Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 21.Б15-мм

Инструмент для автоматической генерации тестов на подсистему памяти RISC-V CPU

Печенев Данила Евгеньевич

Отчёт по производственной практике
в форме «Производственное задание»

Научный руководитель:
руководитель Лаборатории YADRO СПбГУ, Я. А. Кириленко

Консультант:
ведущий инженер-программист, ООО «ЯДРО ЦЕНТР ИССЛЕДОВАНИЙ И РАЗРАБОТКИ», Д. В. Донцов

Санкт-Петербург
2023

Оглавление

Введение

В настоящее время разработчиками со всего мира активно развивается расширяемая, открытая и свободная процессорная архитектура RISC-V. В России лидирующей компанией-разработчиком микропроцессорных технологий и инструментов на базе архитектуры RISC-V является компания Syntacore, сооснователь консорциума RISC-V.

В процессе разработки нового аппаратного обеспечения в Syntacore возникает необходимость в создании и использовании приложений для измерения производительности различных компонентов процессора. Такие приложения в рамках этой работы мы будем называть *тестами* или *бенчмарками*. Они в разной степени нагружают определенные составляющие процессора, например, подсистему памяти (кэши разного уровня и DRAM), MMU и TLB, предсказатель переходов, префетчер. Результаты запуска таких тестов позволяют оценить и сравнить с теоретически ожидаемыми показателями фактическую производительность различных компонентов CPU, а также скорость их взаимодействия.

Используемые на данный момент наборы тестов, написанных на C, не обладают достаточной гибкостью изменения для разных экспериментов. Во многом это связано с компиляторными оптимизациями: даже при самых слабых опциях оптимизации компилятор способен существенно влиять на последовательность инструкций, которую нам бы хотелось получить. Кроме того, результаты некоторых тестов не всегда точны из-за возможности запусков только под Linux: эффекты работы операционной системы (многозадачность, прерывания, демоны), промахи в TLB снижают точность замеров.

В этой связи поступил запрос на создание инструмента, который позволял бы генерировать бенчмарки, запускаемые под baremetal (режим работы без операционной системы), и основу которых составляли бы неизменяемые ассемблерные вставки. Такой подход позволит решить описанные выше проблемы. После обсуждения с коллегами было принято решение начать разработку инструмента с реализации генерации тестов на подсистему памяти, чему и посвящена эта работа.

1. Постановка задачи

Целью работы является разработка инструмента, способного автоматически генерировать тесты на подсистему памяти RISC-V CPU.

Допущения. Тесты генерируются под режим baremetal — это позволяет избежать эффектов работы операционной системы, а также контролировать, использовать MMU либо нет. Сгенерированные тесты при запуске сами считают и выводят свой результат.

На инструмент накладываются следующие *требования*, принципиально отличающие его от других существующих решений.

- Шаблон доступа к памяти (последовательность операций load и store в определенном порядке) является конфигурируемым параметром.
- Предусмотрена возможность указать, будут ли инструкции зависимыми или нет. Под этим понимается:
  - Зависимые инструкции — все пары подряд идущих инструкций являются зависимыми, то есть для выполнения следующей инструкции необходим результат выполнения предыдущей.
  - Незасивимые инструкции — все пары подряд идущих инструкций являются независимыми, то есть могут выполняться параллельно при наличии у процессора такой возможности.
- Есть возможность генерирования как тестов, взаимодействующих с памятью напрямую, так и тестов, использующих MMU (на данном этапе достаточно будет транслировать виртуальные адреса в физические как 1:1).
- Возможно генерировать тесты как с выровненным, так и с невыровненным доступом к памяти.

Для достижения цели были поставлены следующие задачи.

- Определить и обосновать формат конфигурации теста, включая доступные параметры и ограничения на них.
- Реализовать генерацию тестов на подсистему памяти RISC-V CPU.
- Проверить средствами моделирования сгенерированные бенчмарки для некоторых популярных сценариев, а именно тесты на измерение:
  - времени доступа к L1 кэшу;
  - времени доступа к L2 кэшу (при условии промаха в L1 кэш);
  - времени доступа к DRAM (при условии промаха в кэш и правильного предсказания адресов запросов префетчером);
  - пропускной способности L1 кэша;
  - времени доступа к L1 кэшу при промахе в TLB;
  - времени доступа к L2 кэшу при промахе в TLB (при условии промаха в L1 кэш).

2. Глоссарий

В этом разделе мы поясняем и приводим определения некоторых узкоспециализированных ключевых терминов, которые будут встречаться далее.

Нас будут интересовать тесты для L1, L2 кэшей и DRAM.

DRAM (dynamic random access memory) — энергозависимая память с произвольным доступом, используемая в компьютере в качестве оперативной памяти.

В тестах на подсистему памяти принято использовать два вида инструкций: загрузки данных из памяти и сохранения данных в память. Для этих терминов мы будем использовать их английские названия ввиду их крайней общеупотребимости и для более четкой передачи смысла.

load-инструкция — инструкция, выгружающая данные из памяти (из кэша какого-либо уровня либо DRAM) в регистр.

store-инструкция — инструкция, сохраняющая данные в память (в кэш либо DRAM).

При оценке производительности подсистемы памяти используются различные характеристики. Две основные из них — это время доступа и пропускная способность.

Время доступа (англ. latency) — это задержка в тактовых циклов, которую создает load-инструкция в цепочке зависимых инструкций.

Пропускная способность (англ. bandwidth, throughput) — это максимальное количество независимых инструкций, взаимодействующих с памятью (load- и store- инструкции в любом соотношении), которые могут быть исполнены за один тактовый цикл. В тестах чаще всего измеряют обратную величину. То есть если результат теста получился 0.33, это значит, что пропускная способность какого-то уровня памяти равна трём.

Префетчер (англ. hardware prefetcher) — элемент CPU, который пытается предсказать будущие запросы данных и начать их выполнение заранее.

3. Обзор

Исследованием производительности микропроцессорных ядер занимается Департамент решений для программно-аппаратного дизайна компании YADRO. В этом разделе мы обсудим бенчмарки для подсистемы памяти, которые используются в нем на данный момент. Кроме того, детальнее обсудим их устройство и ограничения, затронем методы измерения основных характеристик подсистемы памяти, их преимущества и недостатки.

3.1. Тесты на измерение времени доступа

Время доступа является самой важной характеристикой памяти среди всех других. Для измерения времени доступа используются бенчмарки из набора lmbench [1, 3]. Кроме них создаются и используются под разные задачи многочисленные самописные тесты, написанные на C. Измерение времени доступа как правило реализуется с помощью связного списка, каждый элемент которого указывает на адрес следующего элемента. Осуществляя проход по такому списку, мы получаем цепочку зависимых load-инструкций. То есть в C-коде мы будем последовательно выполнять операцию

p = (uint64_t*)*p

Для x86 такой код транслируется в

movq (%rdx), %rdx
movq (%rdx), %rdx
movq (%rdx), %rdx
movq (%rdx), %rdx
...

А для RISC-V в последовательность

ld a4,0(a4)
ld a4,0(a4)

```
ld a4,0(a4)
ld a4,0(a4)
...
```

Список замыкается в кольцо, а самый последний полученный элемент каким-то образом используется (например, выводится на консоль или возвращается в качестве кода возврата), чтобы не дать компилятору «подумать», что результаты прохода по памяти нигде не используются, и эти инструкции можно не генерировать.

На базе этой структуры данных можно реализовать последовательное обращение к ячейкам памяти, расположенным как угодно относительно друг друга.

Популярным способом измерения времени доступа является линейный алгоритм. В нем элементы связного списка расположены линейно, то есть на одинаковом расстоянии друг от друга. Это означает, что мы осуществляем линейный проход по памяти с определенным *шагом (англ. stride)*. Сколько именно операций обращения к памяти мы осуществляем определяется еще одним параметром теста — *количеством прокачиваемой памяти*. То есть мы идем по памяти до тех пор, пока весь наш путь не превышает это значение. Один проход по памяти называется *итерацией (англ. iteration, repetition)*. Дополнительно в качестве параметров теста задаются также количество итераций эксперимента и количество *итераций прогрева (англ. warmup-iteration)*. Последние нужны для прогрева кэша и других аппаратных компонент, например, предсказателя переходов. Ведь если нам хочется измерить время доступа в L1 кэш, и мы начнем делать замеры с первой же итерации, то данные с момента начала измерения будут лежать в DRAM, что не позволит получить корректный результат. Таким образом, итерации прогрева не учитываются при измерении.

Недостатком линейного доступа в память является то, что этот шаблон легко распознается префетчерами всех уровней, включая TLB префетчер, что иногда приводит к занижению значения времени доступа, а именно в области DRAM. Поэтому в lmbench и других тестах реализован алгоритм случайного доступа. В нем адреса элементов спис-

ка случайно распределены по выделенному блоку памяти. Такая схема позволяет наиболее эффективно противостоять префетчерам. В соответствующих тестах используется (условно) случайный алгоритм распределения элементов списка по адресам памяти так, чтобы все множество адресов образовало линейную последовательность с заданным шагом.

Резюмируя, преимуществом используемых на данный момент тестов на измерение времени доступа несомненно является их кроссплатформенность. Достаточно скомпилировать тест, написанный на C, под нужную платформу. Кроме того, в них можно конфигурировать ключевые параметры, такие как шаг, объем прокачиваемой памяти, количество итераций прогрева и замеряемых итераций.

Говоря о недостатках, стоит отметить, что в этих тестах отсутствует возможность регулировать, будут ли обращения к памяти выровненными или нет. Кроме того, lmbench тесты могут быть запущены только под Linux, а значит их исполнение без использования MMU невозможно, что снижает точность таких тестов на больших объемах прокачиваемой памяти из-за промахов в TLB.

3.2. Тесты на измерение пропускной способности

Пропускная способность — это вторая по важности характеристика памяти. В отличие от времени доступа, максимальное значение пропускной способности для разных архитектур показывают разные шаблоны доступа к памяти (однако напомним, что соседние инструкции в тестах на пропускную способность всегда являются независимыми). Иными словами, один и тот же тест может не раскрыть максимальное значение пропускной способности на разных архитектурах.

Так, для архитектуры ARM пропускную способность измеряют с помощью шаблона 1 load + 1 store, а для x86-64 — 2 load + 1 store.

Для каждого шаблона приходится придумывать свой собственный алгоритм, который будет реализован на C. Каждый раз приходится изобретать методы того, как избегать компиляторных оптимизаций с

минимальным (в идеале — нулевым) влиянием на желаемую последовательность инструкций. А именно, необходимо сделать так, чтобы компилятор не смог переставить инструкции местами или выкинуть часть инструкций, поняв, что их результаты не используются далее.

Например, для шаблона 1 load можно применить идею из замеров на время доступа. А для того, чтобы инструкции стали независимыми, мы будем идти не по одному связному списку, а по нескольким сразу (например, по десяти, чего точно будет достаточно). Как и в случае с тестами на время доступа, в конце необходимо будет что-то сделать с полученными элементами, чтобы компилятор не выбросил инструкции. Например, можно сложить значения в этих элементах и вывести на консоль либо вернуть результат в качестве кода возврата.

На данный момент для измерения пропускной способности используется бенчмарк STREAM [2] и самописные тесты. Их преимуществом, как и в случае с тестами на время доступа, являются возможность конфигурирования ключевых параметров и кроссплатформенность. Однако отсутствие возможности менять шаблон доступа к памяти произвольным образом в некоторых случаях сводит преимущество кроссплатформенности на нет: тот же тест не сможет показать максимальный уровень пропускной способности на другой архитектуре. Кроме того, бенчмарк STREAM, позволяющий использовать хотя бы несколько разных шаблонов доступа к памяти, не может быть запущен в режиме baremetal, из-за чего результаты экспериментов на больших объемах прокачиваемой памяти будут неточными ввиду промахов в TLB.

3.3. Остальные тесты

Все остальные тесты являются самописными. Такие тесты непросто писать, поскольку, как уже было сказано ранее, каждый раз приходится придумывать методы обхода оптимизаций компилятора. Кроме этого, приходится придумывать, какие алгоритмы на языке C будут транслироваться в нужную последовательность load/store инструкций (которая порой может быть не столько тривиальной) — и это тоже не

всегда просто. При этом редко когда важно, чтобы тест сходу можно было запускать на разных архитектурах. Характеристики подсистемы памяти, измеряемые тестами для конкретной архитектуры, интересны сами по себе в абсолютном выражении.

Все выше сказанное позволяет предположить, что более эффективным способом создания тестов станет их генерация под режим baremetal при помощи *неизменяемых компилятором ассемблерных вставок (asm volatile-вставок)*. Такой подход помимо базовых параметров, обычно конфигурируемых в разных бенчмарках, позволит варьировать способ обращения к памяти (напрямую или через MMU), менять load-store шаблон доступа к памяти практически без ограничений (теперь не нужно будет «запутывать» компилятор), определять зависимость или независимость инструкций, контролировать выравнивание доступа. А недостаток зависимости тестов от платформы не является столь уж существенным, поскольку, как было отмечено, результаты тестов на разрабатываемых компанией Syntacore процессорных ядрах интересны сами по себе. Тем не менее, при разработке инструмента мы будем стараться абстрагировать алгоритмы генерации ассемблерного кода так, чтобы для поддержки новой архитектуры CPU достаточно было конкретизировать инструкции для этой архитектуры.

Список литературы

[1] GitHub репозиторий lmbench. — URL: https://github.com/foss-for-synopsys-dwc-arc-processors/lmbench/tree/master (дата обращения: 20.12.2023).

[2] Бенчмарк STREAM. — URL: https://www.cs.virginia.edu/stream/ (дата обращения: 20.12.2023).

[3] Документация на lmbench тест на время доступа. — URL: https://lmbench.sourceforge.net/man/lat_mem_rd.8.html (дата обращения: 20.12.2023).