Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 21.Б07-мм

Интринсики RISC-V для OCaml

Габдрахманов Азат Райнурович

Отчёт по учебной практике
в форме «Эксперимент»

Научный руководитель:
ассистент кафедры системного программирования Косарев Д.С.

Санкт-Петербург
2024

Оглавление

Введение

Особенности дизайна языков программирования сильно влияют на сферы применения программ, написанных на них. Принято делить языки программирования на языки более низкого уровня, например, C, которые близки к железу и позволяют использовать низкоуровневые оптимизации, и языки более высокого уровня, на которых проще и быстрее получить корректно работающие программы, при сравнимом уровне разработчика.

Идея этого проекта навеяна разработками компании Janestreet[1], которая занимается в том числе высокочастотчным трейдингом ценных бумаг. Для своих нужд компания Janestreet использует язык программирования OCaml и пытается избегать ошибок с его помощью. К сожалению, как и у многих языков высокого уровня, это вызывает некоторую деградацию производительности. Janesteet пытаются решать эту проблему путём модернизации компилятора в целом, а в частности поддержкой в библиотеке языка интринсиков для архитектур AMD64 и ARM64, а также оптимизаций к ним.

Изучив их проект[2], появилась идея сделать аналогичные интрисики в компиляторе OCaml, но для архитектуры RISC-V. Задуман поход, при котором программист будет вызывать из OCaml внешние (англ. external) функции с соответствующими именами. Когда компилятор будет встречать внешние вызовы, которые он знает, то будут вставляться соответствующие инструкции RISC-V. В противном случае компилятор будет полагаться, что реализация этих функций возьмется из других объектных файлов в процессе линковки.

В данном проекте планируется внедрить интринсики в компилятор нативного кода — ocamlopt. Для внедрения интринсиков будет анализироваться промежуточное представление Cmm , которое является последним перед генерацией ассемблера. Именно на его основе будет изменяться ассемблерный файл.

[1] https://www.janestreet.com/technology/

[2] https://github.com/ocaml-flambda/flambda-backend

1. Постановка задачи

Целью работы является реализация интринсиков для компилятора OCaml под архитектуру RISC-V.

Для ее выполнения были поставлены следующие задачи:

1. Выбрать и реализовать простые интринсики.
2. Сравнить размер ассемблерного кода и производительность инструкций.

2. Обзор инструментов

2.1. RISC-V

Ныне набирает популярность RISC-V — расширяемая открытая и свободная система команд и процессорная архитектура на основе концепции RISC[3]. Философия RISC (Reduced Instruction Set Computer - компьютер с сокращенным набором команд) заключается в том, что по сравнению с инструкциями CISC (Complex Instruction Set Computer - компьютер со сложным набором команд), инструкции RISC написаны более простым кодом, менее сложные. Неправильной интерпретацией слова «сокращенный» (reduced) в данном случае будет идея, что сокращено *количество* инструкций, но на самом деле это слово относится к *сложности* инструкции.

Почему RISC приходит на замену CISC, какие есть преимущества и недостатки каждого из подходов? В CISC процессорах есть множество сложных инструкций, с помощью которых можно поддерживать высокоуровневые конструкции. Ранее это было удобно по нескольким причинам:

- повышение продуктивности работы на ассемблере;
- в связи с медленной и дорогой памятью, лучше использовать сложные инстукции, которые делают много действий и минимально используют память;
- компактный размер кода.

Но есть и недостатки данного подхода

- меньшее количество регистров общего назначения, поскольку для команд декодирования требуется больше транзисторов;
- требуется несколько тактов для выполнения одной инструкции, несмотря на минимальный размер кода;

[3] https://ru.wikipedia.org/wiki/RISC-V

- реализация обходится дороже (требуется больше транзисторов).

В RISC, в свою очередь, инструкции проще, из-за чего может их может потребоваться больше. Цель состоит в том, чтобы компенсировать необходимость обработки большего количества команд за счет увеличения скорости выполнения каждой команды, в частности, за счет реализации конвейера команд, чего может быть проще достичь при наличии более простых команд[4].

2.1.1. Система команд RISC-V

Система команд RISC-V имеет модульную структуру и является расширяемой [3]. Базовый набор инструкций является довольно компактным: всего 11 базовых арифметических инструкций (большинство из них могут встречаться в 2-х формах, давая в сумме 21 инструкцию), 10 инструкций для обращения в память и 8 инструкций для переходов. Итого — 39 инструкций. Большим плюсом является наличие 32 регистров.

2.1.2. Расширения RISC-V

Существует большое многообразие расширений RISC-V Расширения могут представлять из себя относительно сложные операции, по сравнению с базовыми, примером может служить сложение со сдвигом.

В данной работе для реализации интринсиков используются расширения для манипуляции с битами[5]:

- «zbb»[6]
- «xtheadba»[7]

В следующем семестре планируется использовать векторные расширения.

[4] https://en.wikipedia.org/wiki/Reduced_instruction_set_computer
[5] https://github.com/riscv/riscv-bitmanip
[6] https://github.com/riscv/riscv-bitmanip/blob/main/bitmanip/zbb.adoc
[7] https://github.com/T-head-Semi/thead-extension-spec/blob/master/xtheadba/addsl.adoc

Интринсик (от англ. intrinsic) — это функция, при вызове которой компилятор генерирует конкретные аппаратные инструкции.

2.2. OCaml

OCaml — функциональный статически типизированный язык программирования из семейства ML, предлагающий мощную модульную систему, расширяющую стандартную ML, и многофункциональную объектную систему на основе классов.

Основные определения:

byte-code — это форма набора команд, предназначенная для эффективного выполнения программным интерпретатором

ocamlrun — интерпретатор байт-кода OCaml

ocamlc — компилятор байт-кода OCaml, включает в себя:

- ocamlc.byte — компилятор байт-кода, скомпилированный в байт-код
- ocamlc.opt — компилятор байт-кода, скомпилированный нативно

ocamlopt — компилятор нативного кода OCaml, включает в себя:

- ocamlopt.byte — компилятор нативного кода, скомпилированный в байт-код
- ocamlopt.opt — компилятор нативного кода, скомпилированный нативно

C- - — язык, используемый для промежуточного представления кода, называемого cmm[1]

2.2.1. Генерация переносимого байт-кода

Компилятор ocamlc — компилирует файлы в байт-код, после чего ocamlrun - исполняет его. На первом этапе генерации кода вся информация о статическом типе преобразуется в более простую промежуточную лямбда-форму. Именно после создания лямбда-формы имеется два

пути дальнейшей компиляции кода: генерация байт-кода или нативного кода. Большим преимуществом использования байт-кода является простота, переносимость и скорость компиляции. Преобразование из лямбда-формы в байт-код является простым, и это приводит к предсказуемой (но медленной) скорости выполнения, потому что не проводятся оптимизации.

2.2.2. Компиляция быстрого нативного кода

Компилятор ocamlopt компилирует лямбда-форму в быстрые исполняемые файлы нативного кода с дополнительными этапами оптимизации. Cmm — последнее промежуточное представление перед генерацией ассемблерного кода, именно в нем (в данной работе) находится вызов определенной функции по названию и заменяется на конкретные инструкции в ассемблерном коде.

2.2.3. Представление целых чисел в OCaml

Так как в данном проекте будут использоваться битовые расширения, то нужно представлять, как хранятся целые числа в OCaml.

Целое число x хранится как (x << 1) + 1 [2]. То есть число 5 будет хранится как 1011, где последний бит указывает на то, что это именно int, а не указатель. У данного подхода есть свои примущества:

- нет необходимости разыменовывать указатель при получении int;
- при создании целых чисел выделение памяти не требуется;
- меньше работы для сборщика мусора;
- меньшая фрагментация памяти;

Но такое представление данных небесплатное:

- $x + y$ переводится в инструкции процессора $x + y - 1$
- $x * y \to (x >> 1) \cdot (y - 1) + 1$

- $x/y \rightarrow (((x >> 1)/(y >> 1)) << 1) + 1$
- $xlsly \rightarrow ((x - 1) << (y >> 1)) + 1$

3. Существующие решения

В данном разделе будет рассмотрено поддержка интринсиков в проекте flambda2[8]. В данном проекте вызов внешний функции с определнным названием будет заменяться на конкретные инструкции. Объяление внешней функции из OCaml выглядит следующим образом:

Листинг 1: Объявление внешней функции

```
external popcnt: int -> int = "c_popcnt"
```

Как происходит компилирования кода с вызовом внешней функции: пусть функция находится в файле lib.c, а ее вызов в a.ml. Данные файлы будут скомпилированы в объектные файлы, но в объектном файле OCaml'a вместо адреса функции c_popcnt будут стоять нули, потому что она внешняя. Далее, после компановки в объектном файле появится конкретный адрес.

3.1. ARM интринсик «sqrt»

Замена вызова функции на конкретную инструкцию происходит после анализа промежуточного представления cmm.

Листинг 2: Анализ cmm

```
| Cextcall { func = "sqrt" } ->
(Ispecific Isqrtf, args)
```

Что означает, что вызов внешней функции с названием "sqrt" заменяется на значение алгебраического типа для специфичных операций ARM.

Листинг 3: генерация ассемблерного кода в зависимости от операции

```
| Lop(Ispecific Isqrtf as op) ->
      let instr = (match op with
                   | Ispecific Isqrtf -> "fsqrt"
                   | _                -> assert false) in
```

[8] https://github.com/ocaml-flambda/flambda-backend

`{emit_string instr} {emit_reg i.res.(0)}, {emit_reg i.arg.(0)}`

Где последняя строчка является конкретной ассемблерной инструкцией.

4. Реализация

В данном разделе будут рассмотрены два реализованных интринсиков с названиями «addsl» и «popcount»

4.1. Интринсик addsl

При реализации данного интринсика используется расширение «thead»

Условимся, что функция с названием «addsl» представляет из себя следующее:

addsl x y z = x + y * 2 ** z

Листинг 4: как транслируется в ассемблерный код функция let **f x y z =** **x** + $shift\_left$ **y z**

```
srai    a3, a2, 1
addi    a4, a1, -1
sll     a5, a4, a3
add     a0, a0, a5
```

То есть если написать функцию addsl «нативно», то она будет представлять из себя 4 инструкции. Если вызывать функцию из C, то в ассемблерном коде мы увидим call, что может сказаться на производительности.

Листинг 5: Просмотр промежуточного представления cmm

```
| (Cextcall ("addsl",_,_,_), [arg1;arg2;Cconst_int (n,_)])
 when n > 2 && n < 8 ->
     begin match (operation_supported "thead") with
           | true -> ((Ispecific (addsl (n/2))),
  ([arg1;arg2]))
  | false -> super#select_operation op args dbg
        end
```

Листинг 6: Замена вызова на конкретные инструкции

```
| Lop(Ispecific (addsl n)) ->
  `   addi {emit_reg i.arg.(1)} ,{emit_reg i.arg.(1)}, (-1)\n`;
```

```
  `    th.addsl {emit_reg i.res.(0)}, {emit_reg i.arg.(0)}, {emit_reg i.
```

4.2. Интринсик popcnt

Листинг 7: Просмотр промежуточного представления

```
  | (Cextcall ("popcount",_,_,_), [arg1]) ->
  begin match (operation_supported "popcount"), !thead_support with
| true , false ->
((Ispecific (Ipopcounti false)),
     ([arg1]))
| true , true ->  ((Ispecific (Ipopcounti true)),
       ([arg1]))
| false , _ -> super#select_operation op args dbg
  end
```

Листинг 8: Замена на конкретные инструкции

```
  | Lop(Ispecific Ipopcounti t) ->
      begin match t with
      | true ->
        ` cpop {emit_reg i.res.(0)}, {emit_reg i.arg.(0)}\n`;
        ` li {emit_reg reg_tmp}, (-1)\n`;
        ` th.addsl {emit_reg i.res.(0)},
        {emit_reg reg_tmp}, {emit_reg i.res.(0)}, 1\n`;
      | false ->
        ` cpop {emit_reg i.res.(0)}, {emit_reg i.arg.(0)}\n`;
        ` srli {emit_reg i.res.(0)}, {emit_reg i.res.(0)}, 1\n`;
        ` addi {emit_reg i.res.(0)} , {emit_reg i.res.(0)}, (-1)\n`;
      end
```

5. Эксперимент

5.1. Компактность кода

Проверка правильности интринсика фиксируется в ассемблерном коде. Пусть в файле a.ml вызывается внешняя функция shiftadd, которая принимает 3 целых числа.

Компиляция без использования флага -thead:

ocamlopt -I stdlib -S -c a.ml

С использованием флага thead:

ocamlopt -I stdlib -thead -S -c a.ml

Листинг 9: with thead

```
li      a0, 5
li      a1, 7
addi a0 ,a0, −1
th.addsl a0, a1, a0, 1
```

Листинг 10: without thead

```
li      a2, 3
li      a1, 5
li      a0, 7
la      t2, myfunc
call    caml_c_call@plt
```

Код стал более компактным.

5.2. Сравнение нативного умножения со сдвигом и интринсиком

Для замеров использовался миникомпьютер RISC-V Sipeed LicheePi 4A Risc-V TH1520 Для измерения времени работы кода использовался Google Benchmark

Таблица 1: Результаты замеров

Benchmark	Time	CPU	Iterations
addsl	1.63 ns	1.63 ns	429,733,454
caml_addsl	1.36 ns	1.36 ns	616,902,070

Как видно, подход с вызовом внешней функции во времени не выигрывает.

5.3. Два подхода теггирования

Таблица 2: Результаты замеров

Benchmark	Time	CPU	Iterations
with_th	1.09 ns	1.08 ns	644,887,876
without_th	1.09 ns	1.08 ns	644,810,674

Как видно, два подхода работают за одинаковое время

Заключение

На данный момент достигнуты следующие результаты:

1. Выбраны и реализованы инстринсики из расширений zbb thead «addsl» и «popcount».
2. Проведено сравнение ассемблерного кода и производительность инструкций.

Планы на будущее:

1. Замена умножения на константу на 1 или 2 th.addsl.
2. Использовать векторные расширения.

Код проекта доступен в GitHub-репозитории: https://github.com/Azatic/ocaml

Список литературы

[1] YARON MINSKY ANIL MADHAVAPEDDY. Real World OCaml. — URL: https://dev.realworldocaml.org/compiler-backend.html.

[2] Бранков Владимир. What is gained and lost with 63-bit integers? — URL: https://blog.janestreet.com/what-is-gained-and-lost-with-63-bit-integers/.

[3] В. А. Фролов В. А. Галактионов В. В. Санжаров. Исследование технологии RISC-V. — URL: https://www.ispras.ru/proceedings/docs/2020/32/2/isp_32_2020_2_81.pdf.