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

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

Глаз Роман Сергеевич

На данный момент для большинства компьютерных систем память является узким местом, так как производительность ядер процессоров сильно опережает производительность систем памяти.

Политики регулирования тактовых частот ядер процессора в операционной системе Linux, ядро которой в настоящее время является одним из самых распространённых в мире, не учитывают, что изменение тактовой частоты не влияет на изменение времени ожидания инструкций процессора, обращающихся в высокие уровни кешей процессора или ОЗУ, а времена ожидания всего одного обращения могут достигать сотни тактов ядра процессора.

Отсутствие учёта механизма взаимодействия памяти с ядром процессора для регулирования тактовых частот особенно сильно влияет на производительность и энергопотребление приложений реального времени, функционирование которых подразумевает быстрое реагирование операционной системы на увеличение или уменьшение нагрузки на ядро процессора.

В данной работе предлагается политика регулирования тактовых частот, учитывающая механизм взаимодействия памяти с ядром процессора, а также способ её реализации в ядре операционной системы Linux.

Содержание

1 Введение

Для улучшения производительности компьютерных систем, а также для уменьшения энергопотребления этих систем, в различных операционных системах используют механизмы регулирования тактовых частот как центрального процессора (ЦП), так и прочих компонент (графический процессор, нейронный процессор, и т.д.). Такие механизмы оперируют с технологией DVFS (Dynamic Voltage-Frequency Scaling), которая позволяет изменять тактовую частоту электронной схемы (одновременно изменяя рабочее напряжение такой схемы).

В операционной системе Linux (версии 6.1), представляющей собой монолитное ядро, тактовые частоты ядер процессора, как правило, регулируются посредством использования драйвера *cpufreq*, предоставляющего API (Application Programming Interface) для этих целей, который необходимо реализовать вендору, выпускающему процессор. За тактовые частоты прочих компонент системы отвечает API для регистрации драйверов *devfreq* (например, можно добавить возможность регулирования тактовой частоты шины, соединяющей соседние уровни кешей, или тактовой частоты оперативной памяти).

На данный момент Linux использует политики регулирования тактовых частот ядер ЦП, которые не учитывают сложную схему взаимодействия ядер с прочими компонентами компьютерной системы, работающими при тактовой частоте, отличной от частоты ядер: кеши высоких уровней (кеш, системный кеш), оперативная память. В современных реалиях производительность ядер ЦП в основном ограничена производительностью компонент памяти (кеши процессора, оперативная память), взаимодействующих с этими ядрами, поэтому текущие политики регулирования тактовых частот в Linux не применимы в общем случае.

Отсутствие учёта компонент памяти при регулировании тактовых частот ядер ЦП является критичным при выполнении приложений реального времени, такие как компьютерные игры и мессенджеры, где выставление неподходящих тактовых частот приводит к заметному замедлению приложения (деградации производительности) и/или к повышенному энергопотреблению, что негативно сказывается на опыт пользователя.

Для учёта влияния внеядерных компонент на производительность ЦП необходимо построение модели производительности ядер ЦП, которое вызывает трудностей:

1. Необходимо априорное знание структуры компьютерной системы, на которой запущена операционная система, например, количество уровней кешей, их внутреннее устройство.

2. Необходимо знать подробные характеристики блоков ядра ЦП, их внутреннее устройство, для учёта их доли влияния на конечную производительность самого ядра в целом.

3. Необходимо измерить характеристики внеядерных компонент системы, например, зависимость латентность (задержки памяти) от уровня её утилизации (уровень нагрузки её пропускной способности), для учёта их доли влияния на конечную производительность ядра ЦП.

4. Такую информацию, как утилизация оперативной памяти, или какие блоки ядра процессора были использованы за фиксированный промежуток времени, получить во время работы операционной системы, как правило, невозможно, поэтому требуется обходить использование такой информации в модели.

В данной работе в результате исследований предлагается модель производительности ядер ЦП и способ её применения, а также реализация политики регулирования тактовых частот ядер процессора.

Архитектура ARM big.LITTLE, которая доминирует на рынке мобильных платформ на сегодняшний день, представляет собой концепцию, в которой одновременно используются различные типы ядер процессора (гетерогенная архитектура): ЦП разделяется на кластеры, часть которых состоит из более энергоэффективных, но менее производительных ядер, а другая часть состоит из производительных ядер, потребляющих значительно большее количество энергии. Параллельно с задачей регулирования тактовых частот процессорных ядер возникает задача выбора ядерного кластера в зависимости от требований и специфики выполняемой задачи, что так же требует создание модели производительности ядер ЦП, делая эту задачу наиболее актуальной.

Для решения обозначенных задач удобно использовать симулятор компьютерных архитектур с открытым исходным кодом, например, Gem5, реализованный на языках программирования C++ и Python, который и будет использоваться в данной работе. Gem5 эмулирует поведение компьютерной системы, подобной реальной системе, включая поведение ядер процессора, поведение оперативной памяти, и т.д. Среди большого числа поддерживаемых архитектур в данной работы будет выбрана архитектура ARM64 (к которой относится ARM big.LITTLE), так как приложения реального времени чаще всего встречаются в мобильных устройствах, где эта архитектура доминирует по количеству устройств.

Для возможности построения модели производительности ядер ЦП, а также для реализации политики регулирования тактовых частот в операционной системе Linux, запущенной режиме эмуляции с помощью Gem5, в данной работе дополнительно:

1. Реализованы *cpufreq* драйвера под платформу эмулятора Gem5.

2. Добавлена возможность измерения статистики микроархитектурных событий в Gem5 (универсальных для любой архитектуры), относящихся к кешам, для возможности использования этой статистики в Linux с помощью драйвера *perf*.

3. Реализованы части API $devfreq$ для возможности регулирования тактовых частот прочих компонент системы Gem5 (при их поддержке технологии DVFS).

Исследования и реализации, проведённые в данной работе, откроют возможность сторонним разработчикам/исследователям собирать данные микроархитектурных событий, связанных с кешами, для различных тактовых частот ядер ЦП, позволяя им проводить собственные исследования в области производительности ЦП.

2 Постановка задачи

2.1 Цели работы

Цели работы:

1. Разработать универсальную модель производительности ядра процессора с учётом механизмов взаимодействия ядра с системой памяти.

2. Реализовать политику регулирования тактовых частот ядра процессора в операционной системе Linux, используя разработанную модель производительности.

2.2 Задачи

Задачи, которые необходимо решить для достижения целей:

1. Провести исследование существующих политик регулирования тактовых частот ядер центрального процессора, выявить их узкие места. Провести обзор литературы на тему альтернативных способов регулирования тактовых частот.

2. Реализовать в операционной системе Linux:

   (a) Драйверы, позволяющие изменять тактовые частоты ЦП в эмулируемой системе архитектурным симулятором Gem5.

   (b) Возможность сбора статистики микроархитектурных событий ядер ЦП, связанных с работой кешей процессора, для архитектуры ARM64.

3. Разработать модель производительности ядер ЦП:

   (a) Спроектировать теоретическую модель производительности ядер ЦП.

   (b) Реализовать набор бенчмарков, подходящий для нахождения коэффициентов построенной модели, и собрать для них статистику микроархитектурных событий при различных тактовых частотах ядер ЦП.

   (c) Найти коэффициенты модели на основе измеренных данных.

4. Реализовать политику регулирования тактовых частот ядер ЦП в ядре операционной системы Linux, используя разработанную модель.

3 Обзор существующих решений

3.1 Регулирование тактовых частот ЦП

3.1.1 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 тактовую частоту для более экономного расходования энергии, не сильно потеряв в производительности.

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

Практически все современные процессоры используют DVFS для регулировки тактовых 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 компьютерных систем, предоставляется возможным регулировать тактовые частоты только целиком всего кластера, то есть всех ядер в кластере одновременно, а не каждого ядра по отдельности.

Регулирование тактовых 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 тактовых частот процессорных ядер, регулирование тактовых частот прочих компонент рассматриваться не будет.

3.1.2 Политики регулирования частот в ядре Linux

Единственным способом регулирования тактовых частот ядер процессора в ядре Linux является использование драйвера $cpufreq$, который предоставляет несколько политик регулирования [1]:

1. ”Performance” – тактовая частота ядра процессора выставляется в максимальное значение.

2. ”Powersave” – тактовая частота ядра процессора выставляется в минимальное значение.

3. "Userspace" – тактовая частота ядра процессора выставляется пользователем.

4. "Ondemand" – тактовая частота ядра процессора выставляется в зависимости от доли времени использования ядра за фиксированный промежуток времени (далее обозначается как утилизации ядерного времени).

5. "Conservative" – поведение аналогично политике "Ondemand", но частота меняется менее резко, небольшими шагами.

6. "Schedutil" [2] – поведение аналогично политике "Ondemand", но утилизация отслеживается не для ядер процессора, а для каждого потока исполнения; используется более продвинутая система подсчёта утилизации ядерного времени для потока исполнения: не только за фиксированный промежуток времени, а за несколько таких промежутков подряд с учётом весов каждого промежутка времени (более давние промежутки менее важны); в случае использования не CFS (Completely Fair Scheduler) планировщика, который является стандартным в Linux версии 6.1, а дедлайн планировщика (DL) или реального времени (RT), тактовая частота выставляется в максимальное значение, т.к. производительность в таком случае является критичным для работы приложений, а важность энергопотребления выходит на второй план.

Политика "Performance" приводит к излишнему энергопотреблению, "Powersave" – к потере производительности, "Ondemand" и "Conservative" – не отслеживают утилизацию ядерного времени отдельно по потокам исполнения, к тому же не учитывают, что производительность ядра ЦП (которая обратно пропорциональна утилизации ядерного времени) может быть нелинейна относительно тактовой частоты (например, если процессор проводит основное время в ожидании выполнения транзакции-чтения из памяти), поэтому могут выставлять заведомо завышенное значения тактовой частоты, что приводит к дополнительному энергопотреблению, или медленно реагировать на повышение утилизации ядерного времени, что приводит к деградации производительности.

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

Ещё один существенный недостаток всех политик, рассмотренных выше, является предположение, что частота любого ядра процессора может изменяться независимо от остальных. Как было упомянуто в 3.1.1, в современных мобильных системах ядра группируются в кластеры и частоты могут меняться только целиком для всего кластера.

3.1.3 Подсчёт утилизации ядерного времени в Linux

В ядре Linux версии 6.1 в стандартном планировщике CFS (Completely Fair Scheduler) используется решение [3], в котором при подсчёте утилизации ядерного времени прило-

жением учитывается тот факт, что ядра различного типа (в разных кластерах) имеют разную производительность, и что в рамках одного ядра ЦП 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 (приложения). Также не учтено, что зависимость производительности, измеряемой в количестве исполненных инструкций в единицу времени, обычно нелинейна относительно тактовой частоты ядра процессора (так как скорость исполнения инструкций зависит не только от тактовой частоты ядра, но и характеристик компонент памяти).

Таким образом, авторы вводят величину ёмкости производительности ядра процессора, измеряемой количеством исполненных инструкций в единицу времени при заданной тактовой частоте при условии максимальной утилизации ядерного времени. Величина утилизации вводится как часть ёмкости производительности, использованной при заданной утилизации ядерного времени (т.е. меньше утилизация ядерного времени – меньше величина утилизации).

Более формально, если за фиксированный интервал времени τ i-ое ядро процессора было задействовано время Δt, то утилизация ядерного времени равна $\Delta t/\tau$, а для i-ого ядра процессора формула утилизации имеет следующий вид:

$$util = 1024 \cdot \frac{\Delta t}{\tau} \cdot \frac{freq_{cpu_i}}{freq_{cpu_i}^{max}} \cdot \frac{capacity_{cpu_i}}{\max_i\{capacity_{cpu_i}|_{freq_{cpu_i}=freq_{cpu_i}^{max}}\}} = inv_{i,freq}, \tag{1}$$

где $freq_{cpu_i}$ – тактовая частота ядра процессора, при которой была измерена утилизация ядерного времени Δt, $freq_{cpu_i}^{max}$ – максимально возможная тактовая частота ядра процессора, $capacity_{cpu_i}$ – ёмкость производительности i-ого ядра при заданной тактовой частоте, $capacity_{cpu_i}|_{freq_{cpu_i}=freq_{cpu_i}^{max}}$ – ёмкость этого же ядра при максимально возможной тактовой частоте. Коэффициент 1024 введён для удобства.

Ёмкости производительности вычисляются единожды с помощью измерений для конкретных сценариев исполнения (приложений) и применяются во всех остальных случаях без каких-либо обоснований, причём с линейной зависимостью производительности от тактовой частоты ядра, что неверно в большинстве случаев.

В реальности используется утилизация, подсчитанная не за один промежуток времени, а сразу за несколько промежутков времени (обычно одинаковых по длительности), например, PELT (Per Entity Load Tracking) [2] использует экспоненциально движущееся среднее значение утилизаций, подсчитанных по формулам выше.

В качестве альтернативы PELT, который официально представлен в ядре Linux в версии 4.19, компанией Google был разработан WALT (Window Assisted Load Tracking) [4], который предназначен специально для мобильных устройств, где важна быстрая реакция со стороны ядра на изменение утилизации ядерного времени, чтобы избежать

деградации производительности. WALT использует утилизации существенно меньшего количества промежутков времени (5 вместо 32) в отличие от PELT 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" [2], WALT или PELT используются не только внутри планировщика, но и для выбора тактовой частоты ядра процессора, т.е. "Schedutil" переиспользует подсчитанное значение утилизации потока исполнения для регулирования частот (только в случае CFS планировщика).

Таким образом, в Linux версии 6.1 ни стандартный планировщик, ни системы регулирования частот не учитывают, что производительность ядра процессора не пропорциональна тактовой частоте в общем случае.

3.1.4 Альтернативные подходы к регулированию частот

В работе [5] предлагается использовать формулу расчёта тактовой частоты ядра процессора на основе предыдущей предсказанной частоты по линейной формуле с коэффициентами, отвечающими за степень резкости изменения самой частоты.

Те же авторы в статье [6] используют двухслойную полносвязную нейронную сеть, избегая построения каких-либо теоретических построений или предположений. На вход нейронной сети подаются характеристики рабочей нагрузки (приложения), такие как количество исполненных инструкций, количество промахов в кешах инструкций/данных, в режиме реального времени в качестве признаков описания, а на выходе ожидается значение требуемой тактовой частоты ядра процессора. В качестве функции потерь для такой сети используется значения деградации производительности после применения значения тактовой частоты на выходе сети.

Авторы работы [7] на основе анализа существующих алгоритмов регулирования тактовых частот пришли к выводу, что модели, имеющие динамические параметры, являются более точными на практике чем модели, имеющие только статические параметры, хотя одновременно являются более медленными в вычислениях (т.к. необходимо регулировать динамические параметры). Дополнительно отмечается, что для построения модели производительности ЦП использование симуляторов является оптимальным для промышленного решения с точки зрения точности, скорости, сложности, масштабируемости и цены реализации.

Патент [8] предполагает использование метрики затраченных тактов на инструкцию для определения тактовой частоты ядра процессора; авторы также предлагают группировать потоки исполнения со схожими значениями данной метрики для исполнения на одних ядрах процессора.

Метрика, использующая количество промахов в кеш последнего уровня и количество тактов-ожидания ядра процессора (например, во время обращения в ОЗУ), ис-

пользуется в исследовании [9]. Авторы используют диапазон минимально возможного и максимально возможного значений метрики, биективно отображая такие значения на диапазон доступных тактовых частот ядра процессора.

Ещё одной попыткой использовать алгоритмы машинного обучения для временных потоков данных является подход статьи [10], в которой применена архитектура LSTM: на вход сети подаётся вектор значений счётчиков микроархитектурных событий, а на выходе сети вектор предсказанных временных параметров потока исполнения: утилизация ядерного времени, количество активных потоков исполнения за фиксированный промежуток времени и количество прерываний ядра процессора, сгенерированных за тот же промежуток времени. Для регулирования тактовых частот используются только значения на выходе LSTM, в качестве функции потерь используется евклидово расстояние векторов. Авторы явно используют предположение линейности производительности ядра процессора от тактовой частоты.

Подход, объединяющий использования DVFS как для регулирования тактовых частот ядер, так и для регулирования тактовых частот систем памяти, был предложен в статье [11]. Подход отличается от остальных комплексным рассмотрением политики регулирования частот: с помощью модели энергопотребления оценивается, выгодно ли изменять частоту ОЗУ, насколько это повлияет на производительность ядер процессора. Тактовая частота ядер оценивается исходя из оценки величины затраченных тактов на инструкцию, причём используются счётчики таких микроархитектурных событий, как количество обращений в каждый из уровней кешей, а также количество тактов, затраченных на обращение в каждый из всех уровней кешей, что позволяет оценивать время задержки обращения в память. Однако, в большинстве устройств информация о количестве затраченных тактов на обращения к конкретному уровню памяти недоступна; как отмечают сами авторы, данная работа предназначена для регулирования частот для серверных систем.

3.2 Симулятор компьютерных архитектур Gem5

В исследованиях в области компьютерных архитектур, их оптимизации или оценки результатов новых идей, связанных с параметрами таких архитектур, чаще всего используют не конечное устройство, а эмуляцию 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 симулятор с открытым исходным кодом Gem5 [12]. Он поддерживает большинство современных архитектур: ARM, ALPHA, MIPS, Power, SPARC, AMD64 и т.д.. Для симуляции возможно использовать несколько типов ядер процессора [13],

среди которых ”Atomic Simple”, ”Timing Simple”, ”Minor”, ”O3”.

”Atomic Simple” является функциональной симуляцией машинных инструкций, ”Timing Simple” – простейшей потактовой моделью исполнения машинных инструкций, ”Minor” – полноценная модель конвейера ядра процессора, а ”O3” – модель конвейера ядра процессора, поддерживающая спекулятивное исполнение (Out-of-Order).

Gem5 поддерживает 2 режима симуляции: эмуляция системных вызовов (SE – syscall emulation), которая 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 ARM64, т.к. именно она используется в большинстве мобильных систем на сегодняшний день. В качестве ядра процессора выбрано ”Minor”, т.к. оно обеспечивает хорошую точность симуляции и на практике ядра такого типа реально используются в мобильных устройствах в качестве энергоэффективных. Ядро процессора ”O3” обладает более сложным конвейером и поддерживает спекулятивное исполнение инструкций, такие ядра обычно используют как производительные (с повышенным энергопотреблением), но симуляция такого типа процессорного ядра занимает больше в разы больше времени, чем ”Minor”, из-за чего предпочтение было отдано ”Minor”.

Компанией ARM Ltd. в симуляторе Gem5 было разработано процессорное ядро HPI (High Performance In-order) [14], которое является ядром ”Minor” с определённой конфигурацией параметров, 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 Устройство архитектуры системы процессор-память

4.1.1 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, состоящим из логически разделённых этапов исполнения. Каждая стадия состоит из некоторых вычислительных блоков, которые взаимодействуют с блоками из другим стадий конвейера.

В зависимости от исполняемой рабочей нагрузки на каждый стадии конвейера утилизируются различные блоки ядра процессора, поэтому скорость выполнения таковой рабочей нагрузки сильно зависит от её особенностей (типа) и особенностей исполнения таковых нагрузок на блоках процессора.

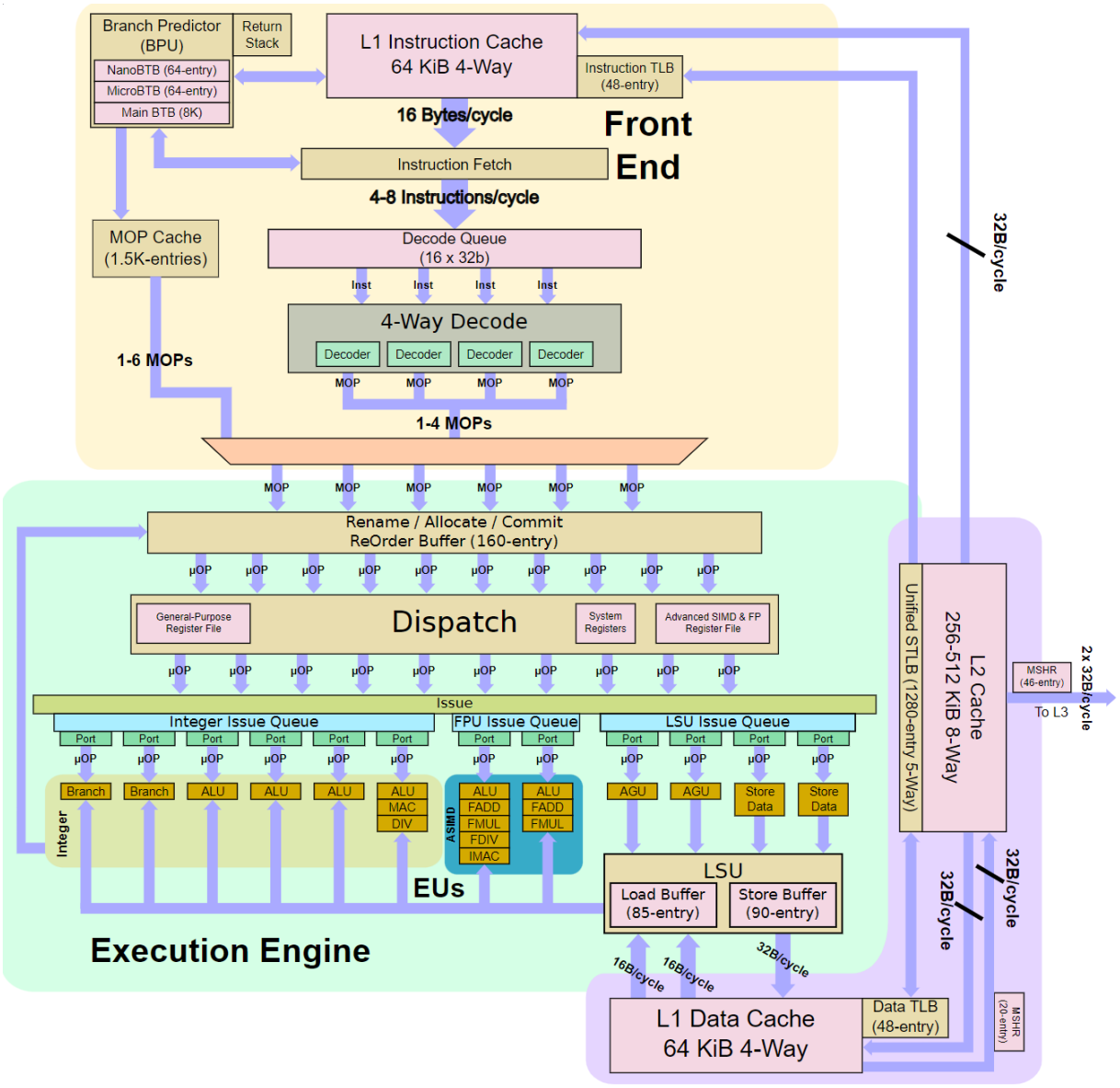
В наиболее общем виде можно разбить конвейер исполнения инструкции на процессоре на несколько последовательных стадий:

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 несколько более узконаправленных стадий, применяются дополнительные оптимизации для ускорения исполнения инструкций а также их распараллеливания (например, кеширование декодированных инструкций и предсказатель ветвлений). Наиболее актуальным примером являются ядра процессора с механизмом спекулятивного (внеочередного) выполнения инструкций (Out-of-Order), 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 инструкции, а прошедшие через стадию переименование регистров для избавления от зависимостей по данным (для дополнительного распараллеливания инструкций).

Рис. 1: Схема конвейера процессора Arm Cortex A77 [15]



В данной работе построение модели производительности проводится для ядра процессора без механизма внеочередного исполнения (In-Order) для избежания введения дополнительных динамических коэффициентов, так как в режиме реального времени получить какую-либо информацию или статистику о влияния механизма OoO на задержки обращений ядра процессора в кеши и память невозможно. Теоретически наличие OoO не сильно скажется на точности модели ввиду того, что задержки обращений ядра в кеши процессора и память на порядки превышают задержки тактов, во время которых инструкции всё ещё исполняются после обращения в память благодаря OoO.

Наибольший интерес в данной работе представляет организация обращений ядра процессора в память: на этапе записи и чтения данных в память используются буферы, накапливающие информацию о необходимых записях или чтениях, обращения в память происходят независимо от работы конвейера ядра процессора до тех пор, пока не

появляется зависимость инструкций по данным, ещё не прочтённым из памяти. В случае ядра без механизма внеочередного исполнения запись и чтение могут происходить моментально без накопления данных в дополнительным буферах.

Схема современного конвейера показана на рис. 1 на примере ядра Arm Cortex A77.

4.1.2 Организация системы 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, электронные книги и устройства схожего профиля.

Одна из основных подсистем системы на кристалле – подсистема памяти. Наиболее быстродействующими элементами памяти являются регистры ядра процессора, они же имеют наименьшее количество памяти, более высокую цену для производства и самую малую плотность расположения в электронной схеме.

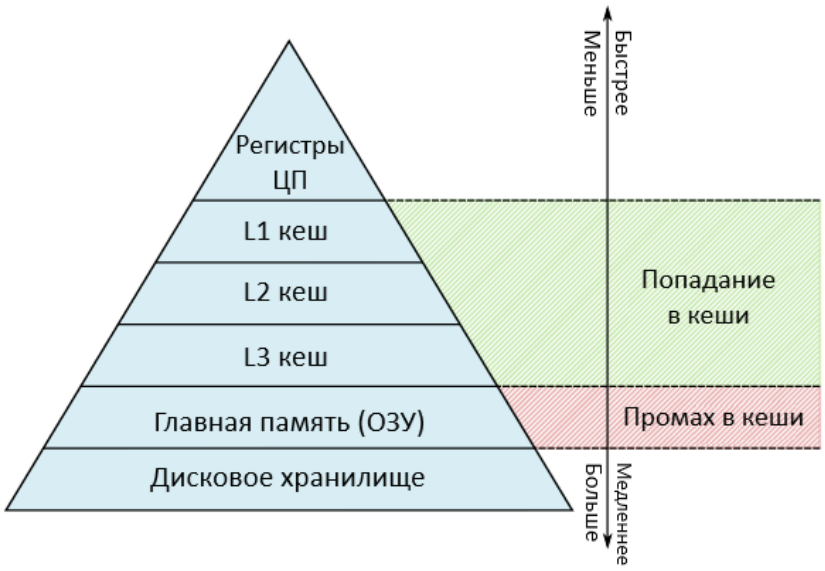
Учитывая малый объём возможного хранения данных с помощью регистров, а также что любая вычислительная система обладает локальностью (как по данным, так и по времени), почти всегда вводят дополнительные уровни памяти – более дешёвые в производстве, имеющий больший объём и более высокую плотность ячеек 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 процессора и объём их памяти напрямую зависит от требований к исполнению современных приложений и особенностей архитектуры мобильной системы. Всегда являются компромиссом: добавление дополнительного уровня кеша – снижение вероятности обращения в DDR память (наиболее медленная память) – и введение постоянной дополнительной задержки при обращении в DDR, если данные в кешах отсутствуют.

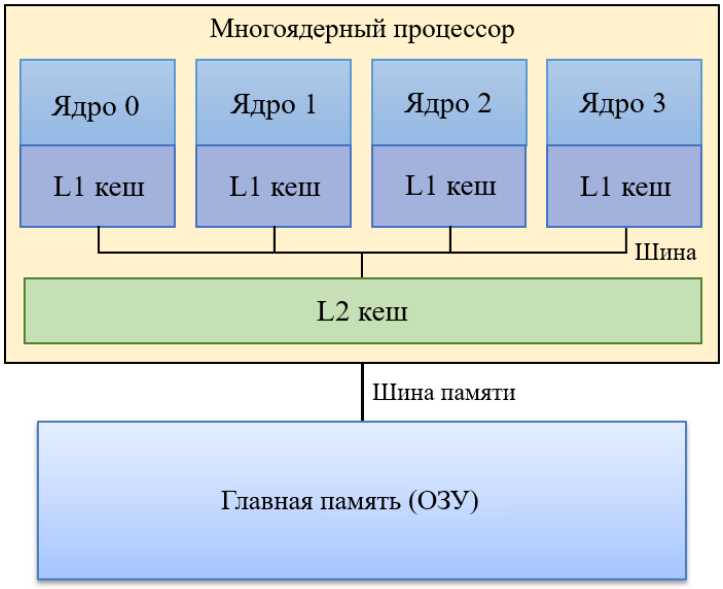
Почти все современные процессоры имеют как минимум 2 уровня кешей. В системах на кристалле, как правило, последним (дополнительным) уровнем является системный кеш, который обычно не вносят в список уровней кешей процессора, так как он

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



используется не только самим процессором, но также и различными периферийными устройствами, такими как графический ускоритель, ускоритель нейронных сетей и т.д.. Пример системы с 2-мя уровнями кешей показан на рис. 3.

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



Каждое ядро процессора имеет собственный кеш инструкций и кеш данных, на которые для краткости ссылаются как на единый кеш 1-ого уровня. Остальные уровни кешей, как правило, являются объединёнными (для инструкций и данных). 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

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 одного кластера, либо целиком всем кластером ядер (такое поведение характерно для современных 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, твердотельные накопители и др.) в данной работе рассматриваться не будут, под памятью будет иметься в виду только кеши процессора и ОЗУ.

4.1.3 Латентность памяти и утилизация её пропускной способности

Для краткости обозначения отношения величины затраченных тактов *cycles* ядром процессора на исполнение количества инструкций *instrs* введём соотношение $cpi = cycles/instrs$.

Для построения модели производительности ядра процессора, т.е. для правильного подсчёта утилизации ядра и наиболее энергоэффективного регулирования тактовых частот, 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 латентность памяти – промежуток времени между отправлением запроса на получение данных в память и между самим получением данных.

Таким образом, кеши процессора и DDR память имеют свои собственные динамические характеристики, приведённые ранее. Как правило, низкие уровни кешей работают на той же тактовой частоте, что и само процессорное ядро, поэтому значение латентности таких кешей можно выразить через такты ядра процессора. Сложнее ситуация обстоит с более высокими уровнями кешей и DDR они имеют отдельные источники тактирования, значение латентности нельзя выразить через такты процессорного ядра, к тому же она зависит от утилизации пропускной способности шин, ведущих к этим элементам памяти, то есть от количества транзакций в единицу времени.

Модель производительности ядра процессора требует учёта латентности памяти, поэтому необходимо рассмотреть существующие решения для её определения и/или возможного учёта в модели.

Авторы статьи [16] предлагают 2 модели для учёта латентности памяти в рамках алгоритма регулирования тактовых частот. Первая модель разбивает такты ядра процессора на 2 компоненты: относящиеся к ожиданию транзакций памяти и относящиеся непосредственно к вычислительным блокам ядра процессора. Утверждается, что при изменении частоты ядра процессора изменяется только количество тактов, отно-

сящихся к памяти (тактовая частота которой независима от частоты ядра). На этом и основывается алгоритм регулирования частот, использующий перерасчёт времени исполнения через такты при различных частотах. Вторая модель является улучшением первой модели: дополнительно предполагается, что в группе инструкций, которые обращаются в память подряд, следует учитывать только первую инструкцию в формуле для перерасчёта тактов, 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 на сегодняшний день.

В работе также отмечается, что следует учитывать ROB в OoO процессорах: из тактов, относящихся к транзакциям памяти, следует вычесть ту часть тактов, которую тратит процессор на спекулятивное исполнение инструкций после инструкций-обращений в память.

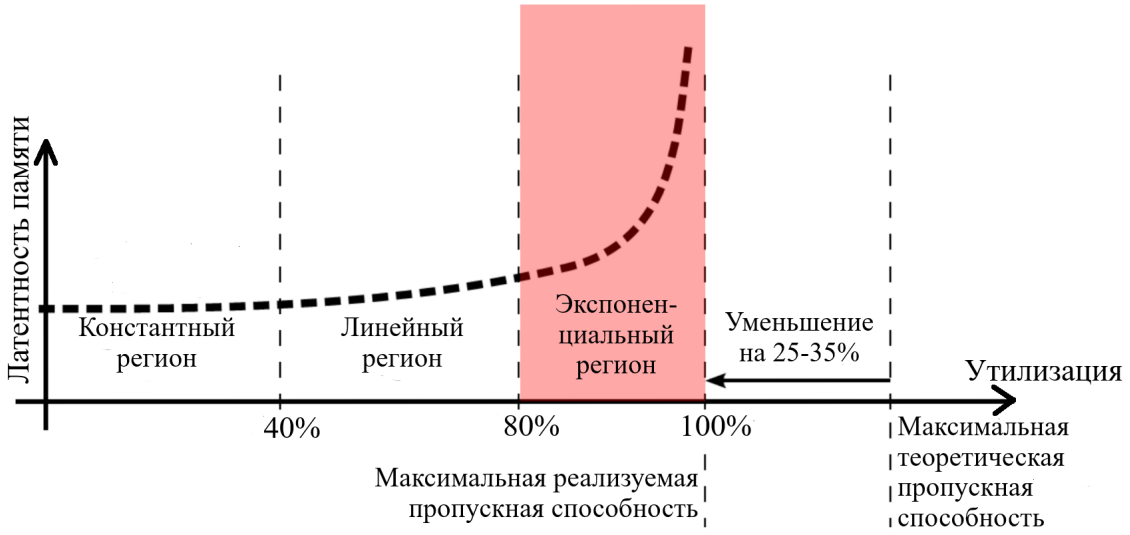
Подход использования аналитической формулы влияния утилизации пропускной способности памяти на её латентность рассматривается в статье [17]: применяется метрика *cpi*, которая включает в себя слагаемые, связанные с тактами, потраченными на время ожидания транзакций-обращений в память. Авторы вводят понятие блокирующего фактора для латентности памяти: в зависимости от рабочей нагрузки при одинаковом количестве обращений в память влияние латентности на метрику *cpi* может меняться вплоть до десятка раз из-за параллельных обращений в память, что коррелирует с результатами работы [16]. 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 чувствительные к пропускной способности (низкое значение блокирующего фактора). Однако данное исследование ограничивается рассмотрением обращений только в DDR память при фиксированных тактовых частотах всех устройств.

Зависимость латентности памяти от утилизации пропускной способности шин, ведущих непосредственно к физической памяти, представляет собой монотонную возрастающую функцию, которую можно положить константой при значениях утилизации, не сильно близкой к максимально возможной ([18]), при приближении утилизации к своему максимуму значение латентности резко растёт.

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 обраще-

Рис. 4: Пример зависимости латентности памяти от утилизации пропускной способности



ний в память, для которого будет использовано обозначение α^{par}.

2. Латентность памяти является монотонной возрастающей функцией от утилизации пропускной способности шин, которую можно приближённо положить константой почти на всём интервале утилизации; при стремлении утилизации к своему пределу происходит быстрый рост латентности до значений, в разы превышающих значения при среднем уровне утилизации пропускной способности. Пример такой зависимости изображён на рис. 4.

4.2 Модель производительности ядра процессора

Главной характеристикой производительности процессора является количество инструкций, исполняемых в единицу времени – чем больше это значение, тем быстрее исполняется рабочая нагрузка (приложение). Причём заданное количество инструкций исполняется за разное количество процессорных тактов, которые, в свою очередь, обратно пропорциональны времени исполнения этих инструкций.

Пусть за время τ ядро процессора непрерывно исполняло инструкции и исполнило $instrs$ инструкций за $cycles$ тактов при заданной тактовой 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 эффективности ядра процессора: чем выше/ниже значение ipc/cpi, тем эффективнее работа ядра процессора. Однако кроме типа процессорного ядра на эти значения также влияют тактовая частота самого ядра и тактовые

частоты остальных компонент системы. Например, при повышении частоты ядра процессора величина *ipc* либо остаётся такой же, если отсутствуют инструкции, связанные с обращением высокие уровни кешей или ОЗУ, либо уменьшается.

За время τ процессор часть тактов тратит на исполнение инструкций непосредственно на ядерных 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 тактов, т.е. обрабатывались бы бесконечно быстро, а значит эта величина ограничивается снизу возможностями ядра процессора. Обозначим данное соотношение как $cpi_{cpu} \equiv \frac{cycles_{cpu}}{instrs}$.

В свою очередь величина $cycles_{mem}$ характеризуется только лишь компонентами памяти, не зависящими от конвейера ядра процессора. Пусть, например, имеется 2 уровня кешей и ОЗУ, тогда время доступа к определённому уровню кеша характеризуется величиной $cycles_{L_i}$, представляющую собой латентность кеша (время задержки обращения), выраженную в ядерных тактах, 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 ОЗУ, если кеша высшего уровня не осталось. Время доступа к оперативной памяти обозначим $cycles_{ram}$.

Выразим времена доступа через такты ядра процессора и латентности, и введя обозначение $time_*^{-1} = \alpha_*^{par} \cdot n_* \cdot \frac{lat_*}{freq_*}$, где α_*^{par} – коэффициент параллелизма обращений в соответствующую компоненту памяти (см. 4.1.3), n_* – количество таких обращений за время τ, lat_* – средняя латентность одного обращения, выраженное не в ядерных тактах, а в собственных тактах компоненты памяти, получим:

$$cycles_{mem} = \left(time_{L_1}^{-1} + time_{L_2}^{-1} + time_{ram}^{-1}\right) \cdot freq_{cpu}, \tag{4}$$

$$cpi = cpi_{cpu} + cpi_{mem}, \tag{5}$$

$$cpi_{mem} = \left(\frac{n_{L_1}}{instrs} \cdot \frac{\alpha_{L_1}^{par} \cdot lat_{L_1}}{freq_{L_1}} + \frac{n_{L_2}}{instrs} \cdot \frac{\alpha_{L_2}^{par} \cdot lat_{L_2}}{freq_{L_2}} + \frac{n_{ram}}{instrs} \cdot \frac{\alpha_{ram}^{par} \cdot lat_{ram}}{freq_{ram}}\right) \cdot freq_{cpu}, \tag{6}$$

Заметим, что из формулы следует, что повышение тактовой частоты ядра процессора ведёт к увеличению *cpi* и уменьшению *ipc*, а повышение тактовых 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 тактовых частотах, что и само ядро. Например, кеш первого уровня (кеши инструкций и данных) всегда оперирует с тактовой частотой ядра процессора ($freq_{L_1} = freq_{cpu}$), а значит формулу можно упростить до более простой:

$$cpi = cpi_{cpu} + \alpha^{par}_{L_1} \cdot lat_{L_1} \cdot \frac{n_{L_1}}{instrs} + \left(\frac{n_{L_2}}{instrs} \cdot \frac{\alpha^{par}_{L_2} \cdot lat_{L_2}}{freq_{L_2}} + \frac{n_{ram}}{instrs} \cdot \frac{\alpha^{par}_{ram} \cdot lat_{ram}}{freq_{ram}} \right) \cdot freq_{cpu}. \tag{7}$$

Заметим, что можно ввести обозначение, $cpi^{L_1}_{cpu} = cpi_{cpu} + \alpha^{par}_{L_1} \cdot lat_{L_1} \cdot \frac{n_{L_1}}{instrs}$, полностью убрав из рассмотрения параметры обращения в кеш 1-ого уровня.

$$cpi = cpi^{L_1}_{cpu} + \left(\frac{n_{L_2}}{instrs} \cdot \frac{\alpha^{par}_{L_2} \cdot lat_{L_2}}{freq_{L_2}} + \frac{n_{ram}}{instrs} \cdot \frac{\alpha^{par}_{ram} \cdot lat_{ram}}{freq_{ram}} \right) \cdot freq_{cpu}. \tag{8}$$

Так как в данной работе не предполагается регулирование тактовых частот устройств памяти, то обозначим для удобства величины $C_* = lat_*/freq_*$; также обозначим $npi_* = \frac{n_*}{instrs}$:

$$cpi = cpi^{L_1}_{cpu} + \left(\alpha^{par}_{L_2} \cdot C_{L_2} \cdot npi_{L_2} + \alpha^{par}_{ram} \cdot C_{ram} \cdot npi_{ram} \right) \cdot freq_{cpu}. \tag{9}$$

Заметим, что величины C_{L_2} и C_{ram} не являются константами в общем случае, так как имеют смысл латентности обращения в память, которая зависит от утилизации пропускной способности шины, ведущей к соответствующей компоненте памяти (см. 4.1.3).

Латентность оперативной памяти определяется как текущим уровнем утилизации этой памяти (шины, ведущей к 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 ячейкам DDR памяти очень важна).

Коэффициенты $\alpha^{par}_{L_2}$ и α^{par}_{ram}, как правило, коррелируют, так как определены спецификой обращения рабочей нагрузки в память, поэтому можно положить $\alpha^{par}_{L_2} = \alpha^{par}_{ram} \equiv \alpha^{par}$. Данное приближение обосновывается тем фактом, что даже если значения коэффициентов не коррелируют для какого-то приложения, то влияние слагаемого, ответственного за обращение в кеш 2-ого уровня, мало по сравнению с влиянием остальных слагаемых на cpi, а корректировка $\alpha^{par}_{L_2}$ описанным приближением слабо скажется на cpi, т.к. произойдёт компенсация ошибки динамическим регулированием параметра α^{par}.

Финальная формула:

$$cpi = cpi^{L_1}_{cpu} + \alpha^{par} \cdot (C_{L_2} \cdot npi_{L_2} + C_{ram} \cdot npi_{ram}) \cdot freq_{cpu}. \tag{10}$$

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

1. Явно введён коэффициент параллелизма обращений в память в α^{par}, который можно регулировать динамически во время регулирования тактовой частоты ядра процессора (см. главу 4.4.3).

2. Такие величины, как npi_{L_2} и npi_{ram} можно получить в режиме реального времени практически во всех современных компьютерных системах, в том числе в системах с архитектурой ARM64, которая используется в данной работе.

3. Величины C_{L_2} и C_{ram} можно положить константами и вычислить заранее, так как в случае увеличения латентности памяти при приближении пропускной способности шин к максимальным значениям ответственность за увеличение этих коэффициентов неявно можно переложить на параметр α^{par}, который уже предполагает динамическое регулирование.

4. Величина $cpi_{cpu}^{L_1}$ не предполагает явного вычисления через другие формулы; достаточно знать все остальные коэффициенты, тогда эту величину можно найти по формуле (10), далее использовать её для оценки cpi на других тактовых частотах.

5. 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 в коэффициенте $cpi_{cpu}^{L_1}$ и не требуют дополнительных вычислений.

6. Коэффициент α^{par} при динамическом регулировании будет также полезен при создании модели энергопотребления оперативной памяти: мощность энергопотребления пропорциональна количеству обращений в память в единицу времени, а коэффициент пропорциональности зависит от величины α^{par}.

Таким образом, для применения модели необходимо иметь:

1. Значения C_{L_2} и C_{ram}, вычисленные заранее.

2. Алгоритм регулирования параметра α^{par}.

3. Возможность измерения величин cpi, npi_{L_2} и npi_{ram} в режиме реального времени.

4.3 Модель производительности ядра процессора применительно к архитектуре ARM

Рассмотрим способ применения описанной выше модели в случае архитектуры ARM64. Требуется уметь измерять величины cpi, npi_{L_2} и npi_{ram}. Величина npi_{L_2} может быть найдена через количество промахов в кеш 1-ого уровня, а величина npi_{ram} – через количество промахов в кеш 2-ого уровня (последний уровень).

Следует воспользоваться следующими счётчиками микроархитектурных событий ([19]):

1. CPU_CYCLES – количество тактов ядра процессора;
2. $INST\_RETIRED$ – количество исполненных инструкций;
3. $L1I\_CACHE\_REFILL$ – количество перезаполнений кеш-линий с инструкциями в кеше инструкций 1-ого уровня из-за промахов в этот кеш при чтении инструкции; некешируемые промахи в кеш и операции синхронизации кешей не считаются;
4. $L1D\_CACHE\_REFILL$ – количество перезаполнений кеш-линий с данными в кеше данных 1-ого уровня из-за промахов в этот кеш; некешируемые промахи в кеш и операции синхронизации кешей не считаются;
5. $L2D\_CACHE\_REFILL$ – работает аналогично сумме счётчиков событий $L1D\_CACHE\_REFILL$ и $L1I\_CACHE\_REFILL$, но применительно ко 2-ому уровню кеша.

Заметим, что счётчик микроархитектурных событий $L2I\_CACHE\_REFILL$, измеряющий количество перезаполнений кеш-линий в кеше 2-ого уровня, как правило, не реализуется в ядрах ARM64, т.к. кеши 2-ого и выше уровней всегда объединённые для инструкций и данных.

В счётчиках типа $*\_CACHE\_REFILL$ существует очень важный нюанс, не оговорённый в спецификациях счётчиков микроархитектурных событий компании ARM Ltd., но упоминаемый в иных документах (см. [20]): в платформах архитектуры ARM не существует счётчиков, которые считают промахи в кеши, вместо них используются счётчики, считающие количество перезаполнений кеш-линий (кешами более высокого уровня или оперативной памятью). Возможна ситуация, когда один промах в кеш вызывает несколько перезаполнений кеш-линий (например, если промах случился по адресу, по которому значение пересекает 2 соседние кеш-линии), или наоборот: несколько промахов в кеш могут быть разрешены благодаря одному перезаполнению кеш-линии.

Таким образом, величины cpi, npi_{L_2} и npi_{ram} можно найти через счётчики микроархитектурных событий следующим образом:

$$cpi = \frac{CPU\_CYCLES}{INST\_RETIRED}, \tag{11}$$

$$npi_{ram} = \frac{L2D\_CACHE\_REFILL}{INST\_RETIRED}. \tag{12}$$

$$npi_{L_2} = \frac{L1I\_CACHE\_REFILL + L1D\_CACHE\_REFILL}{INST\_RETIRED} - npi_{ram}, \tag{13}$$

Заметим, что чаще всего в современных мобильных системах на архитектуре ARM64 после последнего уровня процессорного кеша располагают дополнительный кеш – системный кеш, который предназначен для периферийных устройств и находится между оперативной памятью и процессором. Он может быть как больше по размеру, чем последний уровень процессорного кеша, так и меньше. Наличие системного кеша ведёт к тому, что для определения значения n_{ram} недостаточно воспользоваться счётчиками, описанными выше; единственным счётчиком микроархитектурных событий, который может подойти в данном случае, является $LL\_CACHE\_MISS\_RD$ – данный счётчик считает количество перезаполнений кеш-линий в кеш последнего уровня, в которых данные взяты из оперативной памяти, либо из другого чипа, либо из соседнего процессорного кластера с использованием технологии Intercluster peering.

Согласно документации ARM64 ([20]), если счётчик $LL\_CACHE\_MISS\_RD$ не реализован, то есть бит $EXTLLC$ в регистре $CPUECTLR$ выставлен в ноль, то значения счётчика $LL\_CACHE\_MISS\_RD$ будут совпадать со значениями счётчика $L2D\_CACHE\_REFILL$ (в случае процессора с 2-мя уровнями кешей). В случае наличия системного кеша игнорирование этого счётчика может заметно ухудшит работу модели, однако в данной работе наличие системного кеша не предполагается, так что данный счётчик будет проигнорирован.

4.4 Способ применения модели производительности ядра процессора в ядре Linux

В этой главе будут использоваться термины утилизации, введённые в главе 3.1.3.

Для применения описанной ранее модели производительности, необходимо разработать формулы для вычислений следующих значений:

1. Утилизация потока исполнения на основе утилизации ядерного времени этим потоком за фиксированный промежуток времени.

2. Тактовая частота, которую необходимо выставить, на основе значения подсчитанной утилизации (или утилизаций за несколько промежутков времени).

3. Коэффициент параллелизма обращений в память α^{par} для следующего промежутка времени.

4.4.1 Вычисление утилизации потока исполнения

По определению утилизация потока исполнения – часть ёмкости производительности, использованной при заданной утилизации ядерного времени, а ёмкость производительности – количество исполненных инструкций в единицу времени при заданной тактовой частоте при условии максимальной утилизации ядерного времени (см. 3.1.3).

Пусть τ – величина промежутка времени, за которую будет подсчитана утилизация потока исполнения. Рассмотрим для упрощения только 1 ядро процессора, тактовая частота которого фиксированна. Тогда, согласно определениям, приведённым выше, справедливо

$$util = 1024 \cdot \frac{\Delta t}{\tau} \cdot \frac{instrs/s}{(instrs/s)|_{freq_{cpu}=freq_{cpu}^{max}}} = 1024 \cdot \frac{\Delta t}{\tau} \cdot \frac{instrs}{(instrs)|_{freq_{cpu}=freq_{cpu}^{max}}} \tag{14}$$

где Δt – время активной работы ядра процессора, $\frac{\Delta t}{\tau}$ – утилизация ядерного времени, $instrs/s$ – количество выполняемых инструкций в единицу времени при заданной тактовой частоте, а $(instrs/s)|_{freq_{cpu}=freq_{cpu}^{max}}$ – максимально возможное для рассматриваемой рабочей нагрузки значение выполняемых инструкций в единицу времени, достигаемое повышением тактовой частоты ядра до своего максимального значения.

Заметим, что

$$instrs/s = \frac{instrs}{cycles} \cdot \frac{cycles}{s} = \frac{1}{cpi} \cdot freq_{cpu} = \frac{freq_{cpu}}{cpi}, \tag{15}$$

тогда формула утилизации принимает следующий вид:

$$util = 1024 \cdot \frac{\Delta t}{\tau} \cdot \frac{\frac{freq_{cpu}}{cpi}}{\left(\frac{freq_{cpu}}{cpi}\right)\Big|_{freq_{cpu}=freq_{cpu}^{max}}} = 1024 \cdot \frac{\Delta t}{\tau} \cdot \frac{freq_{cpu}}{freq_{cpu}^{max}} \cdot \frac{cpi|_{freq_{cpu}=freq_{cpu}^{max}}}{cpi}. \tag{16}$$

Возникший множитель $\frac{cpi|_{freq_{cpu}=freq_{cpu}^{max}}}{cpi}$ явно предполагался равным единице в решении, описанном в главе 3.1.3, в формуле (1).

Подставив формулу модели производительности (10) в формулу (16), а также обозначив $lat_{gen} = \alpha^{par} \cdot (C_{L_2} \cdot npi_{L_2} + C_{ram} \cdot npi_{ram})$, получим:

$$util = 1024 \cdot \frac{\Delta t}{\tau} \cdot \frac{freq_{cpu}}{freq_{cpu}^{max}} \cdot \frac{cpi_{cpu}^{L_1} + lat_{gen} \cdot freq_{cpu}^{max}}{cpi_{cpu}^{L_1} + lat_{gen} \cdot freq_{cpu}}. \tag{17}$$

В формуле (17) предполагается использовать значения $cpi_{cpu}^{L_1}$ и lat_{gen}, посчитанные в конце предыдущего промежутка времени для расчёта утилизации за текущий промежуток времени.

Если $cpi_{cpu}^{L_1} \ll lat_{gen} \cdot freq_{cpu}$, то

$$util \approx 1024 \cdot \frac{\Delta t}{\tau} \cdot \frac{freq_{cpu}}{freq_{cpu}^{max}} \cdot \frac{lat_{gen} \cdot freq_{cpu}^{max}}{lat_{gen} \cdot freq_{cpu}} = 1024 \cdot \frac{\Delta t}{\tau}, \tag{18}$$

то есть утилизация перестаёт зависеть от частоты ядра процессора, что и следовало ожидать.

Если $cpi_{cpu}^{L_1} \gg lat_{gen} \cdot freq_{cpu}$, то есть вклад тактов, затраченных на ожидание обращений в память, мал по сравнению с остальным числом тактов, то

$$util \approx 1024 \cdot \frac{\Delta t}{\tau} \cdot \frac{freq_{cpu}}{freq_{cpu}^{max}} \cdot \frac{cpi_{cpu}^{L_1}}{cpi_{cpu}^{L_1}} = 1024 \cdot \frac{\Delta t}{\tau} \cdot \frac{freq_{cpu}}{freq_{cpu}^{max}}, \tag{19}$$

то есть формула совпадает с формулой (1), использованной в ядре Linux.

Рассмотрим теперь более общий случай, когда тактовая частота в одном промежутке времени τ не фиксирована, как предполагалось до этого (см. формулу (17)), а может изменяться (например, из-за того, что в ядерном кластере соседнее ядро решило повысить тактовую частоту). Пусть i – номер подынтервала, на котором процессор был активен и его тактовая частота равнялась $freq_{cpu,i}$.

$$util = 1024 \cdot \frac{\Delta t}{\tau} \cdot \frac{\langle instrs/s \rangle_t}{\langle (instrs/s)|_{freq_{cpu,i}=freq_{cpu}^{max}} \rangle_t}, \ \Delta t = \sum_i \Delta t_i. \tag{20}$$

$$util = 1024 \cdot \frac{\sum\limits_i \Delta t_i}{\tau} \cdot \frac{\frac{1}{\tau}\sum\limits_i (instrs/s)_i \cdot \Delta t_i}{\frac{1}{\tau}\sum\limits_i (instrs/s)_i \cdot \Delta t_i|_{freq_{cpu,i}=freq_{cpu}^{max}}} = \tag{21}$$

$$= 1024 \cdot \frac{\Delta t}{\tau} \cdot \frac{\sum\limits_i \frac{instrs_i}{\Delta t_i} \cdot \Delta t_i}{\sum\limits_i (\frac{instrs_i}{\Delta t_i} \cdot \Delta t_i)|_{freq_{cpu,i}=freq_{cpu}^{max}}} = 1024 \cdot \frac{\Delta t}{\tau} \cdot \frac{\sum\limits_i instrs_i}{\sum\limits_i instrs_i|_{freq_{cpu,i}=freq_{cpu}^{max}}}.$$

$$\frac{instrs_i|_{freq_{cpu,i}=freq_{cpu}^{max}}}{instrs_i} = \frac{(\frac{freq_{cpu}}{cpi})|_{freq_{cpu}=freq_{cpu}^{max}}}{\frac{freq_{cpu,i}}{cpi_i}} = \frac{freq_{cpu}^{max}}{freq_{cpu,i}} \cdot \frac{cpi_i}{cpi|_{freq_{cpu,i}=freq_{cpu}^{max}}} = \tag{22}$$

$$= \frac{freq_{cpu}^{max}}{freq_{cpu,i}} \cdot \frac{cpi_{cpu}^{L_1} + lat_{gen} \cdot freq_{cpu}^{max}}{cpi_{cpu}^{L_1} + lat_{gen} \cdot freq_{cpu,i}},$$

откуда можно получить финальную для утилизации:

$$util = 1024 \cdot \frac{\Delta t}{\tau} \cdot \frac{\sum\limits_i instrs_i}{\sum\limits_i instrs_i \cdot \frac{freq_{cpu}^{max}}{freq_{cpu,i}} \cdot \frac{cpi_{cpu}^{L_1} + lat_{gen} \cdot freq_{cpu}^{max}}{cpi_{cpu}^{L_1} + lat_{gen} \cdot freq_{cpu,i}}}. \tag{23}$$

Важно отметить, что хотя суммирование проводится в течение одного промежутка времени τ, значение $cpi_{cpu}^{L_1}$ всё ещё считается в конце предыдущего промежутка времени в предположении, что оно останется таким же в текущем промежутке, так же как и величина lat_{gen} (после регулирования параметра α^{par}, подробнее в главе 4.4.3).

Однако, теперь расчёт $cpi_{cpu}^{L_1}$ в конце каждого промежутка не такой тривиальный: в каждом подынтервале i текущего промежутка времени необходимо сделать следующее:

$$cpi_i = cpi_{cpu,i}^{L_1} + lat_{gen,i} \cdot freq_{cpu,i} \Rightarrow cycles_i = cycles_{cpu,i}^{L_1} + lat_{gen,i} \cdot freq_{cpu,i} \cdot instrs_i \Rightarrow \quad (24)$$

$$\Rightarrow cycles_{cpu,i}^{L_1} = cycles_i - lat_{gen,i} \cdot freq_{cpu,i} \cdot instrs_i \Rightarrow cpi_{cpu}^{L_1} := \frac{\sum\limits_i cycles_{cpu,i}^{L_1}}{\sum\limits_i instrs_i}, \text{ т.е.}$$

$$cpi_{cpu}^{L_1} := \frac{\sum\limits_i cycles_i - \alpha^{par} \cdot (C_{L_2} \cdot n_{L_2,i} + C_{ram} \cdot n_{ram,i}) \cdot freq_{cpu,i}}{\sum\limits_i instrs_i}, \quad (25)$$

именно такое значение будет посчитано в конце текущего промежутка времени и использованного в следующем промежутке по формуле (23).

Вдобавок отметим, что

$$lat_{gen} := \alpha^{par} \cdot \frac{\sum\limits_i C_{L_2} \cdot n_{L_2,i} + C_{ram} \cdot n_{ram,i}}{\sum\limits_i instrs_i}. \quad (26)$$

4.4.2 Выбор тактовой частоты ядра процессора

Выбор тактовой частоты основывается на посчитанной утилизации по формуле (23). Предположим, что за промежуток времени τ ядро процессора было активным время $\Delta t < \tau$. Смысл регулирования тактовой частоты заключается в том, что если Δt не сильно близко к значению τ, то тактовую частоту можно понизить так, чтобы Δt всё ещё оставалось меньшим величины τ, т.е. чтобы приложение всё ещё успевало выполнять требуемую работу.

Если $\Delta t > \tau$ на текущей тактовой частоте, то, как минимум, тактовую частоту требуется повышать, но насколько повышать – аналитически выявить невозможно, поэтому предлагается следующий алгоритм: если такая ситуация произошла только 1 раз, тактовая частота повышается на n уровней (возможные значения тактовых частот всегда дискретны), если после этого она произошла ещё раз, то тактовая частота повышается до максимально возможного значения.

Чтобы ситуация $\Delta t > \tau$ происходила более редко, предлагается так же, как и в решениях, описанных в главе 3.1.3, выставлять тактовую частоту с запасом по утилизации, то есть условие выбора тактовой частоты $\Delta t < \tau$ заменяется на $\Delta t < \gamma \cdot \tau$, $\gamma < 1$. Часто выбирают $\gamma = 0.8$, но выбор такого значения лучше осуществлять под определённую специфику запускаемых рабочих нагрузок (приложений).

Величина Δt в условии $\Delta t < \gamma \cdot \tau$ выражается через параметры модели производительности, посчитанной утилизации и тактовой частоты ядра процессора, откуда

и извлекается минимальное значение тактовой частоты для соблюдения данного условия (неравенства). Тактовая частота находится из предположения, что в следующий промежуток времени τ утилизация $util$ приложения останется такой же (как посчитанное значение), а тактовая частота выбирается полностью на весь промежуток времени, поэтому необходимо использовать формулу утилизации (17):

$$util = 1024 \cdot \frac{\Delta t}{\tau} \cdot \frac{freq_{cpu}}{freq_{cpu}^{max}} \cdot \frac{cpi_{cpu}^{L_1} + lat_{gen} \cdot freq_{cpu}^{max}}{cpi_{cpu}^{L_1} + lat_{gen} \cdot freq_{cpu}}, \ \frac{\Delta t}{\tau} < \gamma \Rightarrow \tag{27}$$

$$\Rightarrow \frac{util}{1024} \cdot \frac{freq_{cpu}^{max}}{freq_{cpu}} \cdot \frac{cpi_{cpu}^{L_1} + lat_{gen} \cdot freq_{cpu}}{cpi_{cpu}^{L_1} + lat_{gen} \cdot freq_{cpu}^{max}} < \gamma \ (\gamma < 1) \Rightarrow \tag{28}$$

$$\Rightarrow freq_{cpu} > \frac{\tilde{util} \cdot freq_{cpu}^{max} \cdot cpi_{cpu}^{L_1}}{cpi_{cpu}^{L_1} + lat_{gen} \cdot freq_{cpu}^{max} \cdot (1 - \tilde{util})}, \text{ где } \tilde{util} = \frac{util}{1024 \cdot \gamma}. \tag{29}$$

Если $\tilde{util} > 1$, т.е. $freq_{cpu} > \tilde{util} \cdot freq_{cpu}^{max} > freq_{cpu}^{max}$, а если $util \to 0$, то $freq_{cpu} > 0$, то есть граничные случаи удовлетворяют определению утилизации.

Таким образом, тактовая частота ядра процессора на следующий промежуток времени выбирается исходя из формулы (29). Значение α^{par} берётся после этапа регулирования (подробнее в главе 4.4.3) в конце текущего промежутка времени, а величины $cpi_{cpu}^{L_1}$ и lat_{gen} считаются по формулам (25) и (26) прямо перед расчётом тактовой частоты после этапа регулирования α^{par}.

4.4.3 Алгоритм регулирования коэффициента параллелизма обращений в память

Утилизация рассчитывается суммированием слагаемых во множителях по формуле (23) в течение всего промежутка времени на основе параметров, посчитанных в конце предыдущего промежутка времени; тактовая частота вычисляется из посчитанного значения утилизации и параметров, посчитанных уже в конце текущего промежутка времени. Одним единственным параметром, регулирование которого должна осуществляться внешним способом, является α^{par}.

В начале инициализации модели параметр $\alpha^{par} := 1$ полагается равным единице. Далее в конце каждого промежутка времени осуществляется его регулирование.

Чтобы избежать колебательных процессов и расхождений параметра α^{par}, предлагается ввести агрегацию таких параметров за несколько промежутков времени подряд – вычисляться будет новое значение α^{par}, но применяться в формулах агрегированное за несколько последних промежутков времени после добавления нового значения. Одной из наиболее эффективных и простых эвристик является использование экспоненциально взвешенного скользящего среднего значения за последние n промежутков; предлагается выбрать $n = 5$, аналогично значению, выбранному в алгоритме подсчёта утилизации WALT (см. главу 3.1.3).

Само значение коэффициента α^{par}, которое будет использовано в экспоненциально взвешенном скользящем среднем значении, предлагается искать через сравнение величин $cpi_{cpu}^{L_1}$, подсчитанных за предыдущий и за текущие промежутки времени. Во-первых, это компонента является зависимой от параметра α^{par} в формуле (25). Во-вторых, она не требует знаний и тактовых частот, которые были выставлены на ядре процессора.

В любом случае параметр α^{par} возможно регулировать лишь в случае изменений тактовых частот для получения обратной связи, поэтому требуется некоторый период релаксации для установления подходящего значения α^{par} для рабочей нагрузки (приложения).

Идея заключается в следующем наблюдении: если значение $cpi_{cpu}^{L_1}$ выросло по сравнению со значением в предыдущий промежуток времени, то, скорее всего (т.к. нет гарантии установления источника изменения этой величины), исходя из формулы (25), параметр α^{par} оказался меньше своего реального значения и требует корректировки в сторону повышения. Поэтому новое значение коэффициента параллелизма для j-ого промежутка времени можно положить равным

$$\alpha_j^{par} := \alpha_{j-1}^{par} \cdot \frac{cpi_{cpu,j}^{L_1}}{cpi_{cpu,j-1}^{L_1}}, \tag{30}$$

где $cpi_{cpu,j}^{L_1}$ – значение $cpi_{cpu}^{L_1}$ за j-ый промежуток.

Заметим, что при таком подходе в случае некорректного значения α^{par} ошибка компенсируется использованием экспоненциально взвешенного скользящего среднего значения. Для j-ого промежутка в результате будет использован следующий коэффициент параллелизма:

$$\alpha_{avg}^{par} = \frac{1}{n} \sum_{i=0}^{n-1} \beta^i \alpha_{j-i}^{par},\ 0 < \beta < 1. \tag{31}$$

5 Описание практической части

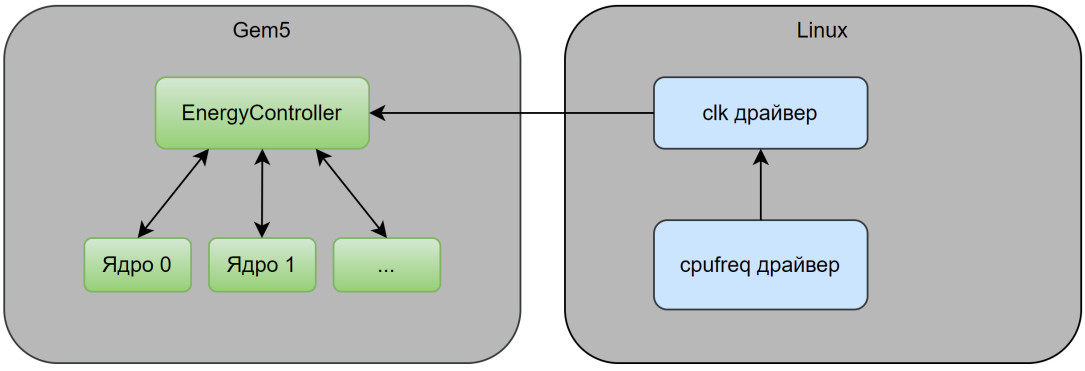
5.1 Драйверы Linux для управления тактовой частотой ядра процессора Gem5

Для построения модели производительности и реализации политики регулирования частот центрального процессора необходимо реализовать драйвер управления тактовыми частотами в ядре Linux. Интерфейсы, позволяющие регулировать тактовые частоты как в пространстве пользователя, так и пространстве ядра в Linux версии 6.1 уже реализованы в драйвере *cpufreq*, остаётся реализовать низкоуровневую часть, связанную непосредственно с устройством взаимодействия Linux и Gem5.

В Gem5 уже присутствуют механизмы, позволяющие изменять тактовые частоты многих устройств. Любой класс, являющийся наследником класса *ClockedObject*, способен регулировать тактовые частоты своих объектов, если их реализация зависит от этих частот. Например, класс *BaseCPU*, описывающий абстрактное ядро процессора, является наследником *ClockedObject* использует параметры тактовой частоты для управления производительностей работы конвейера и внутренних вычислительных блоков.

В Gem5 существует контроллер, регулирующий тактовые частоты всех устройств во время симуляции – *EnergyController*. Несмотря на то, что, как правило, параметры внутреннего устройства компьютерной системы, на которой запускается ядро Linux, передаются в ядро через *device tree* файлы ([21]), в Gem5 реализованы интерфейсы в виде дополнительных регистров, значения в которых изменяются в зависимости от способов обращения к ним: даже при чтении регистра его состояние может измениться. *EnergyController* осуществляет контроль доступа к таким регистрам, посылая уведомление о необходимости изменения тактовой частоты нужному устройству (или его контроллеру) в системе в зависимости от команд и регистров, к которым произошло обращение.

Рис. 5: Схема взаимодействия Gem5 и драйверов Linux



Дополнительные регистры для внешнего взаимодействия с *EnergyController* из

ядра Linux с помощью *device tree* переведены с соответствующий диапазон физических адресов, которым во время запуска ядра Linux ставятся в соответствие виртуальные адреса через таблицу виртуальных страниц.

В ядре Linux существует дополнительный интерфейс-драйвер *clk* для описания устройств (как настоящих, так и виртуальных), в котором необходимо реализовать низкоуровневое взаимодействие с регистрами, контролирующими тактовые частоты, по физическим адресам. Чтобы $cpufreq$ мог контролировать тактовые частоты своими интерфейсами, необходимо написать собственную реализацию ряда функций, которые будут использовать реализацию *clk* драйвера для управления тактовыми частоты.

Таким образом, был реализован интерфейс драйвера *clk* для взаимодействия с *EnergyController* в симуляторе Gem5 и интерфейс драйвера $cpufreq$, использующий реализацию *clk* для возможности регулировать тактовые частоты. Схема взаимодействия компонент представлена на рис. 5.

5.2 Реализация счётчиков микроархитектурных событий, связанных с кешами

Из главы 4.3 следует, что необходимо использовать счётчики микроархитектурных событий CPU_CYCLES, $INST\_RETIRED$ и события, связанные с кешами $L1I\_CACHE\_REFILL$, $L1D\_CACHE\_REFILL$ и $L2D\_CACHE\_REFILL$, которые не реализованы в Gem5, как и все остальные счётчики, связанные с кешами.

Так как кеши в Gem5 реализованы как абстрактные устройства посредством классов в C++, кеши, разделяемые между несколькими ядрами, должны уметь распознавать источник возникновения транзакции, проводимой с кешом. Например, в процессоре с 2-мя уровнями кешей при перезаполнении кеш-линии со стороны ОЗУ из-за промаха в кеш счётчик микроархитектурных событий $L2D\_CACHE\_REFILL$ необходимо увеличить именно для того ядра, от которого изначально произошёл промах в кеш.

Политика регулирования тактовых частот в Linux будет реализована для процессора с 3-мя, а не 2-мя уровнями кешей (2 уровня были рассмотрены лишь для примера), поэтому вместо счётчиков $L1I\_CACHE\_REFILL$ и $L1D\_CACHE\_REFILL$ необходимо брать в рассмотрение $L3D\_CACHE\_REFILL$, т.к. 2-ой уровень кеша оперирует с тактовой частотой процессора, поэтому его можно не рассматривать (аналогично тому, как были убраны из рассмотрения кеши 1-ого уровня в главе 4.2).

В Gem5 транзакции с устройствами и шинами, их соединяющими, происходят с помощью пакетов, внутри которых содержится информация об исходном отправителе на протяжении всего пути пакета. Таким образом, требуется перехватить пакет в месте, где происходит само микроархитектурное событие, и, если инициатором транзакции (отправитель пакета) является ядро процессора, увеличить счётчик для этого ядра в

соответствии с типом пакета (дял некоторых типов пакета и вовсе ничего не требуется увеличивать).

В Gem5 для архитектуре ARM64 уже реализованы все архитектурные регистры (счётчики), предназначенные для работы с микроархитектурными событиями, а в Linux драйвер $perf$ уже реализует функционал, позволяющий использовать такие счётчики как из пространства ядра, так и из пространства пользователя.

Таким образом, были реализованы механизм распознавания инициатора транзакции, проводимой с кешем, и увеличения счётчиков микроархитектурных событий в зависимости от типа транзакции для ядра-инициатора.

5.3 Вычисление коэффициентов для модели производительности ядра процессора

Чтобы найти численные значения C_{L_2} и C_{ram} (см. главу 4.2) в формуле

$$cpi = cpi_{cpu}^{L_1} + \alpha^{par} \cdot (C_{L_2} \cdot npi_{L_2} + C_{ram} \cdot npi_{ram}) \cdot freq_{cpu},$$

необходимо измерить значения cpi, npi_{L_2}, npi_{ram} для различных тактовых частот $freq_{cpu}$ для определённого набора рабочих нагрузок (приложений). Так как значение $cpi_{cpu}^{L_1}$ находится динамически по формуле (25), то его мы будем игнорировать. Так как по-умолчанию $\alpha^{par} = 1$, это же значение будет взято для вычисления остальных коэффициентов.

Как уже было упомянуто в главе 5.2, в финальной реализации политики регулирования тактовых частот будет система 3-мя уровнями кешей и следующей формулой:

$$cpi = cpi_{cpu}^{L_2} + \alpha^{par} \cdot (C_{L_3} \cdot npi_{L_3} + C_{ram} \cdot npi_{ram}) \cdot freq_{cpu}, \tag{32}$$

поэтому находится будут коэффициенты C_{L_3} и C_{ram}. Все формулы, рассмотренные в предыдущих главах, остаются применимы, за исключением замены $cpi_{cpu}^{L_1}$ на $cpi_{cpu}^{L_2}$, C_{L_2} на C_{L_3} и npi_{L_2} на npi_{L_3}.

В качестве ядра процессора будет использоваться ядро HPI (High Performance In-order) [14] без спекулятивного выполнения инструкций. Размеры кешей инструкций и данных 1-ого уровня – 32 Кб, 2-ого уровня – 512 Кб, 3-его уровня – 4 Мб.

Набор приложений должен быть таким, чтобы часть приложений полностью попадало в кеши 1-ого и 2-ого уровней, часть совершало промах во 2-ой уровень, но попадание в 3-ий, и остальная часть совершала промах в 3-ий уровень кеша, из-за чего происходит обращения в DDR (ОЗУ). Для таких целей возможно использование параметризованного бенчмарка, который рандомизированно обращается в диапазон некоторого массива, а диапазон итеративно меняется так, чтобы через какое-то количество итераций все диапазоны были пройдены.

Предлагается ко вниманию следующий параметризованный бенчмарк, генерирующий набор бенчмарков

```
unsigned long seed = 123456789;
unsigned long a = 6364136223846793005;
unsigned long c = 1442695040888963407;

static int ARRAY[ARRAY_SIZE];

unsigned long rand_number()
{
    seed = a * seed + c;
    return seed;
}

int main()
{
    for (unsigned long iter = 0; iter < N_ITERS; ++iter) {
        unsigned long idx = SHIFT * iter + (rand_number() % SHIFT);
        ARRAY[idx % ARRAY_SIZE] = ARRAY[(ARRAY_SIZE - idx) % ARRAY_SIZE];
    }

    return 0;
}

```

В качестве функции генерации случайных чисел был использован линейный конгруэнтный генератор, параметры которого предложены Дональдом Кнутом [22]. Параметры $ARRAY\_SIZE$ и $SHIFT$ определяют поведение бенчмарка, в то время как параметр N_ITERS не имеет существенного влияния на исход и может быть выбран произвольным образом.

Величины npi_{L_3} и npi_{ram} являются инвариантами для рассматриваемых бенчмарков, т.е. график зависимости $cpi(freq_{cpu})$ является графиком полинома 1-ой степени: $lat_{gen} = \alpha^{par} \cdot (C_{L_3} \cdot npi_{L_3} + C_{ram} \cdot npi_{ram}) = const$ для каждого бенчмарка, а значения npi_{L_3} и npi_{ram} не зависят от тактовой частоты. Таким образом, значение lat_{gen} может быть найдено применением линейной регрессии к формуле (32) для каждого бенчмарка. График с экспериментальными данными и применением линейной регрессии приведён на рис. 6.

Экспериментальные данные значений lat_{gen} для каждого бенчмарка приведены на рис. 7 (плоскость на рис. является аппроксимация данных линейной регрессией). Из приведённого графика можно наблюдать, что не существует явного вида зависимости $lat_{gen}(npi_{L_3}, npi_{ram})$, так как существуют дополнительные факторы, главным

Рис. 6: График зависимости $cpi(freq_{cpu})$ для набора бенчмарков

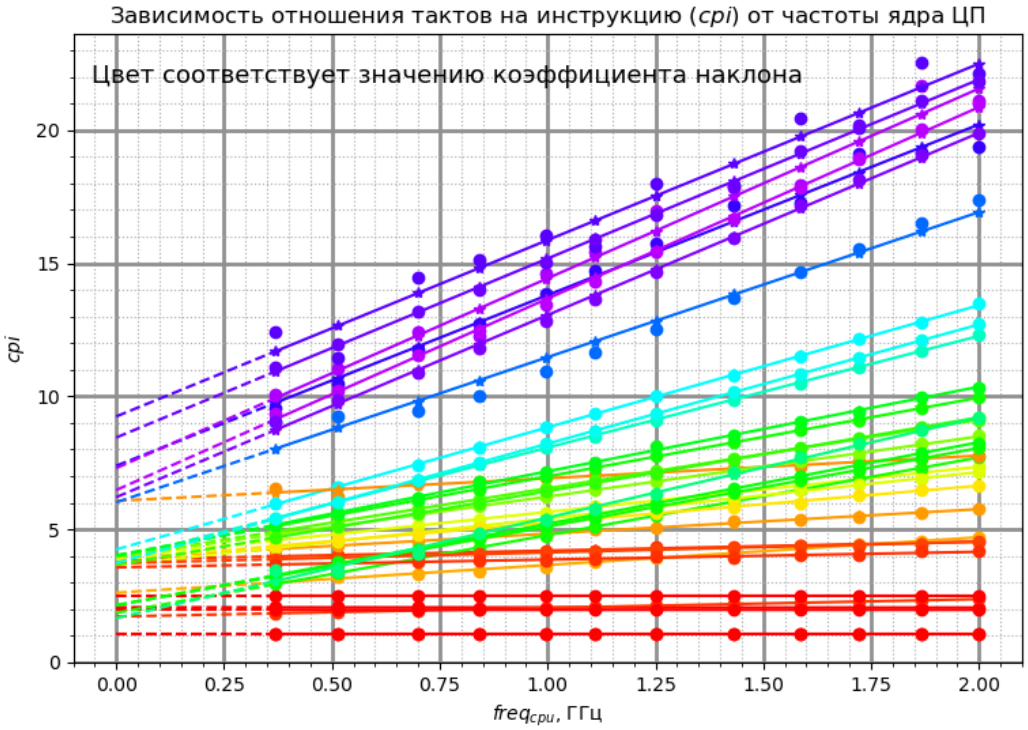
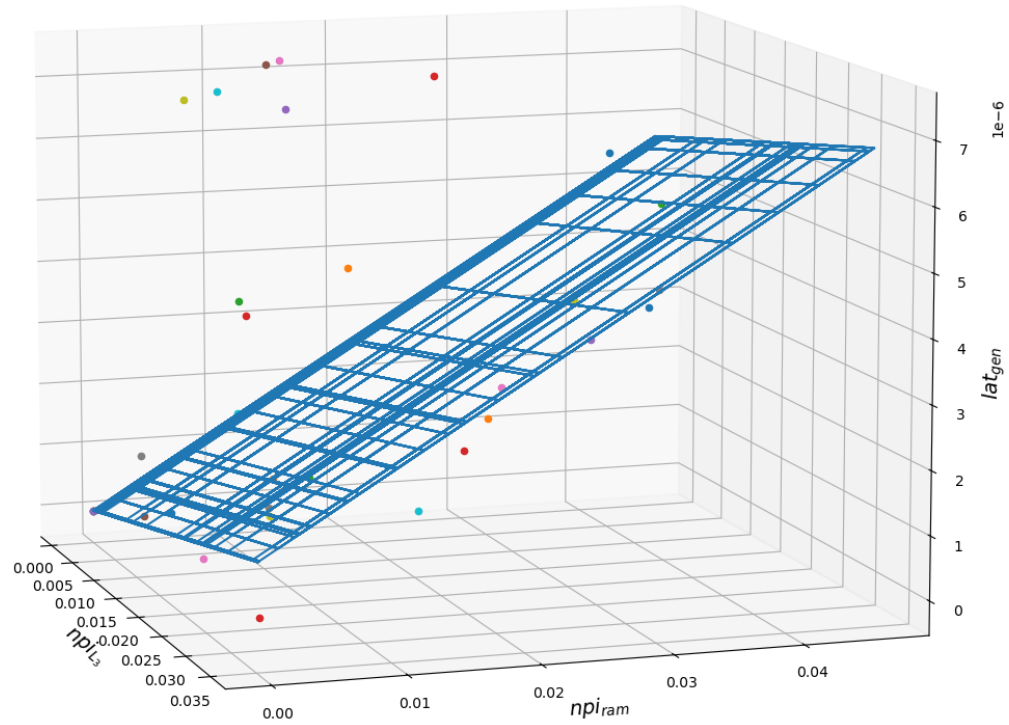


Рис. 7: График 3D зависимости $lat_{gen}(npi_{L_3}, npi_{ram})$



из который является коэффициент параллелизма обращений в память, который индивидуален для каждого приложения. С помощью теоретической формулы $lat_{gen} = \alpha^{par} \cdot (C_{L_3} \cdot npi_{L_3} + C_{ram} \cdot npi_{ram})$, принимая $\alpha^{par} = 1$, найдём усреднённые значения C_{L_3}

и C_{ram} из линейной регрессии:

$$C_{L_3} = 3.26727 \cdot 10^{-5} \text{ кГц}^{-1},\ C_{ram} = 1.22416 \cdot 10^{-4} \text{ кГц}^{-1}. \quad (33)$$

5.4 Реализация политики регулирования тактовых частот в ядре Linux

Реализация политики регулирования тактовых частот подразумевает реализацию следующих компонент:

1. Модуль, собирающий статистику микроархитектурных событий для каждого потока исполнения и ядра процессора (событий, описанных в главе 5.3).

2. Модуль, вычисляющий утилизацию для каждого потока исполнения и ядра процессора по формуле (23), регулирующий параметр α^{par} по формулам (30) и (31), а также выбирающий тактовую частоту ядра по формуле (29).

3. Модуль, выставляющий максимальную тактовую частоту среди выбранных с помощью модели для каждого ядра в ядерном кластере (т.к. тактовые частоты могут выставляться только целиком для всего кластера).

В качестве промежутка времени τ, использованном во всей работе, выбран квант времени планировщика, равный $CONFIG\_HZ$, настраиваемый в конфигурации ядра Linux. В конфигурации выставлено значение $CONFIG\_HZ = 100$, что означает частоту 100 Гц, то есть квант времени $\tau = 10$ мс.

Обновление статистики микроархитектурных событий необходимо фиксировать при следующих событиях:

1. По таймеру каждые τ мс в функции `scheduler_tick(void)`.

2. В функции `perf_rotate_context(struct perf_cpu_context *cpuctx)` при ротации незакреплённых микроархитектурных событий. Если количество измеряемых событий превышает количество физически доступных счётчиков, часть событий можно открепить, чтобы измерения по ним чередовались на одном счётчике и собирали менее точную усреднённую статистику.

3. В функции `context_switch(struct rq *rq, ...)` при смене контекста исполнения на ядре процессора.

Выставление тактовой частоты ядер необходимо совершать при следующий событиях:

1. По таймеру каждые τ мс в функции `scheduler_tick(void)` (после обновления статистики счётчиков микроархитектурных событий).

2. В функции `finish_task_switch(struct task_struct *prev)` при пробуждении ядра (перехода из состояния сна в активное) и назначении потока исполнения.

3. В функции `set_task_cpu(struct task_struct *p, unsigned int new_cpu)` в случае миграции потока исполнения на новое ядро процессора.

С коэффициентами $\gamma = 0.8$ в формуле (29) и $\beta = 0.8$ (31) тестирование на искусственном бенчмарке, основу которого составляет комбинация бенчмарков, приведённых в главе 5.3, приводит к результатам в виде уменьшения энергопотребления на 6.4%. Измерение энергопотребление было внедрено в Gem5 в виде формулы для мощности энергопотребления $P = C \cdot V^2 \cdot freq_{cpu}$, где ёмкость C считалась равной константой, а напряжение смоделировано формулой $V = V_0 \sqrt{\frac{freq_{cpu}}{freq_{cpu}^{min}}}$.

Анализ оптимизации производительности требует отдельного исследования, т.к. его нужно проводить для значительно большего количества бенчмарков с различными спецификами исполнения. В среднем на 1 секунду симуляции приходится порядка $2000 - 4000$ секунд, это означает, что для тестирования 10 бенчмарков, каждый из которых длиться по 100 секунд в симуляции, необходимо 41 сутки только для измерения данных, что затруднительно выполнить в данной работе.

6 Заключение

В данной работе были выполнены все поставленные цели и решены следующие задачи:

1. Проведено исследование существующих политик регулирования тактовых частот ядер центрального процессора, выявлены узкие места таких политик, проведён обзор литературы на тему альтернативных способов регулирования тактовых частот.

2. Разработана модель производительности ядер центрального процессора, учитывающая архитектуру взаимодействия ядер с системой памяти.

3. Разработан способ применения модели производительности к политике регулирования тактовых частот.

4. Модифицирован симулятор Gem5 для возможности измерения статистики микроархитектурных событий, связанных с кешами процессора, из пространства пользователя и из пространства ядра в операционной системе Linux.

5. Разработан набор бенчмарков, позволяющих вычислить коэффициенты в модели производительности.

6. Собраны статистические данные счётчиков микроархитектурных событий, необходимых для обеспечения работоспособности модели производительности, для набора бенчмарков с помощью симулируемой системы с процессором архитектуры ARM64.

7. Вычислены коэффициенты в модели производительности на основе собранных статистических данных.

8. В ядре операционной системы Linux версии 6.1:

   (a) Реализованы интерфейсы драйверов *clk*, *cpufreq*, *devfreq* для обеспечения возможности изменения тактовых частот ядер процессора и прочих компонент симулируемой системы Gem5 как из пространства ядра, так и из пространства пользователя.

   (b) Реализована политика регулирования тактовых частот ядер центрального процессора.

Планируемые исследования в будущем по данной теме:

1. Применение современных методов машинного обучения для создания более качественного алгоритма регулирования коэффициента параллелизма обращений в память в модели производительности ядер.

2. Построение модели энергопотребления процессора и оперативной памяти.

3. Разработка отдельной политики регулирования тактовых частот оперативной памяти на основе построенной модели производительности.

4. Разработка политики рекомендаций планировщику ядра Linux о размещении потоков исполнения на ядрах процессора на основе информации коэффициентов в модели производительности.

Список литературы

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

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

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

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

[5] Liang Wen Yew, Chang Ming Feng, Chen Yen Lin, and Wang Jenq Haur. Performance evaluation for dynamic voltage and frequency scaling using runtime performance counters // Applied Mechanics and Materials. — 2013. — Vol. 284. — P. 2575–2579.

[6] Chen Yen-Lin, Chang Ming-Feng, Yu Chao-Wei, Chen Xiu-Zhi, and Liang Wen-Yew. Learning-directed dynamic voltage and frequency scaling scheme with adjustable performance for single-core and multi-core embedded and mobile systems // Sensors. — 2018. — Vol. 18, no. 9. — P. 3068.

[7] Haririan Parham. DVFS and its architectural simulation models for improving energy efficiency of complex embedded systems in early design phase // Computers. — 2020. — Vol. 9, no. 1. — P. 2.

[8] Johnson Darrin Paul, Saxe Eric Christopher, and Smaalders Bart. Frequency scaling of processing unit based on aggregate thread CPI metric. — 2012. — Jul. 10. — US Patent 8,219,993.

[9] Hebbar Ranjan and Milenković Aleksandar. Pmu-events-driven dvfs techniques for improving energy efficiency of modern processors // ACM Transactions on Modeling and Performance Evaluation of Computing Systems. — 2022. — Vol. 7, no. 1. — P. 1–31.

[10] Thethi Sukhmani Kaur and Kumar Ravi. Power optimization of a single-core processor using LSTM based encoder–decoder model for online DVFS // Sādhanā. — 2023. — Vol. 48, no. 2. — P. 37.

[11] Deng Qingyuan, Meisner David, Bhattacharjee Abhishek, Wenisch Thomas F, and Bianchini Ricardo. Coscale: Coordinating cpu and memory system dvfs in server systems // 2012 45th annual IEEE/ACM international symposium on microarchitecture / IEEE. — 2012. — P. 143–154.

[12] 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.

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

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

[15] Overview Cortex A77. — Cortex A77 Documentation : 2020. — https://en.wikichip.org/wiki/arm_holdings/microarchitectures/cortex-a77.

[16] 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.

[17] 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.

[18] 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.

[19] Arm Ltd. — Arm PMU description : 2024. — https://github.com/ARM-software/data/tree/master/pmu.

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

[21] Linux Kernel Documentation. — Devicetree : 2024. — https://docs.kernel.org/devicetree/usage-model.html.

[22] Knuth Donald Ervin et al. The art of computer programming. — Addison-Wesley Reading, MA, 1973. — Vol. 3.