Санкт-Петербургский государственный университет

Белошапкин Михаил Юрьевич

Ускорение эмулятора RISC-V, порожденного спецификацией SAIL

Отчёт по учебной практике
в форме «Решение»

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

Санкт-Петербург
2024

Оглавление

1. Введение

В современном мире существует множество различных процессорных архитектур, каждая из которых имеет свою область применения. При этом программа, собранная под одну систему команд, не будет работать на компьютере с другой архитектурой. Для решения этой проблемы существуют так называемые эмуляторы процессоров.

Эмулятор процессора – это программное или программно-аппаратное средство, которое позволяет имитировать работу настоящего физического микропроцессора. На данный момент существует множество таких инструментов, одним из них является Sail. Sail позволяет при помощи специального одноименного языка создавать описание ISA семантики инструкций, и по данному описанию генерировать соответствующие эмуляторы. Таким образом, любой инженер может при помощи этого инструмента описать свою произвольную систему команд и получить готовый симулятор.

На данный момент на Sail реализовано несколько распространенных архитектур, в том числе RISC-V. RISC-V является открытым стандартом, который разработан на базе концепции RISC, и в последние годы он набирает все большую популярность. Причной тому является его открытость и гибкость, позволяющие применять данные процессоры в различных областях, от встроенных систем до высокопроизводительных вычислений.

Sail является перспективным инструментом, поэтому увеличение его быстродействия является актуальной задачей. В рамках данной работы рассматриваются различные методы ускорения эмулятора RISC-V, порожденного спецификацией Sail.

2. Постановка задачи

Целью работы является оптимизация эмулятора архитектуры RISC-V, порожденного спецификацией Sail. Для достижения данной цели были сформулированы следующие промежуточные задачи.

- Сделать обзор предметной области.
- Применить один или несколько методов оптимизации.
- Провести тестирование производительности и проанализировать полученные результаты.

3. Обзор

3.1. SAIL

Sail – это инструмент, который позволяет программно эмулировать различные процессорные архитектуры. Основой данного решения является компилятор, который принимает на вход текст на специальном формальном языке Sail, а на выходе генерирует программу на C или OCaml. Sail представляет собой функциональный язык, по синтаксису схожий с Rust. С помощью него пользователь описывает семантику микрокоманд процессора, а полученная в результате трансляции программа и является эмулятором, который в качестве аргумента принимает ELF-файл, собранный под описанную архитектуру. На данный момент на Sail реализованы некоторые наиболее распространенные архитектуры, в том числе RISC-V, MIPS, X86. Также Sail предоставляет пользователям дополнительную функциональность, например: автоматическая генерация тестов, верификация описанной модели.

Сгенерированный эмулятор запускает исполняемые файлы так, как если бы они работали на чистом железе без промежуточного слоя в виде операционной системы, при этом запускаемые бинарные файлы не могут использовать никакие библиотеки, что накладывает некоторые ограничения.

3.2. Используемые инструменты

Стоит кратко описать набор инструментов, с помощью которых производилось исследование.

Perf – утилита для анализа производительности программ для ОС Linux. Данный инструмент позволяет производить статистическое профилирование: замерять производительность как отдельной программы, так и ядра системы. В ходе исследования данный инструмент использовался для анализа стека вызовов.

GDB – консольный отладчик, работающий на Unix-подобных операционных системах. С помощью данного инструмента можно исполнять

программу пошагово, следить за состоянием переменных и т. д. Данная программа использовалась для изучения внутреннего устройства сгенерированного эмулятора.

time – стандартная Linux-утилита для замера времени работы программ.

4. Подходы к оптимизации

Существуют различные подходы к ускорению и оптимизации программ, однако в случае с эмулятором RISC-V наиболее подходящими являются низкоуровневые и алгоритмические. В рамках первого семестра было принято решение рассмотреть оптимизацию с использованием так называемых интринзик.

4.1. Интринзики

Интринзики – это специальные встроенные функции, поддерживаемые на уровне компилятора. При трансляции исходного текста в месте вызова интринзик компилятор порождает особый код, характерный именно для конкретной функции, иными словами, будет сгенерирована последовательность инструкций, специфичных для определенной архитектуры. Данные встроенные функции используются тогда, когда необходимо ускорить вычисления. Таким образом, интринзики ухудшают переносимость программы, но в то же время повышают ее быстродействие. Компилтор GCC, с помощью которого собирается эмулятор, также поддерживает встроенные функции, полный список которых представлен в документации[1].

Суть оптимизации с применением интринзик заключается в том, что исполнение инструкции будет происходить не напрямую, а с помощью встроенной в компилятор функции. Например, в системе команд эмулируемого процессора есть инструкция, которая по своей семантике полностью совпадает с какой-либо инструкцией физического процессора, на котором запускается программа-симулятор. Тогда логичным решением было бы использовать для вычисления уже существующую команду, а не считать ее программно.

Для проверки эффективности данного метода было принято решение рассмотреть некоторую RISC-V инструкцию, вычисление которой можно было бы заменить одной интринзикой. При этом логика испол-

[1]https://gcc.gnu.org/onlinedocs/gcc/x86-Built- in-Functions.html

нения выбранного оператора должна быть достаточно сложна, чтобы применение встроенной функции могло заметно уменьшить алгоритмическую трудоемкость его вычисления.

4.1.1. Внедрение

В качестве примера такой инструкции был выбран CPOP [3] – данный оператор считает количество единиц в машинном слове. При помощи отладчика исходном тексте эмулятора было найдено место, которое отвечает за его вычисление. На листинге 1 представлен цикл, в котором и происходит вычисление:

Листинг 1: Вычисление CPOP

```
for_start_8849: ;
  {
    if ((zuz32491 > zgsz39259)) goto for_end_8850;
    bool zgaz37141;
    {
      fbits zgaz37140;
      zgaz37140 = (UINT64_C(1) & (zuz32489 >> zuz32491));
      zgaz37141 = eq_bit(zgaz37140, UINT64_C(1));
    }
    if (zgaz37141) {
      {
        sail_int zgsz316231;
        CREATE(sail_int)(&zgsz316231);
        CONVERT_OF(sail_int, mach_int)(&zgsz316231, INT64_C(1));
        add_int(&zuz32490, zuz32490, zgsz316231);
        KILL(sail_int)(&zgsz316231);
      }
      zgsz39261 = UNIT;
    } else {  zgsz39261 = UNIT;  }
    zuz32491 = (zuz32491 + zgsz39260);
    goto for_start_8849;
  }
for_end_8850: ;
```

Здесь в качестве счетчика цикла выступает переменная zuz32491, а количество единиц – результат вычисления CPOP, – хранится в zuz32490. Более того, в случае увеличения счетчика единиц внутри функций CREATE и KILL будут вызваны malloc и free, при том вызва-

ны они будут не более 32 и 64 раз для 32- и 64-битных эмуляторов соответственно. В качестве оптимизации предлагается заменить данный цикл на одну встроенную функцию, которая выполняет ту же задачу.

Компилятор GCC имеет встроенную функцию __builtin_popcount, которая принимает целое число и возвращает количество единиц в двоичном представлении переданного числа. Представленный на листинге цикл и был заменен на эту интринзику.

4.1.2. Тестирование

Для оценки эффективности примененной оптимизации необходимо произвести замеры времени на некотором наборе тестов. Для этого можно подобрать такие программы на языке C, при компиляции которых будет сгенерировано достаточно большое количество необходимых инструкций. Но можно пойти другим путем: бинарные файлы с нужным набором инструкций можно генерировать автоматически с помощью Python-скрипта. Так, были получены несколько исполняемых файлов, которые содержат от 10000 до 50000 инструкций CPOP.

На тестовых бинарных файлах были запущены два варианта эмулятора: с оптимизацией и без нее. Время исполнения эмулятора измерялось при помощи утилиты /usr/bin/time. Замеры показали, что внедренная оптимизация практически никак не повлияла на время исполнения, конкретные значения представлены в таблице 1 (представлены усредненные значения за 10 замеров). При прогоне тестов при помощи утилиты perf было выяснено, что на функцию, вычисляющую инструкции, тратится около одного процента процессорного времени. При этом статистика показывает, что наибольшая часть времени работы программы приходится на системные вызовы malloc и free. Следует заметить, что внедрение встроенных функций уменьшает количество этих системных вызовов, однако сложно оценить заранее, насколько реально эта оптимизация повлияет на быстродействие эмулятора.

	С оптимизацией	Без оптимизации
10000	9.29	9.51
20000	14.86	15.02
30000	24.72	24.69
40000	30.74	30.80
50000	38.14	38.92

Таблица 1: Результаты замеров

5. Заключение

Были получены следующие результаты

- Сделан обзор предметной области.
- Внедрена оптимизация с использованием встроенных функий GCC.
- Проведены замеры производительности.

Список литературы

[1] The Sail instruction-set semantics specification language : Rep. / Technical report published by Cambridge University ; executor: Gray Kathryn E, Sewell Peter, Pulte Christopher et al. : 2017

[2] The RISC-V instruction set manual / Waterman Andrew, Lee Yun- sup, Patterson David, Asanovic Krste, level Isa Volume I User, Wa- terman Andrew, Lee Yunsup, and Patterson David // Volume I: User- Level ISA', version.— 2014.—Vol. 2

[3] RISC-V Bitmanip Extension / Claire Wolf // Version 0.94-draft

[4] GCC documentation / https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html (Дата: 10.02.2024)

[5] De Melo Arnaldo Carvalho. The new Linux perf tools // Slides from Linux Kongress.— 2010.—Vol. 18.—P. 1–42.