Министерство образования и науки Российской Федерации
Московский физико-технический институт
(национальный исследовательский университет)

Физтех-школа радиотехники и компьютерных технологий
Кафедра микропроцессорных технологий в интеллектуальных системах управления

Выпускная квалификационная работа бакалавра

Модель производительности центрального процессора в рамках ядра Linux

Автор:
Студент Б01-008 группы
Глаз Роман Сергеевич

Научный руководитель:
Кринов Пётр Сергеевич, к. ф.-м. н.

Москва 2024

Аннотация

Модель производительности центрального процессора в рамках ядра Linux

Глаз Роман Сергеевич

На данный момент в мире насчитывается более 4-ёх миллиардов пользователей смартфонов. Несмотря на широкий диапазон возможностей смартфона, производительность, длительное время автономной работы и надежность являются основными критериями его конкурентоспособности.
Преобладающая часть смартфонов использует архитектуру ARM big.LITTLE – архитектура ARM/ARM64, основой которой является использование различных типов ядер – энергоэффективных с низкой производительностью и высокопроизводительных, имеющих большое энергопотребление.
Также одной из основных технологий, используемых процессорами ARM, является Dynamic Voltage and Frequency Scaling (DVFS), благодаря которой удаётся снизить энергопотребление процессора, снижая его частоту работы и напряжение, если нагрузка на процессор небольшая.
Для разработки алгоритмов планировщика операционных систем, который учитывает как наличие ядер разной производительности, так и технологию DVFS, очень удобным инструментом является симулятор микроархитектуры, позволяющий использовать подробную статистику использования различных компонент симулируемой системы. В данной работе используется симулятор Gem5 и Linux в качестве исследуемой операционной системы.
В данной работе:

1. Разработаны компоненты симулятора Gem5, предоставляющие возможность создавать гетерогенные (мультикластерные) архитектуры с возможностью использования ARM Performance Monitor Unit (PMU) счётчиков, связанных с иерархией кешей.

2. Реализованы драйверы ядра Linux (версии 6.1) для платформ Gem5, предоставляющие поддержку DVFS как для процессоров (cpufreq драйвер), так и прочих компонент системы (компоненты devfreq драйвера).

3. Создана модель, определяющая влияние частот процессора и прочих компонент системы на производительность процессора для заданной рабочей нагрузки.

4. Разработано решение в подсистеме планировщика ядра Linux, использующее предлагаемую модель для контроля частоты процессорных ядер.

Содержание

1 Введение

Для улучшения производительности компьютерных систем, а также для уменьшения потребляемой этими системами энергии, в различных операционных системах используют подсистемы, ответственные за регулирование частот как центрального процессора (Central Processing Unit – CPU), так и прочих компонент.

В операционной системе Linux, представляющую собой монолитное ядро, частоты процессора, как правило, меняются посредством использования драйвера *cpufreq*, предоставляющий API (Application Programming Interface) для регистрации интерфейса изменения частот произвольного CPU. За частоты прочих компонент системы отвечает драйвер-API *devfreq* (например, изменение частоты шины, соединяющую соседние уровни кешей, или изменение частоты работы DDR памяти).

На данный момент Linux использует политики изменения частот процессора (governor policies), реализация которых не учитывает сложную структуру компьютерной системы, в которой могут присутствовать компоненты, работающие при частоте, отличной от CPU: кеши высоких уровней (L3 кеш, системный кеш), оперативная память (Random Access Memory – RAM).

Если же попробовать взять во внимание внепроцессорные компоненты, возникает ряд трудностей:

1. Для работы такой политики требуется знать структуру системы, на которой запущена операционной система, например, количество уровней кешей и уровни кешей, работающие на частоте, отличной от частоты CPU.

2. Кроме структуры системы, необходимо заранее измерить подробные характеристики рассматриваемых компонент системы, например, зависимость задержки памяти (*latency*) от уровня её утилизации (*bandwidth*).

3. Даже при готовом алгоритме часть информации, такую как утилизация оперативной памяти (*RAM bandwidth*) или какие блоки процессора были использованы за фиксированный интервал времени, получить во время работы операционной системы либо затруднительно, либо вовсе невозможно.

Архитектура *ARM big.LITTLE* представляет собой концепцию, в которой одновременно используются различные типы ядерных кластеров: кластеры, состоящие из более энергоэффективных, но менее производительных ядер, и производительные кластеры, потребляющую большое количество энергии.

В системах, использующих архитектуру *ARM big.LITTLE*, которая является одной из самых используемых на данный момент времени, параллельно с задачей выбора частот процессорных ядер возникает задача выбора ядерного кластера в зависимости от требований и специфики выполняемой задачи. Обе задачи имеют одинаковую цель: уменьшить потребляемую энергию системы, сохранив требуемую производительность.

В рамках описываемой работы решается только первая из перечисленных задач: выбор частот процессорных ядер, однако в любом случае потребуется построение моделей для алгоритма изменения частот процессорных ядер отдельно для каждого имеющегося кластера.

ARM использует технологию *DVFS* (Dynamic Voltage-Frequency Scaling) для изменения частот процессорных ядер, а *cpufreq*, в свою очередь, является интерфейсом для работы ядра операционной системы Linux с этой технологией.

Для исследования выше обозначенных задач удобно использовать симулятор компьютерных архитектур, например, *Gem5*, который и будет использоваться в данной работе. *Gem5* позволяет создать систему, подобную системе, построенную на архитектуре

ARM big.LITTLE, т.е. гетерогенную (мультикластерную) систему на основе архитектуры *ARM/ARM64*.

На данный момент сообществом *Gem5* уже реализазованы такие важные компоненты, как *DVFSHandler* (обработчик *DVFS*) и *EnergyController* (энергетический контроллер), которые предоставляют возможность реализации собственного *cpufreq* драйвера, котролирующего частоту так называемого *ClockDomain*, а также прочих драйверов для работы *DVFS* технологии остальных компонент системы. Однако никто не предпринимал попытки реализовать *DVFS* драйвер в рамках *Gem5* для внепроцессорной подсистемы (через *devfreq* API), а существующая реализация поддержки *DVFS* для процессорных ядер через *cpufreq* сильно устарела (исходно написана для ядра Linux версии 3.x) и реализована с рассчётом на то, что ни один другой драйвер не сможет переиспользовать текущую реализацию в своих целях (например, любой *devfreq* драйвер).

Очень важной компонентой алгоритмов реального времени, использующих знания об архитектуре процессора, является использование счётчиков производительности (performance counters), которые предоставляют информацию об определённых архитектурных и микроархитектурных событиях (например, количество исполненных процессорных инструкций или количество обращений процессорного ядра в кешу уровня L2). ARM процессоры имеют такие счётчики – PMU (Performance Monitoring Unit) и AMU (Activity Monitoring Unit) счётчики. Именно с помощью них становится возможным создать алгоритм изменения частот, который учитывает как микроархитектурные события процессора при исполнении определённой рабочей нагрузки, так и возможность построения алгоритма для ядер различной микроархитектуры (производительные или энергоэффективные ядра).

В *Gem5*, к сожалению, реализованы только самые простые PMU события, такие как количество исполненных инструкций и количество процессорных циклов, полностью отсутствуют события, связанные с кешами. Одна из задач текующей работы – предоставить возможность работы в *Gem5* с PMU событиями, связанными с кешами.

2 Постановка задачи

3 Обзор существующих решений

3.1 Микроархитектура процессора

Современный CPU (Central Processing Unit – центральный процессор) представляет собой систему взаимосвязанных между собой процессорных ядер, не обязательно одинаковых.

Исполнение инструкции современным процессорным ядром является многостадийным pipeline'ом (конвейером) из различных блоков. В зависимости от исполняемого workload'а утилизируются различные блоки, поэтому скорость исполнения workload'а сильно зависит от его особенностей (типа).

В наиболее общем виде можно разбить pipeline исполнения инструкции на процессоре на несколько последовательных стадий:

1. Fetching (чтение инструкции из памяти);
2. Decoding (декодирование прочитанной инструкции);
3. Execution (исполнение инструкции);
4. Memory (чтение/запись в память при необходимости);
5. Writeback (запись результата инструкции в регистры процессора).

Чаще всего приведённые выше стадии дробятся на более локальные стадии, применяются дополнительные оптимизации для ускорения исполнения инструкций а также их распараллеливания (например, кеширование декодированных инструкций и предсказатель ветвлений). Наиболее актуальным примером являются OoO (Out-of-Order – внеочередное исполнение) процессоры, в которых используется алгоритм Томасуло, позволяющий реализовать исполнение машинных инструкций не в порядке их следования в машинном коде, а в порядке готовности к выполнению, за счёт чего значительно увеличивается скорость исполнения инструкций.

В типичной реализации OoO используется ROB (Re-order buffer), который представляет собой циклический буфер и накапливает инструкции для обеспечения возможности их переупорядочить. Обычно в ROB попадают не исходные машинные иструкции, а прошедшие через стадию register-renaming (переименование регистров) для избавления от data-hazards (зависимости по данным), а значит для дополнительного распараллеливания инструкций.

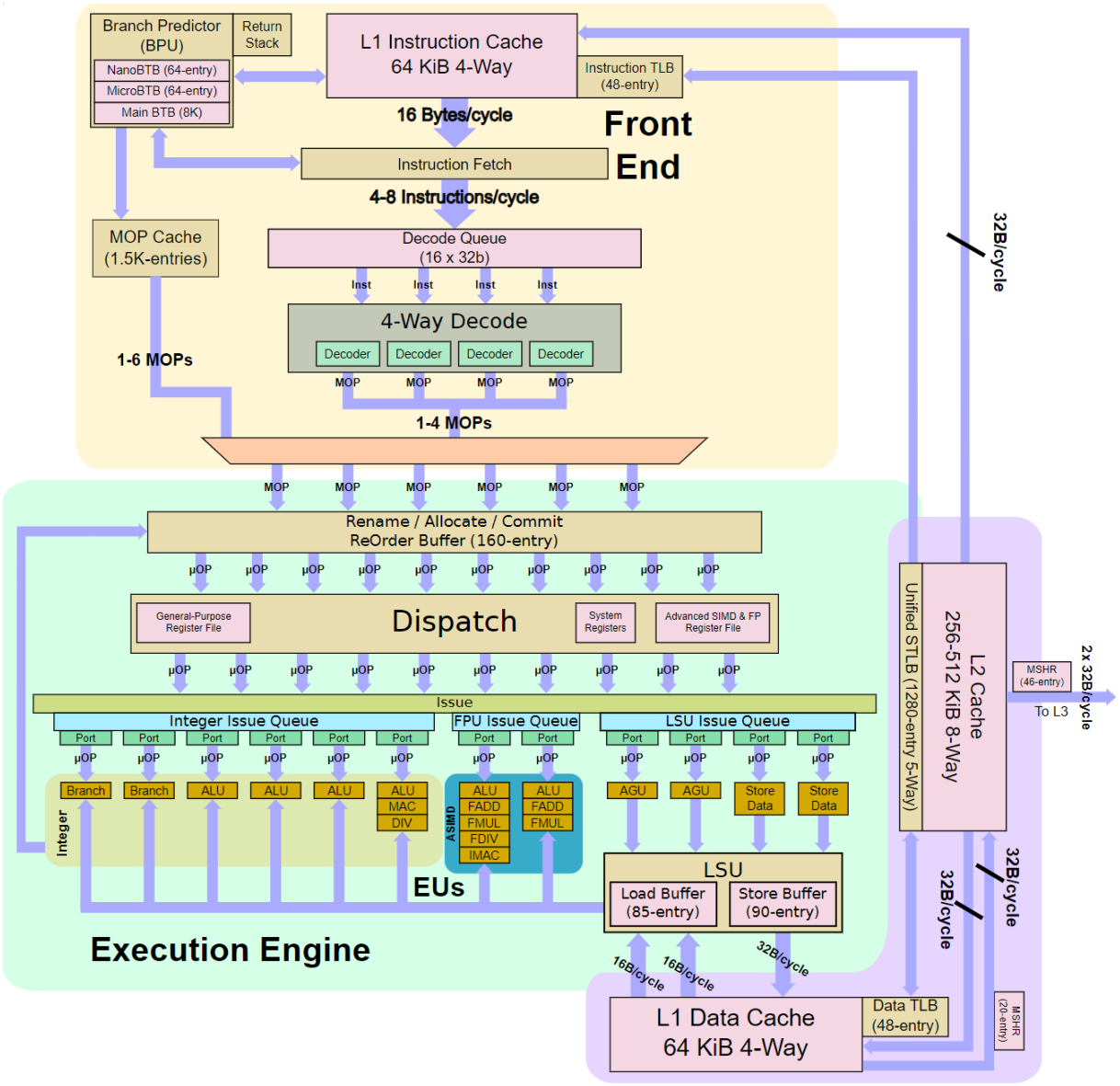
Наибольший интерес в данной работе представляет организация обращения процессора в память, также затрагивается аспекты, связанные с OoO исполнением.

3.2 Подсистема памяти в современных SoC

Процессор в современным мобильных системах встроен в так называемую SoC (System-on-chip – система на кристалле) систему – электронная схема, которая выполняет цели компьютера и размещена на одной интегральной схеме. Таким образом, в такую систему встроены сразу CPU, таймеры, счётчики, интерфейсы для периферийных устройств, ОЗУ, ПЗУ и даже GPU (Graphics Processing Unit – графический процессор). Именно таким образом устроены современные смартфоны, фотоаппараты, умные часы, элекронные книги и схожие устройства.

Одна из подсистем SoC,

Рис. 1: Схема конвейера процессора Arm Cortex A77 [1]



3.3 Исследования на тему bandwidth и latency

[TODO] В работе [2] рассматривается подход использования аналитической формулы влияния bandwidth и latency памяти на производительность суперскалярного процессора в рамках рабочих нагрузок, связанных с big data областью. В этом подходе используется схожий принцип, предлагаемых в данной работе: использование показателя *cpi*, который включает в себя слагаемые, связанные с циклами, потраченными на время ожидания транзакций-обращений в память. Авторы исследуют влияние значений bandwidth и latency на производительность, которую они выражают через *cpi*, для различных workload'ов: latency-bound и bandwidth-bound. Однако данное исследование ограничивается рассмотрением обращений только в DDR память при фиксированных частотах всех устройств.

[TODO] Авторы работы [3] предлагают 2 модели для учёта latency памяти в рамках DVFS алгоритма. Первая модель разбивает циклы CPU на 2 компоненты: относящиеся к ожиданию транзакций памяти и относящиеся непосредственно к вычислительным блокам CPU. Утверждается, что при изменении частоты CPU изменяются только циклы, относящиеся к памяти, на этом и основывается алгоритм DVFS, использующий

перерасчёт времени исполнения через циклы при различных частотах CPU. Вторая модель является модификацией-улучшением первой модели: дополнительно предполагается, что в группе инструкций, которые обращаются в память подряд, следует учитывать только первую инструкцию в формуле для перерасчёта циклов, которые относятся к памяти, однако это требует наличие дополнительной информации на уровне кешей: среди всех промахов в кеши (т.е. обращений в следующий уровень памяти) следует учитывать только первые промахи среди группы промахов (промахов, находящихся на расстоянии порядка latency обращения в вышележащие уровни памяти во времени).

В работе также отмечается, что следует учитывать ROB в OoO процессорах: из циклов, относящихся к транзакциям памяти, следует вычесть ту часть циклов, которую тратит CPU на исполнение инструкций, расположенных в логическом порядке после инструкции-обращений в память, так как такие инструкции исполняются спекулятивно.

3.4 Симулятор архитектуры Gem5

3.5 Capacity Awate Scheduling в ядре Linux

3.6 DVFS алгоритмы

4 Исследование и построение решения задачи

4.1 Описание модели производительности процессора

Главной характеристикой производительности процессора является количество инструкций, исполняемых в еденицу времени – чем больше это значение, тем быстрее исполняется рабочая нагрузка (программа). Причём заданное количество инструкций исполнется за разное количество процессорных циклов, которое, в свою очередь, обратно пропорциональны времени исполнения этих инструкций.

Пусть за время τ процессор исполнил $instrs$ инструкций за $cycles$ циклов при заданной частоте процессора $freq_{cpu}$. Тогда, очевидно, выполняется следующее соотношение:

$$cycles = \frac{freq_{cpu}}{\tau} \tag{1}$$

На практике чаще всего используют такие величины как $cpi = cycles/instrs$ и $ipc = instrs/cycles$ в качестве меры производительности процессора: чем выше/ниже значение ipc/cpi, тем лучше процессор. Однако кроме типа процессорного ядра на эти значения также влияют частота самого ядра и частоты остальных компонент системы. Например, при повышении частоты процессора величина ipc либо остаётся такой же, что возможно при так называемой cpu-bound нагрузке (непосредственно связанной с исполнением на самом ядре), либо уменьшается, что характерно для memory-bound нагрузок (процессор часто обращается в кеши или оперативную память, из-за чего циклы тратятся впустую).

За время τ процессор часть циклов тратит на исполнение инструкций непосредственно на процессорных блоках (в том числе вычислительных), а оставшуюся часть на ожидание операций, связанных с обращением в память (в кеши или оперативную память): обозначим эти величины $cycles_{cpu}$ и $cycles_{mem}$, тогда справедливо

$$cpi = \frac{cycles}{instrs} = \frac{cycles_{cpu}}{instrs} + \frac{cycles_{mem}}{instrs} \tag{2}$$

Важно отметить, что $\frac{cycles_{cpu}}{instrs}$ – значение cpi, если бы все обращения в кеши и оперативную память занимали 0 циклов, т.е. работали бы бесконечно быстро, а значит количество циклов ограничивалось снизу возможностями процессора. Обозначим данное соотношение как $cpi_{cpu} \equiv \frac{cycles_{cpu}}{instrs}$.

В свою очередь величина $cycles_{mem}$ характеризуется только лишь внешними компонентами системы, не зависящими от процессора. Пусть, например, если имеется 2 уровня кешей и оперативная память. Время доступа к определённому уровню кеша характеризуется средней величиной $cycles_{L_i}$, представляющую собой латентность кеша (время задержки обращения), выраженную в процессорных циклах, где i – номер уровня кеша. После обращения в кеш возможны 2 ситуации: либо попадание к кеш, либо промах и обращение в следующий уровень кеша или оперативную память. Время доступа к оперативной памяти обозначим $cycles_{ram}$.

Выражая времена доступа через собственные частоты и латентности, получим:

$$cycles_{mem} = n_{L_1} \cdot \frac{lat_{L_1}}{freq_{L_1}} \cdot freq_{cpu} + n_{L_2} \cdot \frac{lat_{L_2}}{freq_{L_2}} \cdot freq_{cpu} + n_{ram} \cdot \frac{lat_{ram}}{freq_{ram}} \cdot freq_{cpu}, \tag{3}$$

$$cpi = cpi_{cpu} + \frac{n_{L_1} \cdot \frac{lat_{L_1}}{freq_{L_1}} + n_{L_2} \cdot \frac{lat_{L_2}}{freq_{L_2}} + n_{ram} \cdot \frac{lat_{ram}}{freq_{ram}}}{instrs} \cdot freq_{cpu}, \tag{4}$$

где lat_{L_1}, lat_{L_2} и lat_{ram} – латентности, выраженные в собственных циклах (а не процессорных), а n_{L_1}, n_{L_2} и n_{ram} – количество обращений к соответствующим уровням кешей или оперативной памяти за время τ.

Заметим, что из формулы следует, что повышение частоты процессора ведёт к увеличению *cpi* и уменьшению *ipc*, а повышение частот кешей и оперативной памяти, наоборот, ведёт к уменьшению *cpi* и увеличению *ipc*.

Чаще всего уровни кешей, наиболее близкие к процессорному ядру, имеют такой же источник тактирования, что и процессорное ядро, т.е. такие кеши оперируют на тех же частотах, что и сам процессор. Например, если бы кеш L1 оперировал с частотами процессора ($freq_{L_1} = freq_{cpu}$), тогда формулу можно упростить до более простой:

$$cpi = cpi_{cpu} + n_{L_1} \cdot \frac{lat_{L_1}}{instrs} + \frac{n_{L_2} \cdot \frac{lat_{L_2}}{freq_{L_2}} + n_{ram} \cdot \frac{lat_{ram}}{freq_{ram}}}{instrs} \cdot freq_{cpu} = \tag{5}$$

$$= cpi_{cpu}^{L_1} + \frac{n_{L_2} \cdot \frac{lat_{L_2}}{freq_{L_2}} + n_{ram} \cdot \frac{lat_{ram}}{freq_{ram}}}{instrs} \cdot freq_{cpu}, \tag{6}$$

то есть можно занести константную часть для заданной нагрузки, незавсимую от каких-либо частот, в cpi_{cpu}, что упрощает формулу. Однако, теперь $cpi_{cpu}^{L_1}$ зависит не только от характера утилизации нагрузкой процессорных блоков, как cpi_{cpu}, но также и от характеристик кеша L1 и количества обращений в него (т.е. от уровня его утилизации).

Латентности кешей и оперативной памяти могут и не быть константными значениями, т.к. могут зависеть от характера и частоты обращений к ним. Так, латентность оперативной памяти определяется как текущим уровнем утилизации этой памяти (зависимость latency от bandwidth), так и особенностями обращения к ней (например, в случае DDR памяти, где элементарными ячейками являются банки данных, состоящие из строк и столбцов, существует кеш строки, который значительно влияет на скорость обращения к ячейке памяти, т.е. локальность обращения к ячейке DDR памяти очень важна).

У описываемой модели есть ряд преимуществ, которые делают её лучше аналогичных моделей:

1. Построив модели, с помощью которых можно вычислять lat_{L_2} и lat_{ram}, в режиме реального времени можно вычислить величину $cpi_{cpu}^{L_1}$ для заданной нагрузки без построения соответствующей модели: достаточно знать значения $cycles$, $instrs$, n_{L_2} и n_{ram}.

2. Зная величину $cpi_{cpu}^{L_1}$, можно вычислить *cpi* (а значит и *ipc*) для любого набора частот процессора и прочих компонент, частоты которых входят в вышеописываемую формулу, причём не требуются знания о характере событий, происходящих внутри процессора, и даже о событиях, происходящих внутри кеша уровня L1.

3. В случае физического ядра, разделённого на 2 виртуальных (так называемый SMT - Simultaneous multithreading), по-прежнему можно пользоваться формулой выше, т.к. становится неважно, как именно виртуальные ядра разделяют между собой процессорные блоки и как именно они взаимодействуют с кешом уровня L1.

Таким образом, для применения модели необходимо:

1. Иметь готовые модели для поиска lat_{L_2}, lat_{ram}.

2. Уметь каким-либо образом вычислять значения $cycles$, $instrs$, n_{L_2} и n_{ram}.

3. Знать списки возможных частот $freq_{cpu}$, $freq_{ram}$, $freq_{L_2}$.

4.2 Модель производительности применительно к архитектуре ARM

Рассмотрим способ применения описанной выше модели в случае архитектуры ARM. Для подсчёта значений $cycles$, $instrs$, n_{L_2} и n_{ram} можно ипользовать следующие PMU счётчики процессора:

1. CPU_CYCLES – количество затраченных процессорных циклов;
2. $INST\_RETIRED$ – количество исполненных инструкций;
3. $L1I\_CACHE\_REFILL$ – количество чтений инструкций (instruction fetches), которые отсутствуют в кеше инструкций уровня L1 (промах в L1 кеш инструкций), поэтому инструкции вынуждены читаться из кешей более высокого уровня. Некешируемые промахи в кеш и операции синхронизации кешей не считаются;
4. $L1D\_CACHE\_REFILL$ – количество чтений и записей данных (data loads and stores), или операций обращений в таблицу страниц (page table walks), которые не смогли найти нужную информацию в кеше данных уровня L1 (промах в L1 кеш данных), поэтому данные вынуждены выгружаться из кешей более высокого уровня. Некешируемые промахи в кеш и операции синхронизации кешей не считаются;
5. $L2D\_CACHE\_REFILL$ – работает как сумма счётиков $L1D\_CACHE\_REFILL$ и $L1I\_CACHE\_REFILL$, но применительно к уровню кеша L2. В случае промаха в кеш дальнейшее обращение может происходить не в кеш более высокого уровня, если таковой отсутствует, а сразу в оперативную память.

В счётчиках типа $\_CACHE\_REFILL$ существует очень важный нюанс, не оговорённый в спецификациях PMU счётчиков компании ARM Ltd., но упоминаемый в иных документах: в платформах архитектуры ARM не существует PMU счётчиков, которые считают промахи в кеши, вместо них используются счётчики, считающие количество перезапонений кеш-линий (кешами более выского уровня ил оперативной памятью). Возможна ситуация, когда один промах в кеш вызывает несколько перезаполнений кеш-линий (например, если промах случился по адресу, который пересекает 2 соседние кеш-линии), или наоборот: несколько промахов в кеш могут быть разрешены благодаря одному перезаполнению кеш-линии.

Заметим, что чаще всего в современных мобильных системах на архитектуре ARM после последнего уровня процессорного кеша располагают дополнительный кеш – системный кеш (system cache), который предназначен для периферийных устройств. Он может быть как больше по размеру, чем последний уровень процессорного кеша, так и меньше. Наличия системного кеша ведёт к тому, что для определения значения n_{ram} недостаточно воспользоваться счётчиками, описанными выше.

Единственный счётчик, который умеет считать количество промахов в кеш, является $LL\_CACHE\_MISS\_RD$. Причиной тому является то, что кеш последнего уровня, как правило, является системным кешом, который находится за пределами CPU на чипе. Данный счётчик считает количество промахов, когда в результате данные берутся из оперативной памяти, из другого чипа или из соседнего процессорного кластера с использованием технологии Intercluster peering. Нас интересует только первый случай.

Согласно документации ARM, если счётчик $LL\_CACHE\_MISS\_RD$ не реализован, то есть бит $EXTLLC$ в регистре $CPUECTLR$ выставлен в ноль, то значения $LL\_CACHE\_MISS\_RD$ будут совпадать со значениями $L2D\_CACHE\_REFILL$, так что в случае наличия системного кеша такая ситуация может заметно ухудшить

работу модели (по-прежнему предполагается система из 2 уровней кеша и возможно системного кеша).

4.3 Модель производительности применительно к ядру Linux

4.4 Реализация модели в планировщике ядра Linux

5 Описание практической части

6 Заключение

Список литературы

[1] Overview Cortex A77. — Cortex A77 Documentation : 2024. — https://en.wikichip.org/wiki/arm_holdings/microarchitectures/cortex-a77.

[2] Clapp Russell, Dimitrov Martin, Kumar Karthik, Viswanathan Vish, and Willhalm Thomas. Quantifying the performance impact of memory latency and bandwidth for big data workloads // 2015 IEEE International Symposium on Workload Characterization / IEEE. — 2015. — P. 213–224.

[3] Keramidas Georgios, Spiliopoulos Vasileios, and Kaxiras Stefanos. Interval-based models for run-time DVFS orchestration in superscalar processors // Proceedings of the 7th ACM international conference on Computing frontiers. — 2010. — P. 287–296.

[4] Ltd. Arm. — Arm Neoverse N1 Core: Performance Analysis Methodology : 2021.

[5] Documentation Linux Kernel. — Capacity Aware Scheduling : 2024. — https://docs.kernel.org/scheduler/sched-capacity.html.

Приложение

Список используемых обозначений и терминов

1. CPU (Central Processing Unit) – ЦП (центральный процессор);
2. CPU core – ядро центрального процессора;
3. *cycles* – количество циклов процессорного ядра за определённый промежуток времени;
4. *instrs* – количество исполненных инструкций процессорным ядром за определённый промежуток времени;
5. $ipc = \frac{instrs}{cycles}$, $cpi = \frac{cycles}{instrs}$;
6. Pipeline – конвейер (обычно применятся в контексте схемы исполнения инструкции процессорным ядром);
7. OoO (Out-of-Order) – парадигма в современных высокопроизводительных CPU, основанная на спекулятивном исполнении инструкций для повышения значения *ipc*;
8. ROB (Re-order buffer) – циклический буфер, используемый для работы алгоритма Томасуло для поддержки OoO в CPU;
9. Data hazard – зависимость по данным (употребляется в контексте использования одинаковых регистров в соседних инструкциях, из-за чего инструкции необходимо исполнить строго в порядке их логического следования).
10. SoC (System-on-Chip) – интегральная схема, включающая в себя большинство или все компоненты компьютерной системы, такие как CPU, интерфейсы памяти, устройства ввода-вывода и так далее;
11. GPU (Graphics Processing Unit) – графический процессор;
12. Latency – задержка (латентность) в рамках какого-либо события (например, обращения в память), выраженная во времени или циклах какого-то устройства;
13. Throughput – утилизация устройства, выраженная в единицах транзакций, связанных с этим устройством, в единицу времени;
14. Bandwidth – пропускная способность устройства (максимально возможная утилизация устройства);
15. Workload – определённая рабочая нагрузка, исполняемая на устройстве;
16. DDR SDRAM (Double Data Rate Synchronous Dynamic Random-Access Memory) – класс интегральных схем памяти, используемой в компьютерах (обычно в качестве оперативной памяти);
17. LPDDR SDRAM (Low-Power Double Data Rate SDRAM) – класс интегральных схем памяти, спецификой которого является низкое потребление энергии (как правило, используемый в мобильных компьютерах);
18. DDR (Double Data Rate) – общее обозначение, которое подразумевает DDR SDRAM или LPDDR SDRAM память;