Производительность языка Rust

May 2026

Об авторах

Юрий Грибов (yugr, the_real_yugr), https://github.com/yugr

- Разработчик и фанат системного ПО (компиляторов, рантаймов, инструментов верификации и т. п.)
- Руководитель компиляторной команды

Захар Акимов, https://github.com/z-inform

- Разработчик general-purpose и AI компиляторов
- Эксперт по ML in compilers

Особенности языка Rust

- Язык для высокопроизводительного системного программирования
  - Zero cost/overhead abstractions: don’t pay for what you don’t use + no (or minimal) runtime overhead (итераторы, closures, traits, etc.)
  - Возможность низкоуровневого тюнинга (SIMD, inline assembler, компиляторные интринсики типа __builtin_expect, etc.)
- Поддерживает современные парадигмы программирования
  - Процедурное, функциональное и объектно-ориентированное программирование
  - Статический и динамический полиморфизм
  - Метапрограммирование
- Язык сокращает количество видов UB за счёт более строгих правил и динамических проверок
  - Особенно memory safety

Производительность Rust: Минусы

- Проверки в рантайме
- Обязательная инициализация
- Целочисленные переполнения являются defined behavior
- Компромиссы между производительностью и надёжностью в стандартной библиотеке
- ...

Производительность Rust: Плюсы

- Больше возможностей для alias analysis
  - По сути restrict-by-default
- Агрессивные оптимизации структур
  - Оптимизация порядка полей и niche optimizations
- Более эффективные стандартные контейнеры
- …

Цели доклада

https://pxhere.com/ru/photo/870592

- Насколько “дороги” проверки Rust на практике?
  - В сравнении с Rust без проверок и C/C++
  - Каковы непосредственные (лишние инструкции) и вторичные (влияние на другие оптимизации, давление на I$/BTB) последствия проверок
- Способен ли от них избавиться Sufficiently Smart (Heroic) Compiler © ?
- И если нет, то как с ними может справиться программист?
- Есть ли какие-то особенности языка которые наоборот помогают компилятору?

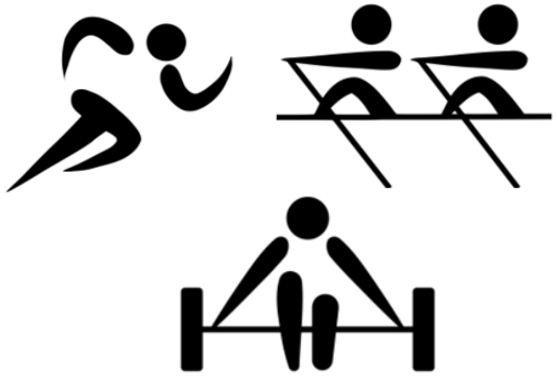
Мнение создателей языка

- Steven Klabnik:
  - "Generally Rust is the same order of magnitude as C. Sometimes faster, sometimes slower”
  - “Rust prioritizes memory safety above all else. But speed is a close second"

Анализ накладных расходов

- Как оценить накладные расходы от проверок?
- В компиляторе Rust нет флагов для отключения проверок безопасности
- Мы сравним обычный Rust и пропатченный Rust с отключенными проверками
  - Репозиторий с модифицированными версиями компилятора: yugr/rust-private
  - Базовая ветка yugr/baseline основана на теге 1.87.0
    - Для стабильности измерений отключена рандомизация хэштаблиц и усилено выравнивание функций

Бенчмарки



- Набор бенчмарков компилятора Rust
- Внутренние бенчмарки крупных open-source проектов из разных областей:
  - Gamedev (Bevy, Veloren)
  - Базы данных (SpacetimeDB)
  - Работа с текстом (Zed, Ruff, regex)
  - Кодеки (rav1e, oxipng)
  - Линейная алгебра (nalgebra)
  - Асинхронный runtime (tokio)
  - Пакетные менеджеры (uv)
- Многие бенчмарки уже были сильно оптимизированы (с помощью unsafe и т.п.)

Akovin, https://commons.wikimedia.org/wiki/File:Olympic_types_of_sport_with_icons.png

Rust и C++

- Rust и C++ близки по выразительным возможностям:
  - Поддержка нескольких парадигм программирования (процедурной, ООП и функциональной)
  - Статический и динамический полиморфизм
  - Etc.
- Область применения языков совпадает – высокопроизводительное системное программирование
  - Zero cost/overhead abstractions
  - Возможность низкоуровневого тюнинга
- Те же backend-оптимизаторы (LLVM, GCC) и технологии оптимизации (PGO, LTO, etc.)
- В “безопасном” диалекте C++ (Hardened C/C++) есть аналогичные рантайм-проверки

Чего не будет в докладе: Другие аспекты языков

- Эргономичность, выразительность (в частности self-referential data structures), безопасность, etc.
- На эти темы есть множество других докладов!
  - Rust Features that I Want in C++ (David Sankel, CppNow 2022)
  - The existential threat against C++ and where to go from here (Helge Penne, NDC 2024)
  - Rust vs C++ with Steve Klabnik and Herb Sutter (SW Engineering Daily podcast)

Чего не будет в докладе: Параллельный код

- Одним из преимуществ Rust считается простота написания корректного параллельного кода
  - Fearless Concurrency with Rust
- Тема требует отдельного рассмотрения и глубокой экспертизы
  - C++ Russia 2027 anyone?

Чего не будет в докладе: Небезопасный код

- Язык Rust состоит из двух частей:
  - Безопасный код (*б*ольшая часть кода на Rust)
  - Небезопасный код:
    - “Things the Rust authors will implore you not to do” (Rustonomicon)
    - “Aspects of the language you may not use every day and useful in very specific situations” (Rust Book)
    - В основном предназначен для оптимизации горячих участков кода
    - Используется для тюнинга в реальных проектах (в том числе и рассматриваемых нами бенчмарках)
- В этом докладе мы фокусируемся на оптимизациях для *безопасного* кода
  - (А также без third-party библиотек, SIMD и компиляторных интринсиков)
  - Небезопасные конструкции лишают язык его основного преимущества
  - Небезопасные оптимизации хорошо известны и (как правило) очевидны

Tricia J, https://www.flickr.com/photos/cazatoma/8746664127

Чего не будет в докладе: Неидиоматичный код

- Nikita Popov, lead-мэйнтэйнер LLVM:
  - “Languages like C, C++ and Rust are by-design on equal footing“
  - “You can only compare how easy it is to write performant code or how performant idiomatic code is”
- Scott McMurray, Rust compiler team:
  - “Clang and rustc are both producing LLVM IR and running mostly the same LLVM optimization passes”
  - “Given enough work they can always produce essentially the same thing, making the comparisons pointless”

Bounds checks

Детальный разбор

Переполнения буфера

- Morris Worm (1988)
- Запись чрезмерно большого объёма данных в переменную программы

```
char local_buf[32];
sprintf(buf, “Message from user: %s”, received_data);
```

- Переполнение стека (stack overflow)
  - Атаки Stack Smashing, Return-to-libc, Return-Oriented Programming
  - Наиболее стандартизованный и технологичный вид атак
- Переполнение кучи (heap overflow)

https://www.rawpixel.com/image/5958324/free-public-domain-cc0-photo

Распространённость buffer overflow уязвимостей

- Лидирующие позиции в рейтинге наиболее опасных уязвимостей
  - Mitre CWE Top 25 2024 (места 2, 6, 20)
- 70% уязвимостей в продуктах Microsoft и Chromium вызваны ошибками памяти
- До 80% ошибок памяти связано с buffer overflow
  - 80% CVE и 46% KEV в 2024
  - 40% атак (Google Project Zero, 2024)

Memory safety в Rust

- Фронтенд генерирует проверки стандартных индексных доступов
  - Массивы, slices, контейнеры, etc.
  - Проверки могут быть удалены если оптимизатор может доказать безопасность
- Также используются более безопасные алгоритмы в стандартной библиотеке
  - Например сортировка не упадёт при использовании некорректного компаратора

Nick Youngson, https://www.picpedia.org/highway-signs/s/safety.html

Накладные расходы

- Выполнение сравнения и условного перехода (1-2 ALU слота)
- Дополнительный регистр для хранения длины для сравнения
- Давление на I$ (промахи в кэш) из-за кода проверки и обработчика паники
  - Давление на branch predictor для never-taken переходов практически отсутствует
- Отключение других оптимизаций
  - Прежде всего векторизатора

Оптимизации в компиляторе

- Во многих случаях компилятор способен
  - Гарантировать корректность доступа и удалить проверки
  - Или вынести проверки из цикла (O(N) -> O(1))
- Эти оптимизации делаются на уровне LLVM
  - InstCombine, SimpleLoopUnswitch, LoopVectorize, etc.
  - Также должны оптимизировать Hardened C/C++
- Оптимизатор был сильно улучшен и многие проблемы прошлых лет успешно оптимизируются в текущей версии Rust
  - Например известные примеры Алекса Кладова (matklad)

Успешная оптимизация

Все проверки вынесены в пролог функции

```rust
pub fn foo(a: &[i32], b: &[i32]) -> i32 {
    let mut ans = 0;
    let n = a.len();
    for i in 0..n {
        ans += a[i] * b[i];
    }
    ans
}
```

```
foo:
        .cfi_startproc
        test    rsi, rsi
        je      .LBB0_1  // n == 0 => вернуть 0
        lea     rax, [rsi - 1]
        cmp     rcx, rax
        jbe     .LBB0_6  // b.len() < n => паника
        xor     eax, eax
        xor     ecx, ecx
        .p2align        4
.LBB0_4:                 // Цикл (без проверок!)
        mov     r8d, dword ptr [rdx + 4*rcx]
        imul    r8d, dword ptr [rdi + 4*rcx]
        lea     r9, [rcx + 1]
        add     eax, r8d
        mov     rcx, r9
        cmp     rsi, r9
        jne     .LBB0_4
...
```

https://godbolt.org/z/aY744zdfe

Использовались флаги
-Cno-vectorize-loops
-Cno-vectorize-slp
-Cllvm-args=-unroll-allow-remainder=0

Неуспешная оптимизация

Проверки остались в цикле

```rust
pub fn foo(v: &[i32]) -> i32 {
    let mut ans = 0;
    let chunk_len = 4;
    let num_chunks = v.len() / chunk_len;
    for i in 0..num_chunks {
        for j in 0..chunk_len {
            ans += v[i * chunk_len + j];
        }
    }
    ans
}
```

```
bar:
        ...
        .p2align        4
.LBB0_2:                // Цикл (с проверками!)
        lea     rcx, [r8 - 3]
        cmp     rcx, rsi
        jae     .LBB0_7
        lea     rcx, [r8 - 2]
        cmp     rcx, rsi
        jae     .LBB0_7
        lea     rcx, [r8 - 1]
        cmp     rcx, rsi
        jae     .LBB0_7
        cmp     r8, rsi
        jae     .LBB0_6
        add     eax, dword ptr [rdi + 4*r8 - 12]
        add     eax, dword ptr [rdi + 4*r8 - 8]
        add     eax, dword ptr [rdi + 4*r8 - 4]
        add     eax, dword ptr [rdi + 4*r8]
        add     r8, 4
        dec     rdx
        jne     .LBB0_2
```

https://godbolt.org/z/176s9nEfE

Использовались флаги
-Cno-vectorize-loops
-Cno-vectorize-slp
-Cllvm-args=-unroll-allow-remainder=0

Проблемы в оптимизаторе: delayed panics

- В Rust RFC 560 было предложено разрешить компилятору комбинировать вместе связанные проверки
  - Если это полезно для оптимизации
  - И поведение программы не изменится (с точностью до сообщения об ошибке)
- Похоже на классическую концепцию Imprecise Exceptions
- Решение о допустимости подобных оптимизаций пока не принято и в случае индексных проверок эта оптимизация сейчас не делается ☹
- Пример: rust #50759

```
use std::fs::{read, write};

fn main() -> Result<(), std::io::Error> {
    let h = read("/tmp/test")?;
    // Каждый доступ проверяется отдельно ☹
    // (достаточно было бы проверить h[15])
    let b = [h[0], h[1], ... h[15]];
    write("/tmp/test", &b)?;
    Ok(())
}
```

Workarounds: Общие соображения

- В циклах используйте слайсы, а не контейнеры типа Vec
  - Это упрощает работу alias analysis
  - Пример: fschutt/fastblur #3
- Избегайте сложной индексной арифметики
  - На практике всё что сложнее прибавления константы или сложения двух индексных переменных
  - Даже аффинные операции: Taking Advantage of Auto-Vectorization in Rust
- В нетривиальных случаях не полагайтесь на оптимизатор
  - Его возможности могут (немонотонно) меняться в разных версиях компилятора
  - “Auto-vectorization is not a programming model”
- Эти же советы актуальны и для Hardened C/C++!

Workarounds

- Удаление проверок
- Сокращение числа проверок
- Снижение стоимости проверок

Workarounds

- **Удаление проверок**
- Сокращение числа проверок
- Снижение стоимости проверок

Workarounds: Unsafe-код

- Наиболее простое и надёжное решение
- Используется внутри контейнеров и итераторов стандартной библиотеки
- Несколько вариантов:
  - Использование указателей
  - Методы без проверок типа get_unchecked
  - Хинты для компилятора std::hint::unreachable_unchecked, std::hint::assert_unchecked

```
fn drop(&mut self) {
    // fill the hole again
    unsafe {
        let pos = self.pos;
        ptr::write(
            self.data.get_unchecked_mut(pos),
            self.elt.take().unwrap());
    }
}
```

rust #36072

https://workinjurysource.com/preventing-slip-and-fall-injuries-at-work/

Workarounds: Использование ограниченных типов

- Ограниченность индекса можно выразить с помощью типов
  - Enums или bool
- Пример: rust #102303

```rust
pub fn foo(array: &[i16; 3],
           i: usize) -> i16 {
    array[i]
}
```

```rust
pub enum Enum {
    A, B, C
}

pub fn foo(array: &[i16; 3],
           i: Enum) -> i16 {
    array[i as usize]
}
```

Workarounds

- Удаление проверок
- **Сокращение числа проверок**
- Снижение стоимости проверок

Workarounds: Ручные asserts

- Позволяет сделать однократную проверку вместо нескольких

```
// optimizer hint to eliminate bounds checks
// within loops
assert!(coefficients.len() == 64);

let mut temp = [Wrapping(0); 64];

// columns
for i in 0..8 {
    if coefficients[i + 8] == 0
        && coefficients[i + 16] == 0
        && coefficients[i + 24] == 0
        && coefficients[i + 32] == 0
        && coefficients[i + 40] == 0
        && coefficients[i + 48] == 0
        && coefficients[i + 56] == 0
    {
        ...
```

jpeg-decoder #167

Workarounds: Reslicing

- Reslicing, preslicing или subslicing
- Раннее конструирование слайсов заданного размера
- Заставляет компилятор сделать *однократную* проверку при создании слайса
- Пример: [dropbox/rust-brotli]

```
for i in 0..n {
    black_box(vec[i]);
}
```

```
let slice = &vec[0..n];
for i in 0..n {
    black_box(slice[i]);
}
```

Lalada, https://www.pexels.com/photo/close-up-of-a-person-cutting-bread-19359855/

Workarounds: Reslicing

- Выделение смещённых слайсов лучше делать в два шага
- Это позволит компилятору не задумываться о возможном переполнении
- Пример: Optimizing rav1d, an AV1 Decoder in Rust

```
&vec[i..(i + n)]
```

```
&vec[i..][..n]
```

Workarounds: Ограничение диапазона итерации

- При итерации над несколькими контейнерами лучше явно использовать общий безопасный диапазон

```rust
pub fn foo(x: &mut [i32], y: &[i32]) {
    let n = x.len();
    for i in 0..n {
        x[i] ^= y[i];
    }
}
```

https://godbolt.org/z/P8de874c1

```
.LBB1_2:
  cmp rcx, rax
  je  .LBB1_5
  mov r8d, dword ptr [rdx + 4*rax]
  xor dword ptr [rdi + 4*rax], r8d
  lea r8, [rax + 1]
  mov rax, r8
  cmp rsi, r8
  jne .LBB1_2
```

```rust
pub fn foo(x: &mut [i32], y: &[i32]) {
    let n = std::cmp::min(x.len(), y.len());
    for i in 0..n {
        x[i] ^= y[i];
    }
}
```

https://godbolt.org/z/6TzEPaaEs

```
.LBB0_2:
  mov     ecx, dword ptr [rdx + 4*rax]
  xor     dword ptr [rdi + 4*rax], ecx
  lea     rcx, [rax + 1]
  mov     rax, rcx
  cmp     rsi, rcx
  jne     .LBB0_2
```

Компиляция с
-Cno-vectorize-loops -Cllvm-args=-unroll-allow-remainder=0

Workarounds: Итераторы

- Многие итераторы используют информацию о размере контейнера для сокращения числа проверок

```rust
pub fn foo(x: &mut [i32], y: &[i32]) {
    let n = x.len();
    for i in 0..n {
        x[i] ^= y[i];
    }
}
```

```rust
pub fn foo(x: &mut [i32], y: &[i32]) {
    // This also works:
    //   for (x, y) in x.iter_mut().zip(y)
    x.iter_mut()
        .zip(y)
        .for_each(|(x, y)| *x ^= *y);
}
```

https://godbolt.org/z/P8de874c1

```
.LBB1_2:
  cmp rcx, rax
  je  .LBB1_5
  mov r8d, dword ptr [rdx + 4*rax]
  xor dword ptr [rdi + 4*rax], r8d
  lea r8, [rax + 1]
  mov rax, r8
  cmp rsi, r8
  jne .LBB1_2
```

https://godbolt.org/z/je69173xv

```
.LBB0_2:
  mov     ecx, dword ptr [rdx + 4*rax]
  xor     dword ptr [rdi + 4*rax], ecx
  lea     rcx, [rax + 1]
  mov     rax, rcx
  cmp     rsi, rcx
  jne     .LBB0_2
```

Компиляция с
-Cno-vectorize-loops -Cllvm-args=-unroll-allow-remainder=0

Недостатки итераторов

- “Miracle of zero cost abstraction has a limit" (fridsun at Reddit)
- Сильное разбухание и замедление debug-кода
- Компилятор может не заинлайнить длинные цепочки итераторов
- Лямбды в итераторах принимают ссылки на элементы, что может вызывать проблемы с LLVM alias analysis (rust #106539)
- Chained-итераторы (chain, flat_map, flatten, ..=, etc.) генерируют неэффективный код при использовании в for-циклах (“external iteration”)
  - Рекомендуется их применять только с .for_each, .fold, .find и т.п. (“internal iteration”)
  - Are iterators even efficient?

Karen Arnolds, https://www.publicdomainpictures.net/en/view-image.php?image=235675&picture=man-carrying-stack-of-boxes

Workarounds: Порядок индексации

- Данные лучше читать начиная с самого дальнего элемента

```
let a = x[0];
let b = x[1];
let c = x[2];
```

```
let c = x[2];
let b = x[1];
let a = x[0];
```

или

```
let a = x[0];
let b = x[1];
let c = x[2];
```

```
let [a, b, c] = data[..3] else {
    panic!()
};
```

или

```
let a = x[0];
let b = x[1];
let c = x[2];
```

```
let [a, b, c] = data[..3]
    .try_into().unwrap();
```

Workarounds: Ручная векторизация

- Программист может вручную переписать код с использованием векторов
- Стандартная библиотека предоставляет
  - Платформо-независимый слой std::simd (nightly only)
  - Платформенные интринсики std::arch (unsafe)
- Также есть масса third-party библиотек

```rust
pub fn sum_simd(data: &[i32]) -> i32 {
    const LANES: usize = 8;  // AVX

    let mut acc = i32x8::splat(0);

    for chunk in data.chunks_exact(LANES) {
        let v = i32x8::from_slice(chunk);
        acc += v;
    }

    let mut tail_acc = 0i32;
    for &x in data.chunks_exact(LANES).remainder() {
        tail_acc += x;
    }

    acc.reduce_sum() + tail_acc
}
```

https://godbolt.org/z/G3bfGEhnG

Workarounds

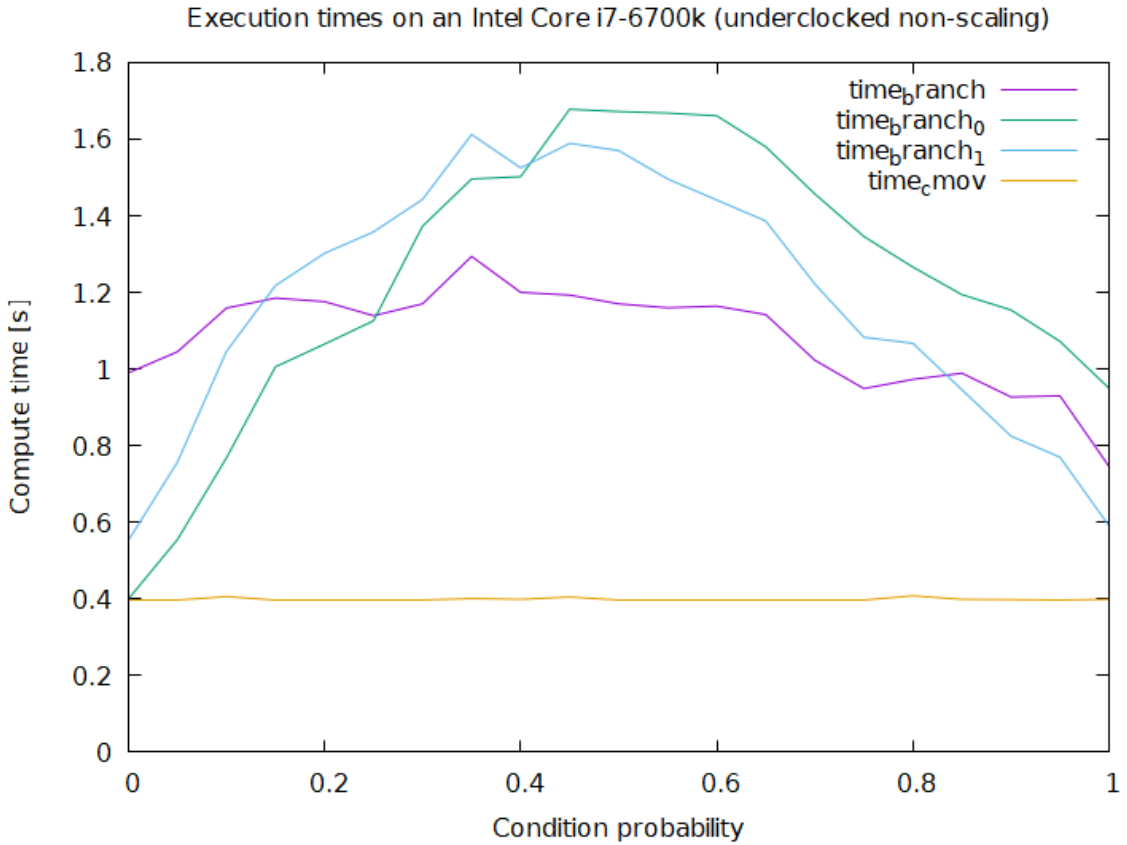
- Удаление проверок
- Сокращение числа проверок
- **Снижение стоимости проверок**

Workarounds: Арифметика вместо проверок

- Проверки можно заменить на инструкцию conditional move

```
let i = cmp::min(i, sums.len() - 1);
sums[i] = ...
```

- CMOV может быть как быстрее, так и медленнее чем бранч
- Недостаток: ошибка не будет обнаружена и программа продолжит работу



https://github.com/marcin-osowski/cmov

Workarounds: Арифметика вместо проверок

- Проверки можно заменить на маскирование если длина является 2^N (или может быть дополнена до 2^N)

```
const SIZE: usize = 100;
const LOOKUP_TABLE_SIZE: usize = SIZE.next_power_of_two();

let table: [u64; LOOKUP_TABLE_SIZE] = ...
... table[table & (LOOKUP_TABLE_SIZE - 1)] ...
```

- AND *как правило* может выполняться на большем числе ALU
- К сожалению на данный момент оптимизация применима только для массивов

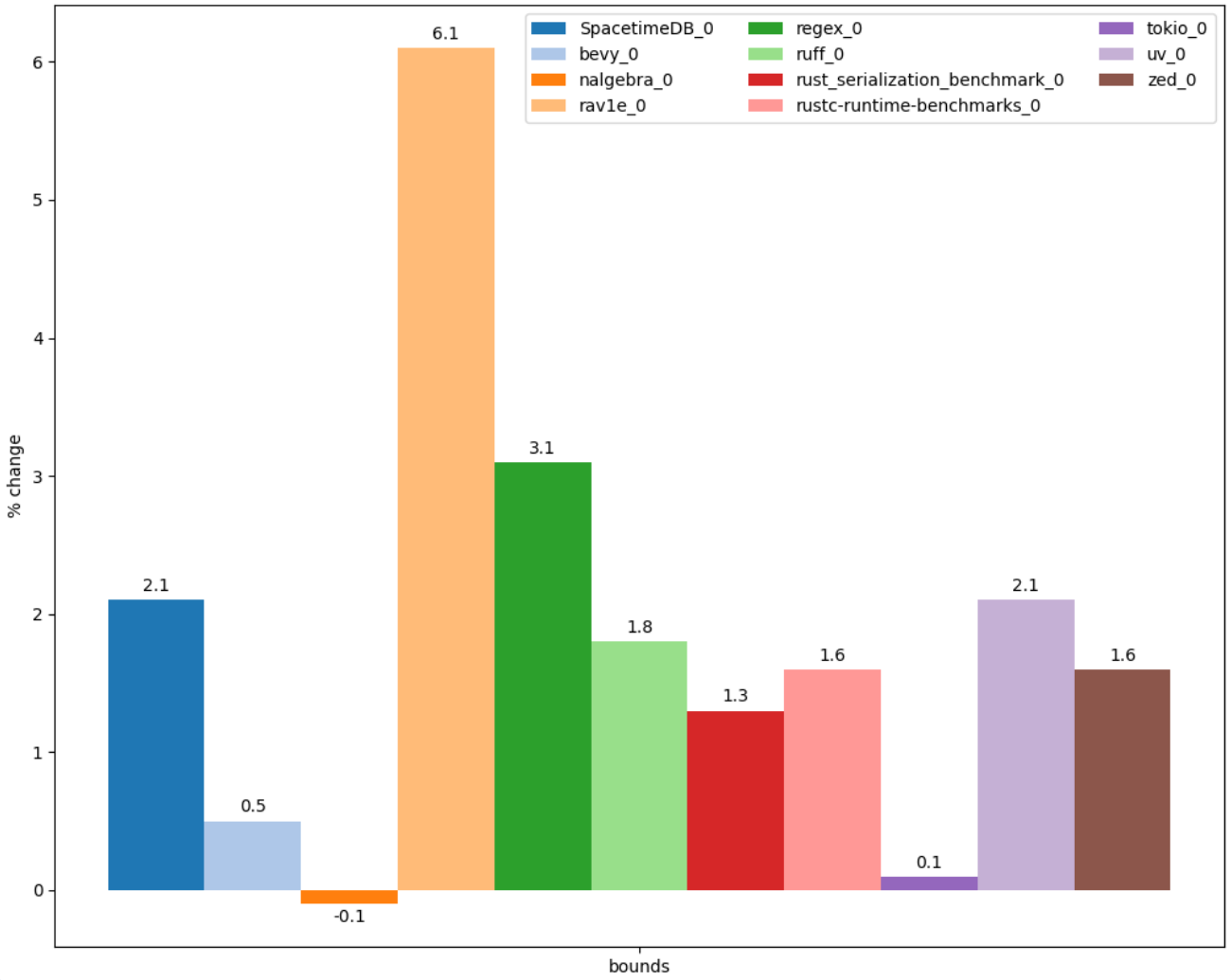
Распространённость

- 20% проверок в коде компилятора Rust – проверки индексов
  - Данные получены на основе анализа бинарного кода
- Среди циклов с проверками 80% содержат только индексные проверки
  - Эти проверки мешают их векторизации
  - Данные получены с помощью LLVM плагина
  - Проанализированы rustc и oxipng

Методика анализа

Ветка компилятора

Рост производительности при отключении проверок

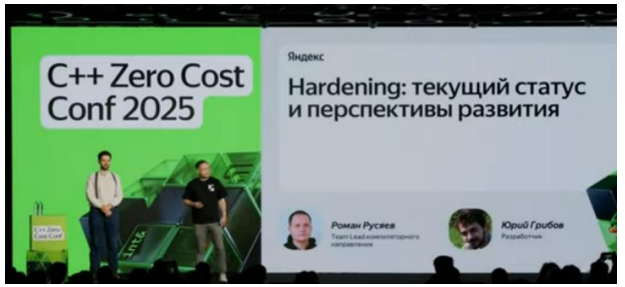


Отключалась инструментация в пользовательском коде и проверки в стандартной библиотеке (контейнеры, slices, strings, etc.)

[Ветка компилятора]

Hardened C/C++

- В C/C++ доступны ограниченные решения для обеспечения memory safety:
  - StackProtector
  - _FORTIFY_SOURCE
  - Hardened STL (C++ only)
  - -fsanitize=bounds, -fsanitize=object-size
- Полную проверку корректности индексов сделать невозможно из-за отсутствия информации о диапазонах для raw pointers
  - Например только 20% нетривиальных обращений в память в коде Clang могут быть проверены
  - C++ Safe Buffers может быть решением, но требует нетривиальной модификации кода
- Дополнительная информация:
  - Hardening: текущий статус и перспективы развития (C++ Zero Cost 2026)



Mario Simoes, https://snl.no/Mount_Everest

Rust vs. Hardened C++

```rust
pub fn foo(x: &[i32], i: usize) -> i32 {
    x[i]
}
```

rustc -O -Cpanic=abort

https://godbolt.org/z/jGT6adEYM

```
foo:
  cmp     rdx, rsi
  jae     .LBB0_2
  mov     eax, dword ptr [rdi + 4*rdx]
  ret
.LBB0_2:
  push    rax
  lea     rax, [rip + .Lanon.1]
  ...
  call    qword ptr [rip + panic_bounds_check]
```

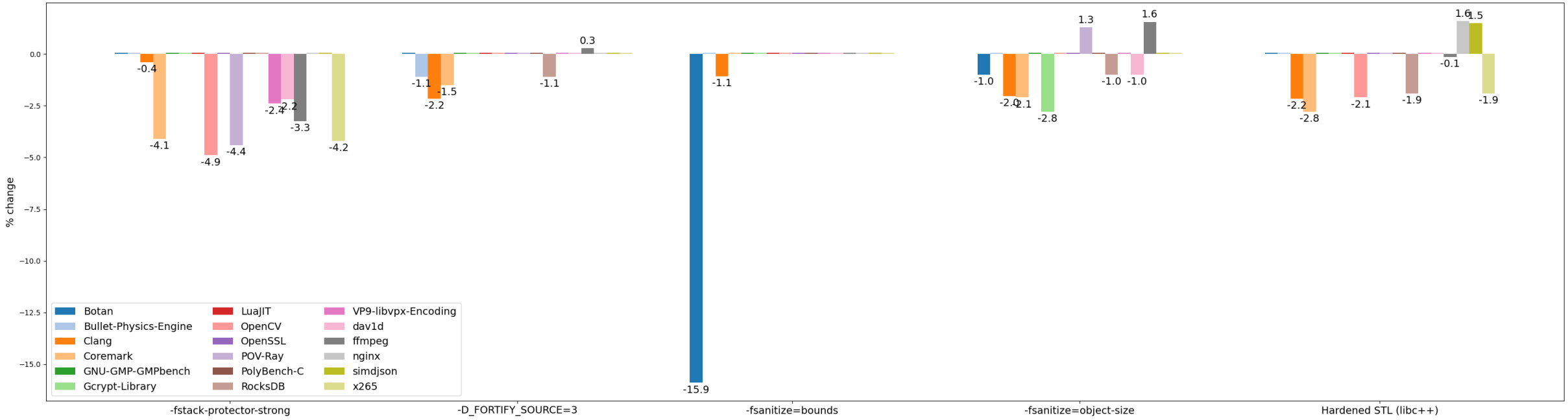
```cpp
extern "C"
int foo(std::span<int> s, size_t i) {
  return s[i];
}
```

gcc -O2 -D_GLIBCXX_ASSERTIONS

https://godbolt.org/z/6MG5vdefK

```
foo:
.LFB821:
  cmp     rdx, rsi
  jnb     .L3
  mov     eax, DWORD PTR [rdi+rdx*4]
  ret
.foo.cold:
.L3:
  push rax
  ...
  call    std::__glibcxx_assert_fail
```

Накладные расходы в Hardened C++



Неожиданные улучшения производительности связаны с аномалиями оптимизатора (неудачной векторизацией, отменой инлайнинга, etc.)

Выводы

- Проверки индексов в Rust применяются более агрессивно чем в Hardened C/C++
  - В Hardened C/C++ можно проверить в 5 раз меньше доступов чем в Rust
- Накладные расходы в среднем могут достигать 3-4% (и сравнимы с Hardened C/C++)
  - Но отдельные тесты могут замедляться на десятки процентов (как впрочем и в Hardened C/C++)
- Существует множество способов избежания проверок в критических частях кода
  - Некоторые из них будут не нужны если решат проблему с delayed panics (rust #50759)

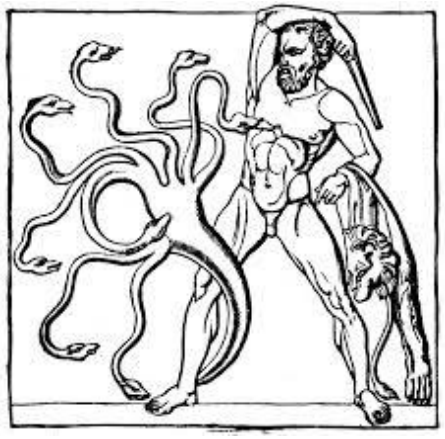
Рекомендуемые материалы

- How to avoid bounds checks in Rust without unsafe (Sergey Davidoff aka Shnatsel, Rust Secure WG lead)
  - “Библия” борьбы с индексными проверками
- Story-time: C++, bounds checking, performance, and compilers (Chandler Carruth, SW foundations & PLs technical lead, создатель языка Carbon)

Правила алиасинга

Детальный разбор

Алиасинг указателей



- Универсальная концепция в языках программирования
- Могут ли два разных указателя (ссылки) в программе адресовать одну и ту же память?
- В разных языках приняты различные подходы:
  - Никакие два указателя в программе не могут ссылаться на одни и те же данные (например Fortran 77)
  - Любые указатели могут ссылаться на одни и те же данные (например ранние версии языка C)
  - Промежуточные варианты
    - Например strict (type-based) aliasing в C89 и restrict-аннотации в C99

https://commons.wikimedia.org/wiki/File:Hercules_slaying_the_Hydra.gif

Почему важен алиасинг?

- Неограниченный алиасинг удобен для программиста, но снижает возможности компилятора
- Отсутствие алиасинга даёт больше возможностей многим оптимизациям
  - Векторизация, CSE, GVN, LICM, DSE, etc.
  - И даже используется в процессоре (для спекулятивного выноса загрузок)
- Поэтому важнейшим компонентом любого компилятора является модуль Alias Analysis
  - Идентификация указателей, которые могут (или не могут) алиаситься

```
int alias(int *a, int *b) {
  *a = 0;
  *b = 1;
  return *a;
}

int noalias(int * restrict a, int * restrict b) {
  *a = 0;
  *b = 1;
  return *a;
}
```

```
alias:
  mov     DWORD PTR [rdi], 0
  mov     DWORD PTR [rsi], 1
  mov     eax, DWORD PTR [rdi]
  ret

noalias:
  mov     DWORD PTR [rdi], 0
  xor     eax, eax
  mov     DWORD PTR [rsi], 1
  ret
```

https://godbolt.org/z/nqdax4vEP

Теория алиасинга

- Точный внутрипроцедурный alias analysis
  - NP-полон в отсутствие динамических аллокаций (Pointer-induced aliasing, 1991)
  - Неразрешим иначе (Undecidability of static analysis, 1992)
- В реальных программах алиасинг встречается очень редко
  - 98% указателей в бенчмарках SPEC ссылаются только на один объект
    - Dynamic Points-To Sets (Mock et al., 2001)
- Существующие Alias Analysis консервативно переоценивают алиасинг в 1.5-20 раз
  - Dynamic Points-To Sets (Mock et al., 2001)
  - Speculative Alias Analysis for Executable Code (Fernandez et al., 2002)
  - ORAQL — Optimistic Responses to Alias Queries in LLVM (Doerfert et al., 2023)

Алиасинг в Rust

- Ограничения Borrow checker на работу со ссылками упрощают alias analysis:
  - Mutable-ссылка не может алиаситься ни с какими другими (mutable или shared) ссылками
  - Mutable-ссылка, пока она существует, является единственным указателем на объект
  - Объект, адресуемый shared-ссылкой, не может быть изменён пока она существует
- На практике это позволяет добиться функциональности, сходной с restrict в C/C++

```
pub fn foo(a: &mut i32,
           b: &mut i32) -> i32 {
    *a = 0;
    *b = 1;
    *a
}
```

```
define i32 @foo(
  ptr noalias ... %a,
  ptr noalias ... %b) {
    store i32 0, ptr %a
    store i32 1, ptr %b
    ret i32 0
}
```

```
foo:
    mov     dword ptr [rdi], 0
    mov     dword ptr [rsi], 1
    xor     eax, eax
    ret
```

https://godbolt.org/z/6WoTzoo5v

Ограничения

- Информация об указателях предоставляется с помощью атрибута noalias у аргументов функций
  - Ссылки, возникающие *внутри* функций, никак не помечаются и с точки зрения LLVM могут алиаситься
- Решение возможно с помощью alias.scope metadata, но пока не реализовано (#16515)
- Workaround: создание dummy-функции для генерации noalias-аннотаций
  - Пример: rustc

```
pub fn foo(a: *mut i32,
           b: *mut i32) -> i32
{
    let a = unsafe { &mut *a };
    let b = unsafe { &mut *b };
    *a = 1;
    *b = 2;
    *a
}
```

```
foo:
    mov     dword ptr [rdi], 1
    mov     dword ptr [rsi], 2
    mov     eax, dword ptr [rdi]
    ret
```

https://godbolt.org/z/WjY9rxex7

Ограничения

- Частный случай указанной выше проблемы – алиасинг ссылок на элементы структур данных
- Например при передаче в функцию двух векторов компилятор не сможет определить что их буферы не алиасятся
- Workaround: передавать в функции raw slices

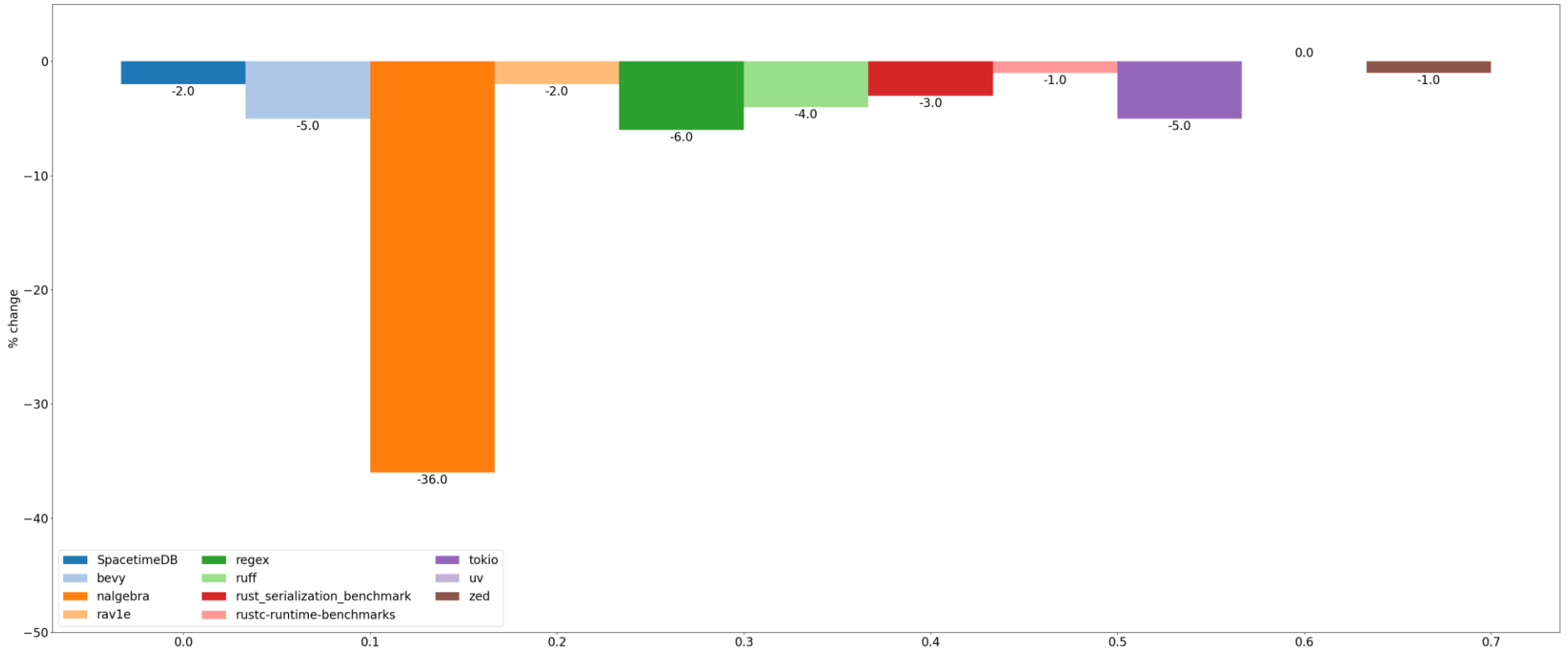
```
pub fn foo(a: &mut Vec<i32>,
           b: &Vec<i32>) {
    let n = a.len();
    let b = &b[..n];
    for i in 0..n {
        a[i] = b[i] + 100;
    }
}
```

```
pub fn foo(a: &mut [i32],
           b: &[i32]) {
    let n = a.len();
    let b = &b[..n];
    for i in 0..n {
        a[i] = b[i] + 100;
    }
}
```

Ограничения

- Правила алиасинга распространяются только на ссылки
- Указатели в Rust будут алиаситься
  - Причём сильнее чем в C/C++ из-за отсутствия strict aliasing

Эффективность правил алиасинга Rust

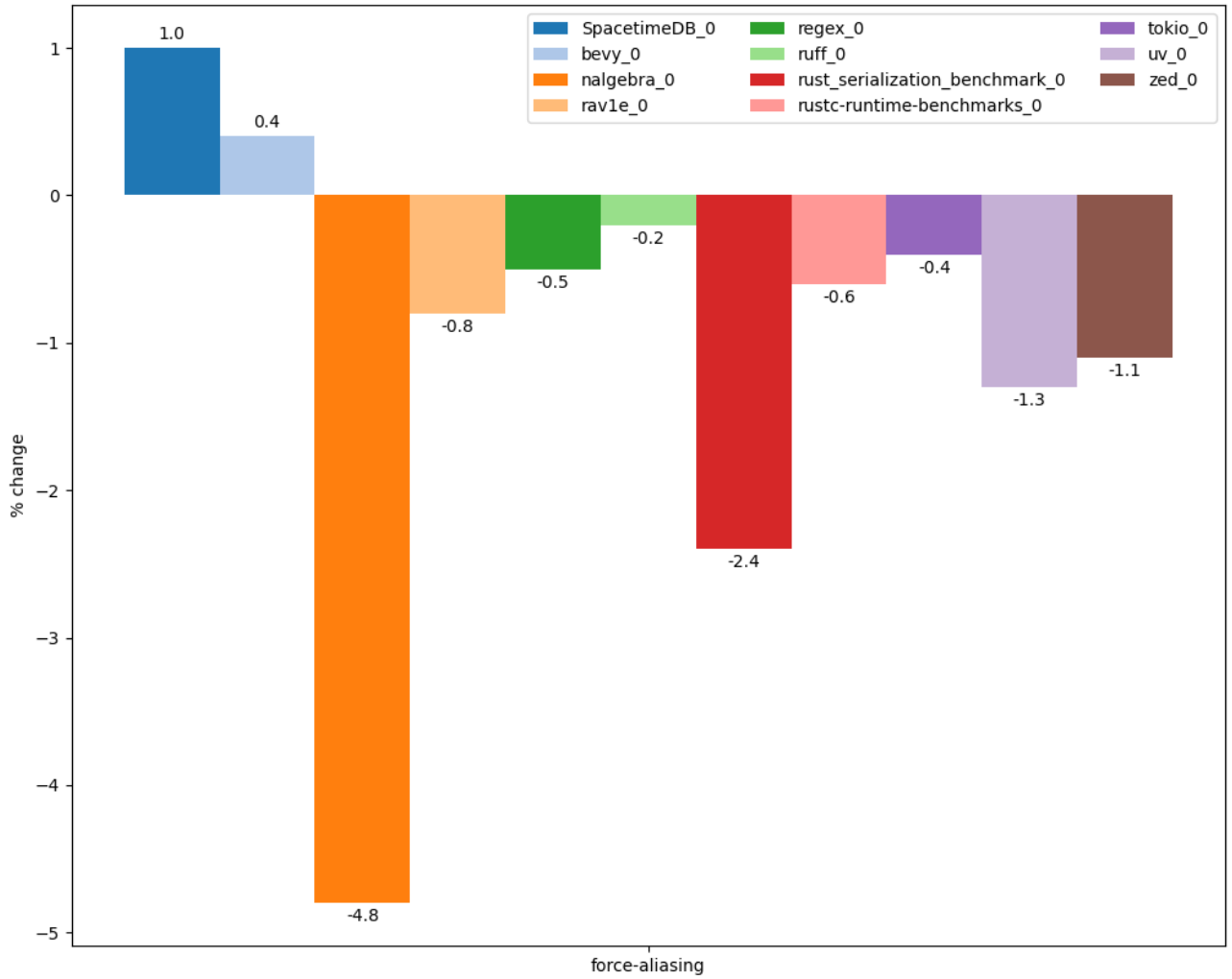


Точность алиас-анализа – число пар указателей, для которых анализ смог дать чёткий ответ (MustAlias / NoAlias)

Ухудшение точности Alias Analysis в Rust при отключении правил алиасинга

[Методика анализа] [Ветка компилятора]

Ухудшение производительности при отключении правил алиасинга



[Ветка компилятора]

Ситуация в C/C++

- C и C++ запрещают алиасинг указателей на разные типы
  - Кроме char *, signed/unsigned, scalar/vector и т.п.
- Но многие проекты игнорируют это правило с помощью -fno-strict-aliasing

- C и C++ предоставляют возможность явного запрета алиасинга программистом с помощью ключевого слова restrict
  - На практике эта функциональность используется редко (в Rust долго не могли включить правила aliasing из-за скрытых багов в LLVM)

Выводы

- Правила алиасинга в Rust позволяют
  - Повысить точность анализа в среднем на 5%
  - Улучшить производительность в среднем на 1-2%
- После дополнительных улучшений (aliasing metadata) можно ожидать прироста производительности

Рекомендуемые материалы

- О сложности Alias Analysis:
  - Pointer-induced aliasing (Landi, 1991)
  - Undecidability of static analysis (Landi, 1992)
- Об алиасинге в реальном коде:
  - Dynamic Points-To Sets (Mock et al., 2001)
  - Speculative Alias Analysis for Executable Code (Fernandez et al., 2002)
  - ORAQL — Optimistic Responses to Alias Queries in LLVM (Doerfert et al., 2023)
- Де-алиасинг в процессоре:
  - Memory Disambiguation on Skylake

Обязательная инициализация

Детальный разбор

Обязательная инициализация

- Неинициализированные переменные могут приводить к утечкам данных и другим ошибкам
  - Утечки паролей и адресов (утечка адреса компрометирует защиту ASLR)
- Распространённость:
  - 10% CVE root cause в продуктах Microsoft в 2018 (из Killing Uninitialized Memory)
  - 12% exploitable багов в Android (из P2723)

Alex Dos Santos, https://www.pexels.com/photo/vintage-water-tap-dripping-in-black-and-white-29725026/

Пример

```
void foo() {
  char password[32];
  ...
}
void bar() {
  char message[1024];
  if (cond) strcpy(message, “...”);
  // Сливаем пароль если !cond
  printf(message);
}
void baz() {
  foo();
  bar();
}
```

Подход Rust

- Поэтому Rust требует инициализации всех переменных
  - Локальные, глобальные, динамические
- Переменная должна присваивается на всех путях до первого использования

```
pub fn foo() {
  let x = 10;
  x
}
```

OK

```
pub fn foo(cond: bool) -> i32 {
  let x;  // No mut needed
  if cond {
    x = 10;
  } else {
    x = 20;
  }
  x
}
```

OK

```
pub fn foo(cond: u32) -> i32 {
  let x;  // No mut needed
  if cond > 10 {
    x = 10;
  } else if cond <= 10 {
    x = 20;
  }
  x
}
```

NOT OK

Другие языки

- Многие другие языки также требуют инициализацию от программиста (или предоставляют автоинициализацию нулями):
  - C#, Swift, Go, Java
  - Hardened C/C++ (-ftrivial-auto-var-init), C++26 (P2795)
    - Только локальные переменные
      - Глобальные переменные и sbrk/mmap-буферы уже инициализируются
      - Для кучи нужен hardened allocator
    - Включены по умолчанию в Android user/kernel-space и Chrome

Проблема с обязательными инициализациями

- Во многих случаях обязательные инициализации излишни
  - Буду перезаписаны другими значениями в дальнейшем
- Компилятор не всегда может удалить ненужные инициализации
- Пример:

```
let mut file = File::open("example.bin")?;

while (...) {
    let mut buffer = [0u8; 1024];  // memset not removed

    let n = file.read_exact(&mut buffer)?;

    ...
}
```

Распространённость лишних инициализаций

- Сколько ненужных инициализаций приходится делать?
- Статическая оценка (с помощью LLVM-плагина):
  - Hardened Clang: +4% stores в циклах (бенчмарк Clang)
- Динамическая оценка (с помощью Valgrind-плагина):
  - Компиляция большого C++ файла:
    - Clang: 11% неиспользуемых записей
    - Hardened Clang (-ftrivial-auto-var-init): 23% неиспользуемых записей (2x!)

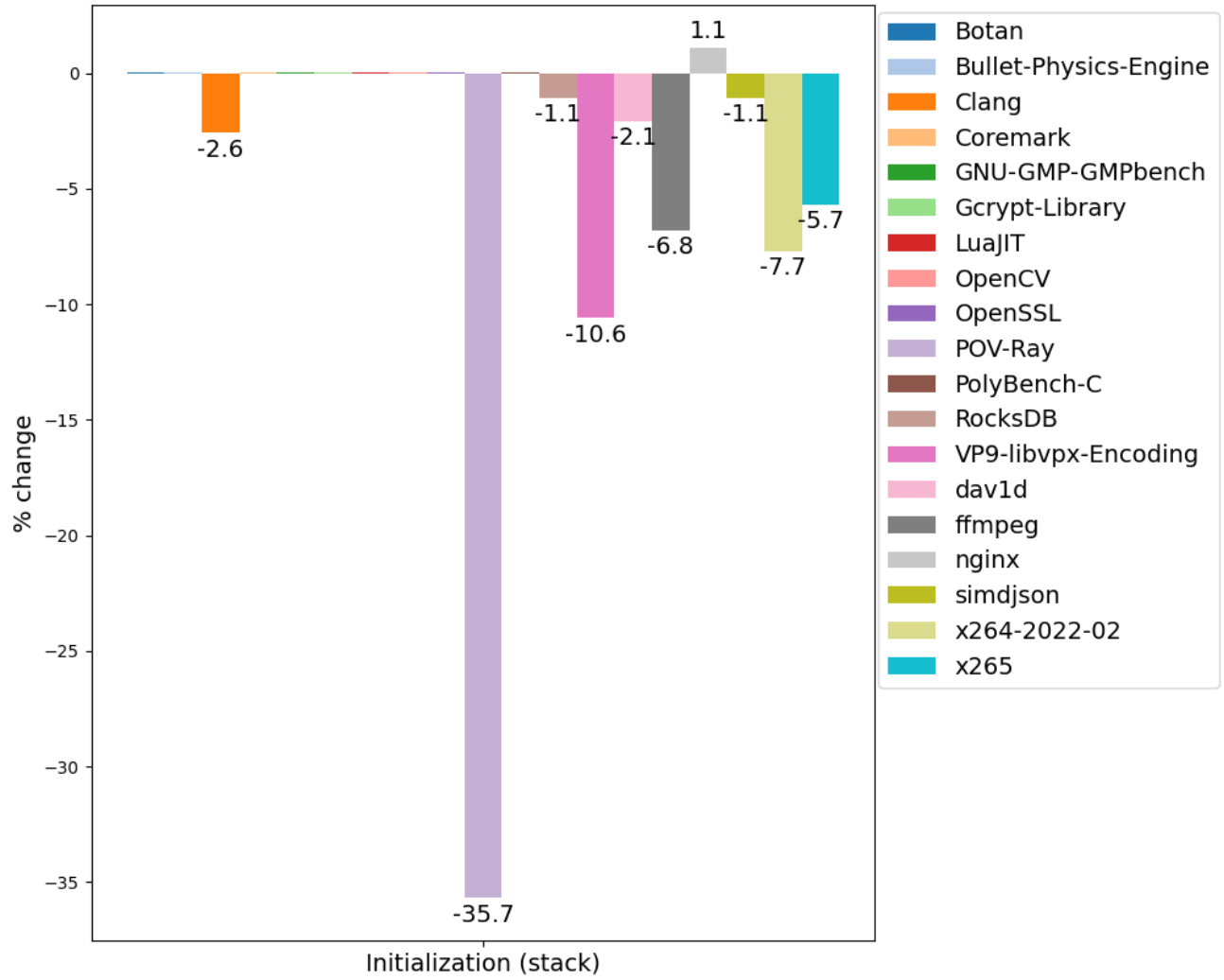
Накладные расходы

- К сожалению измерить накладные расходы от лишних инициализаций в наших Rust-бенчмарках невозможно
  - Так как потребовалась бы значительная модификация кода
- Проиллюстрируем накладные расходы на примере
  - Сообщений от пользователей
  - Замеров Hardened C/C++

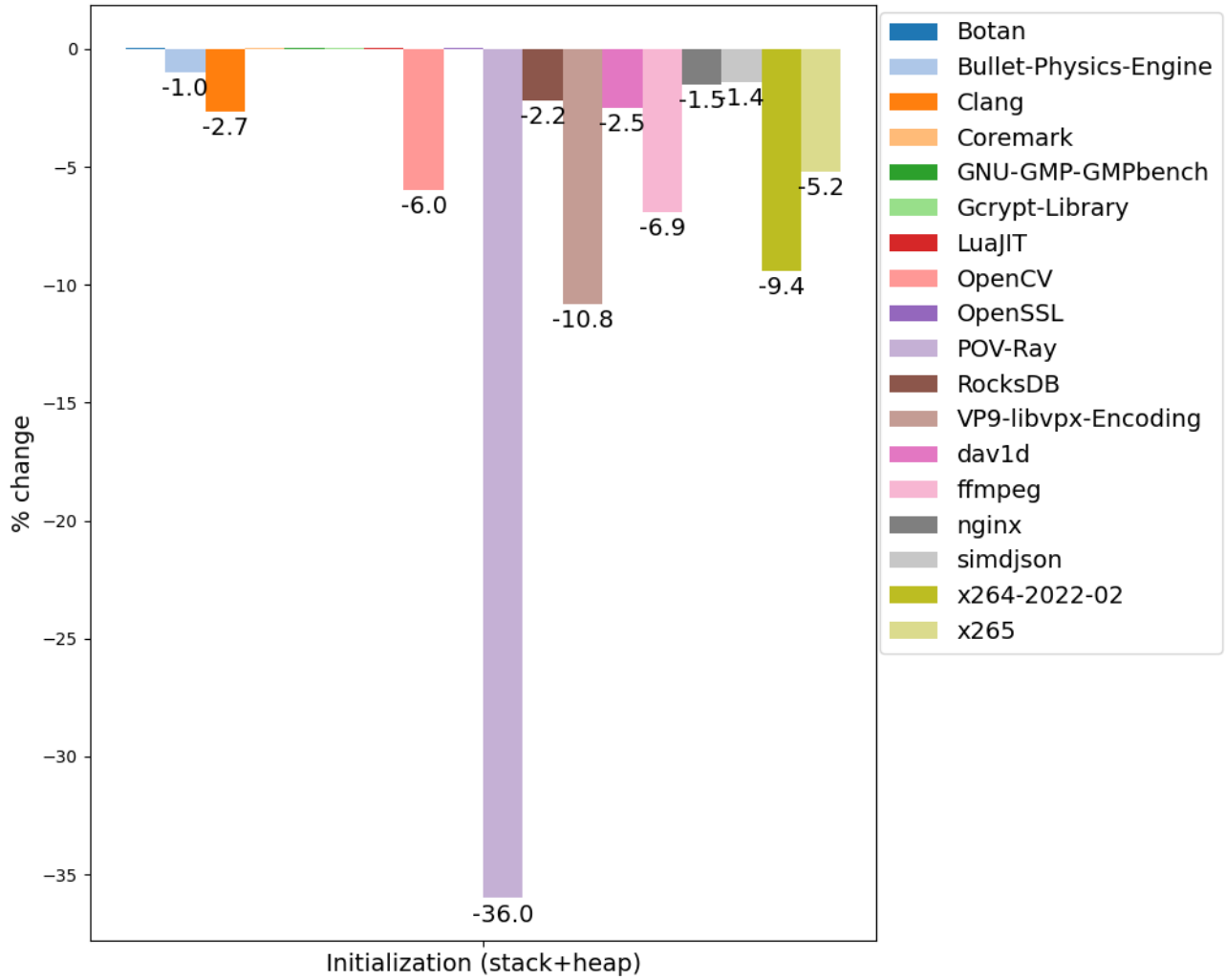
Накладные расходы в Rust

- Какие последствия могут иметь лишние инициализации для производительности?
  - 7% в stdlib file IO (rust #26950)
  - 20-30% в Hyper HTTP (комментарий в tokio #1744)
  - 30% в симуляторе Shadow (shadow #1643)
  - 25% в stdlib TcpStream (RFC #837)
  - 1.5% в rav1d (Making the rav1d Video Decoder 1% Faster)

Накладные расходы при включении стековых инициализаций (на примере C++)



Накладные расходы при включении инициализаций стека и кучи (на примере C++)



- Использовался грубый подход к инициализации кучи (LD_PRELOAD + memset)
- При более аккуратной реализации накладные расходы были бы ниже

Оптимизации в компиляторе

- Инициализации скалярных переменных очень дёшевы и оптимизируются многими SSA-проходами в LLVM
  - DCE, BDCE, ADCE, InstCombine, etc.
- Оптимизации сохранений в память представляют наибольшую проблему
  - DeadStoreElimination, MoveAutoInit, MemCpyOptimizer
  - В меньшей степени InstCombine

Workarounds

- Нельзя просто взять и создать ссылку на неинициализированный объект и потом заполнить его:

  “Creating a reference with &/&mut is **only allowed if** the pointer is properly aligned and **points to initialized data**” (addr_of_mut)

- Рекомендуемое решение: тип MaybeUninit и различные API для работы с ним
- К сожалению работа с MaybeUninit не всегда удобна

Workarounds: Стандартные типы

Массивы:

```
let buf :: [u8; 4] = unsafe {
    let mut buf = MaybeUninit::<[u8;
4]>::uninit();
    let ptr = buf.as_mut_ptr();
    for i in ... {
        ptr.offset(i).write(...);
    }
    buf.assume_init()
};
```

Структуры:

```
let role :: Role = unsafe {
    let mut uninit =
MaybeUninit::<Role>::uninit();
    let ptr = uninit.as_mut_ptr();
    // Нужна именно запись
    // (а не присваивание)
    // чтобы не пытаться удалить
    // неинициализированный String
    addr_of_mut!((*ptr).name)
        .write("basic".to_string());
    (*ptr).flag = 1;
    (*ptr).disabled = false;
    uninit.assume_init()
};
```

Workarounds: Векторы

```
let mut buf = Vec<u8>::new();

let old_len = buf.len();
buf.reserve(data.len());

unsafe {
    let ptr = buf.spare_capacity_mut();
    for i in ... {
        ptr[idx].write(...);
    }
    buf.set_len(len + data.len());
}
```

Workarounds: IO

```
let mut buf = [MaybeUninit::uninit(); 4096];
let mut buf = BorrowedBuf::from(&mut buf[..]);

rd.read_buf(buf.unfilled())?;

let data : &[u8] = buf.filled();

// Дальше можно использовать buf.filled()
```

Выводы

- Расходы на инициализации в Rust и Hardened C++ сопоставимы
  - Выше чем в C++26 из-за инициализаций кучи
- Могут быть достаточно большими в некоторых сценариях (5-10% и более) и потребовать ручного тюнинга кода:
  - Использование специальных библиотечных API типа MaybeUninit

Рекомендуемые материалы

- Killing Uninitialized Memory (video)
  - Доклад от Microsoft о внедрении аналогичных проверок в Visual Studio
- P2795: Erroneous behavior for uninitialized reads
  - To be integrated to C++26

Локализация символов

Детальный разбор

Локализация символов

- Функции, объявленные в единице трансляции (крейт в Rust, cpp-файл в C++), могут быть локальными для неё или глобальными
  - В C/C++ контролируется с помощью static или anonymous namespaces
  - В Rust с помощью атрибутов pub, pub(crate) и других
- В Rust и C++ выбраны противоположные решения о дефолтном поведении:
  - В Rust функции по умолчанию локальны для своего модуля (top-level функции – для своего крейта)
  - pub(crate)-функции видимы в пределах своего крейта
  - pub-функции видны в пределах всей программы

Локализация в Rust

```rust
pub(crate) fn foo1(x: i32) {
    println!("Hello world {}", x);
}

mod mymod {
    pub(super) fn foo2(x: i32) {
        println!("Goodbye world {}", x);
    }

    pub fn foo3(x: i32) {
        println!("Nice world {}", x);
    }
}

pub use mymod::foo3;

pub fn bar() {
    foo1(1);
    mymod::foo2(2);
    mymod::foo3(3);
}
```

```
$ rustc +baseline test.rs -O --emit=asm --
crate-type=rlib -o- \
    | rustfilt \
    | grep globl
        .globl  test::mymod::foo3
        .globl  test::bar
```

Экспортируются
только foo3 и bar!

Оптимизации в компиляторе

- Локальные функции лучше оптимизируются компилятором
- Как правило такие оптимизации связаны с изменением API или ABI:
  - Межпроцедурное распространение констант (IPSCCP)
  - Замена передачи по ссылке на передачу по значению (ArgumentPromotion)
  - Удаление мёртвых аргументов или возвращаемых значений (DeadArgumentElimination)
  - Релаксация calling convention (замена на coldcc в Transforms/IPO/GlobalOpt.cpp)
  - Межпроцедурная аллокация регистров (IPRA)
- А также
  - Более агрессивный inlining (например всегда инлайнятся однократно вызываемые локальные функции)
  - Улучшенное обнаружение мёртвого кода

Оптимизации в линкере

- В случае динамических библиотек локальные функции
  - Могут вызываться напрямую (минуя PLT/GOT)
  - Не должны разрешаться динамическим загрузчиком
- Детали в докладе Динамические библиотеки и способы их ускорения

Пример оптимизации

```rust
#[inline(never)]  // Simulate inliner fail
fn foo(b: bool, seed: i32, s: &[i32]) -> i32 {
    let mut ans = seed;
    for &x in s {
        ans += if b { 1 } else { ans ^ x }
    }
    ans
}

pub fn bar(n: i32, s: &[i32]) -> i32 {
    let mut ans = 0;
    for i in 0..n {
        ans += foo(true, i, s);
    }
    ans
}
```

https://godbolt.org/z/MM7Tchx49

```
foo:
        lea     eax, [rsi + rdi]
        ret
```

Цикл в foo был полностью оптимизирован

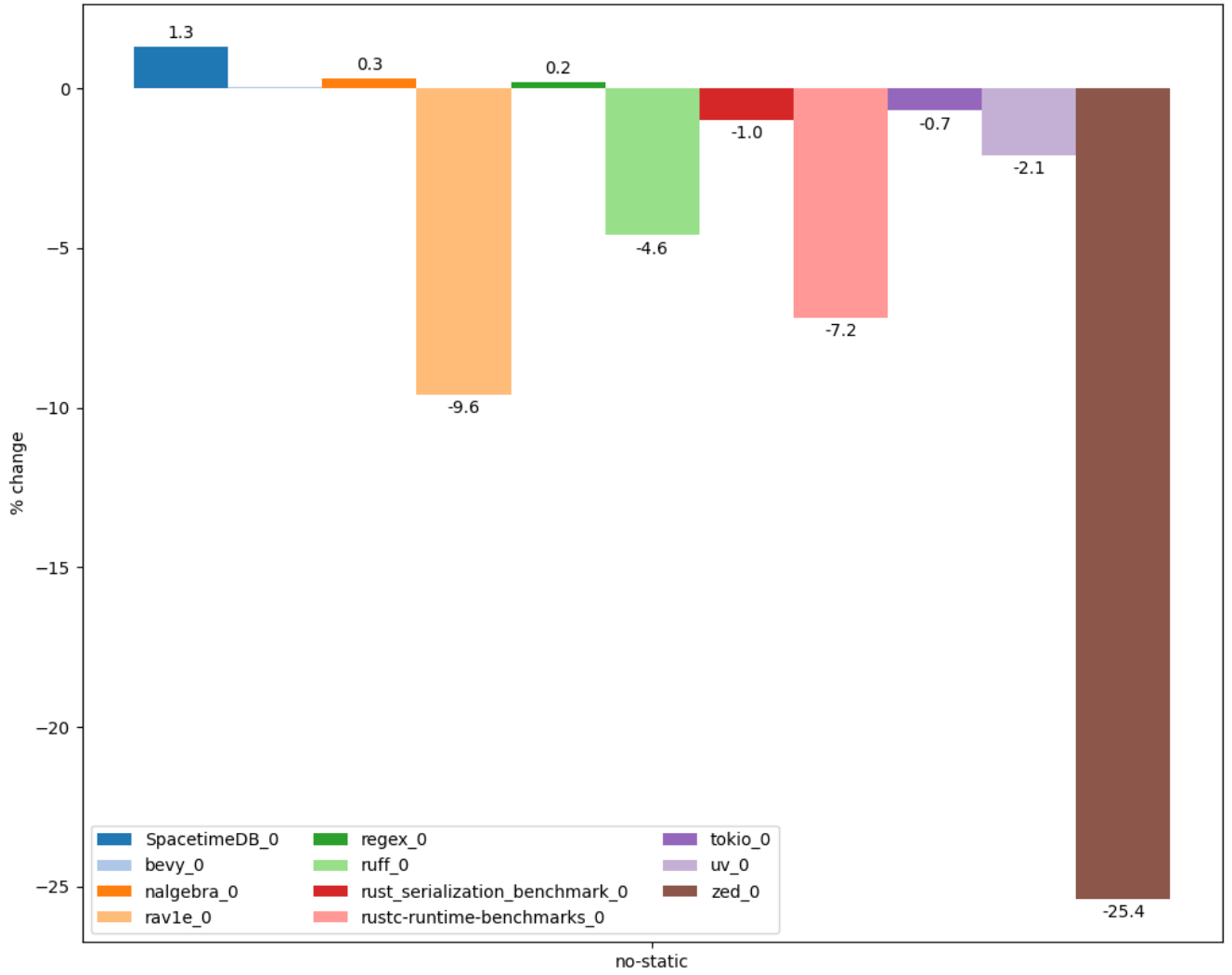
Сравнение с C/C++

- Многие авторитетные руководства рекомендуют локализовывать функции:
  - Google C++ Style Guide
  - LLVM Coding Standard (Restrict Visibility)
  - C++ Core Guidelines
  - Chromium C++ Style Guide
- Слабым местом C++ является невозможность локализовать private-методы классов
  - Попытка такой оптимизации

Распространённость

- Сколько локальных функций в реальных программах на Rust?
  - 80-90%
  - (Для сбора статистики использовался патч)
- В C/C++ результаты сильно зависят от проекта:
  - 80-90% (clang, ffmpeg, git, opencv, bitcoin)
  - 40-50% (openssl, tmux, gcc, libuv)
  - (Для сбора статистики использовалась утилита Localizer)

Замедление при отключении локализации



[Патч для компилятора]

Выводы

- Локализация функций позволяет компилятору генерировать существенно более производительный код (5% и более)
- В Rust по умолчанию принято более эффективное (с точки зрения производительности) решение

Рекомендуемые материалы

- Efficient C Tips #5 – Make ‘local’ functions ‘static’

Отсутствие inplace-инициализации

Детальный разбор

Введение

- В C++ существуют механизм placement new для конструирования объектов в произвольном буфере в памяти:

```
void *addr = some_alloc(sizeof(A));
A *ptr = new(addr) A(...);  // Или std::construct_at
```

- А также perfect forwarding:

```
std::vector<LargeStruct> vec;
vec.emplace_back(large_struct_initializer);
```

- В языке Rust аналогичный механизм отсутствует
  - Следствие обязательной инициализации объектов
- Проблема:

```
// Сможет ли компилятор убрать копирования (со стека в кучу)?
let b = Box::new(LargeStruct { ... });
```

Зачем нужны placement new?

```rust
const SIZE: usize = 1024 * 1024 * 1024;

fn main() {
    let b: Box<[usize]> = if cfg!(fast) {
        vec![15; SIZE].into_boxed_slice()
    } else {
        // Проблемное копирование
        Box::new([15; SIZE])
    };
    black_box(&b);
}
```

```
# Сможет ли компилятор оптимизировать копию?
$ LIMIT=$((8 * 1024))
$ rustc +baseline -O test.rs
$ (ulimit -s $ LIMIT; ./test)
thread 'main' has overflowed its stack
fatal runtime error: stack overflow

# Работает с некопирующим dedicated API
$ rustc +baseline --cfg fast -O test.rs
$ (ulimit -s $ LIMIT; ./test)
```

Зачем нужны placement new?

```
struct A {
  std::array<int, 1024> vals = {};

  A() = default;
  A(const A &) = default;

  A(int val) {
    std::fill(vals.begin(), vals.end(), val);
  }
};

int main() {
  std::vector<A> aa;
  aa.reserve(NITER);

  for (size_t i = 0; i < NITER; ++i) {
#ifdef USE_PLACEMENT_NEW
    aa.emplace_back(i);
#else
    A a(i);
    aa.push_back(a);
#endif

    asm("" :: "r"(&aa) : "memory");
  }

  return 0;
}
```

USE_PLACEMENT_NEW
быстрее на ~10%!

Оптимизации в компиляторе

- Move/copy elision
  - Обнаружение и удаление ненужных перемещений или копирований
- Может работать более агрессивно чем в C++
  - Наличие high-level IR (MIR)
  - Shallow move/copy (по сути move/copy в Rust это просто вызов memcpy)
- Компиляторы C++ не поддерживают полноценный copy elision
  - Работают ограниченные оптимизации типа RVO, NRVO, etc.
  - Прежде всего из-за отсутствия high-level IR (ClangIR to the rescue?)
  - См. работу Романа Русяева по Ultimate Copy Elision (GitHub со ссылкой на реализацию)
- Реализация в компиляторе:
  - MIR (high-level IR): CopyProp, DeadStoreElimination, DestinationPropagation, RenameReturnPlace
  - LLVM: MemCpyOptPass

Детальный разбор

Проблема с move/copy elision

- Зависит от версии компилятора
  - Не всегда монотонно!
- Работает только в release-режиме
  - В debug-сборке копии остаются и могут вызвать переполнение стека или по крайней мере сильное замедление
- Языковая поддержка inplace-инициализации является одной из целей на 2026 год:
  - In-place initialization

Workarounds

- Те же что для обязательных инициализаций:
  - Использование специальных API для создания неинициализированных буферов и их явная инициализация
  - Например Box/Rc::new_uninit, Vec::with_capacity/set_len
- А также другие API например Vec::into_boxed_slice

Выводы

- Copy elision в Rust работает эффективнее чем в C++
- Но не может компенсировать отсутствие явной inplace-инициализации
  - Placement new и perfect forwarding
- Разработчики языка осознают эту проблему и активно работают над её решением:
  - Rust Project Goals 2026: In-place initialization
  - Rust Project Goals 2026: MIR move elimination

Рекомендуемые материалы

- Ultimate Copy Elision
  - Исчерпывающий обзор copy elision в C++ от Романа Русяева

Оптимизации расположения полей

Детальный разбор
(пока не готов)

Выравнивание полей структур

- Многие языки (C, C++, C#, Swift) сохраняют порядок полей структуры, указанный в исходном коде
- Для выравнивания адресов полей между ними может добавляться паддинг
  - Выравнивание необходимо из-за аппаратных ограничений
  - Невыровненные обращения могут вызывать исключения или замедление работы
- Выравнивание полей ускоряет доступ к ним, но ухудшает локальность данных в D$

```
struct RandomOrder {
  uint8_t  m_byte1;
  uint64_t m_long;
  uint8_t  m_byte2;
  uint32_t m_int;
  uint8_t  m_byte3;
};
```

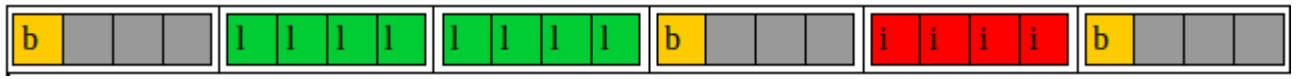


Image source

Структуры в Rust

- Rust ABI не фиксирует порядок полей в структурах
- Это позволяет компилятору свободно его модифицировать
- Оптимизация порядка полей для
  - Снижения D$ pressure (посредством уменьшения требуемого паддинга)
  - Оптимизации расположения ниш (см. ниже)
- В теории компилятор также может размещать вместе поля, которые часто используются совместно
  - Можно воспользоваться PGO или статическим анализом (как -fipa-struct-reorg в GCC)
- А также выносить редко используемые поля (structure peeling)

Оптимизация порядка полей

- Для уменьшения размера поля сортируются в порядке уменьшения выравнивания

```rust
pub fn fun(val: Foo) -> i64
{
    val.a
        + (val.b as i64)
        + (val.c as i64)
}
```

```rust
pub struct Foo {
    pub b: bool,
    pub a: i64,
    pub c: i16,
} // 16 bytes total
```

```
fun:
  movzx   ecx, byte ptr [rdi + 10]
  add     rcx, qword ptr [rdi]
  movsx   rax, word ptr [rdi + 8]
  add     rax, rcx
  ret
```

https://godbolt.org/z/MxYh7bv34

```rust
#[repr(C)]
pub struct Foo {
    pub b: bool,
    pub a: i64,
    pub c: i16,
} // 24 bytes total
```

```
fun:
  movzx   ecx, byte ptr [rdi]
  add     rcx, qword ptr [rdi + 8]
  movsx   rax, word ptr [rdi + 16]
  add     rax, rcx
  ret
```

https://godbolt.org/z/GMoejn7rd

Jeffrey Bary, https://www.flickr.com/photos/70118259@N00/3173642531

Что есть в C++

- Многие статические анализаторы предупреждают о неэффективном расположении полей
  - -Wpadded, clang-tidy, PVS-studio, etc.
  - Pahole (на базе debuginfo)
- clang-reorder-fields – LLVM-based инструмент для перестановки полей для оптимизации паддинга
- В теории при использовании Fat LTO компилятор может делать аналогичные оптимизации автоматически
  - Так сейчас уже делает LCC (МЦСТ)

Устройство Rust enums

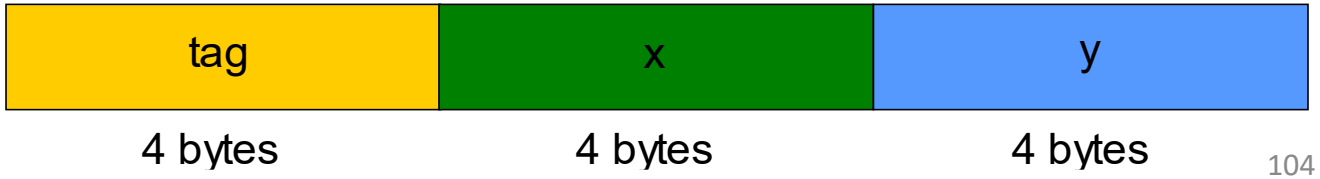
- Enums в Rust являются аналогом std::variant в C++
- Могут содержать различные типы данных
  - Конкретный тип определяется в рантайме и хранится в специальном поле – теге
- Объект типа enum содержит
  - Тег (tag, discriminant) с индексом хранимого варианта
  - Данные этого варианта

```rust
pub enum Foo {
    A {x: i32, y: i32},
    B {x: i32, y: i32},
}

pub fn fun(val: Foo) -> i32 {
    match val {
        Foo::A {x, ..} => x,
        Foo::B {y, ..} => y,
    }
}
```

https://godbolt.org/z/n8a1x9nWr

```
fun:
  mov     eax, dword ptr [rdi]
  mov     eax, dword ptr [rdi + 4*rax + 4]
  ret
```



Ниши

- Некоторые комбинации битов могут не соответствовать никакому осмысленному значению кодируемого типа
- Например
  - &x или NonZero<i32> не могут быть нулями
  - bool (8 бит) это только 0b0 или 0b1
    - Остальные 254 значения некорректны
  - char (32 бита) может иметь значения только в диапазонах [0x0, 0xD7FF] и [0xE000, 0x10FFFF]
- Такие диапазоны невалидных значений (не битов!) называются нишами (niche)
- Паддинги в структурах **не могут** использоваться для ниш
- Могут использоваться для хранения тега enum-типа (вместо отдельного поля)
  - Т.н. Discriminant Elision

https://nowmediagroup.tv/blog/10-must-read-books-why-multidisciplinary-reading-is-a-must/

Примеры ниш

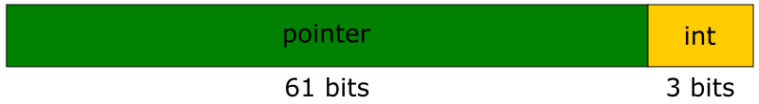
- Самый яркий (и гарантированный языком) пример оптимизации ниш – тип Option<T>, где T – NonNull тип (например &U)

$$\texttt{Option<\&U>} = \begin{cases} \text{None} & = 0x0 \\ \text{Some(\&u)} & = \text{addr\_of!(u)} \end{cases}$$

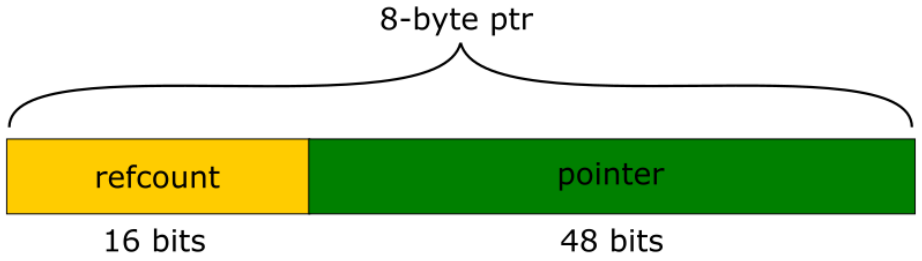
- Одна ниша может использоваться для нескольких тегов:
  - `size_of::<Option<Option<Option<bool>>>>() == 1`
- Аналогичная оптимизация возможна для младших бит ссылки (или указателя)
  - У выровненных указателей несколько младших бит всегда 0
  - Их можно использовать как нишу
  - Пока реализовано только в nightly (-Zreference-niches, rust #3204)

Что есть в C++

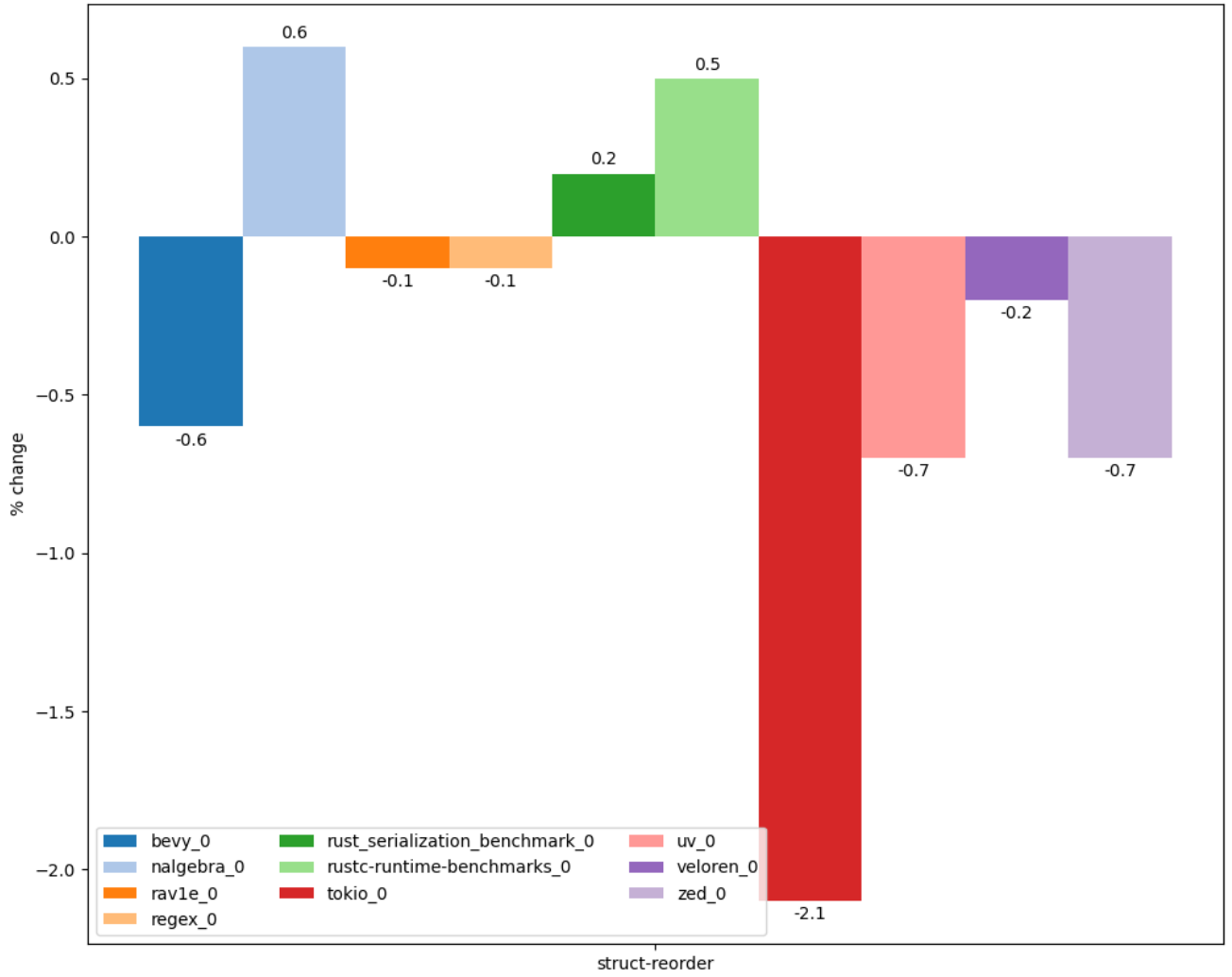
- В std::optional и std::variant нет аналога оптимизаций ниш
  - Можно реализовать аналог std::optional с такой оптимизацией
- llvm::PointerIntPair
  - Использует младшие биты указателя для хранения значений int
  - Аналог -Zreference-niches в Rust



- Lock-free реализации atomic shared_ptr (e.g. в Folly) используют свободные биты адреса для хранения reference count
  - Это позволяет использовать атомарные инструкции меньшей ширины



Замедление при выключении переупорядочивания



- Отключили shouldMergeGEPs и store merging из-за их чувствительности к порядку полей

[Ветка компилятора]

Выводы

- Rust не гарантирует порядок полей внутри структур
  - Это даёт свободу оптимизаций
  - Но требует осторожности в unsafe коде
- Это позволяет компилятору агрессивно переупорядочивать поля и переиспользовать их для тегов
  - Аналогичные оптимизации можно вручную реализовать и в C++
- Влияние оптимизации самой по себе невелико, но она может включать или отключать другие оптимизации
  - Как в компиляторе и процессоре
  - Как положительно, так и отрицательно может влиять на производительность

Рекомендуемые материалы

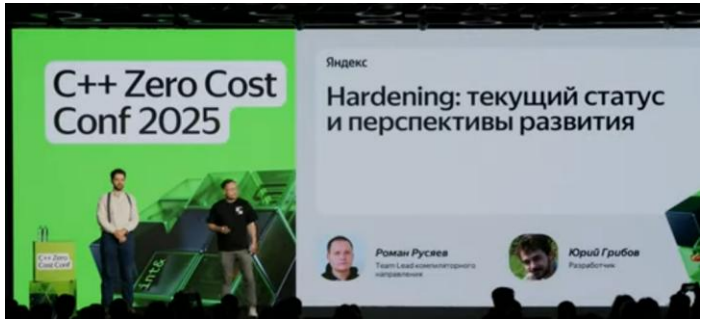
- The Lost Art of Structure Packing (by ESR)

Арифметические проверки

Детальный разбор

Введение

- Переполнение арифметических операций может приводить к серьёзным системным сбоям
  - ~1% CVE и 1.5% KEV в 2024
  - 23 место в рейтинге Mitre CWE Top 25 2024 (8 в рейтинге 2019 года)
- Наиболее известные примеры:
  - Инцидент с облучателем Therac-25 (1985)
  - Катастрофа ракеты Ariane 5 (1996)
- История (pre-Rust):
  - `-ftrapv` появилась в GCC в 2000 (patch for -ftrapv option)
    - За фичей не следили и она быстро протухла (например BZ #35412 открыт в 2008)
  - Работы John Regehr над чекером IOC в 2010
  - Создание UBsan в 2014 (на волне популярности Asan)
    - State-of-the-art решение
- Дополнительная информация:
  - Hardening: текущий статус и перспективы развития (C++ Zero Cost)



J.R. Armstrong, Famous Failures Revisited

Пример ошибки (OpenSSH 3.3)

```
nresp = packet_get_int();
if (nresp > 0) {
  // Переполняем целое число до нуля здесь ...
  response = xmalloc(nresp*sizeof(char*));
  // ... и вызываем heap buffer overflow тут
  for (i = 0; i < nresp; i++)
    response[i] = packet_get_string(NULL);
}
```

Дефолтные арифметические проверки Rust

- Некоторые (не самые опасные...) арифметические проверки включены по умолчанию
  - Деление на ноль
  - Знаковое переполнение при делении (INT_MIN / -1)
  - Ограничение диапазона при преобразовании float в integer
    - Можно отключить с помощью -Zsaturating-float-casts=off
- Целочисленные переполнения в контейнерах стандартной библиотеки
  - С помощью assert!, checked_add, etc.

Важные проверки, отсутствующие в Rust

- По историческим причинам некоторые важные проверки отсутствуют
  - И даже не могут быть включены по опции
- Например переполнение при преобразовании типа с помощью as
  - Нужно пользоваться альтернативным API try_into/ try_from

```
fn main() {
    0x100i32 as i8;
}
```

```
$ ./test
0
```

```
use std::convert::TryFrom;

fn main() {
    i8::try_from(0x100i32).unwrap();
}
```

```
$ ./test

thread 'main' panicked at test.rs:5:32:
called `Result::unwrap()` on an `Err` value:
TryFromIntError(())
```

Проверки переполнений в Rust

- По умолчанию целочисленные переполнения в пользовательском коде проверяются *только* в debug-сборках
- Two’s complement в release
- Можно включить проверки по опции -C overflow-checks=on
- Причины: слишком большие накладные расходы
  - “Plan is to turn on checking by default *if* we can get the performance hit low enough and also that it should leave room in the future to move towards universal overflow checking if it becomes feasible” (Niko Matsakis, ко-lead Rust language design team)

Накладные расходы

- Накладные расходы от проверок:
  - Сравнение + предсказуемый условный переход (1-2 ALU слота + I$ pressure)
  - Мешают другим оптимизациям (vectorizer, etc.)
- С первым (менее важным) пунктом *возможно* могла бы помочь аппаратная поддержка проверок в процессоре

```rust
pub fn foo(xs: &[i32]) -> i32 {
    let mut ans = 0;
    for x in xs {
        ans += x;
    }
    ans
}
```

```
$ rustc ... test.rs
...
.LBB0_5:
        // Цикл успешно векторизуется
        movdqu  xmm2, xmmword ptr [rdi + 4*rax]
        paddd   xmm0, xmm2
        ...
$ rustc ... -Coverflow-checks=on test.rs
...
.LBB0_3:
        // Цикл не векторизуется
        add     eax, dword ptr [rdi + rcx]
        jo      .LBB0_6
        add     rcx, 4
        cmp     rsi, rcx
        jne     .LBB0_3
        ...
```

https://godbolt.org/z/1x49jffv3

https://godbolt.org/z/43h1c87M5

Defined overflow

- В отличие от C/C++ целочисленное переполнение не является Undefined Behavior
  - Паника или Two’s complement
- Это лишает оптимизатор дополнительной информации
- Может помешать точности анализов (особенно ScalarEvolution) и оптимизаций в LLVM
  - См. также How undefined signed overflow enables optimizations in GCC

Defined overflow: Пример

Из доклада Garbage In, Garbage Out (Chandler Carruth)

```
for (;;) {
    int c1 = buf[i1];
    int c2 = buf[i2];
    if (c1 != c2)
        return c1 - c2;
    i1++;
    i2++;
}
```

```
.LBB0_1:
  movzx   eax, byte ptr [rdx + rsi]
  movzx   edi, byte ptr [rdx + rcx]
  inc     rdx
  cmp     al, dil
  je      .LBB0_1
```

https://godbolt.org/z/Yssq68Gja

```
loop {
    let c1 = unsafe {
      *buf.get_unchecked(i1 as usize)
    } as i32;
    let c2 = unsafe {
      *buf.get_unchecked(i2 as usize)
    } as i32;
    if c1 != c2 {
        return c1 - c2;
    }
    i1 += 1;
    i2 += 1;
}
```

Компилятор вынужден допустить переполнение

```
.LBB0_2:
  movsxd  rdi, edi
  movzx   eax, byte ptr [rdx + rdi]
  movsxd  rsi, esi
  movzx   ecx, byte ptr [rdx + rsi]
  inc     edi
  inc     esi
  cmp     al, cl
  je      .LBB0_2
```

https://godbolt.org/z/c1sMcxfsP

Проблемы с inclusive ranges

- Циклы с включенной верхней границей оптимизируются хуже чем обычные:

```
for i in 1..=n {
    total += i;
}
```

- Из-за возможного переполнения (при `n==T::MAX`) компилятор выполняет их с дополнительной проверкой:

```
while i <= n {
    total += i;
    if i >= n {
        break;
    }
    i += 1;
}
```

- Можно помочь компилятору с помощью явного вычисления границы:

```
for i in 1..(n+1)
```

- Или воспользовавшись internal iteration (.for_each, .fold, etc.)

Оптимизации в компиляторе

- Проверки арифметических переполнений намного хуже оптимизируются компилятором
- При сборке компилятора Rust (с -Coverflow-checks=on) удаляется ~30% арифметических проверок
  - Данные получены на основе анализа бинарного кода
- Это основная причина их отключения в release-сборке

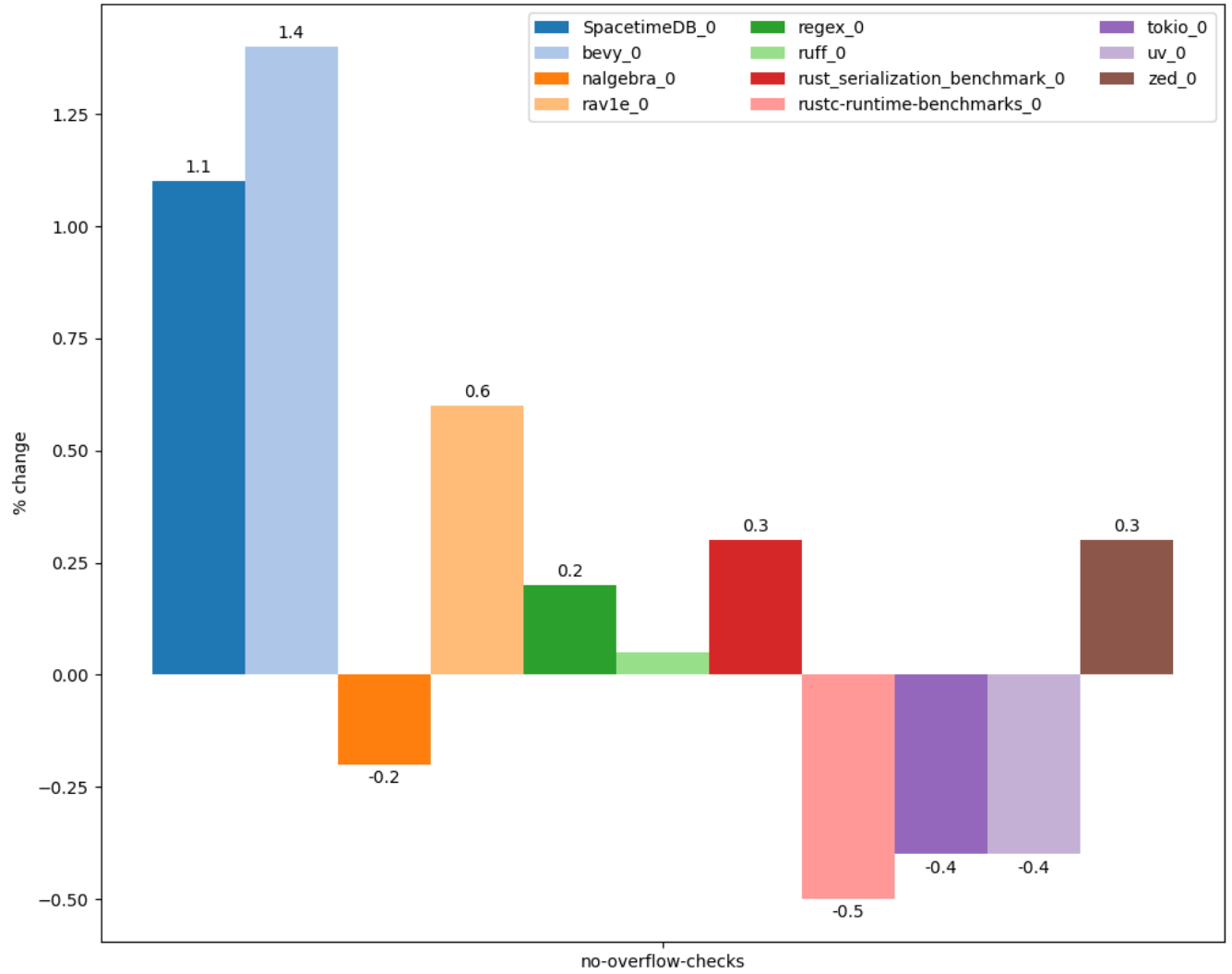
Workarounds

- Небезопасные API для арифметики:
  - unchecked_add, wrapping_add, checked_add, etc.
  - to_int_unchecked
  - std::hint::unreachable_unchecked

```
if x <= i32::MIN >> 1 || x >= i32::MAX >> 1 {
    unsafe { std::hint::unreachable_unchecked() }
}
x * 2 / 2
```

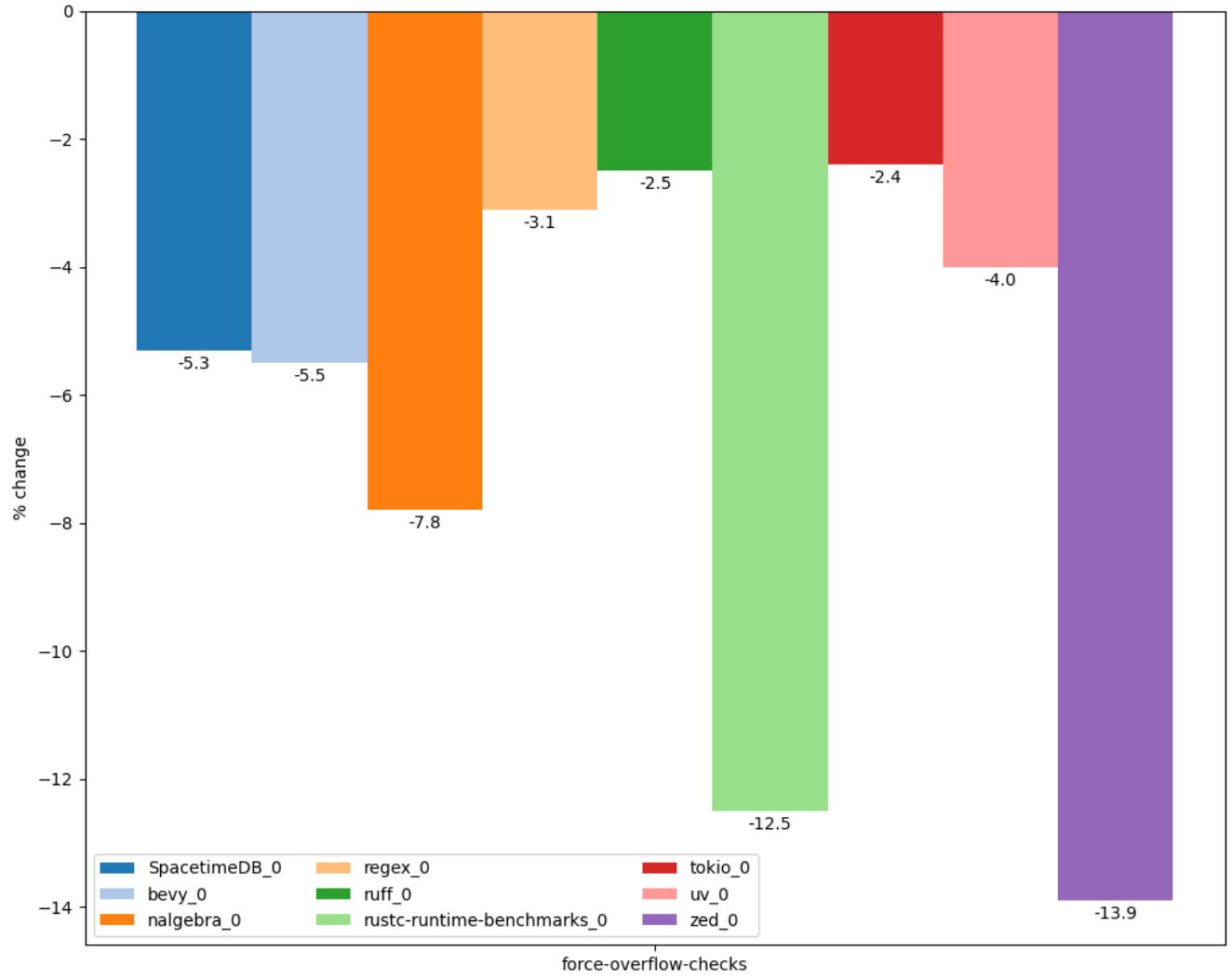
  - (проверки в стандартной библиотеке *не могут* быть отключены)
- Специальные типы стандартной библиотеки:
  - NonZeroU32 для отключения некоторых проверок деления
  - Wrapping<T> для переполнений

Улучшение при отключении дефолтных проверок (деление на ноль и т.д.)



[Ветка компилятора]

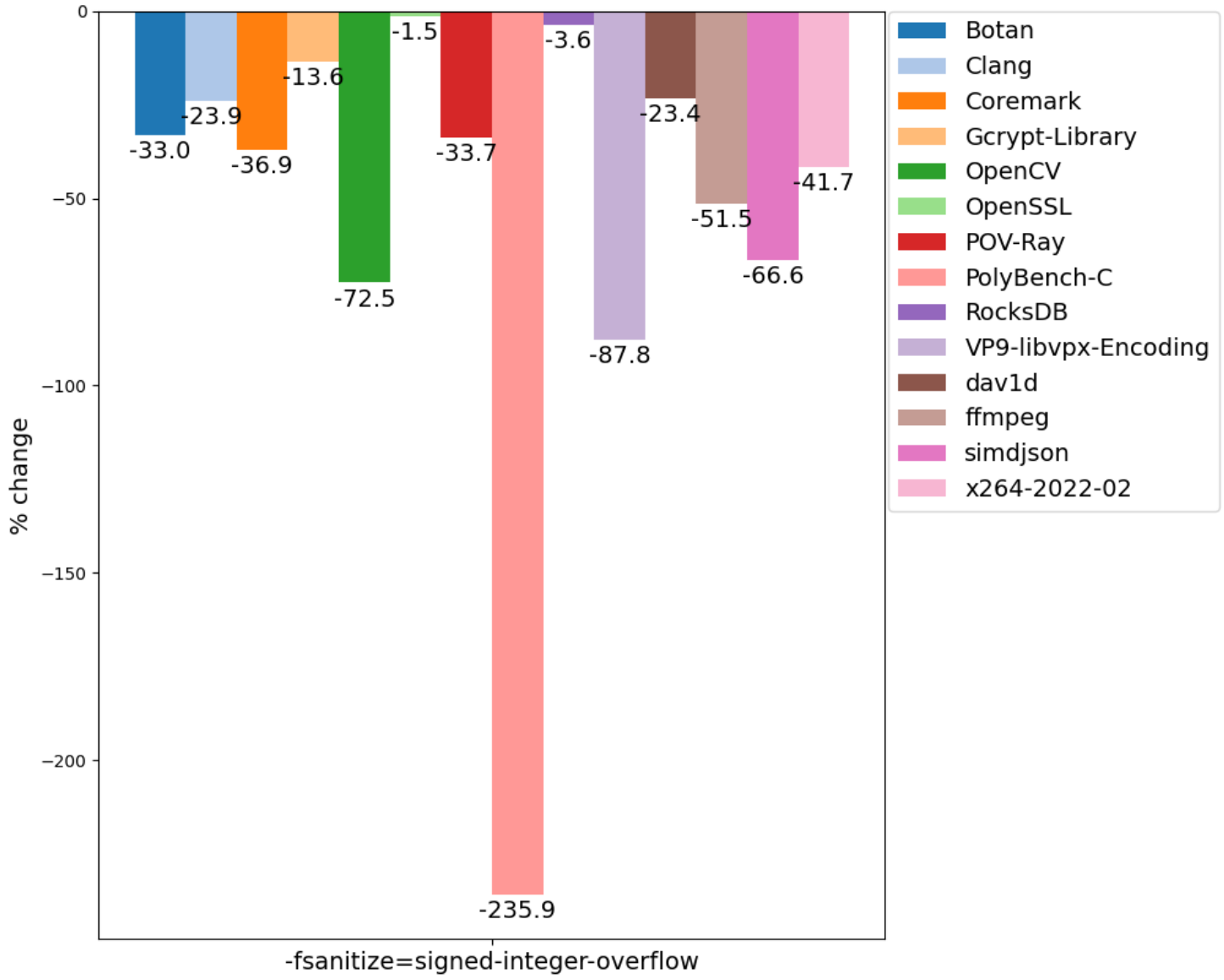
Замедление при включении целочисленных переполнений



- Некоторые тесты отсутствуют из-за ошибок
- Существенный рост числа проверок (+30% проверок в rustc)

[Ветка компилятора]

Замедление в C++



- Некоторые тесты отсутствуют из-за ошибок
- Даже б*о*льший оверхед в UBsan для C/C++

Выводы

- В Rust целочисленное переполнение не является UB
  - Мешает компилятору проводить некоторые агрессивные оптимизации
- В языке приняты неочевидные решения о дефолтных арифметических проверках
  - Действительно важные ошибки типа приведения с потерями не проверяются, зато проверяется деление на ноль
  - Могут давать замедление ~1%
- Целочисленные переполнения имеют слишком большие накладные расходы (до 10%)
  - Аналогично UBsan
  - Вряд ли будут включены по умолчанию в будущем

Рекомендуемые материалы

- RFC 560: Integer Overflow (а также её обсуждение и обзор)
  - Попытка включить проверки переполнений в release
- Understanding Integer Overflow in C/C++
  - Статья John Regehr
- How undefined signed overflow enables optimizations in GCC
  - Статья о пользе Undefined Behavior для integer overflow в C/C++
- Exploiting Undefined Behavior in C/C++ Programs for Optimization: A Study on the Performance Impact
  - Статья о пользе UB, в том числе signed overflow, с замерами

Стандартная библиотека

Детальный разбор

Введение

- По сравнению с STL в стандартной библиотеке Rust делается больше обязательных проверок и используются более устойчивые (и медленные) алгоритмы
  - За счёт встроенной системы сборки Cargo в Rust легко экспериментировать с альтернативными реализациями
- Но в некоторых ситуациях могут использоваться более быстрые контейнеры и алгоритмы (по сравнению с STL)

Проверки в стандартной библиотеке

- Стандартная библиотека Rust содержит большое количество неотключаемых проверок (assert!, checked_add, etc.)
- Основные виды проверок:
  - Корректность индексов в контейнерах (bounds checks, уже рассматривались выше)
  - Отсутствие переполнений при аллокациях (capacity overflow)
  - Успешность аллокаций памяти в контейнерах
  - Корректные обращения к UTF-8 контейнерам &str и String
  - API-специфичные проверки (аксиомы компараторов, etc.)
- Некоторые аналогичные проверки доступны и в STL
  - По макросам _GLIBCXX_ASSERTIONS и _LIBCPP_HARDENING_MODE (Hardened STL)

Пример

```rust
pub fn foo(length: usize) -> Vec<i32> {
    vec![-1; length]
}
```

```
lea     r15, [4*rsi]
xor     r13d, r13d
mov     rax, rsi
// Проверяем что 4*length не переполняется
shr     rax, 62
jne     .LBB0_11
// Проверяем что 4*length <= isize::max
movabs  rax, 9223372036854775804
cmp     r15, rax
ja      .LBB0_11
...
// Проверяем успешность аллокации
call    qword ptr [rip + rustc::rust_alloc]
test    rax, rax
je      .LBB0_11
...
```

https://godbolt.org/z/oa67f1j5z

Пример

```rust
use std::io;

fn main() {
    let mut input = String::new();
    io::stdin()
        .read_line(&mut input)
        .unwrap();
    println!("{}", input);
}
```

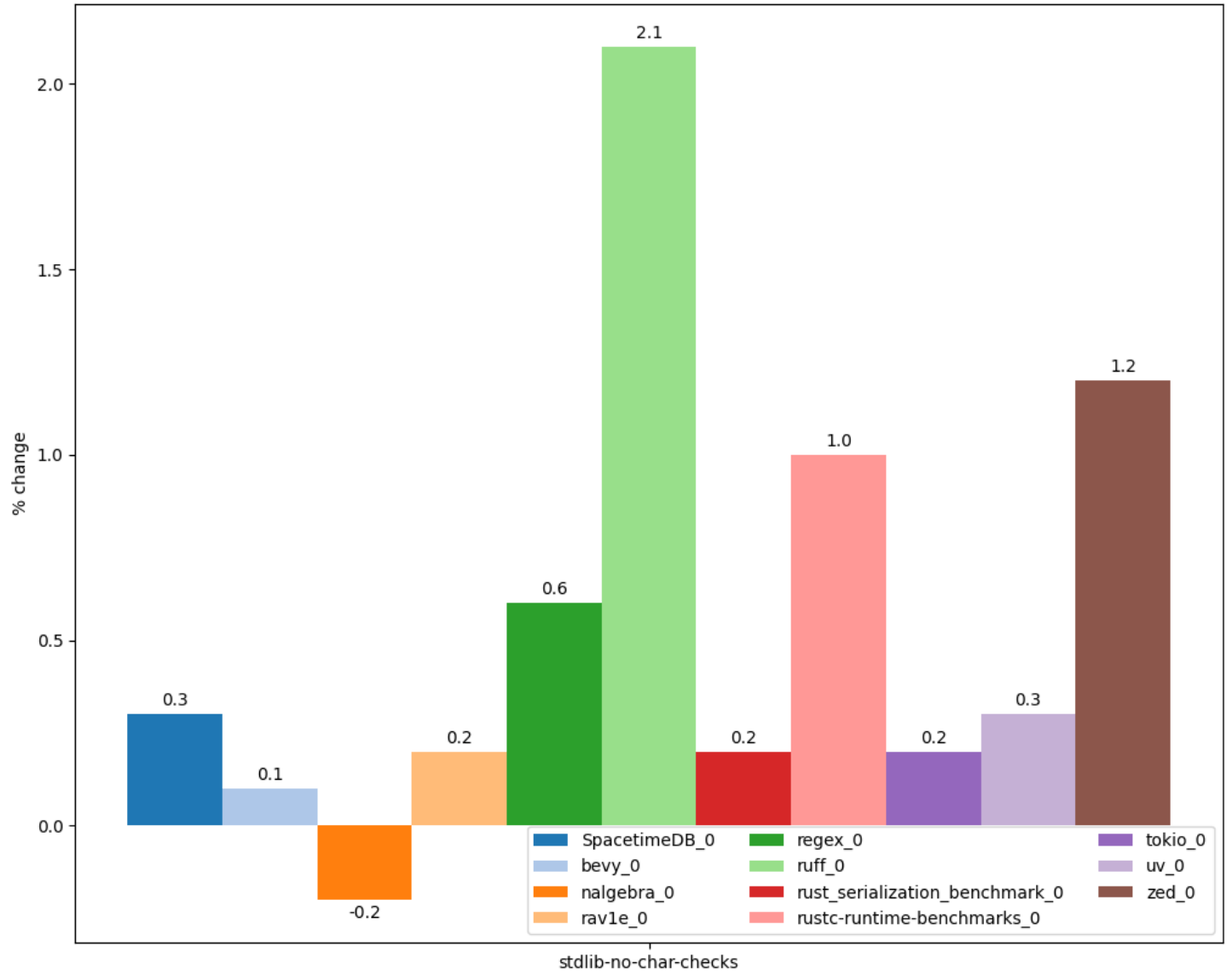
```
$ echo Привет | ./test
Привет

$ echo Привет | iconv -f UTF-8 -t KOI8-RU | ./test
thread 'main' panicked at test.rs:5:39:
called `Result::unwrap()` on an `Err` value:
Error { kind: InvalidData, message: "stream did not contain valid UTF-8" }
```

Другие накладные расходы

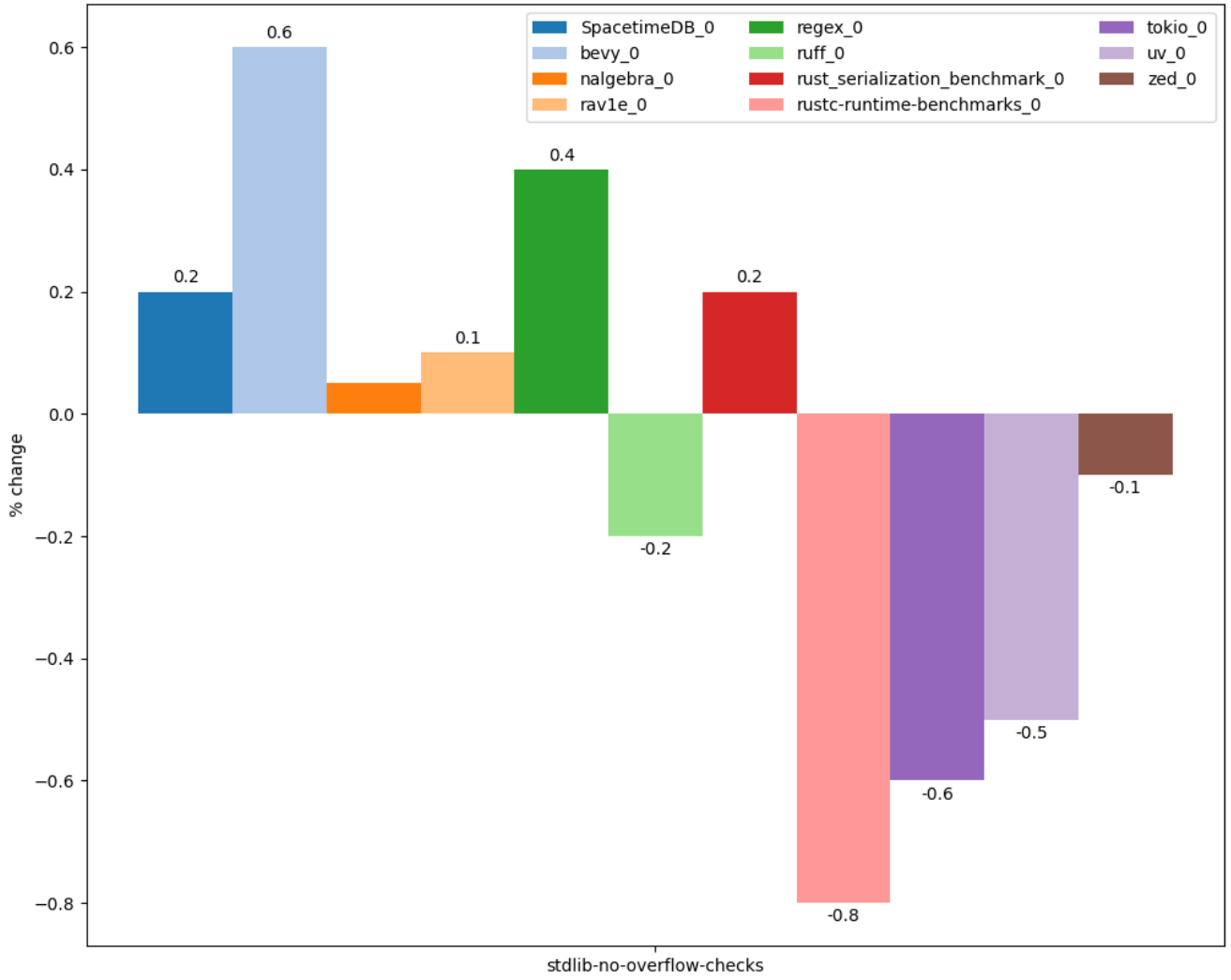
- Дефолтный алгоритм хэширования (SipHash) устойчив к DoS-атакам, но является более медленным чем аналогичные алгоритмы в STL
  - 2x замедление по сравнению с fxhash на простом микробенчмарке
- Ввод-вывод в Rust по умолчанию не буферизуется
  - Без использования BufReader программы могут работать в десятки раз медленней
- io::stdout() строчно-буферизуем даже для не-TTY вывода (rust #60673)
  - Это может сильно замедлять command-line утилиты (в ситуциях типа “./prog > file”)
- Макросы print!/println! используют монопольный доступ для обеспечения корректного выводы из параллельных потоков

Улучшение при отключении проверок UTF-8



[Ветка компилятора]

Улучшение при отключении проверок переполнений в контейнерах



[Ветка компилятора]

Оптимизации в стандартной библиотеке

- Хэштаблицы:
  - Нет требования валидности ссылок после изменения таблицы (как в std::unordered_map)
  - Это позволяет использовать open-addressing вместо chaining (меньше аллокаций, cache-friendly)
- В деревьях поиска используется B-tree (вместо RB-tree как в std::map)
  - Меньше аллокаций, cache-friendly
  - 4x ускорение на простом микробенчмарке
- Современные ILP-friendly алгоритмы сортировки (ipnsort, driftsort)
  - STL-алгоритмы написаны для упрощения векторизации при сортировке примитивных типов
  - 65% ускорение на простом микробенчмарке

Выводы

- В стандартной библиотеке языка есть как факторы, негативно влияющие на производительность, так и более эффективные (по сравнению с STL) алгоритмы и контейнеры
- Недостатки стандартной библиотеки не критичны из-за
  - Наличия большого числа альтернативных реализаций
  - Простоты интеграции внешних библиотек в инфраструктуре Cargo

Рекомендуемые материалы

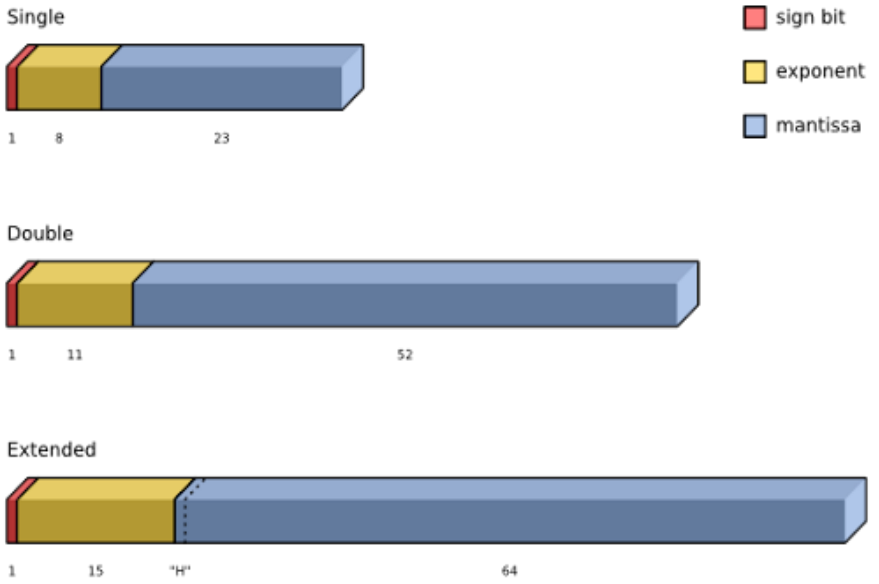
- Safety vs Performance. A case study of C, C++ and Rust sort implementations

Отсутствие режима fast-math

Детальный разбор

Вычисления с плавающей запятой

- Стандарт IEEE 754
  - Типы и их представление
  - Операции над ними
  - Округление
- Большинство архитектур, языков и компиляторов следуют стандарту (почти полностью)
- Причины расхождений
  - Аппаратная специфика
  - Больше возможностей для оптимизации
- fast-math – собирательный термин для оптимизаций, связанных с ослаблением требований IEEE 754



Виды fast-math флагов

- Алгебраические:
  - Ассоциативность
  - Беззнаковость нуля
  - Замена деления на умножение
  - Объединение операций (например FMA)
- Запрет нестандартных значений:
  - NaN, Infinity

```
double foo(double val) {
  return (val + 3.0) / 3.0;
}
```

```
.LCPI0_0:
        .quad   0x4008000000000000 # 3.0
foo:
        movsd   xmm1, qword ptr [rip +
.LCPI0_0]
        addsd   xmm0, xmm1
        divsd   xmm0, xmm1
        ret
```

clang-21 -O2
https://godbolt.org/z/Tqvsv4Tqs

```
.LCPI0_0:
        .quad   0x4008000000000000 # 3.0
.LCPI0_1:
        .quad   0x3fd5555555555555 # 1/3
foo:
        addsd   xmm0, qword ptr [rip +
.LCPI0_0]
        mulsd   xmm0, qword ptr [rip +
.LCPI0_1]
        ret
```

clang-21 -O2 -freciprocal-math
https://godbolt.org/z/6sn7G4r4P

Как на это смотрят языки программирования

- У разных языков различные отношения к соблюдению IEEE 754
- Стандарт C запрещает fast-math оптимизации (в Annex F)
  - Но обычно их можно включить с помощью флагов (например -ffast-math)
  - Часто используется в игровых движках
- Аналогичное поведение для языка Swift
- Java и C# разрешают совершать вычисления с б*о*льшей точностью, чем результирующий тип
- Go разрешает объединение операций
  - Но только если этому не препятствуют явные преобразования типов
- Fortran явно разрешает нарушать IEEE 754 ради производительности

Оптимизации: Векторизация

- -fassociative-math и -fno-signed-zeros позволяет компилятору переупорядочить итерации цикла и векторизовать вычисления

```
#include <math.h>

float sum256(float *arr1,
             float *arr2) {
  float s;
  int i;
  s = 0.0f;
  for (i = 0; i < 256; i++) {
    s += arr1[i] * arr2[i];
  }
  return s;
}
```

```
.LBB0_1:
  movss   xmm1, dword ptr [rdi + 4*rax]
  mulss   xmm1, dword ptr [rsi + 4*rax]
  addss   xmm0, xmm1
  # один элемент за итерацию цикла
  inc     rax
  cmp     rax, 256
  jne     .LBB0_1
```

clang-21 -O2 -fno-unroll-loops
https://godbolt.org/z/v5vvEb1Th

```
.LBB0_1:
  movups  xmm1, xmmword ptr [rdi + 4*rax]
  movups  xmm2, xmmword ptr [rsi + 4*rax]
  mulps   xmm2, xmm1
  addps   xmm0, xmm2
  # 4 элемента за итерацию цикла
  add     rax, 4
  cmp     rax, 256
  jne     .LBB0_1
```

clang-21 -O2 -fno-unroll-loops -fassociative-math -fno-signed-zeros
https://godbolt.org/z/j6P1ohcTr

Оптимизации: Упрощение выражений

- Fast-math флаги расширяют возможности компилятора по вычислению выражений во время компиляции
- -fassociative-math + -fno-signed-zeros

```
double foo(double val) {
  return 3.0 + val - 3.0;
}
```

```
foo:
  addsd   xmm0, qword ptr [rip + .LCPI0_0] # +3.0
  addsd   xmm0, qword ptr [rip + .LCPI0_1] # -3.0
  ret
```

clang-21 -O2
https://godbolt.org/z/hjzWejv38

```
foo:
  ret
```

clang-21 -O2 -fassociative-math -fno-signed-zeros
https://godbolt.org/z/a5GME55MM

- -fassociative-math + -fno-signed-zeros + -freciprocal-math

```
double foo(double val) {
  return 3.0 * val / 3.0;
}
```

```
foo:
  movsd   xmm1, qword ptr [rip + .LCPI0_0]
  mulsd   xmm0, xmm1 # * 3.0
  divsd   xmm0, xmm1 # / 3.0
  ret
```

clang-21 -O2
https://godbolt.org/z/nancn6j8x

```
foo:
  ret
```

clang-21 -O2 -fassociative-math -fno-signed-zeros -freciprocal-math
https://godbolt.org/z/dMeKc8PW9

Оптимизации: Сложные инструкции

- Некоторые архитектуры поддерживают более сложные операции над числами с плавающей точкой
  - Например, Fused Multiply Add в x86-64
- Алгоритм округления в этих инструкция может отличаться от последовательного выполнения элементарных операций
- Флаг -ffpcontract=on позволяет компилятору использовать эти инструкции

```
double foo(double *val) {
  double acc = 1;
  for (int i = 0; i < 100; ++i) {
    acc = acc * val[i] + val[i + 1];
  }
  return acc;
}
```

```
.LBB0_1:
        mulsd   xmm0, xmm1 # multiply
        movsd   xmm1, qword ptr [rdi + 8*rax]
        addsd   xmm0, xmm1 # add
        inc     rax
        cmp     rax, 101
        jne     .LBB0_1
        ret
```

clang-21 -O2 -fno-unroll-loops
https://godbolt.org/z/qTTY4r7Yx

```
.LBB0_1:
        vmovsd  xmm2, qword ptr [rdi + 8*rax]
        vfmadd213sd     xmm0, xmm1, xmm2 # fused multiply-add
        inc     rax
        vmovapd xmm1, xmm2
        cmp     rax, 101
        jne     .LBB0_1
        ret
```

clang-21 -O2 -fno-unroll-loops -ffp-contract=on -march=haswell
https://godbolt.org/z/zvPTo9fz4

Оптимизации: Finite only

- -ffinite-math-only разрешает оптимизации, основанные на том, что операнды не принимают NaN или Inf значений
- Это позволяет:
  - Удалять проверки на NaN и Inf
  - x == x => true
  - x / x => 1
- А с -fno-signed-zeros:
  - x – x => 0
  - 0 / x => 0
  - x * 0 => 0

```
int foo(double val) {
  return val == val;
}
```

```
foo:
  ucomisd xmm0, xmm0
  setnp   al
  ret
```

clang-21 -O2
https://godbolt.org/z/v5YEq7hbY

```
foo:
  mov     al, 1
  ret
```

clang-21 -O2 -ffinite-math-only
https://godbolt.org/z/TYab9sndr

```
double foo(double val) {
  return val * 0;
}
```

```
foo:
  xorpd   xmm1, xmm1
  mulsd   xmm0, xmm1
  ret
```

clang-21 -O2
https://godbolt.org/z/3e8sM1dh9

```
foo:
  xorps   xmm0, xmm0
  ret
```

clang-21 -O2 -ffinite-math-only -fno-signed-zeros
https://godbolt.org/z/zoWrsfr1h

Оптимизации: Errno

- Флаг -fno-math-errno позволяет компилятору оптимизировать проверки, связанные с выставлением переменной errno в математических функциях
- В частности это разрешает инлайнинг и упрощение библиотечных функций
- Стандартные математические методы Rust компилируются в LLVM интринсики, не выставляющие errno

```c
#include <math.h>

double foo(double x) {
  return sqrt(x);
}
```

```
foo:
        xorpd   xmm1, xmm1
        ucomisd xmm0, xmm1
        jb      sqrt@PLT
        sqrtsd  xmm0, xmm0
        ret
```

clang-21 -O2

```
foo:
        sqrtsd  xmm0, xmm0
        ret
```

clang-21 -O2 -fno-math-errno

```rust
pub fn foo(num: f64) -> f64 {
    num.sqrt()
}
```

```
foo:
        sqrtsd  xmm0, xmm0
        ret
```

Подводные камни: Проверки

- Появление Inf или NaN значений при использовании -ffinite-math-only приводит к UB
- Если отсутствие этих значений обеспечивалось явными проверками, то они будут удалены компилятором

```
#include <math.h>

int foo(float f) {
  return isnan(f);
}
```

```
foo:
  xor     eax, eax
  ucomiss xmm0, xmm0
  setp    al
  ret
```

clang-21 -O2
https://godbolt.org/z/MPPoc5oea

```
foo:
  xor     eax, eax
  ret
```

clang-21 -O2 -ffinite-math-only
https://godbolt.org/z/nv33ffaqx

Подводные камни: Округление

- Некоторые алгоритмы используют специфику IEEE 754 и специальный порядок вычислений для уменьшения погрешности округления
- С -fassociative-math, компилятор “оптимизирует” эти алгоритмы так, как будто погрешности округления нет
- Алгоритм Кэхэна

```
float sum(float *arr) {
  float s;
  int i;
  s = 0.0f;
  for (i = 0; i < 256; i++) {
    s = s + arr[i];
  }
  return s;
}
```

```
sum_kahan:
  xorps   xmm0, xmm0
  xor     eax, eax
.LBB0_1:
  addss   xmm0, dword ptr [rdi +
4*rax]
  inc     rax
  cmp     rax, 256
  jne     .LBB0_1
  ret
```

clang-21 -O2 -ffast-math -fno-unroll-loops -fno-vectorize

https://godbolt.org/z/rYebWsfco

```
float sum_kahan(float *arr) {
  float s, c, t, y;
  int i;
  s = c = 0.0f;
  for (i = 0; i < 256; i++) {
    y = arr[i] - c;
    t = s + y;
    c = (t - s) - y;
    s = t;
  }
  return s;
}
```

Позиция Rust

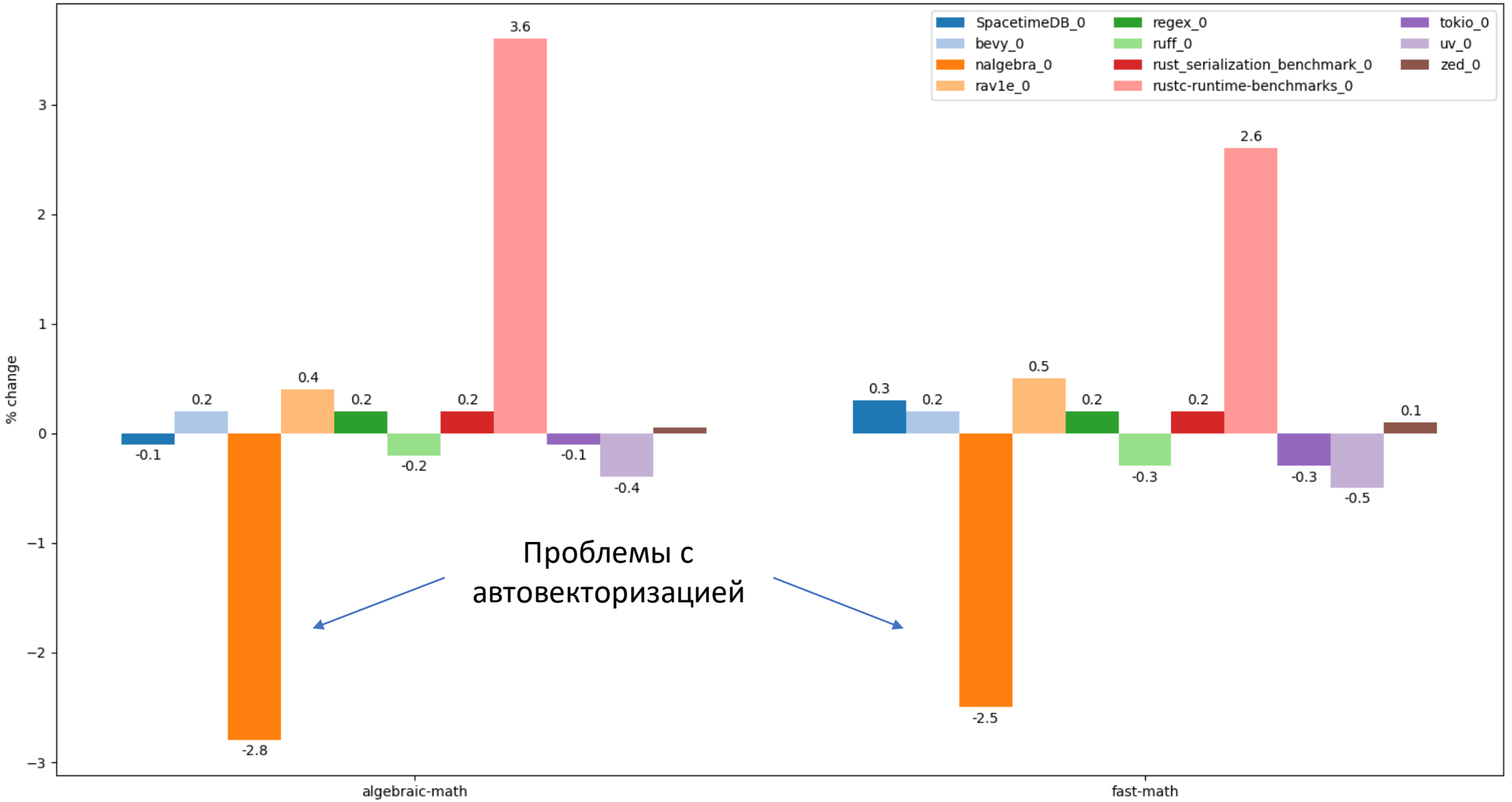
- Rust отказывается от возможности глобального применения fast-math оптимизаций
- Основная мотивация – опасность недостаточно осторожного применения этих оптимизаций, особенно в зависимостях проекта

> “Everything about this flag fundamentally clashes with the idea of robust, compositional system design. It was a terrible mistake to ever add it to C compilers, and Rust should not repeat that mistake. Rust should instead explore alternative, less fragile ways of exposing those semantics.” (Ralf Jung, rust #21690)

Workarounds: Интринсики

- Rust предоставляет fast-math оптимизации на уровне отдельных операций
  - std::intrinsics::f{op}_fast
  - std::intrinsics::f{op}_algebraic
  - f(16/32/64)::algebraic_{op}
- Недостатки:
  - На данный момент эти API недоступны в стабильной версии
  - На данный момент нет типов, скрывающих внутри себя работу с интринсиками
    - Авторам математических библиотек придётся реализовывать код дважды (для fast-math и обычного варианта)

Результаты при включении fast-math



[Ветка компилятора](#) [Ветка компилятора](#)

Выводы

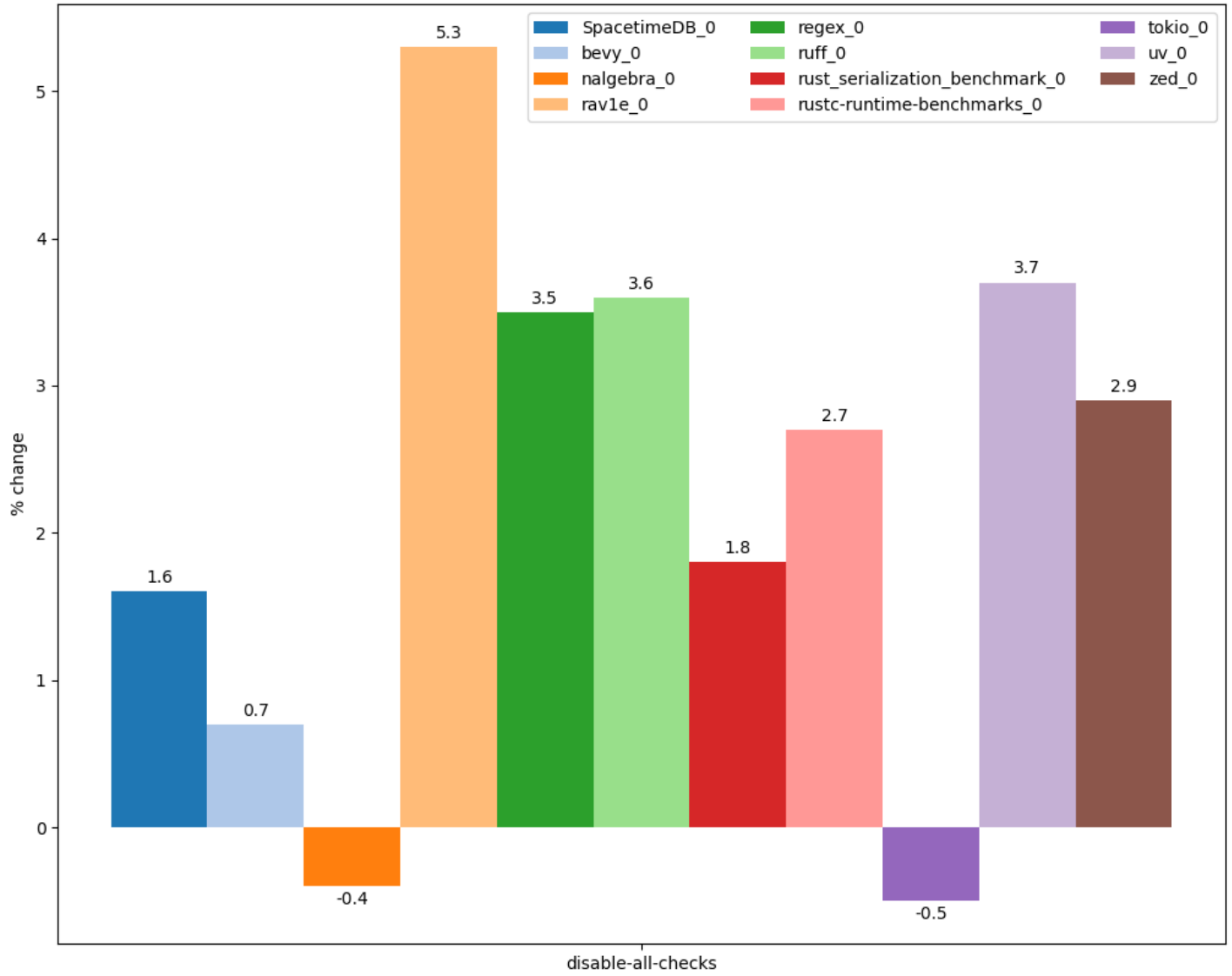
- Rust не предоставляет возможности глобально включить fast-math оптимизации
- Разрешать эти оптимизации можно на уровне отдельных операций с помощью интринсиков
  - Пока только в nightly-версии
- Fast-math может как положительно, так и отрицательно влиять на производительность

Рекомендуемые материалы

- Optimizations enabled by -ffast-math (Krister Walfridsson, GCC)
- Towards Useful Fast-Math (LLVM Dev Meeting 2024)
- Beware of fast-math
- P3715R0: Tightening floating-point semantics for C++
- Taming Floating-Point Sums

Выводы

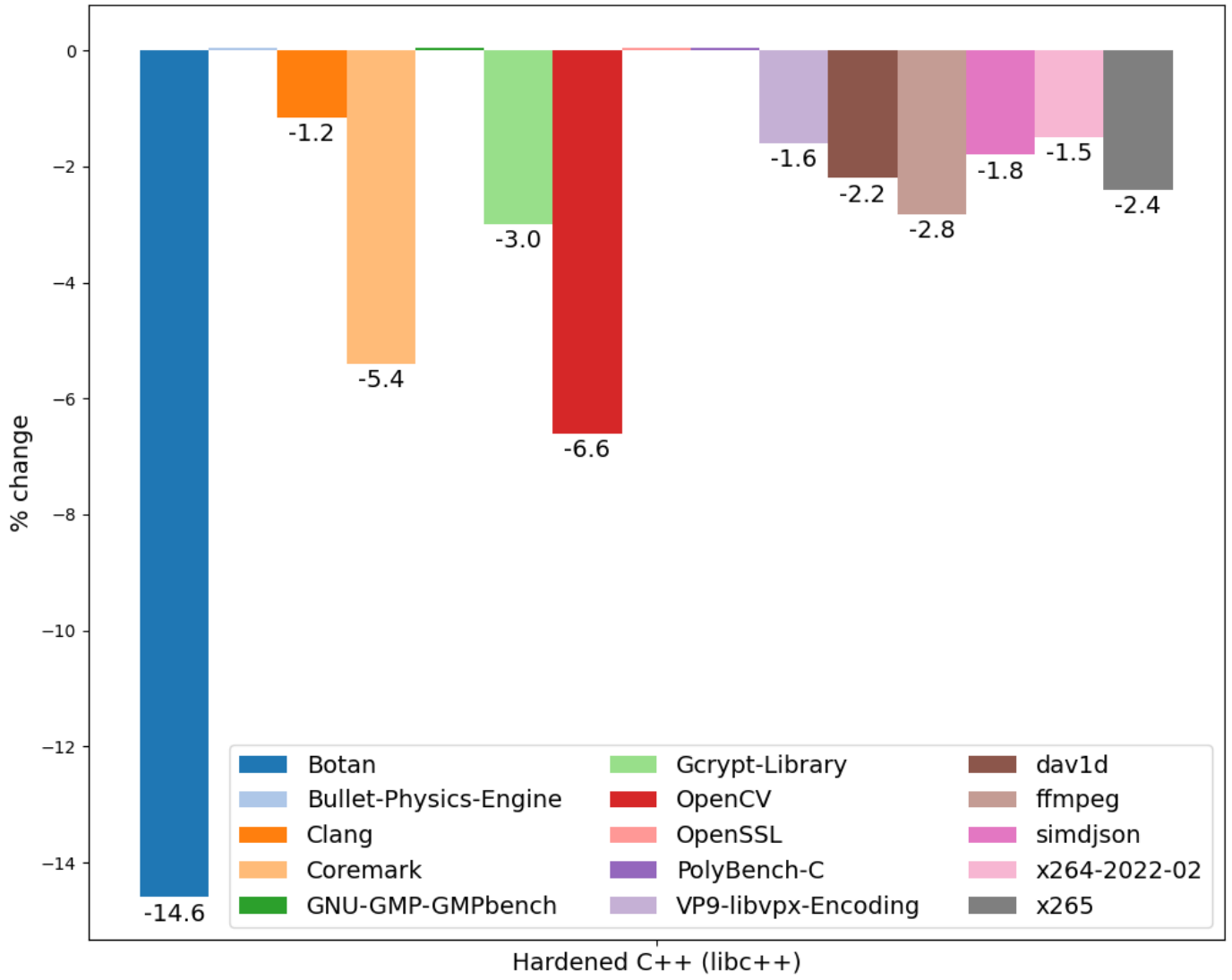
Ускорение при отключении (почти) всех проверок Rust



- Отключались проверки индексов и арифметики, переполнения контейнеров stdlib, проверки UTF-8 в строках
- Не отражены накладные расходы на избыточную инициализацию (полагаем не менее 1%)

[Ветка компилятора]

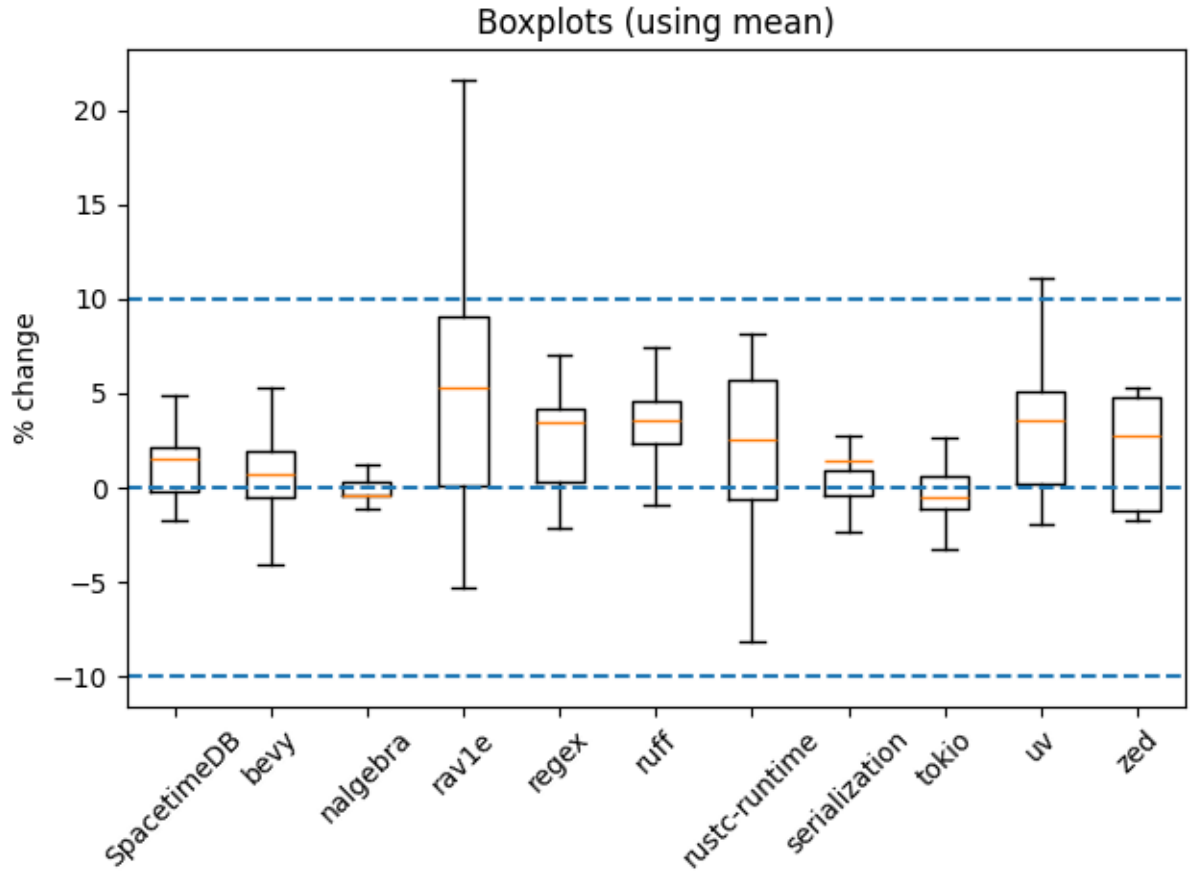
Замедления в Hardened C++



- Включенные защиты:
  - -D_FORTIFY_SOURCE=3
  - -fsanitize=bounds,object-size
  - Hardened STL
  - (Инициализацию не включали чтобы соответствовать Rust)
- В отличие от Rust большинство проектов **не тюнилось** под Hardened C++

[Ветка компилятора]

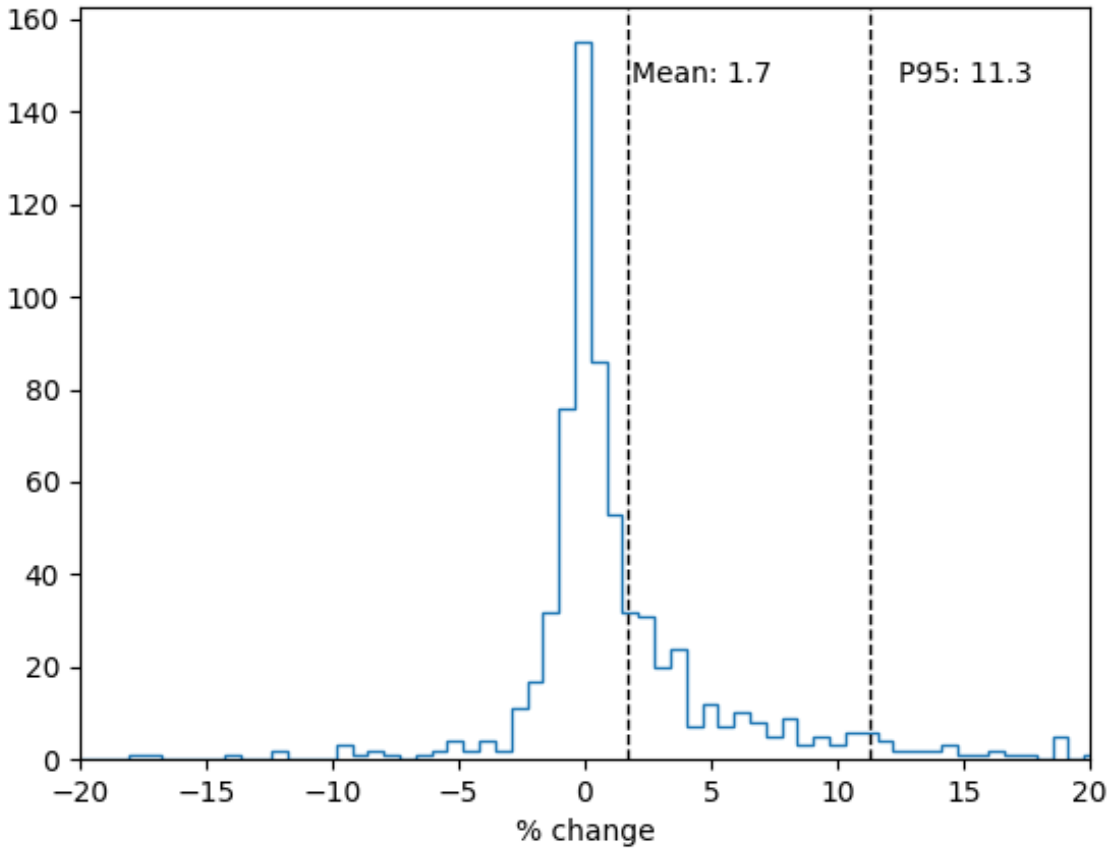
Распределение замедлений в бенчмарках



- Собирались данные по всем бенчмаркам внутри каждого проекта
- Одинаковые тесты с разными параметрами усреднялись
- Это **смещённая** оценка:
  - Сценарии внутри проекта с б*о*льшим числом бенчей доминируют
- Не отражены накладные расходы на инициализацию (не менее 1% для среднего)

[Скрипт для построения графиков]

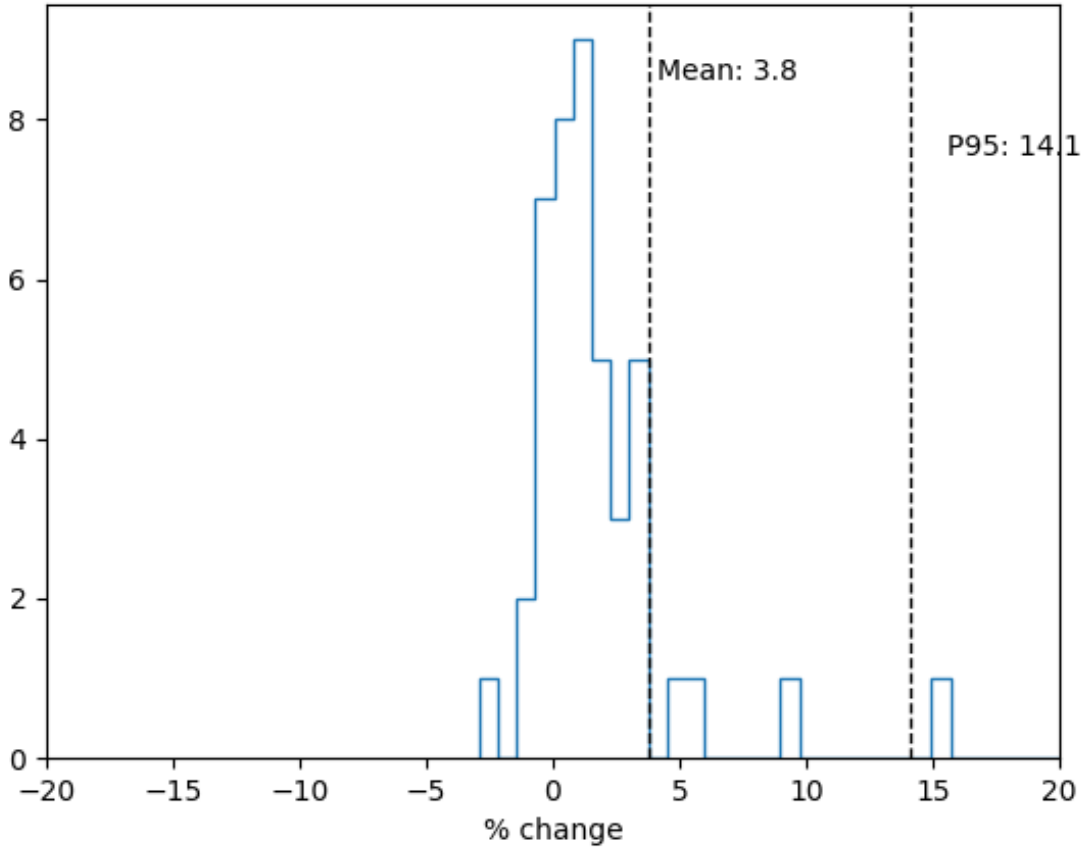
Распределение замедлений в бенчмарках



- Данные носят иллюстративный характер!
- Собирались данные по всем бенчмаркам всех проектов
- Одинаковые тесты с разными параметрами усреднялись
- Это **смещённая** оценка:
  - Проекты с б*о*льшим числом бенчей доминируют
  - Сценарии внутри проекта с б*о*льшим числом бенчей доминируют
- Не отражены накладные расходы на инициализацию (не менее 1% для среднего)

[Скрипт для построения графиков]

Распределение замедлений в бенчмарках Hardened C++



- Данные носят иллюстративный характер!
- Намного меньше бенчмарков чем в случае Rust
- В отличие от Rust большинство проектов **не тюнилось** под Hardened C++

[Скрипт для построения графиков]

Выводы

- Сообщество Rust следит за балансом безопасности и производительности:
  - Бескомпромиссно включаются только проверки memory safety
  - Остальные делаются только если не влияют существенно на производительность
  - Например по умолчанию отключены проверки integer overflow
- Оверхед динамических проверок зависит от проекта
  - В среднем не превосходит 5% (P95 11%) и сравним с Hardened C++
- Факторы, наиболее существенно влияющие на производительность программ:
  - Индексные проверки, обязательная ициализация, отсутствие inplace-инициализации
- Скорее всего будут оптимизироваться в будущих версиях Rust и LLVM
- Но полностью их негативное влияние обойти нельзя (по крайней мере в текущей версии языка)
- Всегда будут требовать тюнинга (в частности небезопасного кода) для достижения оптимальной производительности

3dman_eu, https://www.needpix.com/photo/1171460/weigh-plus-minus-horizontal-plus-minus-balance-sheet-cost-calculation-course

О чём мы не успели рассказать …

- Codegen units
  - Похожи на unity builds в C/C++
- Особенности реализации паник (исключений)
  - Накладные расходы аналогичны исключениям в C++
- GC-sections by default
- Особенности ABI
- Отключение PLT
- Особенности реализации виртуальных методов
- Проверки Stack Clashing

https://vectorportal.com/vector/timeglass/34811

The End

Выражаем благодарность нашим рецензентам:

- Вадим Петроченков
  - https://github.com/petrochenkov
- Александр Чичигин
  - https://github.com/Gabriel-Fallen
- Jakub Beránek
  - https://github.com/kobzol
- Александр Монаков
  - https://github.com/amonakov
  - https://codeberg.org/amonakov
- Per Vognsen
  - https://mastodon.social/@pervognsen

Последняя версия слайдов:
https://github.com/yugr/rust-slides/blob/main/RU.pdf

Приложения

Подготовка доклада

- Собрано и проанализировано более 350 статей, блогпостов и обсуждений по теме доклада из разных источников (materials)
- Общие временные затраты на подготовку (на апрель 2026):
  - > 3 человекомесяцев
- Анализ проводился для
  - Rust 1.87.0 (май 2025)
  - Clang llvmorg-20.1.7 (июнь 2025)

Rust-бенчмарки

- Для сравнения использовались стандартные Criterion-бенчмарки каждого проекта (cargo bench), усреднённые с помощью geomean
  - Некоторые особо медленные тесты в oxipng были отключены
  - Для борьбы с системными шумами бралось минимальное значение средних из нескольких прогонов
  - Важно: 1% geomean может означать что
    - Все тесты в проекте замедлились на 1%
    - 10% тестов замедлилось на 10% (более вероятно)
  - Отдельные тесты могли замедлиться (или ускориться) на десятки процентов
- Оценка с помощью geomean неидеальна
  - Сценарии с бо*л*ьшим количеством бенчей сильнее влияют на результат
- Дополнительные детали здесь

C++ бенчмарки

- Использовался компилятор Clang версии llvmorg-20.1.7
- Llvm-bench (benchmarks/cpp/llvm-bench)
  - Замерялось медианное время O2-компиляции файла CGBuiltin.cpp (самый большой файл в LLVM) компилятором Clang
  - Тестируемый Clang был собран самим собой
- Ffmpeg-bench (benchmarks/cpp/ffmpeg-bench)
  - По мотивам Stack Smashing Protection and Its Performance Impact
  - Замерялось медианное время конвертации файла FLV в MP4
- Phoronix Test Suite (benchmarks/cpp/PTS)
  - Популярный набор бенчмарков с phoronix-test-suite.com
  - Замерялись бенчмарки botan, bullet, coremark, c-ray, dav1d, gcrypt, gmpbench, luajit, nginx, opencv, openssl, polybench-c, povray, rocksdb, simdjson, vpxenc, x265 (см. #55)
  - Использовалось медианное время работы бенчмарка
  - Игнорировались тесты с Stdev > 0.5% и изменениями < 1%
- Для борьбы с системными шумами бралось минимальное значение медиан из 2-3 прогонов

Тестовый сервер

- Все бенчмарки (кроме no-static) были замерены на Intel Xeon E-2236
  - С частотой зафиксированной на 2 ГГц
  - С установленной Ubuntu 24.04
- Сервера настраивались в соответствии с методологией ulnit

TODO

- (Yugr) Померять meilisearch
- (Zak) Fast-math: дополнить анализ (P3715, -fno-math-errno и -fno-trapping-math)
- (Zak) Struct reorder: заполнить анализ