Общая характеристика работы

Актуальность темы. В современном мире, где преобладают мобильные системы, роль серверных вычислений постоянно возрастает. В период с 2013 до 2023 год объем денежных средств, вкладываемых международными компаниями в облачные вычисления увеличился в 5 раз и достиг сотни миллиардов долларов США. Многие современные приложения не могут быть запущены на мобильных и домашних устройствах в силу огромного количества потребляемых ресурсов. Сегодня цена вычислительных мощностей варьируется от 60 до 1500 долларов в месяц за единицу. У одной только российской компании Яндекс на сегодняшний день имеется от 150 000 до 200 000 серверов в 5 датацентрах по всему миру. Соответственно, вопрос производительности программного обеспечения на серверах стоит достаточно остро.

Из года в год различные вендоры вычислительной техники соревнуются в производительности своей продукции. Основными компонентами производительности процессора являются IPC (количество инструкций, исполняемы за один машинный такт), частота и количество выполненных инструкций. Со стороны компилятора можно повлиять на количество инструкций и лучше использовать возможности оборудования, чтобы увеличить IPC.

$$performance = IPC * \frac{frequency}{executed\_instructions} \quad (1)$$

GCC (GNU Compiler Colection) — это хорошо известный компилятор, который содержит десятки различных архитектурно-зависимых и независимых от архитектуры оптимизаций. Имеет открытый исходный код и ориентирован на оптимизацию C, C++ и Fortran.

Настройка компилятора для конкретной архитектуры или набора тестов — это необходимая работа, которая помогает компании продемонстрировать наилучшую производительность. Например, Дмитрий Мельник и др. оптимизировали GCC для ускорения библиотеки растеризации libevas, продемонстрировав, что GCC имеет недостатки в алгоритме распределения регистров для ARM. Кроме того,в этой же работе был проведен анализ различных типов предварительной выборки данных. Многие другие авторы предлагали методы автоматической конфигурации или настройки компилятора . Однако данная работа была сосредоточена не на настройке существующих оптимизаций, а на их усовершенствовании и разработке новых.

В настоящее время большинство компаний используют "SpecCPU 2017" для измерения производительности компьютера . Недавно был представлен новый инструмент оценки производительности под названием CPUBench. В ходе исследования выяснилось, что GCC очень хорошо

настроен для "SpecCPU 2017", в то время как для другого набора тестов (CPUBench) все еще остается много возможностей для улучшения.

Целью данной работы является увеличение производительности серверного процессора китайского производителя путем разработки нового поколения оптимизатора GCC. Для достижения поставленной цели необходимо было решить следующие **задачи**:

1. Выяснить слабые места анализируемой микроархитектуры процессора Kunpeng с помощью моделей и экспериментов.
2. Исследовать текущую производительность тестовых пакетов CPUbench и "SpecCPU 2017", скомпилированных GCC для процессора Kunpeng.
3. Исследовать похожие проблемы в научной литературе и предложить решение, позволяющее увеличить производительность целевых приложений.
4. Исследовать код целевого набора приложений на предмет неоптимальностей с точки зрения целевой микроархитектуры.
5. Разработать продуктовое решение в компиляторе GCC, позволяющее получить ускорение на целевых тестах компании.
6. Протестировать разработанные методы на реальных приложениях.

Тема и содержание диссертационной работы соответствует паспорту научной специальности 2.3.5. математическое и программное обеспечение вычислительных систем, комплексов и компьютерных сетей, в частности, пунктам:

п. 1 – Модели, методы и алгоритмы проектирования и анализа программ и программных систем, их эквивалентных преобразований, верификации и тестирования.

п. 3 – Модели, методы, алгоритмы, языки и программные инструменты для организации взаимодействия программ и программных систем.

Научная новизна:

1. Выполнено оригинальное исследование производительности целевой микроархитектуры, которое показало наличие недостатков.
2. Разработано и интегрировано в продукт более десяти различных оптимизаций в компиляторе GCC, которые помогают компенсировать найденные недостатки.
3. Впервые была исследована проблема производительности широких инструкций доступа в память и предложена оптимизация, решающая ее.
4. Впервые был представлен алгоритм автоматического подбора вероятностей условных переходов для компилятора GCC.

Практическая значимость работы заключается в использовании разработанных методов и алгоритмов в компиляторе с открытым

исходным кодом ООО «Техкомпания Хуавэй» openEuler GCC и его инфраструктуре для оптимизации приложений и последующем использовании. Реализованные оптимизации, верификации и методы получения профильной информации позволили получить прирост производительности на целевых приложениях компании.

Теоретическая значимость диссертационной работы заключается в разработке новых алгоритмов и методов микроархитектурных оптимизаций, позволяющих раскрыть потенциал производительности целевой архитектуры.

Основные положения, выносимые на защиту:

1. Улучшение алгоритма преобразования условных переходов.
2. Алгоритм шаблонной оптимизации двойного умножения.
3. Алгоритм разбиения широких инструкций доступа в память.
4. Алгоритм слияния "хвостов" базовых блоков.
5. Алгоритм векторизации "ленивых" вычислений.
6. Методология оценки вероятностей условных переходов.

Достоверность полученных результатов и выводов обеспечивается подробным описанием проведенных экспериментом, а также возможностью их повторения. Результаты находятся в соответствии с данными, полученными другими авторами.

Апробация работы. Основные результаты работы докладывались на:

1. 63-й всероссийской научной конференции «Московского физико–технического института (национального исследовательского университета)» (МФТИ, Физтех), Москва, ноябрь 2020 г.
2. 66-й всероссийской научной конференции «Московского физико–технического института (национального исследовательского университета)» (МФТИ, Физтех), Москва, ноябрь 2024 г.
3. 4-й Международной конференции о достижениях вычислительных технологий и искусственного интеллекта, Касабланка, 2024 г.

Личный вклад. В течение нескольких лет автор данной диссертации являлся техническим руководителем команды из 10 человек по разработке openEuler GCC в России. В течение этого времени и были разработаны представленные в данной работе алгоритмы и оптимизации. Часть из них, такие как улучшение преобразования условных переходов или векторизация циклов с малым числом итераций были разработаны автором полностью самостоятельно. Некоторые оптимизации, такие как векторизация "ленивых" вычислений, разбиение широких инструкций доступа в память, автоматический подбор вероятностей условных переходов и др., были разработаны студентами и инженерами, находящимися под непосредственным руководством автора данной диссертации.

Публикации. Основные результаты по теме диссертации изложены в 5 печатных работах, 2 из которых изданы в журналах, рекомендованных

ВАК, 1 — в периодических научных журналах, индексируемых Web of Science и Scopus, 2 — в тезисах докладов.

Содержание работы

Во **введении** обосновывается актуальность исследований, проводимых в рамках данной диссертационной работы, приводится обзор научной литературы по изучаемой проблеме, формулируется цель, ставятся задачи, излагается научная новизна и практическая значимость представленного исследования. В последующих главах сначала производится обзор существующих решений по данной тематике, затем описывается методология получения новых оптимизаций а также принципы и сложности замера итоговой производительности. В последней главе описаны алгоритмы, которые позволили улучшить производительность приложений и преодолеть недостатки целевой архитектуры.

Первая глава посвящена обзору существующих алгоритмов оптимизации для ARM64 и других подобных RISC-архитектур. Классические и известные компиляторные оптимизации (такие как удаление мертвого кода, поиск общих подвыражений, "ленивое" перемещение кода и подобные) не описываются в данной работе и считаются общеизвестными, однако могут быть описаны их модификации, которые позволяют получить улучшение производительности для архитектуры ARM64. GCC является одним из наиболее известных статических трансляторов кода. Количество оптимизирующих проходов этого компилятора исчисляется сотнями, и, тем не менее, его улучшение происходит до сих пор, и каждый год обнаруживаются новые возможности для оптимизации даже в классических проходах.

В разделе 1.1 приводится описание методологии распределения регистров и ее современные улучшения. Распределение регистров наравне с выбором и расстановкой инструкций является одной из самых сложных оптимизаций в компиляторе. В основе распределения регистров лежит проблема ограниченности размера регистрового файла аппаратуры. Компилятор внутри себя чаще всего использует так называемые виртуальные регистры, количество которых неограниченно, и компилятор не заботится об их переиспользовании. Существуют два базовых алгоритма решения данной задачи: раскраска графа и линейное сканирование. Алгоритм раскараски графа чаще всего дает лучшее решение, однако, когда речь идет о системах, к которым выдвигаются требования высокой производительности, например, бинарная трансляция, то часто используется метод линейного санирования. Рассматривается оптимизация, в которой вводится понятие "активная в будущем" переменная - переменная, которая будет использоваться инструкциями

при дальнейшем сканировании. Это позволило добиться того, что для 90 % инструкций и 80 % методов модель анализа интервалов жизни стала ненужной. Другая рассмотренная работа показывает, что современный рассматриваемый в работе компилятор все еще может генерировать лишние инструкции загрузки из памяти в следствие неточности округлений, связанных с использованием целочисленных вероятностей в компиляторе GCC.

В разделе 1.2 рассматривается оптимизация векторизации, развитие которой тесно связано с развитием современных векторных архитектур. Векторизация до сих пор является слабым местом для компиляторов. До сих пор авторы многих публикаций вынуждены векторизовать программный код приложений и библиотек вручную. В разделе приводятся примеры статей, в которых авторы занимаются ручной векторизацией кода. расширений. В рассматриваемых работах по автовекторизации указывается, что основной преградой для векторизации является сложная структура графа потока управления, которая препятствует стандартным алгоритмам векторизации кода. Так, например, в листинге 1 исполнение условия является управлением, препятствующим векторизации.

Листинг 1 Пример цикла, содержащего управления

```
int arr[n], a[n],  b[n], out[n];
... code ...
for (int i = 0; i< n; i++) {
        if (cond1[i] & cond2[n-1]){
                out[i] = a[i] + b[i];
        }
}
... code ...
```

Другое направление, которое стоит отметить - это "библиотечная векторизация функций". В таких работах пользователи в ручном режиме превращают отдельные участки кода в библиотечные функции. Так, например, векторизация стандартных математических функций позволяет добиться значительного ускорения незримо для пользователя. К сожалению, такой подход не позволит векторизовать математическую функцию, написанную собственноручно.

В разделе 1.3 обсуждается проблема задержек, связанных с взаимодействием приложения и подсистемы памяти. Известно, что время доступа в оперативную память значительно превышает время исполнения одной инструкции. Для решения этой проблемы были созданы различные

техники кэширования и предзагрузки данных в кэш. Однако на кристалле не возможно разместить слишком сложную логику ввиду ограничений на размеры и потребление мощности, к тому же у компилятора или пользователя имеется больше информации о структуре программы, чем у аппаратуры во время исполнения. Классическим примером такой оптимизации является предзагрузка данных для циклов. Такая оптимизация позволяет добиться ускорения до 90 % на отдельных тестах.

Одним из современных направлений является разработка алгоритмов для предзагрузки данных при косвенных доступах в память. В работе сотрудников Кэмбриджского университета упоминается разработка алгоритма предзагрузки данных в компиляторе LLVM для косвенного доступа (рисунок 1). Их подход нацелен на системы с высокопроизводительными вычислениями и позволил получить ускорение от 30 % до 270 %, к сожалению не были продемонстрированы результаты на тестах SpecCPU.

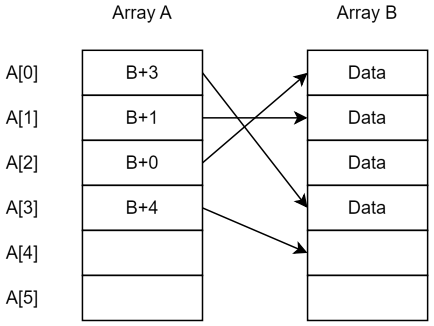

Рис. 1 — Пример косвенной адресации данных

В разделе 1.4 описывается задача подбора оптимальных параметров компиляции для целевого приложения на целевой платформе. Во время компиляции приходится решать очень много NP-полных задач. До сих пор огромное количество алгоритмов компилятора предполагают эвристические параметры и методы. Современное популярное направление нацелено на автоматизацию получения этих параметров или их полную замену. В обзорной статье была выделена общая схема автотюнера (Рисунок 2). Вверху рисунка: набор данных проходит через различные этапы обучения, на котором создается модель на основе обучающего набора данных. Внизу: набор данных проходит через различные этапы тестирования, где обучающая модель используется для прогнозирования результата. Описано множество работ, которые получают прирост производительности более 20 % благодаря этой технологии, однако до сих пор существует множество нерешенных проблем. Одной из таких проблем

является создание тренировочной базы тестов для задач машинного обучения. Для обучения модели количество тестов должно измеряться тысячами, и тесты SpecCPU не являются подходящими для такой задачи. Другая проблема связана с огромным пространством поиска оптимальных опций. Для сокращения такого пространства вводится алгоритм поиска критических флагов. Интересно, что помимо программы, алгоритм принимает на вход документацию компилятора GCC. Утверждается, что такой подход позволяет получить лучшие цифры по сравнению с другими системами автоматической настройки компилятора.

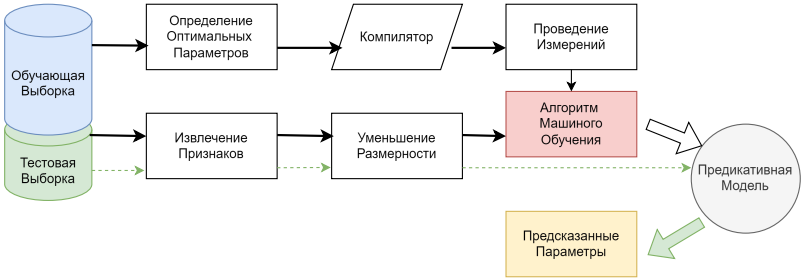

Рис. 2 — Общая схема автоматической настройки компилятора

В конце первой главы, **в разделе 1.5**, обсуждаются оптимизации с использованием профиля, и, хотя в данном исследовании динамическая профильная информация была недоступна, методы динамической оптимизации могут быть использованы в статических трансляторах с определенными ограничениями.

Использование профиля позволяет разрешить такие задачи, как девиртуализация, расстановка базовых блоков, выравнивание адресов и т.д. Необходимо сказать, что подобные исследование упирались в нежелание пользователей перекомпилировать свои приложения, а также в проблему зависимости профиля исполнения от входных данных. Поэтому долгое время популярными были только системы бинарной трансляции, которые могут динамически подстраиваться под меняющийся поток данных. Более того, давление бизнеса привело к тому, что использование профиля для получения результатов SpecCPU стало запрещено, для этого была выделенная отдельная категория "SpecCPU speed", которая позволяет показывать наилучший результаты независимо для каждого приложения.

В 2016 году Google разработала систему автоматического сбора профиля и оффлайн рекомпиляции приложений пользователей. Во время запуска пользовательских приложений фоном собиралась статистика с помощью сэмплирующего профилировщика, затем, когда сервер простаивал, запускался механизм обработки профильных данных и

процесс рекомпиляции. Такой подход по заверению авторов увеличивал производительность приложений на 10 %.

Сергей Лисицын в своей диссертации предлагает разрешить проблему зависимости профиля от входных данных с помощью версионирования отдельных участков программы, выбор между которыми делается динамически во время исполнения. Из недостатков подобного решения можно отметить увеличение размеров исполняемого файла.

С популяризацией машинного обучения появилась возможность генерации качественного синтетического профиля. Авторы статьи натренировали бустинг над деревьями для генерации профильной информации, что в свою очередь позволило компилятору использовать этот профиль и применять соответствующие оптимизации. С помощью данного подхода авторам удалось добиться ускорения в 1.6 процента в среднем с максимальным результатом в 16 % на интерпретаторе языка Python.

Во второй главе рассматривается методология поиска различных неоптимальностей в коде приложений и замера производительности. Для решения поставленной задачи - улучшения компилятора - необходимо определить неоптимальные места в коде, сгенерированным компилятором, которые при исполнении показывают недостаточную эффективность или вызывают задержку конвейера исполнения.

В разделе 2.1 дается описание целевой платформы, под которую разрабатывались оптимизации. Для проведения исследований был выбран широко распространенный сервер компании Huawei - Kunpeng920. Он базируется на архитектуре ARM V8.2-A. Основным конкурентом данного процессора на архитектуре ARM является Ampere Altra Server. Исследуемая модель процессора создана по 7-нанометровой технологии и оснащена 64 ядрами с тактовой частотой 2.6 ГГц. Модель включает в себя ряд аппаратных ускорителей, в том числе криптографии (MD5, HMAC, CMAC, AES, DES/3DES, SHA1, SHA2) и алгоритмов сжатия (GZIP, LZS, LZ4). Каждый чип состоит из двух вычислительных кристаллов (SCCL - Рисунок 3) и одного кристалл интерфейса (SICL - Рисунок 4).

Кристалл интерфейса, соединенный через общую шину, содержит модуль ускорителя криптографии, интерфейсы ввода/вывода, PCIE и т.п. Каждый вычислительный кристалл содержит 8 кластеров центрального процессора (CCL). В свою очередь, кластер центрального процессора состоит из 4х вычислительных ядер, 4х блоков кэширования первого уровня (64K для данных и 64K для инструкций), 4х блоков кэшировния второго уровня и блока тэгов для кэша третьего уровня. Кэш третьего уровня располагается отдельно внутри вычислительного кристалла, присоединенный к общей шине, в нем также могут храниться данные из других вычислительных кристаллов. Каждое ядро представляет собой 4-х

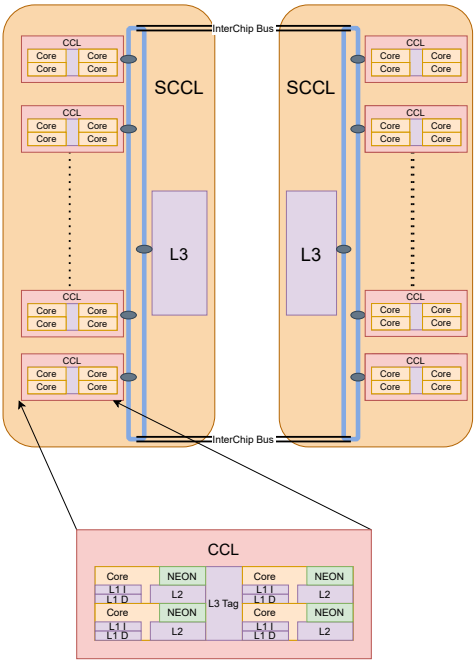

Рис. 3 — Схема целевого чипа

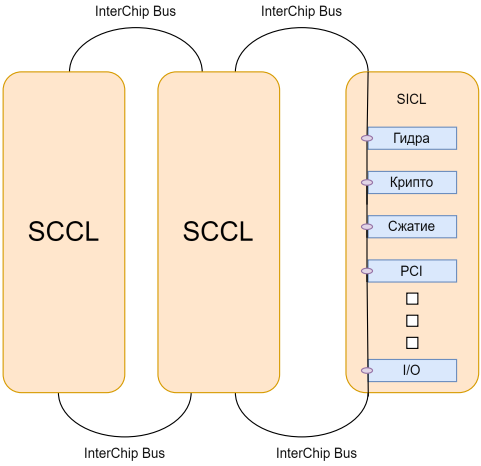

Рис. 4 — SICL модуль

канальный суперскалярный модуль с возможностью изменения порядка исполнения (superscalar, out-of-order).

В разделе 2.2 описаны два основных пакета приложений, на которых демонстрировались результаты разработанных методов : SpecCPU и CPUBench. "SpecCPU 2017" - набор тестов для оценки производительности вычислительных систем. Большая часть текущего исследования сосредоточена на улучшение производительности тестов с целочисленной арифметикой, однако некоторые общие подходы также применимы и к программам, использующим вычисления с плавающей точкой. Считается, что набор тестов SpecCPU является представителем современного рынка вычислений, поэтому многие компании при покупке вычислительных систем сравнивают производительность с использованием именно этого набора тестов.

В 2023 году Китайский институт электроники и стандартизации выпустил новый набор тестов производительности для вычислительных систем. В отличие от набора SpecCPU, интерфейс CPUBench разработан на языке python, а сам пакет имеет в себе программы, написанные на языке java. Авторами утверждается, что данный набор тестов является своеобразным расширением "SpecCPU 2017", которое нацелено на лучшее покрытие мирового рынка (в том числе китайского). Было продемонстрировано на 14 различных платформах, что данный набор тестов сохраняет корреляцию производительности, показываемую пакетом SpecCPU.

Можно заметить некоторую схожесть пакета "CPUBench int" с пакетом "SpecCPU int". Так, вместо интерпретатора языка perl представлен интерпретатор более современного языка python, добавлен дополнительный алгоритм компрессии/декомпрессии, парсинг XML документов заменен на парсер JSON файлов. А вот алгоритмов искусственного интеллекта здесь не наблюдается, зато присутствуют базы данных MySQL и криптографический инструмент openssl. Что касается тестов с плавающей точкой, большинство представленных программ в SpecCPU можно разделить на 2 категории: работа с графическими объектами и научные вычисления, в то время как в пакете CPUBench из графических приложений можно увидеть только алгоритм трассировки лучей. Однако CPUBench содержит в себе библиотеку градиентного бустинга lightgbm, активно применяющуюся в машинном обучении.

В разделе 2.3 излагается методология получения результирующих цифр, основанная на многолетнем научном опыте. Пусть REF_TIME_i - время исполнения теста, на некоторой референтной машине, пусть наш набор содержит M тестов, каждый из которых мы запускаем N раз, то итоговое значение производительности оценивается по следующей формуле.

$$RATE = \left(\prod_{i=1}^{M} \frac{REF\_TIME_i}{MEDIAN(TIME_{i1}, TIME_{i2}, ..., TIME_{iN})} \right)^{\frac{1}{M}} \quad (2)$$

Также важной частью считается настройка окружающей системы, направленная на уменьшение флуктуаций времени исполнения тестов и лучшей утилизации тестовой системы. Автор разбирает следующие проблемы, с которыми он столкнулся во время проведения замеров для алгоритмов, реализованных в данной диссертации.

1. Троттлинг.
2. Неполная утилизация ресурсов системы.
3. Рандомизация размещения адресного пространства (ASLR).
4. Частота обновления оперативной памяти.
5. Занятость ресурсов внешними программами.

В разделе 2.4 приводятся основные методы изучений целевых приложений для последующей разработки компилятивных алгоритмов. Рассмотрены методы различного типа профилирования: **gprof**, **perf**, **callgrind**, симуляция целевой архитектуры на **GEM5** и обратная разработка с помощью **Radare2**.

Третья глава посвящена описанию разработанных оптимизаций. Не все представленные оптимизации были приняты сообществом openEulerGCC [1] по разным причинам. Некоторые из описанных далее оптимизаций будут представлены сообществу позднее, а некоторые, возможно, будут заменены другими подходами. Тем не менее автор считает, что исследованные подходы также представляют научный интерес.

В разделе 3.1 описаны улучшения существующих оптимизаций в компиляторе GCC. Изложение начинается с преобразования условных переходов (If-conversion), которое заменяет инструкцию перехода и зависящий от него поток управления предикатным исполнением, соединяя тем самым две различные ветки потока управления в одну для последующего совместного исполнения. Обычно эта оптимизация основана на представлении SSA, однако в компиляторе GCC используется другой подход. На этом этапе SSA-форма отсутствует, что может привести к невозможности проведения преобразования из-за одинаковых имен определений в базовых блоках. Предлагаемое решение содержит принудительное переименование регистров. Регистры коллизий определяются как:

$$rename\_candidates = DEFS_{left\_bb} \cap USES_{right\_bb} \quad (3)$$

Если $rename\_candidates[i]$ все еще жив в конце базового блока BB, то трансформация не может быть применена.

[1] https://gitee.com/src-openeuler/gcc/

Подобная оптимизация имеет определенные ограничения: слишком агрессивное преобразование может привести к излишнему перекладыванию данных на стек в случае нехватки регистров или к замедлению производительности, связанному с ограниченной возможностью аппаратуры распаралеливать исполнение скалярного кода (Super Scalar). Поэтому вводятся функции стоимости для данной оптимизации: главным, однако далеко не единственным критерием применения преобразования условных переходов является итоговый размер базового блока. В разработанной модели этот параметр может задаваться пользователем, однако на исследуемой машине эмпирически был выведен ограничивающий размер результирующего базового блока, равный 48 инструкциям.

Другим примером, приводимым в данном разделе, является векторизация циклов с небольшим числом итераций. В ходе текущего исследования было обнаружено, что векторизация генерирует "хвосты" (т. е. векторизованный код для меньшего коэффициента векторизации), но не использует их, когда фактическое количество итераций равно коэффициенту векторизации "хвоста". Чтобы это исправить, предлагается простое решение в виде добавление дополнительной проверки во время исполнения и , если итерационное пространство все еще может быть векторизовано, то такой цикл необходимо исполнить в "хвостовой" части оригинальной векторизации.

Раздел 3.2 посвящен шаблонным оптимизациям. Первым примером является оптимизация двойного умножения - шаблонное преобразование компилятора, предназначенное для преобразования алгоритма 64-битного умножения в эффективные инструкции. Таким образом, программа может лучше использовать возможности аппаратуры и повысить производительность всего приложения.

Такие вычисления отыскиваются с использованием существующего механизма поиска шаблонов GCC и преобразуются в одиночные умножения более широких типов.

Еще одним примером шаблонных оптимизация является шаблонное преобразование криптографических алгоритмов. Целевая машина имеет на плате расширение криптографии, которое включает специальные инструкции для crc32 и AES. Их можно использовать напрямую через встроенный язык ассемблера или встроенные функции компилятора. Добавлены оптимизационные проходы внутри компилятора, которые могут определять возможность использования инструкций в соответствии с семантикой кода. Оптимизация отыскивает весь алгоритм, включая предварительно рассчитанные таблицы, и статически проверяет, что все предварительно рассчитанные таблицы не изменяются во время выполнения.

Следующие шаги описывают преобразование шаблона AES:

1. **Сбор ссылок на таблицы AES**: Разработан оптимизационный проход внутри компилятора GCC для поиска ссылки на соответствующие таблицы шифрования/дешифрования AES. Такие инструкции являются отправной точкой для дальнейшего анализа.
2. **Формирование раундов AES**: Анализируются ссылки на таблицы и собираются инструкции, выполняющие вычисления, относящиеся к AES. связывая их вместе в блоки и раунды.
3. **Проверка шаблона AES**: Анализируются раунды и связываются вместе.
4. **Генерация кода AES**: Генерируется код AES для всех найденных раундов.

Последним примером шаблонных оптимизаций является шаблонная подстановка инструкций. Одной из основных задач компилятора является выбор наиболее подходящих инструкций, которыми можно будет лаконично и в то же время оптимально с точки зрения производительности выразить внутреннее представление программы.

Так, арифметическое выражение вида

$$B = (((A >> 15)\&0x00010001) << 16) - ((A >> 15)\&0x00010001)$$

может быть транслировано в (Листинг 2)

Листинг 2 Оптимальный выбор инструкций

```
uzp1 v17.8h, v18.8h, v17.8h
```

В разделе 3.3 описывается два метода оптимизации косвенных переходов: статический и динамический.

Статический подход основан на анализе сигнатур функций. Предлагается анализировать сигнатуры функций для определений и вызовов. Если сигнатура вызываемой функции совпадает с сигнатурой определения то функция определения считается кандидатом. В случае единственного кандидата, косвенный вызов функции заменяется вызовом найденной процедуры. Если же кандидатов несколько, то приходится выстраивать цепочку сравнений адресов переходов. что не всегда является оптимальным

Динамический метод наследует идею JIT-компиляции (Just-in-Time compilation), однако здесь предлагается добавить буферный участок кода, который будет модифицироваться во время исполнения приложения. Статический компилятор (GCC) оборачивает косвенный вызов функции

в специальное библиотечное макро-определение "DDL_GOTO" или "DDL_CALL". Внутри определения вместо каждой инструкции перехода генерируется "окно" в виде заранее определенного количества NOP инструкций и вызовов библиотечных функций сбора статистики и замены инструкций NOP условными переходами при превышении счетчиков. Оригинальный косвенный переход сохраняется в самом конце. Во время исполнения пользовательской программы отдельные ее участки начинают работать, как JIT-компиляторы: собирать статистику и принимать решение о ретрансляции участка. Однако накладывается ограничение в виде конечного пространства в памяти для трансформации, которое было заложено статическим компилятором. Такой подход позволяет динамически ранжировать таблицу переходов во время исполнения.

Статический подход способствовал улучшению тестов tpcc и tpch. Динамический подход не показал эффективности на всем тестовом пакете, однако на небольших мотивационных тестах наблюдается улучшение производительности до 200 %, когда количество адресов перехода находится в диапазоне от 4 до 8, в иных случаях может наблюдаться деградация. Деградация в 10 % также наблюдалась на отдельных тестах пакета "SpecCPU 2017".

Раздел 3.4 посвящен разбиению широких инструкций доступа в память. В процессе исследования было обнаружено, что в двух случаях использование широкого доступа к памяти может возникнуть потеря производительности.

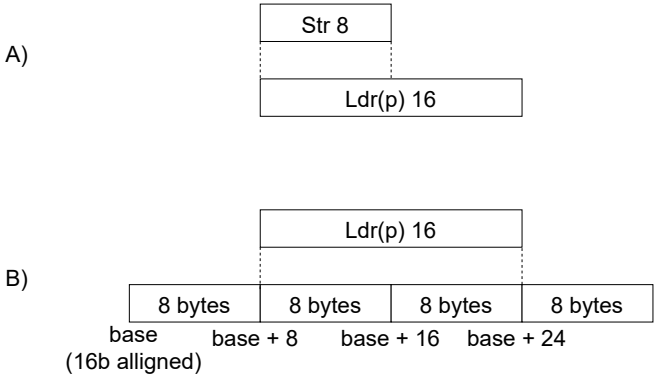

Рис. 5 — Два случая, когда использование широкого доступа в память приводит к замедлению

Один из них (рисунок 5) - это хорошо известный "невыровненный доступ". Было выявлено, что широкий доступ к памяти должен быть кратен его формату, в противном случае производительность снижается (даже когда речь идет о загрузке 2х независимых регистров). С другой

стороны, было показано, что Kunpeng 920 не может быстро обрабатывать зависимости чтения после загрузки в память, если они имеют разные размеры. Аппаратная оптимизация пересылки сохраняемого значения (store-to-load forwarding) не может быть выполнена в таком случае.

Для решения этой проблемы был разработан алгоритм, который находит инструкцию определения для базового регистра адреса широкой загрузки. Затем алгоритм ищет все использования этого определения, кроме оригинального. Если алгоритм находит сохранение в память с той же базой, то проверяется целесообразность разделения исходного широкого доступа к памяти. Было выявлено, что такой подход разумен, если расстояние между загрузкой и сохранением составляет менее 16 инструкций.

В разделе 3.5 описывается использование информация из анализа диапазона значений в компиляторе для уменьшения размеров типов переменных, что можэет благоприятно сказаться на последующей векторизации.

Раздел 3.6 описывает проблему семантики "ленивых" вычислений с точки зрения производительности конечного кода. Дело в том, что такая удобная с точки зрения программирования семантическая конструкция препятствуем стандартным алгоритмам преобразования условных переходов и векторизации. Так, код из листинга 3 не может быть векторизован базовым алгоритмом из-за возможности выхода за границу массива и последующей ошибки сегментации.

Листинг 3 Кандидат для векторизации "ленивых" вычислений

```
... code ...
if (arr[len] != const1 || arr[len +1] != const2
        || arr[len+2] != const3  || arr[len+3] != const4) {
        /* some code */
}
... code ...
```

Решением данной проблемы стало версионирование кода на векторизованный участок и скалярный. Передача управления на векторизованный участок происходит только в случае, если динамическая проверка подтверждает, что все доступы в векторизованном коде будут находится в одной странице памяти.

Данный подход вставляет в код одну дополнительную проверку, следовательно, в некоторых случаях время исполнения программы может увеличиться, однако на целевых тестах такого не наблюдалось, наоборот, наблюдалось ускорение теста gzip.

В разделе 3.7 описывается алгоритм слияния "хвостов" базовых блоков. Было продемонстрировано, что самая современная версия компилятора (GCC 14.2) генерирует в определенных ситуациях множество копий одного и того же кода. Объединение таких кусков может помочь оптимизации преобразования косвенных переходов, а также уменьшить размер исполняемого файла. Алгоритм состоит в поиске частичных общих участков и объединении их в единый базовый блок-наследник. Оптимизация не показала существенного улучшения производительности целевых приложений, однако деградации также не было обнаружено.

Раздел 3.8 кратко описывает оптимизацию предзагрузки косвенных доступов в память. Алгоритм пытается распознать циклы на межпроцедурном уровне, найти индуктивные переменные, а затем вставить предзагрузку данных на случай следующего вызова функции для косвенной адресации. Здесь возникает та же проблема, что и для векторизации "ленивых" вычислений, поэтому приходится вставлять схожую динамическую проверку на то, что все загрузки из памяти, которые необходимо сделать для вычисления следующего адреса доступа, находятся в уже доступных страницах. Также использует информация, если она доступна, о размере массивов, чтобы минимизировать накладные расходы динамической проверки.

Раздел 3.9 посвящен новой методологии подбора вероятностей условных переходов. Идея заключается в попытке повторить производительность, полученную с помощью технологии PGO без использования реального профиля. Для решения этой проблемы предлагается воспользоваться моделью машинного обучения, которая в качестве признаков будет принимать информацию об условном переходе и о структуре кода вокруг него. Тренировочный набор данных был создан с использованием набора программ из пакета ExeBecnh. Пакет содержит сотни маленьких приложений, которые можно быстро перетранслировать при необходимости.

На рисунке 6 изображен процесс сбора тренировочного набора данных.

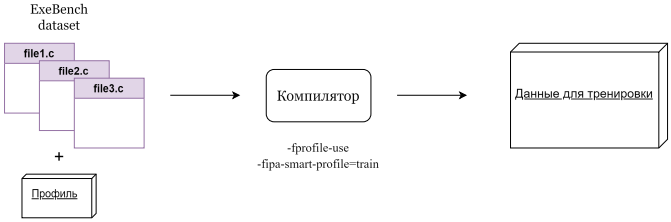

Рис. 6 — Сбор данных для тренировки

Для обучения используется библиотека **XGBoost**, которая строит решающие деревья над собранным набором данных, модель сохраняется в бинарном формате, для последующего использования в компиляторе (рисунок 7).

Рис. 7 — Тренировка модели

Наконец, обученная модель может прогнозировать вероятности переходов без использования каких-либо данных профиля, а новый проход в компиляторе GCC включает оптимизации с профилем. Библиотека **XGBoost** имеет API на языке C, который позволяет интегрировать этап прогнозирования в проход без использования **Python**. Достаточно обучить модель один раз, чтобы потом использовать ее постоянно. Во время процесса компиляции анализ собирает информацию об условных переходах внутри программы в векторе признаков и передает ее функции **XGBoost** для прогнозирования. Результатом стало улучшение производительности тестов gromacs и openfoam на 10 % и 5 % соответственно, однако присутствовала деградация в 5 % на тесте phyml. Такой результат показывает целесообразность дальнейших исследований в этой области.

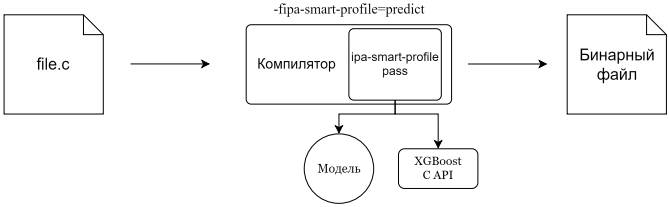

Рис. 8 — Запуск модели во время компиляции целевого приложения

Ранее упоминались существенные преимущества преобразования условных переходов. Однако **в разделе 3.10** продемонстрирована в некотором смысле обратная оптимизация. Рассмотрим немного упрощенный пример из теста phyml на листинге 4.

Такое выражение накладывает на переменную ns ограничение ($ns == 4$ или $ns == 20$). К сожалению работа с такого рода решеткой очень затруднительна для оптимизации распространения констант. Поэтому предлагается облегчить работу таким проходам

Листинг 4 Пример кандидата для оптимизации разбиения условных выражений из теста phyml

```
if(tree->mod->ns == 4 || tree->mod->ns == 20) {
        foo(tree);
}
```

(локальному и глобальному распространению константных выражений) трансформировав такой в листинг 5.

Листинг 5 Преобразованный листинг 4

```
if(tree->mod->ns == 4) {
        foo(tree);
} else if (tree->mod->ns == 20) {
        foo(tree);
}
```

Такой подход может приводить к существенному увеличению размера целевого кода, поэтому оптимизация накладывает ограничения на количество инструкций внутри функции(й).

В разделе 3.11 приводятся результаты замеров производительности на тестах пакета SpecCPU (Рисунки 11 и 12) и CPUBench (Рисунки 9 и 10).

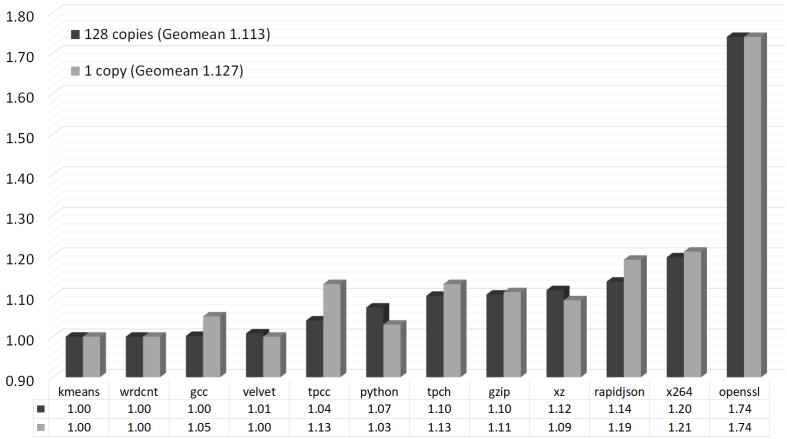

Рис. 9 — Результаты замеров производительности на тестах пакета "CPUBench int"

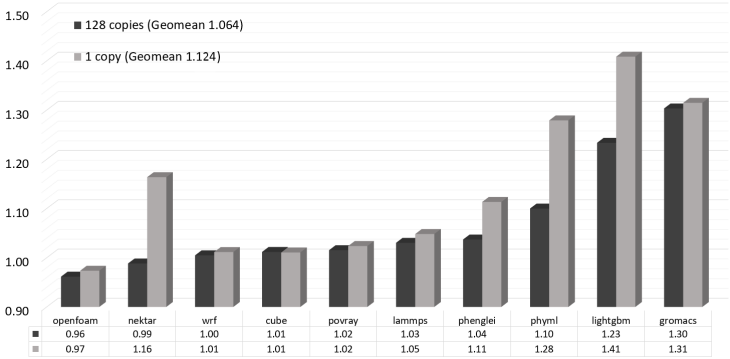

Рис. 10 — Результаты замеров производительности на тестах пакета "CPUBench fp"

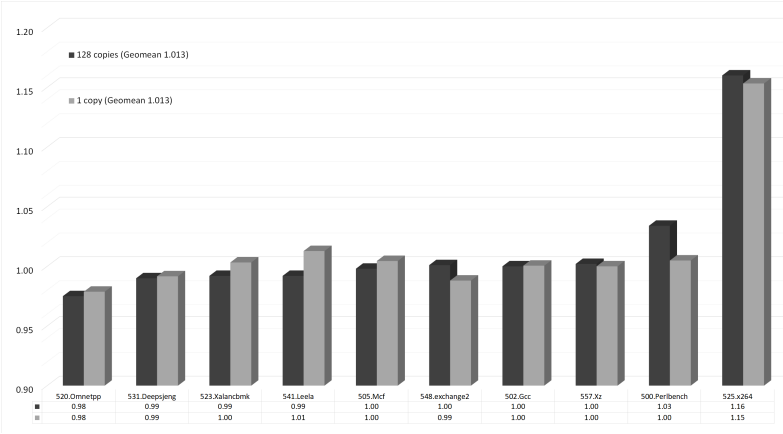

Рис. 11 — Результаты замеров производительности на тестах пакета "SpecCPU int"

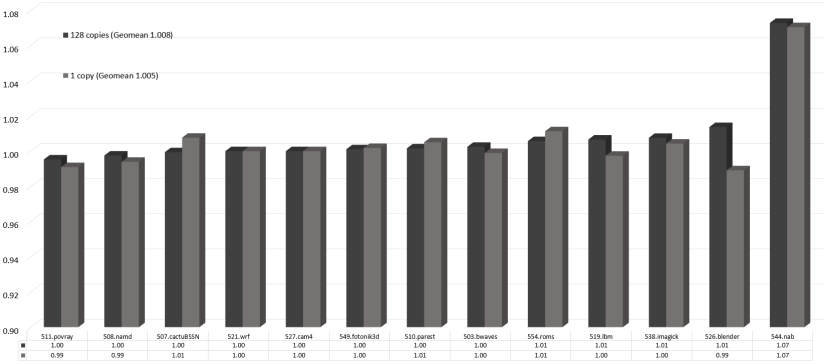

Рис. 12 — Результаты замеров производительности на тестах пакета "SpecCPU fp"

В **заключении** приведены основные результаты работы, которые заключаются в следующем:

1. На основе анализа современных технологий оптимизации приложений были выдвинуты гипотезы и направления исследования для последующей их оптимизации с учетом недостатков целевой архитектуры.
2. Предварительный анализ производительности с помощью таких приложений как **perf, radare2, GEM5** позволил доказать наличие возможности для оптимизаций целевых приложений на исследуемой микроархитектуре.
3. Для выполнения поставленных задач было создано семь дополнительных проходов в компиляторе GCC, а также предложено 5 улучшений существующих оптимизаций.
4. Разработанное решение позволило продемонстрировать улучшение производительности в 11 % на целевых тестах пакета "CPUBench int" с улучшением до 74 % на отдельных приложениях.
5. Разработанное решение позволило продемонстрировать улучшение производительности в 6 % на целевых тестах пакета "CPUBench fp" с улучшением до 40 % на отдельных приложениях. Для тестов пакета "CPUBench fp" улучшение производительности на одном ядре существенно отличается и составляет 12 % на целевых тестах.

В заключение хочется выразить благодарность всем коллегам и студентам, без которых данная работа не была бы возможной. Отдельная благодарность и большая признательность выражается научном руководителю Доброву А.Д за поддержку и помощь на всем научном пути автора.

Публикации автора по теме диссертации

1. Development of GCC Optimizations to Speed Up CPUBench Integer Benchmarks on ARMv8.2 [Текст] / C. Viacheslav [и др.] // Chinese Journal of Electronics. — 2022. — Т. 34. — С. 1—8. — URL: https://cje.ejournal.org.cn/en/article/doi/10.23919/cje.2024.00.105.
2. *Chernonog, V. V.* Статический и динамический подходы к преобразованию косвенных переходов [Текст] / V. V. Chernonog, I. L. Diachkov, A. D. Dobrov // Современные информационные технологии и ИТ-образование. — 2023. — Т. 19, № 2. — С. 355—364.
3. *Chernonog, V. V.* Оптимизация инструкций широкого доступа в память в архитектуре AArch64 [Текст] / V. V. Chernonog, E. Gadzhiev, A. D. Dobrov // Современные информационные технологии и

ИТ-образование. — 2024. — Т. 20, № 1. — URL: http://sitito.cs.msu.ru/index.php/SITITO/article/view/1067.

4. *Черноног, В. В.* Векторизация ленивых вычислений в линейном коде [Текст] / В. В. Черноног, И. М. Егоров // ТРУДЫ 66-й Всероссийской научной конференции МФТИ. Радиотехника и компьютерные технологии. — 2024.

5. *Черноног, В. В.* Применение алгоритмов машинного обучения в задаче региональной оптимизации в системе бинарной трансляции [Текст] / В. В. Черноног // ТРУДЫ 63-й Всероссийской научной конференции МФТИ. Радиотехника и компьютерные технологии. — 2020.

Черноног Вячеслав Викторович

Исследование и разработка нового поколения оптимизатора GCC для ARM64

Автореф. дис. на соискание ученой степени канд. тех. наук

Подписано в печать ___.___._____. Заказ № _____
Формат 60×90/16. Усл. печ. л. 1. Тираж 100 экз.
Типография _____