

Санкт-Петербургский государственный университет

Михайлов Илья Игоревич

Выпускная квалификационная работа

Динамическая бинарная трансляция в RISC-V с помощью Instrew

Уровень образования: бакалавриат

Направление *02.03.03 «Математическое обеспечение и администрирование
информационных систем»*

Основная образовательная программа *СВ.5162.2021 «Технологии программирования»*

Научный руководитель:
доцент кафедры системного программирования, к. ф.-м. н., Луцив Д. В.

Консультант:
инженер-исследователь, Лаборатория технологий программирования инфраструктурных
решений СПбГУ, Кутуев В. А.

Рецензент:
исполнительный директор, ПАО «Сбербанк России», Сергеев Ф. В.

Санкт-Петербург
2025

Saint Petersburg State University

Ilya Mikhaylov

Bachelor's Thesis

Dynamic binary translation for RISC-V with Instrew framework

Education level: bachelor

Speciality *02.03.03 "Software and Administration of Information Systems"*

Programme *CB.5162.2021 "Programming Technologies"*

Scientific supervisor:
C.Sc., docent D. V. Luciv

Consultant:
Research Engineer at "Laboratory of Programming Technologies for Infrastructure Solutions at
St. Petersburg State University", V. A. Kutuev

Reviewer:
Principal Engineer "Sberbank", F. V. Sergeev

Saint Petersburg
2025

Оглавление

Введение	5
1. Обзор	7
1.1. Динамические бинарные трансляторы	7
1.2. Динамические бинарные трансляторы с поднятием в LLVM IR	8
1.3. Выводы	9
2. Постановка задачи	10
3. Обзор Instrew	11
3.1. Архитектура	11
4. Реализация поддержки клиента Instrew	13
4.1. Компиляция	13
4.2. Исправление ошибок времени исполнения	14
5. Тестирование Rellume	18
5.1. Обновление версии LLVM до 20	18
5.2. Результат	18
6. Тестирование Instrew	20
6.1. Unit-тесты	20
6.2. Тестирование Instrew на Coreutils	21
7. Замеры и сравнение производительности	23
7.1. Обзор бенчмарков	23
7.2. Условия эксперимента	24
7.3. Замеры производительности на LINPACK	25
7.4. Замеры пропускной способности на xhHash	28
8. Реализация CI	29
8.1. Self-hosted runner	29
8.2. Эмуляция с помощью QEMU-user	29

8.3. Результат	30
Заключение	31
Список литературы	33

Введение

Бинарная трансляция — форма трансляции, где на вход подается последовательность инструкций одной архитектуры процессора (так называемая **исходная архитектура** или **гостевая архитектура**), а на выход получаем инструкции другой архитектуры (**целевая архитектура** или **хост архитектура**). Стоит отметить, что гостевая и целевая архитектуры могут совпадать. Бинарную трансляцию можно классифицировать по времени выполнения на **статическую** и **динамическую**. При статической трансляции выполнение программы происходит после трансляции, а при динамической — во время трансляции. Она может осуществляться с целью эмулировать какой-либо набор инструкций, чтобы запускать программы, скомпилированные не под назначенную архитектуру, либо же скомпилированные под другое расширение набора команд. Примерами эмуляторов являются: QEMU-user¹, Boxc64², Rosetta 2. Также исходные инструкции можно транслировать в инструкции той же архитектуры для профилирования и отладки транслируемой программы; для выполнения бинарного инструментария, а именно для проведения оптимизаций и/или внесения исправлений, например, когда рекомпиляция/декомпиляция осложнена. Примерами таких инструментов являются Valgrind³, Pin, Dynamo.

Бинарные трансляторы могут использовать поднятие (лифтинг) инструкций, то есть трансляцию в некоторое более высокоуровневое, чем двоичный код, промежуточное представление. QEMU использует TCG IR, Valgrind использует VEX IR. Эти промежуточные представления нацелены скорее на точность соответствия исходных и целевых инструкций, чем на генерацию производительного двоичного кода. В качестве альтернативного промежуточного представления для лифтинга инструкций при динамической бинарной трансляции перспективным выглядит использование LLVM IR. Фреймворк LLVM обладает развитой экосистемой с JIT-компилятором и обширным набором оптимизаций

¹<https://www.qemu.org/> (дата обращения 26.05.2025)

²<https://box86.org/> (дата обращения 26.05.2025)

³<https://valgrind.org/> (дата обращения 26.05.2025)

для генерации производительного двоичного кода, что может заметно повысить производительность динамической бинарной трансляции. Например, динамический бинарный транслятор (далее ДБТ) Instrew⁴, применяющий поднятие кода до LLVM IR, на архитектурах x86_64 и AArch64 показал⁵ значительно большую производительность, чем QEMU-user и HQEMU⁶.

RISC-V⁷ — молодая и активно развивающаяся открытая процессорная архитектура. На текущий момент в свободной продаже доступны различные процессоры, имеющие RISC-V ядра, например, Allwinner D1⁸, SpacemiT K1⁹ и другие. Однако все они имеют разный набор расширений, что не позволяет запускать код скомпилированный под один процессор на другом, также не всё программное обеспечение, написанное под архитектуры x86_64, ARM, портировано на RISC-V. Наличие производительного эмулятора может помочь в развитии экосистемы ПО на RISC-V, поскольку позволит пользователям использовать привычные программы, которые не доступны для их аппаратуры. Таким образом, было принято решение портировать или разработать ДБТ с поднятием в LLVM IR, который поддерживает RISC-V как хост-архитектуру.

⁴<https://github.com/aengelke/instrew> (дата обращения 26.05.2025)

⁵<https://mediatum.ub.tum.de/doc/1614897/1614897.pdf> (дата обращения 26.05.2025)

⁶<https://dl.acm.org/doi/10.1145/2259016.2259030> (дата обращения 26.05.2025)

⁷<https://riscv.org/> (дата обращения 26.05.2025)

⁸<https://d1.docs.aw-ol.com/en/> (дата обращения 26.05.2025)

⁹<https://www.spacemit.com/key-stone-k1/> (дата обращения 26.05.2025)

1. Обзор

Для обзора предлагается рассмотреть динамические бинарные трансляторы с поднятием инструкций в LLVM IR, а также современные ДБТ для эмуляции и инструментирования с целью сбора функциональных требований к ДБТ с поднятием в LLVM IR и поддержкой архитектуры RISC-V, как хоста. В данной работе не будет рассмотрен ДБТ Rosetta 2, так как не имеет поддержки RISC-V и не является программным обеспечением с открытым исходным кодом, хоть и существуют попытки реверс-инжиниринга¹⁰.

1.1. Динамические бинарные трансляторы

1.1.1. Эмуляторы

QEMU-user — динамический бинарный транслятор с открытым исходным кодом, предназначенный для эмуляции обширного числа архитектур, который также поддерживает аппаратную виртуализацию (QEMU-system). В качестве промежуточного представления используется TCG Intermediate Representation, в которое поднимаются базовые блоки транслируемой программы[1]. Транслятор, исходный код и транслируемый код находятся в одном адресном пространстве во время исполнения[3]. Поддерживает RISC-V как хост-архитектуру и как целевую архитектуру. Поддерживает многопоточность.

Box64 — динамический бинарный транслятор с открытым исходным кодом, позволяющий запускать x86/x86_64 программы на архитектурах ARM и RISC-V¹¹. Использует JIT-компилятор dynarec, который транслирует базовые блоки в хост архитектуру¹². Замеры с помощью nbench показали, что box64 работает быстрее QEMU¹³.

¹⁰<https://ftri.github.io/ProjectChampion/> (дата обращения 26.05.2025)

¹¹<https://box86.org/2024/08/box64-and-risc-v-in-2024/> (дата обращения 26.05.2025)

¹²<https://box86.org/2024/07/revisiting-the-dynarec/> (дата обращения 26.05.2025)

¹³<https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10077985> (дата обращения 26.05.2025)

1.1.2. Инструменты для бинарного анализа

Valgrind — динамический транслятор для бинарной инструментации с открытым исходным кодом и большим числом плагинов/динамических анализаторов, такие как: Memcheck, Callgrind, Cachegrind. В качестве промежуточного представления используется VEX IR, в которое поднимаются суперблоки транслируемой программы, где суперблок — последовательность инструкций, в котором может быть несколько точек выхода[6]. Транслятор, плагин и транслируемая программа находятся в одном адресном пространстве. Поддерживает архитектуру RISC-V.

1.2. Динамические бинарные трансляторы с поднятием в LLVM IR

1.2.1. Instrew

Instrew — динамический бинарный транслятор с поднятием инструкций в LLVM IR, открытым исходным кодом и с архитектурой клиент-сервер. Поддерживает x86_64 и AArch64, как хост архитектуру процессора, и x86_64, AArch64, RISC-V, как целевую архитектуру. С помощью бинарного лифтера Rellume поднимает функции в LLVM IR. Может использоваться как для эмуляции, так и для бинарного инструментирования[3]. Не поддерживает многопоточность.

1.2.2. HQEMU

HQEMU[5] — многопоточный динамический бинарный транслятор, использующий QEMU TCG для трансляции инструкций в TCG IR и тулчейн LLVM для оптимизаций. Транслятор и оптимизатор могут запускаться в разных потоках с целью снизить затраты на трансляцию и оптимизацию соответственно.

1.2.3. DBILL

DBILL[2] — инструмент для динамического бинарного анализа, использующий TCG IR и LLVM IR, как промежуточные представления, основан на HQEMU. Благодаря фронтэнду QEMU и тулчейну LLVM поддерживается множество хост и целевых архитектур. Из-за двух промежуточных представлений и поднятия в TCG только базовых блоков, есть проблемы с производительностью. Исходный код не был найден.

1.3. Выводы

По результатам замеров[4], Instrew оказался более производительным, чем QEMU и Valgrind. Хотя Box64 тоже оказывается производительнее чем QEMU, но не имеет API для производства динамического бинарного анализа, что в сравнении с Instrew делает его менее универсальным. Далее данная работа будет сосредоточена именно на ДБТ Instrew с поднятием в LLVM IR и добавлении поддержки архитектуры RISC-V для такого транслятора.

2. Постановка задачи

Целью работы является добавление поддержки архитектуры RISC-V для ДБТ Instrew. Для ее выполнения были поставлены следующие задачи:

1. провести обзор архитектуры Instrew;
2. портировать ДБТ Instrew на RISC-V:
 - исправить ошибки компиляции;
 - исправить ошибки, выявленные при внутреннем интеграционном тестировании:
 - динамического транслятора Instrew;
 - лифтера Rellume;
3. выполнить тестирование на библиотеке стандартных утилит Coreutils и исправить выявленные ошибки;
4. добавить в CI поддержку сборки и тестирования на RISC-V;
5. выполнить замеры производительности на xxHash и LAPACK и сравнить с QEMU-user и Vex64.

3. Обзор Instrew

3.1. Архитектура

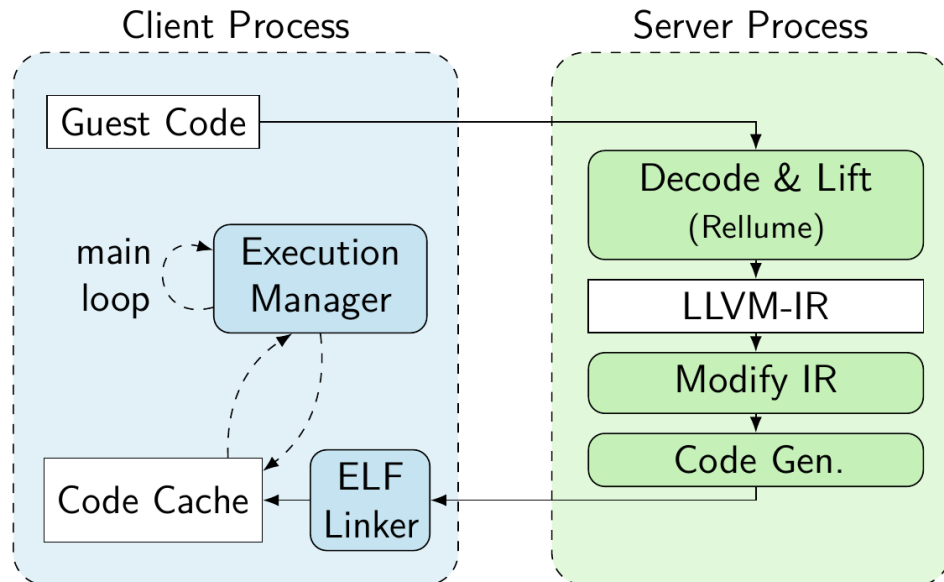


Рис. 1: Архитектура Instrew

Instrew реализует клиент-сервер архитектуру (рис. 1¹⁴):

- клиент (также instrew-client), написанный на языке C и ассемблере, и выполняющий следующие функции:
 - загрузка целевой программы в адресное пространство;
 - эмуляция состояния процессора целевой программы;
 - отправка функций целевой программы серверу;
 - динамическая линковка ELF объектов, полученных от сервера;
 - загрузка, разрешение символов и проведение релокаций для полученного ELF объекта;
 - эмуляция системных вызовов;
 - вызов функции, полученной после линковки ELF объекта;

¹⁴<https://mediatum.ub.tum.de/doc/1614897/1614897.pdf> (дата обращения 26.05.2025)

- кэширование;
- сервер, написанный на языке C++, выполняющий функции:
 - декодирование и поднятие функции целевой программы в LLVM IR с помощью Rellume;
 - оптимизация LLVM IR;
 - кодогенерация;
 - создание ELF объекта;

В результате изучения документации проекта и его исходного кода были выявлены следующие модули, требующие изменения для портирования:

- модуль ELF Linker в клиентской части;
- модули Rellume и Code Gen в серверной части.

4. Реализация поддержки клиента Instrew

4.1. Компиляция

Для начала необходимо скомпилировать Instrew, а именно:

- определить платформенно-специфические макросы и константы;
- написать точку входа в программу на ассемблере RISC-V (листинг 1);
- написать реализацию `syscallN` (системный вызов с `N` аргументами), которая используется эмуляции системных вызовов. В листинге 2 представлена реализация для эмуляции системного вызова с двумя аргументами.

Из трудностей можно выделить, что при написании точки входа в программу был просмотрен код наиболее распространенных реализаций `libc` (в `glibc` это файл `start.S`, в `musl` — `crt_arch.S`), где была обнаружена плохо задокументированная псевдоинструкция `lla`¹⁵.

Листинг 1: Точка входа в программу

```
ASM_BLOCK(  
    .text;  
    .type _start, %function;  
    .weak _DYNAMIC;  
    .hidden _DYNAMIC;  
    .globl _start;  
_start:  
    mv fp, x0;  
    mv a0, sp;  
    lla a1, _DYNAMIC;  
    andi sp, sp, 0xffffffffffffff0;  
    jal __start_main;  
);
```

Листинг 2: Системный вызов

```
static size_t syscall12(int n, size_t a, size_t b)  
{
```

¹⁵<https://reviews.llvm.org/D49661> (дата обращения 26.05.2025)

```

    register size_t a7 __asm__("a7") = n;
    register size_t a0 __asm__("a0") = a;
    register size_t a1 __asm__("a1") = b;
    __asm__ volatile ("ecall\n\t" : "+r"(a0) : "r"(a7), "0"(a0), "r"(a1) : "
        memory");
    return a0;
}

```

4.2. Исправление ошибок времени исполнения

После успешной компиляции пробуем запустить Instrew. В данном разделе хочется рассказать о трудностях отладки клиента Instrew.

4.2.1. Специфические флаги сборки

Запуск клиента Instrew без аргументов приводил к ошибке сегментации. Среди инструментов, доступных на RISC-V и помогающих обнаруживать источник ошибок работы с памятью, можно выделить динамический анализатор Address Sanitizer (далее — ASan). Он включается при помощи опций компилятора, который инструментирует код и заменяет вызовы аллокации памяти (`malloc`) собственной реализацией. Однако использование ASan с Instrew невозможно, так как Instrew для выделения и освобождения памяти использует собственные функции: `mem_alloc_data` и `mem_alloc_code`. Таким образом отладка производилась только при помощи GNU Debugger (GDB). В результате отладки с помощью GDB и просмотра карты памяти процесса (memory mappings), было обнаружено, что семантика флага **-static-pie** при компиляции с помощью gcc-13 отличается на платформах x86_64 и riscv64, а именно:

- подключается динамический компоновщик ld-linux.so;
- но клиент Instrew сам является динамическим компоновщиком и сам обрабатывает свои релокации.

Из-за этого происходит ошибка сегментации, так как клиент Instrew пытается писать в зарезервированную ld-linux.so память. Эта версия

компилятора установлена по умолчанию в дистрибутивах Linux, используемых на платах с процессором RISC-V, на которых производилась сборка. Эта проблема была решена для gcc-13, использованием флагов `-Wl,-static, -Wl,-pie, -Wl,--no-dynamic-linker, -Wl,-z,text`. Также было проверено, что новая версия компилятора, gcc-14, корректно обрабатывает данный флаг. В результате в систему сборки были добавлены проверки, позволяющие избежать данной проблемы.

4.2.2. Связь атрибутов и релокаций

Далее также при запуске клиента Instrew без аргументов программа завершается с кодом возврата (exit code) **-ENOEXEC**. Оказалось, что это происходит при обработке одной из релокаций: функция **memset** помечена атрибутом **weak**, из-за чего в итоге меняется тип релокации. Но эту релокацию Instrew не может обработать.

Данная проблема была решена удалением данного атрибута на этапе препроцессинга при сборке под RISC-V.

4.2.3. Отладка программ, использующих `fork()` и `fexecve()`

После успешного запуска клиента Instrew, пробуем запустить сервер Instrew. Чтобы можно было пройти по программе в GDB, необходимо понимать как вызывается `instrew-client`:

1. делается `fork()` процесса, где дочерний процесс является клиентом Instrew;
2. в дочернем процессе вызывается `fexecve`, который запускает программу `instrew-client`.

Для того чтобы можно было отладить клиент Instrew, используем `follow-fork-mode child`. Однако не удалось понять, как отлаживать программу запущенную через `fexecve` в GDB, в связи с чем используем скрытый флаг `--stub` в Instrew, чтобы запускать клиент через `execve`, указав при этом путь к файлу клиента. Далее при запуске GDB после `fork()` устанавливаем `follow-exec-mode new`.

4.2.4. Релокации и PLT-трамплины

При запуске unit-тестов Instrew обнаруживаем ошибку «Unhandled relocation». Нужно добавить необходимые релокации для успешного прохождения тестов.

В документации RISC-V psABI¹⁶ описаны вычисления релокаций по ее типу. Также примеры реализации релокаций можно найти в репозитории ядра Linux¹⁷. Пример добавленной релокации представлен в листинге 3.

Листинг 3: Релокация

```
uint64_t syma = (uintptr_t) sym + patch_data->addend;
uint64_t pc = patch_data->patch_addr;
int64_t prel_syma = syma - (int64_t) pc;

case R_RISCV_32_PCREL:
    if (!rtld_elf_signed_range(prel_syma, 32, "R_RISCV_32_PCREL"))
        return -EINVAL;
    rtld_blend(tgt, 0xffffffff, prel_syma);
    break;
```

Для корректного разрешения символов нужно добавить PLT-трамплины. Они используются для поиска нужного символа в PLT (Procedure linkage table). Их реализация специфична для RISC-V ABI, так как обычно они являются набором машинных команд и встраиваются во время выполнения программы. Реализация PLT-трамплина приведена в листинге 4.

Листинг 4: PLT-трамплин

```
*((uint32_t*) code_ptr+0) = 0x00000297 | ((offset + 0x800) & 0xffff000);
// auipc, t0, offset[31:12]
*((uint32_t*) code_ptr+1) = 0x0002b283 | (offset << 20);
// ld t0, offset[11:0](t0)
*((uint32_t*) code_ptr+2) = 0x00028067;
// jalr, x0, 0(t0)
*((uint32_t*) code_ptr+3) = 0x00000013;
// nop
```

¹⁶<https://github.com/riscv-non-isa/riscv-elf-psabi-doc/blob/master/riscv-elf.adoc#relocations> (дата обращения 26.05.2025)

¹⁷<https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/riscv/kernel/module.c> (дата обращения 26.05.2025)

4.2.5. PLT

При тестировании Instrew на Coreutils было обнаружены проблемы, с разрешением некоторых символов из `libgcc_s.so` и `libatomic.so` для RISC-V хоста. Добавлены дополнительные символы в PLT, а также `libatomic.so` в качестве зависимости к `instrew-client`. Изменения в PLT представлены в листинге 5.

Листинг 5: PLT

```
#if defined(__riscv)
PLT_ENTRY("__muldi3", __muldi3) // libgcc
PLT_ENTRY("__multi3", __multi3) // libgcc
PLT_ENTRY("__umoddi3", __umoddi3) // libgcc
PLT_ENTRY("__udivdi3", __udivdi3) // libgcc
PLT_ENTRY("__atomic_fetch_add_4", __atomic_fetch_add_4) // libatomic
PLT_ENTRY("__atomic_exchange_4", __atomic_exchange_4) // libatomic
#endif //defined(__riscv)
```

5. Тестирование Rellume

Rellume написан на C++ с использованием LLVM и не содержит ассемблерных вставок, как клиент Instrew. Благодаря этому для работы Rellume на RISC-V не потребовалось вносить изменений в его исходный код. Однако ни один из его unit-тестов не проходит успешно на платформе RISC-V. В процессе отладки было выявлено, что в `tests/test_driver.cc` для тестирования поднятых инструкций, используется динамический компоновщик LLVM. При его вызове, в `llvm::RuntimeDyldELF` происходит ошибка сегментации. В связи с этим не можем проверить корректность работы Rellume на RISC-V.

5.1. Обновление версии LLVM до 20

Чтобы решить проблему с недостаточной поддержкой RISC-V в динамическом компоновщике LLVM, LLVM был обновлён до 20-ой версии, последней на момент 26 мая 2025 г.. Был проведен поиск `llvm-20-dev` для RISC-V по следующим ресурсам:

- <https://github.com/llvm/llvm-project/releases>
- <https://apt.llvm.org/>
- <https://packages.debian.org/experimental/llvm-20-dev>

На момент тестирования собранный под RISC-V `llvm-20-dev` был найден только на `packages.debian.org`, в ветке `debian experimental`.

При обновлении версии LLVM потребовалось внести несколько изменений для корректной компиляции Rellume. Код доступен по данной ссылке.¹⁸

5.2. Результат

После внесения изменений, ошибки в unit-тестах Rellume сохранились, однако теперь сообщение об ошибке содержит номера релокаций,

¹⁸<https://github.com/aengelke/rellume/compare/master...mikhaylovilya:rellume:unlock-llvm-19-20> (дата обращения 26.05.2025)

которые еще не реализованы в `llvm::RuntimeDyldELF`.

Листинг 6: Нереализованные релокации в `RuntimeDyldELF`

```
5/9 rellume / emulation-aarch64-jit      ERROR      2.07s   killed by signal 6 SIGABRT
>>> /home/debian/repos/rellume/build-clang-20/tests/test_driver -A aarch64 -j tests/
    parsed_cases_aarch64.txt
stderr:
    LLVM ERROR: Unimplemented reloc type: 54
    (test program exited with status code -6)

6/9 rellume / emulation-x86_64-pic      ERROR      4.99s   killed by signal 6 SIGABRT
>>> /home/debian/repos/rellume/build-clang-20/tests/test_driver -A x86_64 -p tests/
    parsed_cases_x86_64.txt
stderr:
    WARNING: this target does not support the llvm.readcyclecounter intrinsic. It is being
           lowered to a constant 0
    LLVM ERROR: Unimplemented reloc type: 55
    (test program exited with status code -6)
```

6. Тестирование Instrew

6.1. Unit-тесты

6.1.1. Релокации

При запуске unit-тестов Instrew были обнаружены ошибки вида «Unhandled relocation». Нужно добавить реализацию соответствующих релокаций для успешного прохождения тестов.

Различные типы релокаций вычисляются по-разному. Выражения для их вычислений описаны в документации RISC-V psABI¹⁹. Также примеры реализаций релокаций можно найти в Github репозитории ядра Linux²⁰.

Листинг 7: Релокация

```
uint64_t syma = (uintptr_t) sym + patch_data->addend;
uint64_t pc = patch_data->patch_addr;
int64_t prel_syma = syma - (int64_t) pc;

case R_RISCV_PCREL_HI20:
    if (!rtld_elf_signed_range(prel_syma, 32, "R_RISCV_CALL_PLT"))
        return -EINVAL;
    rtld_blend(tgt + 4, 0xfff00000, prel_syma << 20);
    rtld_blend(tgt, 0xfffff000, prel_syma + 0x800);
    break;
```

6.1.2. PLT

При тестировании были обнаружены проблемы, с разрешением некоторых символов из libgcc и libatomic для RISC-V хоста. Для исправления данных ошибок были добавлены дополнительные символы в PLT, а к зависимостям клиента Instrew была добавлена библиотека libatomic.

Листинг 8: PLT

¹⁹<https://github.com/riscv-non-isa/riscv-elf-psabi-doc/blob/master/riscv-elf.adoc#relocations> (дата обращения 26.05.2025)

²⁰<https://github.com/torvalds/linux/blob/master/arch/riscv/kernel/module.c> (дата обращения 26.05.2025)

```
#if defined(__riscv)
PLT_ENTRY("__muldi3", __muldi3) // libgcc
PLT_ENTRY("__multi3", __multi3) // libgcc
PLT_ENTRY("__umoddi3", __umoddi3) // libgcc
PLT_ENTRY("__udivdi3", __udivdi3) // libgcc
PLT_ENTRY("__atomic_fetch_add_4", __atomic_fetch_add_4)
PLT_ENTRY("__atomic_exchange_4", __atomic_exchange_4)
#endif //defined(__riscv)
```

6.1.3. Результат

В результате исправления данных ошибок, все unit-тесты Instrew проходят успешно.

6.2. Тестирование Instrew на Coreutils

В качестве набора программ для тестирования был выбран Coreutils²¹ (версия 9.6.38), так как:

- у Coreutils не очень много зависимостей:
 - glibc, libgmp, libcrypto, libacl, libattr, libcap;
- Coreutils поддерживает кросс-компиляцию и статическую компиляцию.

В качестве тестового стенда используем:

- Debian 12 6.12.17-1 (2025-03-01) riscv64 GNU/Linux, как виртуальную машину²²

6.2.1. Подготовка тест-кейсов

Для тестирования подготавливаем 2 вариации Coreutils:

- с guest архитектурой x86_64, статически скомпонованный;

²¹<https://www.gnu.org/software/coreutils/> (дата обращения 26.05.2025)

²²https://gitlab.com/api/v4/projects/giomasce%2Fdqib/jobs/artifacts/master/download?job=convert_riscv64-virt (дата обращения 26.05.2025)

- с guest архитектурой riscv64, статически скомпонованный.

Чтобы скросскомпилировать на платформе x86_64, получив статические программы, были использованы следующие флаги:

Листинг 9: Статическая кросскомпиляция

```
bash -c './configure LDFLAGS="--static" --disable-xattr --disable-libcap --disable-libsmack --without-selinux --host-riscv64 --linux-gnu && make'
```

Coreutils не поддерживает запуск тестов в произвольной директории, поэтому во всех тестовых скриптах был изменен путь к директории, где лежат собранные программы.

Для запуска тестовых программ, был сделан скрипт, который по программе создает обёртку, где с помощью Instrew вызывается нужная тестовая программа. Скрипты для запуска текстов находятся в репозитории instrew-artifacts²³.

Листинг 10: Скрипт, для создания обёрток

```
for name in (find /home/debian/builds/cu-x86-gcc-static/ --
    executable -type f); do echo "/home/debian/repos/instrew/build
    -gcc-14-release/server/instrew name @" > (basename name);
    chmod +x (basename name); done
```

6.2.2. Результат

Guest Архитектура	Статический ср	Статический cat	Статический factor
rv64	проходят 2 из 10 тестов	проходят 1 из 4 тестов	проходят 3 из 4 тестов
x86_64	проходят 5 из 10 тестов	проходят 3 из 4 тестов	проходят 3 из 4 тестов

²³<https://github.com/mikhaylovilya/instrew-artifacts/tree/master> (дата обращения 26.05.2025)

7. Замеры и сравнение производительности

В качестве эксперимента используем сравнение замеров производительности эмуляции x86_64 Instrew, QEMU-user и Vex64, также сравнение производительности эмуляции riscv64 на Instrew, QEMU-user, а также запуская без эмуляции. Замеры производились на плате Banana PI BPI-F3.

7.1. Обзор бенчмарков

Были рассмотрены следующие бенчмарки:

- SPEC CPU 2017
- Внутренний бенчмарк SuiteSparse LAGraph
- Внутренний бенчмарк xxHash
- Внутренний бенчмарк LINPACK

В оригинальной диссертации по Instrew [4] сравнение производительности Instrew и QEMU-user проводилось на бенчмарке SPEC CPU 2017. Однако этот бенчмарк является платным, поэтому было принято решение рассмотреть проекты с открытым исходным кодом, содержащие бенчмарки.

Также был рассмотрен внутренний бенчмарк SuiteSparse LAGraph, но Instrew не поддерживает многопоточность, а SuiteSparse GraphBLAS архитектурно является многопоточной библиотекой, которая использует OpenMP.

В итоге для замеров производительности были выбраны xxHash и LINPACK.

7.2. Условия эксперимента

7.2.1. Окружение

- Плата: Banana Pi BPI-F3, процессор SpacemiT K1, 8 ядер, 4гб ОЗУ;
- ОС: Bianbu 2.1;
- Версии компиляторов:
 - gcc-14: (Bianbu 14.2.0-4ubuntu2 24.04bb1) 14.2.0;
 - g++-14: (Bianbu 14.2.0-4ubuntu2 24.04bb1) 14.2.0;
 - glibc: GNU C Library (Ubuntu GLIBC 2.39-0ubuntu8bb1) stable release version 2.39.

Также компиляция бенчмарков и кросс-компиляция box64 под RISC-V проводилась в следующем окружении

- Ноутбук: HP Laptop 15s-eq2039ur, 6 ядер, 12 потоков, 16гб ОЗУ;
- ОС: Debian 6.7.12-1 bpo12+1 (2024-05-06) x86_64 GNU/Linux;
- Версии компиляторов:
 - gcc-12: gcc (Debian 12.2.0-14+deb12u1) 12.2.0;
 - g++-12: g++-12 (Debian 12.2.0-14+deb12u1) 12.2.0;
 - riscv64-linux-gnu-gcc-12: riscv64-linux-gnu-gcc-12 (Debian 12.2.0-13) 12.2.0;
 - riscv64-linux-gnu-g++-12: riscv64-linux-gnu-g++-12 (Debian 12.2.0-13) 12.2.0;
 - glibc: GNU C Library (Debian GLIBC 2.36-9+deb12u10) stable release version 2.36;
 - glibc riscv64: GNU C Library (Debian GLIBC 2.36-8) stable release version 2.36.

7.2.2. Версии эмуляторов

- Версия instrew: коммит a3198b8²⁴;
- Версия box64: v0.34²⁵;
- Версия qemu-x86_64: qemu-x86_64 version 8.2.2 (Debian 1:8.2.2+ds-0ubuntu1.4);
- Версия qemu-riscv64: qemu-riscv64 version 8.2.2 (Debian 1:8.2.2+ds-0ubuntu1.4);

Box64 собирался на ноутбуке (кросскомпиляция) со следующими флагами:

Листинг 11: Флаги кросскомпиляции Box64

```
cmake .. -DCMAKE_C_COMPILER=riscv64-linux-gnu-gcc-12 -D RV64=1 -D RV64_DYNAREC=ON -D  
CMAKE_BUILD_TYPE=RelWithDebInfo
```

7.3. Замеры производительности на LINPACK

Компиляция LINPACK вариации x86 проводилась на ноутбуке со следующими флагами:

Листинг 12: Флаги компиляции linpack-generic-static-x86_64

```
gcc -Wall -Wextra -O3 -std=c11 -pedantic -static linpack.c -o linpack-generic-static-x86_64
```

Компиляция LINPACK вариации riscv64 проводилась на плате со следующими флагами:

Листинг 13: Флаги компиляции linpack-rv64

```
gcc-14 -Wall -Wextra -O3 -std=c11 -pedantic -static linpack.c -o linpack-generic-static-  
x86_64
```

²⁴<https://github.com/mikhaylovilya/instrew/tree/tmp-host-riscv64> (дата обращения 26.05.2025)

²⁵<https://github.com/ptitSeb/box64/tree/v0.3.4> (дата обращения 26.05.2025)

Аналогичными утилитами была зафиксирована частота процессора до 1.6Ghz и замеры выполнялись на фиксированном ядре. Данные²⁶ проходят тест на нормальность²⁷. Результаты производительности на бенчмарке LINPACK для гостевых архитектур x86_64 и RISC-V представлены на рис. 2 и 3 соответственно.

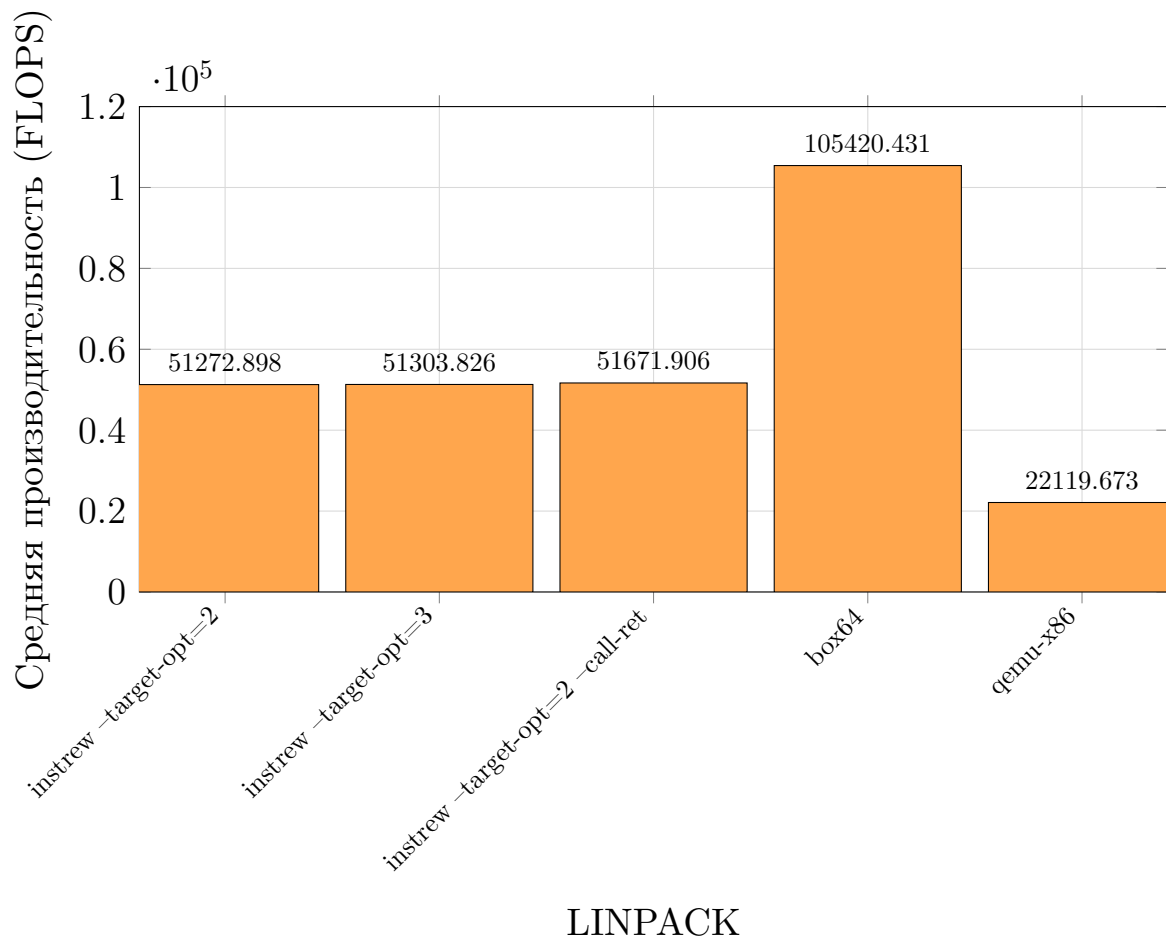
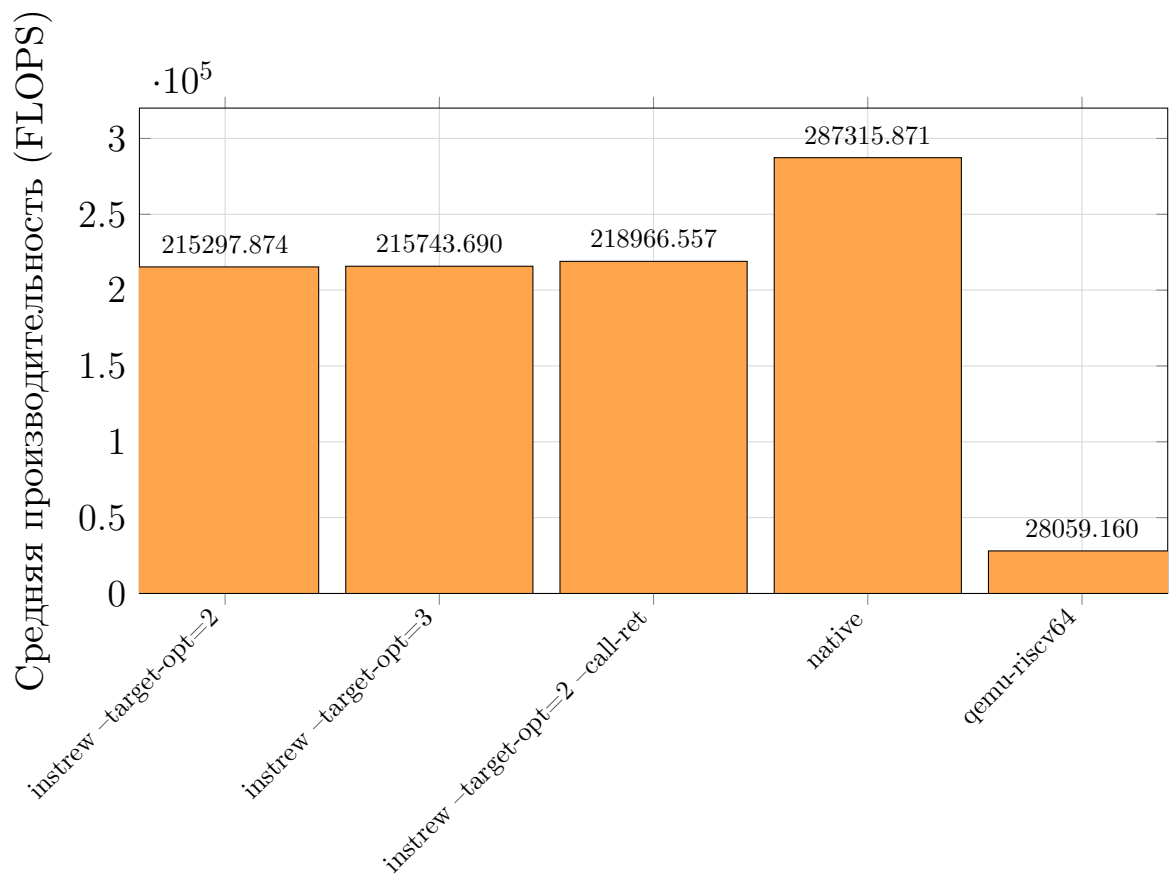


Рис. 2: Сравнение производительности (FLOPS) на linpack-generic-static-x86

²⁶<https://github.com/mikhaylovilya/instrew-artifacts/blob/master/bench-data/benchresults-raw-linpack.txt> (дата обращения 26.05.2025)

²⁷https://github.com/mikhaylovilya/instrew-artifacts/blob/master/bench/measurement_linpack.ipynb (дата обращения 26.05.2025)



LINPACK

Рис. 3: Сравнение производительности (FLOPS) для linpack-rv64g

7.4. Замеры пропускной способности на xxHash

Компиляция x86_64 бенчмарка²⁸ производилась с помощью запуска `make` в директории `xxHash/tests/bench` на ноутбуке. Аналогично компиляция riscv64 бенчмарка производилась также с помощью команды `make` на плате. Для запуска был сделан скрипт, была зафиксирована частота процессора с помощью `cpupower-set` до 1.6Ghz и программа выполнялась на одном зафиксированном ядре с помощью утилиты `taskset`. Результаты замеров доступны в репозитории²⁹. На рис. 4 представлена пропускная способность ХХН128 на данных различного размера.

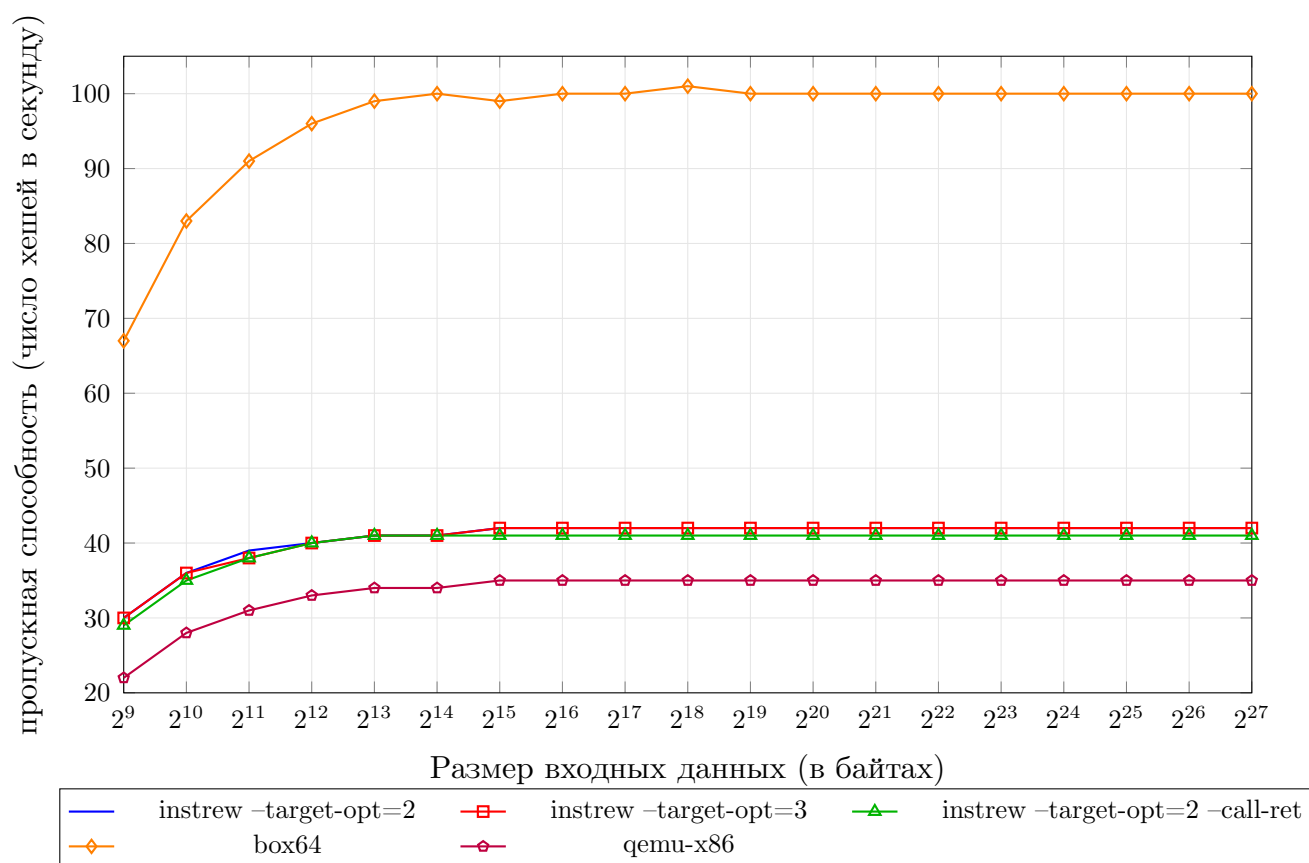


Рис. 4: Сравнение пропускной способности xxHash (ХХН128) для данных от 512 Б до 128 МиБ.

²⁸<https://github.com/Cyan4973/xxHash/tree/dev/tests/bench> (дата обращения 26.05.2025)

²⁹<https://github.com/mikhaylovilya/instrew-artifacts/blob/master/bench-data/benchresults-raw-xxhash.txt> (дата обращения 26.05.2025)

8. Реализация CI

Github Actions CI нативно не поддерживает архитектуру RISC-V³⁰. Для интеграции сборки и тестирования в CI на архитектуре RISC-V существуют два варианта:

- self-hosted runner на RISC-V;
- эмуляция RISC-V, например с помощью QEMU-user.

8.1. Self-hosted runner

Для того чтобы развернуть runner на виртуальной машине или плате, необходимо установить на неё Dotnet SDK. В Github были найдены сборки runner'а и Dotnet SDK для платформы RISC-V³¹.

- На виртуальной машине удалось развернуть только Dotnet SDK версии 9.x.x, на момент тестирования, runner мог использовать только версию Dotnet 8.x.x, которую не удалось заставить корректно работать³². Данный подход в качестве решения не подходит, так как для использования runner'а в связке с Dotnet 9.x.x необходимо пересобирать runner.
- На плате Sipeed Lichee PI4A удалось развернуть Dotnet версии 8.x.x и runner. Удалось добавить runner к репозиторию, как self-hosted, и интегрировать сборку и запуск тестов.

8.2. Эмуляция с помощью QEMU-user

Для удобства эмуляции с помощью QEMU-user был использован setup-alpine³³, так как поддерживает платформу RISC-V. Переписываем .yaml файл под Alpine Linux.

³⁰<https://docs.github.com/en/actions/using-github-hosted-runners/using-github-hosted-runners/about-github-hosted-runners#standard-github-hosted-runners-for-public-repositories> (дата обращения 26.05.2025)

³¹https://github.com/dkurt/github_actions_riscv (дата обращения 26.05.2025)

³²https://github.com/dkurt/dotnet_riscv/issues/10#issuecomment-2220630571 (дата обращения 26.05.2025)

³³<https://github.com/jirutka/setup-alpine/tree/v1> (дата обращения 26.05.2025)

8.3. Результат

Оба описанных способа были опробованы. Для внедрения в проект был выбран способ с эмуляцией, так как:

- Github Actions runner плохо поддерживает RISC-V;
- поддерживать self-hosted runner ресурсоёмко и не входит в задачи данной работы;
- для мейнтейнера неудобно использовать self-hosted runner.

Заключение

В результате ВКР были выполнены следующие задачи:

1. выполнен обзор архитектуры Instrew и построена UML-диаграмма классов клиента Instrew;
2. выполнено портирование Instrew на архитектуру RISC-V:
 - добавлены процессорно-специфические патчи, очистка кэша инструкций и эмуляция системных вызовов;
 - добавлена реализация частей RISC-V ABI: релокации, недостающие системные вызовы, недостающие символы PLT;
 - обновлена версия LLVM до 20-ой для запуска внутренних интеграционных тестов Rellume;
3. выполнено тестирование на Coreutils:
 - добавлены скрипты для развертывания тестируемого окружения;
 - исправлены ошибки Instrew;
4. в CI добавлена поддержка сборки для RISC-V:
 - проведен обзор способов добавления поддержки сборки на RISC-V;
 - добавлена поддержка сборки и запуска внутренних интеграционных тестов через setup-alpine;
5. выполнены замеры производительности Instrew на RISC-V:
 - выполнен обзор бенчмарков;
 - выполнены замеры производительности на xxHash и LAPACK;
 - проведено сравнение результатов замеров производительности Instrew с QEMU-user и с Vex64.

Код доступен по данной ссылке:

<https://github.com/aengelke/instrew/pull/20>.

Имя пользователя: mikhaylovilya (Илья Mikhaylov)

Список литературы

- [1] Bellard Fabrice. QEMU, a fast and portable dynamic translator // Proceedings of the Annual Conference on USENIX Annual Technical Conference. — ATEC '05. — USA : USENIX Association, 2005. — P. 41.
- [2] [DBILL: an efficient and retargetable dynamic binary instrumentation framework using llvm backend](#) / Yi-Hong Lyu, Ding-Yong Hong, Tai-Yi Wu et al. // Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. — VEE '14. — New York, NY, USA : Association for Computing Machinery, 2014. — P. 141–152. — URL: <https://doi.org/10.1145/2576195.2576213>.
- [3] Engelke Alexis, Schulz Martin. [Instrew: leveraging LLVM for high performance dynamic binary instrumentation](#) // Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. — VEE '20. — New York, NY, USA : Association for Computing Machinery, 2020. — P. 172–184. — URL: <https://doi.org/10.1145/3381052.3381319>.
- [4] Engelke Alexis Friedrich. Optimizing Performance Using Dynamic Code Generation : Ph. D. thesis / Alexis Friedrich Engelke ; Technische Universität München. — 2021. — P. 163. — URL: <https://mediatum.ub.tum.de/1614897>.
- [5] [HQEMU: a multi-threaded and retargetable dynamic binary translator on multicores](#) / Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew et al. // Proceedings of the Tenth International Symposium on Code Generation and Optimization. — CGO '12. — New York, NY, USA : Association for Computing Machinery, 2012. — P. 104–113. — URL: <https://doi.org/10.1145/2259016.2259030>.
- [6] Nethercote Nicholas, Seward Julian. [Valgrind: a framework for heavy-weight dynamic binary instrumentation](#) // Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. — PLDI '07. — New York, NY, USA : Association for Com-

puting Machinery, 2007. — P. 89–100. — URL: <https://doi.org/10.1145/1250734.1250746>.