Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 21.Б11-мм

Поддержка RISC-V Bitmanip расширения в компиляторе clang

Калашников Матвей Михайлович

Отчёт по учебной практике
в форме «Производственное задание»

Научный руководитель:
старший преподаватель каф. ИАС К.К. Смирнов

Санкт-Петербург
2024

Оглавление

Введение

RISC-V – это открытая, расширяемая, модульная ISA, которая активно развивается сообществом разработчиков по всему миру. Благодаря своим преимуществам и открытости RISC-V привлекла к себе инвестиции множества государств и крупных компаний. По данным международной организации RISC-V International [7], 3 950 членов RISC-V из 70 стран вносят свой вклад и сотрудничают в составлении открытых спецификаций RISC-V. В сентябре 2022 года в России был создан Альянс RISC-V [10], а в декабре 2022 года Европейский союз выделил 270 миллионов евро на создание RISC-V аппаратного и программного обеспечения [2]. Одним из преимуществ RISC-V является расширяемость и модульность. RISC-V ISA состоит из базовых наборов команд и модульных расширений ISA, а также явно поддерживает пользовательские расширения, специфичные для конкретной области работы.

Существует множество расширений для удовлетворения различных вычислительных потребностей. Организация по стандартизации RISC-V постоянно внедряет новые расширения наборов команд для расширения функциональных возможностей RISC-V, такие как V-расширение для векторных вычислений или B-расширение для манипуляций с битами [6].

Расширение B, или bit manipulation instruction-set extension позволяет значительно повысить скорость выполнения задач, которые зависят от операций на уровне отдельных битов. Такие операции часто применяются в задачах криптографии, сжатия, обработки изображений и при работе с сетью. Один из разработчиков процессоров на архитектуре RISC-V, компания SiFive, пишет о 35% улучшении производительности алгоритма криптографического хэширования при использовании расширения Bitmanip [8].

Однако добавление расширения не гарантирует улучшения производительности. В ходе летней школы Матмеха СПБГУ было выяснено, что компиляторы не всегда распознают шаблоны кода, которые могут быть заменены на инструкции из расширения [11]. Некоторые инструк-

ции используются только если явно обратиться к ним используя встроенные функции, которые по особому обрабатываются компилятором. Для достижения максимальной производительности необходимо либо вручную оптимизировать код, используя инструкции Bitmanip, либо улучшить компилятор, чтобы он использовал инструкции расширения.

Одна из инструкций, которая не всегда может использоваться компилятором, это инструкция подсчета конечных нулей в бинарном представлении числа или ctz(count trailing zeros). Для решения этой проблемы необходимо разработать оптимизацию, позволяющую распознать соответствующий паттерн кода.

1. Постановка задачи

Целью работы является улучшение распознавания компилятором clang шаблонов кода, соответствующих инструкциям расширения Bitmanip.

1. Выполнить обзор возможностей компилятора clang по оптимизации кода и изучить существующие оптимизации.

2. Реализовать оптимизацию, позволяющую распознавать идиому подсчета конечных нулей.

2. Обзор технологий

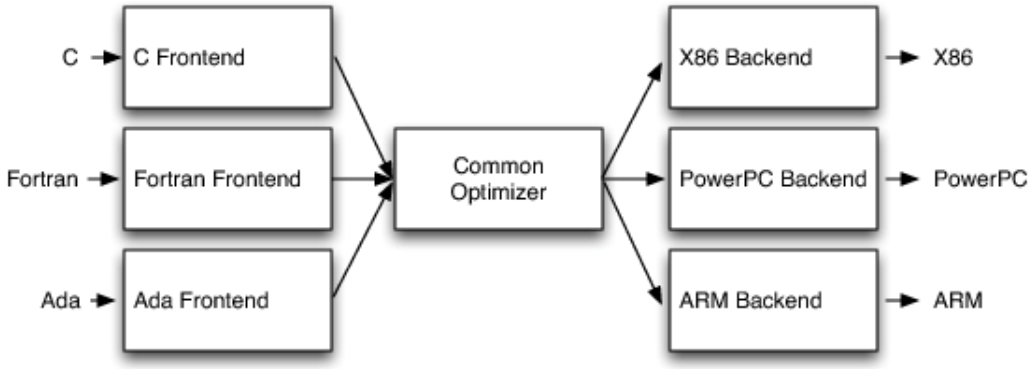
Открытый и модульный характер архитектуры RISC-V позволяет использовать широкий спектр расширений, дополняющих базовый набор команд. Базовый набор содержит только команды для целочисленных операций (например, сложения и вычитания). Расширения предоставляют специализированные инструкции для различных задач. Они делятся на стандартные и пользовательские. Стандартные расширения разрабатываются RISC-V International и предназначены для расширения функциональных возможностей архитектуры. Туда входят расширения для умножения и деления, работы с числами с плавающей запятой и пр. [6]. Также некоторые организации или исследователи могут самостоятельно разрабатывать расширения инструкций, для узкоспециализированных задач. Например для искусственного интеллекта или для интернета вещей.

Расширение B (Bit Manipulation Extension) относится к категории стандартных расширений. Оно добавляет набор инструкций, предназначенных для ускорения операций на уровне битов, таких как подсчет битов, битовые сдвиги, перестановки, вставки и пр. Оно делится на несколько небольших расширений, сгруппированным по общим функциям и случаям использования: zba, zbb, zbc и zbs.

Расширение zbb содержит логические инструкции с отрицанием, подсчет конечных/лидирующих нулей, подсчет количества единичных битов, циклические сдвиги и др. [5]. Именно в него входит инструкция ctz.

2.1. Обзор оптимизаций

Clang – это один из ведущих компиляторов с открытым исходным кодом, предназначенный для языков C, C++, Objective-C и смежных. Одной из сильных сторон clang является модульная конструкция, построенная на базе фреймворка LLVM. LLVM обеспечивает инфраструктуру для промежуточного представления (IR) кода и различных этапов оптимизации и генерации кода, что делает clang очень гибким.



Clang использует трехфазный дизайн компилятора. Фронтенд разбирает исходный код, выполняет проверку ошибок и строит AST. После этого AST преобразуется в LLVM IR к которому впоследствии применяются оптимизации. В итоге бэкэнд транслирует оптимизированный LLVM IR в машинный код [3]. Стоит обратить внимание на этапы оптимизации и кодогенерации. На этих этапах мы можем распознавать определенные паттерны кода и оптимизировать их.

2.1.1. Этап кодогенерации

На этапе кодогенерации LLVM IR трансформируется в инструкции, предназначенные для определенной архитектуры. Здесь появляются возможности для использования специфических для платформы инструкций для повышения производительности сгенерированного кода.

Для детального описания целевой архитектуры в LLVM используется такой инструмент как TableGen [9]. TableGen — это предметно-ориентированный язык, с помощью которого определяется наборов команд процессора, правила их выбора и другая платформо-специфичная информация. Каждая инструкция, относящаяся к целевой архитектуре, определяется с указанием ее имени, операндов и формата кодирования. Также TableGen поддерживает правила сопоставления шаблонов. Генератор кода сравнивает фрагменты LLVM IR с этими шаблонами, чтобы определить, можно ли заменить некоторую последовательность инструкций на специфичную для платформы инструкцию. Сопоставление шаблонов не подходит для сложных преобразований из-за того, что инфраструктура SelectionDAG работает только в пределах одного

базового блока. Таким образом этот инструмент не подходит для распознавания идиомы подсчета конечных нулей.

2.1.2. Этап оптимизаций

В основе конвейера оптимизации LLVM лежат этапы оптимизации, которые делятся на анализирующие, трансформирующие и утилитарные. Этапы анализа исследуют код, не изменяя его, предоставляя информацию, используемую на последующих этапах оптимизации. Этапы преобразования напрямую изменяют IR для повышения эффективности, уменьшения размера кода или оптимизации использования памяти. Утилитарные этапы помогают в этом процессе, обеспечивая верификацию кода и пр.

Оптимизирующие проходы могут работать сразу с несколькими базовыми блоками. Для распознавания шаблона подсчета конечных нулей необходимо использовать трансформирующий проход. loop-idiom – это один из проходов, который выполняет оптимизацию циклов. Там уже реализованы некоторые шаблоны, которые преобразуются в LLVM интринсики ctlz, cttz, ctpop (count leading zeros и count trailing zeros соответственно). LLVM интринсики – это встроенные функции с задокументированной семантикой. Такую функцию можно только вызвать и нельзя определить. Таким образом будет правильнее всего добавить распознавание нового шаблона именно в проход loop-idiom.

3. Описание решения

В данном разделе представлены методы, которые необходимы для разработки оптимизационного прохода LLVM. Здесь описан процесс сборки LLVM и clang, метод сборки и тестирования оптимизационных проходов вне дерева исходного кода LLVM, а также описана реализация оптимизации loop-ctz-idiom.

3.1. Сборка

LLVM – это большой проект с миллионами строк кода. Он состоит из множества инструментов и модулей. Конфигурация для сборки проекта строится с помощью инструмента CMake, который также позволяет задать множество дополнительных опций [4]. Для сборки LLVM и clang использовалась седующая команда:

```
cmake -G Ninja -S ~/llvm-project/llvm -B ~/llvm_linux/ \
-DLLVM_ENABLE_PROJECTS=clang -DLLVM_TARGETS_TO_BUILD=RISCV \
-DCMAKE_BUILD_TYPE=DEBUG -DBUILD_SHARED_LIBS=ON \
-DLLVM_PARALLEL_LINK_JOBS=1
```

Первые два аргумента задают папку с исходным кодом и папку для артефактов сборки. Следующая опция указывает, что вместе с LLVM необходимо собрать clang. Опция LLVM_TARGETS_TO_BUILD позволяет выбрать целевые платформы, для которых компилятор сможет сможет генерировать код. Это значительно ускоряет процесс сборки, так как не нужно компилировать код для всех возможных целевых платформ. Опция BUILD_TYPE=DEBUG отключает все оптимизации при сборке. Это необходимо для разработки LLVM, так как оптимизированный код может быть сложнее отлаживать. Сборка разделяемых библиотек сокращает объем артефактов сборки, а ограничение параллельной компоновки необходимо, чтобы сократить использование оперативной памяти.

3.2. Разработка вне исходного кода LLVM

Так как сборка даже части LLVM занимает довольно много времени, необходимо было сократить время сборки, чтобы упростить разработку и отладку. Разработчики LLVM предоставляют возможность собирать и тестировать проходы оптимизации вне дерева исходного кода LLVM [1]. Это позволяет компилировать только код одного оптимизационного прохода. Далее с помощью инструмента opt можно применить этот оптимизационный проход к LLVM IR. Для этого использовалась следующая команда:

```
opt -load-pass-plugin libLoopIdiomCtz.so \
-p loop-ctz-idiom  input.ll \
-S -print-module-scope > output.ll
```

Она применяет оптимзационный проход loop-ctz-idiom из libLoopIdiomCtz.so к файлу input.ll. Чтобы вывести полный LLVM IR модуля используются флаги -S и -print-module-scope. Инструмент opt дает широкие возможности для применения оптимизаций по отдельности и отладки оптимизационных проходов.

3.3. Реализация оптимизации

В LLVM есть множество проходов, которые преобразуют и упрощают циклы. loop-idiom – это один из таких проходов. Он заменяет некоторые шаблоны циклов, которые копируют или заполняют массив, на инструкции memcpy/memset. Также некоторые циклы заменяются на код с использованием интринсиков cttz/ctlz/ctpop.Но этот проход распознает не все паттерны, которые можно оптимизировать. Вот пример одного из паттернов, которые компилятор не распознает:

```
int count_trailing_zeroes(uint32_t n) {
    int count = 0;

    if (n == 0){
        return 32;
```

```
    }
    while ((n & 1) == 0) {
        count ++;
        n >>= 1;
    }

    return count;
}
```

Это одна из возможных реализации функции подсчета конечных нулей. В этом случае оптимизация может заменить весь цикл на один интринсик cttz.

Перед проходом loop-idiom, проход LoopSimplify создает базовый блок while.body.preheader перед телом цикла, и базовый блок cleanup.loopexit после тела цикла.

Решение представляет собой несколько функций, интегрированных в проход loop-idiom. Сначала проверяется существование самого тела цикла. После этого проверяется существование проверки на 0. При ее отсутствии, после передачи в функцию нуля, цикл будет выполняться бесконечно, при этом если заменить его на на интринсик будет возвращен размер int. После выполнения этих проверок в базовый блок while.body.preheader записывается интринсик cttz. Если значение переменной count не равно нулю результат интринсика прибавляется к ней. Все использования переменной count после цикла заменяются на новую переменную. Сам цикл удалять необязательно, так как он будет удален позже loop-deletion проходом.

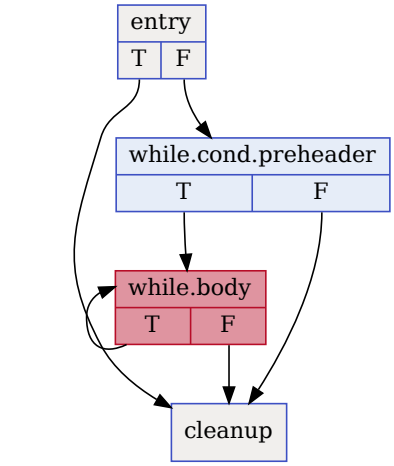


Рис. 1: CFG функции перед оптимизацией

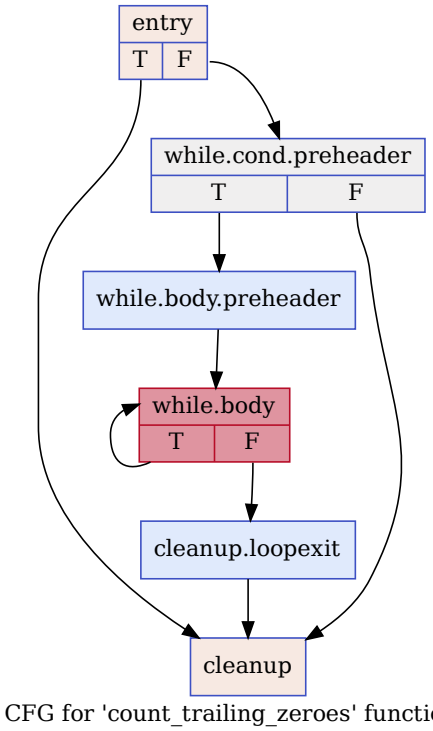


Рис. 2: CFG функции после оптимизацией

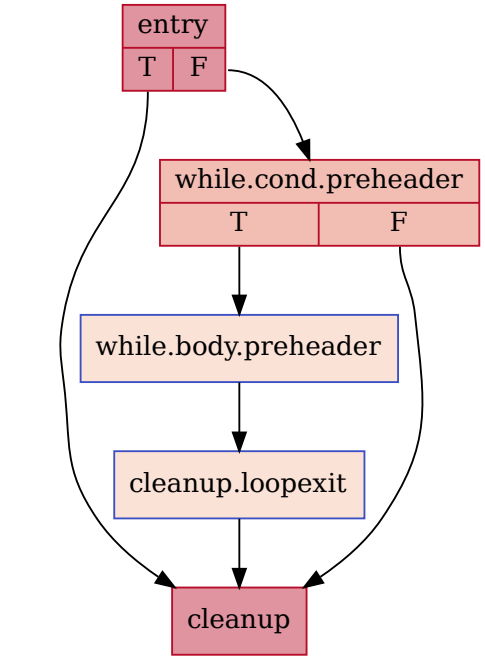


Рис. 3: CFG функции после удаления цикла

4. Распознаваемые паттерны циклов

Реализованный оптимизационный проход позволяет распознавать следующие паттерны кода и заменять тело цикла на интринсик. На примере первой функции покажем как происходит оптимизация.

```
int ctz(uint32_t n)
{
    int count = 0;
    if (n == 0)
    {
        return 32;
    }
    while ((n & 1) == 0)
    {
        count += 1;
        n >>= 1;
    }
    return count;
}
```

```
int ctz(uint64_t n)
{
    int count = 0;
    if (n != 0)
    {
        while ((n & 1) == 0)
        {
            n >>= 1;
            count += 1;
        }
    }
    else
    {
        return 64;
    }
    return count;
}
```

Сначала с помощью следующей команды сгенерируем начальный LLVM IR. Флаги -Xclang и -disable-llvm-passes позволяют полностью отключить оптимизации, несмотря на то что в команде присутствует флаг -O3. Указать флаг оптимизации необходимо, чтобы эти оптимизации выполнились в будущем. Также важно заметить, что эта оптимизация выполняется не только на уровне O3, а еще и на уровнях O1, O2 и Os.

```
clang example.c --target=riscv64  -march=rv64gc_zbb \
-emit-llvm -S -O3 -Xclang  -disable-llvm-passes \
-target gcc-riscv64-linux-gnu -o example.ll
```

После этого с помощью инструмента opt можно вывести LLVM IR, как до, так и после оптимизации. Покажем как выглядит LLVM IR функции подсчета конечных нулей перед оптимизацией, с помощью следующей команды:

```
opt -print-before=loop-idiom -print-module-scope -O3 -S example.ll
2> examplebefore.ll
```

```
i32 @count_trailing_zeroes(i32 noundef signext %n){
entry:
  %cmp = icmp eq i32 %n, 0
  br i1 %cmp, label %cleanup, label %while.cond.preheader

while.cond.preheader:
  %and4 = and i32 %n, 1
  %cmp15 = icmp eq i32 %and4, 0
  br i1 %cmp15, label %while.body.preheader, label %cleanup

while.body.preheader:
  ; После оптимизации в этом месте вызывается интринсик
  ; %countres = call i32 @llvm.cttz.i32(i32 %n, i1 true)
  br label %while.body

while.body:
  %count.07 = phi i32 [ %add, %while.body ], [ 0, %while.body.preheader ]
  %n.addr.06 = phi i32 [ %shr, %while.body ], [ %n, %while.body.preheader ]
  %add = add nuw nsw i32 %count.07, 1
  %shr = lshr exact i32 %n.addr.06, 1
  %0 = and i32 %n.addr.06, 2
  %cmp1 = icmp eq i32 %0, 0
  br i1 %cmp1, label %while.body, label %cleanup.loopexit

cleanup.loopexit:                                 ; preds = %while.body
  ; После оптимизации узел Phi изменяется и вместо счетчика %add в нем начинает использоваться
  ; результат вычисления cttz.
  ; %add.lcssa = phi i32 [ %countres, %while.body ]
  %add.lcssa = phi i32 [ %add, %while.body ]
  br label %cleanup

cleanup:
  %retval.0 = phi i32 [32, %entry], [0, %while.cond.preheader], [%add.lcssa, %cleanup.loopexit]
  ret i32 %retval.0
}
```

После удаления неиспользуемого цикла и всех остальных оптимизаций LLVM IR этой функции выглядит следующим образом:

```
define dso_local signext i32 @count_trailing_zeroes(i32 noundef signext %n) local_unnamed_addr #0
entry:
  %cmp = icmp eq i32 %n, 0
```

```
  %and4 = and i32 %n, 1
  %cmp15 = icmp eq i32 %and4, 0
  %0 = tail call i32 @llvm.cttz.i32(i32 %n, i1 true)
  %spec.select = select i1 %cmp15, i32 %0, i32 0
  %retval.0 = select i1 %cmp, i32 32, i32 %spec.select
  ret i32 %retval.0
}
```

Кроме вызова интринсика здесь сохранены инструкции которые отвечают за условие перед циклом и за предусловие самого цикла. В итоге тот код скомпилируется в следующий ассемблерный код для архитектуры RISC-V:

```
0000000000000000 <count_trailing_zeroes>:
       0:  01 e1          bnez    a0, 0x0 <count_trailing_zeroes>
       2:  13 05 00 02    li      a0, 32
       6:  82 80          ret

0000000000000008 <.LBB0_2>:
       8:  93 75 15 00    andi    a1, a0, 1
       c:  1b 15 15 60    ctzw    a0, a0
      10:  fd 15          addi    a1, a1, -1
      12:  6d 8d          and     a0, a0, a1
      14:  82 80          ret
```

Этот ассемблерный код содержит необходимую инструкцию ctzw. Суффикс w в данном случае означает, что инструкция работает только с младшими 32 битами входа. Инструкции andi, addi и and остались от предусловия цикла и в данном случае не изменяют результат подсчета конечных нулей.

Заключение

В результате работы над учебной практикой в течение зимнего семестра были выполнены следующие задачи:

- Проведен обзор возможностей clang по оптимизации а также изучен соответствующий проход.
- Реализована оптимизация, распознающая один из шаблонов подсчета конечных нулей, и заменяющая его на соответствующую инструкцию.

Код находится в следующих репозиториях.

- out of tree pass [12].
- Реализация интегрированая в LLVM [13].

Список литературы

[1] Github with out of tree pass tutorial. — URL: https://github.com/banach-space/llvm-tutor (дата обращения: 12 марта 2024 г.).

[2] HPC writing. EC выделил средства на создание чипов RISC-V. — URL: https://www.hpcwire.com/2022/12/16/europe-to-dish-out-e270-million-to-build-risc-v-hardware-and-so (дата обращения: 12 марта 2024 г.).

[3] LLVM architecture. — URL: https://aosabook.org/en/v1/llvm.html (дата обращения: 25 февраля 2024 г.).

[4] LLVM building. — URL: https://llvm.org/docs/CMake.html (дата обращения: 12 марта 2024 г.).

[5] RISC-V Bitmanip public review release. — URL: https://github.com/riscv/riscv-bitmanip/releases/tag/1.0.0 (дата обращения: 12 марта 2024 г.).

[6] RISC-V Instruction Set Architecture Extensions: A Survey. — URL: https://ieeexplore.ieee.org/document/10049118 (дата обращения: 12 марта 2024 г.).

[7] RISC-V International. — URL: https://riscv.org/about/ (дата обращения: 2 марта 2024 г.).

[8] SiFive Core IP 21G1. — URL: https://www.sifive.com/blog/sifive-core-ip-21g1 (дата обращения: 25 февраля 2024 г.).

[9] TableGen Overview. — URL: https://llvm.org/docs/TableGen/index.html (дата обращения: 26 февраля 2024 г.).

[10] Альянс RISC-V в России. — URL: https://riscv-alliance.ru/ (дата обращения: 12 марта 2024 г.).

[11] Пример результатов работы инструмента, по определению инструкций поддерживаемых компилятором. — URL: https:

//github.com/vacmannnn/riscv-check/blob/main/examples/clang17_0_4.csv (дата обращения: 25 февраля 2024 г.).

[12] Реализация оптимизации out of tree. — URL: https://github.com/foreverjun/llvm_cttz_pass/tree/ctz-pass (дата обращения: 2 марта 2024 г.).

[13] Реализация оптимизации интегрированная в LLVM. — URL: https://github.com/foreverjun/llvm-project (дата обращения: 2 марта 2024 г.).