kind of important

- `short`: $\geq$ 16 bit
- `int` $\geq$ 16 bit, *commonly* 32 bit
- `long` $\geq$ 32 bit, *no consensus!*
- `long long`$\geq$ 64 bit
- `char`: Equivalent to a byte
- `signed char`: Equivalent to a signed byte

enumerations

Numbers, but Named?!

```c
if (status == 4) {
    status = 5;
    break;
} else {
    status = 0;
    continue;
}
```

- magic numbers
- often: encode status

```c
if (status == SUCCESS) {
    status = COMPLETE;
    break;
} else {
    status = START;
    continue;
}
```

- Clearly readable

USAGE

```c
enum enum_name {
    FIRST_CONSTANT,
    SECOND_CONSTANT,
    THIRD_CONSTANT,
    DIRECTLY_ASSIGNED = 7,
    NEGATIVE_NUMBERS = -5,
    SUCCESSOR,
    REASSIGNMENT_IS_VALID = 7
}; // this semicolon is important!
```

- Counting up, starting at zero
- Used without enum name

```c
int main() {
        printf("%d %d %d %d %d %d %d\n",
            FIRST_CONSTANT, SECOND_CONSTANT, THIRD_CONSTANT,
            DIRECTLY_ASSIGNED, NEGATIVE_NUMBERS, SUCCESSOR,
            REASSIGNMENT_IS_VALID);
    }
```

Result?

```
0 1 2 7 -5 -4 7
```

🚴Performance: Well, it's numbers. Use often!

Functions

They're (mostly) like Java!

- Functions perform calculations that are repeatedly needed
- Functions can have side effects ↗
- Functions have a name (good for code structure)
- Functions take parameters
- Functions return (at most) one value (use `void` for zero)
- There is no Java-"`this`"

```
int add(int first, int second) {
    for (int i = 0; i < second; ++i)
        first += 1;
    return first;
} // look ma, no semicolon
```

Yes, it's wrong. Here's a fixed version:

```
int add_fixed(int first, int second) {
    switch(second < 0) {
        case false:
            for (int i = 0; i < second; ++i)
                first += 1;
        default:
            return first;
        case true:
            for (int i = 0; i > second; --i)
                first -= 1;
            return first;
    }
}
```

Disclaimer: Code written by trained professionals in a secure environment. Do not try at home.

Write Your Own Type

MOTIVATION

- Often: more information than one number
- Idea: structure data, name

Whats not in a C struct?

- Code
- Default values
- Inheritance / OOP
- Visibility limits

DEFINITION

```
struct coord {
int x;
int y;
}; // this semicolon is (again) important!

struct rect {
struct coord upper_left; // structs can contain structs
struct coord lower_right;
};
```

INITIALIZATION

```
int main() {
struct coord undefined; // so it begins
struct coord zero_initialized = {};
struct coord value_initialized = {1, 2};
struct coord copy = value_initialized;

struct rect undef;
struct rect zero_init = {};
struct rect value_init = {copy, copy};
struct rect explicit_def = {{1, 2}, {3, 4}};
}
```

Bonus Question: Is this valid and what is the result?

`struct coord coord = {1};`

Valid! `coord.x == 1 && coord.y == 0`

Usage

HOW TO USE?

```
struct coord {int x; int y;};
struct rect { struct coord up_left; struct coord low_right;};

struct coord swap(struct coord input) {
    return (struct coord) {input.y, input.x};
}

int main() {
    struct coord undefined;
    undefined.x = 1; // select member with "."
    undefined.y = 2; // now completely defined!
    printf("%d\n", undefined.x); // => "1"

    struct coord init = {4, 5};
    struct rect rect  = {init, {7, 8}};
    // "." can be chained for recursive access
    printf("%d %d\n", rect.up_left.x, rect.low_right.y); // => "4 8"
    printf("%d\n", swap(rect.up_left).x); // => "5"
}
```

WHEN TO USE?

- Collect semantically related entities
- Collect values often used together
- Return multiple values from functions
- Name things

WHEN NOT TO USE?

- To pass unrelated parameters to function

Recap

What did we learn today?

- What is efficiency and why is that important?
- What do you already know about performance?
- What is our performance model?
- Introduction into C
  - Control Flow
  - Builtin Types
  - `enum`
  - Functions
  - `struct`

And next lecture? ↗

- What is memory?
- What are pointers and arrays?
- What is `malloc()`?
- Performance discussions of `struct`s and functions
- `union`s