Software Systems

Lecture 2

Jonathan Dönszelmann
Vivian Roest

Delft University of Technology, The Netherlands

January 24, 2024

Previous Lectures

- Threads
- Spawning and Joining
- Channels
- Threadpools
- Asynchronous programming

Today

How to find and improve the performance of (existing) programs.

Today

How to find and improve the performance of (existing) programs.

Question:

Do you know any techniques already?

Today

How to find and improve the performance of (existing) programs.

1. Running code less
2. Make code that has to run faster

Today

How to find and improve the performance of (existing) programs.

1. Running code less
2. Make code that has to run faster

But first! How do we measure performance?

Measuring performance

1. Benchmarking
2. Profiling

Measuring performance: Benchmarking

```
time ./some/binary
time cargo run
```

Warning:

Timing `cargo run` will also measure compile time. Make sure you compile first.

1. Very crude way measure performance
2. Not very accurate
3. Very useful to get an estimate!

Measuring performance: Benchmarking

Question:

Why is this not very accurate?

Measuring performance: Benchmarking

Remove influences of things we don't want to measure

Average results to remove random fluctuations

Measuring performance: Benchmarking

- Rust used to have this built-in
- Now Rust only has the framework[1], which extrenal libraries can take advantage of. e.g. `criterion`
- `criterion` will:
    - Take samples out of many runs
    - Warm up
    - Show difference with old runs
    - Detect outliers
    - Give statistical significance (p-value)

Demo!

---

[1] Technically it is still all there, but it's recommended to use an external library for it

Measuring performance: Profiling

Question:

Did you use profiling in Advanced Computing Systems?

Measuring performance: Profiling

- Measure execution times of different parts of an application
- On Linux: Perf
  - Hardware counters (`perf stat`)
  - Sampling
- On Windows: use WSL.
- On MacOS: Use DTrace

`https://perf.wiki.kernel.org/index.php/Tutorial`

Measuring performance: Profiling

- Measure execution times of different parts of an application
- On Linux: Perf
    - Hardware counters (`perf stat`)
    - Sampling
- On Windows: use WSL. On MacOS: Use DTrace

Demo!

Measuring performance: Profiling

Other useful tools:

- Valgrind
- Cachegrind
- Callgrind
- Dumb but useful: htop

Other kinds of performance

- for example: memory usage
- always a tradeoff

Question:

Can you think of other measures of performance?

Increasing performance

Split into three categories:

- Algorithmic improvement (being smart)
- Making code faster
- Running less code

Memoization

```rust
fn fibonacci(n: u64) -> u64 {
    match n {
        0 => 1,
        1 => 1,
        n => fibonacci(n-1) + fibonacci(n-2),
    }
}
```

- `fibonacci(5)` needs `fibonacci(4)` and `fibonacci(3)`
- `fibonacci(4)` needs `fibonacci(3)` and `fibonacci(2)`

If we added some memory we could “cache” the results of fibonacci

Memoization

```rust
use memoize::memoize;

#[memoize]
fn fibonacci(n: u64) -> u64 {
    match n {
        0 => 1,
        1 => 1,
        n => fibonacci(n-1) + fibonacci(n-2),
    }
}
```

- Adds a mapping from parameters to results
- “caches” the results of fibonacci to reduce calls

Lazy evaluation

- Iterators don't compute values immediately
- Only when `nth` is called do we run code

```
vec![1, 2, 3, 4].into_iter().map(|i: i32| i.pow(2)).nth(2)
```

- Slower in general
- Sometimes faster due to a smart compiler
- May be faster when few items end up needing processing

IO Buffering

- Every time `read_exact` is used, a system call is performed
- Goal: perform fewer system calls
- read as much as we can per system call

```rust
use std::fs::File;
use std::io::Read;

fn main() {
    let mut file = File::open("very_large_file.txt").unwrap();
    let mut buf = [0; 5];

    while file.read_exact(&mut buf).is_ok() {
        // process the buffer
    }
}
```

IO Buffering

- Read as much as possible
- Read reads from internal buffer until empty
- When empty: perform another system call

```rust
use std::io::{BufReader, Read};

fn main() {
    let mut file = File::open("very_large_file.txt").unwrap();
    // only line changed
    let mut reader = BufReader::new(file);

    let mut buf = [0; 5];

    while reader.read_exact(&mut buf).is_ok() {
        // process the buffer
    }
}
```

Note: also works for (network) sockets and other file-like objects

Lock contention

Lock before slow operation

```rust
let result = Arc::new(Mutex::new(0));

for i in 0..3 {
    let r = result.clone();
    spawn(move || {
        let mut guard = r.lock();

        *guard += slow(i);
    });
}
```

Lock after slow operation

```rust
let result = Arc::new(Mutex::new(0));

for i in 0..3 {
    let r = result.clone();
    spawn(move || {
        let answer = slow(i);

        *r.lock() += answer;
    });
}
```

- Try to make critical sections smaller
- Hint: use scope blocks to be explicit about critical sections

Moving code outside of a loop

```rust
fn example(a: usize, b: u64) {
    for _ in 0..a {
        some_other_function(b + 3)
    }
}
```

Compute `b + 3` first:

```rust
fn example(a: usize, b: u64) {
  let c = b + 3;
  for _ in 0..a {
    some_other_function(c)
  }
}
```

Compilers are good at this: `https://godbolt.org/z/n58GjhsP5`

Memory allocation

- Memory allocation and deallocation takes time. For example:
  - `Vec::new()`, `HashMap::new()`, `String::new()`
  - Resizing any of the above
  - `Box::new()`
  - Cloning any of the above
  - `Arc::new()`, `Rc::new()`
  - Dropping any of the above
- Time depends on allocator you use (this can be changed!)

Question:

What can we do to reduce allocation?

Memory allocation

- Static allocation
- Preallocating to the right size
- Moving allocations outside an inner loop
- Arena allocation:
    - Small special-purpose allocator
    - (usually) no individual freeing supported
    - free all at once

Memory Allocation

```rust
struct Doggo {
    cuteness: u64,
    scritches_required: bool,
}

// Create a new arena to allocate into.
let bump = Bump::new();

// Allocate values into the arena.
let scooter = bump.alloc(Doggo {
    cuteness: u64::MAX_VALUE,
    scritches_required: true,
});

// Mutable references to the just-allocated value are returned.
assert!(scooter.scritches_required);
scooter.cuteness += 1;
```

From: `https://docs.rs/bumpalo/latest/bumpalo/#example`

Memory Allocation

Question:

How is this different to a `Vec`?

Making code faster

Question:

Do you know any techniques already to create faster code?

Making code faster

Question:

Do you know any techniques already to create faster code?

Inlining

```
// compiler may
// ignore this
#[inline]
fn example_1() {
    ...
}

// not this
#[inline(always)]
fn example_2() {
    ...
}

fn main() {
    example_1();
    example_2();
}
```

- Calling functions works with call instructions
- This represents some overhead
- What if we pasted the body of a function at the callsite?
- Larger code size, (sometimes) faster code

Compiler Options

- The compiler has multiple optimization levels
- 0, 1, 2 and 3 for "speed"
- 's' and 'z' for "size" (code size)
- The default is very low ⟶ fast(er) compile times
- To select better options: `cargo run --release`

Changing Compiler Options

`Cargo.toml`

```toml
[package]
...

[dependencies]
...

# for cargo run
[profile.dev]
opt-level = 1

# for cargo run --release
[profile.release]
opt-level = 3
```

- `opt-level` can be 0-3,‘s’,‘z’
- 3 not always the best: experiment!
- Optimization levels are a tradeoff, 3 may be fast but the code size could be huge
- `https://godbolt.org/z/nTnef7888`

Link-Time optimization

`Cargo.toml`

```
[package]
...

[dependencies]
...

# for cargo run
[profile.release]
opt-level = 3
lto = true
```

- Each crate compiled separately
- No optimiziation between crates
- With `lto` there is

Target CPU

1. Modern CPUs sometimes have specialized hardware
2. Not every CPU has the same hardware
3. Programs cannot assume they can use this hardware
4. With `target-cpu` you choose a specific cpu

Compile time:

```
RUSTFLAGS="-C target-cpu=native" cargo run
```

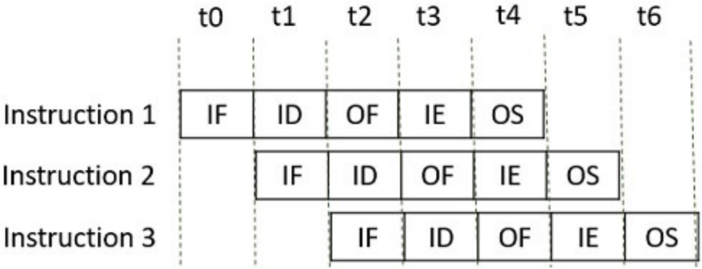
Target CPU (runtime)

```rust
#[inline(always)]
fn foo_impl() { ... }

// This generates a stub for CPUs that support SSE4:
#[target_feature(enable = "sse4")]
unsafe fn foo_sse4() {
    // inlining here will recompile
    // foo_impl for sse4
    foo_impl()
}

// This generates a stub for CPUs that support AVX:
#[target_feature(enable = "avx")]
unsafe fn foo_avx() {
    foo_impl()
}
```

Branch prediction

```
if a > b {
    do_x();
} else {
    do_y();
}
```

- Condition only available late due to pipelining
- Predict the outcome of the condition
- Start executing most-likely branch
- Wrong prediction ⟶ performance penalty



	t0	t1	t2	t3	t4	t5	t6
Instruction 1	IF	ID	OF	IE	OS		
Instruction 2		IF	ID	OF	IE	OS	
Instruction 3			IF	ID	OF	IE	OS

Cold paths

```rust
#[cold]
fn rarely_executed_function() { }
```

- Mark rarely used functions
- Generated code will favor optimizing other paths

Better branch prediction?

Question:

How do we know what to inline and what to mark as cold?

PGO: better branch prediction

- Manually figuring out what to inline / mark as cold is hard
- Profiling can help! ⟶ "Profile-Guided Optimization"
- See lecture notes how to do this[2]

1. Build special 'Instrumented' binary
2. Gather statistics while running
3. Build a better program

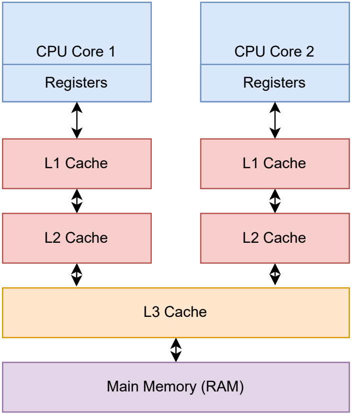
Note:

If the actual workload is substantially different from the instrumented run then the program could perform worse!

---

[2] `https://cese.pages.ewi.tudelft.nl/software-systems/part-1/lecture-notes/lecture-2.html`

Caching



- Closer caches are smaller and faster
- Smaller code and smaller data may mean more of it fits in cache
- Consecutive data is usually better
- Optimized code may be larger
- Benchmark! (cachegrind)

Rust: Zero cost abstractions

Rust provides abstractions:

- Iterators
- Traits & Generics
- Built-in collections
- Closures (lambda functions)

Question:

Does this mean Rust is slower than C?

Rust: Zero cost abstractions

Question:

Does this mean Rust is slower than C?

- Not necessarily!
- Abstractions have no cost if you don't use them (unlike Python)
- If you do use them, they are close to what you could manually make
- The compiler is very smart!

Rust: Static dispatch

```
struct A; struct B;
trait X {
    fn something(&self);
}
impl X for A {
    fn something(&self) { ... }
}
impl X for B {
    fn something(&self) { ... }
}

fn run_something<T: X>(x: T) {
    x.something();
}

run_something(A);
run_something(B);
```

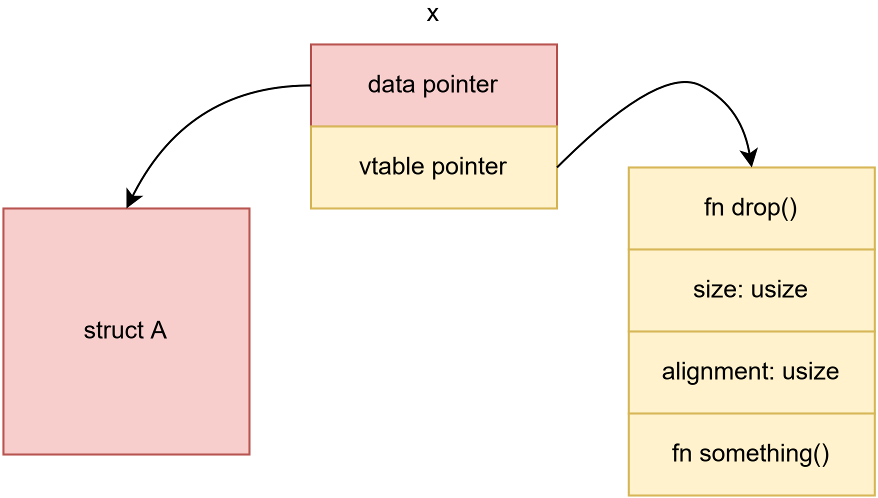
- Static Dispatch
- Covered in Software Fundamentals (Lecture 5)
- Code is duplicated for different types

Rust: Dynamic dispatch

```
struct A; struct B;
trait X {
    fn something(&self);
}
impl X for A {
    fn something(&self) { ... }
}
impl X for B {
    fn something(&self) { ... }
}

fn run_something(x: &dyn X) {
    x.something();
}

run_something(&A);
run_something(&B);
```

- Dynamic Dispatch
- Only one version of code
- Vtables!

Rust: Dynamic dispatch



x

data pointer

vtable pointer

struct A

fn drop()

size: usize

alignment: usize

fn something()

Vtable: A description of a type

Rust: Struct layout

```rust
struct A {
    a: u8;
    b: u32;
}

#[repr(packed)]
struct A {
    a: u8;
    b: u32;
}

#[repr(C, align(2))]
struct A {
    a: u8;
    b: u32;
}
```

- By default, structs have padding
- Packed makes structs smaller
- Alignment can make accessing fields slower

Rust: Memory management

```
bitflags! {
    struct Flags: u8 {
        const A = 0b00000001;
        const B = 0b00000010;
        const C = 0b00000100;
        const D = 0b00001000;
        const E = 0b00010000;
        const F = 0b00100000;
        const G = 0b01000000;
        const H = 0b10000000;
    }
}
```

- A boolean is a byte
- Sometimes, packing tighter helps performance too

Rust: Zero copy design

- Cloning takes time
- If we could keep our data in one place:
  - An arena
  - A buffer (for example: packets from network)
- Don't pass around data, pass around offsets and lengths: slices
- Serde: Serializing and Deserializing (`https://docs.rs/serde`)

Much more!

- No silver bullet: Everything is a tradeoff
- Benchmark & Profile to know for sure!
- Lots of other ways:
    - Algorithmic improvements
    - Better/different hash functions
    - Hardware specific optimizations
    - Using dedicated hardware (ASIC, GPU, etc.)
    - Probabilistic datastructures
    - Operating system configuration

Assignment 2: Performance

- Raytracing
- Deliberately slow
- Apply techniques from lecture