Оглавление

Стр.

Стр.

Введение

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 [1; 2] и достиг сотни миллиардов долларов США. Многие современные приложения не могут быть запущены на мобильных и домашних устройствах в силу огромного количества потребляемых ресурсов [3]. Сегодня цена вычислительных мощностей варьируется от 60 до 1500 долларов в месяц за единицу [4]. У одной только российской компании Яндекс на сегодняшний день имеется от 150 000 до 200 000 серверов в 5 датацентрах по всему миру [5]. 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 [6]. Со стороны компилятора можно повлиять на количество инструкций и лучше использовать возможности оборудования, чтобы увеличить IPC [7].

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

GCC (GNU Compiler Colection) — это хорошо известный компилятор [8], который содержит десятки различных архитектурно-зависимых и независимых от архитектуры оптимизаций [9]. 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. [10] 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 [11—13]. 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 [14; 15]. Недавно был представлен новый инструмент оценки производительности под названием CPUBench [16]. 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 машин, 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), Москва, ноябрь 2020 г.
2. 66-й всероссийской научной конференции Московского физико–технического института (национального исследовательского университета), 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.

Объем и структура работы. Диссертация состоит из введения, 3 глав, заключения и 1 приложения. Полный объём диссертации составляет 101 страницу, включая 35 рисунков и 5 таблиц. Список литературы содержит 144 наименования.

Глава 1. Обзор современных технологий компиляторной оптимизации

Первая глава содержит обзор 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 [17]. 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 [18]. Другим популярным транслятором с открытым исходным кодом является компилятор clang на базе llvm, который тоже не лишен недостатков [19].

В разделе 1.1 приводится описание методологии распределения регистров и ее современные улучшения, которые получают получить улучшение производительности приложений.

В разделе 1.2 рассматривается оптимизация векторизации, развитие которой тесно связано с развитием современных векторных архитектурных расширений.

В разделе 1.3 поднимается проблема задержек, связанных с взаимодействием приложения и подсистемы памяти.

В разделе 1.4 описывается задача подбора оптимальных параметров компиляции для целевого приложения на целевой платформе.

В конце первой главы (раздел 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.

1.1 Распределение регистров

Распределение регистров наравне с выбором и расстановкой инструкций является одной из самых сложных оптимизаций в компиляторе [20; 21].

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. Два классических подхода к решению данной задачи - это линейное сканирование [22; 23] и раскраска графа [24; 25]. Независимо от выбранного метода, он будет основываться на времени жизни переменной в коде. Временем жизни (интервалом жизни) называется последовательность инструкций от непосредственного определения переменной до ее последнего использования. Чтобы посчитать времена жизни всех переменных нужно воспользоваться следующей формулой, которая использует принцип восходящего анализа:

$$LiveOut[BB] = \bigcup_{stmn \in Succ(BB)} USE_{stmt} \cup (LiveOut[stmt] - VarKill_{stmt})) \quad (1.1)$$

где LiveOut(BB) – множество живых переменных на выходе из базового блока (BB), LiveIn(BB) – множество живых переменных на входе в базовый блок, use(BB) – множество переменных базового блока (BB), которые используются в нем до их переопределения, VarKillB – множество переменных базового блока, которые переопределяются.

При использовании раскраска графа в качестве вершин графа выбираются интервалы жизни переменных. Вершины графа соединяются дугой, если интервалы жизни пересекаются, после этого необходимо решить задачу о раскраске графа в N цветов, где N- количество физических регистров на целевой машине. Задача раскраски графа является NP-полной, однако в случае распределения регистров существует возможность исключать "неудобные" вершины для раскраски (переносить значения этих переменных в память), что значительно упрощает сложность алгоритма [24; 25].

Когда же речь идет о системах, к которым выдвигаются требования высокой производительности, например, бинарная трансляция, то часто используется метод линейного санирования, который по ходу своей работы линейно просматривает код, и, если видит конфликтующие интервалы,

Листинг 1.1 Пример интервалов жизни для алгоритма распределения регистров [10]

```
0 MOV R0 0xac434   | R0 [0,10]
1 LDR R1 [R0]      | R1 [1,7]
2 MOV R2 2         | R2 [2,8]
3 LDR R3 [R0,R2]   | R3 [3,6]
4 LDR R4 [R0,8]    | R4 [4,7]
5 ADD R5 R2 R1     | R5 [5,8]
6 MUL R6 R3 R5     | R6 [6,7]
7 MUL R1 R6 R4     | R1 [7,8]
8 ADD R2 R1 R5     | R2 [8,9]
9 ADD R3 R1 R2     | R5 [9,10]
10 STR R3 [R0]
```

Листинг 1.2 Первый шаг алгоритма распределения регистров для 3х физических регистров [10]

```
N INSN              R0  R1  R2  R3  R4  R5  R6
0 MOV R0 0xac434    A
1 LDR R1 [R0]       A   A
2 MOV R2 2          A   A   A
3 LDR R3 [R0,R2]    A   A   A   A
4 LDR R4 [R0,8]     A   A   A   A   A
5 ADD R5 R2 R1      A   A   A   A   A   A
6 MUL R6 R3 R5      A   A   A   A   A   A   A
7 MUL R1 R6 R4      A   A   A       A   A   A
8 ADD R2 R1 R5      A   A   A           A
9 ADD R3 R1 R2      A   A
10 STR R3 [R0]
```

то кладет первую возможную переменную в память [22]. Естественно, что найденное таким образом решение обладает меньшим качеством по сравнению с раскраской графа, тем не менее в статье Яна Роджерса [26] 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.

Листинг 1.3 Результат работы алгоритма распределения регистров [10]

```
N  INSN              R0  R1  R2  R3  R4  R5  R6
0  MOV R0 0xac434    r1
1  LDR R1 [R0]       r1  r2
2  MOV R2 2          r1  r2  r3
3  LDR R3 [R0,R2]    r1  M   r3  r2
4  LDR R4 [R0,8]     r1  M   M   r2  r3
5  ADD R5 R2 R1      M   r1  r2  M   M   r3
6  MUL R6 R3 R5      M   M   M   r1  M   r3  r2
7  MUL R1 R6 R4      M   r1  M       r3  M   r2
8  ADD R2 R1 R5      M   r1  r2          r3
9  ADD R3 R1 R2      r2  r1      r3
10 STR R3 [R0]
```

Так как задача является NP-полной, то часто найденное компилятором решение является неоптимальным. Так, например, в исследовании [10] было обнаружено, что лишние инструкции загрузки из памяти могут генерироваться в следствие неточности округлений, связанных с использованием целочисленных вероятностей в компиляторе GCC. Видно, что в конкретном примере в листингах 1.4 и 1.5 количество загрузок из памяти стало меньше после примененной оптимизации. Интересно отметить, что авторы также использовали следующие опции компиляции для улучшения генерации кода аллокатором регистров:

- **-fira-max-loops-num** - Ограничивает количество циклов внутри функции для региональной оптимизации распределения регистров.
- **-fira-coalesce** - Оптимистическое объединение регистров.
- **-fira-region** - Ограничение региона для распределения регистров.
- **-fno-ira-share-save-slots** - Отключает совместное использование слотов стека для сохранения регистров.
- **-fira-algorithm** - Изменение алгоритма раскраски графа.
- **-fno-ira-share-spill-slots** - Отключает совместное использование слотов стека, выделенных для виртуальных регистров.

Решение NP-полных задач чаще всего предполагает эвристический подход, поэтому нередки случаи использования методов машинного обучения, в частности, обучения с подкреплением для распределения регистров [27], что

Листинг 1.4 Базовый блок с излишней загрузкой из памяти [10]

```
.L133:
ldr lr, [fp, #-84]
mov r3, r1, asr #16
add r1, r1, r0
str r3, [lr, r2, asl #2]
ldr r3, [fp, #24]
add r2, r2, #1
cmp r3, r2
bgt .L133
```

Листинг 1.5 Базовый блок после исправления проблемы с целочисленной вероятностью [10]

```
L133:
mov r3, r1, asr #16
str r3, [lr, r2, asl #2]
add r2, r2, #1
cmp r9, r2
add r1, r1, r0
bgt .L133
```

позволило авторам получить улучшение производительности на целевых тестах в виде "SpecCPU 2017" вплоть до 4-5 % на отдельных программах.

1.2 Векторизация

Векторизация - вид распаралелливания программы, при котором однопоточное приложение, выполняющее одну операцию за единицу машинного времени, модифицируется для выполнения нескольких однотипных простых операций за раз. При этом простые скалярные операции заменяются векторными аналогами, способными производить операции над массивом данных. Чаще всего векторизацию подразделяют на цикловую и линейную. [28]

В процессорах семейства ARM64 существует несколько векторных расширений: NEON, SVE, SVE2.

Векторизация до сих пор является слабым местом для компиляторов. До сих пор авторы многих работ вынуждены векторизовать код вручную. В статье [29] авторы вручную выполняли векторизацию нейронных сетей для ARM64 и RISCV, объясняя это "компиляторными ограничениями". В другой статье авторы были вынужденны писать на ассемблере, чтобы раскрыть потенциал ARM SVE [30]. Обе статьи подчеркивают высокий прирост производительности (до 150%) на приложениях, однако неспособность компиляторов утилизировать в должной мере векторные расширения побуждает таких авторов проводить собственные исследования и разрабатывать ручные оптимизации.

В работе Angela Pohl etc. [28] рассматривается генерация эффективного векторного кода на примере 151 цикла. Авторы утверждают, что основной преградой для векторизации является сложная структура графа потока управления, которая препятствует стандартным алгоритмам векторизации кода. Так, например, в листинге 1.6 исполнение условия является управлением, препятствующим векторизации.

Листинг 1.6 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 ...
```

Авторы предлагают встраивать динамическую проверку, которая отвечает на вопрос, находится ли массив в единственной странице физической памяти. Накладные расходы остаются минимальными поскольку проверка вставляется единожды вне цикла. В главе 3.6 будет рассмотрено улучшение этого подхода, когда встраивание проверки вне цикла невозможно.

В продолжение предыдущей работы Bine Brank и Dirk Pleiter [31] провели исследования трех компиляторов: GCC, FCC, ACfL. было показано, что компиляторы способны векторизовать до 69% циклов из тестового пакета

TSVC2, состоящего из 151 цикла. Также было показано, что компиляторы, основанные на clang, продуцируют отличный код от GCC, в свою очередь GCC смог свекторизировать наименьшее количество циклов среди трех представленных компиляторов (рисунок 1.1).

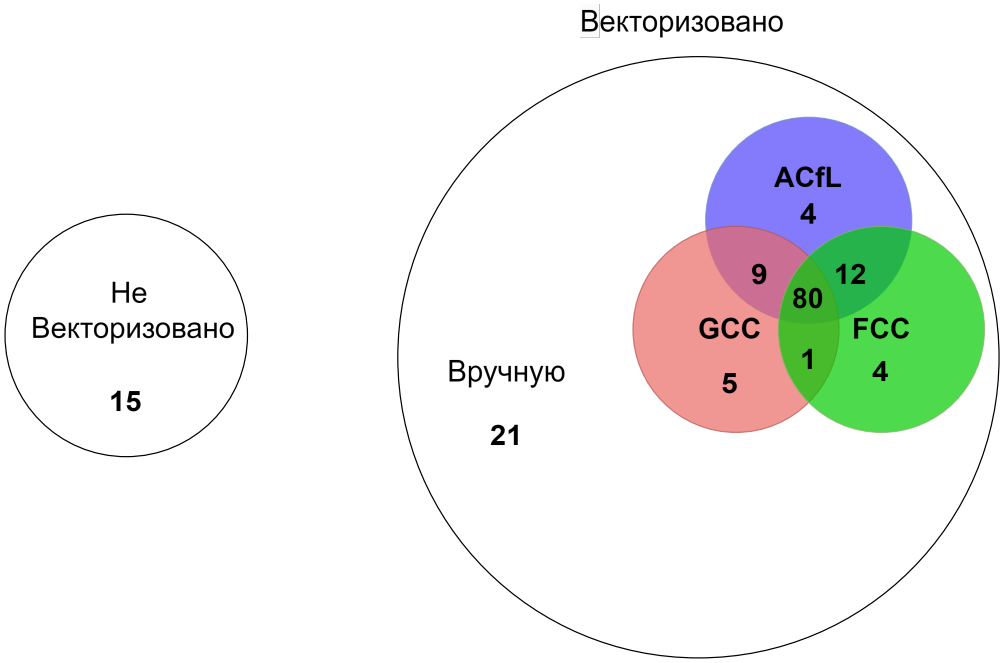

Рисунок 1.1 — Современное покрытие различными векторизаторами тестов TSVC [31]

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 [32] позволяет добиться значительного ускорения незримо для пользователя. К сожалению, такой подход не позволит векторизовать математическую функцию, написанную собственноручно.

1.3 Предзагрузка данных

Эффективное управление данными является одним из ключевых аспектов производительности целевого приложения. Известно, что время доступа в оперативную память значительно превышает время исполнения одной инструкции (рисунки 1.2 и 1.3). Для решения этой проблемы были созданы

различные техники кэширования и предзагрузки данных в кэш [33; 34]. Однако на кристалле не возможно разместить слишком сложную логику ввиду ограничений на размеры и потребляемую мощность, к тому же у компилятора или пользователя имеется больше информации о структуре программы, чем у аппаратуры во время исполнения. Поэтому уже в во второй половины 80-х годов начались попытки внедрения оптимизации предзагрузки данных [35].

Классическим примером такой оптимизации является предзагрузка данных для циклов [36]. Аккуратный анализ индукционных переменных, расстояния адресов доступа в память на чтений в цикле и эвристическое определение числа итераций для предзагрузки циклов позволило авторам статьи [36] получить ускорение в 11 % с пиковым ускорением в 50 % на отдельном тесте для платформы Shenwei.

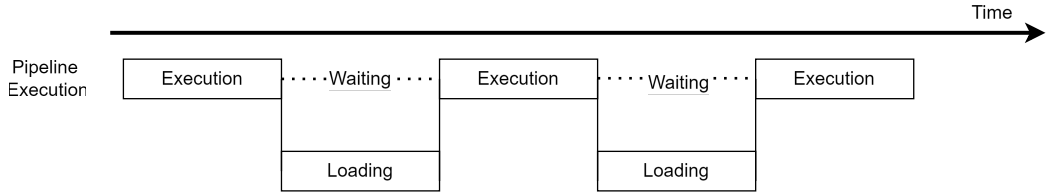

Рисунок 1.2 — Пример задержек конвейера при отсутствии предзагрузки данных [36]

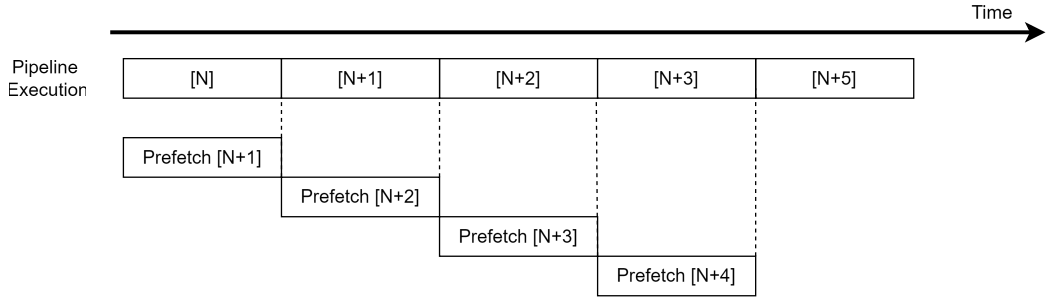

Рисунок 1.3 — Пример идеальной предзагрузки данных [36]

Анализ с использованием профиля и аппаратной поддержки записи последнего перехода на процессорах компании Intel позволила авторам еще одной статьи [37] разработать аналитическую модель с дешевым сбором статистики и быстрым подсчетом расстояния адресов для предзагрузки данных. Это позволило добиться феноменального ускорения в 30 % с ускорением до 90 % на отдельных тестах пакета APT-GET.

Одним из современных направлений является разработка алгоритмов для предзагрузки данных при косвенных доступах в память (пример косвенной адресации на рисунке 1.4). В работе сотрудников Кэмбриджского

университета [38] упоминается разработка алгоритма предзагрузки данных в компиляторе LLVM для косвенного доступа. 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. Стоит также отметить возможность оптимизации для уменьшение

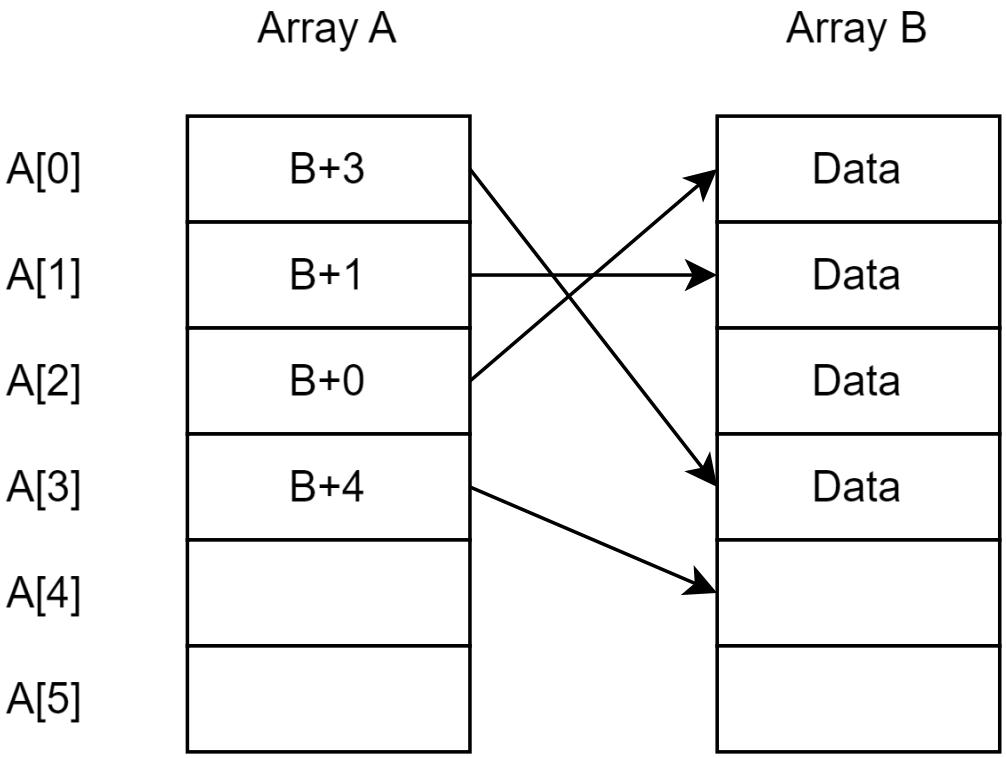

Рисунок 1.4 — Пример косвенной адресации данных

потребления энергии, что является критическим в современных системах. В статье [39] предлагается алгоритм версионирования кода для разных путей с точки зрения потока данных, что позволят сэкономить 10 % энергии и увеличить производительность на 2 %.

1.4 Настройка компилятора

Одним из современных направлений в области оптимизирующих компиляторов является их автоматическая настройка. 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 [40].

Чтобы подчеркнуть популярность данного направления, сошлемся на обзорную статью 2018 года [12], в котором упоминается более 80 статей, исследования которых направлены на проблему подбора опций компиляции, а также более 25 статей, исследующих проблему перестановки оптимизационных проходов. Там же была выделена общая схема автотюнера (Рисунок 1.5). 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.

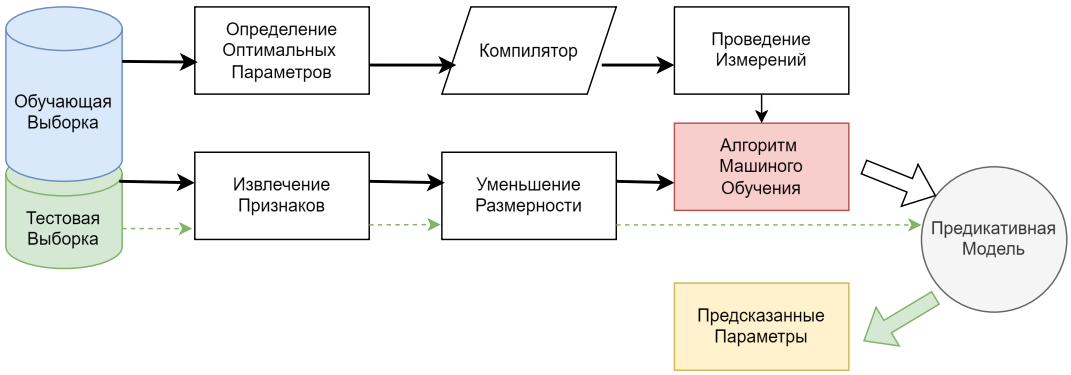

Рисунок 1.5 — Общая схема автоматической настройки компилятора [12]

В 2020 году вышла статья [41], в которой авторы с помощью технологии машинного обучения улучшили генератор кода для ускорителя нейронных сетей. Ускорение достигается за счет подборки двух гиперпараметров (количества ядер и схемы объединения слоев). Авторы утверждают, что их подход показывает практически такой же результат, как и метод полного перебора гиперпараметров, однако время, затраченное на подбор, оказывается значительно меньше.

В 2020 году автор данной диссертации рассматривал применение алгоритмов машинного обучения в системе бинарной трансляции для подбора оптимальных параметров компиляции региона [42]. Трансляция осуществлялась с архитектуры x86 на целевую VLIW-подобную архитектуру. С помощью решающих деревьев удалось добиться улучшения производительности на 10%.

Работа, вышедшая 2021 году [43], рассказывает об оптимизации тестов пакетов "SpecCPU 2006" и "SpecCPU 2017" для платформы SHENWEI.

Впечатляющие цифры в 14 % и 25 % ускорения были получены путем выделения горячих функций и итеративного определения опций компиляции. Такой подход нельзя назвать честным, поскольку очевидно, при подборе опций использовалась информация о входных данных программы. Однако такие цифры могут служить ориентиром для будущих исследований.

Так, например, была разработана система EAtuner [44] - основанная на эволюционных алгоритмах среда для автоматической настройки компилятора и определения подходящих алгоритмов для автотюнера. Было реализовано десять различных эволюционных алгоритмов и оценена их эффективность. Среднее полученное ускорение составило 20 %, однако было замечено, что алгоритмы показывают различную эффективность, зависящую от итераций компиляции. Авторы надеются, что их работа послужит инструментом для генерации тренировочного набора данных компиляции.

Создание наборов данных для последующего обучения компиляторных оптимизаций является отдельной сложной задачей, так как необходимо огромное количество программ, которые могут быть скомпилированы и исполнены, т.е иметь входные и выходные данные. Набор данных ExeBench [45] - первый представленный в открытый доступ набор данных состоящий из исполняемых и компилируемых функций, написанных на языке C. ExeBench содержит 4.5 миллиона компилируемых и 700 тысяч исполняемых функций. Для работы с этим набором данных был также разработан набор сопутствующего программного обеспечения на языке python, который позволяет делать выборки из набора с нужными характеристиками. Другим примером такого набора является CATBench, предлагающий разнообразный набор тестов, полученных из реальных приложений. Эти тесты выбраны для отражения уникальных характеристик задач автонастройки компилятора. Одновременно CATBench предлагает новый набор сложных задач для байесовской оптимизации с помощью экзотических поисковых пространств. Подобно ExeBench, CATBench имеет собственный набор сопровождающего программного обеспечения, включая python-интерфейс и компиляторы TACO [46] и RISE/ELEVATE [47]. Предоставляя стандартизированный набор тестов и унифицированный метод оценки, CATBench обеспечивает воспроизводимость сравнения, одновременно обеспечивая ускоренный прогресс в направлении более эффективных методов автонастройки.

Пространство поиска оптимальных опций достаточно велико, поэтому предлагается его сократить [48]. Для этого 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. В продолжение этой работы [49] предлагается легкий подход к обучению, который использует сравнительно небольшое количество информации о производительности времени выполнения для прогнозирования времени выполнения скомпилированной программы с различными комбинациями флагов оптимизации. Кроме того, чтобы уменьшить пространство поиска, авторами был разработан новый алгоритм роя частиц, который настраивает флаги оптимизации.

1.5 Оптимизации с использованием профиля

В попытках найти оптимальное решение для задачи оптимизации конкретного приложения, еще в 1994 году была предложена техника оптимизации с использованием профиля [50]. Общая схема такой модели изображена на рисунке 1.6. 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 [51]. 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 [52]. На рисунке 1.7 можно увидеть схематично устройство разработанной системы. Во время запуска пользовательских приложений фоном собиралась статистика с помощью

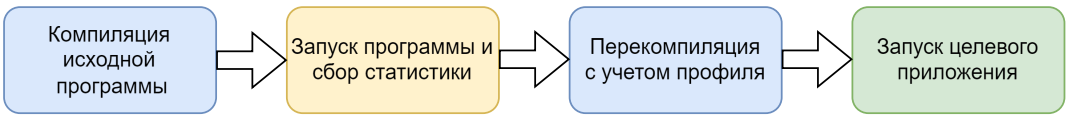

Рисунок 1.6 — Общая схема статической оптимизации с использованием профиля

сэмплирующего профилировщика, затем, когда сервер простаивал, запускался механизм обработки профильных данных и процесс рекомпиляции. Такой подход по заверению авторов увеличивал производительность приложений на 10 %.

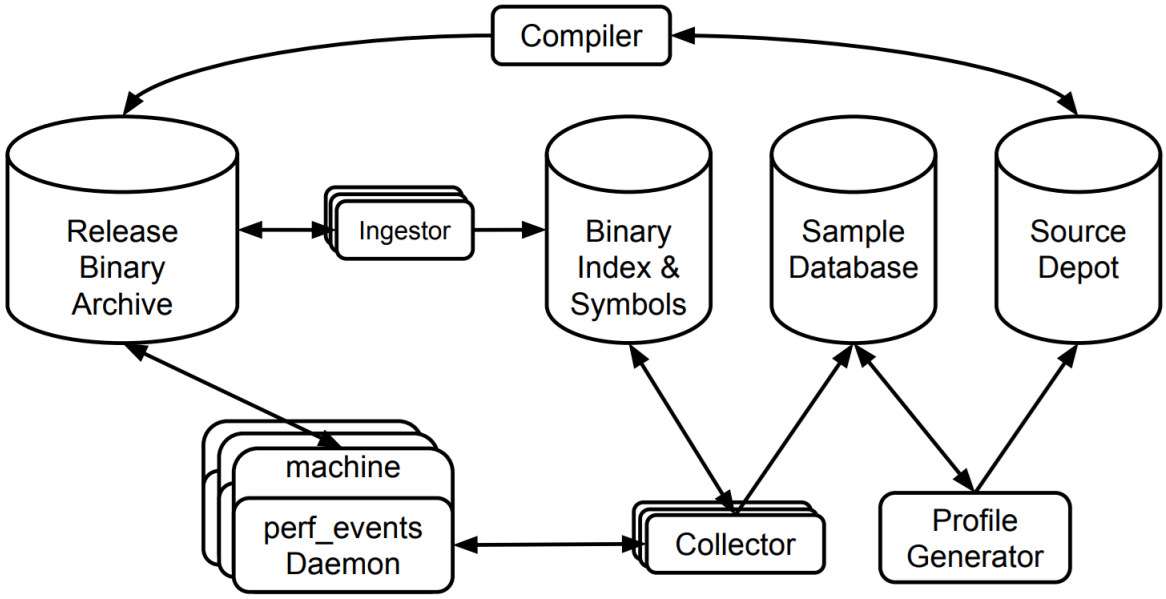

Рисунок 1.7 — Схема системы автоматического сбора профиля и рекомпиляции [52]

Сергей Лисицын в своей диссертации [53] предлагает разрешить проблему зависимости профиля от входных данных с помощью версионирования отдельных участков программы, выбор между которыми делается динамически во время исполнения. Из минусов подобного решения можно отметить увеличение размеров исполняемого файла.

С популяризацией машинного обучения появилась возможность генерации качественного синтетического профиля [54]. 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. На рисунке 1.8а изображена тренировка

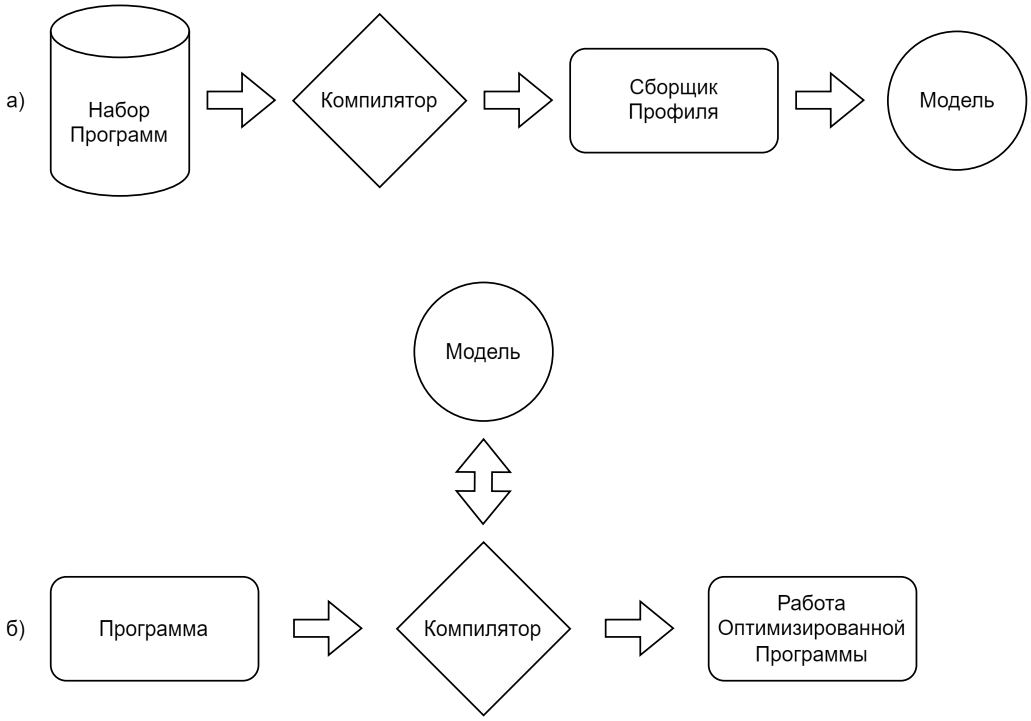

Рисунок 1.8 — Схема компиляции с искусственным профилем

модели, которая заблаговременно проводится разработчиками компилятора, а на рисунке 1.8б изображен рабочий режим, в котором работает компилятор, оказавшись у пользователя.

Глава 2. Методология и окружение

Как упоминалось во Введении, 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.

Во второй главе рассматривается методология поиска неоптимальностей в коде приложений и замера производительности.

В разделе 2.1 дается описание целевой платформы, под которую разрабатывались оптимизации.

В разделе 2.2 описаны два основных пакета приложений, на которых демонстрировались результаты разработанных методов.

В разделе 2.3 излагается методология получения результирующих цифр, основанная на многолетнем научном опыте.

В разделе 2.4 приводятся основные методы изучений целевых приложений для последующей разработки компилятивных алгоритмов. Рассмотрены такие методы как профилирование, симуляция и обратная разработка.

2.1 Целевая платформа

Для проведения исследований был выбран широко распространенный сервер компании Huawei - Kunpeng920. Он базируется на архитектуре ARM V8.2-A [55; 56]. Основным конкурентом данного процессора на архитектуре ARM является Ampere Altra Server [57]. 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 2.1) и одного кристалл интерфейса (SICL - Рисунок 2.2). 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 канальный суперскалярный модуль с возможностью изменения порядка исполнения (superscalar, out-of-order).

Интересной особенностью исследуемого процессора является широкая кэш-линия третьего уровня. Она составляет 128 байт, что в два раза превосходит общепринятое на рынке значение в 64 байта.

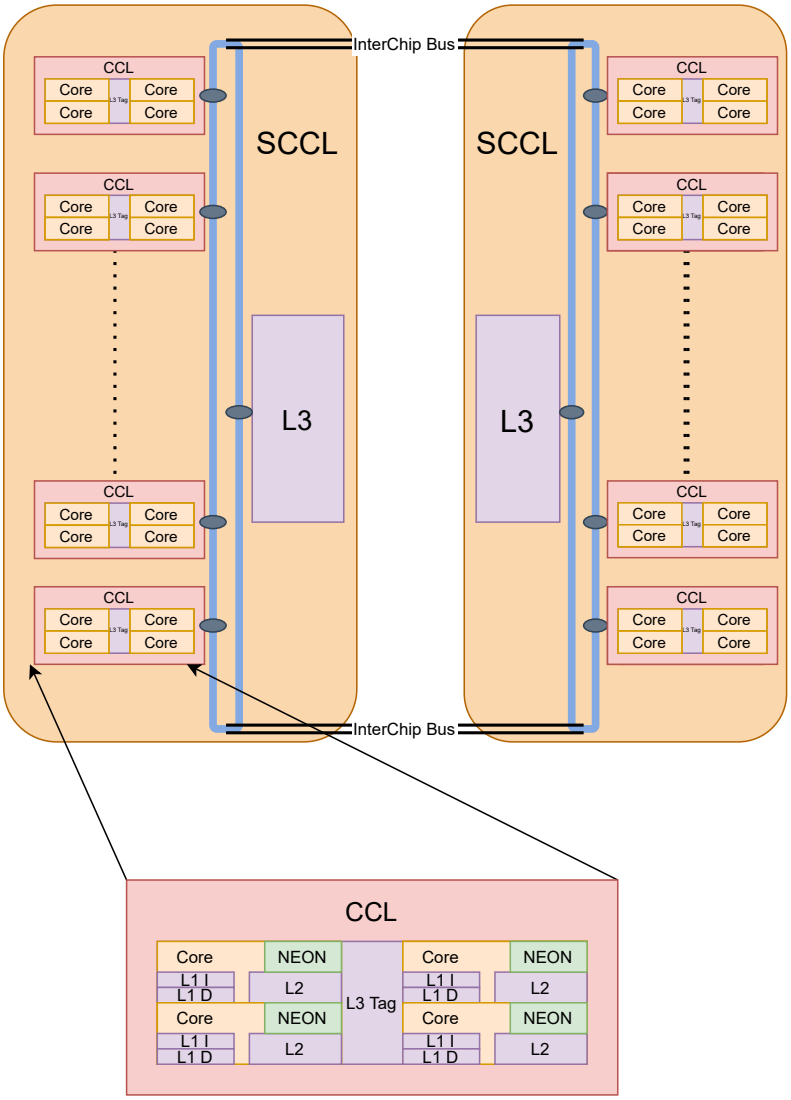

Рисунок 2.1 — Схема целевого чипа

Поддерживаемые расширения:

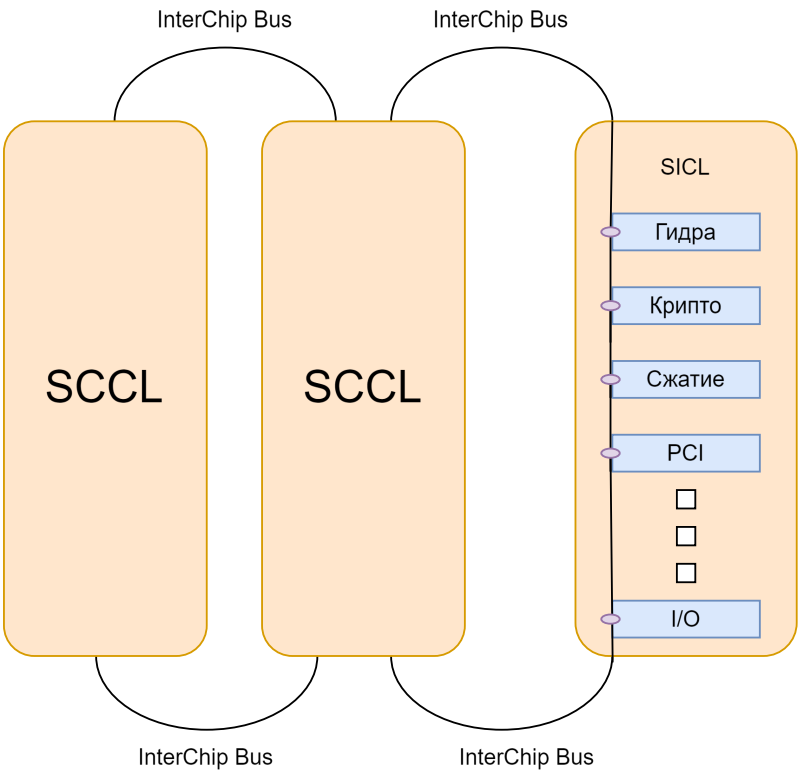

Рисунок 2.2 — SICL модуль

– **NEON** - Векторное расширение ARMv8
– **CRC32** - Расширение для быстрого подсчета чек-суммы CRC32
– **Crypto** - Криптография
– **FP16** - Числа с плавающей точкой половинной точности
– **RAS** - Надежность, доступность и удобство обслуживания. (Reliability, Availability, and Serviceability)

2.2 Тесты производительности

2.2.1 SpecCPU 2017

В проведенном исследовании использовались два набора тестов: "SpecCPU 2017" [15] и CPUBench [16].

"SpecCPU 2017" - набор тестов для оценки производительности вычислительных систем. Существует два поднабора: целочисленный и набор тестов с плавающей арифметикой. 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 [15].

Набор приложений, входящих в пакет "SpecCPU int 2017":

- **perlbench**: Интерпретатор языка Perl, из которого было удалено большинство особенностей, связанных с операционными системами. Включает в себя набор тестов, которые измеряют время выполнения различных операций в Perl [58].
- **gcc**: Известный компилятор языков C/C++/Fortran из коллекции компиляторов GNU [17].
- **mcf**: Моделирует задачу коммивояжера, где необходимо найти оптимальный маршрут для распространения товаров в различных городах, минимизируя расстояние и время пути [59].
- **omnetpp**: OMNeT++ моделирует производительность сети, используя алгоритмы и структуры данных для симуляции различных сценариев, таких как передача данных, маршрутизация и управление трафиком [60].
- **xalancbmk**: Моделирует преобразования XML-документов в HTML- или другие XML-документы с использованием языка XSLT (XSL Transformations) [61].
- **x264**: Свободная и открытая библиотека для кодирования видео, которая обеспечивает высококачественное и быстрое кодирование видео в формате H.264 (MPEG-4 AVC) [62].
- **deepsjeng**: Искусственный интеллект игры в шахматы, имеет больше 2600 ELO [63].
- **leela**: Алгоритм игры в GO, включающий оценку позиции на основе метода Монте-Карло, выборочный поиск по дереву на основе верхних доверительных границ и оценку хода на основе рейтингов ELO [64] .
- **exchange2**: Программа, разработанная для генерации нестандартных и сложных судоку. Использовался на неофициальных соревнованиях, которые могли длиться несколько дней [65].

– **xz**: Содержит компрессионный и декомпрессионный алгоритмы [66].

Набор приложений, входящих в пакет "SpecCPU fp 2017":

– **bwaves**: Численное моделирование взрывных волн. Первоначальная конфигурация задачи состоит из области высокого давления, внутри которой находится небольшая область низкого давления. Сложная интерференционная результирующая картина решается при помощи уравнений Навье-Стокса [67].
– **cactuBSSN**: Моделирование черных дыр и гравитационных волн [68].
– **namd**: Моделирование больших бимолекулярных систем. Почти все время выполнения тратится на расчет межатомных взаимодействий в небольшом наборе функций [69].
– **parest**: Биомедицинская визуализация. Построение трехмерных моделей объектов из нескольких наблюдений на двумерной плоскости (например, МРТ, КТ) [70].
– **povray**: Алгоритм трассировки лучей [71].
– **lbm**: Метод решеточных уравнений Больцмана для моделирования трехмерных моделей несжимаемых жидкостей.
– **wrf**: Модель предсказания погоды, написанная на языке FORTRAN. Содержит огромное количество линейного кода [72].
– **blender**: Рендер трех-мерных моделей [73].
– **cam4**: Модель циркуляции атмосферы [74].
– **imagick**: Производит последовательные манипуляции с двухмерной картинкой (повороты, отражения, размытие и т.д.) [75].
– **nab**: Приложение молекулярного моделирования, выполняющее интенсивные вычисления с плавающей запятой, которые обычно встречаются в области медико-биологических наук [76].
– **fotonik3d**: Вычисляет коэффициент передачи фотонного волновода, используя метод конечных разностей во временной области для уравнений Максвелла [77].
– **roms**: Региональная система моделирования океана. Используется для исследования реакции океана на локальные изменения, такие как ветер или изменение температуры [78].

В наборе тестов SpecCPU существует два основных типа замера - speed и rate. В режиме speed разрешается использовать оптимизации с профилем, опции для каждого теста могут подбираться индивидуально. Однако в данной

диссертации используется режим rate, в котором набор опций для всех тестов должен быть унифицирован и использование профиля запрещено. Запуск возможен как в режиме одной копии - ресурсы машины доступны одному процессу полностью, так и в режиме множественности копий, в котором процессам приходится делить общие ресурсы, такие как шину памяти или кэши. Стоит отметить, что ни в каком из наборов нельзя использовать прямо или косвенно информацию, специфичную для конкретных тестов. Так, например, в 2024 году более 2000 результатов были помечены, как использующие информацию о тестах, а значит нечестные [79].

2.2.2 CPUBench

В 2023 году Китайский институт электроники и стандартизации выпустил новый набор тестов производительности для вычислительных систем [16]. 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.

Целочисленный набор состоит из следующих тестов:

- **x264, gcc, xz**: Схожие с пакетом "SpecCPU int 2017", отличаются наборами входных данных и версиями приложений.
- **gzip**: Архиватор, использующий алгоритм LZMA2 [80].
- **tpcc**: Бенчмарк, который моделирует деятельность розничного дистрибьютора с большим количеством складов и клиентов [81].
- **tpch**: База данных, состоящая из набора бизнес-ориентированных запросов и модификаций данных. Иллюстрируется система принятия решений [82].
- **kmeans**: Java тест, решающий задачу K-ближайших соседей.
- **wordcount**: Java тест, подсчитывающий количество слов в больших файлах.

- **velvet**: Пакет алгоритмов, разработанный для сборки генома и выравнивания секвенирования коротких считываний [83].
- **openssl**: Криптографический инструментарий, реализующий различные алгоритмы шифрования [84].
- **rapidjson**: Библиотека для парсинга Json -файлов [85]
- **python**: Интерпретатор языка Python [86].

Набор тестов с плавающей точкой представлен следующим набором:

- **lightgbm**: Библиотека градиентного бустинга для машинного обучения [87].
- **nektar**: Высокопроизводительный масштабируемый решатель для широкого спектра уравнений в частных производных [88].
- **phenglei**: Программная платформа вычислительной гидродинамики, разработанная Китайским центром исследований и разработок аэродинамики [89].
- **phyml**: Программа анализа белков и генов с помощью алгоритма максимального правдоподобия [90].
- **gromacs**: Пакет используется для моделирования различных биомолекул, имеющих большое количество межатомных связей [91].
- **povray**: Алгоритм трассировки лучей [71].
- **openfoam**: Моделирование задач механики сплошных сред [92].
- **lammps**: Расчеты классической молекулярной динамики, применяется на суперкомпьютерах, имеет высокую степень паралелизации [93].
- **cube**: Гравитационная задача N-тел [94].
- **wrf**: Модель предсказания погоды, написанная на языке FORTRAN. Содержит огромное количество линейного кода [72].

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 присутствую 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.

2.3 Методология измерения

В данной диссертаций используется измерения типа "SpecCPU rate" и ее аналог typcal в CPUBench. В данной методологии предусмотрены следующие шаги:

1. Измерение времени выполнения каждого теста, запущенного в NUM_COPIES копий, в некоторое количество итераций (N)
2. Если было запущено больше одной копии, то временем исполнения теста считается самое большое время среди всех копий.
3. Для каждого теста выбирается медианное время среди проделанных итераций.
4. Обратное медианное время каждого теста умножается на референсное время (REF_TIME), полученное на фиксированной машине. Например, для "SpecCPU 2017" это Sun Fire V490 with 2100 MHz UltraSPARC-IV+.
5. Считается среднее геометрическое по всем тестам. Использование среднего геометрического обосновывается свойством сохранения отношения.
6. Полученное число умножается на количество копий.

Если всего в сете M тестов, то кратко можно записать формулу расчета производительности следующим образом:

$$RATE = \left(\prod_{i=1}^{M} \frac{REF\_TIME_i}{MEDIAN(TIME_{i1}, TIME_{i2}, ..., TIME_{iN})} \right)^{\frac{1}{M}} \tag{2.1}$$

Также важной частью считается настройка окружающей системы, направленная на уменьшение флуктуаций времени исполнения тестов и лучшей утилизации тестовой системы. Эта тема не является прямой темой данного исследования, а лишь косвенного затрагивает ее, но тем не менее весьма важна,поэтому ниже предлагается ознакомится с проблемами, которыми

пришлось столкнуться во время проведения замеров для алгоритмов, реализованных в данной диссертации.

– **Троттлинг**. Технология изменения частоты процессора при превышении критических температур. Часто возникает при нарушении системы охлаждения или неправильной эксплуатации. Проявляется в виде увеличения времени исполнения теста с каждой следующей итерацией [95].

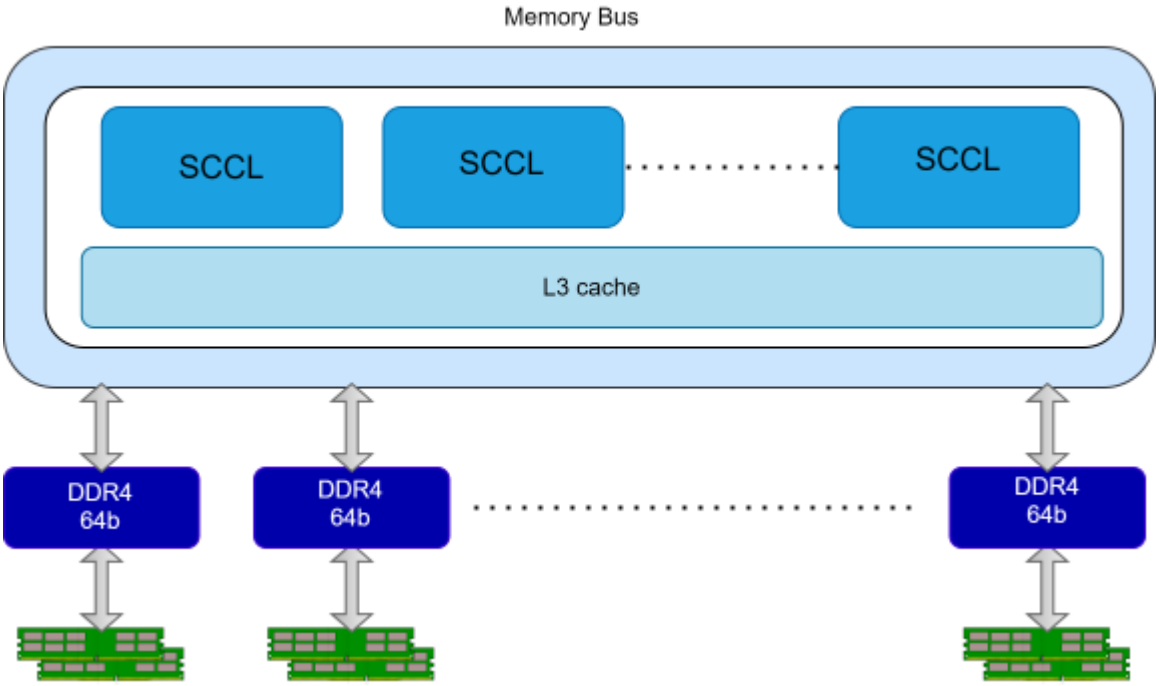

Рисунок 2.3 — Схема подсистемы памяти целевой платформы

– **Неполная утилизация ресурсов системы**. Целевая платформа имеет восемь каналов DRAM, с возможностью подключения до двух плашек на каждый канал (Рисунок 2.3). В случае, если какой-то канал остался незадействованным (Например, вставлены подряд, а не через одну) то будет наблюдаться картина, как на рисунке 2.4. Можно видеть, что неиспользуемые порты приводят к тому, что вычислительным ядрам приходится обращаться за ресурсами памяти в соседние блоки, что значительно замедляет исполнение.

– **Рандомизация размещения адресного пространства (ASLR)**. Технология, изначально разрабатываемая для защиты процессов от различного рода атак с использованием переполнения буфера. При этом адреса расположения исполняемого файла рандомизируются для усложнения предсказания положения объекта в памяти и получения доступа из сторонних процессов [96]. Однако, в случае

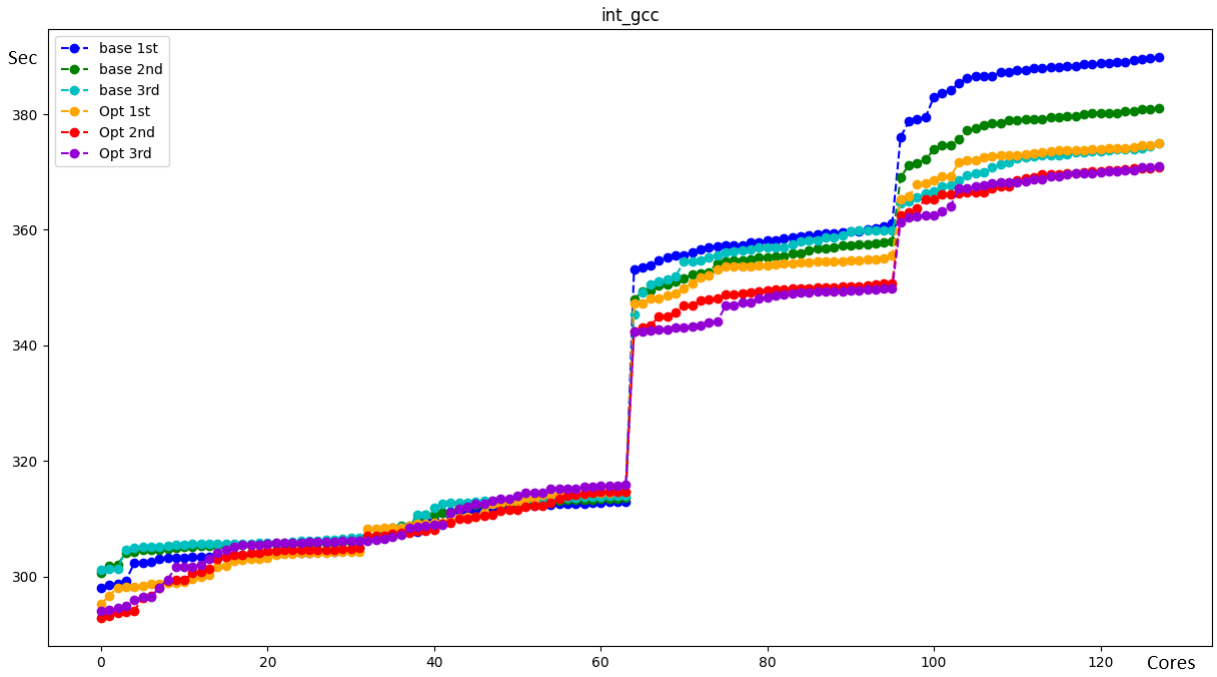

Рисунок 2.4 — Зависимость времени исполнения приложения от номера ядра при неправильном подключении плашек оперативной памяти

аккуратных измерений производительности системы эта технология может приводить к случайному наложению младших частей адресов и, соответственно, попаданию в одну и ту же кэш линию. На рисунке 2.5 можно видеть как на случайном ядре время исполнения отдельной копии увеличивается больше, чем в 2 раза.

– **Частота обновления оперативной памяти** - Оперативная память, являясь энергозависимой памятью, требует постоянного обновления значений в ячейках. Частота обновления этих ячеек регулируется в BIOS. С одной стороны, большая частота обновления уменьшает количество ошибок и связанных с ними задержек, однако с другой стороны, обновление памяти это по своей сути дополнительная загрузка данных, что в высоко нагруженной системе может создавать дополнительные задержки. Поэтому этот параметр приходится подбирать эмпирическим путем, и он оказывает существенное влияние на итоговую производительность системы.
– **Занятость ресурсов внешними программами**. Достаточно очевидный факт: если необходимо измерить производительность системы, то ваш бенчамрк должен быть единственным, исполняемым на этой системе. Однако, организовать тест в изоляции достаточно сложно, так как операционная система время от времени может

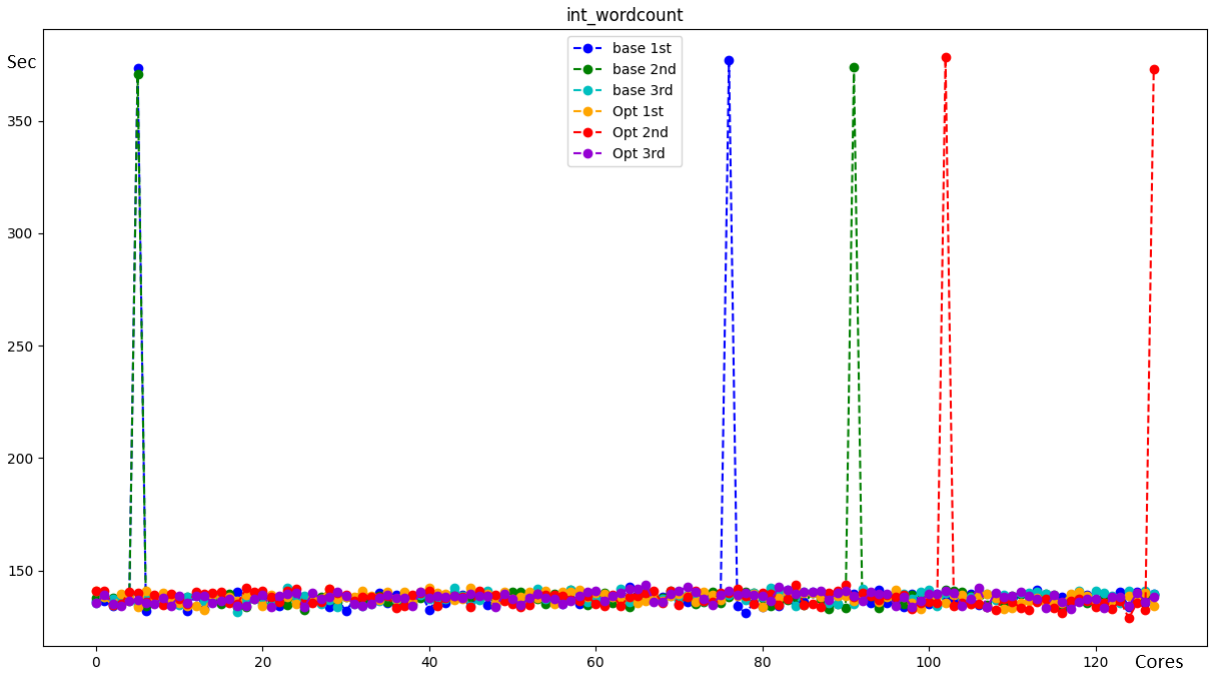

Рисунок 2.5 — Зависимость времени исполнения приложения от номера ядра при замере с включенной ASLR

запускать демонов, планировщик и прочее, методы борьбы с этим индивидуальны и не будут указаны здесь, однако, покажем, как пронаблюдать этот эффект. Если мы производим высокоинтенсивной замер, утилизирующий все ядра и большинство остальных ресурсов системы, а затем отсортируем время выполнения тестов на разных ядрах, то можем получить картину, как на рисунке 2.6. Эти "хвосты" чаще всего означают, что система была занята другими приложениями, а не тестом.

Приведенные проблемы нельзя решить после замера каким-либо "обрезанием хвостов" или "удалением пиков" из выборки. Методология такого не позволяет. Если бы подобные манипуляции были возможны, то это позволило бы вендорам манипулировать данными, ведь тогда пришлось бы вводить какие-либо правила, когда и как можно корректировать данные замеров, что привело бы к махинациям ради получения более высоких цифр производительности.

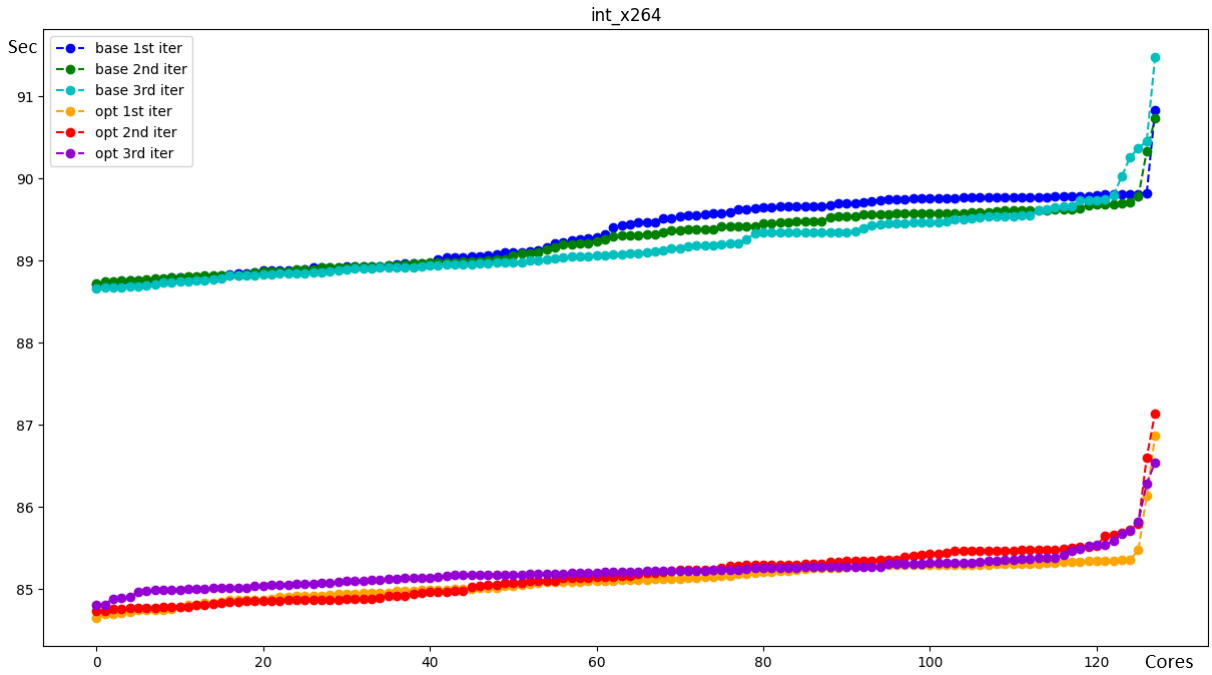

Рисунок 2.6 — Зависимость времени исполнения приложения от номера ядра при замере на загруженной системе

2.4 Выявление возможностей для оптимизации

Данное исследование подразумевает выявление оптимизационных возможностей тестовых приложений для последующей разработки оптимизаций. К сожалению, очень сложно предоставить определенный алгоритм поиска оптимизационных возможностей, так как разные программы оказывают нагрузку разного типа на систему, подход к каждому случаю практически индивидуален, и должен учитывать аппаратные возможности, архитектуру команд и структуру исходной программы. Однако существуют достаточно понятные методологии поиска горячих (высоконагруженных) участков. Рассмотрим методы, которые использовались в данной диссертационной работе.

2.4.1 Профилирование

Профилирование программы - это методика динамического анализа программы, которая способна измерить количество вызовов функций,

загруженность памяти, использование определенных инструкций, временную сложность программы.

Программы профилирования приложений в зависимости от способа сбора информации можно глобально разделить на три типа:

Первый тип профилирующих программ базируется на событиях, которые происходят внутри программы. Для этого в код самого приложения или в код подгружаемых библиотек вставляются дополнительные обработчики на этапе компиляции или линковки приложения. В процессе исполнения программа непосредственно перед заранее определенным программным событием (вызов функции, аллокация памяти, создание объекта и пр.) вызывается встроенным метод-обработчик, который собирает и агрегирует полезную профильную информацию. Классическим примером такого профилировщика является **gprof** [97]. При таком способе внутрипрограмные события собирается с высокой точностью, однако постоянный запуск обработчиков может искажать перфомансную картину приложения (перебивать данные в кэшах и влиять на логику предсказателей). Огромным минусом данной модели является необходимость перекомпиляции программы.

Другой подход - интерпретирующие профилировщики, по своей сути являющиеся бинарными трансляторами/интерпретаторами с возможностью выставления любых программных счетчиков непосредственно в код исполняемой программы во время исполнения [98; 99]. Являются мощнейшими инструментами профилирования, однако оказывают сильное влияние на время исполнение программы, замедляя ее в десятки или даже сотни раз. Тем не менее такой подход не требует перекомпиляции самого приложения и обладает высокой точностью сбора внутрипрограммных событий.

Третий подход и один из основных методов выявления оптимизационных возможностей в рамках данной диссертационной работы - сэмплирующий профилировщик **perf** [100]. Он способен проверять стек вызовов программы через регулярные промежутки времени, благодаря интерфейсу прерываний операционной системы. Профили, собранные таким образом, обычно менее точны и не очень конкретны (могут пропускать вызовы функций), однако позволяют измеряемой программе работать практически на полной скорости. Относительная точность достигается путем агрегации большого количества данных. Такой тип профилирования наиболее удобен для больших приложений, используемых пользователями.

Важной особенностью сборки профильной информации является поддержка аппаратных счетчиков в архитектуре ARM64, реализованная в виде PMU (Performance Monitoring Unit). PMU это специализированный блок процессора, предназначенный для отслеживания различных событий, которые происходят на уровне микроархитектуры [101]. PMU может быть управляем через специальные инструкции, такие как MCR (Move to Coprocessor from Register) и MRC (Move to Register from Coprocessor), позволяющие записывать и считывать данные из блока-монитора. Для контроля этого устройства существует 32 программируемых счетчика, которые могут быть настроены для отслеживания различных событий. Каждый счетчик может быть настроен для отслеживания конкретного события. В качестве примера событий, которые могут быть собраны благодаря PMU, можно привести:

- Промахи в L1/L2/L3 кэш.
- Задержки в очереди на исполнение.
- Задержки, связанные с "голодом" исполняющих устройств (нет доступных для исполнения инструкций).
- Ошибки предсказания переходов.
- Задержки фронт-енда аппаратуры.
- Количество векторных инструкций.
- Количество инструкций с плавающей точкой. (метрика недоступна на исследуемой машине)

Такой метод позволяет очень сильно сузить круг поиска возможных оптимизаций. Так, например, в тестах xz и gzip наблюдалось значительное количество задержек в память, что в последующем послужило мотивацией к разработке дополнительных алгоритмов программной предподкачки данных. В тестах openssl, tpcc, tpch, python наблюдалось высокое количество задержек, связанных с фронт-ендом аппаратуры, частые ошибки предсказания переходов. В последующем этот анализ дал толчок к разработке оптимизации свертки условных переходов. Малое количество векторизованного кода в x264 привело к улучшению алгоритма векторизации.

2.4.2 Симуляция

Иногда общих соображений из главы 2.4.1 недостаточно для определения причин замедления программы. В таких случаях можно произвести исследование на потактововом симуляторе данной машины или близкой к ней по конфигурации и временным параметрам, при отсутствии точной модели. В качестве примера приведем пример анализа из оригинальной работы, посвященной оптимизации инструкций широкого доступа в память [102], позволившее реализовать оптимизацию 3.4.

В качестве потактовой модели был использован симулятор GEM5, содержащий детальную модель микроархитектуры процессора Alpha 21264 [103; 104]. Конфигурация симулируемой системы приведена в таблице 1. Важно подчеркнуть, что моделирование осуществляется в режиме эмуляции системных вызовов (system emulation), в котором отсутствует исполнение кода ядра операционной системы.

Таблица 1 — Настройка симулируемой модели

Архитектура	ARMv8.2-A
Модель процессора	ArmO3CPU
Кэш-память L1	64 кБ кэш данных, 64 кБ кэш инструкций
Кэш-память L2	512 кБ, общий
Ширина этапов конвейера	4 для всех этапов
Количество LSU блоков	2
Оперативная память	DDR4 2400 МГц 512 МБ

На рисунке 2.7 приводится код, содержащий инструкции записи (STR) и широкого чтения (LDP) с одинаковым базовым регистром. В левой части рисунка располагается временная шкала. Каждая точка соответствует одному циклу центрального процессора, там же указаны стадии конвейера:

1. f - fetch - чтение инструкции из ячейки памяти
2. d - decode - разбор инструкции и ее аргументов

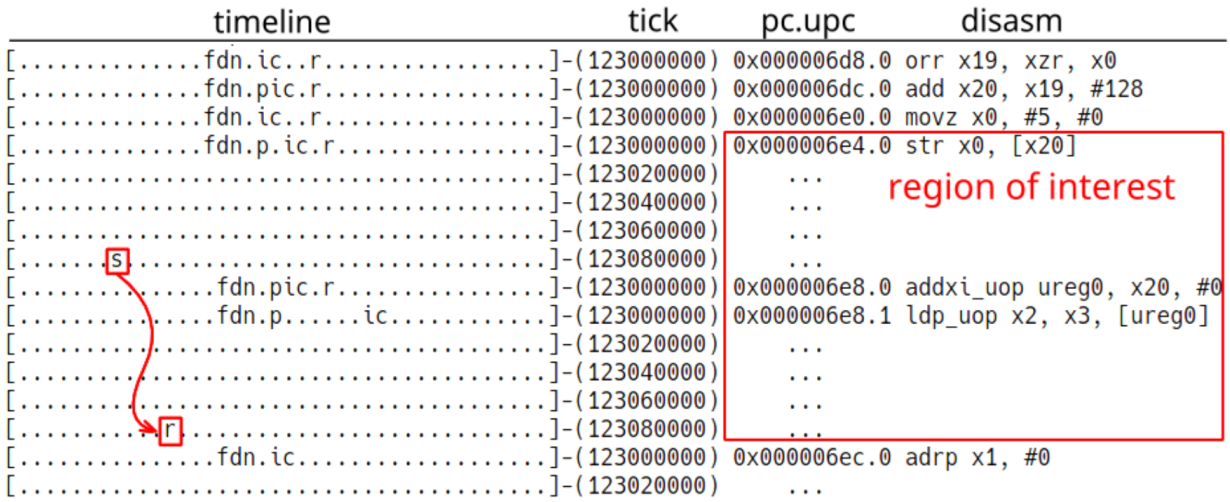

Рисунок 2.7 — Моделирование исполнения широкой инструкции чтения после записи

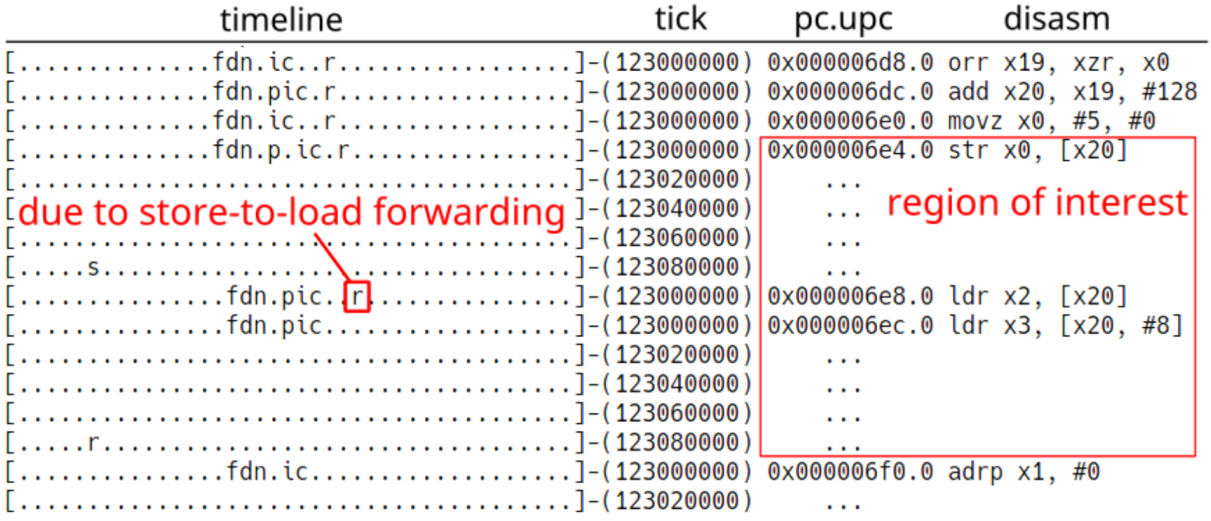

Рисунок 2.8 — Моделирование исполнения двух инструкций, полученных в результате разбиения широкой инструкции чтения памяти

3. n - rename - переименование регистров (маппинг на физические)
4. p - dispatch - отправка инструкции в бек-енд (back-end) процессора
5. i - issue - назначение конкретного вычислительного устройства
6. c - complete - окончание исполнения
7. r - retire - возвращение результата операции пользователю
8. s - store-complete - завершение операции записи в память

В рассматриваемом сценарии процессором выполняется чтение, реализованное через инструкцию LDP, 16 байт данных, но перед чтением с помощью инструкции STR обновляются 8 байт в том же диапазоне памяти. Возникающий конфликт называется зависимостью "чтение после записи".

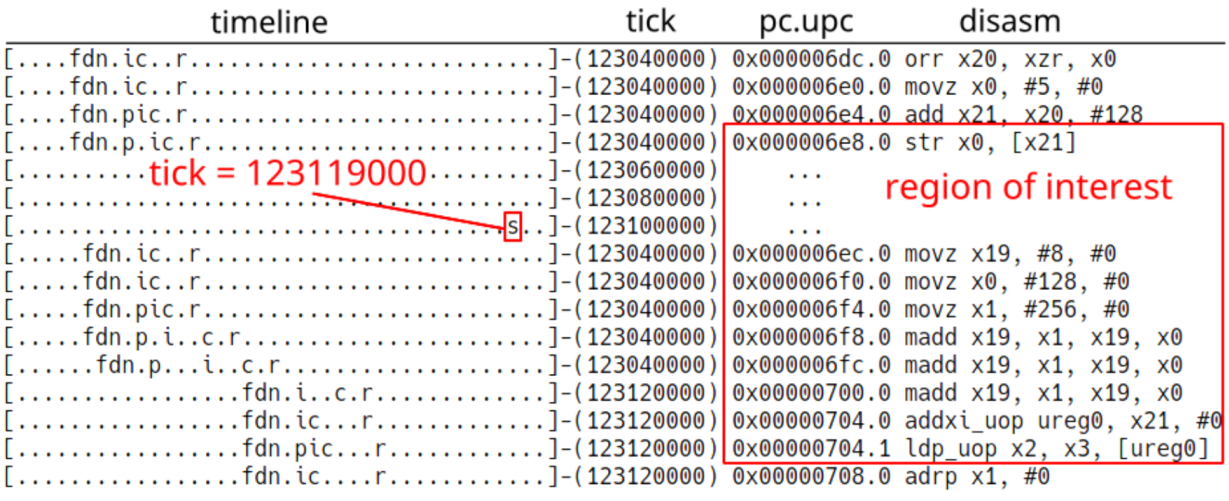

Рисунок 2.9 — Моделирование исполнения арифметических операций перед широкой инструкцией доступа в память

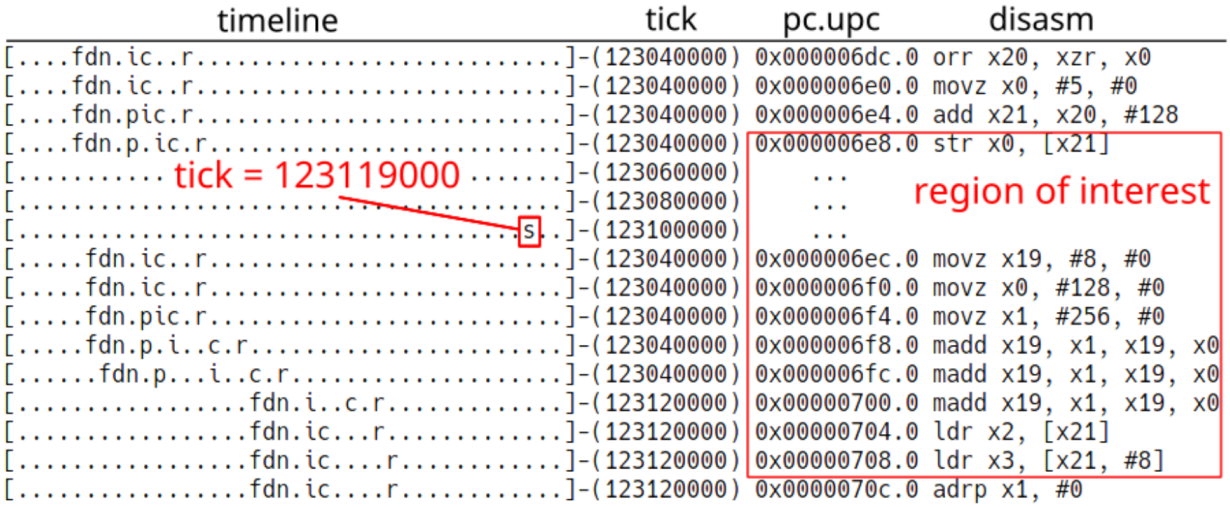

Рисунок 2.10 — Моделирование исполнения арифметических операций перед двумя инструкциями, полученными в результате разбиения широкой инструкции чтения

Видно, что разбитая на микрооперации инструкция LDP переходит на этап r (retire) только спустя 4 такта после завершения записи в память.

На рисунке 2.8 проиллюстрирована работа аппаратного механизма разрешения зависимостей при спекулятивном выполнении операций доступа в память. Видно, что задержки не происходит.

Между рассматриваемыми инструкциями чтения и записи процессором могут выполняться прочие операции, и к моменту начала исполнения операции чтения запись в память полностью завершится. Моделирование такой ситуации для двух ранее рассмотренных случаев продемонстрировано на рисунках 2.9 и 2.10. В обоих случаях на чтение пары значений тратится одинаковое время,

так как в очереди на запись уже нет данных, которые можно взять для первой инструкции чтения. Следовательно, разделение инструкции широкого доступа не всегда целесообразно и требует анализа. В общем случае максимальное число инструкций между двумя конфликтующими операциями доступа в память, которое далее будем называть дистанцией, зависит от конфигурации конкретного процессора и времени выполнения инструкций различного типа, поэтому предлагается определять это значение эмпирическим путем.

2.4.3 Обратная разработка

Чаще всего под обратной разработкой понимают исследование некоторого готового продукта с целью изучения его свойств. Конечно же в проделанной работе тоже имелся такой этап. Для целевого оптимизатора конкурентами являются такие компиляторы как LLVM Clang [105] и intel DPC++ [106]. Обратная разработка позволяет провести быстрый визуальный анализ сгенерированного кода и, что самое важное, модифицировать его без необходимости перекомпиляции. Среди существующих популярных инструментов дизассемблера (IDA, Radare2, Ghidra и др. [107—109]) был выбран radare2. Его преимуществами являются четкое описание установки на своей странице Github, наличие обширной документации и поддержки сообщества на своей странице Wikipedia и своей официальной электронной книге. Пример выводимой информации приложением radare2 можно увидеть на рисунке 2.11.

2.5 Выводы по главе

Во второй главе приводится описание целевой архитектуры, методы исследования приложений и замера цифр производительности.

Рассмотрены проблемы, связанные с процессом измерения цифр производительности, описаны их архитектурные причины.

На примерах исследуемых приложений показано использование программного обеспечения для анализа тестов на целевой архитектуре.

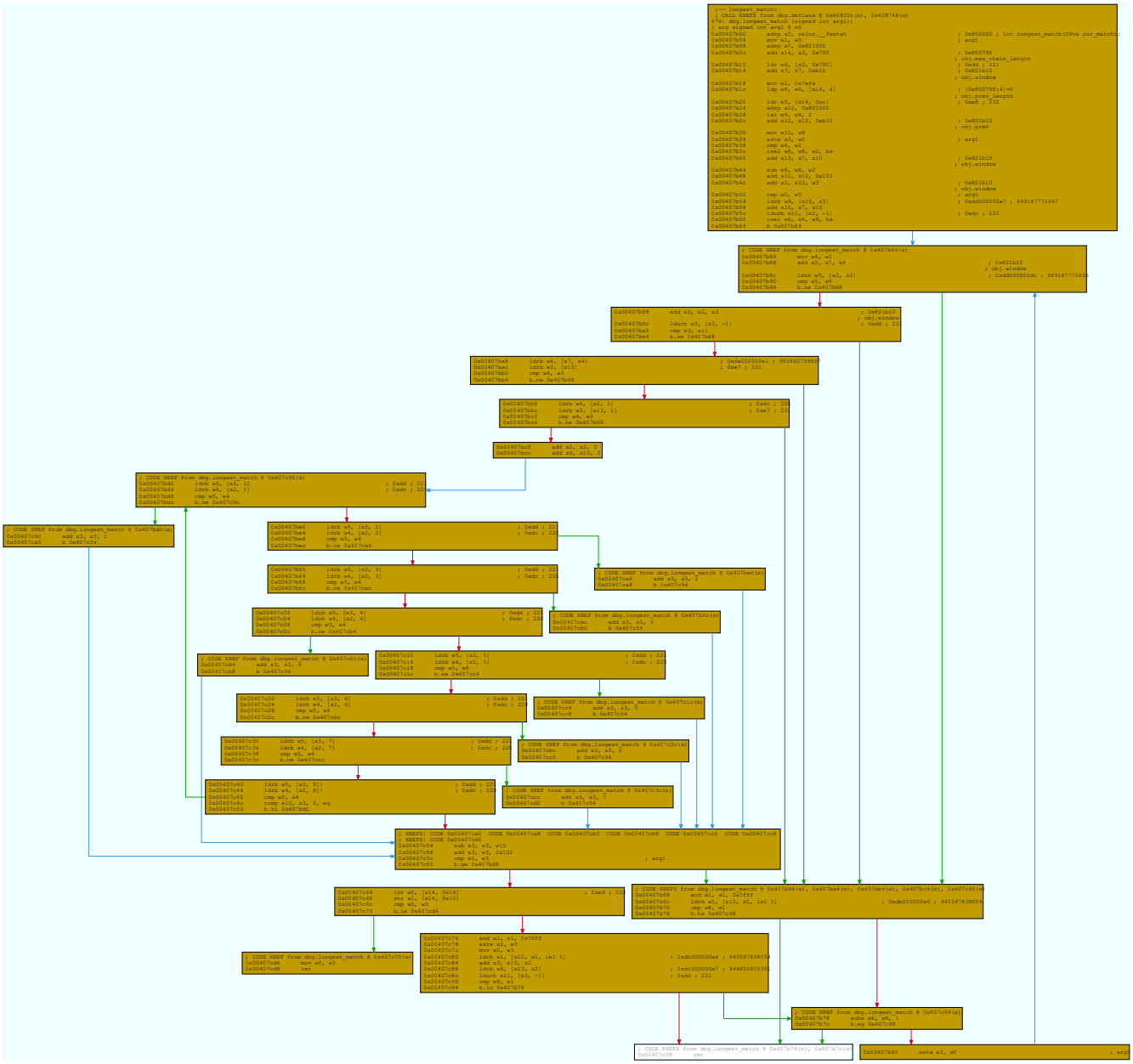

Рисунок 2.11 — Базовые блоки горячего участка кода приложения gzip, полученные с помощью radare2

Глава 3. Разработка оптимизаций

В этой главе описаны разработанные оптимизации. 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.

3.1 Улучшение существующих оптимизаций

Хотелось бы начать описание проделанной работы с улучшений уже существующих оптимизаций. Компилятор GCC разрабатывается с 1987 года. Сотни разработчиков и исследователей привносили свои улучшения все это время, тем не менее в процессе данной работы с учетом специфики целевой машины удалось найти определенное количество недостатков даже в существующих алгоритмах.

3.1.1 Преобразование условных переходов

Преобразование условных переходов (If-conversion) — это хорошо известный метод оптимизации, который 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. В результате целевой код содержит меньшее количество инструкций перехода, что снижает нагрузку на аппаратный предсказатель переходов, однако такой подход позволяет увеличить количество избыточных инструкций во время исполнения [110; 111]. Обычно эта оптимизация основана на представлении SSA, однако в компиляторе GCC

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

используется другой подход. На этом этапе SSA форма отсутствует, что может привести к следующей проблеме (рисунок 3.1): если в одной из ветвей исполнения в качестве регистра назначения используется тот же регистр (reg1 на Рисунке 3.1), что в другом в качестве источника, то после слияния будет создано неправильное определение reg1.

Предлагаемое решение содержит принудительное переименование регистров. Такая трансформация была добавлена при коллизии такого типа. Регистры коллизий определяются как:

$$rename\_candidates = DEFS_{left\_bb} \cap USES_{right\_bb} \tag{3.1}$$

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

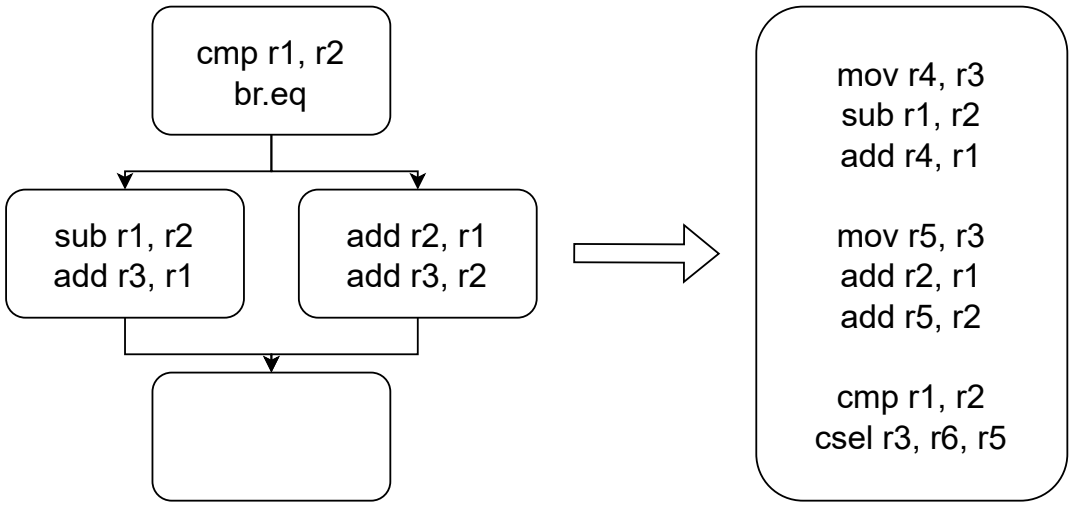

Рисунок 3.1 — Пример некорректного преобразования условных переходов в следствие отсутствия SSA формы

Улучшение преобразования условных переходов в компиляторе GCC было размещено под опцией **-fifcvt-allow-register-renaming**. Такой подход помогает уменьшить количество "хвостовых" базовых блоков в целевых тестах. (Листинг 3.1). Такие базовые блоки были обнаружены путем применения технологии профилирования из главы 2.4.1. Сбор событий при помощи PMU показывал повышенное количество задержек во фортн-энде (front-end) процессора. Причиной этих задержек оказались постоянный переходы в "хвостовые" базовые блоки и обратно.

Конечно же применение подобной трансформации сопряжено с определенными рисками уменьшения производительности. Повсеместное преобразование может привести к увеличенному давлению на регистровый

Листинг 3.1 Пример "хвостовых" базовых блоков, которые будут оптимизированы предложенным улучшением преобразования условных переходов

```
4145bc:   ret
4145c0:   mov     x5, #0x100000000
4145c4:   add     x7, x7, x5
4145c8:   b       4131d0
4145cc:   mov     x2, #0x100000000
4145d0:   add     x6, x6, x2
4145d4:   b       4145a0
4145d8:   mov     x3, #0x100000000
4145dc:   add     x11, x11, x3
4145e0:   b       41454c
4145e4:   mov     x3, #0x100000000
4145e8:   add     x11, x11, x3
4145ec:   b       4144fc
```

файл, что в свою очередь приводит к излишнему использованию стека. 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.

Листинг 3.2 Образец проверки из теста povray (трассировка лучей)

```
if (lf == 0.0 || lf * f < 0)
{
        changes++;
}
```

В качестве примера дополнительных ограничений рассмотрим оптимизацию участка кода из теста povray (трассировка лучей). В одном из основных горячих циклов можно встретить следующую проверку, изображенную на листинге 3.2. В GCC этот код трансформируется в следующий набор базовых блоков (Листниг 3.3) Операция умножения считается достаточно

Листинг 3.3 Листинг 3.2 в представлении GIMPLE GCC

```
<bb 9> [local count: 118111600]:
if (lf_11 == 0.0)
goto <bb 11>; [50.00%]
else
goto <bb 10>; [50.00%]

<bb 10> [local count: 59055800]:
_5 = lf_11 * val_42;
if (_5 < 0.0)
goto <bb 11>; [41.00%]
else
goto <bb 12>; [59.00%]

<bb 11> [local count: 83268678]:
changes_22 = changes_10 + 1;

<bb 12> [local count: 118111600]:
# changes_9 = PHI <changes_10(10), changes_22(11)>
```

тяжелой, поэтому преобразование ее в один базовый блок вместе с простым сравнением с нулем кажется компилятору неэффективным, однако было обнаружено, что для конкретной исследуемой платформы такое преобразование все же увеличивает итоговую производительность программы.

Оптимизация реализована в проходе ifcombine на этапе Gimple. Следующее преобразование было введено для оптимизации целевого кода и оно основано на том факте, что некоторые инструкции быстрее (или дешевле с точки зрения времени), чем условные переходы (рисунок 3.2). Это означает, что объединение двух сравнений в рамках операции AND или OR в один базовый блок, чтобы не выполнять "ленивое" вычисление булевого оператора, является более эффективным. Реализация gcc по умолчанию объединяет текущее сравнение со следующим, только если оно является условным. Эта оптимизация определяет список дешевых gimple-assign-insns, которые также могут быть объединены в один базовый блок в этом проходе.

Введение дешевых инструкций и добавление туда умножения чисел с плавающей точкой позволяет в примере из листинга 3.3 объединить базовый блок, содержащий сравнение float var с 0.0, с блоком с дешевой инструкцией

Листинг 3.4 Листинг 3.3 в представлении GIMPLE GCC после оптимизации преобразования условных переходов

```
<bb 9> [local count: 118111600]:
_5 = lf_11 * val_42;
_36 = _5 < 0.0;
_51 = lf_11 == 0.0;
_18 = _36 | _51;
if (_18 != 0)
goto <bb 10>; [70.50%]
else
goto <bb 11>; [29.50%]

<bb 10> [local count: 83268678]:
changes_22 = changes_10 + 1;

<bb 11> [local count: 118111600]:
# changes_9 = PHI <changes_10(9), changes_22(10)>
```

умножения с плавающей точкой.Таким образом, один условный оператор goto и один базовый блок будут удалены. После этого преобразования оптимизированный фрагмент будет выглядеть следующим образом во внутреннем представлении компилятора GCC (Листинг 3.4).

3.1.2 Векторизация циклов с небольшим числом итераций

Векторизация — это известный метод, использующий параллелизм данных [112]. 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 (VEC_FACTOR). Опять же, чтобы это увидеть пришлось воспользоваться техникой сэмплирующего профилирования из главы 2.4.1, в ходе которой было обнаружено, что несмотря на включенную векторизацию, приложение все равно большую часть времени исполняла скалярную версию участка. Следовательно,

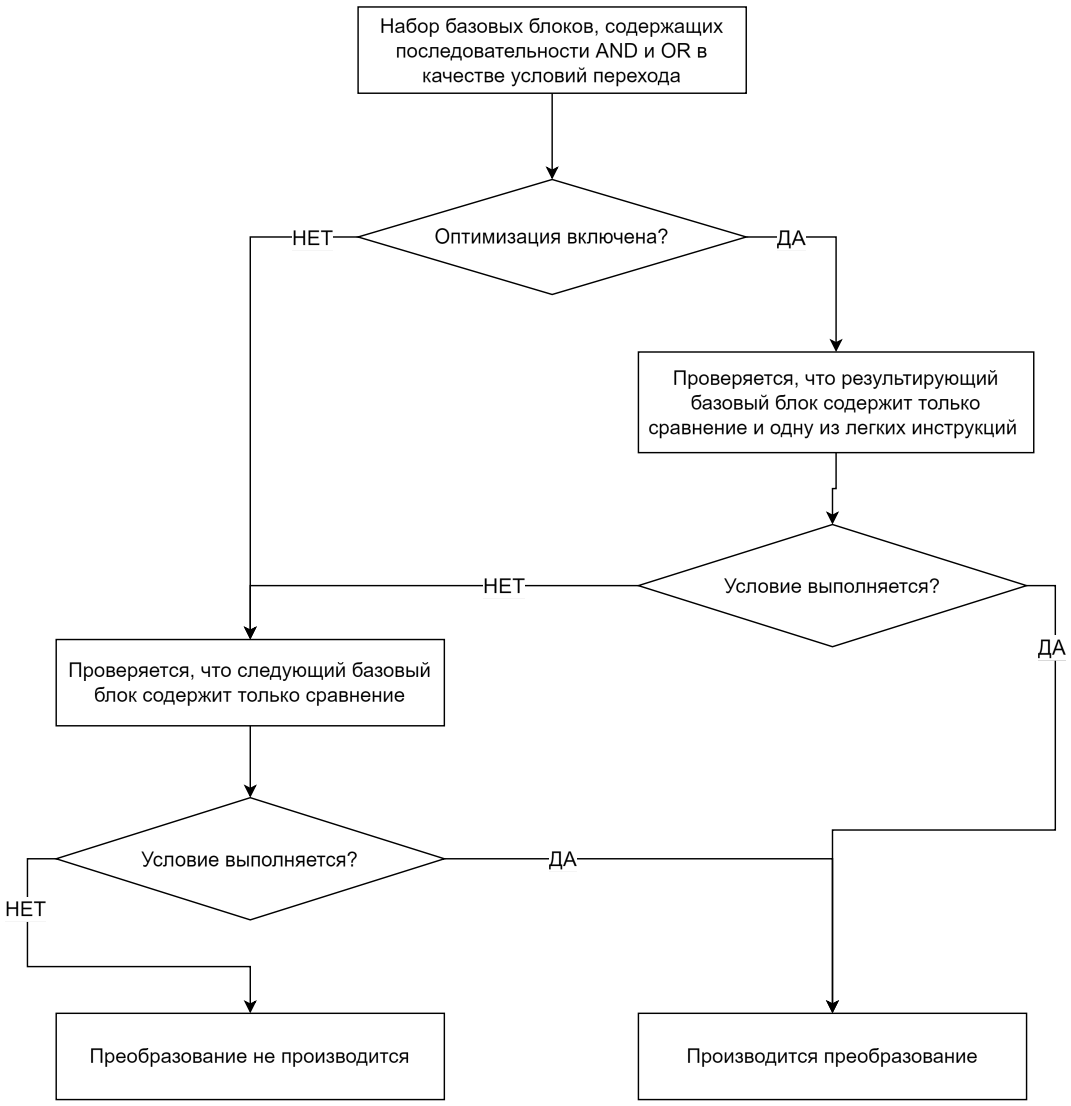

Рисунок 3.2 — Схема улучшения стоимостной эвристики в оптимизации преобразования условных переходов

когда VEC_FACTOR равен, например 8, код будет сгенерирован для $VEC\_FACTOR = 4$ и $VEC\_FACTOR = 2$. Если во время выполнения фактическое количество итераций будет только 4, то будет выбран скалярный вариант. Это небольшое и простое улучшение меняет условие пересечения указателя в заголовке цикла [111].

Рассмотрим простой цикл (Листинг 3.5)

Листинг 3.5 Простой цикл рассматриваемый оптимизацией векторизации

```
\\ a,b,c: any arrays with size N
for (i = 0; i<N; i+=1)
    c[i] = a[i] *b[i]
```

GCC преобразует этот цикл в (Листинг 3.6), и легко видеть, что если, например, $c - a = VEC\_FACTOR/2$ и $N = VEC\_FACTOR/2$, то будет выбрана скалярная версия.

Листинг 3.6 Цикл (Листинг 3.5) после векторизации GCC

```
\\ a,b,c: any arrays with size N
VEC_FACTOR: factor estimated by GCC pass
if  (abs (c - a) < VEC_FACTOR  ||
     abs (c - b) < VEC_FACTOR)
    goto SCALAR;

if (N < VEC_FACTOR)
    goto TAIL;

for (i;i<N;i+=VEC_FACTOR)
    WIDE_C = WIDE_A * WIDE_B
    \\ WIDE_X has VEC_FACTOR size

TAIL:
if (i<N - VEC_FACTOR/2)
    goto SCALAR;

SEMIWIDE_C = SEMIWIDE_A * SEMIWIDE_B
\\ SEMIWIDE has VEC_FACTOR/2 size

SCALAR:
for (i = 0; i<N; i+=1)
    c[i] = a[i] *b[i]
```

Листинг 3.7 Модифицированная проверка для (Листинг 3.6)

```
\\ a,b,c: any arrays with size N
\\ VEC_FACTOR: factor estimated by GCC pass
if (abs (c - a) < min(VEC_FACTOR,N) ||
    abs (c - b) < min(VEC_FACTOR,N))
   goto SCALAR;
```

Чтобы это исправить, предлагается простое решение: заменить строки 1 и 2 в векторизованной версии (Листинг 3.6) следующим кодом (Листинг 3.7). Добавлен параметр **–param=vect-alias-flexible-segment-len**. Эта оптимизация повышает производительность приложения x264.

3.1.3 Векторизация линейного кода

Векторизация линейного кода (SLP - superword level parallelism) [113; 114] является одной из известных проблем в области оптимизирующих компиляторов. В процессе исследования было обнаружено, что современный компилятор GCC не может векторизовать код, представленный на листинге Листинге 3.8. Данный горячий участок кода содержит последовательность из четырех групп инструкций: по 2 выгрузки и 2 загрузки в память.

Листинг 3.8 Пример кода для векторизации из теста cube

```
rho_f[idx1[2]][idx1[1]][idx1[0]] +=
        dx1[0] * dx1[1] * dx1[2] * sim.mass_p_cdm;
rho_f[idx1[2]][idx1[1]][idx2[0]] +=
        dx2[0] * dx1[1] * dx1[2] * sim.mass_p_cdm;
rho_f[idx1[2]][idx2[1]][idx1[0]] +=
        dx1[0] * dx2[1] * dx1[2] * sim.mass_p_cdm;
rho_f[idx1[2]][idx2[1]][idx2[0]] +=
        dx2[0] * dx2[1] * dx1[2] * sim.mass_p_cdm;
rho_f[idx2[2]][idx1[1]][idx1[0]] +=
        dx1[0] * dx1[1] * dx2[2] * sim.mass_p_cdm;
rho_f[idx2[2]][idx1[1]][idx2[0]] +=
        dx2[0] * dx1[1] * dx2[2] * sim.mass_p_cdm;
rho_f[idx2[2]][idx2[1]][idx1[0]] +=
        dx1[0] * dx2[1] * dx2[2] * sim.mass_p_cdm;
rho_f[idx2[2]][idx2[1]][idx2[0]] +=
        dx2[0] * dx2[1] * dx2[2] * sim.mass_p_cdm;
```

В качестве улучшения оптимизации векторизации линейного участка кода были предложены следующие шаги:

– **Группировка инструкций**: Векторизация поддерживает группировку инструкций, адресующих не непрерывный участок памяти. В данном случае проход пытался собрать группы по 8 инструкций, после чего не мог векторизовать сложный набор. Было решено ограничить размер группы размером реального векторного регистра на целевой машине. Такое небольшое изменение позволило существенно упростить дальнейший анализ прохода, так как теперь в горячем участке формируются четыре группы, по 2 загрузки и выгрузки в каждой.
– **Анализ пересечения адресов**: Доступы в память не могут быть векторизованы, если не доказано, что они адресуют непересекающиеся ячейки памяти. Текущая реализация GCC обращает внимание только на базовые адреса инструкций. Например, загрузки

$$LDR \;\; R_1, R_b, 1 \tag{3.2}$$

$$LDR \;\; R_2, R_b, 2 \tag{3.3}$$

считаются непересекающимися, так как имеют общую базу и разное смещение. Однако, если вторую инструкцию заменить на последовательность

$$ADD \;\; R_3, R_b, 1 \tag{3.4}$$

$$LDR \;\; R_2, R_3, 1 \tag{3.5}$$

то анализатор видит различные базовые регистры и не может определить пересечение. Данная проблема была решена добавлением аффинного анализа выражений в проход. С его помощью R_3 раскрывается как $1 * R_b + 1$, и , если вычесть из этого многочлена R_b, то результатом будет константа.
– **Перестановка инструкций**: Наивная группировка инструкций приводит к тому, что группа выражений из листинга 3.9 формирует два вектора $\{\_35, \_33\}$ и $\{\_147, \_35\}$, однако в таком случае чаще лучше бы было сгруппировать их иначе: $\{\_35, \_35\}$ и $\{\_147, \_35\}$. Новая группировка позволит использовать аппаратное умножение вектора на скаляр, что ускоряет итоговые вычисления, за счет экономии процесса сборки лишнего вектора.

Все эти изменения позволили векторизовать исходный участок из целевого теста cube (листинг 3.8), а также улучшить векторизацию линейных участков в компиляторе GCC.

Листинг 3.9 Выражения без перестановок

```
stmt 0: _61 = _35* _147
stmt 1: _66 = _33 * _35
```

3.2 Шаблонные оптимизации

Данный раздел поднимает вопрос шаблонных оптимизаций. Несмотря на свою кажущуюся простоту, этот тип оптимизаций позволяет поднять две важные проблемы. Первая проблема это извечная борьба CISC и RISC подходов к разработке программного обеспечения. За усложнением архитектуры процессора следует усложнение работы транслятора. Другая поднимаемая проблема это современная популяризация различного рода ускорителей (криптография, матричное умножение, архивация и тд.). Дело в том, что ускорители подразумевают загрузку целых алгоритмов в них для исполнения, однако написание кода под этот ускоритель становится отдельной задачей для разработчиков программного обеспечения. В такой парадигме хотелось бы переложить работу поиска алгоритмов на компилятор, чтобы пользователю не приходилось реализовывать множество версий своего алгоритма для каждой платформы.

3.2.1 Оптимизация двойного умножения

Оптимизация двойного умножения — это 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. Это стандартная архитектурно-зависимая оптимизация. Различные архитектуры предоставляют разные наборы команд. Иногда инструкции системы команд просты, что позволяет легко найти аналоги для любой архитектуры (например, инструкция add), но иногда, особенно в CISC-архитектурах, могут встречаться достаточно сложные инструкции, эквивалент которых потребует десятков и даже сотен команд. [115; 116].

Идея таких вычислений основана на максимальных значениях половинного умножения (s - размер в битах).

$$\left(2^{s/2}-1\right)^2 = 2^s - 2^{s/2+1} + 1 < 2^s - 1$$

При разделении аргументов на части $s/2$ широкое умножение можно переписать как:

$$\begin{aligned} res = a \cdot b &= \left(2^{s/2}a_{hi} + a_{lo}\right)\left(2^{s/2}b_{hi} + b_{lo}\right) \\ &= 2^s a_{hi}b_{hi} + 2^{s/2}\left(a_{hi}b_{lo} + a_{lo}b_{hi}\right) + a_{lo}b_{lo} \end{aligned}$$

Результат состоит из двух частей: младшей (от 1 до $2^s - 1$) и старшей (от 2^s до $2^{2s} - 1$). Первое слагаемое и правая часть второго слагаемого станут старшей частью результата. левая часть второго и третьего слагаемых станут младшей частью результата.

$$2^s \leqslant 2^s a_{hi}b_{hi} < 2^{2s}$$

$$\begin{aligned} 2^{s/2} &\leqslant 2^{s/2}\left(a_{hi}b_{lo} + a_{lo}b_{hi}\right) \\ &\leqslant 2^{3s/2}\left(2^{s+1} - 2^{s/2+2} + 1\right) \end{aligned}$$

$$1 \leqslant a_{lo}b_{lo} < 2^s$$

Несложно доказать, что сложения могут привести к переполнению во время этих вычислений. Переименуем слагаемые для лучшего восприятия:

$$\begin{aligned} mid\_res &= a_{hi}b_{lo} + a_{lo}b_{hi} \\ &= mid\_res\_real + mid\_res\_overflow \end{aligned}$$

$$\begin{aligned} res_{lo} &= 2^{s/2}mid\_res\_real_{lo} + a_{lo}b_{lo} \\ &= res\_low\_real + res\_low\_overflow \end{aligned}$$

$$\begin{aligned} res_{hi} &= middle\_res\_real_{hi} + a_{hi}b_{hi} \\ &= res\_high\_real + \frac{res\_low\_overflow}{2^s} \\ &\quad + \frac{mid\_res\_overflow}{2^{s/2}} \end{aligned}$$

$$res = 2^s \cdot res_{hi} + res_{lo}$$

Такие вычисления отыскиваются с использованием существующего механизма поиска шаблонов GCC и преобразуются в одиночные умножения более широких типов. Количество инструкций значительно уменьшено. На CPUBench наблюдалось улучшение производительности в 30 % на тесте OpenSSL.

3.2.2 Шаблонная криптография

В этом разделе поднимается важная тема для различных встраиваемых устройств и расширений архитектуры. В современном мире существует множество специализированных устройств для конкретных задач; например, ускорители для нейронных сетей [117], для научных вычислений [118], для обработки графов [119] и т. д.

В недавнем исследовании [120] Pecceriilo et al., 2022, классифицировали около 100 различных типов ускорителей. Таким образом, в этой тенденции компилятор становится очень практичным инструментом, который может компилировать (потенциально автоматически) и планировать выполнение задач на разных устройствах. В настоящее время разные компании пытаются разработать собственный подход к решению этой задачи [121—123].

Небольшая часть этой глобальной проблемы была решена в ходе нынешнего исследования. Kunpeng 920 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 оптимизации компилятора, которые могут определять возможность использования инструкций в соответствии с семантикой кода.

Оптимизация соотносит весь алгоритмам, включая предварительно рассчитанные таблицы, и статически проверяет, что все предварительно рассчитанные таблицы не изменяются во время выполнения. Оптимизация контролируется флагами **-fcrypto-accel-aes** и **-fcrypto-accel-crc32**. Оба шаблона были реализованы внутри RTL, поскольку они являются

архитектурно-зависимыми. На верхнем уровне логика оптимизаций очень похожа друг на друга. Следующие шаги описывают преобразование шаблона AES:

1. **Сбор ссылок на таблицы AES**: Разработан оптимизационный проход 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.

Для сопоставления внутреннего представления компилятора на уровне RTL использовался собственные шаблонный анализатор.Механизм сопоставления был создан на основе существующих генераторов сопоставлений match.pd GENERIC и GIMPLE. Код для необходимой проверки шаблона генерируется во время компиляции GCC. Анализатор генерирует заранее упорядоченную последовательность аргументов.

Специальные инструкции сократили общее количество инструкций в целевых горячих циклах, что привело к значительному ускорению. Улучшение производительности на 10+ % было достигнуто на тестах openSSL и gzip.

К сожалению, оптимизация crc32 не была принята, поскольку менее общая оптимизация, обеспечивающая большую производительность при использовании gzip, была предложена другими авторами.

3.2.3 Шаблонная подстановка инструкций

Одной из основных основных задачей компилятора является выбор наиболее подходящих инструкций, которыми можно будет лаконично и в то же время оптимально с точки зрения производительности выразить внутреннее представление программы [124]. В данной работе не изменяется работа

стандартного прохода выбора инструкций, однако добавляется несколько небольших шаблонов, который способствуют лучшей утилизации набора команд архитектуры ARM64.

Так, арифметическое выражение вида

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

до внесенных изменений транслировалось в векторной версии в (Листинг 3.10) может быть транслировано в (Листинг 3.11)

Листинг 3.10 Пример неоптимального выбора инструкций №1

```
xtn v18.4h, v17.4s
xtn2 v18.8h, v1.4
```

Листинг 3.11 Оптимальный выбор инструкций для Листинга 3.10

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

Таким же методом был преобразован код (Листинг 3.12) в более оптимальную версию, использующую инструкции smin/smax на (Листинг 3.13)

Листинг 3.12 Пример не оптимального выбора инструкций №2

```
sshr v1.4s, v1.4s, #10
neg v24.4s, v1.4s
mov v20.16b, v1.16b
sshr v24.4s, v24.4s, #31
bic v20.4s, #0xff
cmeq v20.4s, v20.4s, #0
bif v1.16b, v24.16b, v20.16b
```

Листинг 3.13 Оптимальный выбор инструкций для Листинга 3.12

```
movi v2.2d, #0x0 // (outside the loop)
movi v3.2d, #0xff000000ff // (outside the loop)
...
smax v18.4s, v18.4s, v2.4s // (inside the loop)
smin v18.4s, v18.4s, v3.4s // (inside the loop)
uzp1 v17.8h, v18.8h, v17.8h
```

3.3 Девиртуализация

Высокоуровневые языки программирования, такие как Java, C++ и C# и др. применяют динамические таблицы вызовов для реализации парадигмы полиморфизма [125—127]. В такой модели адрес вызываемой функции становится известным только во время исполнения программы. Виртуальные функции в объектно-ориентированных языках являются одним из основных источников косвенных переходов в программном коде. В языках C/C++ также можно использовать указатели на функции, а в GCC существует отдельное расширение, которое позволяет пользователю брать адрес на метку в коде программы [128], что позволяет пользователям создавать косвенные вызовы и переходы самостоятельно.

Несмотря на удобства во время написания программы, реализация переходов через косвенность может замедлить работу центрального процессора, так как для эффективной обработки управляющих инструкций суперскалярному процессору необходимо предсказывать целевой адрес перехода. Для этого используются специальные устройства-предсказатели [129; 130]. Однако, реализация предсказателя косвенных переходов существенно разнится от предсказателя условных. В случае условного перехода существуют всего два конечных адреса передачи управления, но для косвенных переходов это не так, в следствие этого предсказателю приходится запоминать адреса перехода, что может существенно увеличить площадь, занимаемую предсказателем [131]. В литературе описывается принцип виртуального счетчика инструкций, однако это все еще требует дополнительной логики в аппаратуре [132]. Современные решения этой задачи в основном

сосредоточены на разрешении вызовов виртуальных функций в контексте объектно-ориентированного программирования. Недостаток информации о целевых функциях в графе вызова затрудняет межпроцедурные оптимизации и делает невозможной подстановку функций непосредственно в код, что значительно ограничивает возможности компилятора для оптимизации [133; 134].

В статье [135] описывается два метода преобразования косвенных переходов: статический и динамический.

Статический метод по своей сути является расширением стандартной методологии девиртуализации компилятора GCC. Его работа разделена на две части:

1. **Анализ сигнатур функций**: Авторами предлагается анализировать сигнатуры функций для определений и вызовов. Если сигнатура вызываемой функции совпадает с сигнатурой определения то функция определения считается кандидатом.
2. **Трансформация косвенного перехода**: На этом этапе, в случае единственного кандидата, косвенный вызов функции заменяется вызовом найденной процедуры (рисунок 3.3). Если же кандидатов несколько, то приходится выстраивать цепочку сравнений адресов переходов. что не всегда является оптимальным

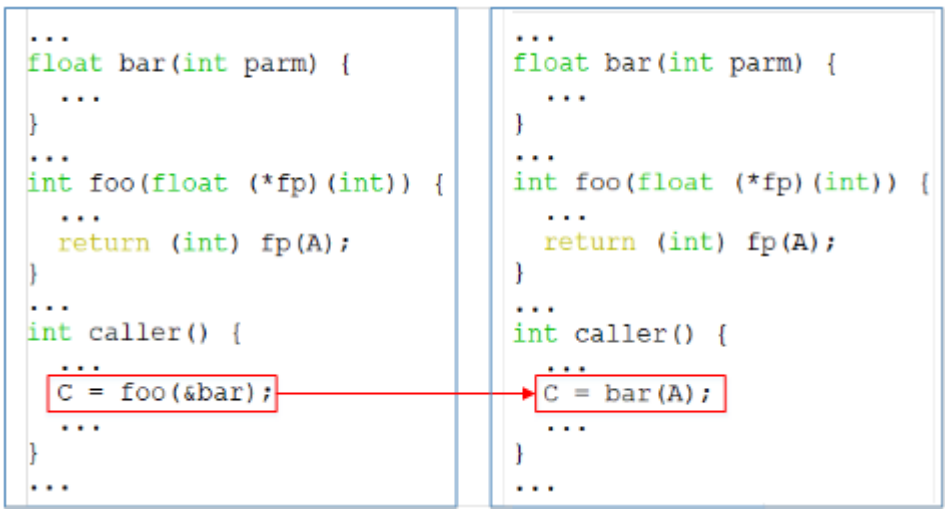

Рисунок 3.3 — Пример замены косвенного вызова

С другой стороны динамический метод использует информацию, собранную во время выполнения программы [136; 137], чтобы изменить косвенные переходы. При этом повторная компиляция программы не происходит. В этом состоит отличие от методов, использующих

профилирование или системы JIT-компиляции (Just-in-Time compilation). Для решения этой задачи предлагается внедрять инструменты для трансформации кода непосредственно в исполняемую программу. В системах JIT перекомпилированный код отдельного метода или функции обычно размещается в новом участке памяти [138]. Однако здесь предлагается добавить буферный участок кода, который будет изменяться во время работы приложения.

Алгоритм

1. **Выбор косвенных переходов**: Может происходить автоматически при помощи транслятора, на основе графа вызовов функций, либо в ручном. В поставляемой библиотеке доступны два макроопределения "DDL_GOTO" и "DDL_CALL при мпомощи которых можно заменить переход goto или косвенный вызов функции на библиотечный примитив.
2. **Трансформация косвенных переходов**: 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.
3. **Запуск программы и сбор статистики**: Этот этап и все последующие происходят во время выполнения программы и не требуют никаких действий от пользователя или компилятора. Каждый раз, когда программа достигает изменённого косвенного перехода, вызывается библиотечная функция для обновления статистики целевых адресов этого перехода. Библиотека собирает статистику отдельно для каждого перехода, который идентифицируется уникальным номером, присвоенным при инициализации.
4. **Трансформация в реальном времени**: Когда накоплено достаточное количество статистических данных, заданное параметром N, запускается функция модификации кода программы. Целевые адреса сортируются по убыванию числа переходов к ним. Если 95 % переходов осуществляется не более чем по k адресам, происходит замена последовательности инструкций NOP на условные переходы

(листинг 3.14). В противном случае происходит вставка исходного косвенного перехода.

5. **Работа оптимизированного перехода**: После изменения перехода сбор статистики целевых адресов прекращается. Тем не менее, информация о количестве выполнений этого перехода продолжает накапливаться, а также фиксируется число случаев, когда в преобразованном коде отсутствует прямой переход к полученному целевому адресу, что означает, что оптимизация для этого адреса не была выполнена и должен сработать первоначальный косвенный переход (расположенный дальше по коду). Если количество пропущенных переходов станет значительным по сравнению с общим числом входов в данный участок кода, цепочка прямых условных переходов будет заменена исходным блоком NOP инструкций, и сбор статистики возобновится. Иными словами, процесс вернется к шагу 2. При этом статистика адресов, собранная на предыдущем этапе выполнения шага 3, будет учитываться с коэффициентом 0.5, что позволит сохранить эту информацию.

Статический подход способствовал улучшению тестов tpcc и tpch. Хотелось бы отметить, что улучшение openssl в статье [135] ошибочно и является следствием некачественной валидации бенчмарка openssl в пакете CPUBench, а также со стороны авторов.

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 (рисунок 3.4). Деградация в 10 % также наблюдалась на отдельных тестах пакета "SpecCPU 2017".

3.4 Разбиение широких инструкций доступа в память

Архитектура ARM имеет сложные инструкции, которые выполняют содержат внутри себя несколько простых, например, арифметических или логических операций. Такие инструкции позволяют эффективно использовать

Листинг 3.14 Псевдокод преобразованного косвенного перехода

```
INT entry_count = 0;
entry_count++;
START_REWRITE_POINT:
NOP
NOP
...
NOP // k-times reapeat
BOOL collect_stat_mode = FALSE;
INT miss_count = 0;
miss_count++;
if ( entry_count == N)
&& (miss_count > entry_count >> 4) {
        if (!collect_stat_mode){
                collect_stat_mode = TRUE;
                CALL DDL_DISABLE_OPTIMIZATON();
        } else {
                miss_count = 0;
                entry_count = 1;
                CALL DDL_REWRITE_INDIRECT();
                collect_stat_mode = FALSE;
        }
}
if (collect_stat_mode) {
        CALL DDL_UPATE_STATISTICS();
}
goto *addr; // original indirect branch
```

ресурсы ЦПУ и уменьшают размер кода. Они повышают производительность и сокращают задержку выполнения операций. Однако для использования сложных инструкций необходимо тщательно понимать их функциональность и ограничения, чтобы обеспечить более качественное выполнение программы. Предыдущие исследования [139] показали, что разделение 128-битных инструкций чтения из памяти может улучшить производительность на отдельных тестах. В ходе данного исследования была продолжена эта работа, и было обнаружено, что в двух случаях использование широкого доступа может возникнуть потеря производительность.

Листинг 3.15 Пример преобразованного на ходу косвенного перехода

```
mov       w9 #0xe20
movk      w9 #0x40, lsl #16
cmp       x28, x9
b.eq      0x400e20
movk      w9, #0xe60
cmp       x28, x9
b.eq      0x400e60
movk      w9, #0xe30
cmp       x28, x9
b.eq      0x400e30
movk      w9, #0xe40
cmp       x28, x9
b.eq      0x400e40
movk      w9, #0xe50
cmp       x28, x9
b.eq      0x400e50
movk      w9, #0xdd4
cmp       x28, x9
b.eq      0x400dd4
movk      w9, #0xe10
cmp       x28, x9
b.eq      0x400e10
nop
nop
...
```

Один из них (см. рисунок 3.5) - 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) [140] не может быть выполнена в таком случае. Подробный разбор данной ситуации на симуляторе целевой микроархитектуры можно увидеть в главе 2.4.2. Поэтому было решено разделить широкие доступы к памяти в эмпирически установленном диапазоне.

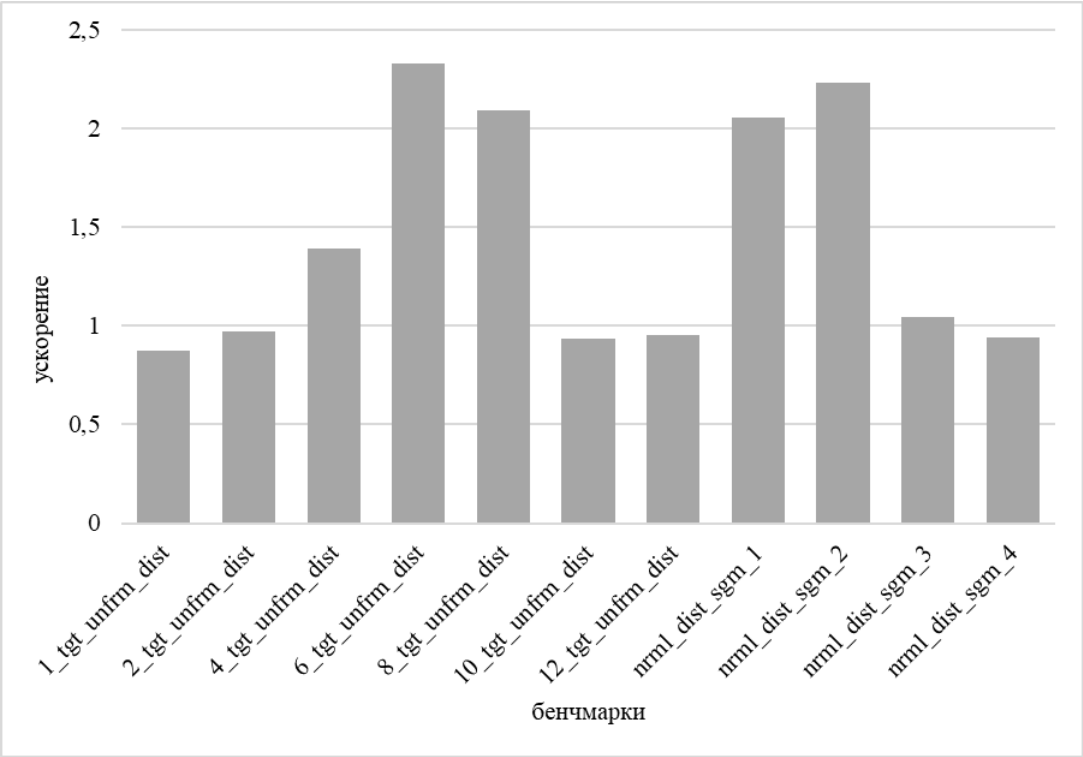

Рисунок 3.4 — Результаты замеров производительности динамического подхода в зависимости от распределения количества целевых адресов

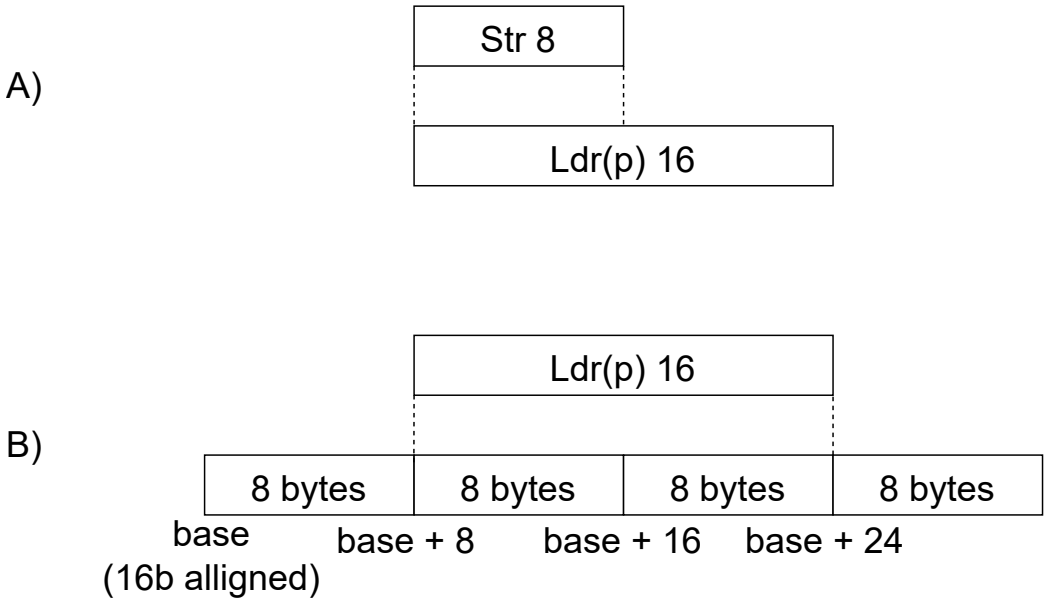

Рисунок 3.5 — Два случая, когда использование широкого доступа в память приводит к замедлению

Для решения этой проблемы был разработан алгоритм (см. рисунок 3.6), 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

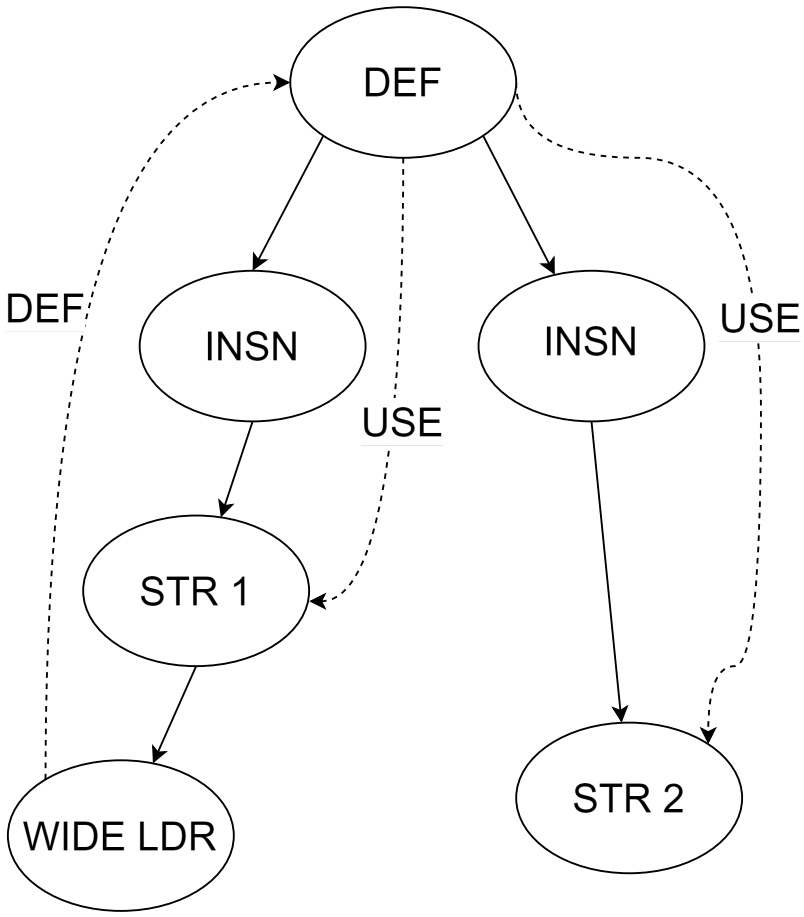

Рисунок 3.6 — Схема алгоритма разделения сложных инструкций

доступа к памяти. Было выявлено, что такой подход разумен, если расстояние между загрузкой и сохранением составляет менее 16 инструкций.

3.5 Уменьшение размеров типов переменных

Анализ диапазона значений переменных в компиляторе [141; 142] может помочь в удалении избыточных условий, улучшении постоянного распространения, удалении избыточных вычислений и т. д. В результате исследования было выявлено, что анализ диапазона значений GCC все еще имеет недостатки. Прежде всего, текущая оптимизация распространение диапазона значений не способна получать диапазоны из неизменяемых структур данных, что может быть полезно для предварительно вычисленных таблиц. Кроме того, GCC не может изменить размер переменной. Например, если диапазон переменной - $\{0, 1\}$, но пользователь использует тип **int** для нее, это неоднозначно, для этого можно испольэто не относится к

переменным с плавающей точкой, потому что точность может быть потеряна. Производитезовать тип **boolean**, то же самое касается **int64_t** и **int32_t**. К сожалению, льность на тесте gzip была улучшена.

3.6 Векторизация "ленивых" вычислений

"Ленивые" вычисления в языках C/C++ являются частью стандарта языка. Такой подход помогает оптимизировать программы в том смысле, что необходимые вычисления производятся только в тот момент, когда они нужны, а не заранее [143]. Например, в строке (if (pointer && pointer[idx] $>$ 0)) загрузка из памяти гарантированно не будет выполнена до того момента, как значение указателя не проверится на отличие от нуля.

Однако такая семантика может приводить к излишним конструкциям, которые в итоге демонстрируют замедление производительности в некоторых случаях. Так, например, в листинге 3.16 такой подход приводит к замедлению исполнения программы из-за того, что в окно суперскалярного процессора помещается слишком мало инструкций. С помощью метода обратной разработки из главы 2.4.3 первоначально была произведена замена целевого кода на векторизованный, что показало улучшение производительности, после этого можно было преступать к разработке самой оптимизации.

Листинг 3.16 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 ...
```

Тем не менее векторизация такого участка кода может привести к выходу за границу массива, и в случае выхода за границу выделенной границы памяти может произойти исключительная ситуация seegmentation fault. Чтобы этого избежать предлагается добавить в компилятор знание о страничном устройстве памяти. Обладая этим знанием компилятор может разрешить программе

производить чтение за пределами массива, если достоверно известно, что эта страница доступна программе во время исполнения. Тогда предлагается следующая схема (см рисунок 3.7): Перед входом в векторизованный код вставляется проверка на доступность всех ячеек памяти. Т.е динамически проверяется, что все доступы в память находятся в одной странице памяти. Если условие выполняется, то исполняется векторизованная версия кода, если же нет, то выбирается оригинальная версия [144].

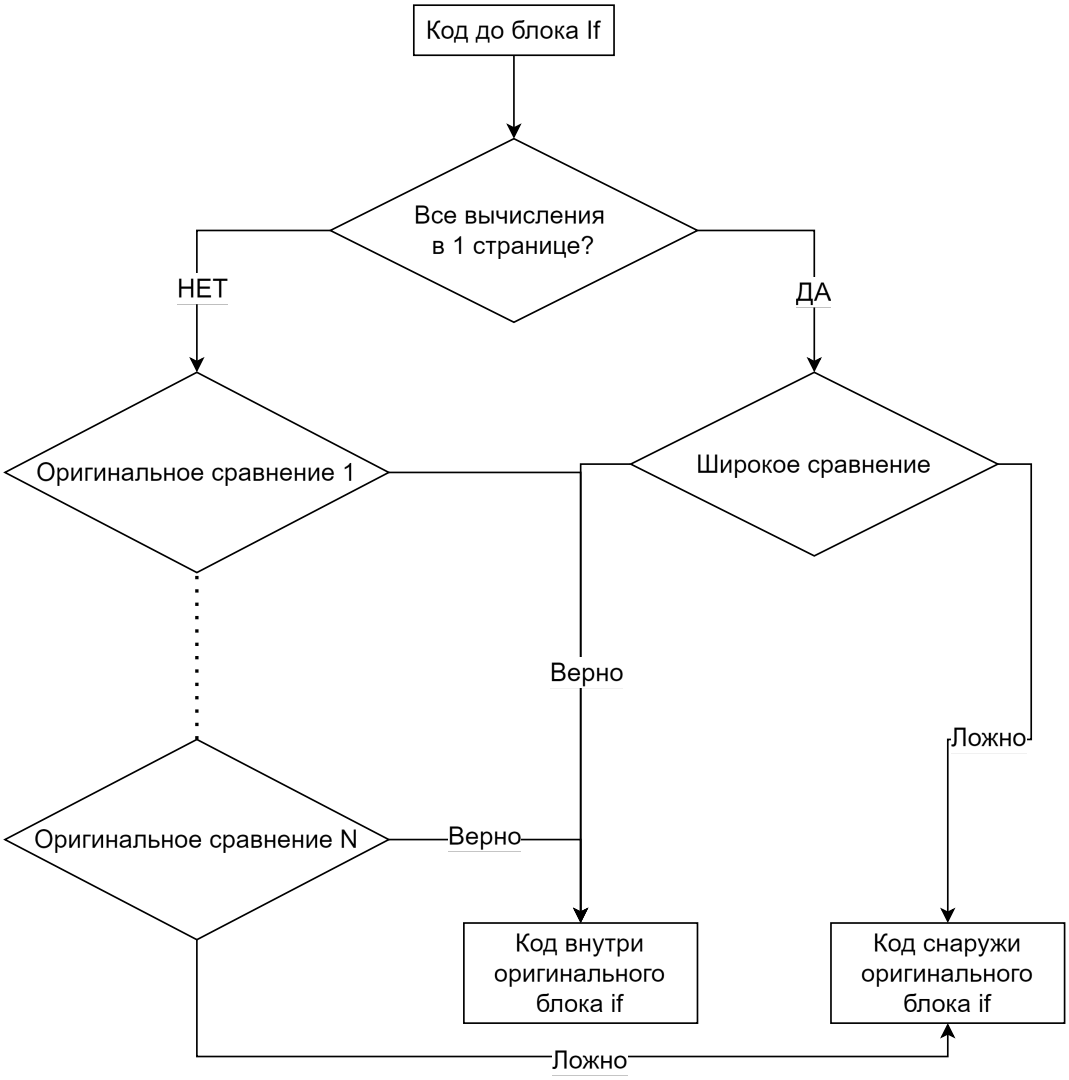

Рисунок 3.7 — Схема векторизации "ленивых" вычислений

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.

3.7 Слияние "хвостов" базовых блоков

Существует множество классических оптимизаций для минимизации количества инструкций на пути исполнения: поиск общих подвыражений (Common Subexpression Elimination, CSE), "ленивое" перемещение кода (Lazy Code Motion, LCM), частичное устранение избыточности (Partial Redundancy Elimination, PRE), перемещение инвариантов цикла (Loop Invariant Code Motion, LICM). Несмотря на все это многообразие, в процессе исследования было обнаружено, что современные компиляторы, GCC в частности, могут генерировать лишние инструкции. Рассмотрим пример из листинга 3.17: при его компиляции с помощью последней на момент написания диссертации версией GCC (14.2) с опцией O3" можно наблюдать код, как на листинге 3.18. Нетрудно заметить, что ассемблерный код

$$add \quad w3, w1, w3 \tag{3.6}$$

$$add \quad w0, w2, w0 \tag{3.7}$$

$$add \quad w0, w0, w3 \tag{3.8}$$

повторяется целых три раза. Справедливости ради, компилятор clang продуцирует иной код (листинг 3.19).

Для устранения этой проблемы предлагается следующий алгоритм:

1. На этапе сбора кандидатов определяются группы базовы блоков с общими наследниками.
2. Для каждой PHI-функции наследника проверяются определения ее аргументов.
3. Если все определение одинаковые, то они опускаются вниз.
4. Если определения выполняют одну и ту же операцию, но имеют различные аргументы, то определения опускаются только в том случае, если это не увеличит количество PHI-функции.

К сожалению данная оптимизация не показала заметных улучшений производительности на целевых тестах CPUBench, однако, на взгляд автора, она заслуживает внимания, так как позволяет уменьшать размер целевого кода без ущерба производительности.

Листинг 3.17 Пример исходного кода для оптимизации слияния "хвостов" базовых блоков

```
int foo(int a, int b, int r, int c, int x, int t)
{
        if (a > 5) {
                c += b + 7;
                x = a;
                r += a;
        } else if (a == 5) {
                c += b + 8;
                x = a;
                r += a;
        } else {
                r += a;
                x = a;
                c += b + t;
        }
        return r + x + c;
}
```

3.8 Предзагрузка косвенных доступов в память

Данная оптимизация была выполнена Дьячковым Ильей Леонидовичем, и ожидается, что в своем более подробном варианте оптимизация войдет в его диссертационную работу. Автор данной диссертации являлся руководителем и соавтором выполненной работы, поэтому позволяет себе кратко рассказать об оптимизации. Подробный обзор существующих решений был приведен в главе 1.3.

Анализ аппаратных событий (см. главу 2.4.1) в приложениях xz и gzip показал большое количество промахов в кэш третьего уровня в горячем цикле. Анализ показал, что в процессе своей работы прилложение очень часто совершает косвенное разыменование указателя. Это значит, что указатель на данные лежит в некой структуре данных (Рисунок 1.4). Сложность данной работы заключалась в том, что цикл находился глубоко вверху дерева вызовов и косвенная адресация находилась в функции, которая вызывалась косвенным вызовом (по указателю) (Рисунок 3.8) [111]. Благодаря анализу

Листинг 3.18 Ассемблер, полученный при компиляции листинга 3.17 с помощью GCC

```
foo(int, int, int, int, int, int):
cmp     w0, 5
ble     .L2
add     w1, w1, 7
add     w2, w0, w2
add     w3, w1, w3
add     w0, w2, w0
add     w0, w0, w3
ret
.L2:
beq     .L6
add     w1, w1, w5
add     w2, w0, w2
add     w3, w1, w3
add     w0, w2, w0
add     w0, w0, w3
ret
.L6:
add     w1, w1, 8
add     w2, w2, 5
add     w3, w1, w3
add     w0, w2, w0
add     w0, w0, w3
ret
```

из главы 3.3 информация о всех кандидатоов для косвенных переходов была уже собрана, оставалось обнаружить цикл, вставить предзагрузку данных и проверку, подобную описанной в главе 3.6. Здесь также используется знание о механизме страничной адресации памяти: вставляется динамическая проверка, проверяющая, что необходимая для предзагрузки цепочка загрузок из памяти лежит в доступных страницах. Чтобы разъяснить это утверждение приведем простой пример, который является упрощенной версией кода приложения xz.

В листинге 3.20 дана функция *foo*, которая вызывается в каком-то другом месте в цикле. Аппаратура очень хорошо справляется с тем, чтобы предсказывать доступ в память *load_ptr* так как его изменение происходит

Листинг 3.19 Ассемблер, полученный при компиляции листинга 3.17 с помощью clang

```
foo(int, int, int, int, int, int):
cmp     w0, #5
add     w8, w5, w1
add     w9, w1, #8
csel    w8, w8, w9, ne
add     w9, w1, #7
cmp     w0, #6
csel    w8, w8, w9, lt
add     w9, w2, w0, lsl #1
add     w8, w3, w8
add     w0, w9, w8
ret
```

чаще всего линейно, с другой стороны предсказание $a->storage[idx]$ дается аппаратуре сложно, потому что она не видит всей картины. Для того чтобы поставить инструкцию $prefetch$ на следующую выгрузку в память $a-> storage[idx]$ необходимо знать следующее значение idx, которое, к несчастью можно получить только загрузив из памяти $*(a->buf+a->a+1)$ (или еще более далекое значение). Наша цель найти переменную индукции в глобальном цикле $(++a->a)$ продвинуть ее, а затем сделать дополнительную выгрузку из памяти, чтобы получить $loaded\_value$ для следующей итерации. В этом месте вставляется проверка, что следующее $loaded\_value$ находится в той же странице памяти, что и текущее и если это так, то производится загрузка из памяти следующего $loaded\_value$ и исполняется инструкция $prefetch$ для следующих выгрузок в память $a->storage[idx0] = val$. Преобразованный код будет выглядеть следующим образом (Листинг 3.21).

Функция $SamePage$ проверяет, что два адреса находятся в одной и той же странице памяти. Стоит отметить, что такая оптимизация далеко не всегда будет давать производительность, так как определение точного шага индукционной переменной может быть затруднительно и не стоит забывать о том, что наша функция в оригинальном коде вызывается по косвенности, так что нам необходимо каким-то образом учитывать вероятность условных переходов и вызовов функций по указателю. На данном этапе ограничимся

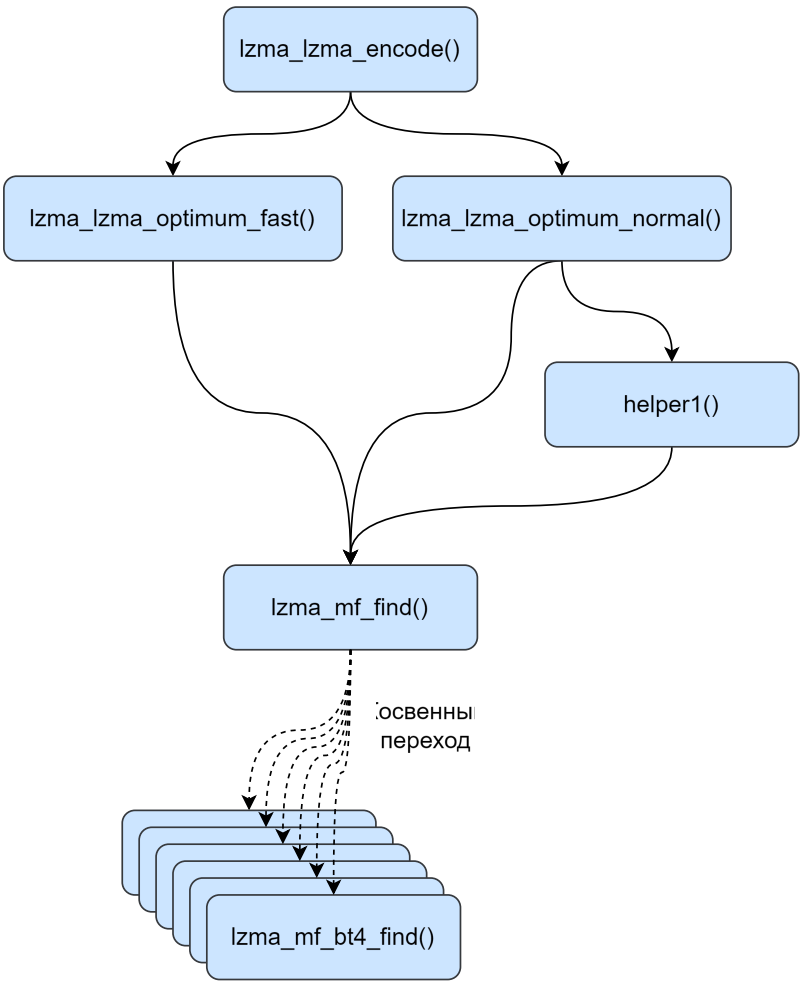

Рисунок 3.8 — Граф вызовов внутри основного цикла xz

статическими вероятностями внутри компилятора GCC, в дальнейшем, в главе 3.9, будет показано, как можно улучшить статическое предсказание переходов.

Представленное решение все еще имеет некоторые ограничения, накладываемые на цепь данных (Data-Flow chain), необходимой для подсчета адреса.

1. Цепочка данных может содержать только одно разыменование указателя.
2. На протяжении всей цепи не должно быть вызовов функций.
3. Цепь данных не содержит Phi-функций, за исключением головы цикла.

Конечно же данные ограничения являются возможностью для последующих исследований.

Листинг 3.20 Образец кода для анализа косвенной предзагрузки данных

```
typedef struct A
{
        uint8_t *buff;
        uint32_t *storage;
        uint32_t a;
} A;

//foo is calling in a loop somewhere
void foo(A *a, uint32_t val)
{
        ++a->a;

        const uint8_t *load_ptr = a->buf +a->a;
        const uint8_t loaded_value  = load_ptr[0];
        const uint32_t idx0 = loaded_value & 0x0F;
        const uint32_t idx1 = loaded_value & 0xF0;
        const uint32_t idx2 = loaded_value & 0xFF;

        // indirect stores that we want to prefetch
        a->storage[idx0] = val;
        a->storage[idx1] = val;
        a->storage[idx2] = val;
}
```

3.9 Автоматический подбор вероятностей условных переходов

В главе 1.5 обсуждались компиляторные оптимизации с использованием профиля. В рамках данной работы появилось желание получить улучшение производительности, сходное с оптимизациями с профилем, но без реального исполнения приложения во время компиляции. В компиляторе GCC существует встроенная возможность компиляции с использованием профиля. Для такого эксперимента необходимо скомпилировать приложение с дополнительной опцией **-fprofile-generate**, затем запустить исполнение аппликации, после чего перекомпилировать с дополнительной опцией **-fprofile-use**. В результате будет получено приложение скомпилированное с использование профиля. Замер производительности показывает среднее увеличение производительности в 5 %

Листинг 3.21 Листинг 3.20 после преобразования

```
typedef struct A
{
        uint8_t *buff;
        uint32_t *storage;
        uint32_t a;
} A;

//foo is calling in a loop somewhere
void foo(A *a, uint32_t val)
{

        ++a->a;

        const uint8_t *load_ptr = a->buf +a->a;
        const uint8_t loaded_value  = load_ptr[0];
        const uint32_t idx0 = loaded_value & 0x0F;
        const uint32_t idx1 = loaded_value & 0xF0;
        const uint32_t idx2 = loaded_value & 0xFF;

        a->storage[idx0] = val;
        a->storage[idx1] = val;
        a->storage[idx2] = val;

        if (SamePage(a->buf + a->a, a->buf + a->a+1))
        {
                const uint8_t *load_ptr_prf = a->buf +a->a +1;
                const uint8_t loaded_value_prf  = load_ptr_prf[0];
                const uint32_t idx0_prf = loaded_value_prf & 0x0F;
                const uint32_t idx1_prf = loaded_value_prf & 0xF0;
                const uint32_t idx2_prf = loaded_value_prf & 0xFF;
                prefetch_for_store(a->storage + idx0_prf);
                prefetch_for_store(a->storage + idx1_prf);
                prefetch_for_store(a->storage + idx2_prf);
        }
}
```

(Таблица 2), однако стоит отметить, что такой запуск практически недостижим в реальной жизни, так как профиль собирался на тех же входных данных, на

которых происходил замер производительности. Тем не менее присутствует значительный потенциал, который хотелось бы использовать.

Таблица 2 — Ускорение приложений "CPUBench fp" при компиляции с использованием профиля

Приложение	**Ускорение**
lightgbm	0.95
nektar	0.99
cube	1.00
openfoam	1.05
lammps	1.06
phenglei	1.06
phyml	1.08
povray	1.08
gromacs	1.14
Geomean	1.045

Ранее обсуждалась статья [54], в которой была применена техника искусственного профиля. К сожалению, использовать их результат напрямую не получится из-за сильного различия инфраструктуры GCC и LLVM. Поэтому для начала была предпринята попытка повторить их результат. В качестве набора признаков для каждого условного перехода собирался набор признаков, описанный в таблице 5. Оптимизация позволяет включить внутри компилятора проходы, которые обычно используются для оптимизаций с профилем (FDO).

Для сбора обучающей выборки был использован набор программ из пакета ExeBecnh [45]. Пакет содержит сотни маленьких приложений, которые можно быстро перетранслировать и собрать данные. Процесс сбора данных выглядит следующим образом (рисунок 3.9): Набор обучающих данных компилируется с использованием своего же профиля и в это время новый проход в компиляторе, названный **ipa-smart-profile**, собирает различную статическую информацию (таблица 5) для каждого условного перехода внутри программы.

Для обучения используется библиотека **XGBoost**, которая строит решающие деревья над собранным набором данных, модель сохраняется в

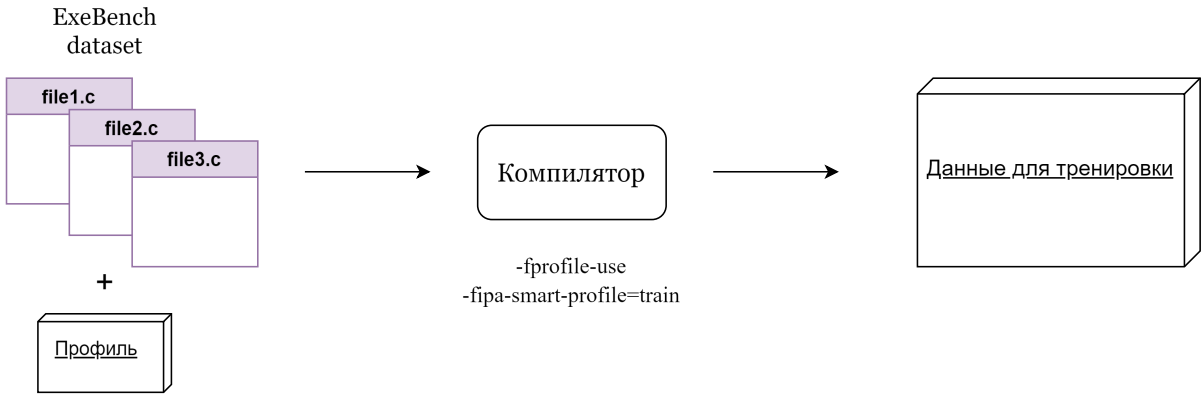

Рисунок 3.9 — Сбор данных для тренировки

бинарном формате, для последующего использования в компиляторе (Рисунок 3.10).

Рисунок 3.10 — Тренировка модели

Наконец, обученная модель может прогнозировать вероятности переходов без использования каких-либо данных профиля, а проход ipa-smart-profile включает оптимизации с профилем. Библиотека **XGBoost** также 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 (Рисунок 3.11).

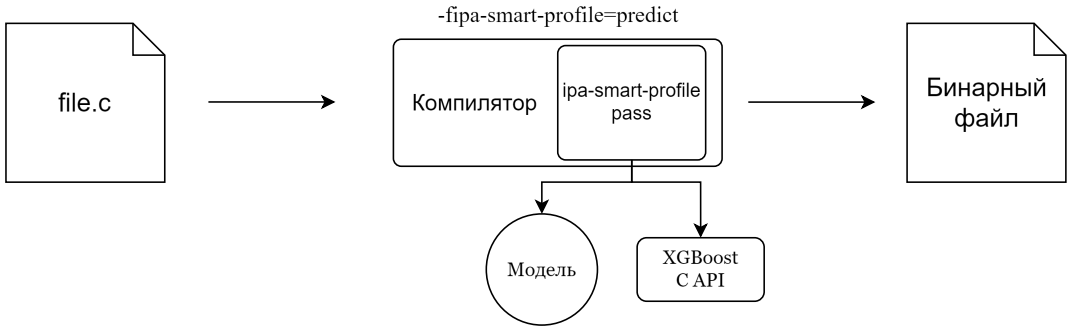

Рисунок 3.11 — Запуск модели во время компиляции целевого приложения

Качество натренированной модели можно оценить при помощи графика на рисунке 3.12. По горизонтальной оси: номер условного перехода в

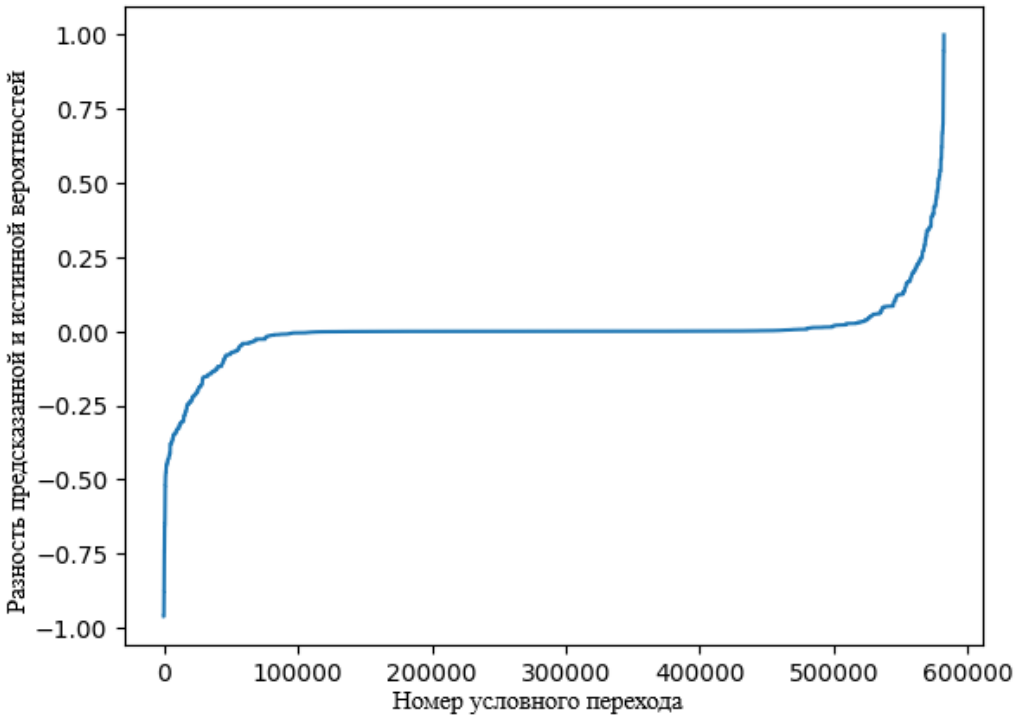

Рисунок 3.12 — S-кривая вероятностей предсказанных переходов на тестовом пакете ExeBench

тестовой выборке из пакета ExeBench после сортировки. По вертикальной оси: разница между предсказанной вероятностью с помощью модели и собранной при помощи инструментарии GCC. тем не менее, полученные результаты не оправдали своих ожиданий (таблица 3). Несмотря на улучшение производительности на тесте **gromacs** в 15 %, большинство тестов показало деградацию производительности. Методу требуется дальнейшее улучшение. Если вернуться к оригинальному исследованию [54], то в нем обучение и замер производительности проходил на одних и тех же данных, что в нашей ситуации неприемлемо.

Для решения проблемы деградации производительности было решено увеличить количество признаков и добавить тем самым больше информации об окружающем коде. Признаки, собираемые для перехода и условного перехода остались прежними (таблицы 4 и 5). Однако теперь информация о базовом блоке собиралась не только для основного базового блока, а также для его наследников степени 2 (условно для детей и внуков). Количество признаков увеличилось и стало больше сотни, а размер модели вырос до нескольких мегабайт. Это помогло побороть большинство деградаций производительности (сохранилась деградация 5% на phyml), однако уменьшило проивзодительность

Таблица 3 — Ускорение приложений "CPUBench fp" при компиляции с использованием предсказанного с помощью решающих деревьев профиля

Приложение	**Ускорение**
phyml	0.80
cube	0.89
nektar	0.90
lightgbm	0.95
lammps	0.98
phenglei	0.99
povray	1.00
openfoam	1.04
gromacs	1.15
Geomean	0.96

на тесте gromacs до 10 %. Тем не менее, это показывает, что подход имеет место быть, а дальнейшее его улучшение возможно с улучшением модели и добавлением признаков.

3.10 Разбиение условных выражений

В начале главы, в разделе 3.1.1, упоминались существенные преимущества преобразования условных переходов. Однако в этом разделе будет продемонстрирована в некотором смысле обратная оптимизация. Рассмотрим немного упрощенный пример из теста phyml на листинге 3.22.

Листинг 3.22 Пример кандидата для оптимизации разбиения условных выражений из теста phyml

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

Дело в том, что 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 (локальному и глобальному распространению константных выражений) трансформировав такой в листинг 3.23.

Листинг 3.23 Преобразованный листинг 3.22

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

Реализованная оптимизация состоит прохода по всем базовым блокам с условными выражениями. Проход ищет выражение, содержащее логическое ИЛИ, которое содержит сравнение некоторой переменной с константой. При этом по этому условию должен происходить переход на базовый блок или цепочку базовых блоков, содержащих вызов функции, один из аргументов которой является предикатом, либо его агрегатом. Естественно, глубина поиска агрегирования регулируется опцией (в работе была ограничена 2).Количество сравнений не является ограничением, так как каждый раз происходит отщепление одного из вариантов.

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

3.11 Выводы по главе и замеры производительности

В третьей главе было описано более десяти различных компиляторных оптимизаций и их улучшений, которые были получены после анализа литературы в главе 1 и при помощи методологии, описанной в главе 2. На момент написания диссертации сообществом были приняты оптимизации,

описанные в главах 3.1, 3.2, 3.3, 3.4, 3.6, 3.8. Автор надеется, что остальные оптимизации будут приняты в ближайшем будущем.

Результаты замеров, проведенных в полном соответствии с методологией, описанной в главе 2.3, показаны на рисунках 3.13 и 3.14. Можно видеть, что высокие цифры увеличения производительности на тестах пакета CPUBench достигаются за счет значительного ускорения криптографического приложения openSSL. Результаты, полученные на тестах "SpecCPU int" показывают ускорение за счет теста x264, который также находится в пакете CPUBench. На тестахс плавающей точкой очень хорошо видно, как предлагаемые оптимизацию перестают работать на большем числе копий. Такое поведение говорит об интенсивном использовании ресурсов памяти этими тестами. Особенно хорошо данных эффект просматривается на тесте nektar (рисунок 3.15). Тем не менее производительность одной копии существенно возросла, отдельные тесты достигают ускорения в 40 %. На наборе "SpecCPU fp" заметного улучшения производительности не наблюдается. Единственные тест, показавший улучшение это nab.

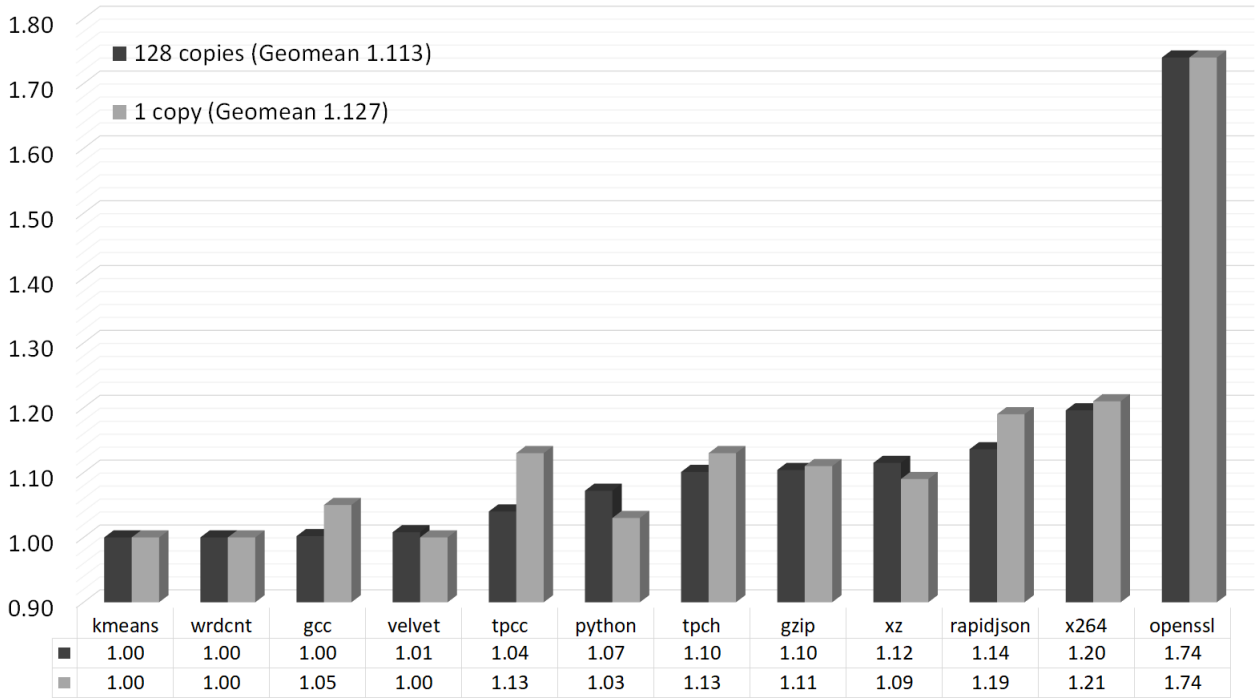

Рисунок 3.13 — Результаты замеров производительности на тестах пакета "CPUBench int"

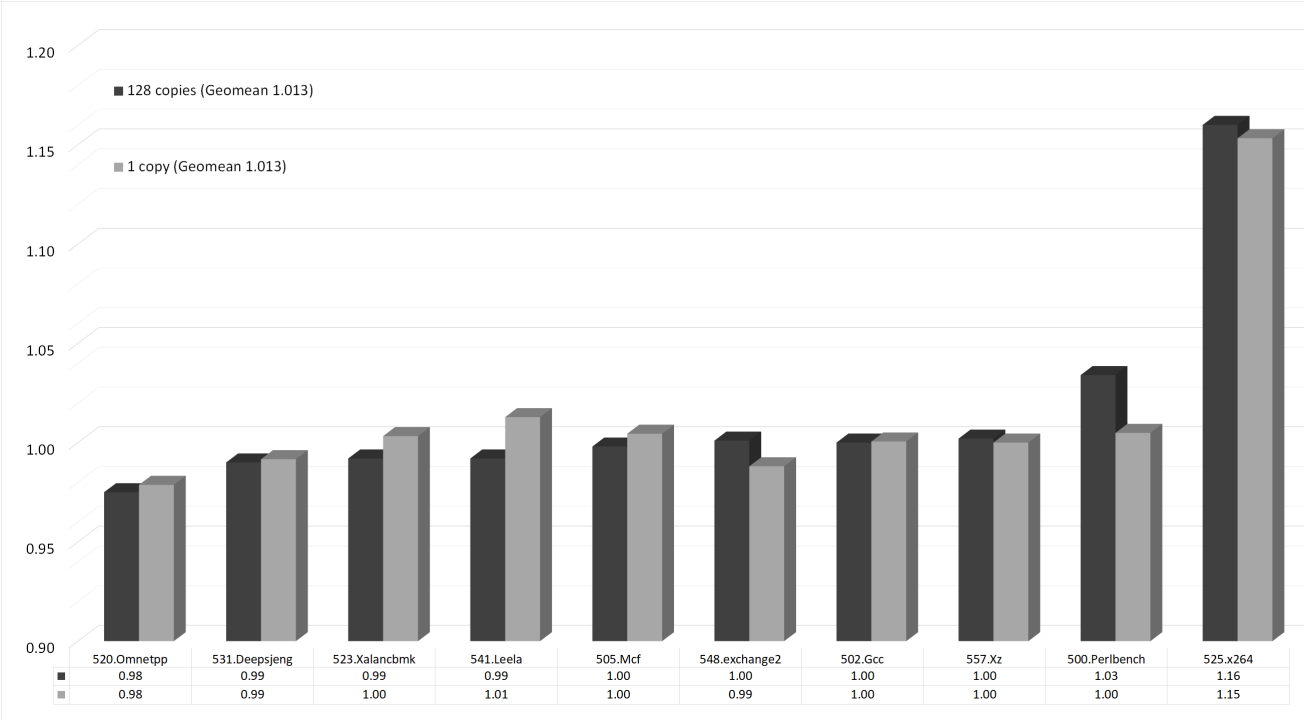

Рисунок 3.14 — Результаты замеров производительности на тестах пакета "SpecCPU int"

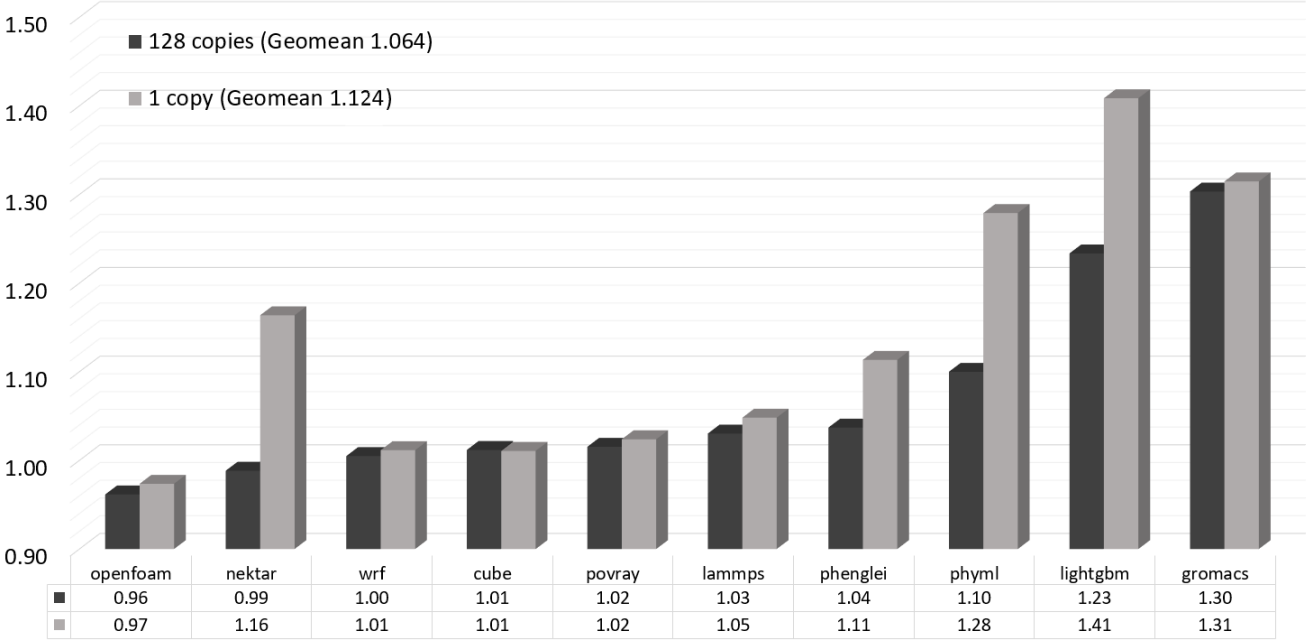

Рисунок 3.15 — Результаты замеров производительности на тестах пакета "CPUBench fp"

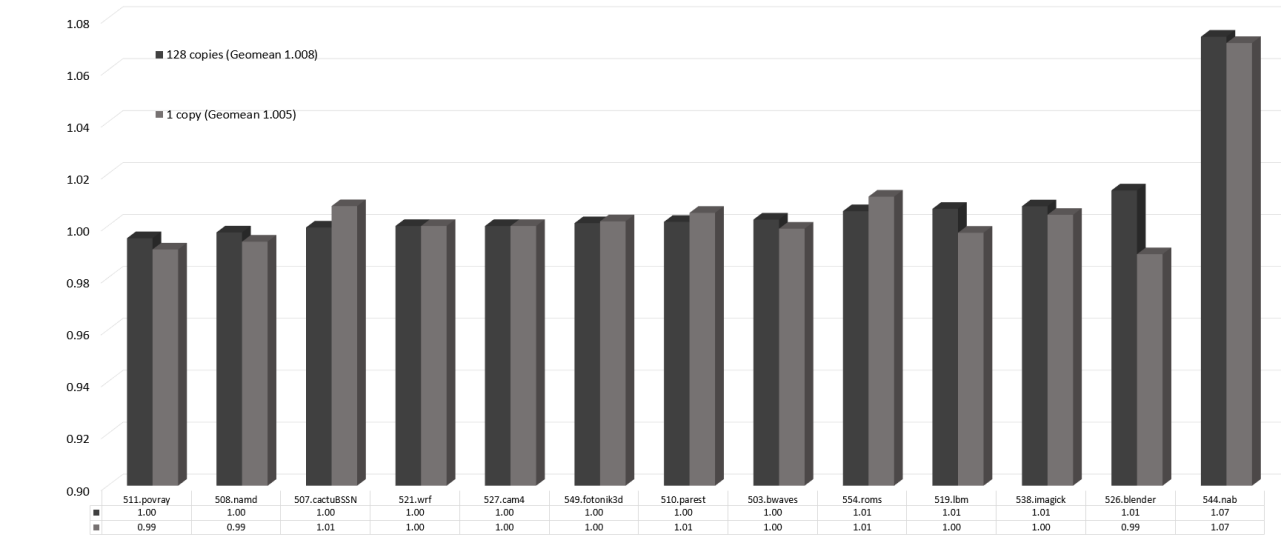

Рисунок 3.16 — Результаты замеров производительности на тестах пакета "SpecCPU fp"

Заключение

Одной из важнейших задач современных трансляторов является полноценное использование ресурсов системы. Чтобы решить эту задачу современные компанию ведут совместную разработку программного и аппаратного обеспечивания. Компиляторы помогают скрывать недостатки и подчеркивать важные особенности аппаратуры. В этой работе был продемонстрирован полноценный процесс, от анализа существующих решений и методологии тестирования, до разработки оптимизаций и замеров производительности.

Основные результаты работы 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. 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.

Словарь терминов

AES - Advanced Encryption Standard - Симметричный алгоритм блочного шифрования.

ASLR - Address Space Layout Randomization -Технология операционной системы рандомизации размещения адресного пространства.

ARM - Advanced RISC Machine - Описание архитектуры компьютера, разработанной компанией ARM Limited.

CCL - CPU CLaster - блок ядер центрального процессора.

CFG - Control Flow Graph - Граф потока управления.

CISC - Complex Instruction Set Computing - Сложная система команд, имеющая произвольный размер машинной инструкции и широкий набор операторов-инструкций.

CPU - Central Processing Unit - Центральное Вычислительное устройство.

CSE - Common Subexpression Elimination - Удаление общих подвыражений.

DFG - Data Flow Graph - Граф потока данных.

DRAM - Dynamic Random Access Memory - Динамическая энергозависимая память произвольного доступа.

GCC - GNU Compiler Collection - Коллекция компиляторов языков C/C++,Fortran, GO, D, ObjC и др.

GNU - GNU is Not Unix - Проект по разработке открытого программного обеспечения, основанный Ричардом Столлманом в 1983 году.

IPC - Instruction per cycle - Количество инструкций, выполняемых за один машинный такт.

LCM - Lazy Code Motion - "Ленивое" перемещение кода.

LICM - Loop Invariant Code Motion - Перемещение инвариантов цикла.

PMU - Performance Monitoring Unit - Блок процессора, предназначенный для исследования отслеживания различных событий.

PRE - Partial Redundancy Elimination - Частичное устранение избыточности.

RISC - Reduced Instruction Set Computer - Вычислитель, использующий упрощенный набор команд. Имеет фиксированный размер машинной инструкции и простые для исполнения операции.

SCCL - Super CPU CLaster - Блок центрального процессора, состоящий из некоторого количества CCL, контроллер памяти и дополнительный уровень кэширования памяти.

SICL Super IO Cluster - Блок центрального процессора, содержащий интерфейсы взаимодействия.

SLP - Superword Level Parallelism - Параллелизм на уровне суперслов.

SSA - Static Single Assignment Form - Внутреннее представление программы компилятором, в котором каждой переменной значение присваивается единожды.

Список литературы

1. *Vailshery, L. S.* Annual spending on cloud IT infrastructure worldwide from 2013 to 2026 [Электронный ресурс] [Текст] / L. S. Vailshery. — 2024. — Режим доступа: https://www.statista.com/statistics/503686/worldwide-cloud-it-infrastructure-market-spending/.
2. *Alam, T.* Cloud Computing and its role in the Information Technology [Текст] / T. Alam // IAIC Transactions on Sustainable Digital Innovation (ITSDI). — 2020. — Т. 1, № 2. — С. 108—115.
3. *Marinescu, D. C.* Cloud computing: theory and practice [Текст] / D. C. Marinescu. — Morgan Kaufmann, 2022.
4. *Bogusch, K.* Cloud Computing Costs in 2024 [Электронный ресурс] [Текст] / K. Bogusch. — 2024. — Режим доступа: https://www.oracle.com/uk/cloud/cloud-computing-cost.
5. *Юзбекова, И.* Миллиарды на серверах: зачем «Яндекс» выходит на рынок госзаказа [Электронный ресурс] [Текст] / И. Юзбекова. — 2021. — https://www.forbes.ru/tekhnologii/446373-milliardy-na-serverah-zacem-andeks-vyhodit-na-rynok-goszakaza.
6. *Hennessy, J. L.* Computer architecture: a quantitative approach [Текст] / J. L. Hennessy, D. A. Patterson. — Elsevier, 2011.
7. *Álvares, A. R.* Instruction visibility in SPEC CPU2017 [Текст] / A. R. Álvares, J. N. Amaral, F. M. Q. Pereira // Journal of Computer Languages. — 2021. — Т. 66. — С. 101062.
8. Performance, power, and energy-efficiency impact analysis of compiler optimizations on the spec cpu 2017 benchmark suite [Текст] / N. Schmitt [и др.] // 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC). — IEEE. 2020. — С. 292—301.
9. *Rodriguez, A.* Compiler Optimizations [Текст] / A. Rodriguez // Deep Learning Systems: Algorithms, Compilers, and Processors for Large-Scale Production. — Springer, 2021. — С. 159—176.

10. A case study: optimizing GCC on ARM for performance of libevas rasterization library [Текст] / D. Melnik [и др.] // Proceedings of GROW. — 2010.

11. Automatic tuning of compiler optimizations and analysis of their impact [Текст] / D. Plotnikov [и др.] // Procedia Computer Science. — 2013. — Т. 18. — С. 1312—1321.

12. A survey on compiler autotuning using machine learning [Текст] / A. H. Ashouri [и др.] // ACM Computing Surveys (CSUR). — 2018. — Т. 51, № 5. — С. 1—42.

13. A collaborative filtering approach for the automatic tuning of compiler optimisations [Текст] / S. Cereda [и др.] // The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems. — 2020. — С. 15—25.

14. Wait of a decade: Did spec cpu 2017 broaden the performance horizon? [Текст] / R. Panda [и др.] // 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA). — IEEE. 2018. — С. 271—282.

15. *Bucek, J.* SPEC CPU2017: Next-generation compute benchmark [Текст] / J. Bucek, K.-D. Lange, J. v. Kistowski // Companion of the 2018 ACM/SPEC International Conference on Performance Engineering. — 2018. — С. 41—42.

16. CPUBench: An open general computing CPU performance benchmark tool [Текст] / H. LU [и др.] // Microelectronics & Computer. — 2023. — Т. 40, № 5. — С. 75—83.

17. *Gough, B. J.* An Introduction to GCC. [Текст] / B. J. Gough, R. Stallman. — Network Theory Limited, 2004.

18. *Theodoridis, T.* Finding missed optimizations through the lens of dead code elimination [Текст] / T. Theodoridis, M. Rigger, Z. Su // Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. — 2022. — С. 697—709.

19. An empirical study of optimization bugs in GCC and LLVM [Текст] / Z. Zhou [и др.] // Journal of Systems and Software. — 2021. — Т. 174. — С. 110884.

20. *Lozano, R. C.* Survey on combinatorial register allocation and instruction scheduling [Текст] / R. C. Lozano, C. Schulte // ACM Computing Surveys (CSUR). — 2019. — Т. 52, № 3. — С. 1—50.

21. Compilers Principles, Techniques [Текст] / V. Alfred Aho [и др.]. — 2007.

22. *Poletto, M.* Linear scan register allocation [Текст] / M. Poletto, V. Sarkar // ACM Transactions on Programming Languages and Systems (TOPLAS). — 1999. — Т. 21, № 5. — С. 895—913.

23. *Subha, S.* A modified linear scan register allocation algorithm [Текст] / S. Subha // 2009 Sixth International Conference on Information Technology: New Generations. — IEEE. 2009. — С. 825—827.

24. *Smith, M. D.* A generalized algorithm for graph-coloring register allocation [Текст] / M. D. Smith, N. Ramsey, G. Holloway // Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation. — 2004. — С. 277—288.

25. *Briggs, P.* Register allocation via graph coloring [Текст] / P. Briggs. — Rice University, 1992.

26. *Rogers, I.* Efficient global register allocation [Текст] / I. Rogers // arXiv preprint arXiv:2011.05608. — 2020.

27. Rl4real: Reinforcement learning for register allocation [Текст] / S. VenkataKeerthy [и др.] // Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction. — 2023. — С. 133—144.

28. *Pohl, A.* Control flow vectorization for arm neon [Текст] / A. Pohl, B. Cosenza, B. Juurlink // Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems. — 2018. — С. 66—75.

29. Vectorizing posit operations on RISC-V for faster deep neural networks: experiments and comparison with ARM SVE [Текст] / M. Cococcioni [и др.] // Neural Computing and Applications. — 2021. — Т. 33. — С. 10575—10585.

30. Using Arm's scalable vector extension on stencil codes [Текст] / A. Armejach [и др.] // The Journal of Supercomputing. — 2020. — Т. 76, № 3. — С. 2039—2062.

31. *Brank, B.* Assessing the State of Autovectorization Support based on SVE [Текст] / B. Brank, D. Pleiter // 2022 IEEE International Conference on Cluster Computing (CLUSTER). — IEEE. 2022. — С. 556—562.

32. *Petrogalli, F.* LLVM and the automatic vectorization of loops invoking math routines:-FSIMDMATH [Текст] / F. Petrogalli, P. Walker // 2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC). — IEEE. 2018. — С. 30—38.

33. *Smith, A. J.* Design of CPU cache memories [Текст] / A. J. Smith. — Computer Science Division, University of California, 1987.

34. *Tse, J.* CPU cache prefetching: Timing evaluation of hardware implementations [Текст] / J. Tse, A. J. Smith // IEEE Transactions on Computers. — 1998. — Т. 47, № 5. — С. 509—526.

35. *Lee, R. L.* The effectiveness of caches and data prefetch buffers in large-scale shared memory multiprocessors [Текст] / R. L. Lee. — University of Illinois at Urbana-Champaign, 1987.

36. Implementation and optimization of data prefetching algorithm based on LLVM compilation system [Текст] / Y. Chai [и др.] // Journal of Physics: Conference Series. Т. 1827. — IOP Publishing. 2021. — С. 012136.

37. Apt-get: Profile-guided timely software prefetching [Текст] / S. Jamilan [и др.] // Proceedings of the Seventeenth European Conference on Computer Systems. — 2022. — С. 747—764.

38. LLVM-based automation of memory decoupling for OpenCL applications on FPGAs [Текст] / A. A. Purkayastha [и др.] // Microprocessors and Microsystems. — 2020. — Т. 72. — С. 102909.

39. *Ekemark, P.* Static Multi-Versioning for Efficient Prefetching [Текст] / P. Ekemark. — 2016.

40. *Leather, H.* Machine learning in compilers: Past, present and future [Текст] / H. Leather, C. Cummins // 2020 Forum for Specification and Design Languages (FDL). — IEEE. 2020. — С. 1—8.

41. DLFusion: An auto-tuning compiler for layer fusion on deep neural network accelerator [Текст] / Z. Liu [и др.] // 2020 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom). — IEEE. 2020. — С. 118—127.

42. *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.

43. *Wei, W.* Compiler Autotuning based on Hot Function for SHENWEI Processor [Текст] / W. Wei, Z. Qi, F. Wang // 2021 IEEE 6th International Conference on Computer and Communication Systems (ICCCS). — IEEE. 2021. — С. 1059—1062.

44. EAtuner: Comparative Study of Evolutionary Algorithms for Compiler Auto-tuning [Текст] / G. Xiao [и др.] // 2024 27th International Conference on Computer Supported Cooperative Work in Design (CSCWD). — IEEE. 2024. — С. 419—426.

45. ExeBench: an ML-scale dataset of executable C functions [Текст] / J. Armengol-Estapé [и др.] // Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming. — 2022. — С. 50—59.

46. Taco: A tool to generate tensor algebra kernels [Текст] / F. Kjolstad [и др.] // 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). — IEEE. 2017. — С. 943—948.

47. RISE & shine: Language-oriented compiler design [Текст] / M. Steuwer [и др.] // arXiv preprint arXiv:2201.03611. — 2022.

48. *Zhu, M.* Compiler Auto-Tuning via Critical Flag Selection [Текст] / M. Zhu, D. Hao // 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). — IEEE. 2023. — С. 1000—1011.

49. *Zhu, M.* Compiler Autotuning through Multiple-phase Learning [Текст] / M. Zhu, D. Hao, J. Chen // ACM Transactions on Software Engineering and Methodology. — 2024. — Т. 33, № 4. — С. 1—38.

50. *Liew, C. W.* Feedback Directed Optimization [Текст] / C. W. Liew. — 1994.

51. *Dange, S. S.* A systematic review on just in time (JIT) [Текст] / S. S. Dange, P. N. Shende, C. S. Sethia // Journal of Emerging Technologies and Innovative Research (JETIR). — 2014. — Т. 1, № 5. — С. 300—304.

52. *Chen, D.* AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications [Текст] / D. Chen, D. X. Li, T. Moseley // Proceedings of the 2016 International Symposium on Code Generation and Optimization. — 2016. — С. 12—23.

53. *Лисицын, С. А.* ОИсследование и оптимизация применения трасс исполнения приложения для статической бинарной трансляции под RISC архитектуры : дис. ... канд. тех. наук : 30.06.22 [Текст] / С. А. Лисицын. — М., 2022. — 105 с.

54. *Rotem, N.* Profile guided optimization without profiles: A machine learning approach [Текст] / N. Rotem, C. Cummins // arXiv preprint arXiv:2112.14679. — 2021.

55. *Reid, A.* Trustworthy specifications of ARM® v8-A and v8-M system level architecture [Текст] / A. Reid // 2016 Formal Methods in Computer-Aided Design (FMCAD). — IEEE. 2016. — С. 161—168.

56. Kunpeng 920: The first 7-nm chiplet-based 64-core arm soc for cloud services [Текст] / J. Xia [и др.] // IEEE Micro. — 2021. — Т. 41, № 5. — С. 67—75.

57. Performance evaluation of Ampere Altra [Текст] / S. Oshima, T. Nagai, T. Katagiri [и др.] // Research Report High Performance Computing (HPC). — 2021. — Т. 2021, № 9. — С. 1—9.

58. *Siever, E.* Perl in a Nutshell [Текст] / E. Siever, S. Spainhour, N. Patwardhan. — O'Reilly Media, Inc, 1998.

59. *Löbel, A.* Solving large-scale multiple-depot vehicle scheduling problems [Текст] / A. Löbel // Computer-Aided Transit Scheduling: Proceedings, Cambridge, MA, USA, August 1997. — Springer, 1999. — С. 193—220.

60. *Varga, A.* A practical introduction to the OMNeT++ simulation framework [Текст] / A. Varga // Recent Advances in Network Simulation: The OMNeT++ Environment and its Ecosystem. — Springer, 2019. — С. 3—51.

61. *Euzenat, J.* XML transformation flow processing [Текст] / J. Euzenat, L. Tardif // Markup Languages Theory and Practice. — 2002. — Т. 3, № 3. — С. 285—311.

62. *Merritt, L.* x264: A high performance H. 264/AVC encoder [Текст] / L. Merritt, R. Vanam // online] http://neuron2. net/library/avc/overview_ x264_v8_5. pdf. — 2006.

63. *Sandin, L.* The SSDF Chess Engine Rating List, 2022-01 [Текст] / L. Sandin // ICGA Journal. — 2021. — Т. 43, № 4. — С. 236—239.

64. How does AI improve human decision-making? Evidence from the AI-powered Go program [Текст] / S. Choi [и др.] // Evidence from the AI-Powered Go Program (April 2022). USC Marshall School of Business Research Paper Sponsored by iORB, No. Forthcoming. — 2022.

65. *Metcalf, M.* A Sudoku program in Fortran 95 [Текст] / M. Metcalf // SIGPLAN Fortran Forum. — New York, NY, USA, 2006. — Апр. — Т. 25, № 1. — С. 4—7. — URL: https://doi.org/10.1145/1124708.1124709.

66. *Koranne, S.* Compression Engines [Текст] / S. Koranne, S. Koranne // Handbook of Open Source Tools. — 2011. — С. 155—164.

67. *Auer, L.* Intracranial pressure oscillations (B-waves) caused by oscillations in cerebrovascular volume [Текст] / L. Auer, I. Sayama // Acta neurochirurgica. — 1983. — Т. 68. — С. 93—100.

68. A scientific application benchmark using the cactus framework [Текст] : тех. отч. / G. Allen [и др.] ; Technical report, Louisiana State University, Center for Computation ... — 2007.

69. NAMD: Biomolecular simulation on thousands of processors [Текст] / J. C. Phillips [и др.] // SC'02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing. — IEEE. 2002. — С. 36—36.

70. *Hoon Lee, J.* Fully adaptive finite element based tomography using tetrahedral dual-meshing for fluorescence enhanced optical imaging in tissue [Текст] / J. Hoon Lee, A. Joshi, E. M. Sevick-Muraca // Optics Express. — 2007. — Т. 15, № 11. — С. 6955—6975.

71. *Plachetka, T.* POV Ray: persistence of vision parallel raytracer [Текст] / T. Plachetka // Proc. of Spring Conf. on Computer Graphics, Budmerice, Slovakia. Т. 123. — 1998. — С. 129.

72. A description of the advanced research WRF version 4 [Текст] / W. C. Skamarock [и др.] // NCAR tech. note ncar/tn-556+ str. — 2019. — Т. 145.

73. *Brito, A.* Blender 3D [Текст] / A. Brito. — Novatec New York, 2007.

74. The mean climate of the Community Atmosphere Model (CAM4) in forced SST and fully coupled experiments [Текст] / R. B. Neale [и др.] // Journal of Climate. — 2013. — Т. 26, № 14. — С. 5150—5168.

75. *Still, M.* The definitive guide to ImageMagick [Текст] / M. Still. — Apress, 2006.

76. Nab: Measurement principles, apparatus and uncertainties [Текст] / D. Počanić [и др.] // Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment. — 2009. — Т. 611, № 2/3. — С. 211—215.

77. *Sullivan, D. M.* Electromagnetic simulation using the FDTD method [Текст] / D. M. Sullivan. — John Wiley & Sons, 2013.

78. Ocean forecasting in terrain-following coordinates: Formulation and skill assessment of the Regional Ocean Modeling System [Текст] / D. B. Haidvogel [и др.] // Journal of computational physics. — 2008. — Т. 227, № 7. — С. 3595—3624.

79. *Robinson, C.* Over 2000 SPEC CPU 2017 Results Flagged for Compiler Optimization [Текст] / C. Robinson // ServeTheHome. — 2024.

80. Comparison of open source compression algorithms on VHR remote sensing images for efficient storage hierarchy [Текст] / A. Akoguz [и др.] // The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences. — 2016. — Т. 41. — С. 3—9.

81. *Leutenegger, S. T.* A modeling study of the TPC-C benchmark [Текст] / S. T. Leutenegger, D. Dias // ACM Sigmod Record. — 1993. — Т. 22, № 2. — С. 22—31.

82. *Barata, M.* An overview of decision support benchmarks: TPC-DS, TPC-H and SSB [Текст] / M. Barata, J. Bernardino, P. Furtado // New Contributions in Information Systems and Technologies: Volume 1. — 2015. — С. 619—628.

83. *Zerbino, D. R.* Velvet: algorithms for de novo short read assembly using de Bruijn graphs [Текст] / D. R. Zerbino, E. Birney // Genome research. — 2008. — Т. 18, № 5. — С. 821—829.

84. *Rescorla, E.* An Introduction to OpenSSL Programming (Par t I) [Текст] / E. Rescorla // Linux J. — 2001. — Т. 2001, № 89. — С. 3.

85. *Keiser, J.* On-demand JSON: A better way to parse documents? [Текст] / J. Keiser, D. Lemire // Software: Practice and Experience. — 2023.

86. *Python, W.* Python [Текст] / W. Python // Python releases for windows. — 2021. — Т. 24.

87. Lightgbm: A highly efficient gradient boosting decision tree [Текст] / G. Ke [и др.] // Advances in neural information processing systems. — 2017. — Т. 30.

88. Nektar++: An open-source spectral/hp element framework [Текст] / C. D. Cantwell [и др.] // Computer physics communications. — 2015. — Т. 192. — С. 205—219.

89. *Zhao, Z.* Design of general CFD software PHengLEI [Текст] / Z. Zhao, L. He, X.-y. HE // Computer Engineering & Science. — 2020. — Т. 42, № 02. — С. 210.

90. New algorithms and methods to estimate maximum-likelihood phylogenies: assessing the performance of PhyML 3.0 [Текст] / S. Guindon [и др.] // Systematic biology. — 2010. — Т. 59, № 3. — С. 307—321.

91. GROMACS: fast, flexible, and free [Текст] / D. Van Der Spoel [и др.] // Journal of computational chemistry. — 2005. — Т. 26, № 16. — С. 1701—1718.

92. *Jasak, H.* OpenFOAM: Open source CFD in research and industry [Текст] / H. Jasak // International journal of naval architecture and ocean engineering. — 2009. — Т. 1, № 2. — С. 89—94.

93. *Gowthaman, S.* A review on mechanical and material characterisation through molecular dynamics using large-scale atomic/molecular massively parallel simulator (LAMMPS) [Текст] / S. Gowthaman // Functional Composites and Structures. — 2023. — Т. 5, № 1. — С. 012005.

94. *Yu, H.-R.* CUBE: an information-optimized parallel cosmological N-body algorithm [Текст] / H.-R. Yu, U.-L. Pen, X. Wang // The Astrophysical Journal Supplement Series. — 2018. — Т. 237, № 2. — С. 24.

95. *Zhang, X.* Hardware Execution Throttling for Multi-core Resource Management. [Текст] / X. Zhang, S. Dwarkadas, K. Shen // USENIX Annual Technical Conference. — 2009.

96. ASLR on the Line: Practical Cache Attacks on the MMU. [Текст] / B. Gras [и др.] // NDSS. Т. 17. — 2017. — С. 26.

97. *Graham, S. L.* Gprof: A call graph execution profiler [Текст] / S. L. Graham, P. B. Kessler, M. K. McKusick // ACM Sigplan Notices. — 2004. — Т. 39, № 4. — С. 49—57.

98. *Reinders, J.* VTune performance analyzer essentials [Текст]. Т. 9 / J. Reinders. — Intel Press Santa Clara, 2005.

99. *Nethercote, N.* Valgrind: a framework for heavyweight dynamic binary instrumentation [Текст] / N. Nethercote, J. Seward // ACM Sigplan notices. — 2007. — Т. 42, № 6. — С. 89—100.

100. *De Melo, A. C.* The new linux'perf'tools [Текст] / A. C. De Melo // Slides from Linux Kongress. Т. 18. — 2010. — С. 1—42.

101. *Hansen, T. E.* Examining the use of ARM PMUs for DVFS and scheduling [Текст] / T. E. Hansen. — 2020.

102. *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.

103. The gem5 simulator: Version 20.0+ [Текст] / J. Lowe-Power [и др.] // arXiv preprint arXiv:2007.03152. — 2020.

104. Performance Error Evaluation of gem5 Simulator for ARM Server [Текст] / Y. Qiu [и др.] // 2023 IEEE 15th International Conference on ASIC (ASICON). — IEEE. 2023. — С. 1—4.

105. *Lattner, C.* LLVM and Clang: Next generation compiler technology [Текст] / C. Lattner // The BSD conference. Т. 5. — 2008. — С. 1—20.

106. Evaluation of Intel's DPC++ Compatibility Tool in heterogeneous computing [Текст] / G. Castaño [и др.] // Journal of Parallel and Distributed Computing. — 2022. — Т. 165. — С. 120—129.

107. *Ferguson, J.* Reverse engineering code with IDA Pro [Текст] / J. Ferguson. — Syngress, 2008.

108. *MESTER, A.* MALWARE ANALYSIS AND STATIC CALL GRAPH GENERATION WITH RADARE2 [Текст] / A. MESTER // Studia Universitatis Babeș-Bolyai Informatica. — 2023. — С. 5—20.

109. A Comprehensive Study on ARM Disassembly Tools [Текст] / M. Jiang [и др.] // IEEE Transactions on Software Engineering. — 2022. — Т. 49, № 4. — С. 1683—1703.

110. *Bruel, C.* If-Conversion [Текст] / C. Bruel // SSA-based Compiler Design. — Springer, 2021. — С. 269—283.

111. 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.

112. *Nuzman, D.* Autovectorization in GCC–two years later [Текст] / D. Nuzman, A. Zaks // Proceedings of the 2006 GCC Developers Summit. Т. 6. — 2006.

113. *Rosen, I.* Loop-aware SLP in GCC [Текст] / I. Rosen, D. Nuzman, A. Zaks // GCC Developers Summit. — 2007. — С. 131—142.

114. *Guo, Z.-J.* A New Improved Algorithm for SLP [Текст] / Z.-J. Guo, H. Liu // International Journal of Performability Engineering. — 2017. — Т. 13, № 7. — С. 1087.

115. Reduced Instruction Set Computer (RISC): A Survey [Текст] / M. Bansal [и др.] // Journal of Physics: Conference Series. Т. 1916. — IOP Publishing. 2021. — С. 012040.

116. *Isen, C.* A tale of two processors: Revisiting the RISC-CISC debate [Текст] / C. Isen, L. K. John, E. John // Computer Performance Evaluation and Benchmarking: SPEC Benchmark Workshop 2009, Austin, TX, USA, January 25, 2009. Proceedings. — Springer. 2009. — С. 57—76.

117. A survey of accelerator architectures for deep neural networks [Текст] / Y. Chen [и др.] // Engineering. — 2020. — Т. 6, № 3. — С. 264—274.

118. Comparing hardware accelerators in scientific applications: A case study [Текст] / R. Weber [и др.] // IEEE Transactions on Parallel and Distributed Systems. — 2010. — Т. 22, № 1. — С. 58—68.

119. *Rahman, S.* Graphpulse: An event-driven hardware accelerator for asynchronous graph processing [Текст] / S. Rahman, N. Abu-Ghazaleh, R. Gupta // 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). — IEEE. 2020. — С. 908—921.

120. A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives [Текст] / B. Peccerillo [и др.] // Journal of Systems Architecture. — 2022. — Т. 129. — С. 102561.

121. *Tavarageri, S.* Automatic Model Parallelism for Deep Neural Networks with Compiler and Hardware Support [Текст] / S. Tavarageri, S. Sridharan, B. Kaul // arXiv preprint arXiv:1906.08168. — 2019.

122. *Chen, C.* Case: A compiler-assisted scheduling framework for multi-gpu systems [Текст] / C. Chen, C. Porter, S. Pande // Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. — 2022. — С. 17—31.

123. Towards intelligent compiler optimization [Текст] / M. Kovac [и др.] // 2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO). — IEEE. 2022. — С. 948—953.

124. *Blindell, G. H.* Instruction Selection [Текст] / G. H. Blindell // Principles, Methods, and Applications. — 2016.

125. *Calder, B.* Quantifying behavioral differences between C and C++ programs [Текст] / B. Calder, D. Grunwald, B. Zorn // Journal of Programming languages. — 1994. — Т. 2, № 4. — С. 313—351.

126. Overview of the IBM Java just-in-time compiler [Текст] / T. Suganuma [и др.] // IBM systems Journal. — 2000. — Т. 39, № 1. — С. 175—193.

127. *Bauer, M.* Novt: Eliminating C++ virtual calls to mitigate vtable hijacking [Текст] / M. Bauer, C. Rossow // 2021 IEEE European Symposium on Security and Privacy (EuroS&P). — IEEE. 2021. — С. 650—666.

128. *Shah, A.* Function Pointers in C-An Empirical Study [Текст] : тех. отч. / A. Shah, B. G. Ryder ; Rutgers University. — 1995.

129. *McFarling, S.* Combining branch predictors [Текст] : тех. отч. / S. McFarling ; Citeseer. — 1993.

130. *Mittal, S.* A survey of techniques for dynamic branch prediction [Текст] / S. Mittal // Concurrency and Computation: Practice and Experience. — 2019. — Т. 31, № 1. — e4666.

131. *Driesen, K.* Accurate indirect branch prediction [Текст] / K. Driesen, U. Hölzle // ACM SIGARCH Computer Architecture News. — 1998. — Т. 26, № 3. — С. 167—178.

132. *Redmond, W.* VPC Prediction: Reducing the Cost of Indirect Branches via Hardware-Based Dynamic Devirtualization [Текст] / W. Redmond, M. Hudson. — 2007.

133. *Li, D. X.* Lightweight feedback-directed cross-module optimization [Текст] / D. X. Li, R. Ashok, R. Hundt // Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization. — 2010. — С. 53—61.

134. *Pande, H. D.* Data-flow-based virtual function resolution [Текст] / H. D. Pande, B. G. Ryder // International Static Analysis Symposium. — Springer. 1996. — С. 238—254.

135. *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.

136. *Baev, I.* Profile-based indirect call promotion [Текст] / I. Baev, Q. I. Center // LLVM Developers Meeting, Oct. — 2015.

137. A study of devirtualization techniques for a Java Just-In-Time compiler [Текст] / K. Ishizaki [и др.] // Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. — 2000. — С. 294—310.

138. *CRAVVFORD, K. M.* A study of JIT implementation and operating problems [Текст] / K. M. CRAVVFORD, J. H. Blackstone Jr, J. F. Cox // The International Journal Of Production Research. — 1988. — Т. 26, № 9. — С. 1561—1568.

139. Microarchitecture-aware code generation for deep learning on single-isa heterogeneous multi-core mobile processors [Текст] / J. Park [и др.] // IEEE Access. — 2019. — Т. 7. — С. 52371—52378.

140. *Shen, J. P.* Modern processor design: fundamentals of superscalar processors [Текст] / J. P. Shen, M. H. Lipasti. — Waveland Press, 2013.

141. *Harrison, W. H.* Compiler analysis of the value ranges for variables [Текст] / W. H. Harrison // IEEE Transactions on software engineering. — 1977. — № 3. — С. 243—250.

142. *Simon, A.* Value-Range analysis of c programs: Towards proving the absence of buffer overflow vulnerabilities [Текст] / A. Simon. — Springer, 2008.

143. *Cukic, I.* Functional programming in C++ [Текст] / I. Cukic. — Simon, Schuster, 2018.

144. *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.

Список рисунков

Список таблиц

Приложение А

Собираемые признаки, для задачи автоматического подбора вероятностей условного перехода

Таблица 4 — Признаки базового блока собранные в компиляторе GCC

Признак	Тип	Описание
num_instr	Int	Количество инструкций в базовом блоке, в котором находится условный переход
num_phis	Int	Количество PHI-функций в базовом блоке, в котором находится условный переход
num_loads	Int	Количество загрузок из памяти в базовом блоке, в котором находится условный переход
num_stores	Int	Количество выгрузок в память в базовом блоке, в котором находится условный переход
num_pred	Int	Количество базовых блоков, управление из которых может перейти в базовый блок с условным переходом
num_succ	Int	Количество базовых блоков, управление в которые может перейти из базового блока с условным переходом
is_entry	Bool	Является ли базовый блок входом в текущую функцию

Таблица 5 — Признаки условного перехода собранные в компиляторе GCC

Признак	**Тип**	**Описание**
lhs_type	Int	Тип левого операнда операции сравнения
rhs_type	Int	Тип правого операнда операции сравнения
is_rhs_const	Bool	Является ли правый операнд сравнения константной величиной
is_rhs_zero	Bool	Является ли правый операнд операции сравнения нулем
loop_depth	Int	Глубина цикловой вложенности базового блока, содержащего операцию сравнения
is_loop_header	Bool	Находится ли операция сравнения в голове цикла
else_edge_flags	Int	Флаги ложной дуги
then_edge_flags	Int	Флаги истиной дуги
dominates_left	Bool	Доминирует ли базовый блок с условием базовый блок ложной дуги
dominates_right	Bool	Доминирует ли базовый блок с условием базовый блок истинной дуги
dom_by_left	Bool	Доминирует ли базовый блок ложной дуги базовый блок с условием
dom_by_right	Bool	Доминирует ли базовый блок истинной дуги базовый блок с условием
current_bb	Struct	Признаки базового блока с условием, структура описана в таблице 4
left_bb	Struct	Признаки ложного базового блока, структура описана в таблице 4
right_bb	Struct	Признаки истинного базового блока, структура описана в таблице 4