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 програм и их переносимости на разные архитектуры без возможности перекомпиляции часто решается при помощи бинарных трансляторов. Существует большое колличество 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 отсутсвует, 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 поддержанную LLVM архитектуру.

Содержание

1 Введение

Проблема переноса программ между различными архитектурами, в ситуациях, когда перекомпиляция из изходных 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 иструкций для конкретной архитектуры комманд. При компиляции программы её код на высокоуровневом языке програмирования (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 отличаются 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 отличатся 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, слинкованная 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 выскопроизводительных систем. В отличии 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 поддержаны. Учитывая наличие не только стандартных, но и пользовательских расширений RISC-V представляет собой бесконечный набор конфигураций процессорных ядер, бинарная совместимость програм 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 програмирования 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 абстрактого синтаксиса является структурой данных, отражающей программу написанную на высокоуровневом языке програмирования. 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 програмирования представлением.
- HIR (High-level Intermediate Representation) – Высокоуровневое промежуточное представление, абстрагирующее код от исходного языка програмирования, 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, незадумываясь о языке програмирования 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 опредеолений 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 Virual Machines) – компиляторная инфраструктура, придуманная Крисом Латтнером в 2004 году [LA04]. Данная платформа выгодно отличается от других кмпиляторов (например GCC) своеё модульной структурой, позволяющей переиспользовать отдельные её компоненты по отдельности. Такая простата в использовании позволило LLVM стать одной из самых популярных компиляторных платформ. На основе инфраструктуры LLVM разработаны:

- Компиляторы таких высокоуровневых языков програмирования 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 програмирования), 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 функию на LLVM IR, рекурсивно вычисляющую факториал 32-битного числа и записывающую каждое промежуточное значение в глобальную переменную:

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

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 будет рассматривать MIR работающий только с физическими регистрами.

```MIR
name: fact
body: |
  bb.0: successors: %bb.1, %bb.3
    $x8 = COPY $x10
    $x10 = ADDIW $x10, -1
    BNE $x10, $x0, %bb.3
  bb.1: successors: %bb.2
    $x10 = ADDI $x0, 1
  bb.2:
    PseudoRET $x10
  bb.3: successors: %bb.2
    liveins: $x8, $x10
    PseudoCALL @fact, implicit-def $x1, implicit $x10, implicit-def $x2, implicit-def $x10
    $x10 = MULW $x10, $x8
    J %bb.2
```

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 соответсвующие инструкции RISC-V. Как видно, MIR – это представление, позволяющее работать с конкретными инструкция 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 комманд. 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 програм 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 – разаботать 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 комманд
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

4 Requirements

4.1 Overview

4.2 Existing System

4.3 Proposed System

4.3.1 Functional Requirements

4.3.2 Quality Attributes

4.3.3 Constraints

4.4 System Models

4.4.1 Scenarios

4.4.2 Use Case Model

4.4.3 Analysis Object Model

4.4.4 Dynamic Model

4.4.5 User Interface

5 Architecture

5.1 Overview

5.2 Design Goals

5.3 Subsystem Decomposition

5.4 Hardware Software Mapping

5.5 Persistent Data Management

5.6 Access Control

5.7 Global Software Control

5.8 Boundry Conditions

6 Case Study / Evaluation

6.1 Design

6.2 Objectives

6.3 Results

6.4 Findings

6.5 Discussion

6.6 Limitations

7 Заключение

7.1 Status

7.1.1 Realized Goals

7.1.2 Open Goals

7.2 Conclusion

7.3 Future Work

List of Figures

Appendix A: Supplementary Material

– Supplementary Material –

Bibliography

[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.

[5] Marcus Aurelius, “Meditations.”