Санкт-Петербургский государственный университет

Кафедра системного программирования

Интринсики RISC-V для компилятора OCaml

Азат Райнурович Габдрахманов, группа 21.Б07-мм

Промежуточный отчет по годовой учебной практике

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

Санкт-Петербург
2024

Введение

- Набирает популярность архитектура RISC-V
- Базовая спецификация “RV32I” содержит лишь 39 инструкций, где нет даже умножения, но есть различные расширения, которые могут быть полезны для компилятора OCaml
- Реализация интринсиков может преследовать две цели:
  - Сокращение размера ассемблерного кода
  - Ускорение работы функций, используя более удачные инструкции

Существующие решения

- Компания Jane Street, занимающаяся в том числе высокочастоточным трейдингом ценных бумаг, модернизирует компилятор OCaml в целом, а в частности реализовывает интринсики для архитектур AMD64 и ARM[1]
- Базовая версия компилятора OCaml[2] (5.1.0) поддерживает RISC-V, но без интринсиков

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

[2] `https://github.com/ocaml/ocaml`

Постановка задачи

Целью работы является внедрение в компилятор OCaml интринсиков специфичных под RISC-V

Задачи:

- Выбрать и реализовать простые интринсики
- Сравнить размер ассемблерного кода и производительность инструкций

Введение в теггированные инты

Целое число x хранится как x*2+1, поэтому арифметические операции в OCaml более сложны:

- x + y переводится в инструкции процессора x + y - 1
- x * y $\rightarrow$ (x » 1) * (y - 1) + 1
- x / y $\rightarrow$ (((x » 1) / (y » 1)) « 1) + 1
- x lsl y $\rightarrow$ ((x - 1) « (y » 1)) + 1

Таким образом, int в OCaml это 2^{63}

Интринсик 1 “addsl”

Потенциальное сокращение ассемблерного кода

external ocaml_addsl : *int* → *int* → *int* → *int* = «*c_addsl*»
addsl = x + y « n для n<4:

- addi y y -1
- th.addsl res x y (n/2)

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

4>2, когда n<4, в остальных случаях thead не используется

Сравнение интринсика «addsl» с нативным вызовом

Для замеров использовался миникомпьютер RISC-V Sipeed LicheePi 4A Risc-V TH1520

Таблица: 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

Интринсик 2 “popcount” + tag int

Вызов внешей функции popcount x заменяется на:

- cpop a0, s2
- li t1, (-1) (использование временного регистра)
- th.addsl a0, t1, a0, 1

Или, если нет расширения thead:

- cpop a0, s2
- srli a0, a0, 1
- addi a0, a0, (-1)

пока непонятно, какой подход к теггированию имеет больше смысла

Сравнения теггирования в «popcount»

Таблица: 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

Результаты

На данный момент достигнуты следующие результаты:

- Выбраны и реализованы инструкции из расширений zbb thead «addsl» и «popcount»
- Проведено сравнение ассемблерного кода и производительность инструкций

future work:

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

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