{codspeed}

Trust Your Benchmarks, Not Your Instincts:

A Rust Performance Quiz

whoarewe

Arthur

Adrien

We're building

{codspeed}

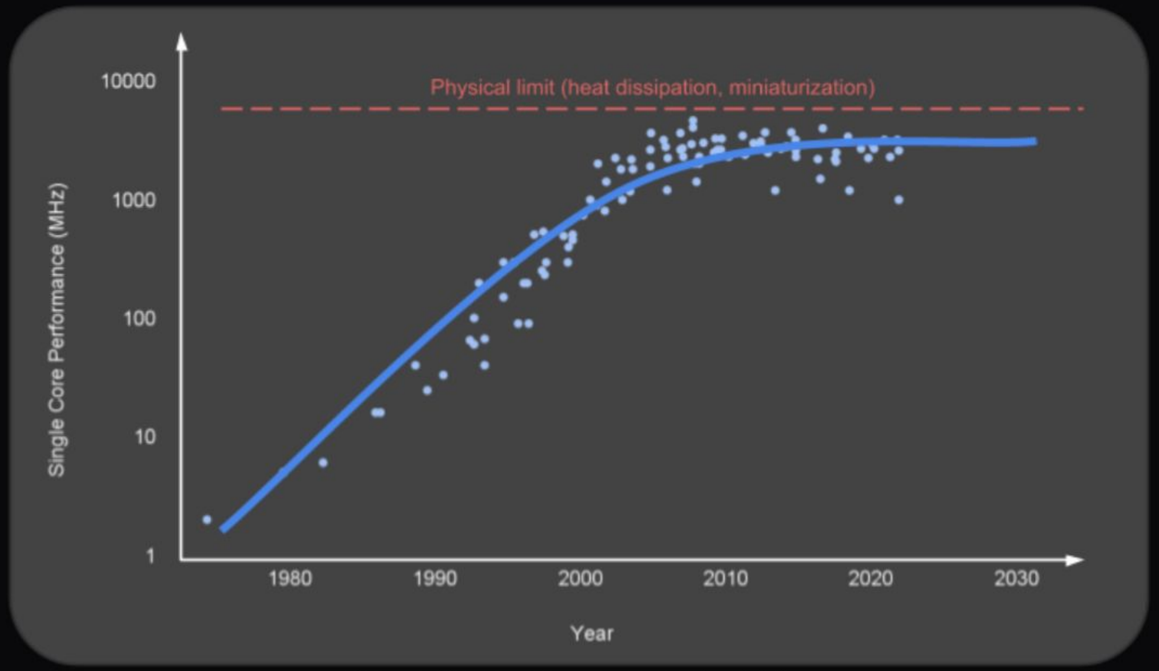
Why performance matters?

- User experience and revenue
  - Every 100ms faster → 1% more conversions (Mobify, earning +$380,000/yr)
- Competitive advantage
  - Performance is a feature
- Cost savings
  - Less compute for same workloads
- Sustainability
  - Reduced data center footprint

Why performance matters even more today?

{codspeed}

Compute performance is unlikely to see further improvements



Source: 50 years of Microprocessor trend data

Demand for computational resources is increasing exponentially

Source: Compute Trends Across Three Eras of Machine Learning

Relying on hardware improvement **does not work anymore**.
We need to **dive in the software** itself and **start optimizing it** heavily.

Getting performance right is hard

- Humans are not machines
- Even the best performance expert will get things wrong

Don’t trust us yet?

Let’s see!

Scan this QR code to join

The quiz

- There is a leaderboard counting points
- **Correct** answers earn points
- **Faster** answers earn more points
- The WINNER receives a prize 🎁
- We'll tell you everytime there is a new question to answer

Measurement

- Performance is measured with Rust 1.90.0 on a bare-metal instance with a 16 cores x86_64 CPU with
- Profile: release
- For each question:
  - Answer with an implementation as the fastest
  - Or answer by saying the performance will be the same (+/- 10% to guarantee an actual change)

Case study #1: Debug vs Release

```
cargo bench --profile debug
```

vs

```
cargo bench --profile bench
```

(uses bench profile by default)

arr size 1000

```rust
pub fn bubble_sort(arr: &mut [i32]) {
    let n = arr.len();
    for i in 0..n {
        for j in 0..n - i - 1 {
            if arr[j] > arr[j + 1] {
                arr.swap(j, j + 1);
            }
        }
    }
}
```

Case study #1: Debug vs Release

```
cargo bench --profile debug
```

vs

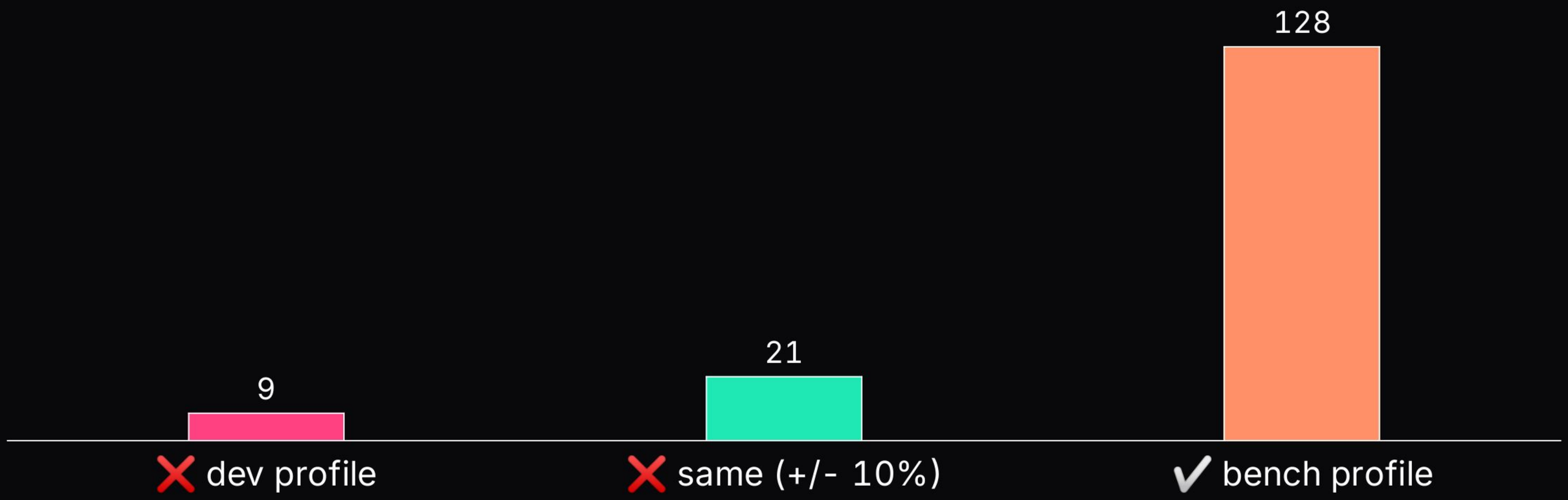
```
cargo bench --profile bench
```

(uses bench profile by default)

arr size 1000

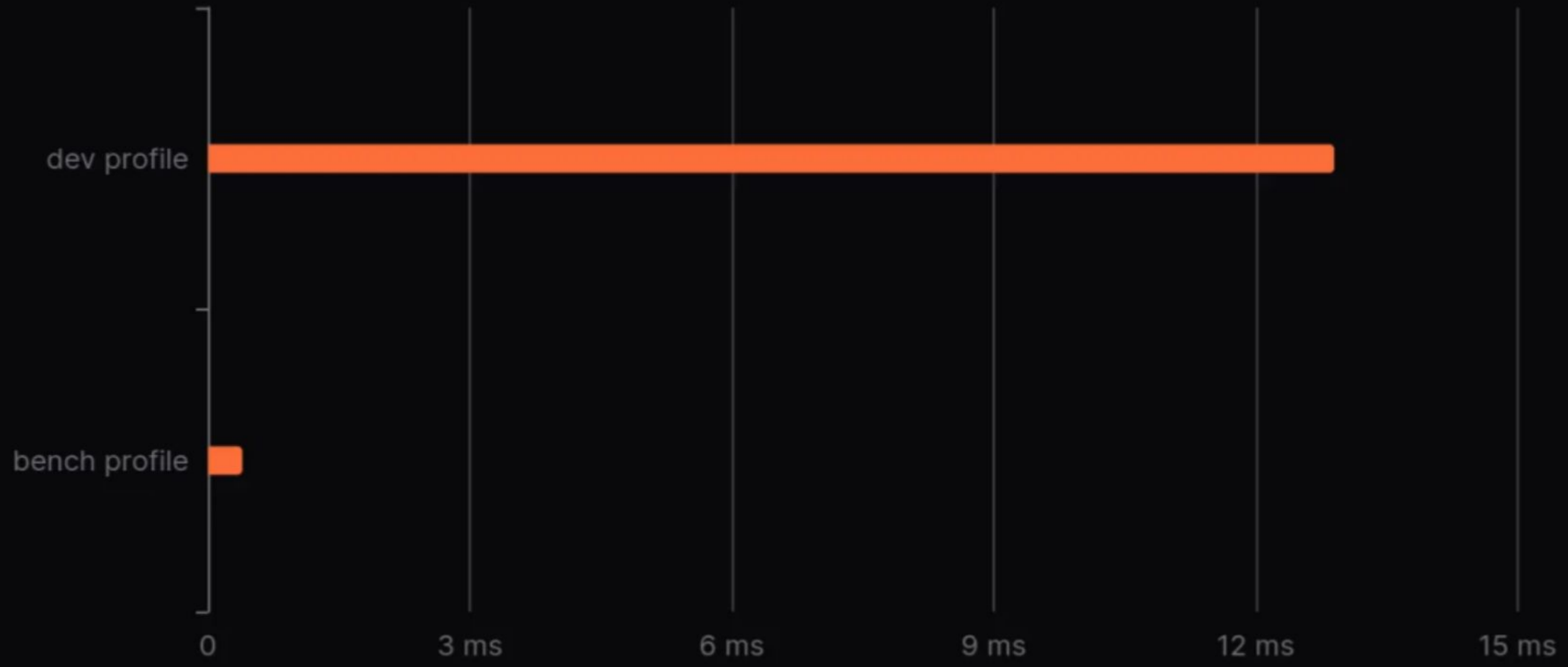
```rust
pub fn bubble_sort(arr: &mut [i32]) {
    let n = arr.len();
    for i in 0..n {
        for j in 0..n - i - 1 {
            if arr[j] > arr[j + 1] {
                arr.swap(j, j + 1);
            }
        }
    }
}
```

Which is faster?



{codspeed}

Answer



`bench_profile` is faster

Why?

```toml
[profile.dev]
opt-level = 0
debug = true
strip = "none"
debug-assertions = true
overflow-checks = true
lto = false
panic = 'unwind'
incremental = true
codegen-units = 256
rpath = false
```

```toml
[profile.release]
opt-level = 3
debug = false
strip = "none"
debug-assertions = false
overflow-checks = false
lto = false
panic = 'unwind'
incremental = false
codegen-units = 16
rpath = false

[profile.bench]
inherits = "release"
debug = true
```

`opt-level` has the most impact
- optimized assembly
- slower build time

Resources: https://doc.rust-lang.org/cargo/reference/profiles.html

Case study #2: Bit shift vs Multiplication with const

x is an argument, y is a const

```
x = 10000
```

Multiply x by 2^y

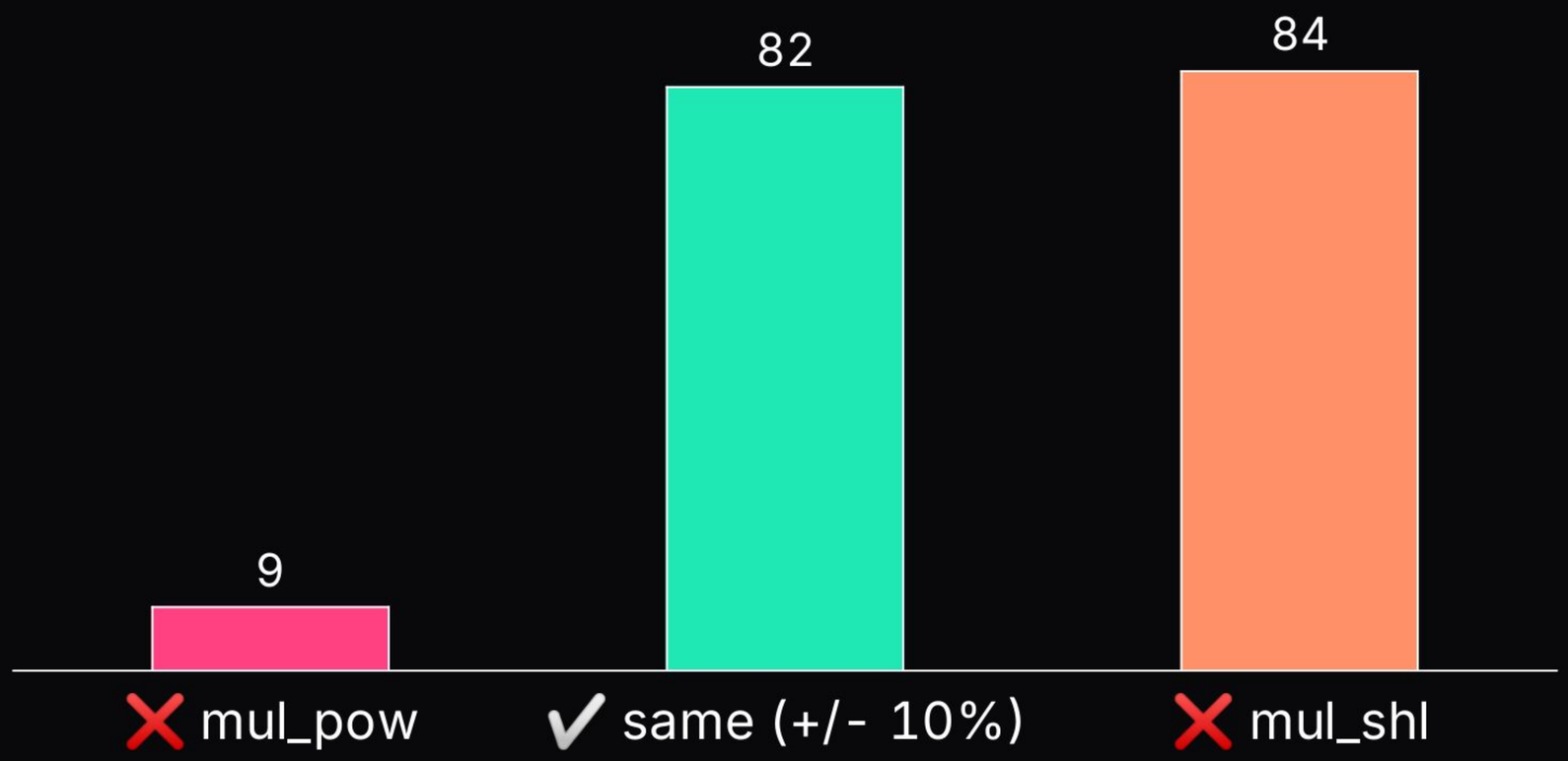
```rust
const y: u32 = 6;

pub fn mul_pow(x: u64) -> u64 {
    x * 2u64.pow(y)
}
```

Shifts x left by y bits
(same as multiplying by 2^y)

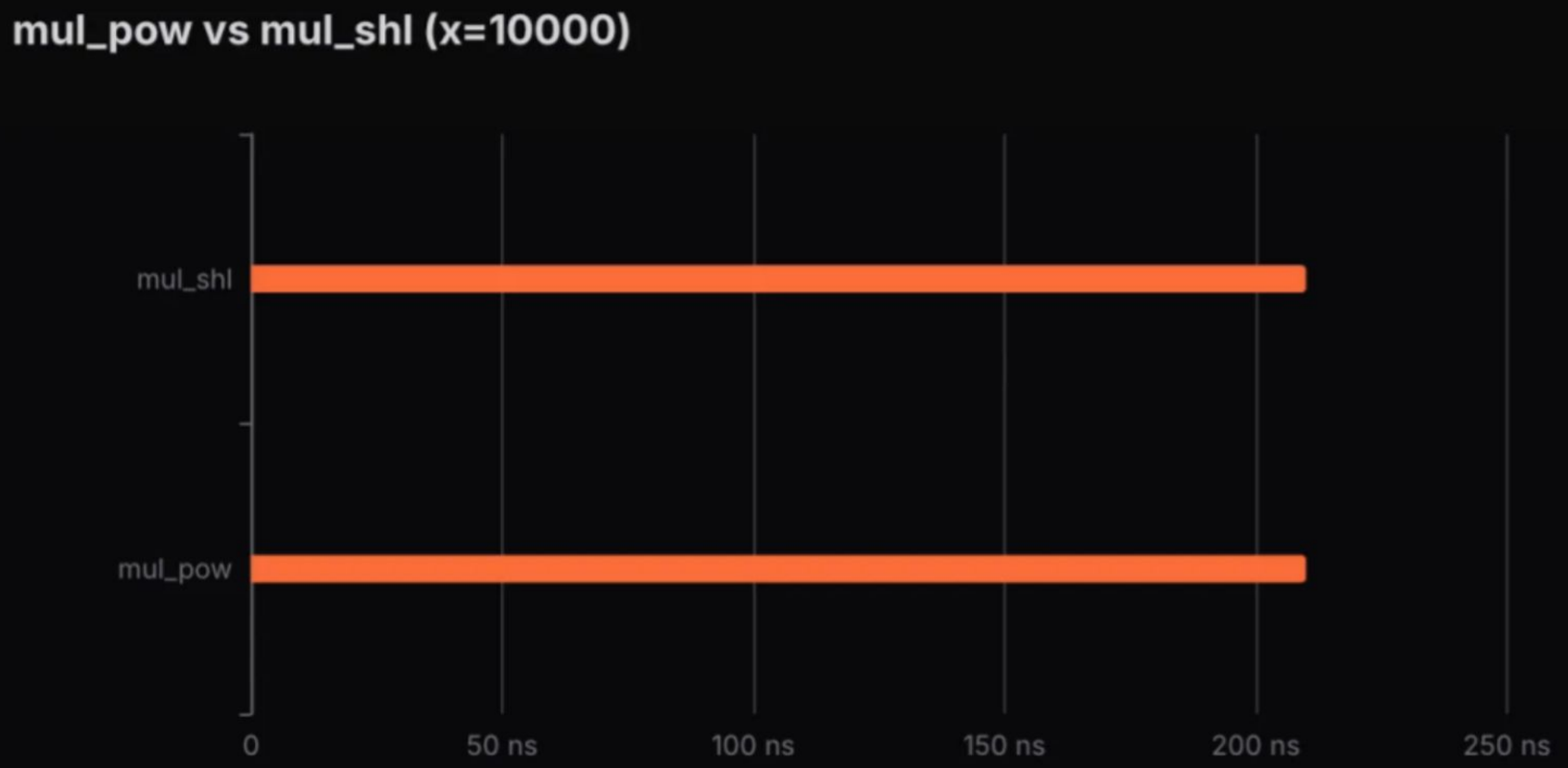
```rust
pub fn mul_shl(x: u64) -> u64 {
    x << y
}
```

Which is faster?

```rust
const y: u32 = 6;

pub fn mul_pow(x: u64) -> u64 {
    x * 2u64.pow(y)
}

pub fn mul_shl(x: u64) -> u64 {
    x << y
}
```



Answer



Same performance

Why?

Rust compilation pipeline:

Let’s look at the assembly code of both implementations

Why?

```rust
const y: u32 = 6;

pub fn mul_pow(x: u64) -> u64 {
    x * 2u64.pow(y)
}

pub fn mul_shl(x: u64) -> u64 {
    x << y
}
```

```asm
mul_pow:
        mov     rax, rdi
        shl     rax, 6
        ret

mul_shl:
        mov     rax, rdi
        shl     rax, 6
        ret
```

How do we end up with the same assembly code?

Why?

`pow` is a `const fn`

Constant Evaluation: the process of computing the result of expressions during compilation

Use `const` where it makes sense to enable compiler optimizations

```rust
pub const fn pow(self, mut exp: u32) -> Self {
    if exp == 0 {
        return 1;
    }
    let mut base = self;
    let mut acc = 1;

    if intrinsics::is_val_statically_known(exp) {
        while exp > 1 {
            if (exp & 1) == 1 {
                acc = acc * base;
            }
            exp /= 2;
            base = base * base;
        }
        acc * base
    } else {
        // ... Faster implementation for dynamic exp ...
    }
}
```

`core::num::uint_macros`

Resources: https://doc.rust-lang.org/reference/const_eval.html

Case study #2: Bit shift vs Multiplication

Now both x and y are arguments

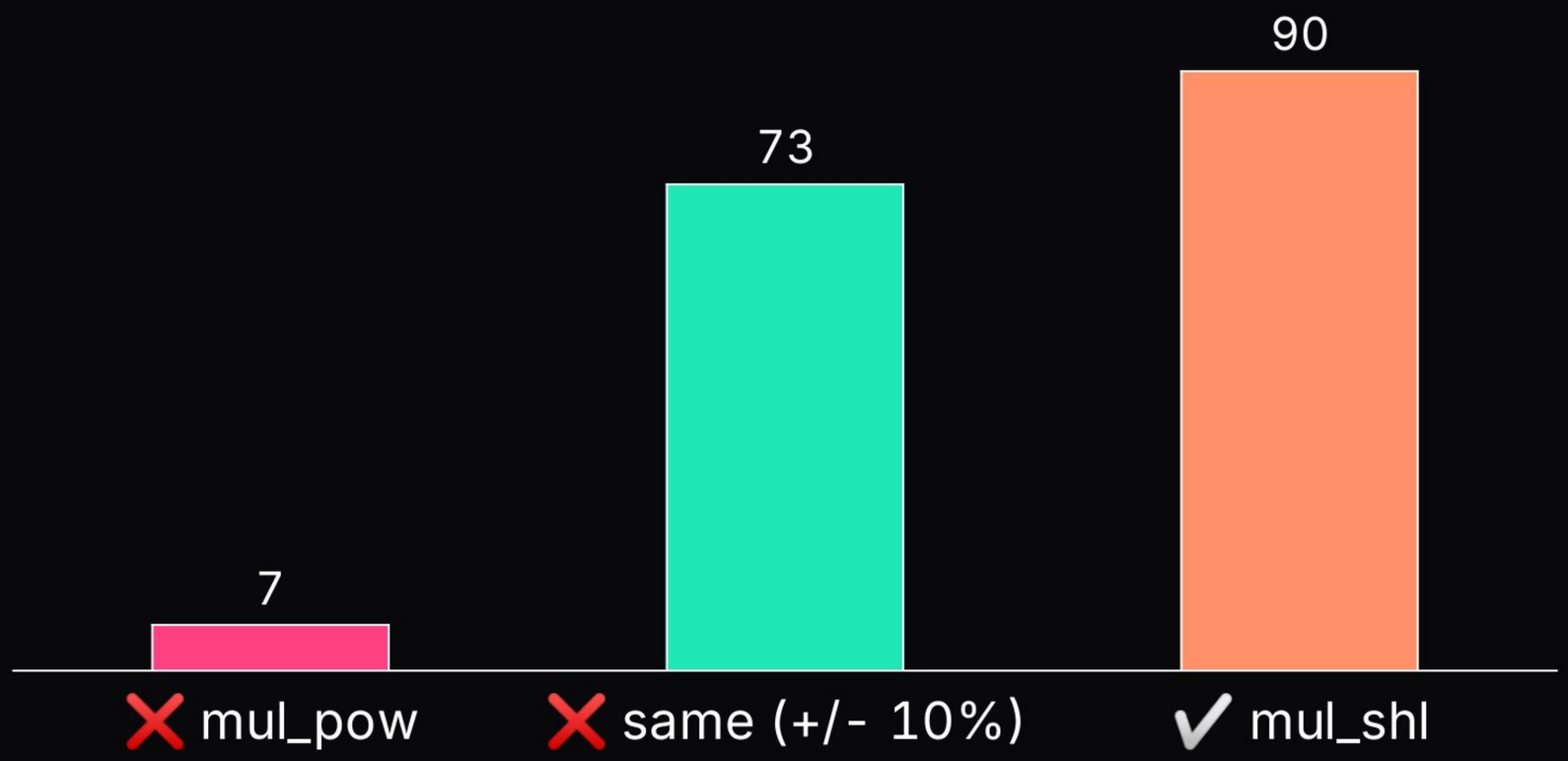
```
x=5, y=64
```

```rust
pub fn mul_pow(x: u64, y: u32) → u128 {
    x * 2u128.pow(y)
}

pub fn mul_shl(x: u64, y: u32) → u128 {
    x << y
}
```

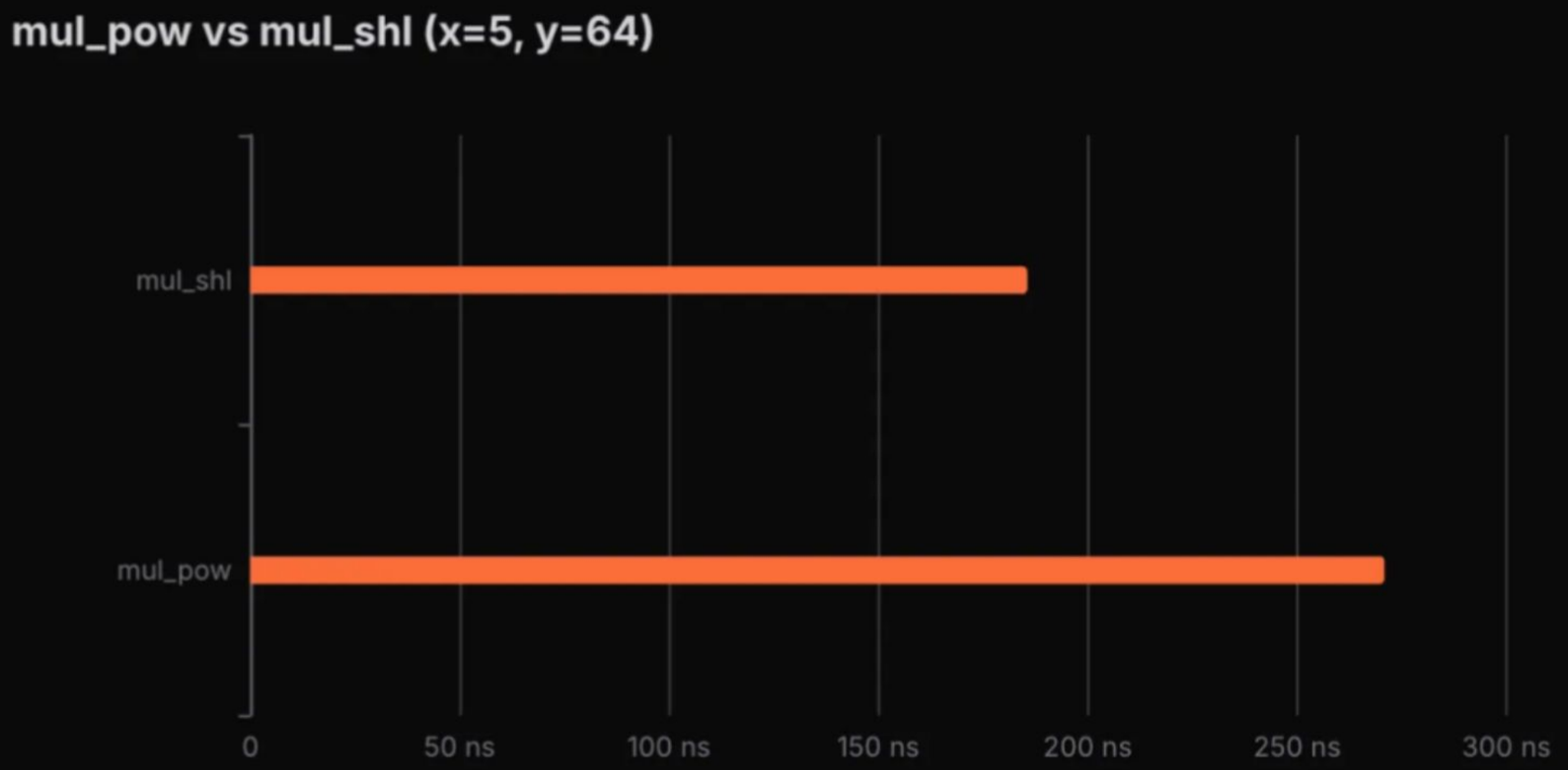
Which is faster?

x=5, y=64

```rust
pub fn mul_pow(x: u64, y: u32) → u128 {
    x * 2u128.pow(y)
}

pub fn mul_shl(x: u64, y: u32) → u128 {
    x << y
}
```



Answer

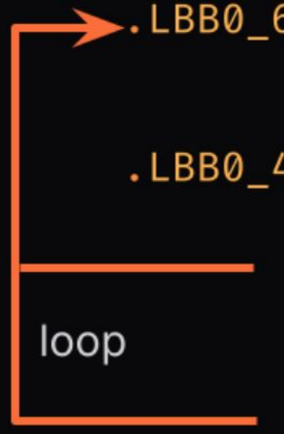


`mul_shl` is faster

Why?

```
mul_shl:
        mov     ecx, esi
        mov     rax, rdi
        shl     rax, cl
        ret
```

```
mul_pow:
        test    esi, esi
        je      .LBB0_1
        mov     ecx, 2
        mov     eax, 1
        jmp     .LBB0_4
.LBB0_6:
        shr     esi
        imul    rcx, rcx
.LBB0_4:
        test    sil, 1
        je      .LBB0_6
        imul    rax, rcx
        cmp     esi, 1
        jne     .LBB0_6
        imul    rax, rdi
        ret
.LBB0_1:
        mov     eax, 1
        imul    rax, rdi
        ret
```



This time the assembly code is way longer for `mul_pow`.

Since `y` is not a `const`, the compiler cannot compute the expression anymore.

Case study #3: Powers

Computing x^10

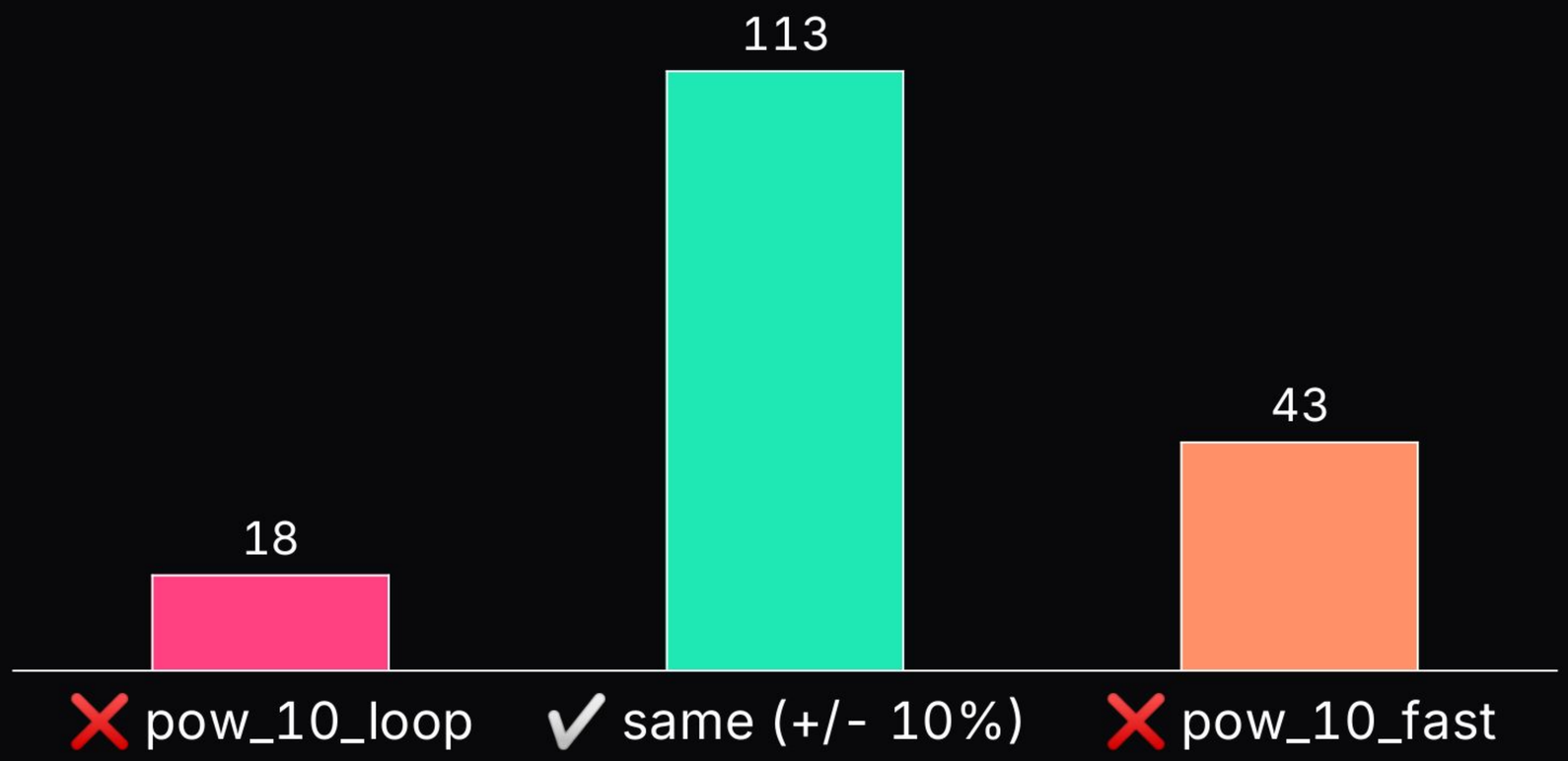
x=1000

```rust
pub fn pow_10_loop(x: u64) -> u64 {
    let mut acc: u64 = 1;
    for _ in 0..10 {
        acc *= x;
    }
    acc
}

pub fn pow_10_fast(x: u64) -> u64 {
    x * x * x * x * x * x * x * x * x * x
}
```

Which is faster?

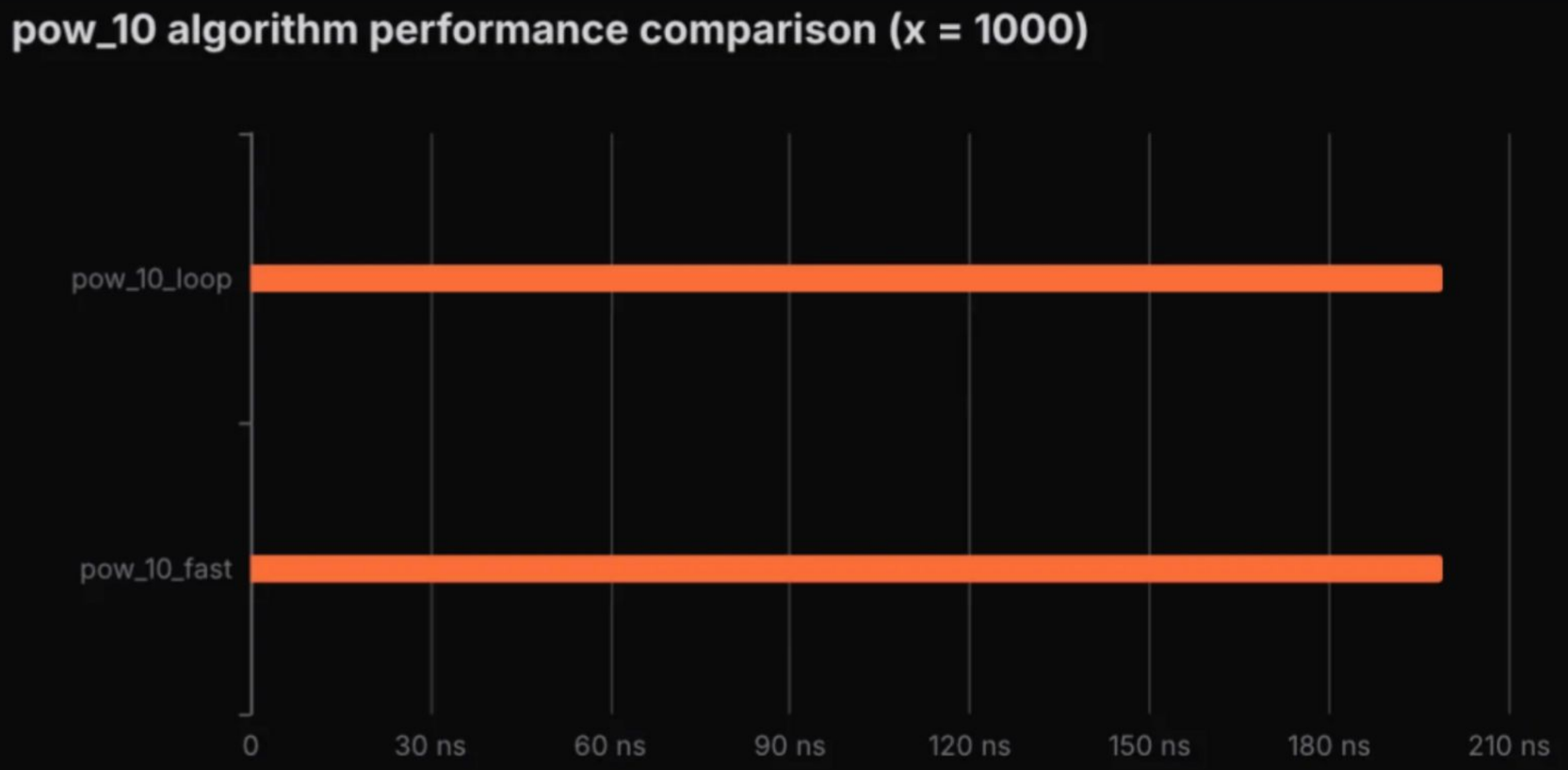
x=1000

```rust
pub fn pow_10_loop(x: u64) -> u64 {
    let mut acc: u64 = 1;
    for _ in 0..10 {
        acc *= x;
    }
    acc
}

pub fn pow_10_fast(x: u64) -> u64 {
    x * x * x * x * x * x * x * x * x * x
}
```



{codspeed}

Answer



Same performance

Why?

```rust
pub fn pow_10_loop(x: u64) -> u64 {
    let mut acc: u64 = 1;
    for _ in 0..10 {
        acc *= x;
    }
    acc
}

pub fn pow_10_fast(x: u64) -> u64 {
    x * x * x * x * x * x * x * x * x * x
}
```

```asm
pow_10_loop:
        imul    rdi, rdi
        mov     rax, rdi
        imul    rax, rdi
        imul    rdi, rax
        imul    rax, rdi
        ret

pow_10_fast:
        mov     rax, rdi
        imul    rax, rdi
        imul    rax, rax
        imul    rax, rdi
        imul    rax, rax
        ret
```

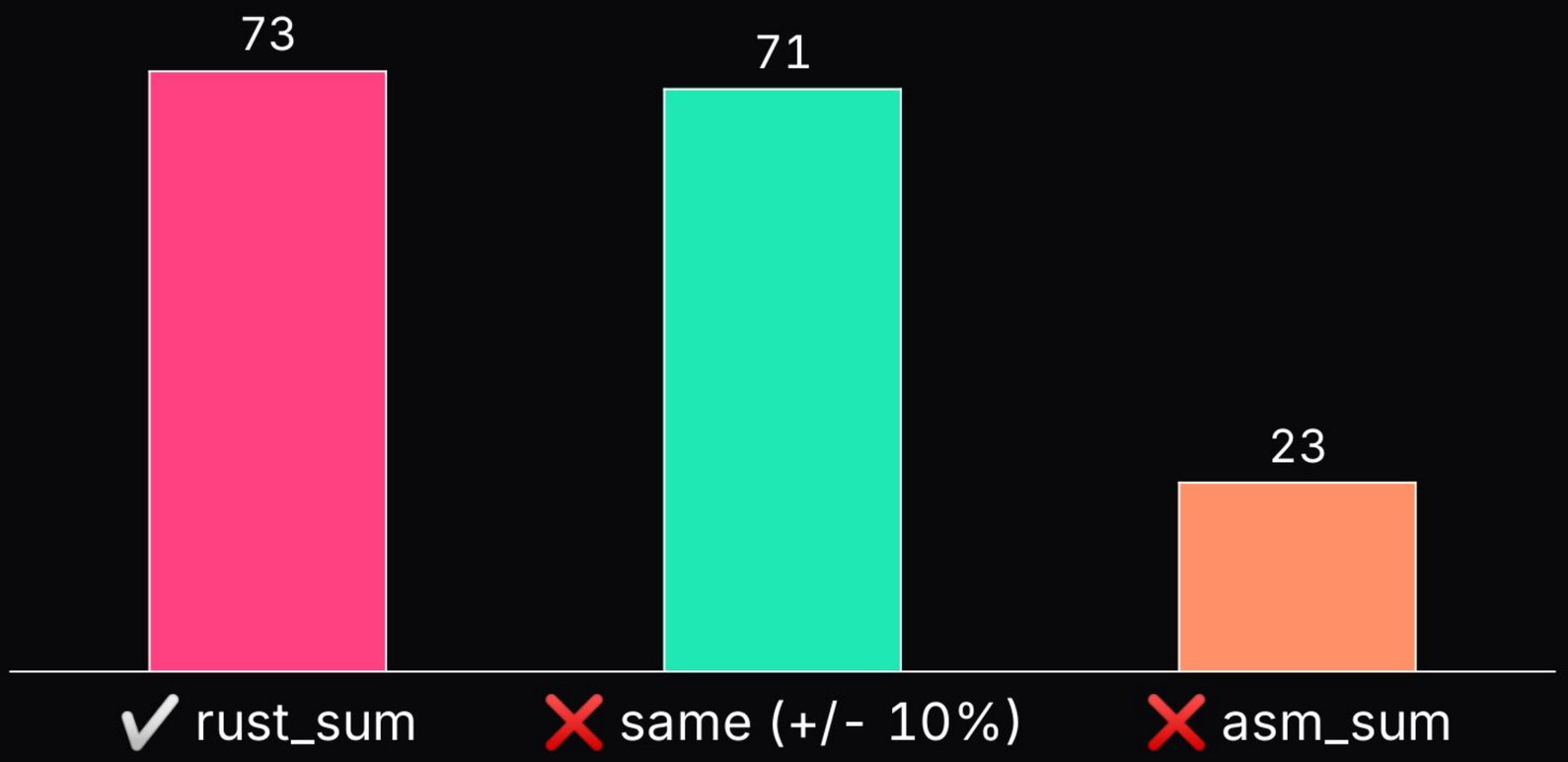
Compiler optimizes and uses loop unrolling on the constant `for` loop

Case study #4: Inline Assembly

```rust
pub fn rust_sum(mut acc: u64, n: u64) -> u64 {
    for i in 0..n {
        acc = acc.wrapping_add(i);
    }
    acc
}

pub fn asm_sum(mut acc: u64, n: u64) -> u64 {
    for i in 0..n {
        unsafe {
            core::arch::asm!(
                "add {acc}, {i}",
                acc = inout(reg) acc,
                i = in(reg) i,
                options(nostack, nomem, preserves_flags),
            );
        }
    }
    acc
}
```

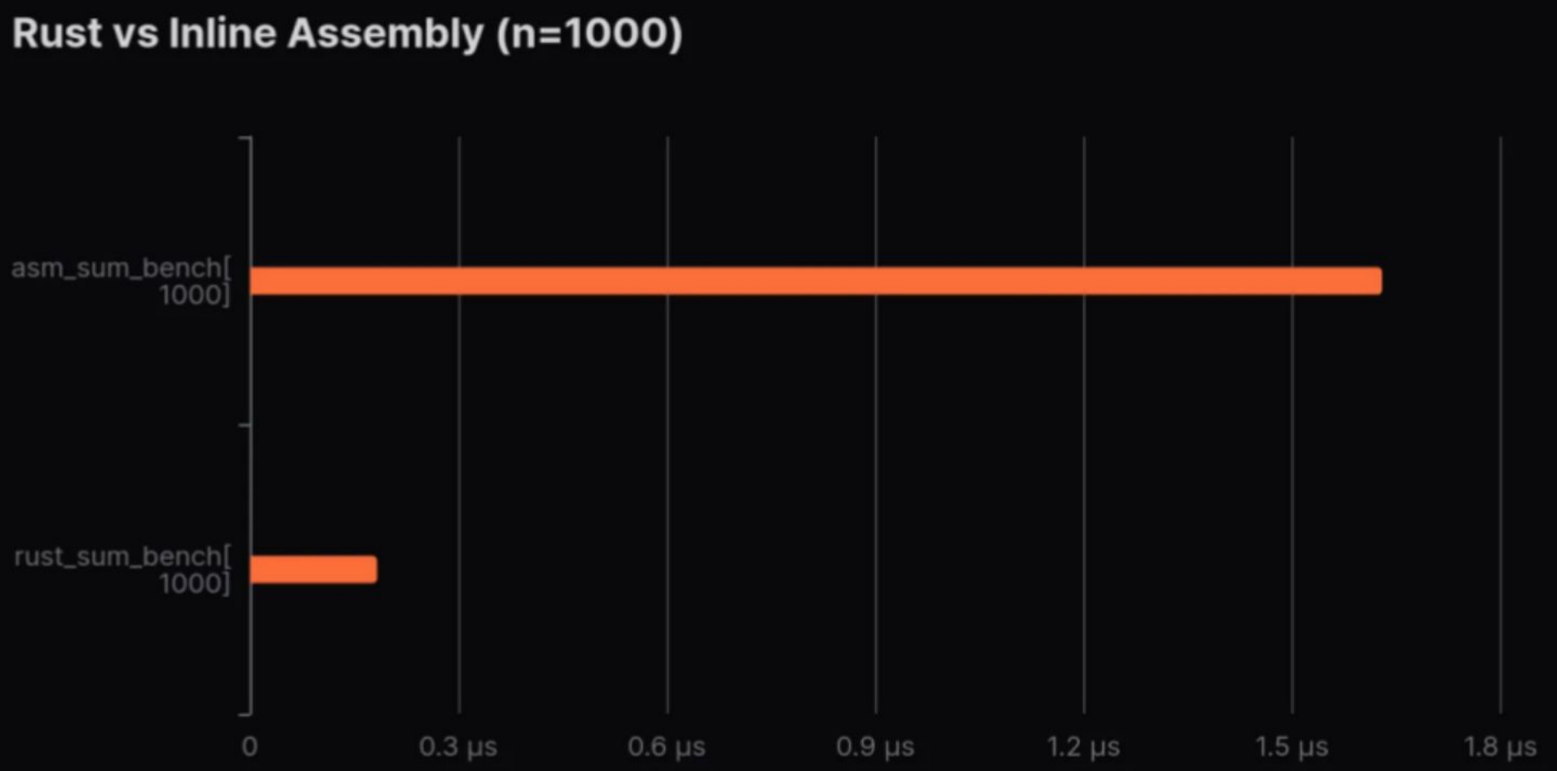
Which is faster?

```rust
pub fn rust_sum(mut acc: u64, n: u64) -> u64 {
    for i in 0..n {
        acc = acc.wrapping_add(i);
    }
    acc
}

pub fn asm_sum(mut acc: u64, n: u64) -> u64 {
    for i in 0..n {
        unsafe {
            core::arch::asm!(
                "add {acc}, {i}",
                acc = inout(reg) acc,
                i = in(reg) i,
                options(nostack, nomem, preserves_flags),
            );
        }
    }
    acc
}
```



{codspeed}

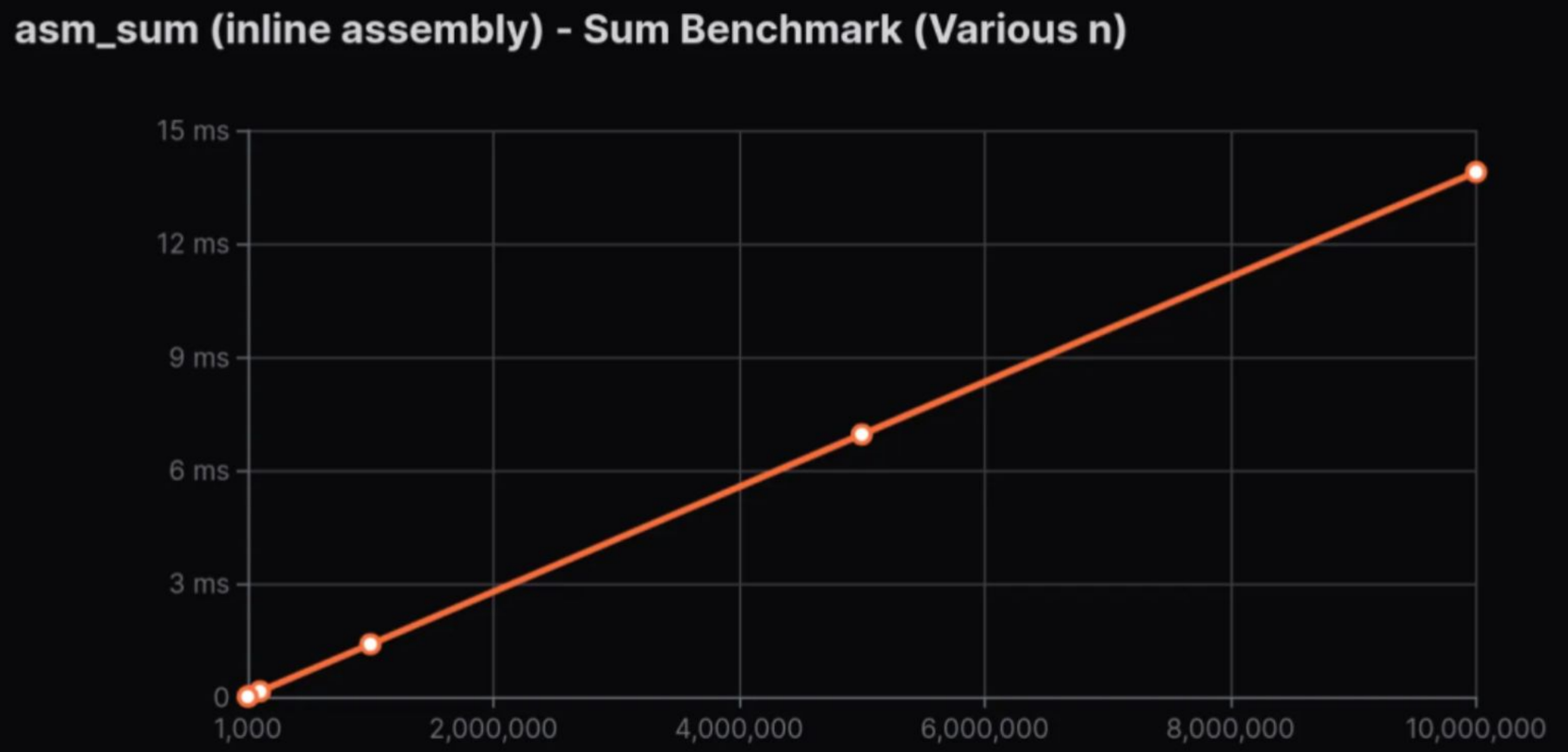
Answer



`rust_sum` is faster

{codspeed}

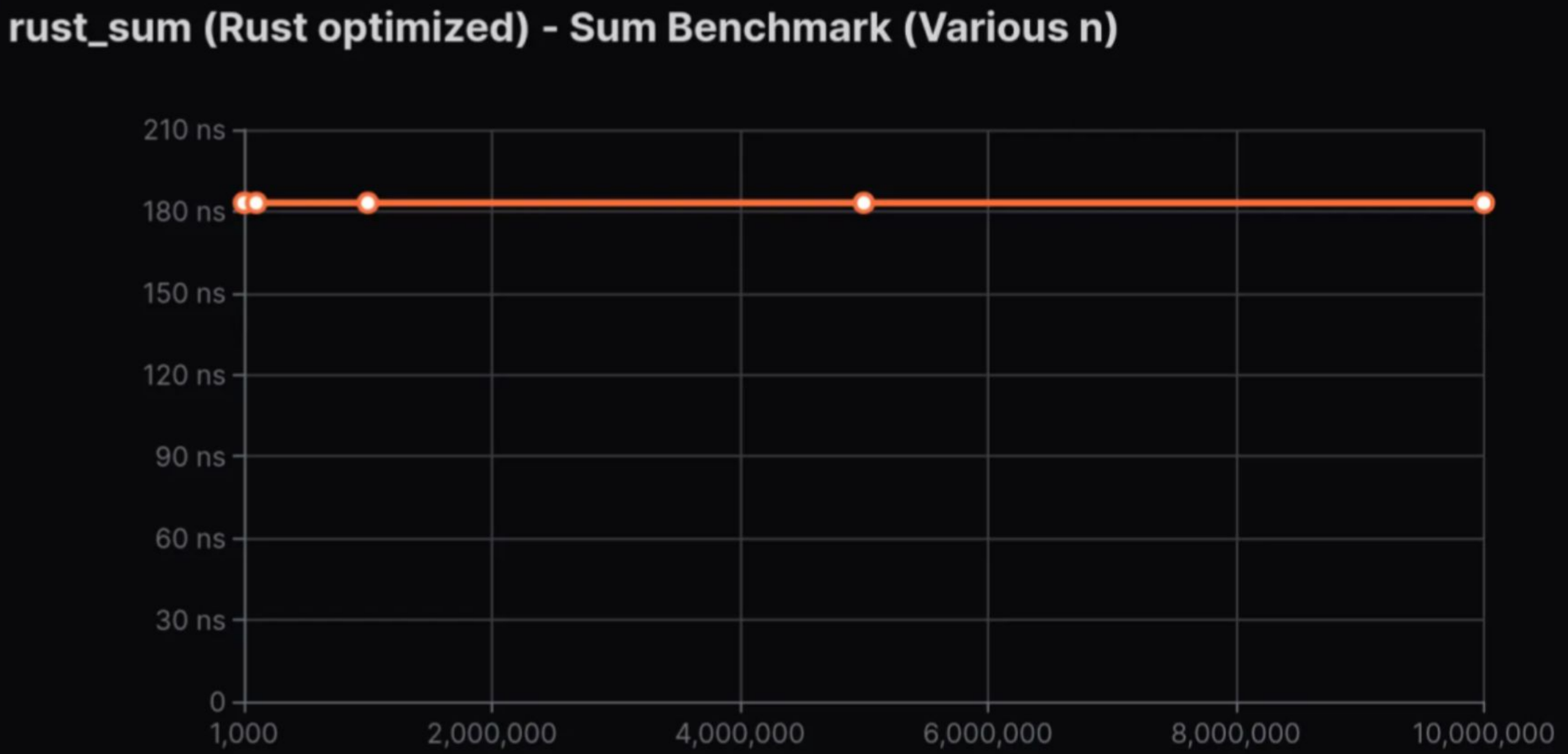
Answer



`asm_sum` is `O(n)`

{codspeed}

Answer



`rust_sum` is `O(1)`

Why?

```rust
pub fn asm_sum(mut acc: u64, n: u64) -> u64 {
    for i in 0..n {
        unsafe {
            core::arch::asm!(
                "add {acc}, {i}",
                acc = inout(reg) acc,
                i = in(reg) i,
                options(nostack, nomem, preserves_flags),
            );
        }
    }
    acc
}
```

```asm
asm_sum:
        mov     rax, rdi            ; rax = a (result = a)
        test    rsi, rsi            ; check if n == 0
        je      .LBB1_3             ; if n == 0 → return a
        xor     ecx, ecx            ; i = 0  (loop counter)

.LBB1_2:
        add     rax, rcx            ; result += i
        lea     rdx, [rcx + 1]      ; next_i = i + 1
        mov     rcx, rdx            ; i = next_i
        cmp     rsi, rdx            ; compare i with n
        jne     .LBB1_2             ; if i ≠ n, continue loop

.LBB1_3:
        ret                         ; return result (in rax)
```

jumps → asm_sum is O(n)

Why?

```rust
pub fn rust_sum(mut acc: u64, n: u64) -> u64 {
    for i in 0..n {
        acc = acc.wrapping_add(i);
    }
    acc
}
```

```asm
rust_sum:
        test    rsi, rsi                ; check if n == 0
        je      .LBB0_2                 ; if yes, jump to return a
        add     rdi, rsi                ; rdi = a + n
        lea     rax, [rsi - 1]          ; rax = n - 1
        add     rsi, -2                 ; rsi = n - 2
        mul     rsi                     ; multiply: rdx:rax = (n - 1) * (n - 2)
        shld    rdx, rax, 63            ; rdx = ((n - 1)*(n - 2)) >> 1  (div by 2)
        add     rdi, rdx                ; rdi += (n - 1)*(n - 2)/2
        dec     rdi                     ; rdi -= 1  → completes to a + n*(n-1)/2

.LBB0_2:
        mov     rax, rdi                ; return value = rdi
        ret
```

No branching in this code except the
first check if n = 0 → rust_sum is O(1)

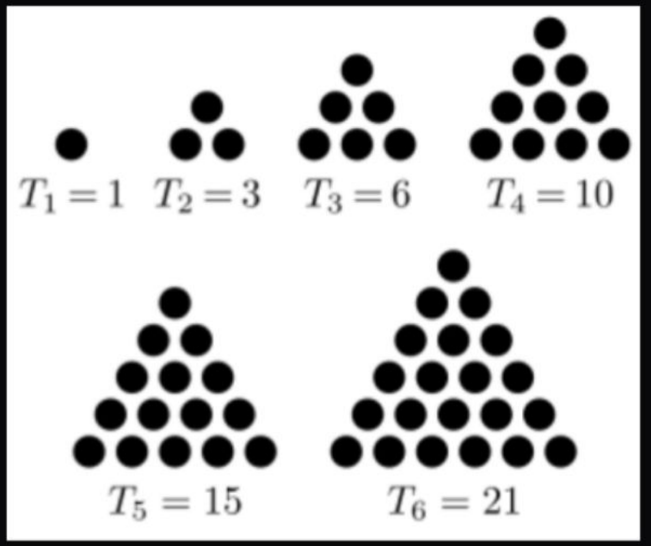
Why?

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2},$$

Triangular Sequence Forumula

```rust
pub fn rust_sum(mut acc: u64, n: u64) -> u64 {
    for i in 0..n {
        acc = acc.wrapping_add(i);
    }
    acc
}
```

No branching in this code except the first check if n = 0 → rust_sum is O(1)



First Triangular Numbers

Case study #5: Bound checks

Sum the elements of an array of integers

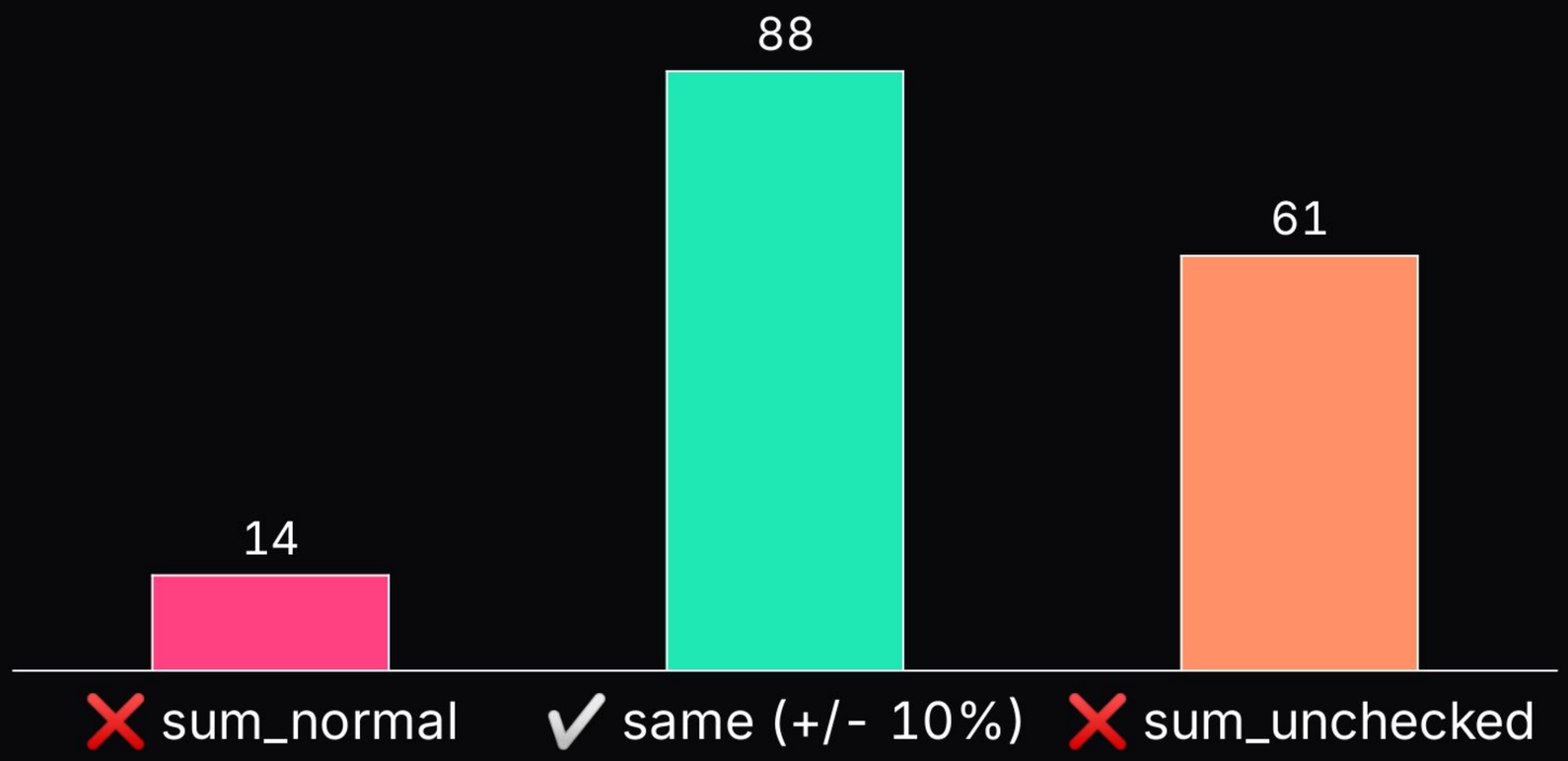
`arr` size 10000 with random integers from 1 to 1000

```rust
pub fn sum_normal(arr: &[u32]) -> u32 {
    let mut sum = 0u32;
    for &value in arr {
        sum = sum + value;
    }
    sum
}

fn sum_unchecked(arr: &[u32]) -> u32 {
    let mut sum = 0u32;
    for &value in arr {
        sum = unsafe { sum.unchecked_add(value) };
    }
    sum
}
```

Which is faster?

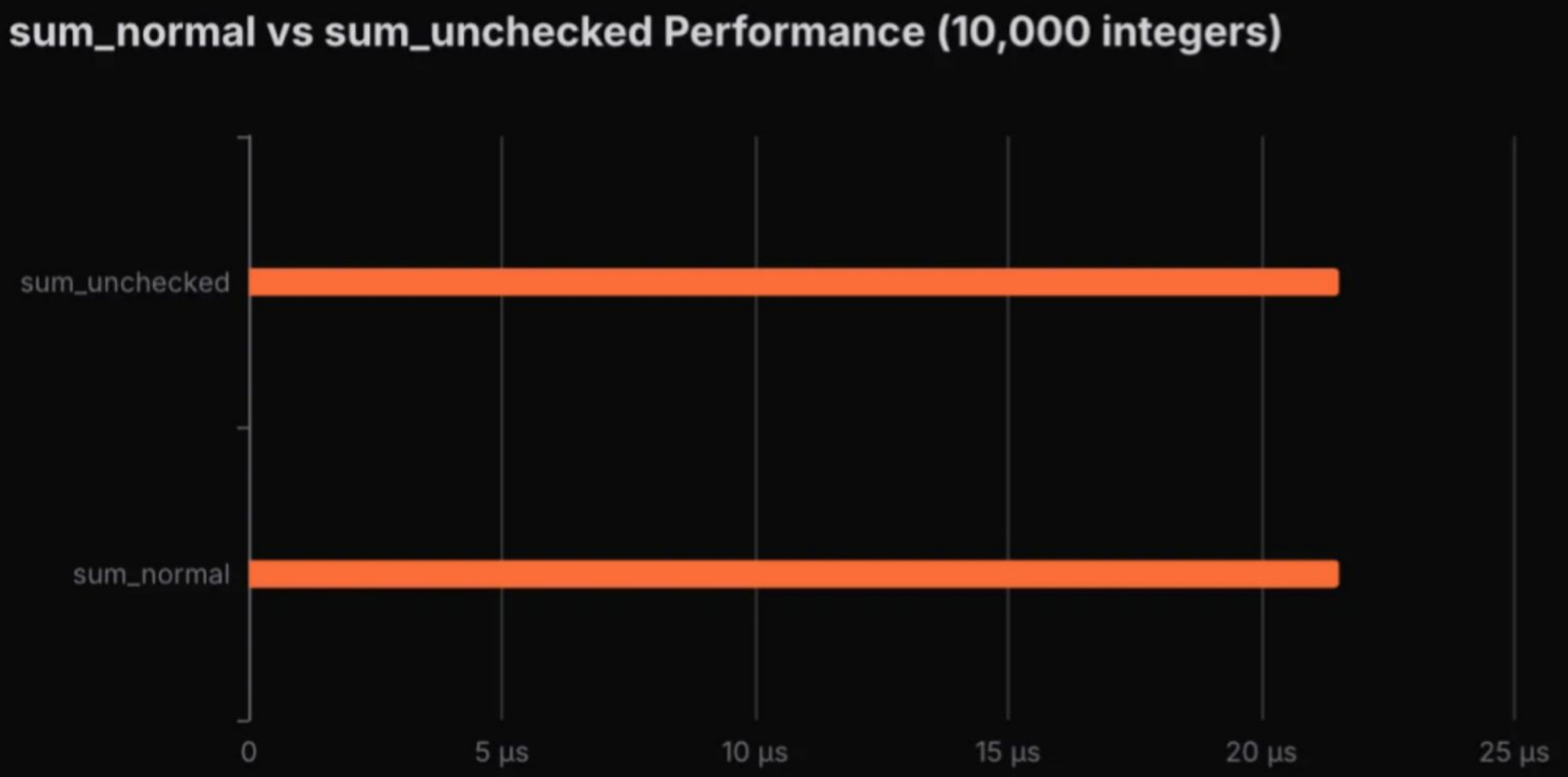
`arr` size 10000 with random integers from 1 to 1000

```rust
pub fn sum_normal(arr: &[u32]) -> u32 {
    let mut sum = 0u32;
    for &value in arr {
        sum = sum + value;
    }
    sum
}

fn sum_unchecked(arr: &[u32]) -> u32 {
    let mut sum = 0u32;
    for &value in arr {
        sum = unsafe { sum.unchecked_add(value) };
    }
    sum
}
```



{codspeed}

Answer



Same performance

Why?

Rust's integer overflow behavior depends on the build profile:

```
[profile.dev]
overflow-checks = true
...
```

Check overflow and panic

```
[profile.release]
overflow-checks = false
...
```

Wrap silently without check

So in release profile, `x + y` is equivalent to `x.unchecked_add(y)`

Enforcing overflow checks

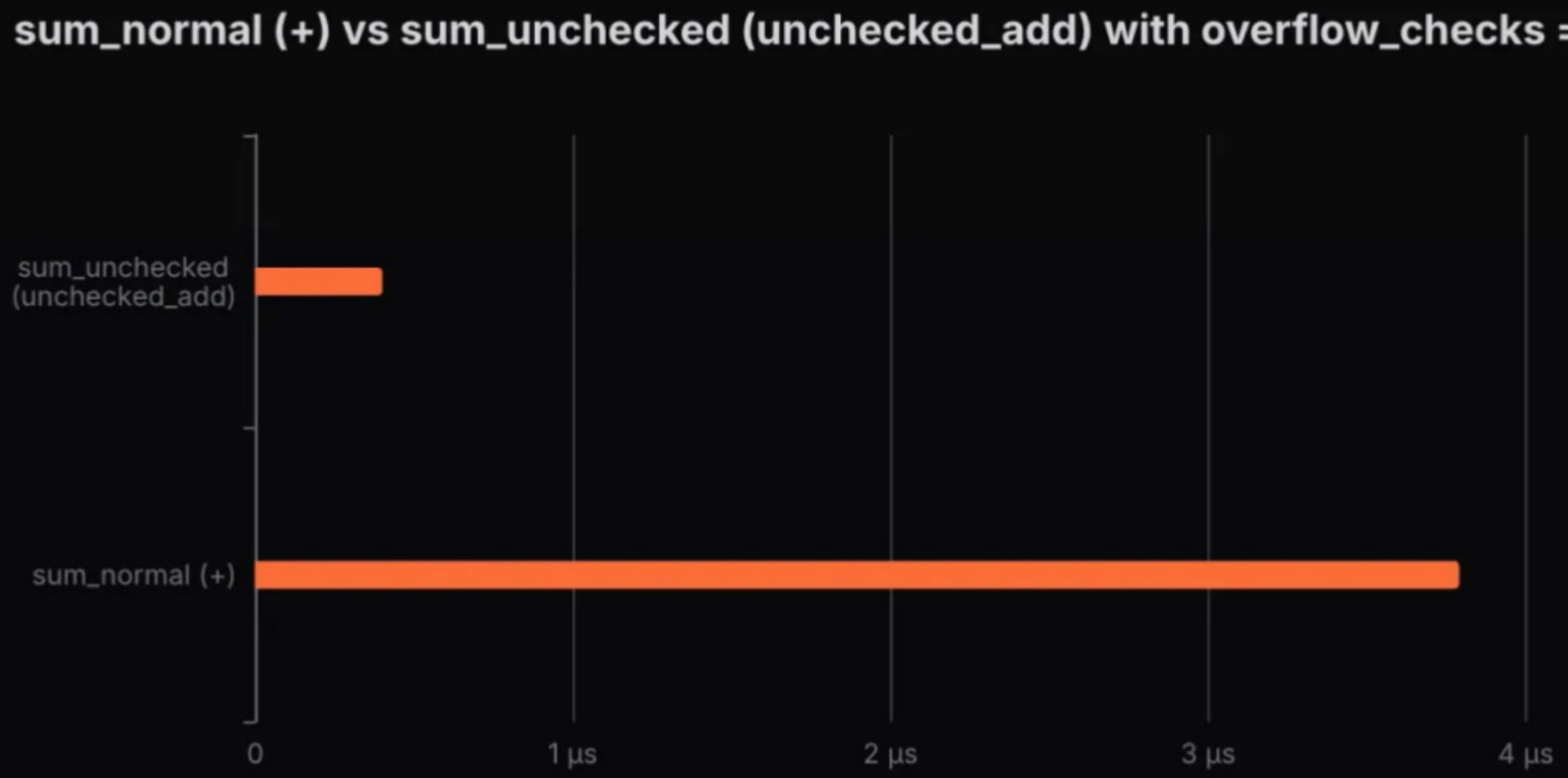
Let's change the profile to enforce overflow checks

```
[profile.release]
overflow-checks = true
...
```

Rerun the benchmark

```rust
pub fn sum_normal(arr: &[u32]) -> u32 {
    let mut sum = 0u32;
    for &value in arr {
        sum = sum + value;
    }
    sum
}

fn sum_unchecked(arr: &[u32]) -> u32 {
    let mut sum = 0u32;
    for &value in arr {
        sum = unsafe { sum.unchecked_add(value) };
    }
    sum
}
```

Enforcing overflow checks - Results



`+` is now indeed way slower (~10x)

Be careful about integer overflow

This compiles and runs fine in `release`

```rust
fn main() {
    let a = u32::MAX;
    println!("{a}");
    let b = a + 1;
    println!("{b}");

    assert!(b == 0);
}
```

stdout

```
4294967295
0
```

Ariane flight V88 exploded on 4 June 1996
because of inadequate overflow protection

Resources: rust-playground

Case study #6: Bound checks with slices

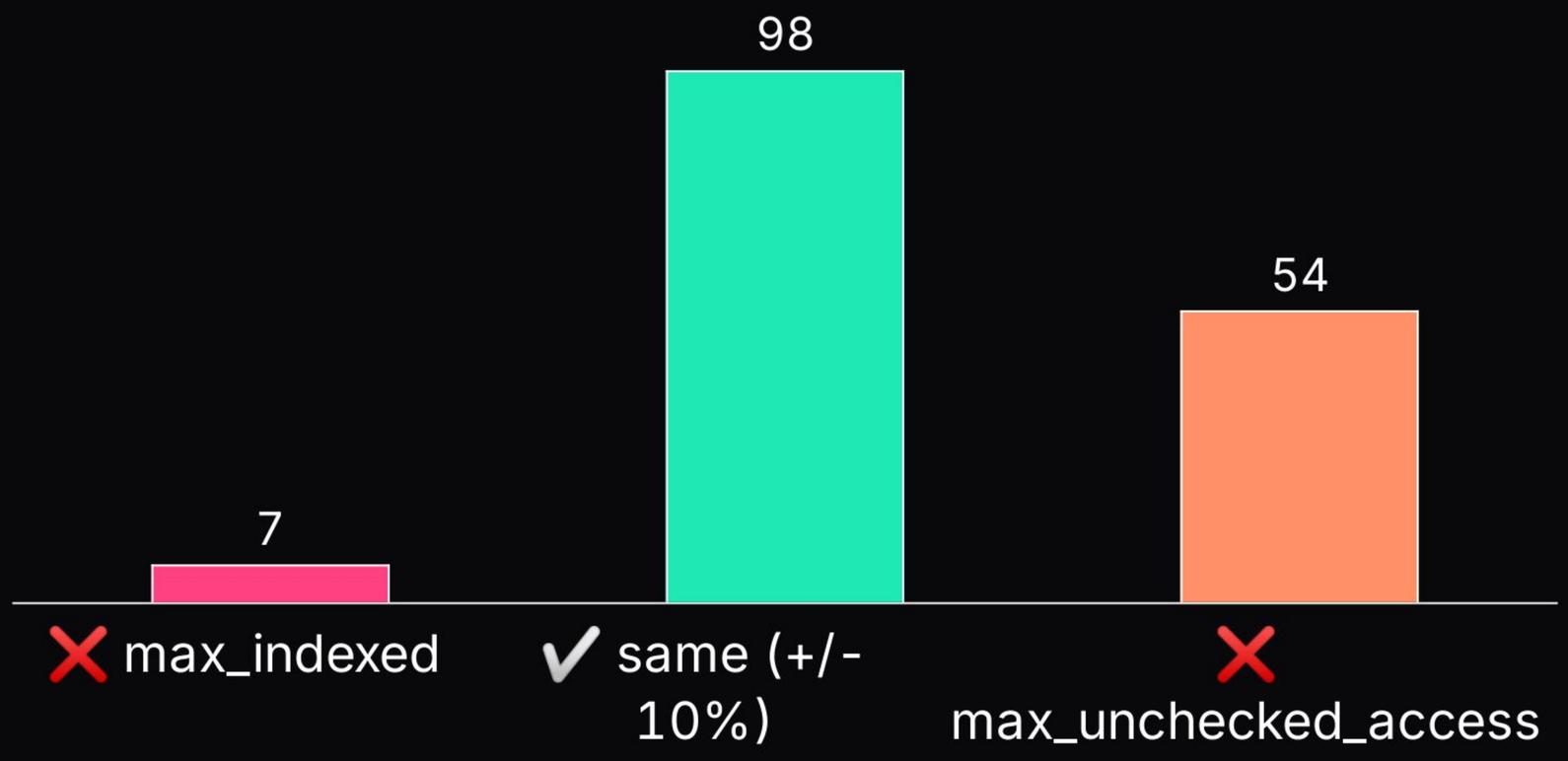
`arr` size 10000 with random integers from 1 to 1000

Get the max item of a slice

```rust
fn max_indexed(arr: &[u32]) -> u32 {
    let mut max = arr[0];
    for i in 1..arr.len() {
        if arr[i] > max {
            max = arr[i];
        }
    }
    max
}

fn max_unchecked_access(arr: &[u32]) -> u32 {
    let mut max = unsafe { *arr.get_unchecked(0) };

    for i in 1..arr.len() {
        let value = unsafe { *arr.get_unchecked(i) };
        if value > max {
            max = value;
        }
    }
    max
}
```

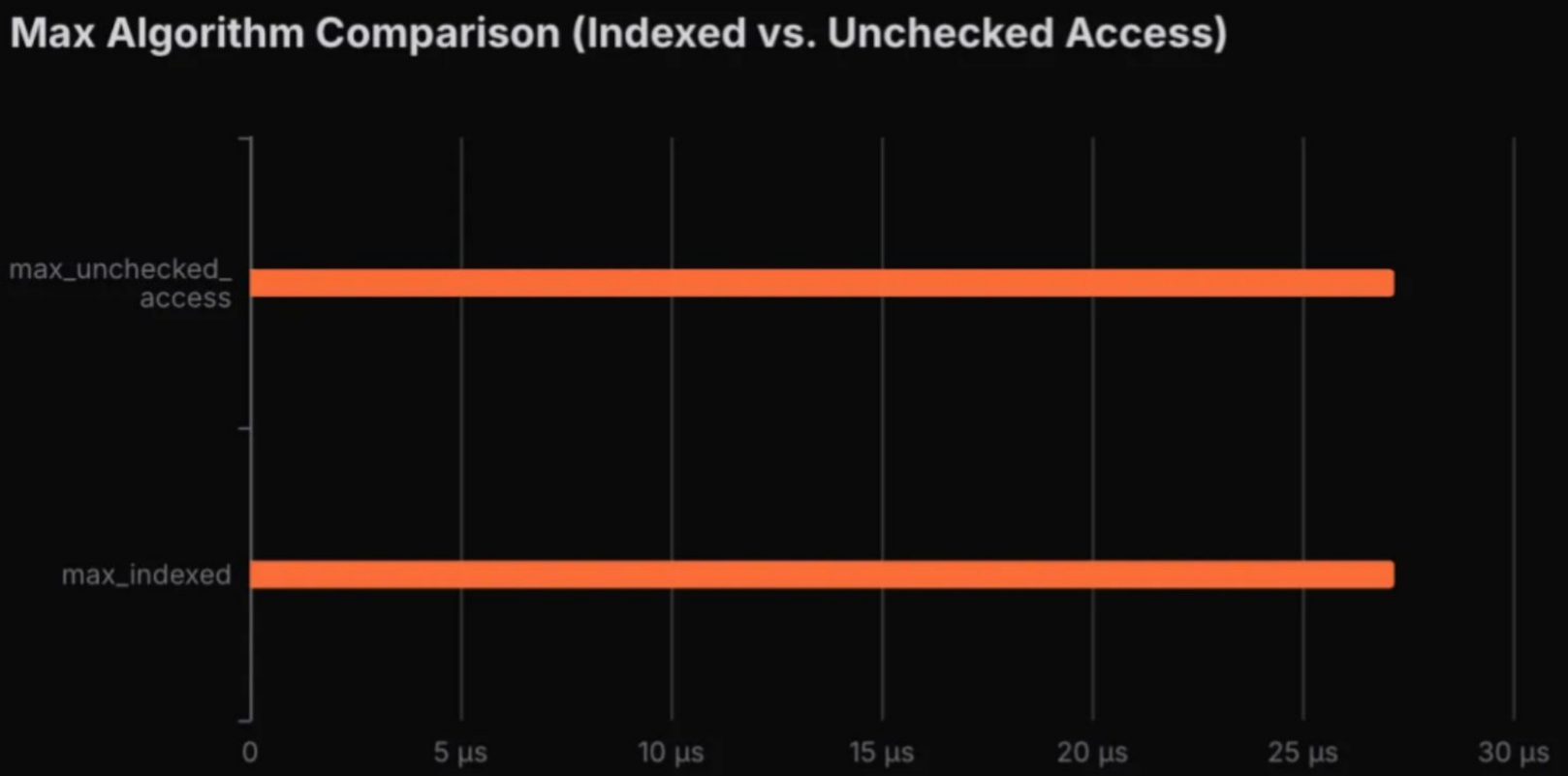
Which is faster?

```rust
fn max_indexed(arr: &[u32]) -> u32 {
    let mut max = arr[0];
    for i in 1..arr.len() {
        if arr[i] > max {
            max = arr[i];
        }
    }
    max
}

fn max_unchecked_access(arr: &[u32]) -> u32 {
    let mut max = unsafe { *arr.get_unchecked(0) };

    for i in 1..arr.len() {
        let value = unsafe { *arr.get_unchecked(i) };
        if value > max {
            max = value;
        }
    }
    max
}
```



{codspeed}

Answer



Same performance

Why

The compiler recognizes the pattern and can prove that `i` will stay in the bounds of the slice.

```
for i in 1..arr.len() {
    if arr[i] > max {
        max = arr[i];
    }
}
```

It removes the checks, making `arr[i]` equivalent to `get_unchecked(i)`

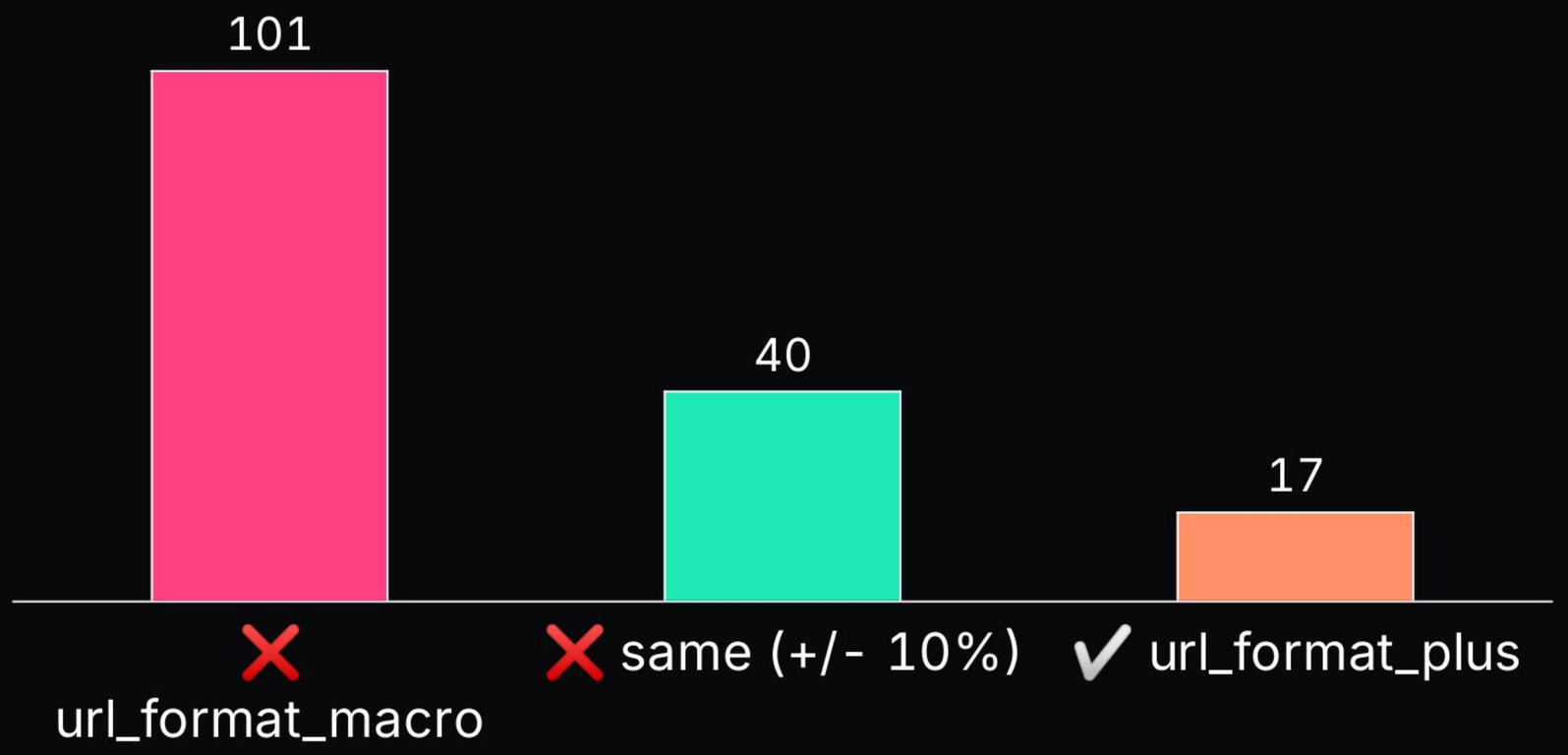
Case study #7: String concatenation

```rust
pub fn url_format_macro(scheme: &str, host: &str, path: &str, query: &str, fragment: &str) -> String {
    format!("{scheme}://{host}{path}?{query}#{fragment}")
}
```

```rust
pub fn url_format_plus(scheme: &str, host: &str, path: &str, query: &str, fragment: &str) -> String {
    scheme.to_string() + "://" + host + path + "?" + query + "#" + fragment
}
```

Which is faster?

```rust
pub fn url_format_macro(
    scheme: &str,
    host: &str,
    path: &str,
    query: &str,
    fragment: &str,
) -> String {
    format!("{scheme}://{host}{path}?{query}#{fragment}")
}

pub fn url_format_plus(
    scheme: &str,
    host: &str,
    path: &str,
    query: &str,
    fragment: &str,
) -> String {
    scheme.to_string() + "://" + host + path + "?" + query + "#" + fragment
}
```

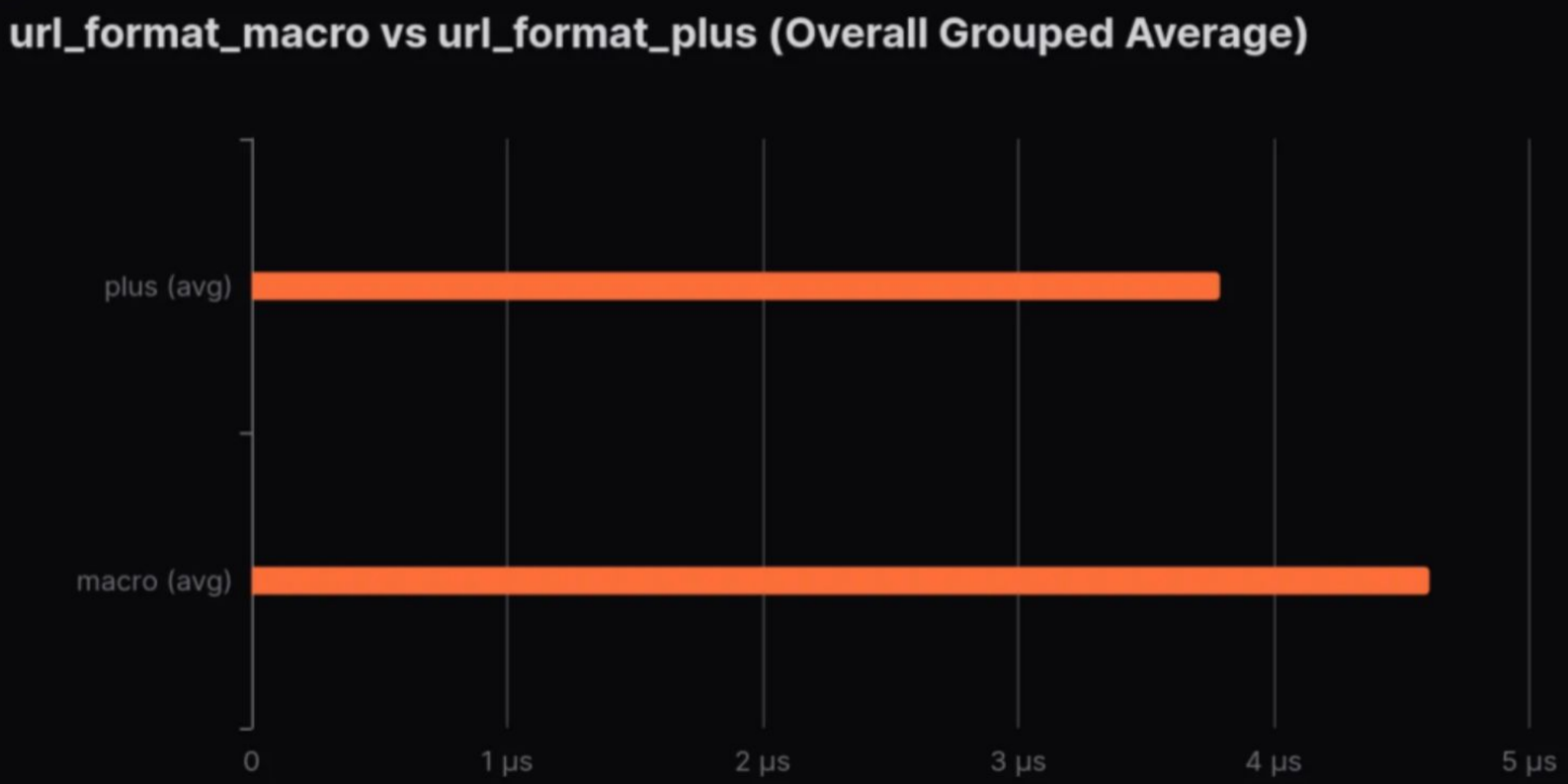


Why?

- **Formatting machinery overhead**:
  - Creating a `fmt::Arguments` object
  - Dynamic dispatch through trait objects
  - Additional abstraction layers to support complex formatting options
- This time the compiler doesn't manage to optimize as much as in other examples

{codspeed}

Answer



`url_format_plus` is faster

But this + chain also has issues

```
scheme.to_string() + "://" + host + path + "?" + query + "#" + fragment
```

```
let temp1 = scheme + "://";
let temp2 = temp1 + host;
let temp3 = temp2 + path;
// ...
```

Every single + operation allocates a new string

String concatenation optimization

```rust
pub fn url_format_push(scheme: &str, host: &str, path: &str, query: &str, fragment: &str) -> String {
    let extra = 3 + 1 + 1; // "://" + "?" + "#"
    let capacity = scheme.len() + host.len() + path.len() + query.len() + fragment.len() + extra;
    let mut url = String::with_capacity(capacity);
    url.push_str(scheme);
    url.push_str("://");
    url.push_str(host);
    url.push_str(path);
    url.push_str("?");
    url.push_str(query);
    url.push_str("#");
    url.push_str(fragment);
    url
}
```

String concatenation optimization results

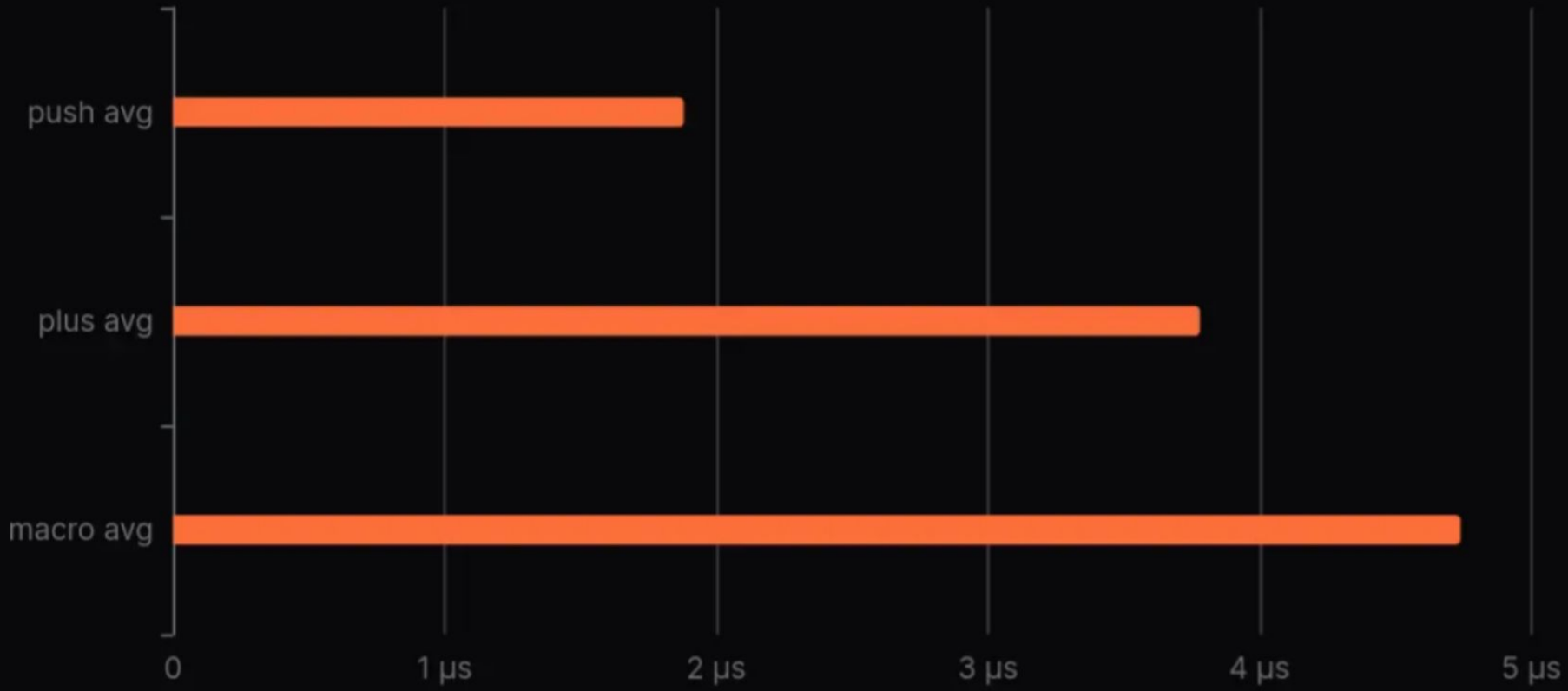
```rust
pub fn url_format_push(scheme: &str, host: &str, path: &str, query: &str, fragment: &str) -> String {
    let extra = 3 + 1 + 1; // "://" + "?" + "#"
    let capacity = scheme.len() + host.len() + path.len() + query.len() + fragment.len() + extra;
    let mut url = String::with_capacity(capacity);
    url.push_str(scheme);
    url.push_str("://");
    url.push_str(host);
    url.push_str(path);
    url.push_str("?");
    url.push_str(query);
    url.push_str("#");
    url.push_str(fragment);
    url
}
```

Allocates memory once

Uses `push_str` to extend the string without creating a new one

{codspeed}

String concatenation optimization results



`url_format_push` is faster: 1.9µs vs 3.8µs vs 4.7µs

Leaderboard

195 players

x 8

🧤 Seb E holds the longest streak record!

Seb E

723 points

1	🧤 Seb E	723p	+723
2	👨‍🔧 David	646p	+646
3	😆 Piotr	641p	+641
4	👩‍🔧 Pascal	612p	+612

Learnings

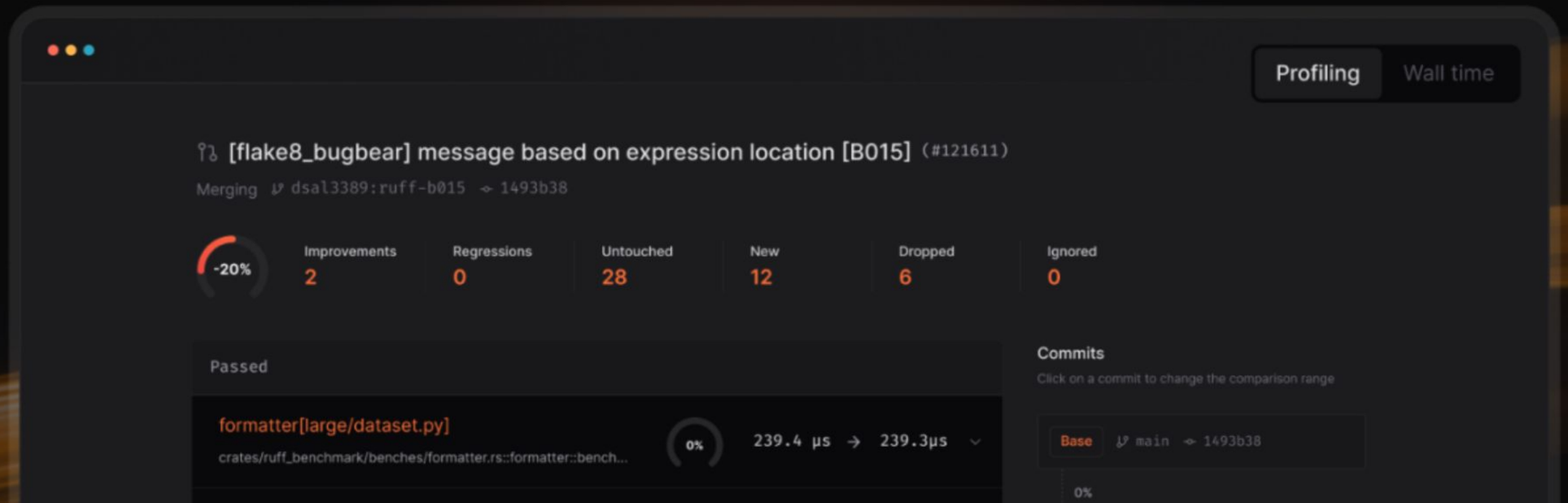
- The compiler optimizes most of the naive patterns for you
- Use const when possible
- Inline assembly can be an optimization killer
- Be careful about integer overflow wrapping by default
- Pre-allocate String of predictable length

Before optimizing, always measure to make sure you're going in the right direction

Be careful about human intuition,
always measure!

{codspeed}

Optimize Performance, Eliminate Regressions



{codspeed}

Free for open source

Loved by performance experts

Vercel CLOUDFLARE Pydantic

ASTRAL ByteDance LangChain

... and many more! (codspeed.io/explore)

{codspeed}

Come by our booth

to continue the discussion!

{codspeed}

Join us for an Happy Hour! 🍕🍻 Tonight at 7pm

Food and drinks are on us!

Registration