Производительность языка Rust

Май 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

Полная версия слайдов

- Это сокращённая версия слайдов
- Полная версия больше на 30%
  - Дополнительные темы, графики, workarounds, пруфлинки и т.п.
- Ещё больше информации есть в нашем репозитории yugr/rust-slides

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

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 множество современных парадигм программирования
  - Процедурное, функциональное, ООП и т.д.
- 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 и enum
  - Оптимизация порядка полей и niche optimizations
- Более эффективные стандартные контейнеры
- …

Цели доклада

- 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 Compiler © ?
- И если нет, то как с ними может справиться программист?
- Есть ли какие-то особенности языка которые наоборот помогают компилятору?

Image source

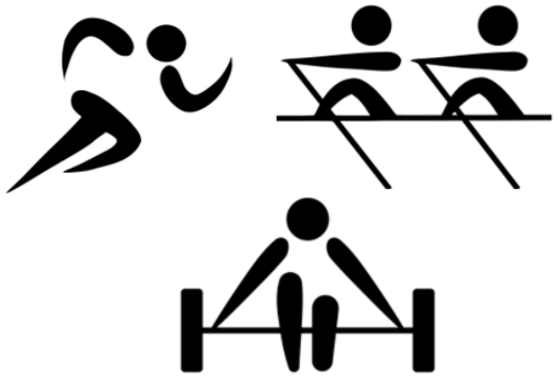
Мнение создателей языка

- 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)
- Эти бенчмарки представляют хорошо оптимизированный код (с помощью unsafe и т.п.)

Image source (Akovin)

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, простота параллельного программирования (Fearless Concurrency with Rust), 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: 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)
    - 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

Image source (Tricia J)

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)

Image source

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
  - Статистика CVE и KEV (2024)
  - 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

Image source (Nick Youngson)

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
...
```

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
```

Workarounds: Общие соображения

- В циклах используйте слайсы, а не контейнеры типа Vec
  - Это упрощает работу alias analysis
- Избегайте сложной индексной арифметики
  - Например x[2 * i + 5]
- 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: 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

Image source

Workarounds: Использование ограниченных типов

- Ограниченность индекса можно выразить с помощью типов
  - Enums или bool

```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: 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

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

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

Image source (Lalada)

Workarounds: Итераторы

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

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

```
.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
```

```
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);
}
```

```
.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
```

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 (“internal iteration”)
  - Are iterators even efficient?

Image source (Karen Arnolds)

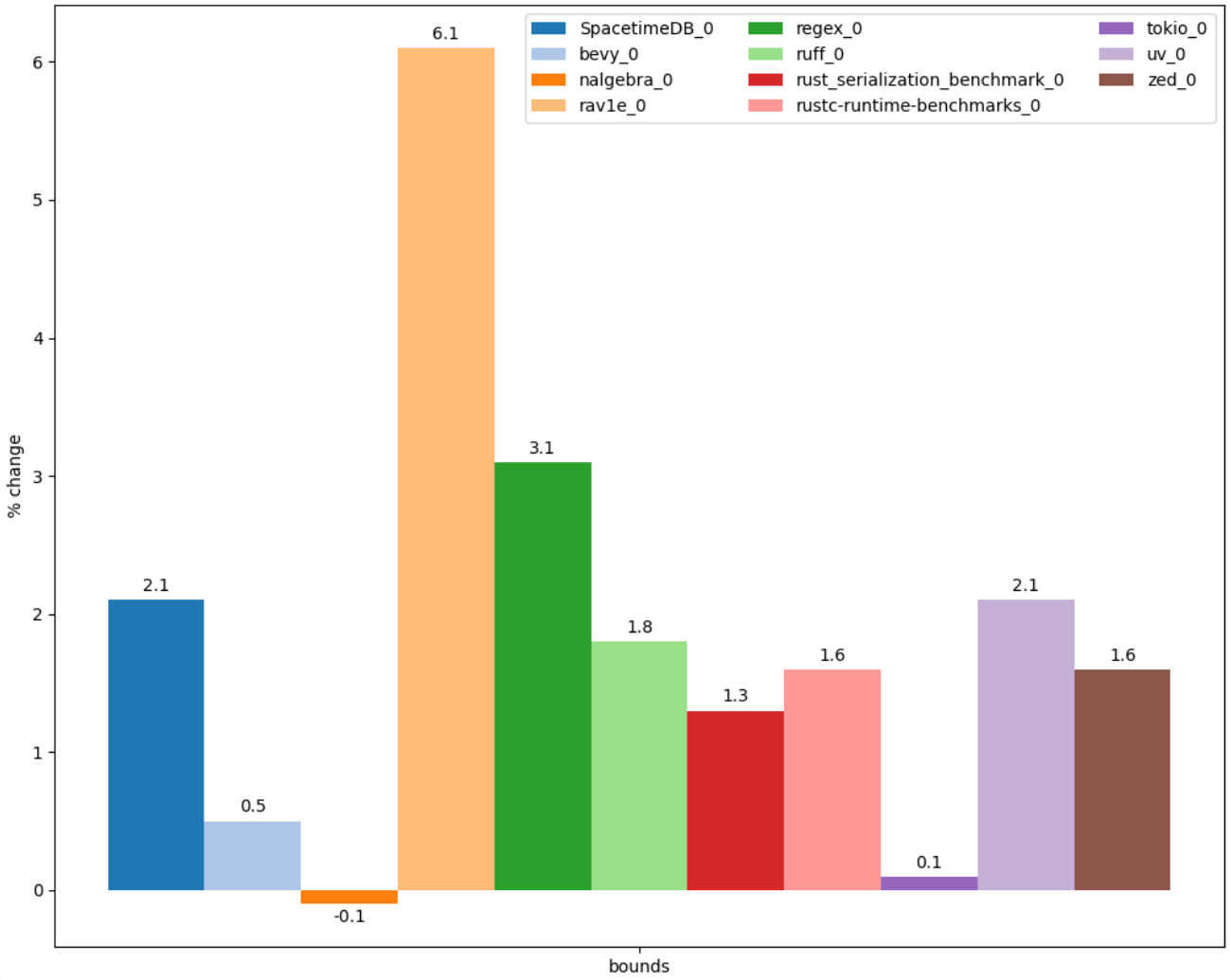
Распространённость

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

Методика анализа

Ветка компилятора

Рост производительности при отключении проверок



[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
  - Только 20% нетривиальных обращений в память в коде Clang могут быть проверены
  - C++ Safe Buffers может быть решением, но требует нетривиальной модификации кода

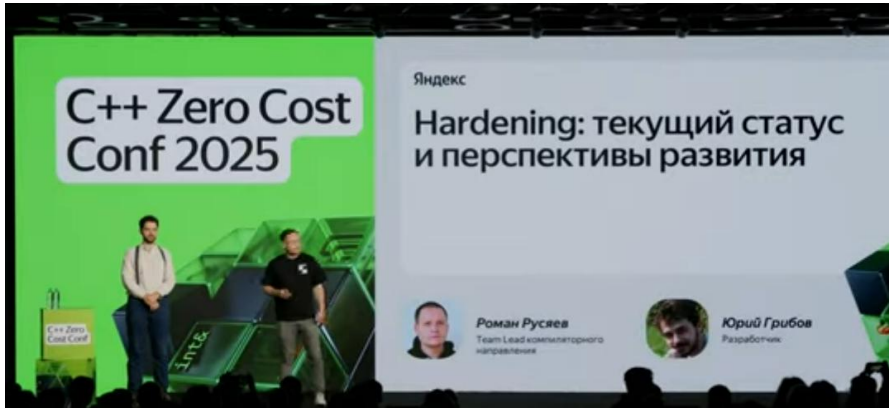


Image source (Mario Simoes)

Rust vs. Hardened C++

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

rustc -O -Cpanic=abort

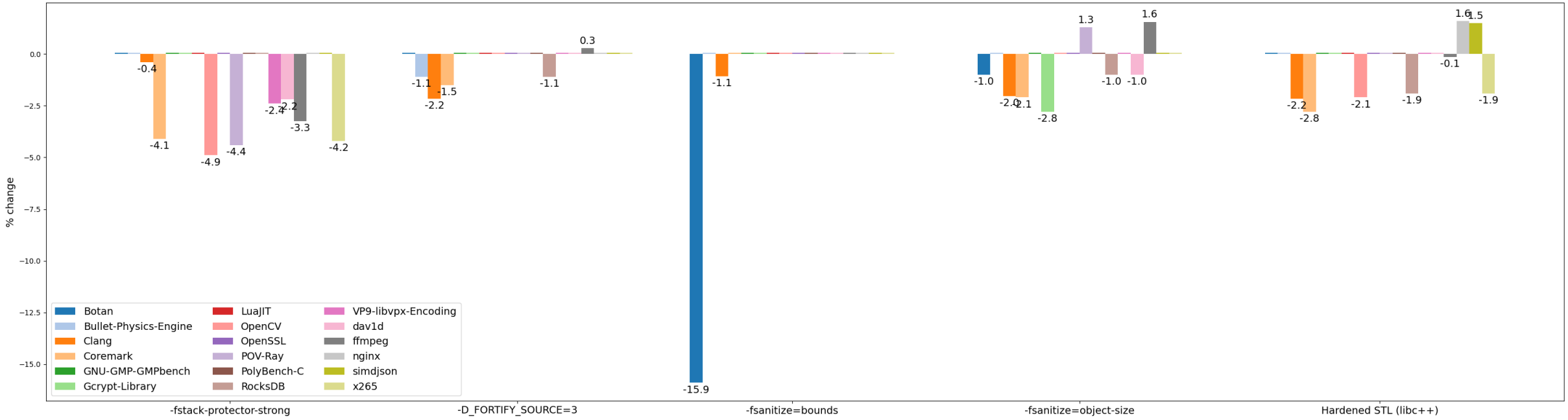
```
foo:
  cmp     rdx, rsi
  jae     .LBB0_2
  mov     eax, dword ptr [rdi + 4*rdx]
  ret
.LBB0_2:
  push    rax
  lea     rax, [rip + .Lanon.1]
  ...
  call    qword ptr [rip + panic_bounds_check]
```

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

gcc -O2 -D_GLIBCXX_ASSERTIONS

```
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.)

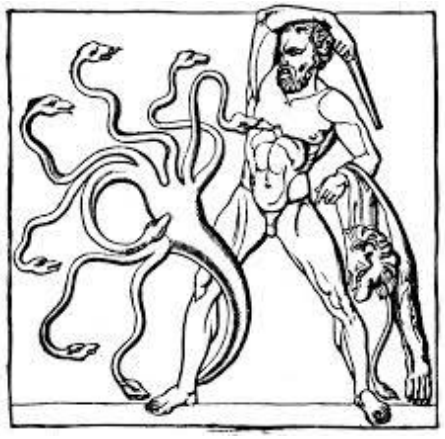
Выводы

- 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

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 (Fortran 77)
  - Любые указатели могут ссылаться на одни и те же данные (ранние версии языка C)
  - Промежуточные варианты
    - Strict (type-based) aliasing в C89 и restrict-аннотации в C99

Image source

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
```

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
```

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

```
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
```

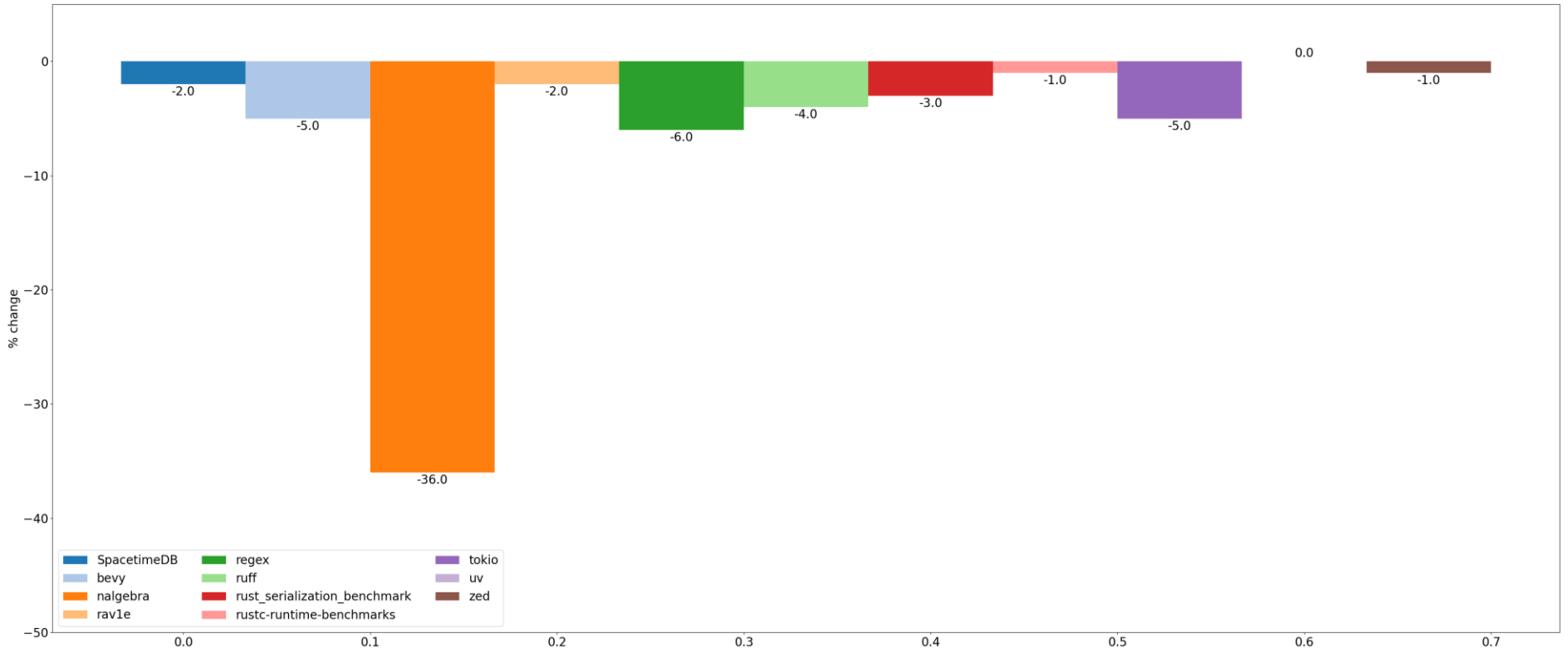
Ограничения

- Частный случай этой 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;
    }
}
```

Эффективность правил алиасинга Rust

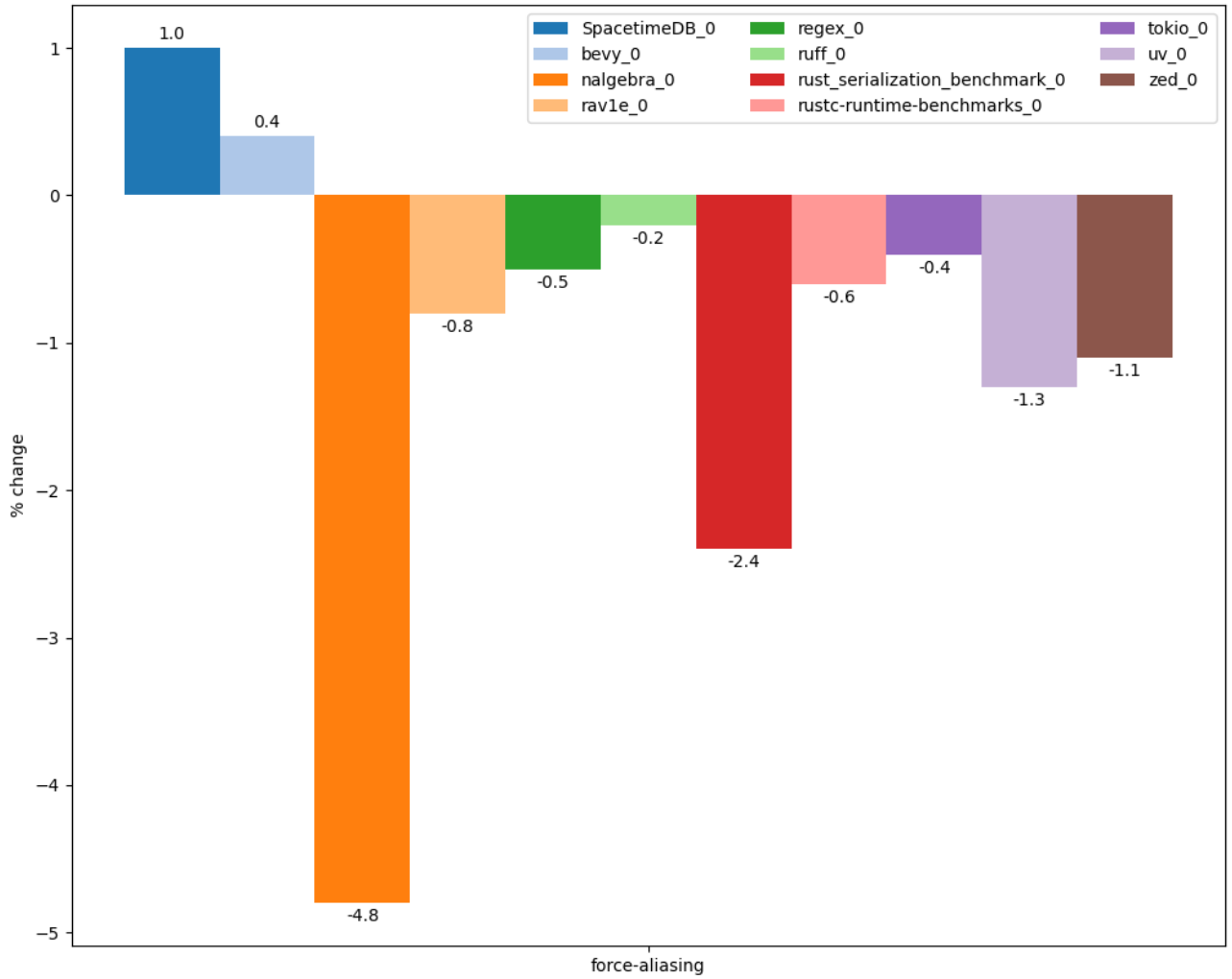


Точность алиас-анализа – число пар указателей, для которых анализ смог дать чёткий ответ (MustAlias / NoAlias)

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

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)

Image source (Alex Dos Santos)

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

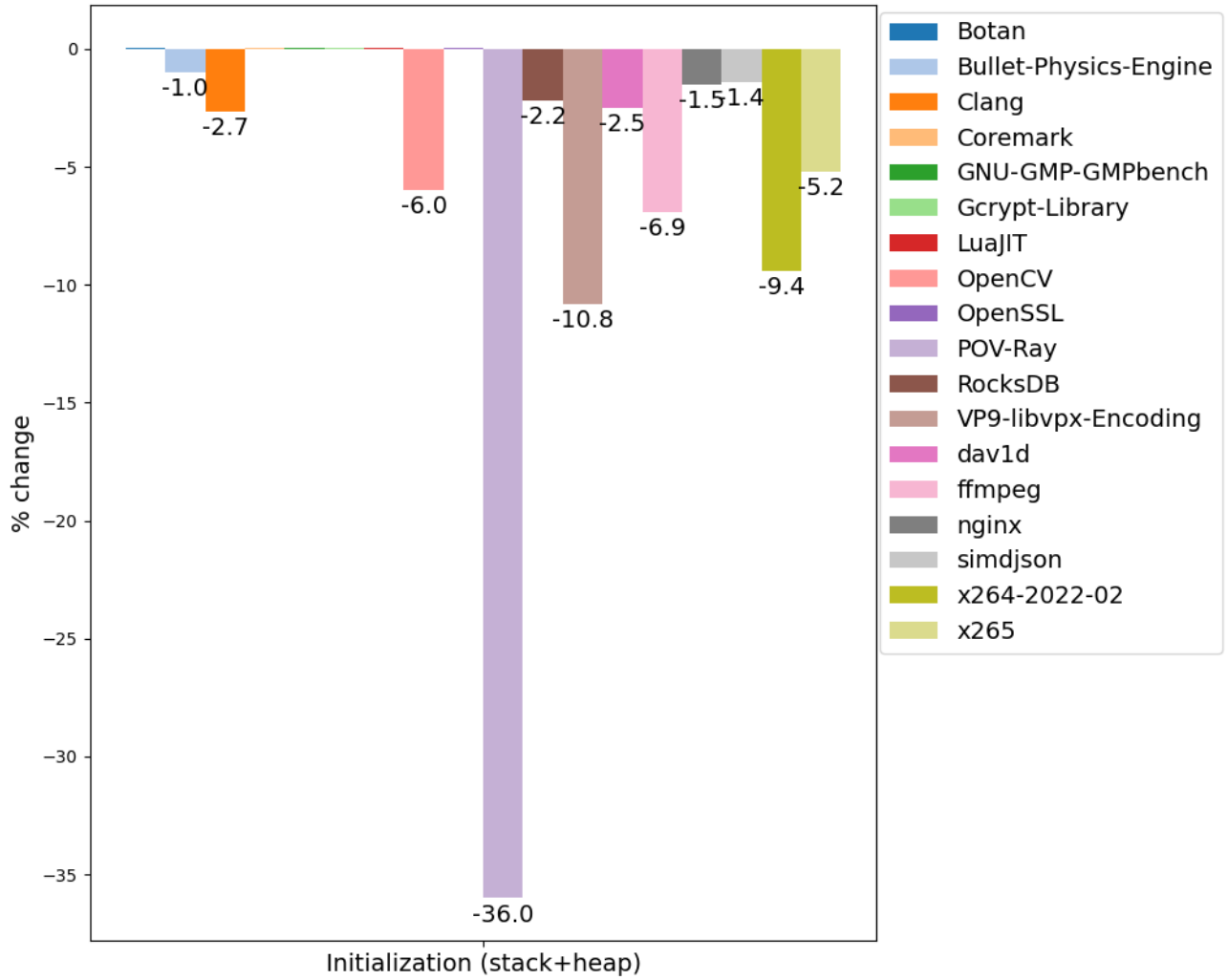
Накладные расходы

- 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 инициализаций стека и кучи (на примере C++)



- Использовалась 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

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

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;
};
```

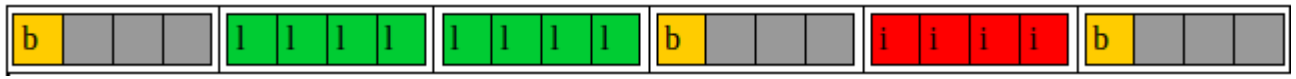


Image source

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
```

```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
```

Image source (Jeffrey Bary)

Что есть в C++

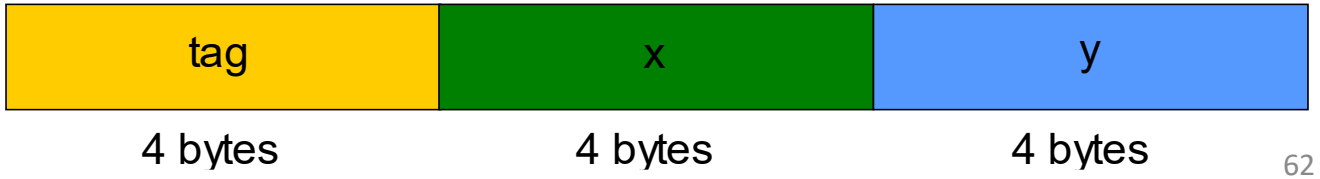
- Многие статические анализаторы предупреждают о неэффективном расположении полей
  - -Wpadded, clang-tidy, PVS-studio, etc.
  - Pahole (на базе debuginfo)
- clang-reorder-fields – LLVM-based инструмент для перестановки полей для оптимизации паддинга
- В теории при использовании Fat LTO компилятор может 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,
    }
}
```

```
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)
- Паддинг в структурах **не может** использоваться для ниш
- Могут использоваться для хранения тега enum-типа (вместо отдельного поля)
  - Т.н. Discriminant Elision

Image source

Примеры ниш

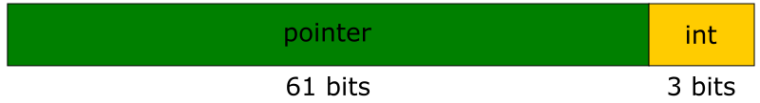
- Самый яркий (и гарантированный языком) пример оптимизации ниш – тип Option<T>, где T – NonNull тип (например &U)

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

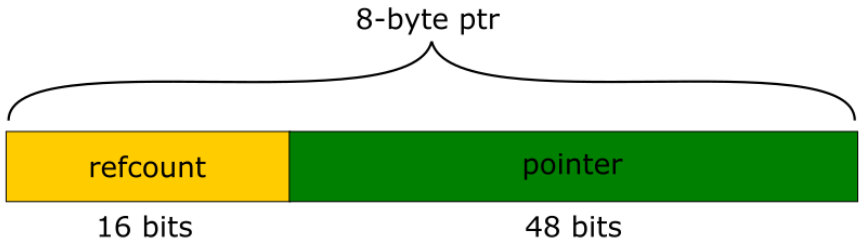
- Одна ниша может использоваться для нескольких тегов:
  - `size_of::<Option<Option<Option<bool>>>>() == 1`
- Аналогичная оптимизация возможна для младших бит ссылки (или указателя)
  - У выровненных указателей несколько младших бит всегда 0
  - Их можно использовать как нишу
  - Пока реализовано только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

Что есть в C++

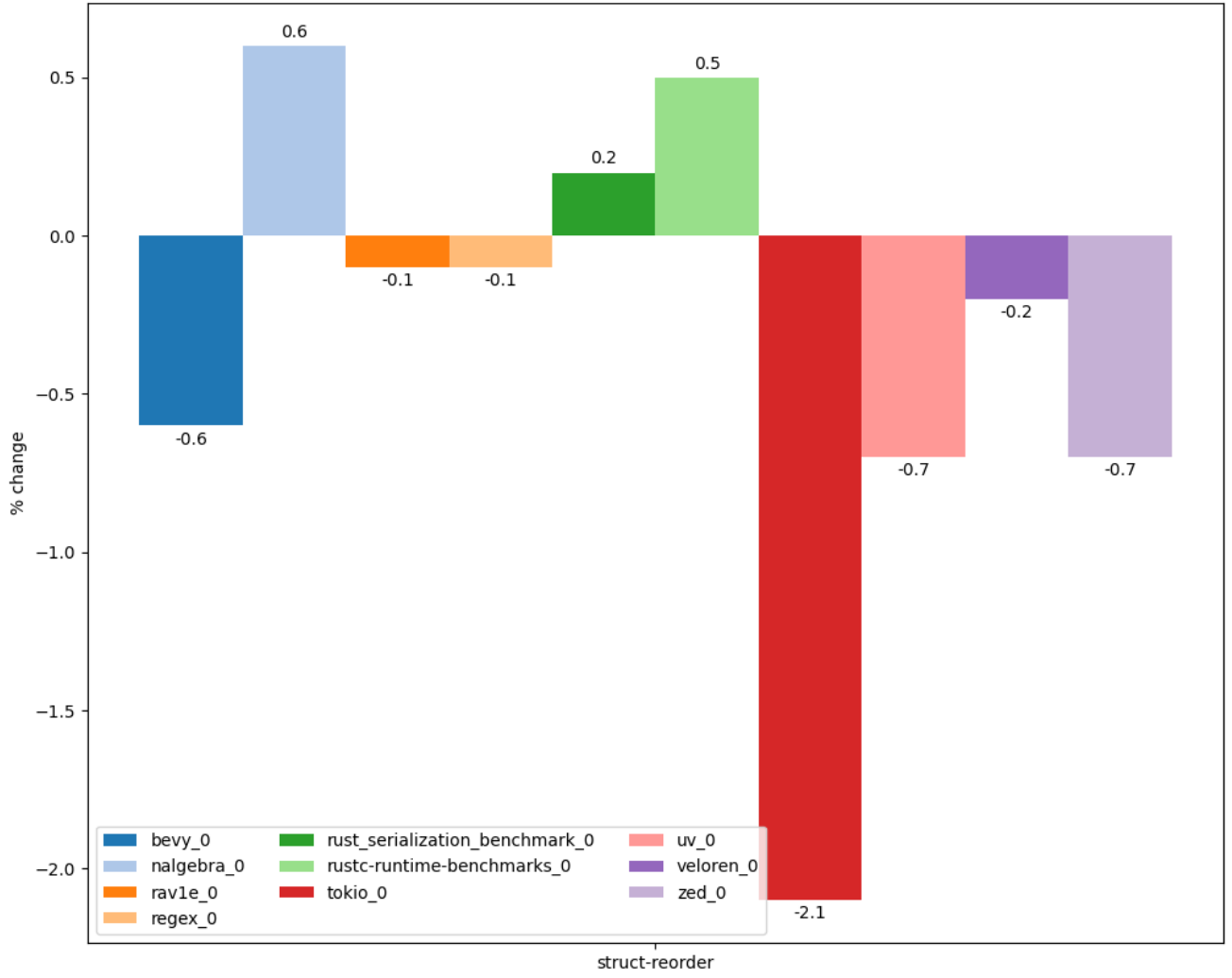
- В std::optional и std::variant нет аналога оптимизаций ниш
  - Можно реализовать аналог std::optional с такой оптимизацией
- llvm::PointerIntPair
  - Использует младшие биты указателя для хранения значений int
  - Аналог -Zreference-niches в Rust



- 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

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:
  - `-ftrapv` появилась в GCC в 2000
  - Работы John Regehr над чекером IOC в 2010
  - Создание UBsan в 2014 (на волне популярности Asan)
    - State-of-the-art решение

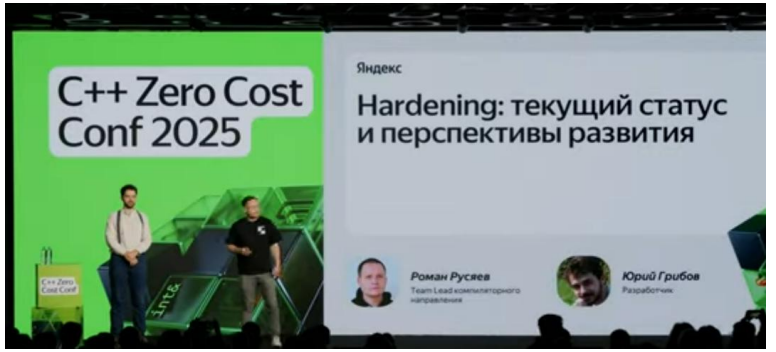


Image source (J.R. Armstrong)

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
- 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

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

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

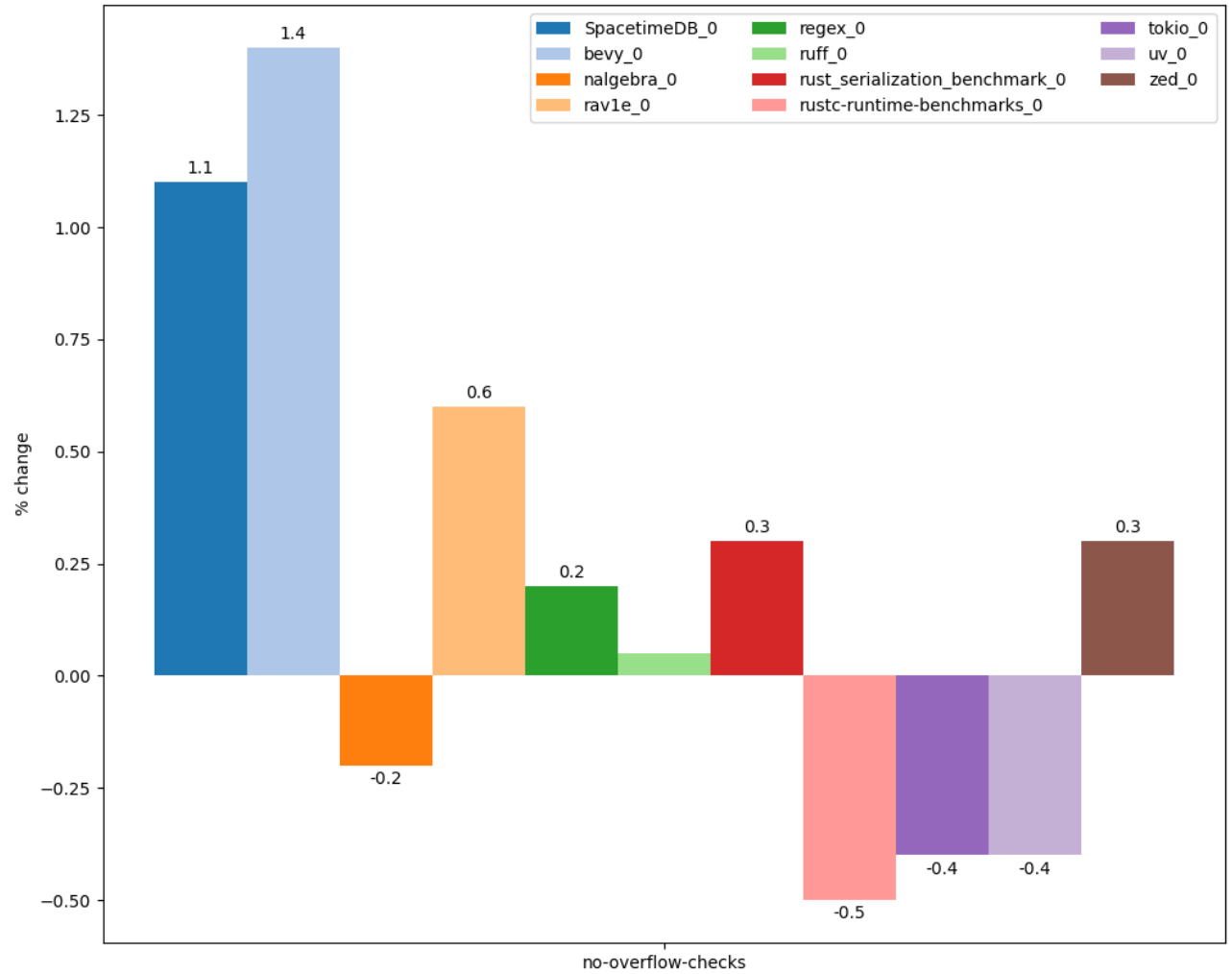
Workarounds

- Специальные 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
```

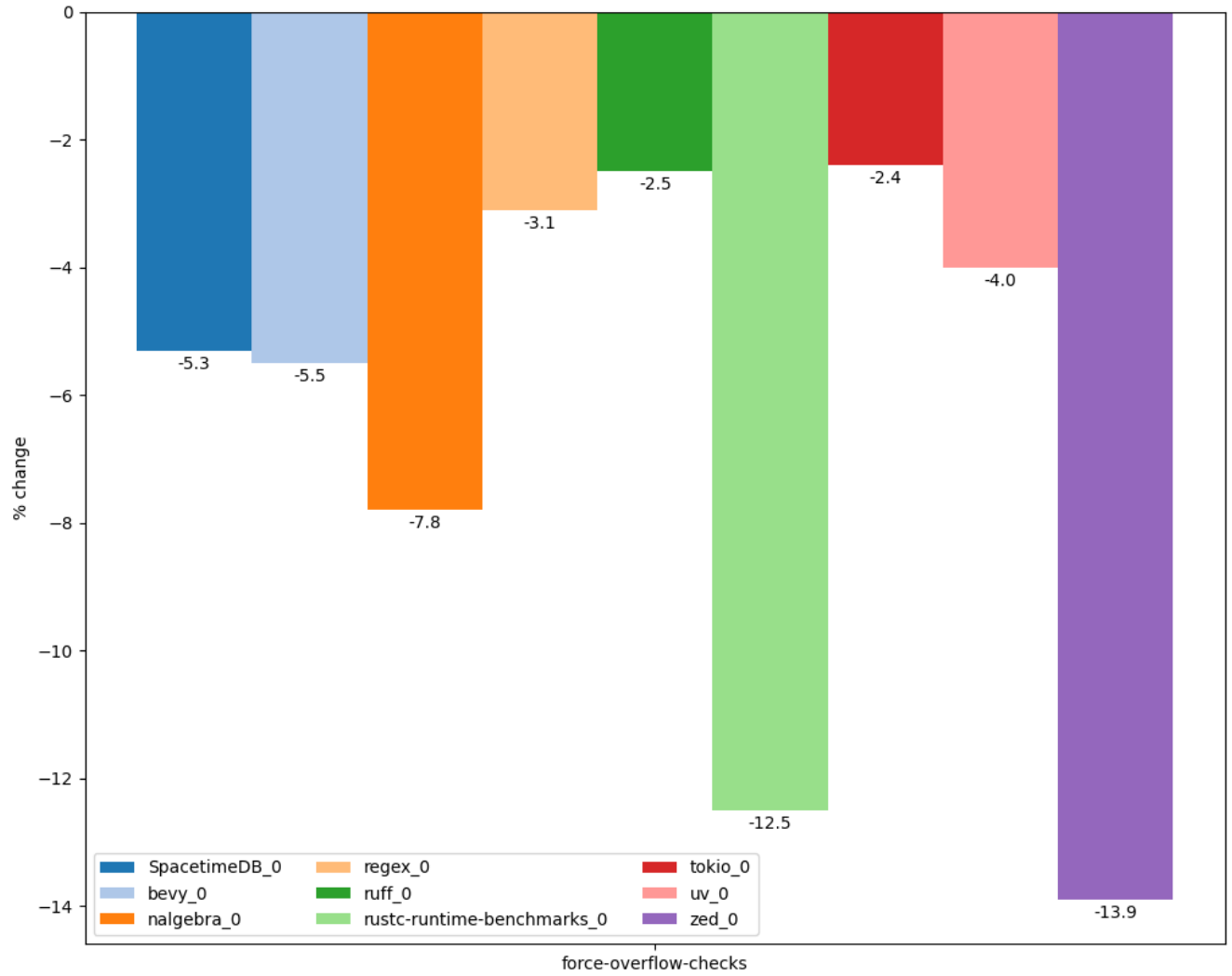
  - (проверки в стандартной библиотеке *не могут* быть отключены)
- Специальные типы стандартной библиотеки:
  - NonZeroU32 для отключения некоторых проверок деления
  - Wrapping<T> для переполнений

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



[Ветка компилятора]

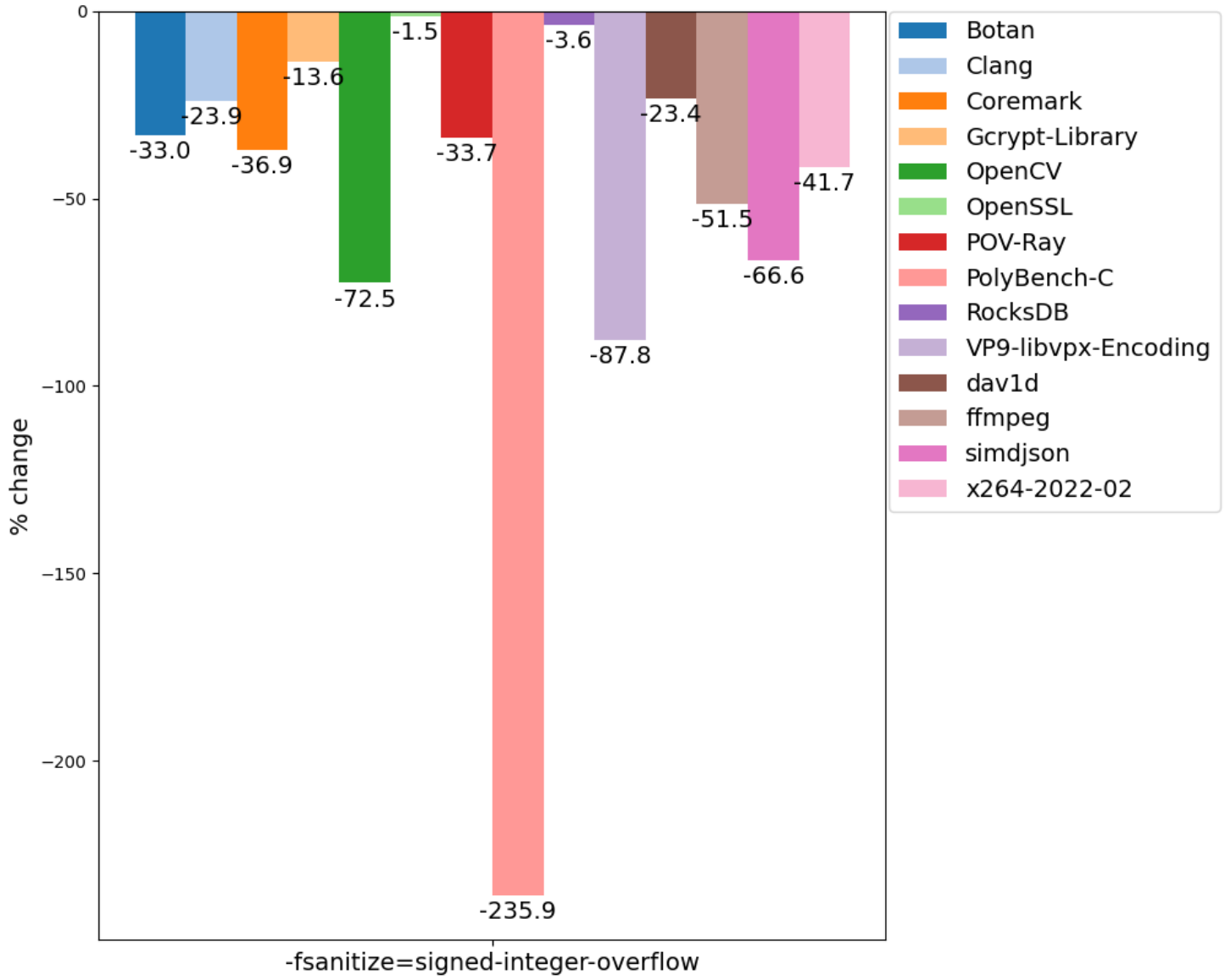
Замедление при включении целочисленных переполнений



- 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 бо*л*ьший оверхед в UBsan для C/C++

Выводы

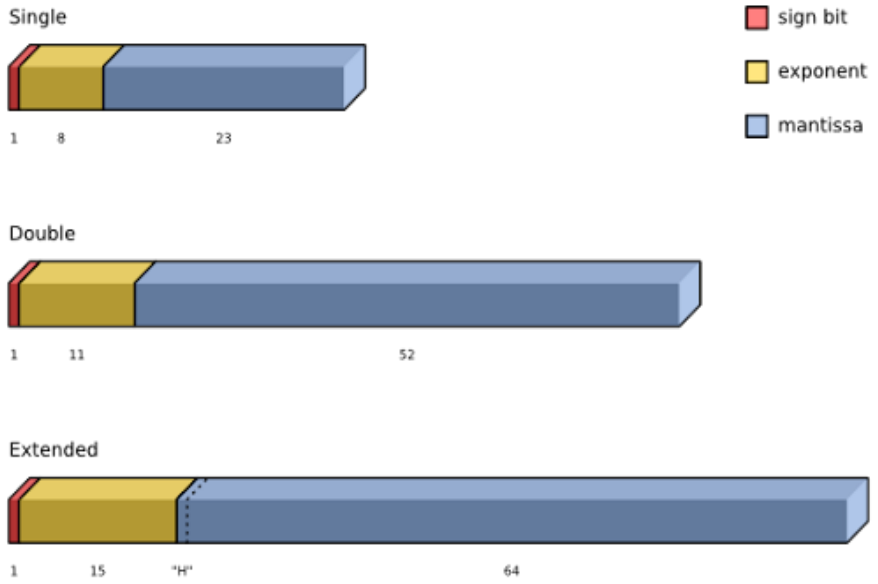
- В Rust целочисленное переполнение не является UB
  - Мешает компилятору проводить некоторые агрессивные оптимизации
- В языке приняты неочевидные решения о дефолтных арифметических проверках
  - Действительно важные ошибки типа приведения с потерями не проверяются, зато проверяется деление на ноль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

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

```
.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

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
- C# разрешает совершать вычисления с *б*ольшей 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

```
.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**

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

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

clang-21 -O2 **-ffinite-math-only**

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

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

clang-21 -O2

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

clang-21 -O2 **-ffinite-math-only -fno-signed-zeros**

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

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

clang-21 -O2 **-ffinite-math-only**

Позиция Rust

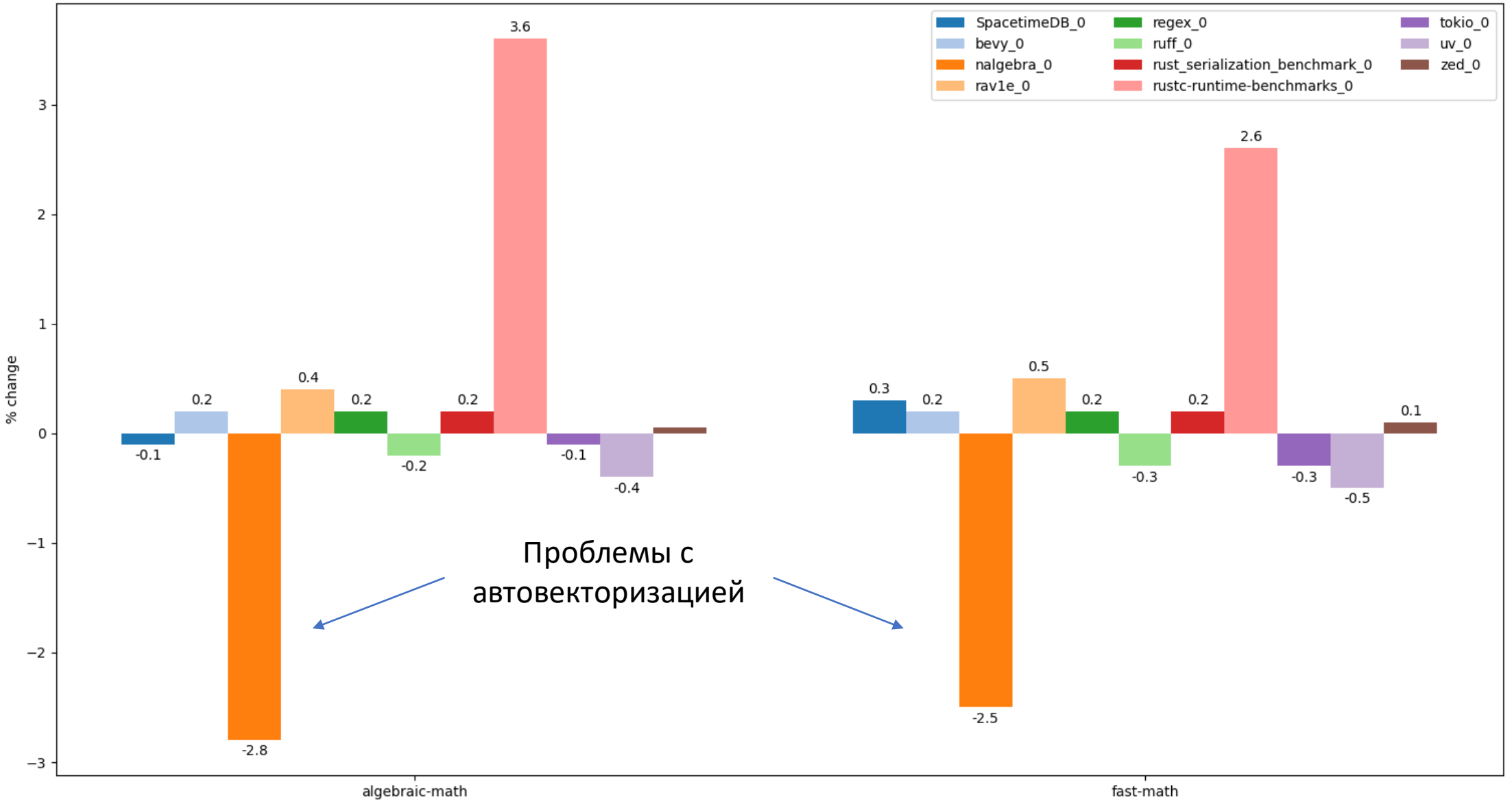
- Rust отказывается от возможности глобального применения fast-math оптимизаций
- Основная мотивация – опасность недостаточно осторожного применения этих оптимизаций, особенно в зависимостях проекта

> “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

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)
  - Глобальными являются только pub-функции

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
  - Межпроцедурная аллокация регистров (IPRA)
- А также
  - Более агрессивный inlining
    - Например, всегда инлайнятся однократно вызываемые локальные функции
  - Улучшенное обнаружение мёртвого кода

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
}
```

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

Цикл в foo был полностью оптимизирован

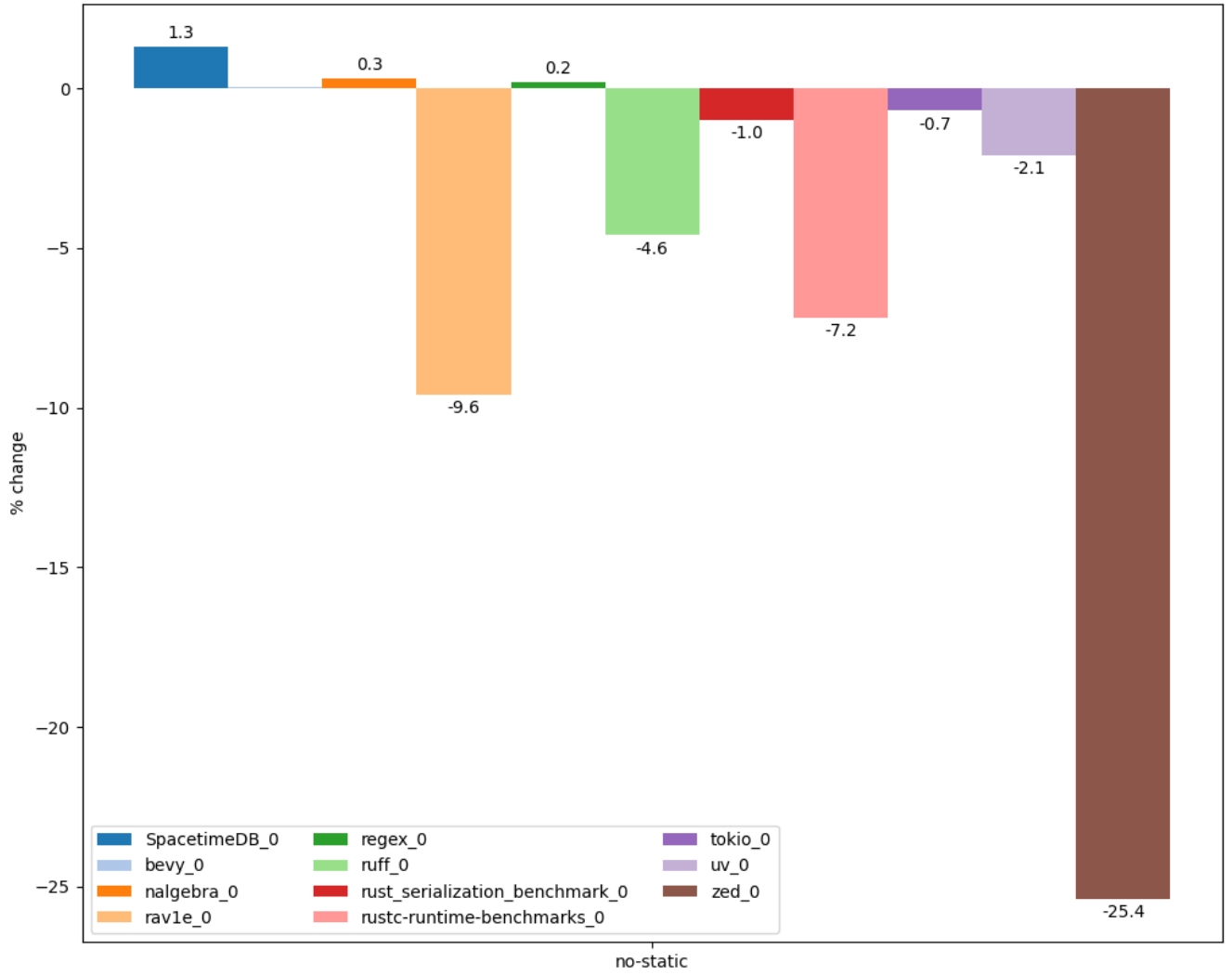
Сравнение с C/C++

- Многие авторитетные руководства рекомендуют локализовывать функции:
  - Google C++ Style Guide
  - LLVM Coding Standard (Restrict Visibility)
  - C++ Core Guidelines
  - Chromium C++ Style Guide

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



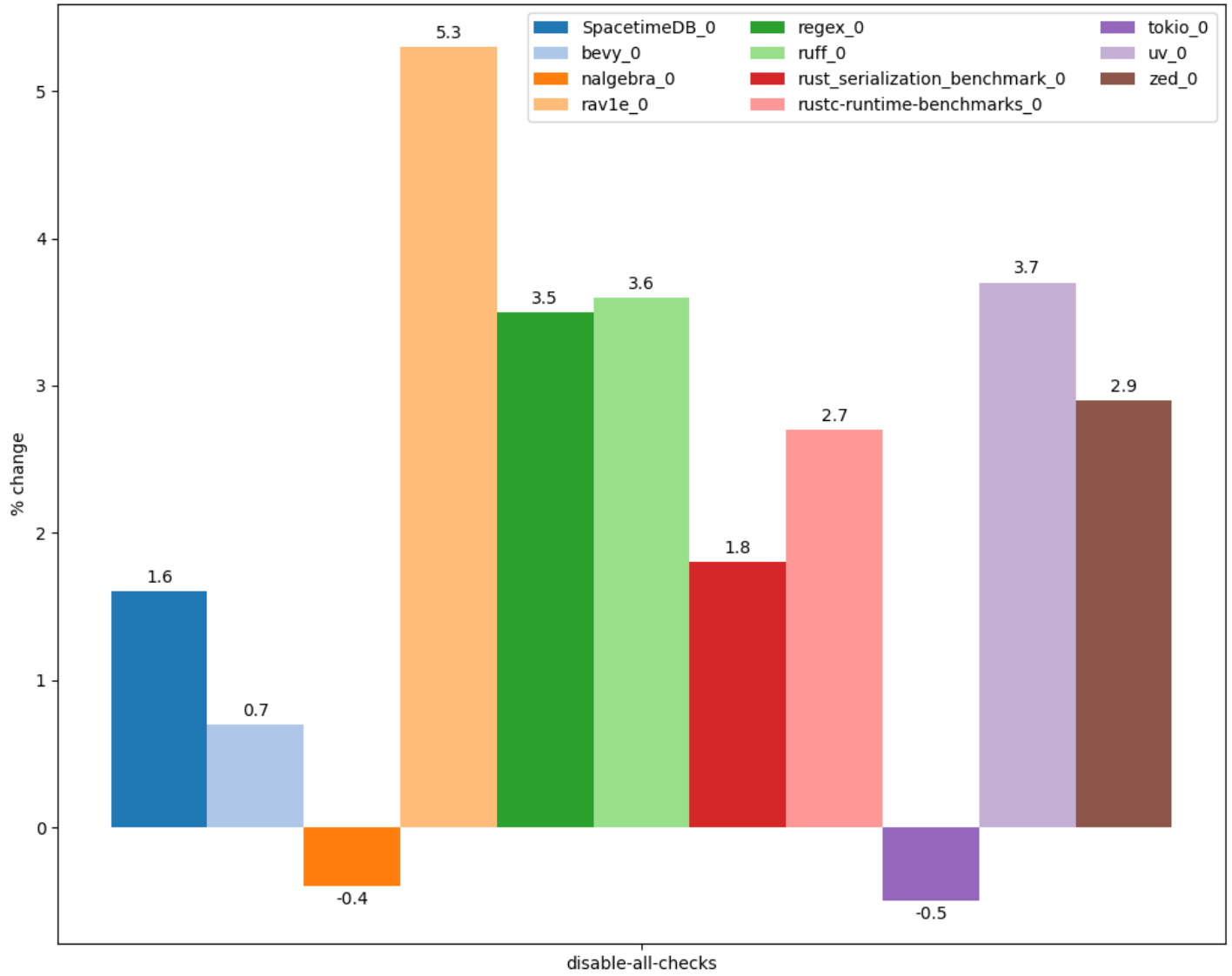
[Патч для компилятора]

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

Выводы

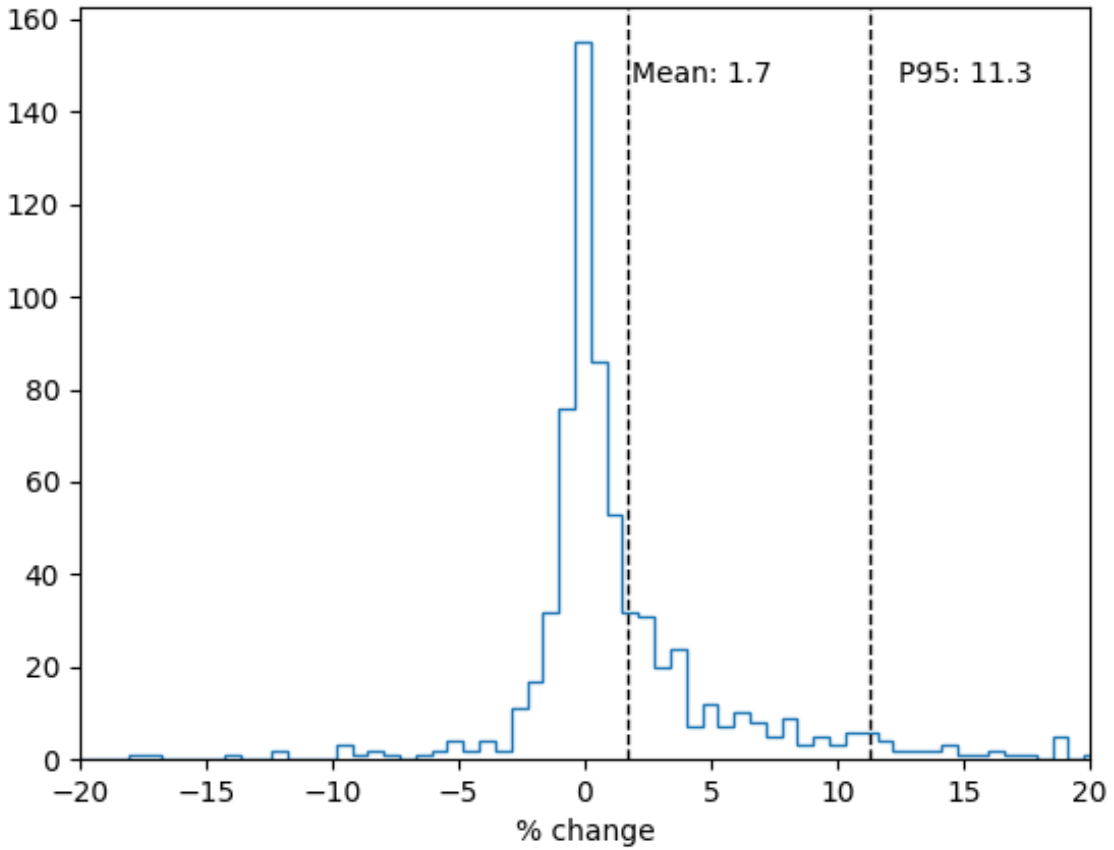
Ускорение при отключении (почти) всех проверок Rust



- 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]

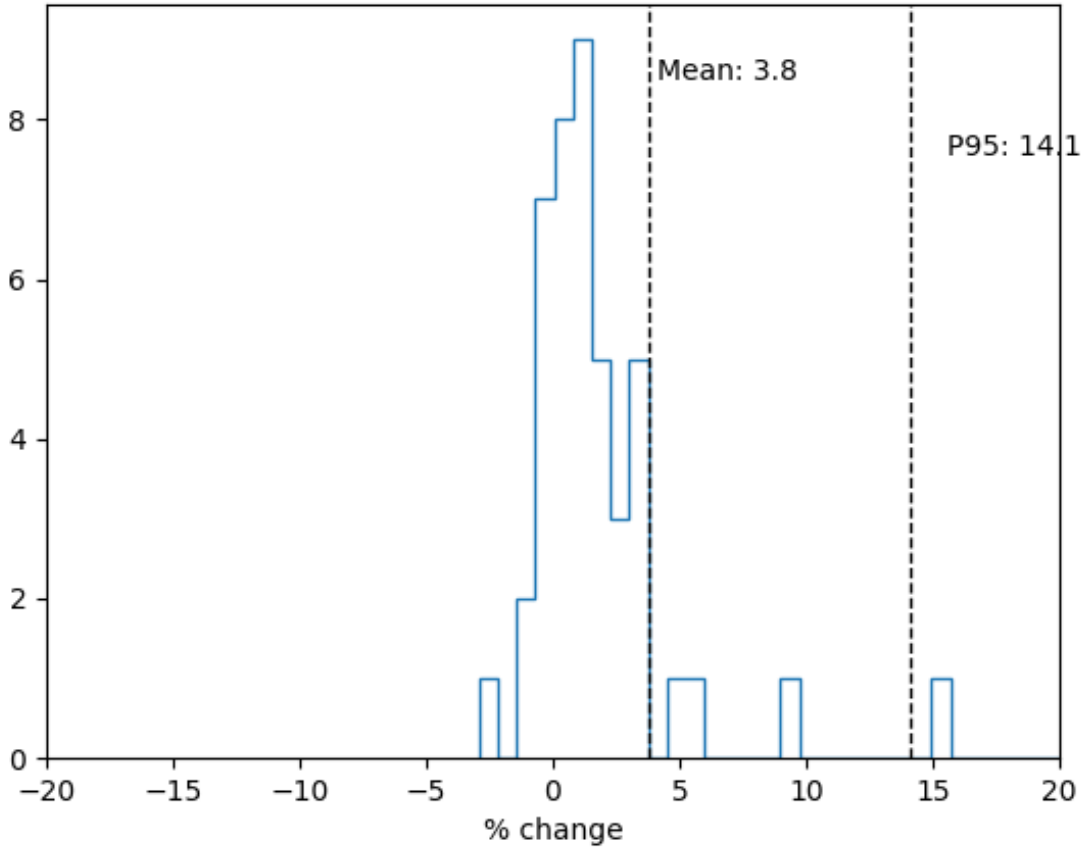
Распределение замедлений в бенчмарках



- 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++
- Включенные защиты:
  - -D_FORTIFY_SOURCE=3
  - -fsanitize=bounds,object-size
  - Hardened STL
  - (инициализацию не включали чтобы соответствовать Rust)

[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 инициализация, 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

Image source (3dman_eu)

О чём мы не успели рассказать …

- Отсутствие inplace-инициализаций
  - Доступна в полной презентации
- Особенности стандартной библиотеки
  - Доступна в полной презентации
- 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

Image source

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

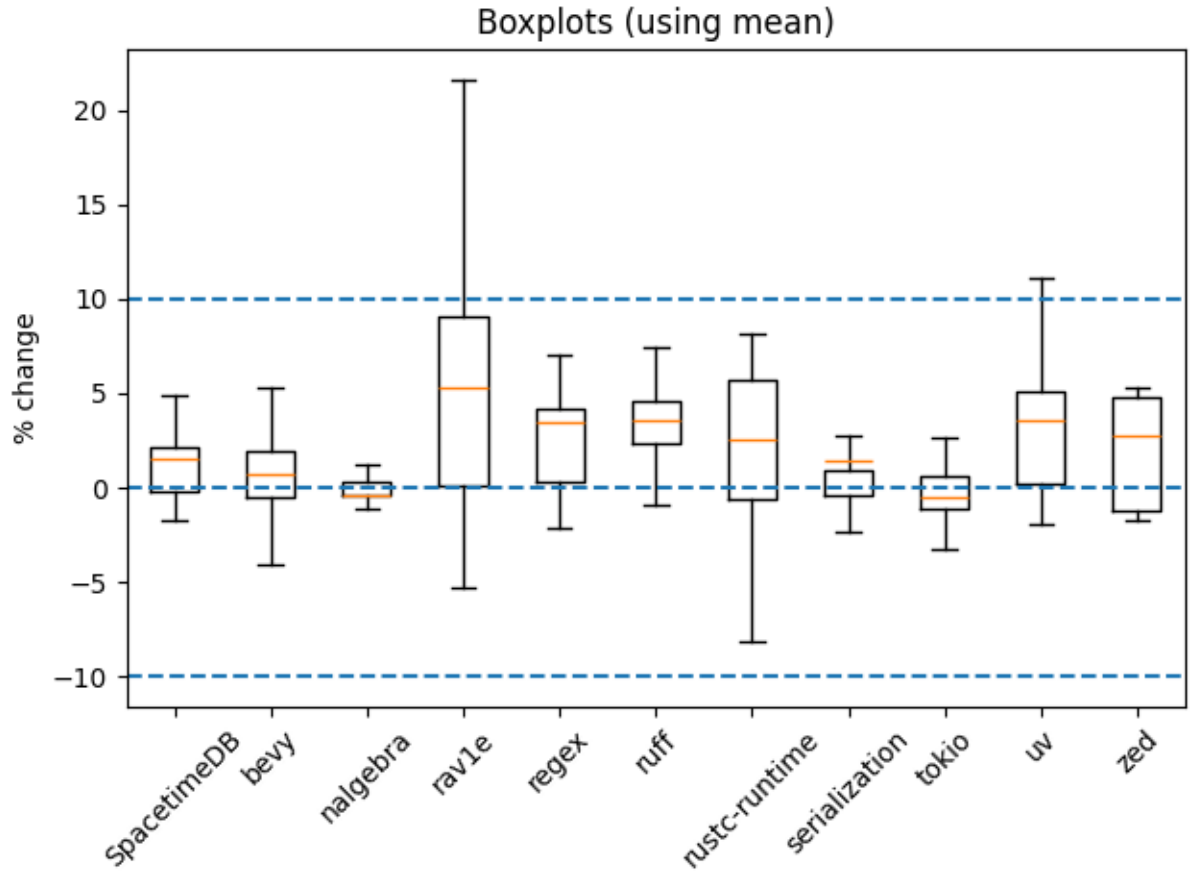
Приложения

Rust primer

```
struct Ok {};
struct Warn { uint32_t code; };

using Event = std::variant<Good, Warn>;

int check(std::span<const Event> events,
          std::optional<double> max) {

  int result = 0;
  const auto mx = uint32_t(max.value_or(1.0));

  for (const auto &e : events) {
    if (const auto *w = std::get_if<Warn>(&e)) {
      if (w->code > mx) {
        ++result;
      }
    }
  }


  return result;
}
```

```
enum Event { Ok, Warn(u32), }

fn check(events: &[Event],
         max: Option<f64>) -> i32 {

    let mut result = 0;
    let mx = max.unwrap_or(1.0) as u32;

    for e in events {
        match e {
            Event::Warn(c) => if *c > mx {
                result += 1;
            },
            _ => (),
        }
    }

    result
}
```

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]

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