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 «КНС ГРУПП», Д. В. Донцов

Санкт-Петербург
2024

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. Отчасти 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 операционной системой: эффекты работы ОС (многозадачность, прерывания, демоны), а также 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.*

- 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.

  В следующем разделе станет ясно, почему в тестах важно контролировать зависимость/независимость инструкций, как этот параметр влияет на то, что измеряет бенчмарк.

- 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 с поддержкой всех необходимых параметров конфигурации.
- Проверить средствами моделирования сгенерированные бенчмарки для некоторых популярных сценариев, а именно тесты на измерение:
  - времени доступа к L1 кэшу;
  - времени доступа к L2 кэшу;
  - времени доступа к L3 кэшу;
  - времени доступа к DRAM (при условии правильного предсказания адреса запроса префетчером);
  - времени доступа к DRAM (при условии неверного предсказания адреса запроса префетчером);
  - пропускной способности L1 кэша;
  - времени поиска страницы при промахе в TLB.

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, L3 кэшей и DRAM.

Кэш-попадание (англ. cache-hit) — ситуация, когда результат обрабатываемого запроса к памяти находится в кэше определенного уровня и его можно вернуть без обращения к памяти с более медленным доступом. Например, L2-hit означает промах в L1 кэш, но попадание в L2 кэш.

Кэш-промах (англ. cache-miss) — ситуация, когда результат обрабатываемого запроса к памяти отсутствует в кэше определенного уровня и для его получения необходимо обращаться к памяти с более медленным доступом. Например, L3-miss означает промах в L3 кэш и, как следствие, промахи в L1 и L2.

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 циклах, которую создает load-инструкция в цепочке *зависимых* инструкций. Зависимость инструкций требуется для избежания префетчинга данных.

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

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 такта на операцию, это значит, что пропускная способность равна трём.

MMU (memory management unit) — компонент процессора, отвечающий за управление доступом к памяти, запрашиваемой процессором. Его ключевыми функциями являются трансляция адресов виртуальной памяти в адреса физической памяти (то есть управление виртуальной памятью), а также защита памяти.

TLB (translation lookaside buffer) — специализированный кэш процессора, используемый для ускорения трансляции адреса виртуальной памяти в адрес физической памяти путем запоминания соответствия виртуальных страниц физическим.

В случае, если в TLB не найдена физическая страница, соответствующая виртуальной, MMU осуществляет поиск страницы в таблицах страниц (процедура page walk).

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 всегда 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, что иногда приводит к занижению значения времени доступа, особенно в области DRAM. Поэтому в lmbench и других тестах реализован алгоритм случайного доступа. В нем адреса элементов списка случайно распределены по вы-

деленному блоку памяти. Такая схема позволяет наиболее эффективно противостоять префетчерам. В соответствующих тестах используется (псевдо) 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, с одной стороны, может снизить точность таких тестов на больших объемах прокачиваемой памяти из-за промахов в TLB, а с другой — делает невозможным оценку влияния промаха в TLB на время доступа.

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 могут быть 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 не так сильно важно, чтобы тест сходу можно было запускать на разных архитектурах. Ведь характеристики подсистемы памяти, измеряемые тестами для конкретной архитектуры, интересны и важны сами по себе в абсолютном выражении. Кроме того, в первую очередь нам интересно сравнивать разрабатываемое процессорное ядро с другими ядрами *той же архитектуры* — для этого подойдут те же тесты.

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.

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.

4. Реализация

В этом разделе мы не будем рассматривать то, как именно был реализован инструмент: его архитектуру, код, алгоритмы генерации корректной последовательности инструкций, удовлетворяющей всем заданным параметрам. Вызвано это тем, что на проект наложено соглашение о неразглашении.

Вместо этого мы детально рассмотрим формат конфигурации теста, а именно структуру конфигурационного файла. Кроме того, мы покажем принципиальную схему сгенерированного теста, рассмотрим, из каких частей он состоит.

4.1. Формат конфигурационного файла

Параметры конфигурации теста записываются в конфигурационный JSON-файл, соответствующий следующей структуре:

`cpu_architecture`: [`value`]
`cpu_part`: [`value`]
`mode`: [`value`]
`hardware_configuration`:
- `start_address`: [`value`]

`test_configuration`:
- `use_mmu`: [`value`]
- `warmup_iterations`: [`value`]
- `stride`: [`value`]
- `load_store_pattern`: [`value`]
- `blocks_number`: [`value`]
- `iterations`: [`value`]
- `dependent_operations`: [`value`]
- `unroll_loop`: [`value`]
- `offset`: [`value`]

Таким образом, конфигурационный файл включает в себя

- Общие параметры;

- Конфигурацию оборудования;
- Конфигурацию теста.

4.1.1. Общие параметры

- cpu_architecture — архитектура CPU. Поддерживаемые значения: risc-v (RISC-V).
- cpu_part — подсистема CPU. Поддерживаемые значения: memory_subsystem (подсистема памяти).
- mode — режим работы теста для указанной подсистемы CPU. Поддерживаемые значения: memory_pass (проход по памяти).

4.1.2. Параметры конфигурации оборудования

- start_address — адрес начала RAM (в байтах). Значение по умолчанию: 0.

4.1.3. Параметры конфигурации теста

Параметры конфигурации теста зависят от компонента CPU, для которого тест генерируется, и режима работы теста. На данный момент поддержаны только подсистема памяти и режим прохода, ниже приводятся параметры для этого случая.

- use_mmu — использовать ли MMU. Если false, то обращения к памяти производятся напрямую. Если true, то виртуальные адреса транслируются в физические 1:1. Значение по умолчанию: false.
- warmup_iterations — количество warmup-итераций. Нужны не только для прогрева кэша, но и, например, для стабилизации работы предсказателя переходов. Значение по умолчанию: 10.
- stride — шаг считывания/записи (в байтах) или код функции stride, задающей последовательность шагов. Во втором случая функция

принимает одно число — номер шага (начиная с 0) и возвращает шаг, соответствующий этому номеру. Значение по умолчанию: 16.

- load_store_pattern — шаблон доступа к памяти. Определяется как последовательность символов l и s, где l — load, s — store. Примеры: l; lls; ls; lssls. Шаблон задает **блок** load-store операций. Значение по умолчанию: l.

- blocks_number — количество выполняемых за одно прохождение по памяти **блоков** load-store операций, задаваемых шаблоном load_store_pattern. Значение по умолчанию: 64.

- iterations — количество прохождений по памяти. Значение по умолчанию: 100.

- dependent_operations — зависимы ли операции. Если true, все пары подряд идущих инструкций будут зависимы, если false — независимы. Значение по умолчанию: true.

- unroll_loop. По умолчанию инструкции одной итерации не дублируются, а тест представляет собой цикл по этим инструкциям. Если unroll_loop равен true, то инструкции для всех итераций будут вставлены в программный код единым полотном без условных переходов, т. е. по сути произойдет раскрутка цикла (loop unrolling). Значение по умолчанию: false.

- offset — сколько байт отступить от начала первой страницы для прохождения по памяти. Значение должно быть меньше размера страницы. Позволяет тестировать запросы к невыровненным данным. Значение по умолчанию: 0 (выровненный доступ).

4.2. Схема сгенерированного теста

Рассмотрим схему генерируемого C-файла сверху вниз:

- Вначале в комментарии указывается конфигурация, для которой сгенерирован тест.

- Включаются необходимые заголовочные файлы. Поскольку тест генерируется под режим baremetal, обычные заголовочные файлы не подойдут — используются специально переопределенные реализации стандартных библиотек, разрабатываемые компанией Syntacore.

- Далее инициализируются данные для теста. Для этого прямо в коде задаются страницы данных и указывается, какому адресу они соответствуют. Например:

```
__attribute__ ((section(".page0x32000"), aligned(0x1000), used))
static uint8_t page0x32000[] = {
    16, 32, 3, 0, 0, 0, 0, 0, 0, 0, 0, 16, 32, 19, 0, 0, ...
};
```

- В случае, если используется MMU, инициализируются многоуровневые таблицы страниц аналогично примеру выше.

- Далее идет код процедуры PRELIMINARY_ACTION(). В ней осуществляется инициализация некоторых регистов специальными значениями для корректного выполнения теста. Кроме того, в случае использования MMU в этой функции происходит загрузка адреса page global directory (PGD) в регистр satp — он хранит в себе этот адрес и именно с прочтения этого регистра начинается процедура page walk.

- Далее описывается процедура POST_ACTION(). В ней содержатся некоторые завершающие действия.

- Затем при помощи asm volatile вставки описывается одна итерация прохода по памяти. Например:

```
__attribute__((always_inline)) static inline void
ONE_ITERATION() {
    asm volatile(
        "ld x9, 0(x9);"
        "ld x9, 0(x9);"
```

```
        "ld x9, 0(x9);"
        "ld x9, 0(x9);"
        "ld x9, 0(x9);"
        "ld x9, 0(x9);"
        "ld x9, 0(x9);"
        "ld x9, 0(x9);"
    );
}
```

- Затем в функции HOTSPOT итерация выполняется указанное в конфигурационном файле количество раз. Перед контрольными итерациями выполняются warmup-итерации. Кроме того, происходит замер количества тактов, которые потребовались на выполнение всех контрольных итераций. Например для случая, когда unroll_loop = true:

```
__attribute__((always_inline)) static inline void
HOTSPOT(uint64_t* result) {
    /*
    WARMUP ITERATIONS
    */
    ONE_ITERATION();
    ONE_ITERATION();
    ONE_ITERATION();
    /*
    EXPERIMENT ITERATIONS
    */
    asm volatile(
        "rdcycle x5;"
    );
    ONE_ITERATION();
    ONE_ITERATION();
    ONE_ITERATION();
    ONE_ITERATION();
```

```
        ONE_ITERATION();
        asm volatile(
            "rdcycle x6;"
            "sub %0, x6, x5;"
            : "=r"(*result)
            :
        );
    }
```

В случае, когда unroll_loop = false, с помощью asm volatile вставок организуется цикл по ONE_ITERATION() необходимое количество раз.

- Наконец, в функции main происходит постановка эксперимента. На консоль выводятся количество исполненных load-store блоков, количество циклов (тактов), которое на это потребовалось, и среднее число циклов на блок инструкций.

Помимо C-файла генерируются некоторые вспомогательные скрипты, в том числе линкер-скрипт для корректного отображения страниц данных на память. Мы не будем их детально рассматривать.

5. Применение инструмента

В этом разделе мы покажем, как можно использовать инструмент для генерации бенчмарков. В качестве примера рассмотрим тесты на время доступа и пропускную способность L1 кэша, время доступа к L2 кэшу.

5.1. Время доступа к L1 кэшу

Параметры cpu_architecture, cpu_part и mode всегда имеют одни и те же значение. start_address будет зависеть от того, на чем мы запускаем тест. Для примера возьмем 0. MMU не используем (use_mmu = false), 10 warmup-итераций достаточно (warmup_iterations = 10).

Шаг (stride) надо выбрать таким, чтобы каждый запрос приходился на новую кэш-линейку L1 кэша. Если размер кэш-линии L1 кэша целевой машины равен 64 байтам, то можно положить stride = 64. Для тестов на время доступа в качестве шаблона доступа используются зависимые load-инструкции. Поэтому load_store_pattern = l,
dependent_operations = true.

Количество блоков (в данном случае просто load-операций) нужно поставить таким, чтобы не выйти за пределы L1 кэша при итерировании. Например, 128 (blocks_number = 128). Далее необходимо выбрать количество контрольных итераций. Возьмем 100 для точности (iterations = 100).

unroll_loop необходимо выставить в false для избежания отсутствия строк кода в кэше инструкций (в таком случае много тактов будет тратиться на чтение инструкций из памяти в кэш, и результат теста получится далеким от истинного). Нас интересует обычный выравненный доступ, поэтому offset = 0. По итогу получаем следующую конфигурацию:

```
cpu_architecture: risc–v
cpu_part: memory_subsystem
mode: memory_pass
```

```
hardware_configuration:
    — start_address: 0
test_configuration:
    — use_mmu: false
    — warmup_iterations: 10
    — stride: 64
    — load_store_pattern: 1
    — blocks_number: 128
    — iterations: 100
    — dependent_operations: true
    — unroll_loop: false
    — offset: 0
```

Запустив инструмент на этой конфигурации, мы получим тест для измерения времени доступа к L1 кэшу.

5.2. Пропускная способность L1 кэша

Возьмем за основу конфигурацию для теста на время доступа к L1 кэшу. Для измерения пропускной способности необходимо, чтобы инструкции были независимы (dependent_operations = false). Кроме того, необходимо будет установить в качестве load_store_pattern такую последовательность load- и store- инструкций, которая раскрывает максимальное значение пропускной способности для данной архитектуры. По итогу мы получим следующую конфигурацию:

```
cpu_architecture: risc–v
cpu_part: memory_subsystem
mode: memory_pass
hardware_configuration:
    — start_address: 0
test_configuration:
    — use_mmu: false
    — warmup_iterations: 10
    — stride: 64
```

```
    — load_store_pattern: 1
    — blocks_number: 128
    — iterations: 100
    — dependent_operations: false
    — unroll_loop: false
    — offset: 0
```

Запустив инструмент на этой конфигурации, мы получим тест для измерения пропускной способности L1 кэша.

5.3. Время доступа к L2 кэшу

Возьмем за основу конфигурацию для теста на время доступа к L1 кэшу. В ней необходимо поменять stride в случае, если величина кэш-линии L2 кэша отличается от величины кэш-линии L1 кэша. Положим, что в нашем случае она не отличается. Главное отличие будет состоять в том, что теперь необходимо выбрать количество блоков операция (т. е. в данном случае просто количество load-операций) таким, чтобы в какой-то момент пришлось вытеснять данные из L1 кэша в L2 кэш. Таким образом в случае итерирования по памяти каждый раз мы будем обращаться к вытесненным данным. Конкретное значение должно быть определено исходя из размера L1 кэша. Положим blocks_number равным 2048 и получим конфигурацию

```
cpu_architecture: risc–v
cpu_part: memory_subsystem
mode: memory_pass
hardware_configuration:
    — start_address: 0
test_configuration:
    — use_mmu: false
    — warmup_iterations: 10
    — stride: 64
    — load_store_pattern: 1
    — blocks_number: 2048
```

— `iterations`: 100

— `dependent_operations`: `true`

— `unroll_loop`: `false`

— `offset`: 0

Запустив инструмент на этой конфигурации, мы получим тест для измерения времени доступа к L2 кэшу.

6. Проверка корректности

Для проверки корректности сгенерированных тестов сначала они были запущены на функциональном симуляторе. Результаты сверялись с эталонными, взятыми с запусков других бенчмарков. В случае неуспеха находились и исправлялись ошибки в процессе генерации.

В случае успеха запуска на симуляторе, т. е. получении ожидаемого результата, тесты также запускались на FPGA для финальной оценки их работоспособности и корректности.

Все тесты, обозначенные в первом разделе, прошли проверу на симуляторе и FPGA.

Заключение

В рамках работы над производственной практикой были сделаны следующие задачи.

- Определен и обоснован формат конфигурации теста, включая доступные параметры и ограничения на них.
- Реализована генерацию тестов на подсистему памяти RISC-V CPU с поддержкой всех необходимых параметров конфигурации.
- Проверены средствами моделирования сгенерированные бенчмарки для некоторых популярных сценариев, а именно тесты на измерение:
  - времени доступа к L1 кэшу;
  - времени доступа к L2 кэшу;
  - времени доступа к L3 кэшу;
  - времени доступа к DRAM (при условии правильного предсказания адреса запроса префетчером);
  - времени доступа к DRAM (при условии неверного предсказания адреса запроса префетчером);
  - пропускной способности L1 кэша;
  - времени поиска страницы при промахе в TLB.

В рамках продолжения работы над развитием инструмента планируется

- добавление новых параметров конфигурации;
- поддержка новых режимов тестов на подсистему памяти;
- поддержка генерации тестов не только на подсистему памяти;
- разработка новых алгоритмов генерации тестов с целью преодоления существующих ограничений на параметры конфигурации.

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).