

Министерство образования и науки Российской Федерации
Московский физико-технический институт (государственный
университет)

Физтех-школа радиотехники и компьютерных технологий
Кафедра Микропроцессорных технологий в интеллектуальных
системах управления
Syntacore

Выпускная квалификационная работа бакалавра

Гибкий подход к подъёму LLVM MIR кода
открытой архитектуры RISC-V в SSA форму
LLVM IR

Автор:

Студент Б01-110 группы
Романов Александр Викторович

Научный руководитель:

Владимиров Константин Игоревич



Москва 2025

Аннотация

Гибкий подход к подъёму LLVM MIR кода открытой архитектуры RISC-V в SSA форму LLVM IR

Романов Александр Викторович

Проблема бинарной совместимости программ и их переносимости на разные архитектуры без возможности перекомпиляции часто решается при помощи бинарных трансляторов. Существует большое количество статических и динамических бинарных трансляторов. Большинство из них работают либо за счёт прямого сопоставления инструкциям и регистрам исходной архитектуры инструкции и регистры целевой архитектуры, либо за счёт паттерн матчинга. Такие решения делают сложным поддержание новых исходных архитектур ввиду чего поддержка относительно молодой микропроцессорной архитектуры RISC-V в существующих трансляторах либо отсутствует, либо сильно ограничена.

В данной работе рассмотрен новый инструмент для подъёма машинно зависимого представления RISC-V кода LLVM MIR в высокоуровневое машинно-независимое представление LLVM IR и его применение для простой статической трансляции бинарного RISC-V кода на любую поддерживаемую LLVM архитектуру.

Содержание

1 Введение	4
1.1 Бинарная совместимость	4
1.2 Архитектура RISC-V	5
1.3 Компилятор LLVM	6
1.3.1 LLVM IR	7
1.3.2 LLVM MIR	8
2 Постановка Задачи	10
3 Обзор существующих решений	11
3.1 Эмуляция	11
3.2 Бинарная трансляция	11
3.3 Лифтеры в высокоуровневое представление	12
3.4 Вывод	13
4 Исследование и построение решения задачи	14
4.1 Использование описания архитектуры из LLVM	14
5 Case Study / Evaluation	16
5.1 Design	16
5.2 Objectives	16
5.3 Results	16
5.4 Findings	16
5.5 Discussion	16
5.6 Limitations	16
6 Заключение	16
6.1 Status	16
6.1.1 Realized Goals	16
6.1.2 Open Goals	16
6.2 Conclusion	16
6.3 Future Work	16
List of Figures	17
Appendix A: Supplementary Material	18
Список Литературы	19

1 Введение

Проблема переноса программ между различными архитектурами, в ситуациях, когда перекомпиляция из исходных файлов сложна, либо вовсе невозможно является одной из актуальных проблем компьютерных технологий. Основным способом решения этой проблемы является бинарная трансляция, которая решает проблемы бинарной совместимости путём преобразования машинного кода исходной архитектуры в машинный код целевой архитектуры. С развитием новых архитектур и операционных систем такое преобразование кода становится всё более сложной, что делает развитие бинарных трансляторов важной задачей современной вычислительной техники.

1.1 Бинарная совместимость

Бинарный код состоит из закодированных инструкций для конкретной архитектуры команд. При компиляции программы её код на высокоуровневом языке программирования (Например C/C++/Fortran) переводится в бинарный код целевой архитектуры и операционной системы.

Бинарной совместимостью называется возможность исполнения бинарного кода, скомпилированного под одну архитектуру команд и операционную систему на других устройствах и системах без модификации этой программы. Бинарная совместимость является одной из фундаментальных проблем в сфере компьютерных технологий в связи с постоянным развитием архитектур набора команд и операционных систем.

Основными проблемами для бинарной совместимости являются:

1. Различные архитектуры команд (ISA). Процессорные архитектуры являются главной причиной бинарной несовместимости. Процессоры каждой архитектуры исполняют свой уникальный набор команд и не работают с другими. Кроме различия в наборе инструкций архитектуры могут также отличаться размерос инструкций. Например, X86 и RISC-V поддерживают инструкции разной длины, в то время как ARM фиксирует длину всех инструкций в 4 байта. Архитектуры также отличаются набором регистров, принципами доступа к памяти а также порядком байт (например big-endian или little-endian). В то время как разница в наборе инструкций чаще всего влечёт за собой быструю остановку программы из-за невалидной инструкции, разница в порядке доступов к памяти при прочих равных может вызывать непредсказуемой поведение программы.
2. Операционные системы (ОС) также играют большую роль в бинарной несовместимости. Набор и мезханизм системных вызовов отличается на разных

платформах (К примеру, `open` для Linux систем и для FreeBSD работают по разному, несмотря на общее название). Наборы системных вызовов также могут отличаться от версии к версии одной операционной системы. Например Windows не имеет фиксированного набора системных вызовов и они часто изменяются между версиями.

3. Соглашение о вызовах обычных функций (ABI) также значительно отличаются даже внутри одной архитектуры (Например программа, написанная под RISC-V процессор с LP64D не будет работать для RISC-V с LP64F).
4. Наконец, окружение запуска (Набор доступных на момент запуска динамических библиотек) также является критически важным для запуска программы и может значительно отличаться как от машины к машине, так и на разных версиях операционной системы (Например программа, скомпилированная динамически для операционной системы Ubuntu не сможет найти динамические библиотеки на устройстве с операционной системой Arch, т.к. эти библиотеки будут установлены по другим путям)

1.2 Архитектура RISC-V

RISC-V – это открытая и расширяемая архитектура команд, разработанная в университете Беркли (Калифорния, США) в 2014 году. Архитектура имеет модульную основу, что в совокупности с открытой лицензией позволяет кому угодно разрабатывать и создавать процессоры, с лёгкостью добавляя расширения, специфичные для их задачи без необходимости платить за лицензию. Такая доступность и расширяемость архитектуры быстро сделала её популярной как в академической среде (В которой RISC-V De facto является стандартом для обучения микроархитектуре), так и в среде разработчиков микроконтроллеров и даже высокопроизводительных систем. В отличие от многих других архитектур (Таких как X86 и ARM) RISC-V является крайне минималистичной и имеет меньше 50 инструкций в базовом наборе команд (В который не входит даже аппаратное умножение и деление чисел). Такая степень модульности означает минимализацию поддержки, необходимой для обратной совместимости, в то время как конкурирующим платформам приходится долго с этим бороться.

Т.к. RISC-V имеет модульную архитектуру, то вопрос бинарной совместимости остро стоит даже между разными конфигурациями RISC-V микропроцессоров. Например, программа, написанная под RV64IC (включающий в себя расширение для сжатых инструкций) никогда не запустится на устройстве с RV64I, на котором

эти сжатые инструкции не поддерживаются. Учитывая наличие не только стандартных, но и пользовательских расширений RISC-V представляет собой бесконечный набор конфигураций процессорных ядер, бинарная совместимость программ между которыми является скорее исключением, чем правилом. Это придаёт вопросу бинарной трансляции между разными RISC-V ядрами высокий приоритет.

1.3 Компилятор LLVM

Компилятор - это программа, переводящая код программы с высокоуровневого языка программирования в машинный код заданной архитектуры. В современном мире подавляющее число компиляторов являются оптимизирующими, т.е. компиляторами, производящими широкий спектр оптимизаций над исходным кодом. В своей работе компиляторы используют различные промежуточные представления для исходного кода, наиболее частыми из которых являются:

- AST (Abstract Syntax Tree) – Дерево абстрактного синтаксиса является структурой данных, отражающей программу написанную на высокоуровневом языке программирования. Такое дерево называется абстрактным, так как оно отражает не реальный синтаксис программы, а лишь его смысловую часть, необходимую для дальнейшей компиляции программы. Естественным образом AST является специфичным для конкретного языка программирования представлением.
- HIR (High-level Intermediate Representation) – Высокоуровневое промежуточное представление, абстрагирующее код от исходного языка программирования, но всё ещё независимое от архитектуры. Состоит из виртуальных инструкций, сохраняющих семантику программы, что позволяет производить на данном представлении большое число машиннонезависимых оптимизаций. HIR также позволяет имплементировать высокоуровневые оптимизации (Например продвижение констант, удаление мёртвого кода или подстановка функций). Такое высокоуровневое представление прекрасно подходит для оптимизации, анализа и преобразования кода, не задумываясь о языке программирования из которого он был получен или о целевой архитектуре.
- LIR (Low-level Intermediate Representation) – форма, в которую компилятор переводит HIR после того, как все возможные высокоуровневые оптимизации были выполнены. Низкоуровневое представление, как следует из его названия близко к машинному коду и состоит уже в основном из конкретных инструкций целевой архитектуры (Или псевдоинструкций, которые раскрываются в конкретные инструкции в процессе оптимизаций). Кроме выбора инструкций и регистров

целевой архитектуры LIR также позволяет производить ряд машинно-зависимых оптимизаций, т.е. изменение машинных инструкций, регистров или их порядка с целью повышения производительности (Например замена инструкции ADD инструкцией LEA на архитектуре X86).

Поговорим чуть более подробно о высокоуровневом промежуточном представлении. Важным понятием в высокоуровневом представлении является SSA (Static Single Assignment) форма кода, придуманная в 1980-х годах [CLZ86]. SSA отличается от других представления HIR тем, что каждая переменная в нём присваивается только один раз. Преимущества такой формы кода заключается в том, что все значения никогда не изменяются, это позволяет строить цепочки определений и использований (def-use chains), по которым удобно анализировать зависимости инструкций по данным. В точках схождения графа управления SSA код вставляет φ -функции, которые принимают пары из значений и базовых блоков из которых пришло управление (Изначально придуманные в [RWZ88]). Сейчас SSA используется во всех конкурентноспособных компиляторах, в том числе GNU Compiler Collection и LLVM.

Наконец поговорим о компиляторе LLVM. LLVM (Low Level Virtual Machines) – компиляторная инфраструктура, придуманная Крисом Латтнером в 2004 году [LA04]. Данная платформа выгодно отличается от других компиляторов (например GCC) своей модульной структурой, позволяющей переиспользовать отдельные её компоненты по отдельности. Такая простота в использовании позволило LLVM стать одной из самых популярных компиляторных платформ. На основе инфраструктуры LLVM разработаны:

- Компиляторы таких высокоуровневых языков программирования как C, C++, Fortran, Rust, Go и т.д.
- Отладчики (LLDB)
- JIT (Just In Time) компиляторы для таких языков как Java, Lua и Scala
- Генераторы случайных тестов [BA24]

Для дальнейших изложений необходимо познакомиться с HIR и LIR представлениями из инфраструктуры LLVM.

1.3.1 LLVM IR

LLVM IR – высокоуровневое промежуточное представление (HIR), используемое компиляторами на основе LLVM. LLVM IR имеет вид набора модулей (Чаще всего полученных из единиц трансляции высокоуровневых языков программирования), каждый из которых может содержать функции, глобальные объекты и

метаинформацию, необходимую для дальнейшей работы с этими модулями. Функции состоят из базовых блоков, которые, в свою очередь, представляют собой список последовательно исполняющихся инструкций, заканчивающийся инструкцией-терминатором. Терминаторы – инструкции, указывающие, какому базовому блоку следует передать управление. Название “Терминатор” появляется из того, что эти инструкции заканчивают базовые блоки.

Все инструкции в LLVM IR работают над значениями в SSA форме. Каждая инструкция, базовый блок, или глобальные объект определяют новое уникальное значение, которое после может быть операндом любой другой инструкции. Все операнды инструкций и их результаты имеют строго определённый тип. При схождении управления LLVM IR вставляет phi функции для выбора нового значения. В качестве примера приведём функцию на LLVM IR, рекурсивно вычисляющую факториал 32-битного числа и записывающую каждое промежуточное значение в глобальную переменную:

```
1  @res = global i32 0, align 4
2  define i32 @fact(int)(i32 %0) {
3      %2 = icmp eq i32 %0, 1
4      br i1 %2, label %3, label %5
5  3:
6      %4 = phi i32 [ %8, %5 ], [ 1, %1 ]
7      ret i32 %4
8  5:
9      %6 = add nsw i32 %0, -1
10     %7 = call i32 @fact(int)(i32 %6)
11     %8 = mul nsw i32 %7, %0
12     store i32 %8, ptr @res, align 4
13     br label %3
14 }
```

LLVM

Listing 1: Вычисление факториала на LLVM IR

1.3.2 LLVM MIR

LLVM MIR (Machine IR)– низкоуровневое промежуточное представление (LIR), используемое компиляторами на основе LLVM. MIR код, так же как и LLVM IR состоит из функций, базовых блоков и инструкций. Однако это представление уже не является SSA формой и инструкции в его составе отвечают конкретным инструкциям целевой архитектуры или являются псевдо-инструкциями, которые раскрываются

в настоящие по мере компиляции. Как уже было сказано, MIR работает не с SSA значениями, а уже с физическими или виртуальными регистрами. В данной работе мы будем рассматривать MIR работающий только с физическими регистрами.

```
1  name: fact
2  body: |
3    bb.0: successors: %bb.1, %bb.3
4      $x8 = COPY $x10
5      $x10 = ADDIW $x10, -1
6      BNE $x10, $x0, %bb.3
7    bb.1: successors: %bb.2
8      $x10 = ADDI $x0, 1
9    bb.2:
10     PseudoRET $x10
11    bb.3: successors: %bb.2
12     liveins: $x8, $x10
13     PseudoCALL @fact, implicit-def $x1, implicit $x10, implicit-def $x2, implicit-def $x10
14     $x10 = MULW $x10, $x8
15     J %bb.2
```

MIR

Listing 2: Вычисление факториала на LLVM MIR для RISC-V

В Listing 2 видно конкретные инструкции RISC-V, а так же две псевдоинструкции: `PseudoCALL` и `PseudoRET`, которые при компиляции этого кода раскрылись бы в соответствующие инструкции RISC-V. Как видно, MIR – это представление, позволяющее работать с конкретными инструкциями и регистрами целевой архитектуры, не теряя при этом информации о базовых блоках, функциях и других высокоуровневых объектах.

2 Постановка Задачи

Разработка бинарных трансляторов является сложной задачей ввиду большого набора архитектур и особенностей их системы команд. В задаче также стоит учитывать производительность транслированного кода, ради улучшения которой над транслированным кодом приходится производить различные оптимизации, написание которых само по себе является достаточно сложной задачей. Также имеет значимость задача восстановления высокоуровневого представления бинарных программ для их анализа, формальной верификации и оптимизации. Наиболее развитым из таких высокоуровневых представлений является LLVM IR. Актуальность открытой и расширяемой архитектуры RISC-V также растёт и всё больше промышленных производителей начинают выпускать чипы на базе этого набора команд.

Предлагается разработать способ поднятия машинного кода открытой и расширяемой архитектуры RISC-V в высокоуровневое представление LLVM IR. В связи с непрерывной разработкой новых расширений для RISC-V нужен способ быстрой их поддержки. Таким образом поднять код требуется способом, позволяющим лёгкое добавление поддержки новых расширений этой архитектуры.

Итого основная цель данной работы – разработать инструмент для восстановления LLVM IR из машинного кода открытой и расширяемой архитектуры RISC-V.

Данную цель разобьём на следующие шаги:

1. Изучить существующие подходы к восстановлению высокоуровневого представления для разных архитектур набора команд
2. Разработать модель для семантического переноса модели RISC-V, а именно RV64IM и RV32IM
3. Разработать инструмент перевода кода в LLVM IR на основе построенной модели и инфраструктуры LLVM

3 Обзор существующих решений

Наиболее распространёнными подходами к решению бинарной несовместимости являются бинарные трансляторы или эмуляторы. Рассмотрим оба понятия.

3.1 Эмуляция

Эмуляторы – инструменты, позволяющие исполнять машинный код другой архитектуры при помощи симуляции всей исходной архитектуры, включая симуляции регистров, особенностей подсистемы памяти и конвейера исполнения. Эмуляторы часто являются самым простым способом исполнения кода несовместимой архитектуры. Такой подход отличается невысокой производительностью, следующей из затрат на точную симуляцию всех параметров исходной системы.

3.2 Бинарная трансляция

Бинарная трансляция – процесс перевода бинарного кода исходной архитектуры в бинарный код целевой архитектуры. Бинарная трансляция может происходить на уровне процессорных микросхем, либо с помощью программ, называемых бинарными трансляторами. В данной работе мы будем фокусироваться на программной бинарной трансляции. Такие трансляторы можно разделить на два типа: Статические и динамические.

- Статическими бинарными трансляторами называются программы, принимающие на вход исполняемый бинарный файл исходной архитектуры целиком и преобразующие его в исполняемый файл целевой архитектуры. В процессе статической бинарной трансляции процесс перевода кода на другую архитектуру и его исполнение разделены: Сначала переводится весь код целиком, после он может быть исполнен. Такой вид бинарной трансляции может быть выполнен без возможности исполнить исходный код. Этот подход усложняется тем, что не весь исполняемый код программы может быть доступен бинарному транслятору. Например, некоторые куски кода и метки могут достигаться программой через косвенные переходы (передача управления при помощи прыжка по адресу, записанному в регистре или в памяти), которые может быть тяжело или вовсе невозможно анализировать без предварительного исполнения. Часто статическая бинарная трансляция производится за счёт декомпиляции – перевода кода в высокоуровневое представление (Например с помощью [Roh19]), обратной инженерии и последующей компиляции на целевую архитектуру.

- Динамические бинарные трансляторы – программы, переводящие исходный машинный код на целевую архитектуру по требованию. В этом подходе транслируется маленький кусок кода (чаще всего в пределах одного базового блока), после чего сразу исполняется на целевой архитектуре и контекст исполнения (значения регистров и памяти) сохраняется. При достижении инструкций перехода начинает транслироваться новый кусок кода или (в случае циклов) исполняться уже транслированный. Стоит отметить, что динамическая трансляция отличается от эмуляции архитектуры, ведь при таком подходе инструкции исходной архитектуры напрямую переводятся в инструкции целевой архитектуры, без симуляции такого контекста исходной архитектуры, как регистров, таблицы прерываний и специфики памяти. Это означает, что динамические бинарные трансляторы в среднем обладают большей производительностью чем эмуляторы, что позволяет применять их в более широком спектре задач. Приведём наиболее иллюстративные примеры динамических бинарных трансляторов:
 - Rosetta – транслятор, который использовался компанией Apple для упрощения перехода их персональных компьютеров с Архитектуры PowerPC на X86 в 2005 году [Ros25].
 - IA-32 Execution Layer – динамический бинарный транслятор, разработанный компанией Intel в 2003 году для производительного исполнения 32-битных программ на более новой 64-битной архитектуре Itanium [Bar+03].
 - Berberis – относительно новый инструмент, разработанный Google для запуска RISC-V Android приложений на устройствах с архитектурой X86_64 [Ber25].

Теперь, получив общее представление о эмуляторах и бинарных трансляторах, мы можем поговорить о способах подъёма машинный код в высокоуровневое представление.

3.3 Лифтеры в высокоуровневое представление

Большая часть бинарных трансляторов работает за счёт прямого сопоставления инструкций и регистров целевой архитектуры инструкциям и регистрам исходной (см [Ros25], [Bel05] и [Ber25]). Такой подход делает процесс бинарной трансляции уникальным для каждого сочетания из исходной и целевой архитектур, что делает поддержку новых целевых архитектур сложной задачей. Подъём кода в промежуточное представление и его перекомпиляция на целевую архитектуру – принцип, позволяющий сильно упростить добавление новых целевых архитектур.

Такой подход можно применять как для статической, так и для динамической бинарной трансляции. Наиболее популярными высокоуровневыми представлениями для подъёма кода являются язык C и LLVM IR. Мы сфокусируемся на подъёме в машинного кода в LLVM IR, т.к. подъём в C чаще применяется для обратной разработки, а не для бинарной трансляции. Рассмотрим существующие инструменты для получения LLVM IR из машинного кода:

- `llvm-mctoll` – инструмент, разработанный компанией Microsoft [YS19]. Проект хорошо поддерживает подъём из X86 и ARM кода. Поддержка других архитектур отсутствует, что делает невозможным его использование для архитектуры RISC-V.
- `mcsema` – другой популярный инструмент для подъёма кода в LLVM IR. Данная программа поддерживает следующие архитектуры: X86, ARM и SPARC, таким образом снова невозможно её использование для наших задач [Fra25].
- `rellume` – единственный инструмент, способный поднимать не только код под ARM и X86, но и подмножество RISC-V [ES20]. К сожалению, поддерживается только 197 инструкций из всей спецификации архитектуры RISC-V и поддержка новых расширений в этом инструменте осложнена.
- `biotite` – новый инструмент для подъёма кода, работающий с подмножеством RISC-V кода, но требующий информацию об исходном коде программы, что не подходит для наших задач [Che+25].

3.4 Вывод

После проведения анализа существующих инструментов для подъёма машинного кода в LLVM IR были выявлены закономерности, которым подчиняются все существующие решения. Оценка присущих им недостатков позволяет выявить возможное направление дальнейшего развития этого направления.

Список инструментов, позволяющих поднимать код архитектуры RISC-V в LLVM IR мал и включает в себя всего две программы (см. [ES20] и [Che+25]). Подавляющее большинство существующих решений поддерживают только архитектуры X86 и ARM.

Существующие инструменты подъёма кода открытой архитектуры RISC-V работают лишь с небольшим её подмножеством. Проблема в поддержке всего подмножества RISC-V заключается в её модульной расширяемой системе. С появлением новых стандартных и вендорских расширений требуется модификация исходного кода лифтера и ручная обработка новых инструкций. Эта задача становится сложной из-за того, что в большинстве существующих решений всем

инструкциям или их комбинациям исходной архитектуры вручную сопоставляются инструкции LLVM IR, что порождает большое число краевых случаев и возможных комбинаций для распознавания паттернов.

Поддержка новых расширений для RISC-V также осложняется необходимостью поддерживать низкоуровневую информацию о регистрах и инструкциях. Такая информация необходима для декодирования бинарного машинного кода. Многие из представленных инструментов используют собственное описание инструкций и работают с ними через это самое описание. Например, в [Che+25] всем поддерживаемым инструкциям вручную сопоставляются их названия и возможные операнды. Далее бинарные файлы декодируются в два шага:

- С помощью программы-дизассемблера исходный машинный код переводится в своё текстовое представление
- Далее, используя информацию о названиях инструкций и их операндах, полученный текст дизассемблера разбивается на инструкции

Очевидным является то, что ручное описание каждой инструкции является предметом потенциальных ошибок. В контексте расширяемой архитектуры RISC-V это значит увеличение числа ошибок с увеличением числа поддерживаемых расширений. Также описание синтаксиса некоторых инструкций (например векторных) требует большой работы.

Таким образом, существующие инструменты для подъёма RISC-V кода в LLVM IR поддерживают лишь малое число инструкций этой архитектуры. Внедрение новых расширений затруднено и архитектура проектов не позволяет работы с вендорскими расширениями без модификации исходного кода и перевыпуска инструмента.

4 Исследование и построение решения задачи

В данной главе мы обсудим возможные решения проблем существующих лифтеров в LLVM IR. Отличительной чертой предложенного подхода будет являться минимализация ручного описания архитектуры.

4.1 Использование описания архитектуры из LLVM

Прежде всего мы решим проблемы поддержания низкоуровневой информации об инструкциях, регистрах и расширениях. Как уже было сказано, многие существующие инструменты поддерживают собственное описание инструкций для работы с машинным кодом. Предлагается полностью решить эту проблему, переиспользовав уже существующие описания. LLVM, являясь компиляторной

инфраструктурой, обязан уметь корректно порождать и читать машинный код. Большим плюсом этой платформы является развитое описание всех инструкций и всех расширений RISC-V (Как и подавляющего большинства других архитектур, таких как X86, ARM, MIPS и даже SPIRV). LLVM обладает информацией о кодировках, операндах, семантике ассемблера всех инструкций из большинства существующих расширений RISC-V. Вместо того, чтобы описывать всю эту информацию вручную предлагается преиспользовать существующее описание инструкций из LLVM. Большим плюсом такого подхода является снижение вероятности ошибки. Все расширения, поддерживаемые в LLVM косвенно тестируются компилятором `clang`. Это означает, что если в описании некоего расширения допущена ошибка, то компиляция под архитектуру процессора, включающего это расширение либо произойдёт ошибка, либо исполнение программы будет некорректным, что приведёт к мотивации исправить недочёт в описании для стабильной работы компилятора `clang`. Таким образом в нашем инструменте будет доступна корректная информация о всех расширениях RISC-V, поддерживаемых компилятором `clang` (Таких расширений на момент написания данной работы больше 100).

Рассмотрим подробнее средства средства взаимодействия с машинным описанием, предоставляемые библиотекой LLVM.

5 Case Study / Evaluation

5.1 Design

5.2 Objectives

5.3 Results

5.4 Findings

5.5 Discussion

5.6 Limitations

6 Заключение

6.1 Status

6.1.1 Realized Goals

6.1.2 Open Goals

6.2 Conclusion

6.3 Future Work

List of Figures

Appendix A: Supplementary Material

– Supplementary Material –

Список Литературы

- [CLZ86] R. Cytron, A. Lowry, and F. K. Zadeck, "Code motion of control structures in high-level languages" in Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, 1986. doi: 10.1145/512644.512651.
- [RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations" in Proceedings of the 15th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, 1988. doi: 10.1145/73560.73562.
- [LA04] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation" in Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, 2004. doi: 10.5555/977395.977673.
- [BA24] К. Владимиров and И. Андреев, "Эффективное построение переупорядочиваний множества операций с памятью в многопоточной программе," 2024, *Международный научный журнал "Современные информационные технологии и ИТ-образование"*. doi: 10.25559/SITITO.020.202401.149-156.
- [Roh19] R. Rohleder, "Hands-On Ghidra - A Tutorial about the Software Reverse Engineering Framework. In Proceedings of the 3rd ACM Workshop on Software Protection, 2019, *Association for Computing Machinery, London, UK*. doi: 10.1145/3338503.3357725.
- [Ros25] "Rosetta." Accessed: Jun. 14, 2025. [Online]. Available: <https://web.archive.org/web/20060113055505/http://www.apple.com/rosetta/>
- [Bar+03] L. Baraz *et al.*, "IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium®-based systems. In Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003.
- [Ber25] "Berberis." Accessed: Jun. 14, 2025. [Online]. Available: https://android.googlesource.com/platform/frameworks/libs/binary_translation/

- [Bel05] F. Bellard, “QEMU, a fast and portable dynamic translator. In Proceedings of the USENIX Annual Technical Conference,” 2005. doi: 10.5555/1247360.1247401.
- [YS19] S. B. Yadavalli and A. Smith, “Raising binaries to LLVM IR with MCTOLL (WIP paper). In proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems,” 2019. doi: 10.1145/3316482.3326354.
- [Fra25] “Framework for lifting x86, amd64, aarch64, sparc32, and sparc64 program binaries to LLVM bitcode.” Accessed: Jun. 15, 2025. [Online]. Available: <https://github.com/lifting-bits/mcsema>
- [ES20] A. Engelke and M. Schulz, “Instrew: Leveraging LLVM for High Performance Dynamic Binary Instrumentation. In proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments ,” Mar. 2020. doi: 10.1145/3381052.3381319.
- [Che+25] C. Chen, S. Sugita, Y. Nada, H. Irie, S. Sakai, and R. Shioya, “Biotite: A High-Performance Static Binary Translator using Source-Level Information. In proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction ,” 2025.
- [14] A. Романов and К. Владимиров, “Гибкий подход к подъёму LLVM MIR кода в SSA форму LLVM IR,” *"Труды 67-й Всероссийской научной конференции МФТИ, Радиотехника и компьютерные технологии. 2025, С 70-71"*.