Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 22.Б10-мм

Поднятие двоичного кода в LLVM IR

Фролов Тимофей Сергеевич

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
ст. преподаватель кафедры ИАС Смирнов К. К.

Санкт-Петербург
2024

Оглавление

Введение

Поднятие кода — перевод кода, скомпилированного под конкретную архитектуру, в более высокоуровневое представление. Поднятие кода может использоваться для:

- анализа кода с использованием существующих методов;
- бинарной трансляции, которая в свою очередь может быть использована для запуска программ, скомпилированных под одну архитектуру, на процессорах другой архитектуры. Например, таким образом работает QEMU[1].

Различные инструменты могут поднимать код как статически, так и динамически.

Особенно полезна бинарная трансляция для архитектуры RISC-V: существует множество расширений[2] — и на конкретных процессорах могут быть реализована почти произвольная комбинация расширений, так что программа, скомпилированная под один конкретный процессор, на другом просто не запустится.

В данной работе рассматривается поднятие кода для бинарной трансляции из архитектуры RISC-V.

В качестве представления, в которое переводится скомпилированный код, может использоваться LLVM IR[3] — промежуточное платформонезависимое представление кода, используемое в LLVM. В данной работе будет рассматриваться инструменты поднятия кода, которые переводят код именно в это представление. Использование LLVM позволяет:

- оптимизировать код после поднятия с использованием существующих инструментов — как самим LLVM, так и его модифицированными версиями для устройств конкретных производителей;

[1]QEMU Documentation/TCGhttps://wiki.qemu.org/Documentation/TCG (дата обращения: 1 июня 2024 г.)

[2]RISC-V Ratified Extensions https://wiki.riscv.org/display/HOME/Ratified+Extensions (дата обращения: 1 июня 2024 г.)

[3]LLVM Language Reference Manual https://llvm.org/docs/LangRef.html (дата обращения: 1 июня 2024 г.)

- использовать существующие методы бинарного анализа, разработанные под LLVM:
  - Static analysis of energy consumption for LLVM IR programs.[4]
  - A static bugs analysis tool based on LLVM IR.[3]

Существует множество инструментов поднятия кода, использующих LLVM IR. Однако поддержка RISC-V в них или отсутствует, или поддерживается только rv64gc, что ограничивает область применимости этих инструментов.

1. Обзор существующих инструментов

В данной секции приведен обзор существующих инструментов поднятия кода до LLVM IR. В первую очередь анализируется поддержка архитектур, возможность рекомпиляции полученного кода, доступность, актуальность.

1.1. Remill+McSema

Remill — библиотека для поднятия бинарного кода, ориентированная на точности подъема инструкций. Remill поддерживает поднятие кода из архитектур AArch64, SPARC32, SPARC64, x86, amd64. На момент написания данной работы проект активно обновляется.[4]

McSema — бинарный транслятор, использующий Remill в качестве инструмента для поднятия кода до LLVM IR. Заархивирован в августе 2022 года.[5] Так же McSema использует IDA Pro для создания графа потока управления. IDA Pro является платным инструментом[6], что может затруднить использование McSema.

1.2. rev.ng

rev.ng — инструмент для поднятия бинарного кода. rev.ng использует QEMU для перевода ассемблерного кода в промежуточное представление QEMU, после чего переводит код из этого представления в LLVM IR. На момент написания данной работы проект активно обновляется.[7]

1.3. llvm-mctoll

llvm-mctoll — инструмент для поднятия бинарного кода, частично поддерживающий поднятие кода из архитектур x86-64 и arm32. Послед-

[4]Исходный код Remill https://github.com/lifting-bits/remill (дата обращения: 1 июня 2024 г.)

[5]Исходный код McSema https://github.com/lifting-bits/mcsema (дата обращения: 1 июня 2024 г.)

[6]IDA Pro – Hex Rays https://hex-rays.com/IDA-pro/ (дата обращения: 1 июня 2024 г.)

[7]Исходный код rev.ng https://github.com/revng/revng (дата обращения: 1 июня 2024 г.)

ний раз проект обновлялся в сентябре 2023 года.[8]

1.4. Rellume+Instrew

Rellume — инструмент для поднятия бинарного кода, ориентированный на генерацию производительного рекомпилируемого кода. Rellume поддерживает поднятие кода из архитектур x86-64, AArch64 и RISC-V64 (rv64imafdc). Последний раз проект обновлялся в апреле 2024 года.[9]

Instrew — динамический бинарный транслятор, использующий Rellume как инструмент для поднятия кода до LLVM IR для последующей рекомпиляции в целевую архитектуру. Последний раз проект обновлялся в апреле 2024 года. В качестве целевых архитектур instrew поддерживает x86-64 и AArch64.[10]

1.5. Результат обзора

Таблица 1: Существующие инструменты бинарной трансляции и поднятия кода

Название	Последняя активность	Рекомпилируемость поднятого кода	Поддержка RISC-V
remill	Март 2024	?	отсутствует
mcsema	заархивирован в апреле 2022	+	отсутствует
rev.ng	Апрель 2024	+	отсутствует
llvm-mctoll	сентябрь 2023	+	отсутствует
Rellume	январь 2024	+	rv64imafdc
instrew	январь 2024	+	как у Rellume

В таблице 1 можно посмотреть итоговое сравнение существующих инструментов поднятия кода.

Эта работа фокусируется на использовании поднятия кода в бинарной трансляции с упором на архитектуру RISC-V. Наиболее развитые инструменты в этом направлении — Instrew и Rellume, поэтому дальнейшая дискуссия будет вестись именно о них. В RISC-V есть множе-

[8]Исходный код llvm-mctoll https://github.com/microsoft/llvm-mctoll (дата обращения: 1 июня 2024 г.)

[9]Исходный код Rellume https://github.com/aengelke/rellume (дата обращения: 1 июня 2024 г.)

[10]Исходный код Instrew https://github.com/aengelke/instrew (дата обращения: 1 июня 2024 г.)

ство расширений[11], однако Rellume поддерживает только rv64imafdc, что ограничивает потенциальную область применимости. Одной из наиболее старых является группа расширений для побитовой манипуляции с числами, включающая в себя расширения Zba, Zbb, Zbc, Zbs. В рамках данной работы было решено добавить их поддержку в инструмент Rellume.

[11]RISC-V Ratified Extensions https://wiki.riscv.org/display/HOME/Ratified+Extensions (дата обращения: 1 июня 2024 г.)

2. Постановка задачи

Целью работы является добавление в Rellume[12] поддержки расширений Zba, Zbb, Zbc, Zbs архитектуры RISC-V. Для её выполнения были поставлены следующие задачи:

- Добавить поддержку декодирования инструкций в frvdec — декодер инструций RISC-V, который используется в Rellume.
- Для каждой из инструкций вышеупомянутых расширений добавить реализацию на языке LLVM IR.

[12]Исходный код Rellume https://github.com/aengelke/rellume (дата обращения: 1 июня 2024 г.)

3. Обзор Rellume

Rellume — инструмент для динамического поднятия бинарного кода, ориентированный на генерацию производительного рекомпилируемого кода. Rellume может использоваться как для бинарной трансляции, так и для других целей.

При поднятии бинарного кода существует два устоявшихся варианта выбора размера участков кода, которые поднимаются за раз: базовые блоки и суперблоки. Базовый блок (линейный участок) — последовательность инструкций, в которой нет условных и безусловных переходов. Суперблок — последовательность инструкций, в которой нет безусловных переходов. В отличие от базового блока, из-за ветвления, в суперблоке может быть несколько точек выхода, и суперблок не обязательно является непрерывным участком памяти. [2] Базовые блоки при поднятии кода могут разбиваться на несколько базовых блоков — для перевода определенных инструкций в LLVM необходимо использовать условные переходы и циклы. Rellume имеет возможность поднимать за раз как отдельные инструкции, так и базовые блоки и суперблоки. В листинге 1 можно посмотреть пример поднятия функции, состоящей из одного суперблока.

Многие существующие инструменты поднятия кода, в том числе Rellume, эмулируют состояние процессора: создаётся структура, содержащая все регистры процессора — и все операции в поднятом коде взаимодействуют с этой структурой: выгружают значения из регистров-аргументов и кладут значение в регистр-назначение.[1]

Для декодирования инструкций расширений x86-64, arm64 и RISC-V64 Rellume использует проекты fadec[13], farmdec[14] и frvdec[15] соответственно.

Часть задач по поднятию кода Rellume не реализует, а предполагает, что их реализует инструмент, который использует Rellume. В числе таких задач реализация системных вызовов и реализация инструкции

[13]Исходный код fadec https://github.com/aengelke/fadec/ (дата обращения: 1 июня 2024 г.)
[14]Исходный код farmdec https://github.com/okitec/farmdec/ (дата обращения: 1 июня 2024 г.)
[15]Исходный код frvdec https://git.sr.ht/~aengelke/frvdec (дата обращения: 1 июня 2024 г.)

cpuid для архитектуры x86-64. На рис. 1 можно видеть взаимосвязь Rellume с его зависимостями и существующими инструментами на его основе, а на рис. 2 можно посмотреть пример взаимодействия Instrew с Rellume.

Листинг 1: Пример поднятия функции, складывающей два числа

Исходный LLVM IR:

```
define noundef i64 @f(i64 noundef %0, i64 noundef %1) {
  %3 = add nsw i64 %1, %0
  ret i64 %3
}
```

Скомпилированный в RISC-V:

```
f:
  add  a0,a0,a1
  ret
```

Поднятый суперблок:

```
define void @S0_100b0(ptr addrspace(1) align 16 dereferenceable(400) %0) {
  %2 = getelementptr inbounds i8, ptr addrspace(1) %0, i64 16 ; $ra register
  %3 = load i64, ptr addrspace(1) %2, align 16
  %4 = getelementptr inbounds i8, ptr addrspace(1) %0, i64 88 ; $a0 register
  %5 = load i64, ptr addrspace(1) %4, align 8
  %6 = getelementptr inbounds i8, ptr addrspace(1) %0, i64 96 ; $a1 register
  %7 = load i64, ptr addrspace(1) %6, align 16
  %8 = add i64 %5, %7 ; $a0 + $a1
  store i64 %3, ptr addrspace(1) %0, align 16 ; store $ra into $pc
  store i64 %8, ptr addrspace(1) %4, align 8 ; store $a0 + $a1 into $a0
  ret void
}
```

Рис. 1: Rellume, его зависимости и инструменты, которые его используют

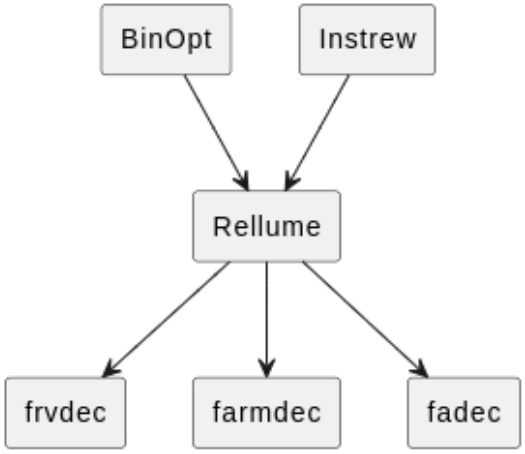
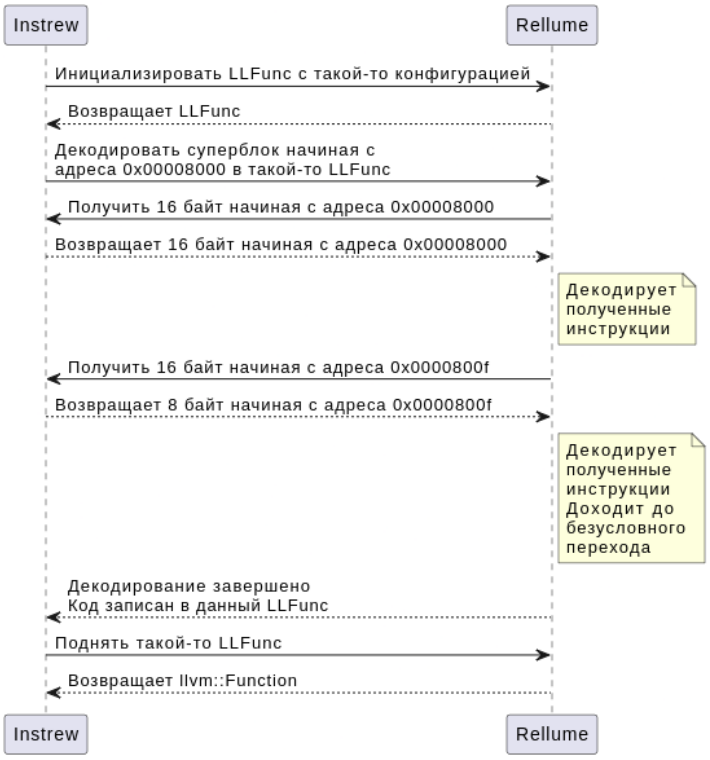


Рис. 2: Пример взаимодействия Rellume с Instrew



4. Реализация поднятия инструкций

4.1. Декодирование инструкций

Чтобы добавить поддержку инструкций в Rellume, для начала необходимо добавить их поддержку в frvdec — декодер инструкций RISC-V64, который используется в Rellume.

Описание формата всех инструкций из расширений Zba, Zbb, Zbc и Zbs находится в официальной спецификации[16].

Для каждой инструкции из расширений Zba, Zbb, Zbc, Zbs добавлена поддержка декодирования[17], полный список которых можно посмотреть в таблицах 2 и 3.

4.2. Поднятие инструкций

В данной работе поднятие инструкции производится по следующим шагам:

1. Посмотреть, есть ли в LLVM инструкция или интринсик, который делает то же, что и поднимаемая инструкция.
2. Если есть — использовать соответствующую инструкцию или интринсик.
3. Если нет — воспроизвести семантику инструкции в терминах LLVM.

В качестве примеров рассмотрим инструкции `clz rd rs`, `shNadd rd rs1 rs2` и `clmul rd rs1 rs2`.

4.2.1. clz

`clz rd rs`: посчитать лидирующие нули в регистре rs, записать в rd. Если rs = 0 — в rd записывается 64 (длина регистра).

[16] RISC-V Bitmanip specification https://github.com/riscv/riscv-bitmanip/releases/tag/1.0.0 (дата обращения: 1 июня 2024 г.)

[17] https://git.sr.ht/~timafrolov/frvdec/log/bitmanip

В LLVM есть интринсик, который считает лидирующие нули в аргументе. Если src = 0 — возвращается poison value, если is_zero_poison = 1, иначе возвращается 64.

```
declare i64 @llvm.ctlz.i64(i64 %src, i1 %is_zero_poison)
```

Итого, переводим `clz rd rs` в

```
StoreGp(rd, CreateBinaryIntrinsic(llvm::Intrinsic::ctlz,
                                  LoadGp(rs1), 0))
```

4.2.2. shNadd

`shNadd rd rs1 rs2`: `rd = rs2 + (rs1 << N)`. Соответствующей инструкции в LLVM нет, поэтому переводим в

```
shifted = CreateShl(LoadGp(rs2), N);
StoreGp(rd, CreateBinOp(llvm::Instruction::Add,
                        LoadGp(rs1), shifted));
```

4.2.3. clmul

`clmul rd rs1 rs2`: посчитать умножение без переноса rs1 на rs2, результат записать в rd. Определение инструкции, приведенное в стандарте расширений bitmanip, можно посмотреть в листинге 2(TOASK: ”умножение без переноса” - это нормально, или есть более адекватный перевод?)

Для получения реализацию на LLVM IR, псевдокод переведён на C и скомпилирован в LLVM IR с использованием clang. Полученный код можно посмотреть в листинге 3. Итоговую реализацию генерации кода для clmul можно посмотреть в листинге 4

Аналогичным образом были переведены все инструкции из расширений Zba, Zbb, Zbc, Zbs[18], полный список которых можно посмотреть в таблицах 2 и 3.

[18] https://github.com/aengelke/rellume/pull/14

Листинг 2: Псевдокод, эквивалентный инструкции clmul. Источник: https://github.com/riscv/riscv-bitmanip/releases/tag/1.0.0

```
let rs2_val = X(rs2);
let rs1_val = X(rs1);
let output : xlenbits = 0;
foreach (i from 0 to xlen by 1) {
  output = if ((rs2_val >> i) & 1)
  then output ^ (rs1_val << i);
  else output;
}
X[rd] = output
```

Листинг 3: Реализация clmul на C, скомпилированная в LLVM IR

Реализация на C:

```c
uint64_t clmul(uint64_t rs1, uint64_t rs2) {
  uint64_t output = 0;
  for (uint64_t i = 0; i < 64; i++) {
    output = ((rs2 >> i) & 1) ? (output ^ (rs1 << i)) : output;
  }
  return output;
}
```

Полученный код на LLVM IR:

```
define noundef i64 clmul(i64 noundef %0, i64 noundef %1) {
  br label %4

3:                                                ; preds = %4
  ret i64 %12

4:                                                ; preds = %2, %4
  %5 = phi i64 [ 0, %2 ], [ %13, %4 ]
  %6 = phi i64 [ 0, %2 ], [ %12, %4 ]
  %7 = shl nuw i64 1, %5
  %8 = and i64 %7, %1
  %9 = icmp eq i64 %8, 0
  %10 = shl i64 %0, %5
  %11 = select i1 %9, i64 0, i64 %10
  %12 = xor i64 %11, %6
  %13 = add nuw nsw i64 %5, 1
  %14 = icmp eq i64 %13, 64
  br i1 %14, label %3, label %4, !llvm.loop !6
}
```

Листинг 4: Итоговая реализация инструкции clmul

```
rs1_val = LoadGp(rs1);
rs2_val = LoadGp(rs2);

// Получаем текущий базовый блок
enter_block = GetInsertBlock();

loop_block = AddBlock();
continuation_block = AddBlock();

// В конце текущего базового блока переходим на тело цикла.
enter_block->BranchTo(*loop_block);
// Начиная с этого момента добавляем инструкции в тело цикла.
SetInsertBlock(loop_block);

output = CreatePHI();
i = CreatePHI();

output->addIncoming(0, *enter_block);
i->addIncoming(0, *enter_block);

condition =
  irb.CreateICmpEQ(irb.CreateAnd(irb.CreateShl(1,i), rs2), 0);
next_output =
  irb.CreateXor(res, irb.CreateSelect(condition,
                                      irb.getInt64(0),
                                      irb.CreateShl(rs1, i)));
next_i = CreateAdd(i, 1);
loop_block->BranchTo(CreateICmpEQ(next_i, 64),
                     *continuation_block,
                     *loop_block);

output->addIncoming(next_output, *loop_block);
i->addIncoming(next_i, *loop_block);

// После исполнения инструкции переходим в новый базовый блок.
SetInsertBlock(cont_block);
StoreGp(rd, next_output);
```

Таблица 2: Добавленные инструкции, часть 1

Инструкция	Расширение	Есть ли интринсик в LLVM
add.uw	Zba	-
andn	Zbb	-
clz	Zbb	+
clzw	Zbb	-
cpop	Zbb	+
cpopw	Zbb	-
ctz	Zbb	+
ctzw	Zbb	-
max	Zbb	+
maxu	Zbb	+
min	Zbb	+
minu	Zbb	+
orc.b	Zbb	-
orn	Zbb	-
rev8	Zbb	+
rol	Zbb	+
rolw	Zbb	-
ror	Zbb	+
rori	Zbb	+
roriw	Zbb	-
rorw	Zbb	-
sext.b	Zbb	+
sext.h	Zbb	+
sh1add	Zba	-
sh1add.uw	Zba	-
sh2add	Zba	-
sh2add.uw	Zba	-
sh3add	Zba	-
sh3add.uw	Zba	-
slli.uw	Zba	-
xnor	Zbb	-
zext.h	Zbb	+

Таблица 3: Добавленные инструкции, часть 2

Инструкция	Расширение	Есть ли интринсик в LLVM
bclr	Zbs	-
bclri	Zbs	-
bext	Zbs	-
bexti	Zbs	-
binv	Zbs	-
binvi	Zbs	-
bset	Zbs	-
bseti	Zbs	-
clmul	Zbc	-
clmulh	Zbc	-
clmulr	Zbc	-

Заключение

- Проведён обзор существующих инструментов поднятия кода в LLVM IR, выбран инструмент, который генерирует рекомпилируемый код, с наиболее развитой поддержкой архитектуры RISC-V.
- Добавлена поддержка декодирования инструкций расширений Zba, Zbb, Zbc, Zbs в декодер RISC-V инструкций frvdec[19].
- Для каждой инструкции из расширений Zba, Zbb, Zbc, Zbs добавлена реализация на языке LLVM IR в инструмент поднятия кода Rellume. Сделан Pull Request в репозиторий Rellume[20].

Продолжение работы

В данной работе была добавлена поддержка расширений Zba, Zbb, Zbc и Zbs в Rellume. В дальнейшем можно продолжать добавлять поддержку других популярных расширений, например, векторных инструкций[21]

Также в данный момент Instrew — бинарный транслятор, использующий Rellume — имеет возможность запускаться только на архитектурах x86-64 и arm64. В дальнейшем можно добавить поддержку запуска Instrew на RISC-V64

[19] https://git.sr.ht/~timafrolov/frvdec/log/bitmanip

[20] https://github.com/aengelke/rellume/pull/14

[21] RISC-V Vector Extension 1.0 https://github.com/riscv/riscv-v-spec/releases/tag/v1.0 (дата обращения: 1 июня 2024 г.)

Список литературы

[1] 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. — URL: https://doi.org/10.1145/3381052.3381319 (дата обращения: 31 мая 2024 г.).

[2] Engelke Alexis Friedrich. Optimizing performance using dynamic code generation : Ph. D. thesis / Alexis Friedrich Engelke ; Technische Universität München. — 2021. — URL: https://mediatum.ub.tum.de/doc/1614897/1614897.pdf (дата обращения: 6 мая 2024 г.).

[3] MLSA: A static bugs analysis tool based on LLVM IR / Hongliang Liang, Lei Wang, Dongyang Wu, Jiuyun Xu // 2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD). — 2016.

[4] Static analysis of energy consumption for LLVM IR programs / Neville Grech, Kyriakos Georgiou, James Pallister et al. // Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems. — SCOPES '15. — New York, NY, USA : Association for Computing Machinery, 2015. — URL: https://doi.org/10.1145/2764967.2764974 (дата обращения: 31 мая 2024 г.).