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

Политика регулирования частот центрального процессора в ядре Linux для приложений реального времени

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 Обзор архитектуры процессор-память

3.1.1 Микроархитектура процессора

Современный ЦП (центральный процессор) представляет собой систему взаимосвязанных между собой процессорных ядер, не обязательно одинаковых.

Исполнение инструкции современным процессорным ядром является многостадийным конвейером из различных блоков. В зависимости от исполняемой рабочей нагрузки утилизируются различные блоки процессора, поэтому скорость исполнения таковой рабочей нагрузки сильно зависит от её особенностей (типа) и особенностей исполнения таковых нагрузок на блоках процессора.

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

1. Чтение инструкции из памяти;
2. Декодирование прочитанной инструкции;
3. Исполнение инструкции на вычислительных блоках;
4. Чтение/запись в память при необходимости;
5. Запись результата исполнения 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 или спекулятивное 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 переименование регистров для избавления от зависимостей по данным (для дополнительного распараллеливания инструкций).

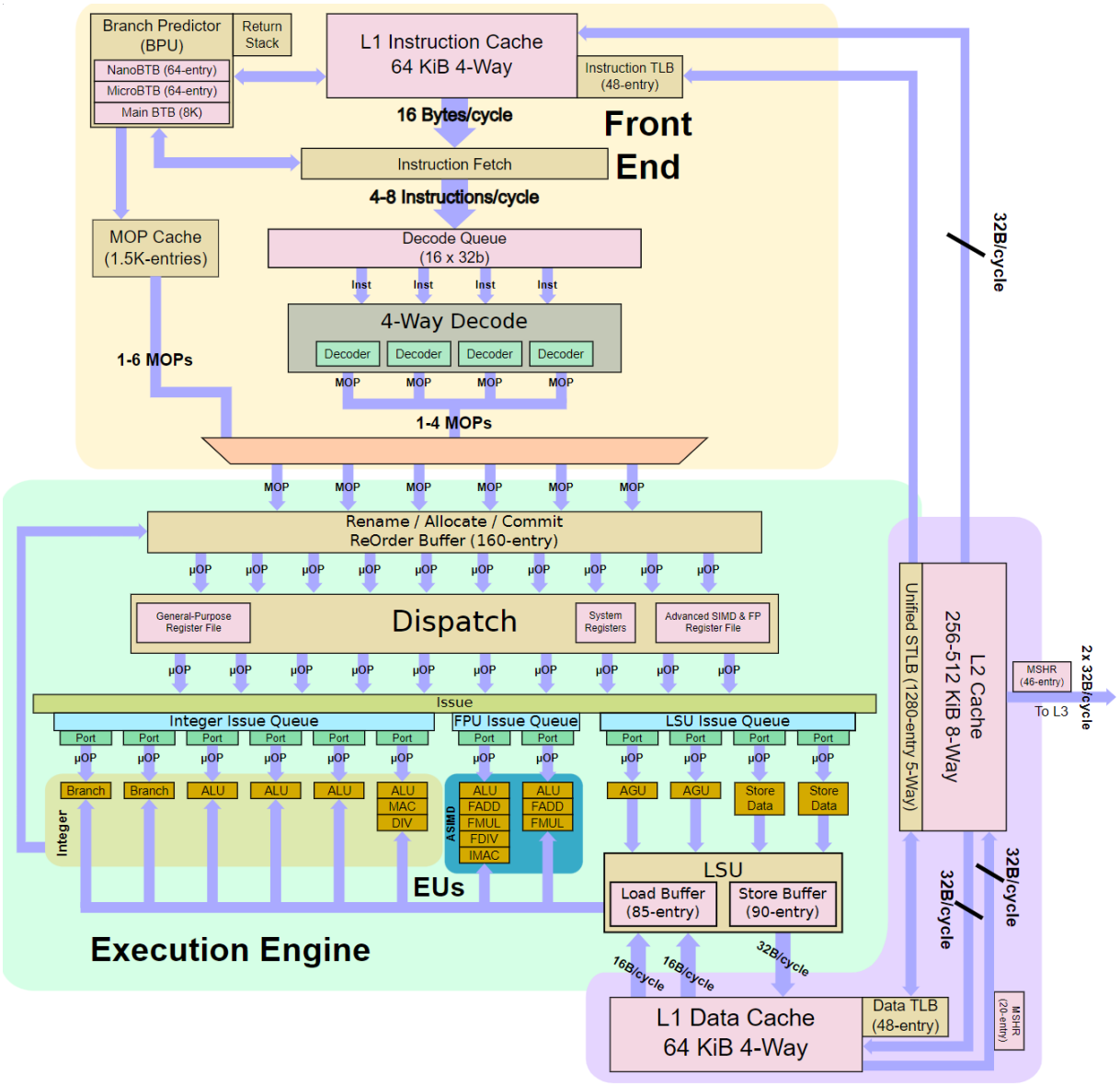
Наибольший интерес в данной работе представляет организация обращения процессора в память, также затрагиваются аспекты, связанные с OoO исполнением.

3.1.2 Организация памяти

Процессор в современных 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 процессор, таймеры, счётчики, интерфейсы для периферийных устройств, ОЗУ, ПЗУ и даже графический ускоритель. Именно таким образом устроены современные смартфоны, фотоаппараты, умные часы, элекронные книги и схожие устройства.

Одна из основных подсистем системы на кристалле – подсистема памяти. Наиболее быстродействующими элементами памяти являются регистры процессора, они же

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



имеют наименьшее количество памяти, более высокую цену для производства и самую малую плотность расположения в электронной схеме.

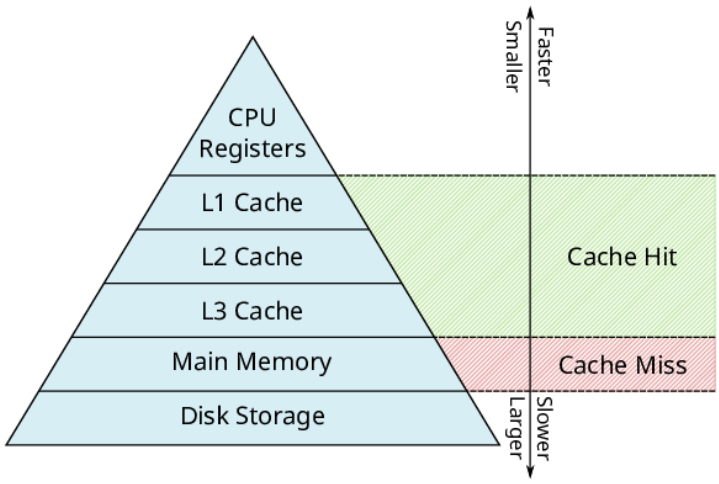
Учитывая малый объём возможного хранения данных с помощью регистров, а также что любая вычислительная система обладает локальностью (как по данным, так и по времени), почти всегда вводят дополнительные уровни памяти – более дешёвые в производстве, имеющий больший объём и более высокую плотность ячеек хранения данных на электронной схеме. Более низкие уровни памяти являются кешами для более высоких. Таким образом, любая система имеет ОЗУ и кеши в качестве промежуточного хранилища данных (см. рис. 2).

Во всех современных мобильных системах в качестве ОЗУ используется LPDDR SDRAM (Low-Power Double Data Rate Synchronous Dynamic Random-Access Memory – динамическая оперативная память синхронного доступа с двойной скоростью передачи данных и с низким энергопотреблением). В дальнейшем для удобства будем использовать более простое обозначение для такой памяти – DDR (Double Data Rate) память.

Количество уровней кешей и объём их памяти напрямую зависят от требований к исполнению современных приложений и особенностей архитектуры мобильной системы,

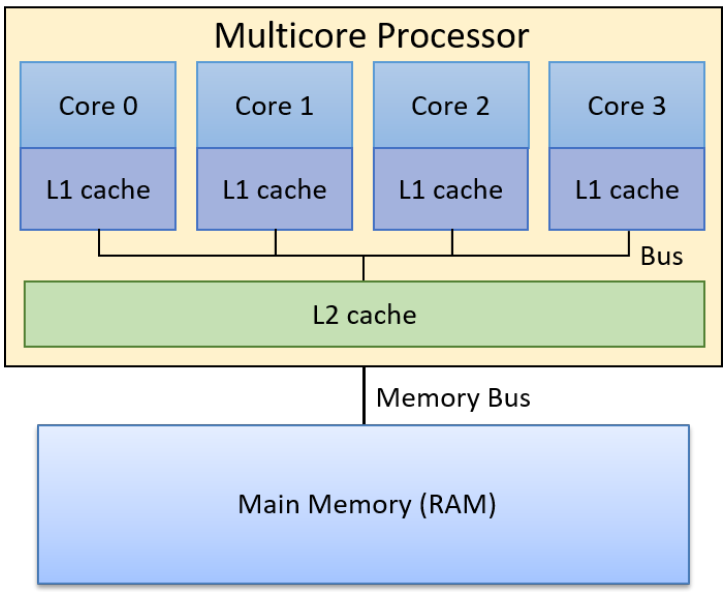
всегда являются компромиссом: с одной стороны, добавление дополнительного уровня кеша – снижение вероятности обращения в DDR память (наиболее медленная память), а с другой – введение постоянной дополнительной задержки при обращении в DDR, если данные в кешах отсутствуют.

Рис. 2: Пример иерархии памяти в системе с 3-мя уровнями кешей



Чаще всего производят мобильные устройства с 2-мя и более уровнями кешей. В системах на кристалле, как правило, последним (дополнительным) уровнем является системный кеш, который обычно не вносят в список уровней кешей процессора, так как он используется не только самим процессором, но также и различными периферийными устройствами, такими как графический ускоритель, ускоритель нейронных сетей и т.д..

Рис. 3: Пример иерархии кешей в многоядерном процессоре (отсутствуют кеш 3-его уровня и системный кеш)



Каждое ядро процессора имеет кеш инструкций и кеш данных 1-ых уровней, на которые для краткости ссылаются как на единый кеш 1-ого уровня. При промахе в кеш 1-ого уровня поиск данных происходит в кеше 2-ого уровня, при промахе в кеш 2-ого –

поиск в кеше 3-его (если таковой имеется), и так до тех пор, пока не возникнет промах в системный кеш, после чего произойдёт обращение в DDR память (см. рис. 3).

Как правило, кеш 2-ого уровня уникален для каждого ядра, хотя в некоторых гетерогенных системах кеш 2-ого уровня может использовать либо группой из 2-ух ядер в марках 1-ого кластера, либо целиком всем кластером ядер (такое поведение характерно для энергоэффективных ядер). Кеш 3-его уровня всегда разделяется между всеми ядрами системы, как и системный кеш вместе с ОЗУ.

ПЗУ, а также компоненты для долгосрочного хранения данных (диски, твёрдотельные накопители и др.) в данной работе рассматриваться не будут.

3.1.3 Технология DVFS

Почти все электронные схемы являются логическими элементами, которые имеют источник тактирования и работают на определённой частоте. Однако некоторым устройствам не выгодно работать при строго фиксированной частоте, например, если ядро процессора часто ждёт операции ввода-вывода, то эффективнее снизить рабочую частоту для более экономного расходования энергии, не сильно потеряв в производительности.

Технология DVFS (Dynamic Voltage-Frequency Scaling) позволяет в режиме реального времени регулировать частоту и напряжение, с которыми работает устройство. Как правило, возможные значения частот являются дискретным набором, причём для каждой частоты существует минимальное значение напряжения, при котором устройство всё ещё остаётся в работоспособном состоянии.

Практически все современные процессоры используют DVFS для регулировки частот ядер. В случае гетерогенных систем, т.е. имеющих кластеры ядер с различными характеристиками (например, более энергоэффективные или более производительные ядра), коими на сегодняшний день являются большинство мобильных систем, предоставляется возможным регулировать частоты только целиком всего кластера, то есть всех ядер, расположенных в нём, а не каждого ядра по отдельности.

Регулирование частот и напряжений используется не только в случае процессорных ядер, но также и в других компонентах компьютерных систем: ОЗУ, шины, кеши и т.д..

Наибольший интерес в данной работе представляет алгоритм регулирования частот процессорных ядер, регулирование частот прочих компонент рассматриваться не будет.

3.2 Алгоритмы и эвристики регулирования частот процессора

3.2.1 Политики регулирования частот в ядре Linux

Самым распространённым способом регулирования частот ядер процессора в ядре Linux является использование драйвера CPUfreq, который предоставляет несколько политик регулирования [2]:

1. ”Performance” – частота ядра процессора выставляется в максимальное значение.
2. ”Powersave” – частота ядра процессора выставляется в минимальное значение.
3. ”Userspace” – частота ядра процессора выставляется пользователем.
4. ”Ondemand” – частота ядра процессора выставляется в зависимости от утилизации ядра за последний промежуток времени.
5. ”Conservative” – аналогично ”Ondemand”, но частота меняется менее резко, небольшими шагами.

6. "Schedutil" [3] – аналогично "Ondemand", но утилизация отслеживается не для ядер процессора, а для каждого потока исполнения. Дополнительно используется более продвинутая система подсчёта утилизации потока исполнения: не только за последний промежуток времени и с учётом весов каждого промежутка времении (более давние промежутки менее важны). В случае использования не CFS (Completely Fair Scheduler), а дедлайн планировщика (DL) или реального времени (RT) частота выставляется в максимальное значение.

Политика "Performance" приводит к излишнему энергопотреблению, "Powersave" – к потере производительности. "Ondemand" и "Conservative" – не отслеживают утилизацию отдельно по потокам исполнения, к тому же не учитывают, что утилизация может быть нелинейна относительно частоты (например, если процессор проводит основное время в ожидании выполнения транзакции-чтения из памяти), поэтому могут выставлять заведомо завышенное значение частоты, что приводит к дополнительному энергопотреблению.

"Schedutil" устраняет большинство недостатков политик регулирования частот, рассмотренных выше, но всё же он не учитывает возможную нелинейность утилизации от частоты ядра процессора.

Ещё один существенный недостаток всех политик, рассмотренных выше, является предположение, что частоты любого ядра процессора может изменяться независимо от остальных. Как было упомянуто в 3.1.3, в современных мобильных системах ядра группируются в кластеры и частоты могут меняться только целиком для всего кластера.

3.2.2 Подсчёт утилизации в планировщике ядра Linux

В ядре Linux на данный момент в стандартном планировщике CFS (Completely Fair Scheduler) используется решение [4], в котором при подсчёте утилизации ядра процессора рабочей нагрузкой учитывается тот факт, что ядра различного типа (в разных кластерах) имеют разную производительность, и что в рамках одного ядра производительность меняется в зависимости от выбранной частоты.

Однако авторы такого решения полностью пренебрегли тем фактом, что отношение максимальных производительностей двух различных ядер не является константой, а сильно зависит от исполняемой рабочей нагрузки. Также не учтено, что зависимость производительности, измеряемой в количестве исполненных инструкций в единицу времени, обычно нелинейна относительно частоты процессора (так как скорость исполнения инструкций упирается не только в частоту ядра, но и в характеристики памяти).

Таким образом, авторы вводят величины ёмкости работы ядра процессора (максимально возможной утилизации), выражающей производительность ядра в виде максимально возможных значений выполненных инструкций в единицу времени, и утилизации ядра рабочей нагрузкой за фиксированный интервал времени τ, которая инвариантна относительно выбора ядра процессора и частоты:

$$util = \frac{\Delta t}{\tau} \cdot \frac{freq_{cpu_i}}{freq^{max}_{cpu_i}} \cdot \frac{capacity_{cpu_i}}{capacity^{max}_{cpu_i}} = inv_{i,freq}, \quad (1)$$

где Δt – суммарное время работы i-ого ядра процессора за промежуток времении τ с частотой $freq_{cpu_i}$. $freq^{max}_{cpu_i}$ – максимально возможная частота i-ого ядра процессора, $capacity_{cpu_i}$ – ёмкость i-ого ядра, характеризующая его производительность, и $capacity^{max}_{cpu_i}$ – максимальная ёмкость среди всех ядер системы.

Ёмкости вычисляются единожды с помощью измерений в конкретных сценариях исполнения (рабочих нагрузках) и применяются во всех остальных случаях без каких-

либо обоснований, причём с линейной зависимостью от частоты ядра, что тоже неверно (в большистве случаев), т.к. такая зависимость определяется рабочей нагрузкой.

В реальности используется утилизация, подсчитанная сразу за несколько промежутков времени (обычно одинаковых по длительности), например, PELT (Per Entity Load Tracking) [3] использует экспоненциально движующееся среднее значение утилизаций из подсчитанных по формулам выше за определённые промежутки времени.

В качестве альтернативы PELT, который официально представлен в ядре Linux, компанией Google был разработан WALT (Window Assisted Load Tracking) [5], который предназначен специально для мобильных устройств, где важна быстрая реакция со стороны ядра на изменение нагрузки процессора, чтобы не потерять в производительности. WALT считает утилизацию за существенно меньшее количество промежутков времени (5 вместо 32) и без экспоненциально движущихся средних значений, но алгоритм подсчёта утилизации за 1 промежуток времени совпадает с описанным выше.

При выборе политики регулирования частот в Linux "Schedutil" [3], WALT или PELT используются не только внутри планировщика, но и для выбора частоты ядра процессора, т.е. "Schedutil" переиспользует подсчитанное значение утилизации потока исполнения для регулирования частот (только в случае CFS планировщика).

Таким образом, в данное время ни стандартный планировщик, ни подсистемы регулирования частот ядра Linux не учитывают особенности влияния рабочих нагрузок на производительность ядер процессора.

3.2.3 Альтернативные алгоритмы регулирования частот

[тут очередной обзор литературы и пример работ на схожую тематику]

3.3 Латентность памяти и утилизация её пропускной способности

Для построения модели производительности ядра процессора, т.е. для правильного подсчёта утилизации ядра и наиболее энергоэффективного регулирования частотами, важно учитывать не только структуру организации памяти в исследуемой системе, а также и динамические характеристики элементов такой системы. Наиболее важными характеристиками являются пропускная способность шин, соединяющих вычислительные подсистемы с подсистемами памяти, утилизация пропускной способности и латетность памяти – промежуток времени между отправлением запроса на получение данных в память и между самим получением данных.

Таким образом, кеши и DDR память имеют свои собственные динамические характеристики, приведённые ранее. Как правило, низкие уровни кешей работают на той же частоте, что и само процессорное ядро (группа ядер – ядерный кластер), поэтому латентность таких кешей выражается через циклы ядра процессора, а не через абсолютное время. Сложнее ситуация обстоит с более высокими уровнями кешей и DDR памятью: они имеют отдельные источники тактирования, латентность выражается в аболютном времени, к тому же она зависит от утилизации пропускной способности шин, ведущих к этим элементам памяти, то есть от количества транзакций в единицу времени.

В данной работе часть предлагаемой модели требует использования латентности памяти, поэтому необходимо рассмотреть существующие решения для её определения и/или возможного учёта в модели.

Авторы работы [6] предлагают 2 модели для учёта латентности памяти в рамках алгоритма регулирования частот. Первая модель разбивает циклы процессора на 2 компоненты: относящиеся к ожиданию транзакций памяти и относящиеся непосред-

ственно к вычислительным блокам ядра процессора. Утверждается, что при изменении частоты ядра процессора изменяются только циклы, относящиеся к памяти (частота которой независима от частоты ядра). На этом и основывается алгоритм регулирования частот, использующий перерасчёт времени исполнения через циклы при различных частотах. 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 времени латентности обращения в вышележащие уровни памяти), что не поддерживается ни одним устройством.

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

В работе [7] рассматривается подход использования аналитической формулы влияния утилизации пропускной способности памяти на её латентность, что влияет на производительность суперскалярного процессора в рамках рабочих нагрузок, связанных с областью вычислений больших данных. В этом подходе используется схожий принцип, предлагаемых в данной работе: использование как показателя производительности *cpi* (отношения числа процессорных циклов к числу исполненных инструкций), которое включает в себя слагаемые, связанные с циклами, потраченными на время ожидания транзакций-обращений в память. Авторы вводят понятие блокирующего фактора для латентности: в зависимости от рабочей нагрузки при одинаковом количестве обращений в память влияние латентности на производительность (*cpi*) может меняться вплоть до десятка раз из-за параллельных обращений в память, что коррелирует с результатами работы [6]. При фиксированном блокирующем факторе значение *cpi* зависит линейно от количества обращений в память в единицу инструкций. Таким образом, все рабочие нагрузки разделены на 2 вида: чувствительные к латентности (высокое значение блокирующего фактора) и чуствительные к пропускной способности (низкое значение блокирующего фактора). Однако данное исследование ограничивается рассмотрением обращений только в DDR память при фиксированных частотах всех устройств.

Авторы работы [8] предлагает метод визуализации утилизации пропускной способности ОЗУ для выявления возможных мест оптимизации производительности: для этого они используют контроллер внутри ОЗУ для вычисления утилизации пропускной способности и латентности.

Зависимость латентности от утилизации пропускной способности представляет собой монотонную возрастающую функцию, которую можно положить константой при значениях утилизации, не сильно близкой к максимально возможной ([9]), при приближении утилизации к своему максимуму латентность начинает сильно расти.

Таким образом, можно сделать основные выводы из существующих работ:

1. Чувствительность производительности к латентности памяти зависит от рабочей нагрузки и может быть очень низкой даже при большом количестве обращений в память (ОЗУ). Такая зависимость определяется уровнем параллелизма обращений в память.

2. Латентность является монотонной возрастающей функцией от утилизации пропускной способности, которую можно приближённо положить константой почти

на всём интеравале утилизации; при стремлении утилизации к своему пределу происходит быстрый рост латентности до значений, на порядки превышающих её нормальное значение.

3.4 Симулятор архитектуры Gem5

При исследовании компьютерных архитектур, их оптимизации или оценки новых идей, связанных с параметрами таких архитектур, чаще всего используют не конечное устройство, а симуляцию архитектуры такого устройства, т.к. это и дешевле в разработке, и открывает больше возможностей в плане вариации параметров компонент архитектуры при оценке их влияния на разрабатываемое решение (оптимизация, алгоритм и т.д.).

Одним из наиболее популярных и продвинутых симуляторов компьютерных систем является Gem5 [10]. Он поддерживает различные архитектуры: ARM, ALPHA, MIPS, Power, SPARC, x86 и т.д.. Для симуляции возможно использовать несколько типов процессоров [11], среди которых "Atomic Simple", "Timing Simple", "Minor", "O3". "Atomic Simple" является функциональной симуляцией машинных инструкций, "Timing Simple" – простейшей потактовой модели исполнения машинных инструкций, "Minor" – модель процессора, не поддерживающего спекулятивное исполнение (Out-of-Order), а "O3" – поддерживающего спекулятивное исполнение.

Gem5 поддерживает 2 режима симуляции: эмуляция системных вызовов (SE – syscall emulation), который поддерживает симуляцию одного приложения с некоторыми ограничениями (например, ввиду отсутствия таблицы страниц, поддерживаемой операционной системой, обращения в виртуальную память упрощено, MMU (Memory Management Unit) не участвует в симуляции), и полная эмуляция системы (FS – full system).

В данной работе используется архитектура ARM, т.к. именно она используется в большинстве мобильных систем на сегодняшний день. В качестве процессора выбран процессор "Minor", т.к. он обеспечивает хорошую точность симуляции и на практике процессоры такого типа реально используются в мобильных устройствах в качестве энергоэффективных ядер. Процессор "O3" обладает более сложным конвейером и поддерживает внеочередное исполнение инструкций, такие ядра обычно используют как производительные (с повышенным энгергопотреблением), но симуляция такого типа процессорного ядра занимает больше времени, чем "Minor", поэтому он не выбран чисто из практических соображений.

Компанией ARM в симуляторе Gem5 было разработано процессорное ядро HPI (High Performance In-order) [12] на основе "Minor" специально для исследовательских целей. В дальнейшем будет использован именно этот тип процессорного ядра.

В качестве режима симуляции выбрана полная эмуляция системы, которая поддерживает полноценный запуск ядра Linux под архитектуру ARM. Именно в режиме полной эмуляции будет произведено дальнейшее исследование-построение модели для регулирования частот процессорных ядер.

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 исполняется за разное количество процессорных циклов, которые, в свою очередь, обратно пропорциональны времени исполнения этих инструкций.

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

$$cycles = \frac{freq_{cpu}}{\tau} \tag{2}$$

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, если отсутствуют инструкции, связанные с обращением высокие уровни памяти (ОЗУ или кеши высокого уровня), либо уменьшается, если процессор часто обращается в кеши высокого уровня 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

$$cpi = \frac{cycles}{instrs} = \frac{cycles_{cpu}}{instrs} + \frac{cycles_{mem}}{instrs} \tag{3}$$

Важно отметить, что $\frac{cycles_{cpu}}{instrs}$ – значение cpi, если бы все обращения в кеши и оперативную память занимали 0 циклов, т.е. эти 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 конвейера ядра 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:

$$cycles_{mem} = n_{L_1} \cdot \frac{lat_{L_1}}{freq_{L_1}} \cdot freq_{cpu} + n_{L_2} \cdot \frac{lat_{L_2}}{freq_{L_2}} \cdot freq_{cpu} + n_{ram} \cdot \frac{lat_{ram}}{freq_{ram}} \cdot freq_{cpu}, \tag{4}$$

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

где lat_{L_1}, lat_{L_2} и lat_{ram} – латентности, выраженные в собственных циклах (а не процессорных). n_{L_1}, n_{L_2} и n_{ram} – характерные количества обращений к соответствующим уровням памяти в случае промаха в предыдущий уровень памяти и попадания в текущий за время τ. Заметим, что эти значения не являются абсолютными значениями количества обращений, так как при обращении в память присутствует параллелизм, как было отмечано в 3.3. Значит эти значения вкладывают как параллелизм обращений, так и их количество.

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 первого уровня оперировал с частотами процессора ($freq_{L_1} = freq_{cpu}$), тогда формулу можно упростить до более простой:

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

$$= cpi_{cpu}^{L_1} + \frac{n_{L_2} \cdot \frac{lat_{L_2}}{freq_{L_2}} + n_{ram} \cdot \frac{lat_{ram}}{freq_{ram}}}{instrs} \cdot freq_{cpu}, \tag{7}$$

то есть можно занести константную часть для заданной нагрузки, независимую 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 первого уровня 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, как отмечано в 3.3. Так, латентность оперативной памяти определяется как текущим уровнем утилизации этой памяти, 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 первого уровня.

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 первого уровня – эти аспекты учитываются в формуле и не требуют дополнительных вычислений.

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

[перепроверить всё, что идёт дальше]

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 : 2020. — https://en.wikichip.org/wiki/arm_holdings/microarchitectures/cortex-a77.

[2] Documentation Linux Kernel. — Linux CPUfreq governors : 2008. — https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt.

[3] Documentation Linux Kernel. — Schedutil : 2013. — https://docs.kernel.org/scheduler/schedutil.html.

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

[5] Qualcomm Technologies Inc. — WALT vs PELT : Redux : 2017. — https://static.linaro.org/connect/sfo17/Presentations/SFO17-307%20WALT%20vs%20PELT.pdf.

[6] 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.

[7] 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.

[8] Eyerman Stijn, Heirman Wim, and Hur Ibrahim. DRAM bandwidth and latency stacks: Visualizing DRAM bottlenecks // 2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) / IEEE. — 2022. — P. 322–331.

[9] David Howard, Fallin Chris, Gorbatov Eugene, Hanebutte Ulf R, and Mutlu Onur. Memory power management via dynamic voltage/frequency scaling // Proceedings of the 8th ACM international conference on Autonomic computing. — 2011. — P. 31–40.

[10] Binkert Nathan, Beckmann Bradford, Black Gabriel, Reinhardt Steven K, Saidi Ali, Basu Arkaprava, Hestness Joel, Hower Derek R, Krishna Tushar, Sardashti Somayeh, et al. The gem5 simulator // ACM SIGARCH computer architecture news. — 2011. — Vol. 39, no. 2. — P. 1–7.

[11] Andreas Sandberg, Stephan Diestelhorst, and William Wang. — Architectural Exploration with gem5 : 2017. — https://www.gem5.org/assets/files/ASPLOS2017_gem5_tutorial.pdf.

[12] Research Ashkan Tousi | Arm. — System Modeling using gem5 : 2017. — https://old.gem5.org/wiki/images/c/cf/Summit2017_starterkit.pdf.

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

Приложение

Список используемых обозначений и терминов

1. $cycles$ – количество циклов процессорного ядра за определённый промежуток времени;
2. $instrs$ – количество исполненных инструкций процессорным ядром за определённый промежуток времени;
3. $ipc = \frac{instrs}{cycles}$, $cpi = \frac{cycles}{instrs}$;
4. OoO (Out-of-Order) – парадигма в современных высокопроизводительных CPU, основанная на спекулятивном исполнении инструкций для повышения значения ipc;
5. ROB (Re-order buffer) – циклический буфер, используемый для работы алгоритма Томасуло для поддержки OoO в CPU;
6. 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;
7. DDR SDRAM (Double Data Rate Synchronous Dynamic Random-Access Memory) – динамическая оперативная память синхронного доступа с двойной скоростью передачи данных – класс интегральных схем памяти, используемой в компьютерах (в качестве ОЗУ);
8. LPDDR SDRAM (Low-Power Double Data Rate Synchronous Dynamic Random-Access Memory) – динамическая оперативная память синхронного доступа с двойной скоростью передачи данных и с низким энергопотреблением – класс интегральных схем памяти, используемой в мобильных компьютерах (в качестве ОЗУ);
9. DDR (Double Data Rate) – общее обозначение, которое подразумевает DDR SDRAM или LPDDR SDRAM память;
10. DVFS (Dynamic Voltage-Frequency Scaling) – технология, позволяющая регулировать рабочие частоту и напряжение устройства в режиме реального времени.