«Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В. И. Ульянова (Ленина)»
(СПбГЭТУ «ЛЭТИ»)

Направление	09.03.01 «Информатика и вычислительная техника»
Профиль	«Организация и программирование вычислительных и информационных систем»
Факультет	Компьютерных технологий и информатики
Кафедра	Вычислительной техники

К защите допустить

Заведующий кафедрой

д. т. н., профессор _____ М. С. Куприянов

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Тема: «Алгоритмы оптимизации машинного кода программ для современных архитектур»

Студент _____ А. А. Когутенко

Руководитель
к. т. н., доцент _____ А. А. Пазников

Консультант по
экономическому обоснованию _____ Т. Н. Лебедева

Консультант от кафедры
к. т. н., доцент, с. н. с. _____ И. С. Зуев

Санкт-Петербург

2023 г.

«Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В. И. Ульянова (Ленина)»
(СПбГЭТУ «ЛЭТИ»)

Направление: 09.03.01 «Информатика и вычислительная техника»
Профиль: «Организация и программирование вычислительных и информационных систем»
Факультет компьютерных технологий и информатики
Кафедра вычислительной техники

УТВЕРЖДАЮ
Заведующий кафедрой ВТ
д. т. н., профессор
(М. С. Куприянов)
«___» _____ 2023 г.

ЗАДАНИЕ

на выпускную квалификационную работу

Студент А. А. Когутенко Группа № 9305

1. **Тема**: Алгоритмы оптимизации машинного кода программ для современных архитектур *(утверждена приказом № _____ от _____)*
   Место выполнения ВКР: СПбГЭТУ «ЛЭТИ», кафедра ВТ
2. **Объект исследования:** машинный код программ.
3. **Предмет исследования:** временна́я эффективность исполнения машинного кода на различных архитектурах.
4. **Цель:** разработка новых способов и подходов к оптимизации исполнения программ без исходных текстов, создание платформы для запуска программ с использованием результатов исследований.
5. **Исходные данные:** стандартные методы оптимизации, сведения об особенностях различных микроархитектур.
6. **Содержание:** описание архитектуры современного компьютера и устройства центрального процессора, демонстрация применения техник оптимизации на C/C++ коде и ассемблерных набросках, различные идеи подходов к

оптимизации, реализация платформы для трансформации и исполнения машинного кода в реальном времени.

7. **Технические требования:** исследования должны производиться на различных процессорах с применением утилит профилирования в 64-битной операционной системе на базе GNU/Linux.
8. **Дополнительный раздел:** экономическое обоснование выпускной квалификационной работы.
9. **Результаты:** рассмотрены особенности применения стандартных техник оптимизации на процессорах с различными микроархитектурами, исследованы новые подходы к оптимизации программ, создана платформа для исполнения и трансформации программ в реальном времени. Отчётные материалы: пояснительная записка, реферат, презентация.

Дата выдачи задания
«___» _____ 2023 г.

Дата представления ВКР к защите
«___» _____ 2023 г.

Студент _____ А. А. Когутенко

Руководитель
к. т. н., доцент _____ А. А. Пазников

«Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В. И. Ульянова (Ленина)»
(СПбГЭТУ «ЛЭТИ»)

Направление: 09.03.01 «Информатика и вычислительная техника»
Профиль: «Организация и программирование вычислительных и информационных систем»
Факультет компьютерных технологий и информатики
Кафедра вычислительной техники

УТВЕРЖДАЮ
Заведующий кафедрой ВТ
д. т. н., профессор
(М. С. Куприянов)
«___» _____ 2023 г.

КАЛЕНДАРНЫЙ ПЛАН

выполнения выпускной квалификационной работы

Тема Алгоритмы оптимизации машинного кода программ для современных архитектур

Студент А. А. Когутенко Группа № 9305

№ этапа	Наименование работ	Срок выполнения
1	Обзор исследований и литературы по теме работы	25.03.23–31.03.23
2	Эксперименты с реальными программами	01.04.23–14.04.23
3	Написание и исследование тестовых примеров	15.04.23–30.04.23
4	Разработка новых методов и подходов к оптимизации	01.05.23–14.05.23
5	Реализация и отладка платформы для запуска и трансформации программы в реальном времени	15.05.23–01.06.23
6	Оформление пояснительной записки	02.06.23–10.06.23
7	Представление работы к защите	22.06.23

Студент _____ А. А. Когутенко

Руководитель
к. т. н., доцент _____ А. А. Пазников

СОДЕРЖАНИЕ

РЕФЕРАТ

Пояснительная записка содержит 62 страницы, 5 рисунков, 11 таблиц, 2 приложения.

Выпускная квалификационная работа посвящена исследованию различных методов и подходов к оптимизации машинного кода с целью улучшения производительности программ.

Целью работы является теоретическая разработка и практическая реализация новых способов оптимизации программ без исходного кода.

В рамках работы проведены исследования различных программ на Си и языке ассемблера. Показано, что в некоторых случаях компиляторы могут давать совершенно неоптимальный код, также возможна непредсказуемая работа программы на различных микропроцессорных архитектурах, что так или иначе приводит к идее динамической оптимизации, подразумевающей перекомпиляцию программы непосредственно во время работы.

Из-за того, что существующие средства, осуществляющие запуск программ таким способом, не обеспечивают гарантии малых накладных расходов, был разработан подход, заключающийся в грамотном подборе точек возврата управления с целью трансформации дальнейшего кода. Также рассмотрен метод оптимизации, связанный с апробацией различных вариантов машинного кода, что позволяет выбрать лучший из них, и, следовательно, добиться наилучшей производительности.

По итогам теоретических исследований разработана платформа, позволяющая запустить внешние программы и их оптимизировать их в реальном времени. Полученные наработки могут быть использованы для создания программного обеспечения, производящего запуск приложений для различных архитектур.

ABSTRACT

The final qualifying work is devoted to the study of various methods and approaches to optimizing machine code in order to improve program performance.

The aim of the work is the theoretical development and practical implementation of new ways to optimize programs without source code.

As part of the work, studies of various programs in C and assembly language were carried out. It is shown that in some cases compilers can produce completely suboptimal code, and unpredictable program operation on various microprocessor architectures is also possible, which somehow leads to the idea of dynamic optimization, implying recompilation of the program directly during operation.

Due to the fact that the existing tools that run programs in this way do not provide guarantees of low overhead, an approach has been developed that consists in the competent selection of breakpoints in order to transform further code. The optimization method associated with the testing of various variants of machine code is also considered, which allows you to choose the best of them, and, consequently, to achieve the best performance.

Based on the results of theoretical research, a platform has been developed that allows you to run external programs and optimize them in real time. The obtained developments can be used to create software that runs applications for various architectures.

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ, СОКРАЩЕНИЯ

Инструментирование – модификация программы с целью отслеживания количественных параметров производительности программы. Подразумевает накладные расходы.

Конвейер – способ организации вычислений, применяющийся в современных процессорах с целью повышения производительности. Подразумевает параллельное исполнение нескольких инструкций.

Кеш-линия – небольшой блок данных, являющийся копией данных из основной памяти. На современных архитектурах, как правило, составляет 64 байта (x86-64) или 128 байт (aarch64).

Гетерогенный процессор – процессор, сочетающий в себе вычислительные ядра с разными характеристиками.

Профилирование – сбор информации о работе программы непосредственно во время выполнения.

PGO (profile-guided optimization) – оптимизация с использованием профилирования.

Динамическая оптимизация (оптимизация «на лету») – трансформация и оптимизация программы в реальном времени.

Точка возврата управления – момент выполнения программы, когда происходит возврат управления оптимизатору.

Глубина трансформации – число байт машинного кода, анализируемое оптимизатором для последующей трансформации.

Апробация трансформации – запуск трансформации на небольшом числе итераций цикла для проверки её эффективности.

ВВЕДЕНИЕ

Какими бы ни были быстрыми компьютеры, время их работы, как правило, зависит от объёма обрабатываемых данных. Это означает, что по мере увеличения количества обрабатываемой информации будет спрос на ускорение работы этих компьютеров. Долгое время он компенсировался интенсивным развитием электроники и технологий производства процессоров и микроконтроллеров; больше времени уделялось и оптимизации самих вычислений. Затем процессоры стали достаточно быстрыми для работы с относительно простыми задачами, связанными с обслуживанием запросов потребителей персональных компьютеров – такой расклад привёл к тому, что оптимизации соответствующих программ стали уделять меньше внимания, чаще из экономических соображений. Однако по мере приближения к физическим ограничениям производители процессоров уже не могут обеспечить прежние темпы развития, что приводит к замедлению действия закона Мура, и какой-либо масштабной революции в этом отношении не предвидится. Следовательно, имеет смысл извлечь максимум из уже созданных технологий – путём оптимизаций на уровне программного обеспечения.

На скорость выполнения программы может влиять целый ряд факторов, среди которых:

1. Используемые алгоритмы. Наиболее существенный фактор, так как оказывает непосредственное влияние на время работы программы. Неоптимальный или плохо написанный алгоритм практически всегда приводит к заметному снижению производительности.
2. Используемые структуры данных. Являют собой представление данных в памяти и способ обращения к ним. Так же, как и неоптимальные алгоритмы, могут значительно замедлить вычисления, а в некоторых случаях – ещё и привести к неоправданным накладным расходам по памяти.

3. Эффективность использования памяти. Избыточный расход памяти также может повлиять как на производительность программы, так и на работу всей системы. Помимо того, современные вычислительные устройства имеют многоуровневую иерархию памяти с использованием кэшей, без учёта особенностей которой также можно потерять в эффективности.
4. Параллелизм. Зачастую современные вычислители имеют не один, а несколько исполнительных блоков (ядер) – обычно это позволяет ускорить программу, однако, не все задачи можно ускорить таким образом.
5. Низкоуровневые оптимизации. Подразумевают учёт определённых особенностей и возможностей процессора – оптимизацию использования конвейера, оптимизацию работы с ветвлениями, выравнивание кода и данных, использование инструкций векторизации, учёт свойств микроархитектуры и множество других.

Кроме того, оптимизации могут отличаться по целевому параметру: это может быть время работы, объём используемой памяти, количество потребляемой энергии, частота использования какого-либо ресурса. В данной работе основное внимание уделено именно ускорению, то есть уменьшению времени выполнения.

Среди перечисленных методов ускорения лишь низкоуровневые оптимизации можно автоматизировать; остальные же требуют непосредственного вмешательства разработчика приложения в исходный код. Если же автоматизировать оптимизацию машинного кода без учёта особенностей исходных текстов, работая только с двоичным файлом, то станет возможным ускорить практически все существующие приложения для целевой платформы.

Оптимизации подобного рода актуальны ещё и по той причине, что приложения зачастую оптимизируются не для конкретного процессора, а лишь для целевой архитектуры – то есть аппаратные особенности целевой

платформы учитываются достаточно редко. Также компиляторы не всегда генерируют оптимальный код – в некоторых случаях отладочный код без оптимизаций может оказаться быстрее релизной версии. Всё это даёт достаточные основания для разработки приложения, которое будет инструментировать и менять машинный код непосредственно «на лету», адаптируя его непосредственно для целевой машины.

Цель работы – разработка новых способов и подходов к оптимизации исполнения программ без исходных текстов, создание платформы для запуска программ с использованием результатов исследований.

Объектом работы является машинный код.

Предмет работы – временна́я эффективность исполнения машинного кода на различных архитектурах.

Для достижения поставленной цели были выделены следующие задачи:

1. Исследовать машинный код реальных программ и выяснить возможности их оптимизации.
2. Разработать минимальные экспериментальные образцы, демонстрирующие предпочтительность той или иной оптимизации.
3. На основе полученных ранее результатов исследовать возможности минимизации накладных расходов при инструментации.
4. Разработать алгоритмы применения оптимизаций.
5. Реализовать наработки в платформе для запуска приложений, осуществляющей трансформацию в реальном времени.

Первый раздел содержит общие сведения об устройстве современных компьютеров, организации и особенностях процессоров.

Второй раздел содержит описание разработанных и исследованных экспериментальных программ, а также результаты их запуска на различных вычислителях и анализ полученных данных.

Третий раздел содержит описание существующих подходов к оптимизации, а также предлагаемого способа инструментации с целью снижения накладных расходов. Рассмотрены различные алгоритмы трансформации «на

лету», исследован метод предварительного просчёта оптимизаций. Кроме того, описаны возможные проблемы и методы их решения.

Четвёртый раздел содержит техническое описание реализованной платформы и демонстрацию её работы.

Пятый раздел содержит экономическое обоснование выпускной квалификационной работы.

1 Сведения об устройстве современного компьютера

Современные вычислители являются достаточно сложными устройствами, сочетающими в себе множество компонентов, большинство из которых так или иначе взаимодействуют друг с другом. От этого во многом зависят возможности оптимизации программ. Рассмотрим их подробнее.

1.1 Архитектура компьютера

Под архитектурой компьютера понимают организацию и структуру аппаратных и программных компонентов вычислительной системы. Она отражает способ взаимодействия её компонентов, ключевыми из которых являются процессор, память, устройства ввода-вывода и шина [1].

1.1.1 Общие сведения

Центральный процессор отвечает непосредственно за выполнение программ, управление их выполнением, управление памятью и периферией, а также за обработку прерываний. Таким образом, он является ключевым звеном в вычислительной системе, так как от него зависит работа практически всех остальных компонентов.

Память в компьютерной системе отвечает за хранение данных и машинных инструкций, необходимых для выполнения программы.

Устройства ввода-вывода, как понятно из названия, отвечают за ввод и вывод информации, без чего работа компьютера была бы лишена смысла.

Роль шины заключается в обеспечении коммуникационного канала, по которому передаются данные и сигналы между различными компонентами вычислительной системы. Также она обеспечивает адресацию памяти и служит для координации работы компонентов [1].

Более детальная структурная схема компонентов компьютерной системы представлена на рисунке 1.1 [2].

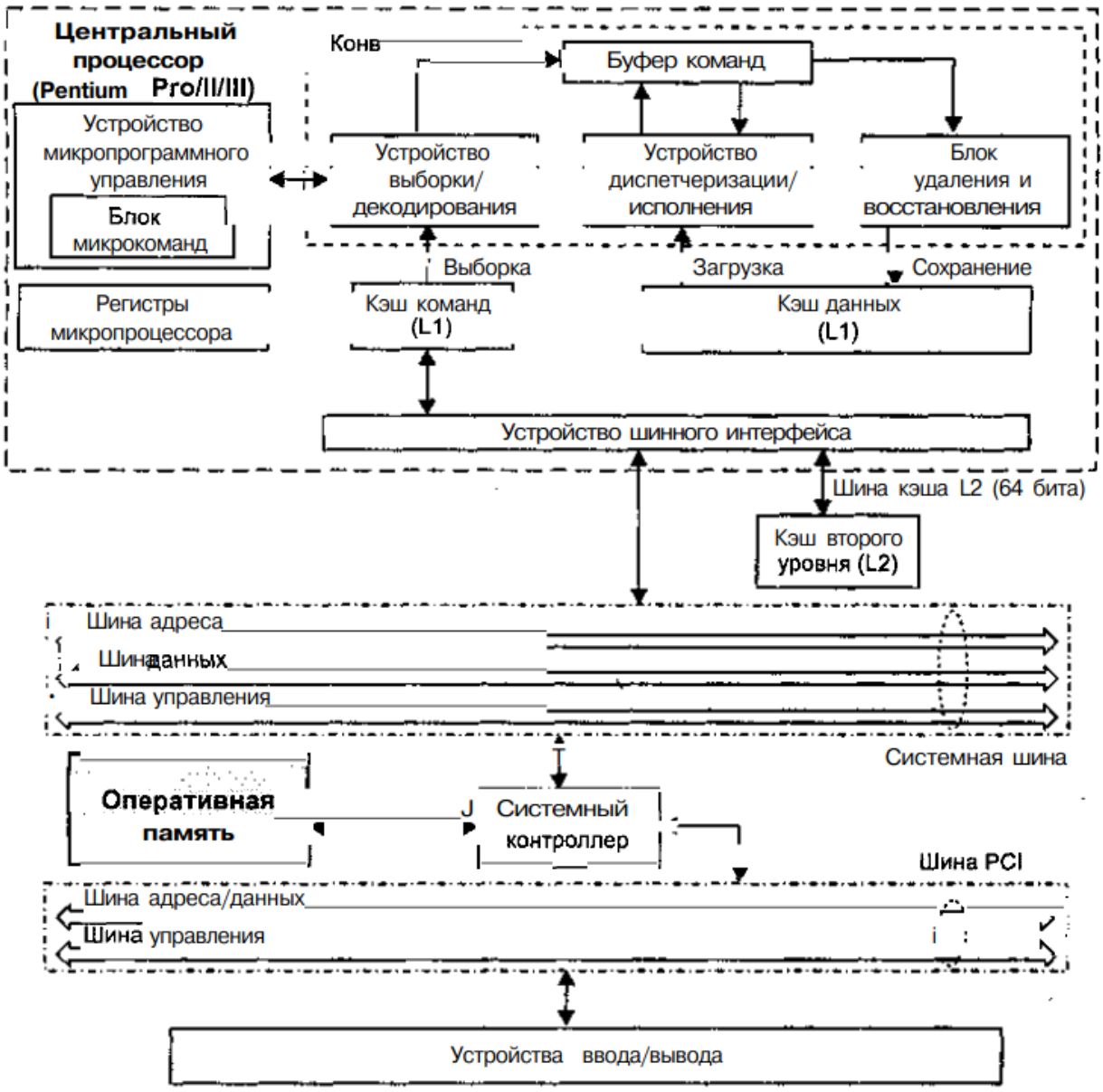

Рисунок 1.1 – Структурная схема компонентов компьютерной системы

Существует два основных подхода к построению компьютерных систем с точки зрения их работы с памятью – архитектура фон Неймана и Гарвардская архитектура.

1.1.2 Архитектура фон Неймана

Одним из вариантов построения компьютерной системы является архитектура фон Неймана. Её отличительной особенностью является то, что у данных и машинного кода программ общая память – то есть в контексте

выполнения программы любую ячейку памяти можно рассматривать и как данные, и как часть программы [3].

Именно архитектура фон Неймана заложена в основу персональных компьютеров, серверов и многих специализированных вычислителей – причиной тому является гибкость и универсальность, удобство программирования, а также относительно низкая стоимость реализации.

Несмотря на весьма широкую распространённость, у такого подхода есть ряд недостатков, среди которых наличие общей шины инструкций и данных, что может привести к снижению производительности, а также проблемы с безопасностью – некорректно работающая программа может серьёзно нарушить работу всей системы (однако, в современных компьютерах, работающих в защищённом режиме, этот недостаток в значительной степени нивелирован).

В дальнейшем будет использована основная особенность данной архитектуры – возможность записать программой данные в память и запустить их выполнение, как программы.

1.1.3 Гарвардская архитектура

Альтернативный подход используется в Гарвардской архитектуре – в ней подразумевается использование отдельного пространства инструкций и отдельного пространства обрабатываемых данных [1]. В ней также предполагается использование отдельных шин для передачи инструкций и передачи данных, что способствует улучшению масштабируемости и позволяет поддерживать высокую производительность.

Гарвардская архитектура нашла применение во встроенных системах и микроконтроллерах, где требуется высокая скорость обработки данных.

1.2 Организация памяти

Чаще всего именно память является узким местом, ограничивающим быстродействие компьютера. В связи с этим разработчики вычислительных

систем стали применять несколько эвристик, уменьшающих зависимость от скорости работы основной памяти и снижающих задержки доступа [4].

1.2.1 Уровни иерархии

В начале 1970-х годов, на заре развития микроэлектроники, основное ограничение эффективности компьютеров заключалось в низкой производительности процессоров. Однако впоследствии развитие вычислительных устройств оказалось гораздо более эффективным, чем устройств хранения данных, из-за чего память стала узким местом, не позволяющим раскрыть потенциал вычислителя полностью. На графике, представленном на рисунке 1.2, показано развитие производительности вычислителей и памяти со временем.

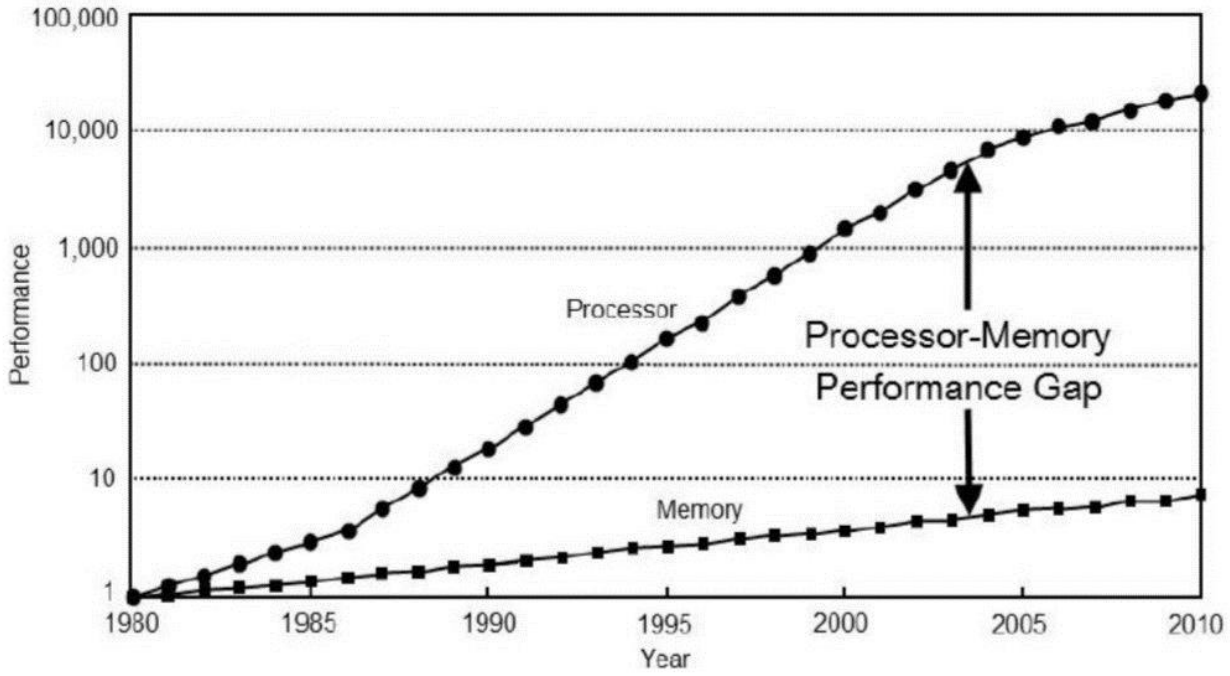

Рисунок 1.2 – Разрыв производительности памяти и процессоров

Чтобы позволить процессорам работать быстрее, не ожидая данных от устройства хранения, стали применять многоуровневую иерархию памяти, где между процессором и основной памятью находится более дорогая и быстрая память, но меньшего объёма. Такую память стали называть кеш-памятью [4].

В современных процессорах выделяют три уровня кеша:

- L1. Самый быстрый и самый маленький кеш. Обеспечивает задержку всего лишь в несколько тактов, и находится наиболее близко к процессору. Обычно имеет размер порядка 32-64 КБ;
- L2. Средний кеш. Возвращает ответ за время, измеряемое десятками тактов процессора. Размер варьируется от 512 КБ до 2 МБ;
- L3. Самый большой, но и самый медленный кеш – но, тем не менее, всё ещё гораздо более быстрый, чем оперативная память. Задержка составляет около двух или трёх сотен тактов. Размер варьируется – в самых дорогих процессорах он может достигать 64 МБ.

Таким образом, важно писать программы так, чтобы код их «горячих» участков умещался в самую быструю кеш-память; в противном случае можно существенно потерять в производительности. Аналогично и с данными – крайне желательно, чтобы они умещались в маленький кеш.

1.2.2 Кеш-линии

Так как программы часто обращаются не только к конкретной ячейке памяти, но и к ячейкам поблизости, разработчики процессоров стали применять ещё одну хитрость, внедрив извлечение не отдельной ячейки, а целого блока ячеек определённой длины [4].

Размер этого блока в старых процессорах на архитектуре x86 составляет 32 байта, в современных – 64 байта. В решениях на архитектуре ARM можно встретить 64 и 128 байт. Выбираемый размер кеш-линии является компромиссом между эффективностью использования пространства и кешей, пропускной способностью памяти и требованиями приложений.

Кеш-линии являются ещё одной важной особенностью, которую необходимо учитывать при создании программ.

1.2.3 Виртуальная память

Виртуальная память являет собой концепцию, согласно которой операционная система и программы могут использовать больший объём памяти, чем есть физической памяти.

Вся физическая память абстрактно делится на страницы определённого размера (обычно 4 КБ). Когда процесс запрашивает память, операционная система сопоставляет для него определённые страницы физической памяти страницам виртуальной. Таким образом, у процесса образуется собственное виртуальное адресное пространство. Когда он обращается к памяти, специальное устройство (MMU – Memory Management Unit) преобразует виртуальный адрес в физический [4].

Главной особенностью здесь является то, что пока к определённой странице виртуальной памяти процесс не обратится напрямую, для этого участка не будет выделена физическая страница – получается своего рода «ленивое» выделение памяти, за счёт которого и становится возможным иметь гораздо большее адресное пространство, чем размер физической оперативной памяти.

Кроме того, страницам виртуальной памяти можно задавать различные атрибуты – в частности, флаги чтения, записи и исполнения. Это позволяет обеспечить защиту памяти от нежелательных эффектов в случае нештатной работы программы.

В современных операционных системах на базе ядра GNU/Linux манипуляции виртуальной памятью возможны с помощью использования системного вызова `mmap()`.

1.3 Особенности современных процессоров

В погоне за производительностью разработчики процессоров придумали целый ряд техник, позволяющих существенно ускорить выполнение программ. Ниже представлено описание некоторых концепций, работающих на уровне процессора.

1.3.1 Конвейер

Представляет собой технику, разбивающую выполнение команд процессора на несколько стадий, тем самым позволяя выполнять одновременно множество команд, каждая из которых находится на разной стадии исполнения [5].

Основные стадии конвейера в процессоре:

1. IF (Instruction Fetch) – загрузка команды.
2. ID (Instruction Decode) – декодирование, определение необходимых операций и операндов.
3. OF (Operand Fetch) – извлечение операндов команды из памяти или регистров.
4. EX (Execute) – выполнение команды.
5. MEM (Memory Access) – при условии взаимодействия команды с памятью осуществляется чтение или запись данных.
6. WB (Write Back) – запись результата выполнения.

Однако, следует заметить, что в некоторых случаях возможны простои конвейера – например, в случаях, когда работа очередной команды зависит от результата предыдущей. Таким образом, следует избегать зависимости по данным при написании программ (если это возможно).

1.3.2 Суперскалярность

Эта концепция позволяет одновременно выполнять несколько инструкций за один такт работы процессора. Так как разные инструкции требуют разные исполнительные блоки, можно «раздать» эти блоки различным инструкциям, таким образом организовав параллельное вычисление [5].

1.3.3 Предсказание ветвлений

Реализует идею вычислений, строящихся на статистическом предположении, по какой ветви пойдёт выполнение условной инструкции. При таком подходе процессор исполняет инструкции наперёд, ещё не зная результат выполнения условия. В современных процессорах предсказатели ветвлений достаточно хороши, чтобы обеспечивать 95% верных переходов на практически любом коде.

Однако, проблема в том, что если процессор «не угадал» переход, то пенальти – сброс конвейера, являющийся очень затратным по времени, поэтому ветвлений всё же лучше избегать. Это можно сравнить с мчащимся

по железной дороге поездом – если машинист поймёт, что поехал не по той ветви дороги с ручной стрелкой, ему придётся вернуться, переключить стрелку в необходимое положение, и снова набирать скорость.

Рисунок 1.3 – Механизм предсказания ветвлений можно сравнить с ручной железнодорожной стрелкой

Предсказатели ветвлений являются предметом постоянного совершенствования, начиная самими механизмами предсказания, заканчивая техническими параметрами предсказателя – размером истории, ассоциативностью, числом уровней.

Однако возможны случаи, в которых доля неудачных предсказаний всегда будет высокой – например, когда при определении ветви нет никакой закономерности.

Для частичного нивелирования проблемы неудачных предсказаний разработчики процессоров ввели команды условного копирования CMOV и

FCMOV, выполняющие копирование в регистр только в случае, если установлен определённый флаг [5][6][7]. На современных процессорах CMOV для целых чисел выполняется всего за один такт [8].

1.3.4 Внеочерёдное исполнение команд

Для улучшения параллелизма была введена техника, осуществляющая прозрачную перестановку независимых машинных инструкций (OoO – Out of Order Execution) [5]. Это позволяет эффективнее загрузить блоки процессора, что улучшает общую производительность процессора.

Однако, подобный подход привёл к тому, что в процессорах появились уязвимости, например, Meltdown и Spectre, поэтому преимущества OoO в некоторых случаях могут оказаться существенно сниженными из-за патчей микрокода.

1.3.5 Векторные расширения

Также ещё носят название векторизации. Реализуют идею SIMD (Single Instruction – Multiple Data), подразумевающую однотипную обработку нескольких ячеек данных за одну инструкцию – например, 256-битные расширения AVX2 позволяют проводить манипуляции одновременно с 8 числами с плавающей запятой одинарной точности [7]. Один из наиболее перспективных методов ускорения, так как весьма часто проблемные с точки зрения производительности участки кода обрабатывают однотипные данные, представимые в виде массива.

1.3.6 Другие особенности

Среди прочих особенностей можно выделить, например, величину накладных расходов к невыровненным ячейкам памяти. На процессоре Intel Pentium 4 тест, осуществляющий невыровненный доступ в определённом регионе памяти, оказался на 64% медленнее, чем выровненный. Однако это достаточно старый процессор, и в современных вычислителях этот эффект либо не наблюдается, либо крайне незначителен – тестирование на AMD

Ryzen 5 5600H и AMD Ryzen 3 3200U не выявило отличий во времени выполнениях при разных способах обращения, а в случае Intel Core i5-10400 замедление оказалось пренебрежимо малым. Это позволяет, например, делать более эффективную упаковку структур, исключив из них скрытые поля для выравнивания – это уплотнит данные, практически гарантированно дав прирост в производительности, так как больше нужных данных будет извлекаться за кеш-линию и больше полезных данных окажется в кеше.

Другой весьма интересный эффект может возникнуть, если «горячий» цикл будет небольшим, но окажется на границах кеш-линий – в таком случае эффективность программы тоже может заметно ухудшиться. Решается подобная проблема с помощью выравнивания таких блоков кода инструкциями, которые ничего не делают – NOP (No Operation).

Наконец, каждая отдельная машина имеет собственные отличия, конфигурацией аппаратной части – частотой памяти, шины и др., и заканчивающиеся свойствами кристалла (например, одни процессоры могут лучше работать в режиме «Turbo», а другие хуже). Кроме того, индустрия движется к созданию гетерогенных процессоров, сочетающих в себе ядра с различным техпроцессом и микроархитектурой – например, актуальные процессоры Intel Alder Lake и Raptor Lake сочетают в себе производительные и энергоэффективные ядра [9][10]. Также оптимизации могут производиться на уровне микроопераций [11]. Всё это означает, что при желании добиться максимальной производительности на конкретном компьютере необходимо учитывать его специфику.

2 Моделирование методов микроархитектурной оптимизации

Для исследования особенностей работы процессоров с различными микроархитектурами составлено несколько экспериментальных примеров, демонстрирующих преимущество той или иной оптимизации, а также отражающих эффективность оптимизации с помощью компилятора (для примеров на языке Си).

Полные тексты всех рассмотренных программ и скрипт для их запуска представлены в приложении А.

2.1 Выравнивание циклов

Как уже упоминалось в части 1.3.6, в некоторых случаях, когда тело цикла попадает в разные кеш-линии, производительность программы может существенно снизиться. С целью проверки этого явления была составлена небольшая программа на языке ассемблера, не делающая ничего существенного (основной цикл заполнен инструкциями No Operation, занимающими один байт), но демонстрирующая снижение производительности.

Для гибкости была реализована возможность выравнивать цикл с помощью внешнего параметра ALIGNMENT, а также регулировка числа NOP в теле цикла параметром INNER_NOPS. Ключевой фрагмент выглядит следующим образом:

```
mov rcx, ITERATIONS
times ALIGNMENT - ($ - _start) nop
loop_begin:
    times INNER_NOPS nop
    loop loop_begin
```

При запуске двоичного файла, скомпилированного с различными опциями на производительном ядре процессора Intel Core i7-13700H (поколение Raptor Lake [10]) на частоте 3.8 ГГц были получены результаты, представленные на рисунке 2.1.

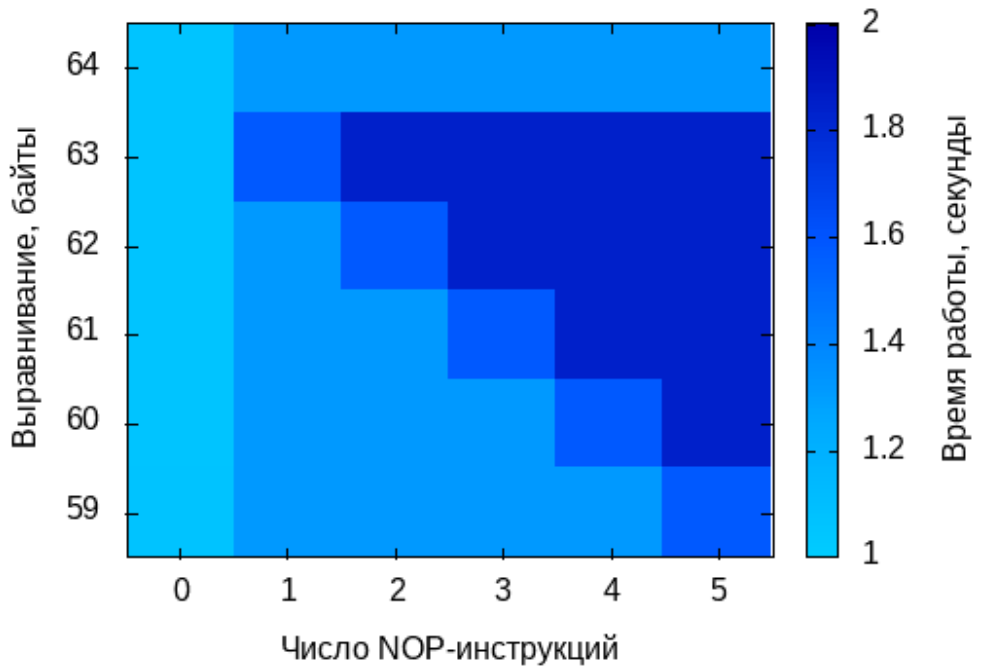

Рисунок 2.1 – Результаты запуска на производительном ядре процессора i7-13700H

Результаты запуска той же программы на энергоэффективном ядре того же процессора дали иные результаты, представленные на рисунке 2.2.

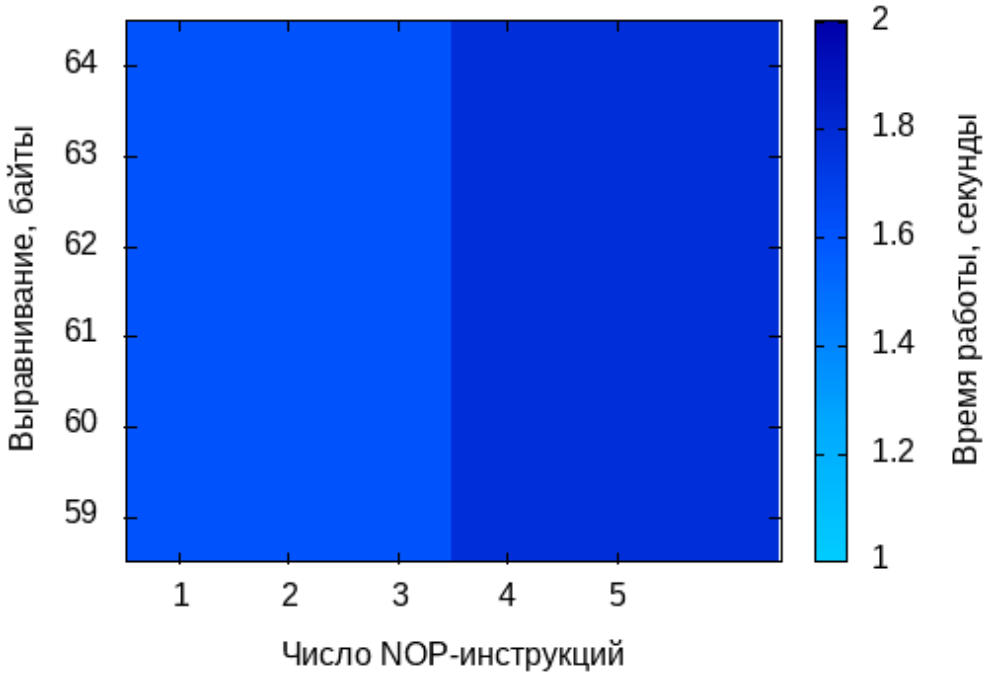

Рисунок 2.2 – Результаты запуска на энергоэффективном ядре того же процессора

Рассмотрим случай первого запуска (P-ядро). Можно заметить, что время работы одной и той же программы никак не зависит от выравнивания при отсутствии NOP-инструкций; при их наличии картина меняется – если при выравнивании по границе в 64 байта программа работает примерно за 1,326 с, то отрицательное смещение всего в один байт приводит к времени работы за 1,852 с, то есть теряется почти 29% производительности.

Во втором же случае (E-ядро) производительность никак не зависит от выравнивания, варьируясь только в зависимости от числа NOP-инструкций.

Для проверки этой особенности, а также того, как с ней справляются существующие программы, осуществляющие модификацию кода в реальном времени (QEMU [12], DynamoRIO и Valgrind [13, 14]), был взят случай с пятью инструкциями NOP и выравниванием по границе в 63 байта (худший) и по 64 байта (лучший). Результаты экспериментов представлены в таблице 2.1.

Таблица 2.1 – Результаты тестирования выравнивания на 8 различных процессорах

	Оригинал	**QEMU**	**DynamoRIO**	**Valgrind**	**Лучший**
Intel Core i7-13700H, P-Core 3.8 GHz	1,852	0,332	1,854	1,624	1,326
Intel Core i7-13700H, E-Core 3.1 GHz	1,785	0,982	2,595	2,365	1,784
AMD Ryzen 5 5600H 3.3 GHz	0,613	0,619	0,624	1,921	0,307
Intel Core i5-10400 2.9 GHz	2,427	20,405	2,400	2,215	1,729
Intel Core i5-9400F 2.9 GHz	2,421	20,069	2,445	2,422	1,731
AMD Ryzen 3 3200U 2.7 GHz	0,807	20,473	0,849	3,995	0,389
Intel Celeron J4105 1.5 GHz	7,387	4,083	8,086	10,791	7,393
Intel Pentium 3558U 1.7 GHz	4,139	38,969	4,170	4,972	2,953

Можно заметить, что лишь на двух процессорах выравнивание не дало эффекта – это всё тот же Intel Core i7-13700H, а также Intel Celeron J4105. В остальных случаях ускорение варьировалось – от примерно 40% для производительного ядра i7-13700H до 107% в случае AMD Ryzen 3 3200U. Такое отличие в результатах можно объяснить особенностями микроархитектуры различных ядер.

Кроме того, достойны внимания результаты запуска с помощью инструментирующих программ. Так, QEMU в одних случаях сократил время работы программы, применив оптимизацию цикла (однако, странно, что он не убрал его вовсе, так как цикл не выполняет какой-либо полезной работы), а в других, напротив, существенно увеличил время работы, что практически наверняка связано с устаревшей версией эмулятора, использовавшегося на машинах.

DynamoRIO, известный как инструмент для инструментации с помощью собственных проходов, в большинстве случаев давал сопоставимое, но всё же большее, чем у оригинала, время работы.

Valgrind также применяет некоторые оптимизации для ускорения программы, однако они оказываются несущественными; к тому же, время работы программы, запущенной через Valgrind, порой существенно превышало время работы оригинальной версии.

Таким образом, уже сейчас можно сделать вывод, что ни один из рассмотренных инструментов не гарантирует того, что накладные расходы окажутся минимальными.

2.2 Разворачивание циклов

Одной из популярных техник при оптимизации циклов является их разворачивание. В таком случае компилятору приходится менять стандартный инкремент счётчика на команду прибавления определённого числа. Однако, можно сделать прибавление другим способом, выполнив нужное число инкрементов с помощью инструкции INC.

Реализуем эти подходы в отдельном примере, осуществляющем суммирование чисел массива с развёрнутым циклом. Основной фрагмент ассемблерного листинга:

```
iterate:
    add rax, [array + rdx * 8 +  0]
    add rax, [array + rdx * 8 +  8]
    add rax, [array + rdx * 8 + 16]
```

```
add rax, [array + rdx * 8 + 24]

%ifdef INC
    inc rdx
    inc rdx
    inc rdx
    inc rdx
%else
    add rdx, 4
%endif

cmp rdx, ARRAY_SIZE
jne iterate
```

Если же при компиляции примера будет установлен параметр INC, для прибавления будет использован инкремент, в противном же случае суммирование будет осуществляться инструкцией ADD.

Результаты запуска программы на различных процессорах представлены в таблице 2.2.

Таблица 2.2 – Результаты тестирования различных вариантов сложения

	ADD	**QEMU**	**DynamoRIO**	**Valgrind**	**4 x INC**
Intel Core i7-13700H, P-Core 3.8 GHz	0,653	0,979	0,708	3,077	0,654
Intel Core i7-13700H, E-Core 3.1 GHz	1,465	1,430	1,598	5,459	1,258
AMD Ryzen 5 5600H 3.3 GHz	0,596	1,095	0,603	3,894	0,605
Intel Core i5-10400 2.9 GHz	0,902	3,889	1,014	5,486	0,965
Intel Core i5-9400F 2.9 GHz	1,217	4,101	1,310	6,413	1,271
AMD Ryzen 3 3200U 2.7 GHz	0,892	3,884	0,956	5,595	0,934
Intel Celeron J4105 1.5 GHz	3,342	5,009	3,450	17,799	3,743
Intel Pentium 3558U 1.7 GHz	1,513	7,089	1,611	11,398	1,572

Взглянув на полученные результаты, можно прийти к выводу, что классический способ прибавления числа быстрее практически на всех процес-

сорах, кроме энергоэффективного ядра i7-13700H, где способ с инкрементированием даёт существенный выигрыш в производительности. На производительном ядре эффективность обоих способов оказалась примерно одинаковой.

Эксперимент может оказаться более наглядным, если избавиться от других операций и свести тело цикла к одной лишь операции суммирования.

Лишь в одном случае запуск с помощью инструмента оказался быстрее (QEMU, энергоэффективное ядро i7-13700H), и всё равно оказался более медленным, чем версия с инкрементированием.

2.3 Предсказание ветвлений и CMOV

Следующий пример призван показать, насколько эффективной может быть инструкция условного копирования. Программа генерирует случайные числа с помощью метода Xorshift32 (используется как быстрый и лёгкий в написании генератор псевдослучайных чисел), и подсчитывает количество тех, что больше определённого порога. Порог подобран таким образом, чтобы в результате подсчёта получилась примерно половина от всего числа сгенерированных чисел. Рассмотрены два варианта реализации:

1. Стандартный, с использованием условного оператора.
2. Та же программа, но вместо условного оператора используется инструкция CMOV.

Основной фрагмент:

```
iterate:
    ; Генерация числа с помощью Xorshift32
    mov rbx, rax
    shl rax, 13
    xor rax, rbx

    mov rbx, rax
    shr rax, 7
    xor rax, rbx
```

```
    mov rbx, rax
    shl rax, 17
    xor rax, rbx

%ifdef BRANCHLESS
    ; Вариант без ветвлений
    xor rdx, rdx      ; Обнуляем регистр, который позднее прибавим
    cmp rax, rbx      ; Сравниваем
    cmovg rdx, rdi    ; Устанавливаем аккумулятор в 1, если истинно
    add rsi, rdx      ; Прибавляем значение аккумулятора
%else
    ; Стандартная реализация
    cmp rax, rbx      ; Сравниваем
    jle continue      ; Игнорируем инкремент, если число меньше
    inc rsi           ; Инкрементируем счётчик
    continue:
%endif

    loop iterate
```

Результаты запуска программы на различных процессорах представлены в таблице 2.3.

Таблица 2.3 – Результаты прогона эксперимента с ветвлениями

	Оригинал	**QEMU**	**DynamoRIO**	**Valgrind**	**CMOV**
Intel Core i7-13700H, P-Core 3.8 GHz	0.594	0.625	0.615	0.962	0.212
Intel Core i7-13700H, E-Core 3.1 GHz	0.609	0.904	0.676	1.476	0.228
AMD Ryzen 5 5600H 3.3 GHz	0.460	0.708	0.479	1.158	0.207
Intel Core i5-10400 2.9 GHz	0.675	2.533	0.738	1.362	0.284
Intel Core i5-9400F 2.9 GHz	0.673	2.738	0.769	1.570	0.277
AMD Ryzen 3 3200U 2.7 GHz	0.633	3.928	0.685	1.773	0.272
Intel Celeron J4105 1.5 GHz	1.438	2.129	1.551	4.222	0.941
Intel Pentium 3558U 1.7 GHz	1.189	5.291	1.272	2.668	0.591

На всех без исключения процессорах реализация, использующая CMOV, оказалась существенно быстрее – выигрыш в производительности составил от 1,5 до 2.67 раз.

Использование инструментирующих программ не дало эффекта.

2.4 Гипотеза Коллатца

Одна из наиболее интересных проблем современной математики – гипотеза Коллатца – интересна тем, что для проверки числа на соответствие гипотезе необходимо использовать условную конструкцию. Так, если на определённой итерации число чётное, то его необходимо разделить на 2; в противном случае умножить на 3 и прибавить единицу. Согласно гипотезе, такими действиями рано или поздно мы получим единицу.

Реализуем простую программу на Си, которая будет проверять соответствие гипотезе чисел от 1 до 10 миллионов:

```
for (uint64_t i = 1; i <= 10000000; ++i)
{
    uint64_t x = i;
    uint64_t iterations = 0;

    while (x != 1 && iterations++ <= 1000)
        if (x % 2 == 0)
            x /= 2;
        else
            x = x * 3 + 1;

    if (iterations == 1000)
        return 1;
}
```

Если же в течение 1000 применений описанной в гипотезе операции получить единицу не удаётся, контрпример считается найденным.

Результаты компиляции различными тулчейнами (GCC, Clang, Intel C/C++ Compiler) и запуска на Intel Core i7-13700H представлены в таблице 2.4.

Таблица 2.4 – Результаты запуска двоичных файлов, собранных различными способами

	Нативно	**QEMU**	**DynamoRIO**	**Valgrind**
GCC 13.1.1 O2	2.344	4.406	2.860	10.517
Clang 15.0.6 O2	1.589	2.253	1.518	9.070
ICC 2023.1.0 O2	1.568	2.103	1.635	8.794
GCC 13.1.1 Ofast march=native	2.345	4.371	2.861	10.531
Clang 15.0.6 Ofast march=native	1.524	2.931	1.574	9.369
ICC 2023.1.0 Ofast march=native	1.529	2.930	1.574	9.376

Если взглянуть на результаты GCC, можно обнаружить, что исполняемый файл, полученный с его помощью, работает за существенно большее время, чем в случае ICC и Clang. При дизассемблировании выясняется, что первый совершенно не использовал инструкцию CMOV, в то время как остальные два применяли агрессивное разворачивание циклов и вышеупомянутую команду.

Ни QEMU, ни DynamoRIO, ни Valgrind не смогли положительным образом повлиять на создавшуюся ситуацию.

2.5 Зависимость от данных

Нередко возникают ситуации, когда в теле цикла фигурируют вычисления с числами, зависящими от входных данных программы.

Предположим, есть программа, генерирующая псевдослучайные числа по определённому шаблону, и производящая деление каждого из них на число, определяемое аргументом командной строки. Здесь возможны два случая:

1. Число не представляет какой-либо возможности оптимизации.
2. Число позволяет ускорить вычисления, заменив деление на более быструю операцию – например, если число будет степенью двойки, то его можно заменить на битовый сдвиг.

При классическом подходе во втором случае будет использоваться машинный код для первого. Но что, если предположить, что программа сможет сама оптимизировать процесс и вычислить результат быстрее?

Реализуем подобную программу на языке Си:

```
#ifndef DIV
uint32_t div = strtoul(argv[1], NULL, 10);
#endif

for (size_t i = 0; i < ITEMS_COUNT; ++i)
#ifdef DIV
    sum += xorshift32() / DIV;
#else
    sum += xorshift32() / div;
#endif
```

Если установлен дефайн DIV, будет использовано его значение; в противном случае при делении будет применяться число, заданное аргументом командной строки.

Результаты компиляции с различными опциями и запуска на производительном ядре i7-13700H с числом 16 на входе представлены в таблице 2.5.

Таблица 2.5 – Результаты запуска двух вариантов с числом 16

	Нативно	**QEMU**	**DynamoRIO**	**Valgrind**
GCC 13.1.1	1.694	2.721	1.706	6.454
Clang 15.0.6	1.691	3.056	1.706	6.333
ICC 2023.1.0	1.653	3.616	1.673	5.480
GCC 13.1.1 const	1.745	2.657	1.757	4.563
Clang 15.0.6 const	1.643	2.315	1.644	4.017
ICC 2023.1.0 const	1.642	2.313	1.648	3.713

Полученные результаты примечательны, во-первых, тем, что ICC оптимизировал таким образом, что время работы практически сравнялось с константной версией – это можно объяснить более детальной информированностью компилятора об особенностях процессоров Intel.

Второй момент заключается в том, что GCC при подстановке константы генерирует более медленный код, хотя, казалось бы, ничто не мешает заменить константу на битовый сдвиг и тем самым ускорить работу программы. При этом, если заменить обычную операцию деления на деление с остатком, результат становится более предсказуемым.

В данном тесте инструменты, осуществляющие инструментацию, показали более малые накладные расходы, особенно DynamoRIO; однако, ни в одном из случаев программа не стала быстрее оригинальной.

2.6 Сортировка методом пузырька

Из более приближённых к реальным задачам можно назвать пузырьковую сортировку, работающую за квадратичное время. Несмотря на то, что на практике она применяется больше для образовательных целей, она также являет собой прекрасный алгоритм для проверки возможностей оптимизации компилятора.

Напишем классическую её реализацию на языке Си:

```
for (size_t i = 1; i < ITEMS_COUNT; ++i)
    for (size_t j = 1; j <= ITEMS_COUNT - i; ++j)
        if (array[j - 1] > array[j])
        {
            int temp = array[j - 1];
            array[j - 1] = array[j];
            array[j] = temp;
        }
```

Скомпилируем получившуюся сортировку без оптимизаций, с базовым набором оптимизаций (O2) и теми же оптимизациями, но без векторизации,

запустим. Результаты для трёх уже взятых ранее компиляторов представлены в таблице 2.6.

Таблица 2.6 – Результат запуска пузырьковой сортировки с оптимизациями и без них

	Нативно	**QEMU**	**DynamoRIO**	**Valgrind**
GCC 13.1.1	3.997	9.734	4.382	27.209
Clang 15.0.6	4.066	8.161	4.381	28.448
ICC 2023.1.0	3.943	7.723	4.280	28.551
GCC 13.1.1 O2	5.790	10.828	5.694	35.434
Clang 15.0.6 O2	2.217	4.688	2.414	10.330
ICC 2023.1.0 O2	2.610	4.066	2.613	10.682
GCC 13.1.1 O2 fno-tree-vectorize	3.019	4.443	3.295	11.927
Clang 15.0.6 O2 fno-tree-vectorize	2.221	4.686	2.418	10.290
ICC 2023.1.0 O2 fno-tree-vectorize	2.613	4.065	2.613	10.696

Равно как и в предыдущем примере, в полученных результатах есть несколько моментов, достойных внимания:

1. Версия сортировки без оптимизаций в случае GCC работает быстрее, чем с указанием параметра O2.
2. При отключении векторизации сортировка от GCC работает лучше, чем при O0, но всё ещё медленнее, чем в случае Clang и ICC.
3. ICC проигрывает Clang, хотя рассчитан на более качественную оптимизацию кода для процессоров фирмы Intel.

Анализ кода, сгенерированного различными компиляторами, выявил, что GCC не вставил ни единой команды CMOV – как результат, была сгенерирована версия с классическими условными переходами, что дало негативный эффект из-за плохой работы предсказателя ветвлений (об этом же говорят и результаты профилирования с помощью утилиты perf [15]). Также замечено, что Clang и ICC применяют крайне агрессивное разворачивание циклов, тем самым делая использование конвейера процессора более эффективным, чего нельзя сказать о GCC.

QEMU, DynamoRIO и Valgrind снова не удалось минимизировать накладные расходы.

Помимо версии сортировки на языке Си был также реализован её вариант на языке ассемблера с применением двух инструкций CMOV (в коде, генерируемом Clang, CMOV встречается 6 раз):

```
outer_loop:

    mov r9, 1       ; Храним значение переменной j

    inner_loop:

        mov edx, [rdi + r9 * 4]         ; arr[j]
        mov eax, [rdi + r9 * 4 - 4]     ; arr[j - 1]

        mov r10d, edx       ; Храним значение одной из переменных
                            ; на случай, если придётся делать обмен

        cmp eax, edx        ; Сравниваем
        cmovg edx, eax      ; Меняем местами
        cmovg eax, r10d

        mov [rdi + r9 * 4], edx       ; Пишем arr[j - 1] в arr[j]
        mov [rdi + r9 * 4 - 4], eax   ; Пишем arr[j] в arr[j - 1]

        inc r9            ; Инкрементируем j

        cmp r9, r8        ; Сравниваем с предельным значением
        jl inner_loop     ; Если не достигли предела, идём в начало

    dec r8            ; Уменьшаем внешний счётчик

    test r8, r8       ; Проверяем на ноль
    jnz outer_loop    ; Если не ноль, переходим в начало внешнего цикла
```

Отдельно были протестированы две реализации сортировки – скомпилированная Clang с параметром Ofast (максимальная оптимизация), и ассемблерный вариант, представленный выше. Результаты запуска на различных процессорах показаны в таблице 2.7.

Таблица 2.7 – Результаты запуска лучшей компиляторной версии и ассемблерной

	clang Ofast	**nasm**
Intel Core i7-13700H, P-Core 3.8 GHz	2.097	2.757
Intel Core i7-13700H, E-Core 3.1 GHz	2.873	1.223
AMD Ryzen 5 5600H 3.3 GHz	2.193	0.876
Intel Core i5-10400 2.9 GHz	3.390	2.587
Intel Core i5-9400F 2.9 GHz	3.800	2.607
AMD Ryzen 3 3200U 2.7 GHz	3.751	4.985
Intel Celeron J4105 1.5 GHz	6.558	7.559
Intel Pentium 3558U 1.7 GHz	5.485	6.754

Результаты получились крайне любопытными:

1. Ассемблерный вариант сортировки оказался не только быстрее компиляторного на энергоэффективном ядре i7-13700H, но и показал 70-процентное преимущество над лучшей реализацией, запускавшейся на производительных ядрах!
2. Можно заметить, что ассемблерная реализация работает быстрее на относительно новых процессорах, представленных в начале списка; исключением является производительное ядро того же i7-13700H, который является самым актуальным процессором Intel, доступным на рынке на момент проведения исследований.

Ошибка в исследованиях маловероятна, так как результаты работы сортировок сверялись путём получения MD5-суммы всех байт, подававшихся в поток вывода.

2.7 Анализ результатов

Запуск рассмотренных экспериментов на различных процессорах приводит к нескольким важным выводам:

1. Не всегда компиляторы генерируют оптимальный код. Даже при компиляции строго для определённого процессора.
2. Не всегда скорость работы той или иной программы на целевой микроархитектуре является предсказуемой.
3. Ни один из рассмотренных инструментов, выполняющих модификацию кода «на лету», не даёт гарантии минимальных накладных расходов.
4. Программа может быть адаптирована в соответствии с входными данными и работать быстрее.
5. Профилирование не всегда приводит к улучшению результатов.

Изменить ситуацию в положительную сторону можно, создав инструмент, который будет оптимизировать исполняемый код в реальном времени, запуская различные его вариации, адаптируя в соответствии с состоянием регистров и выбирая лучшие.

Так как появляется всё больше и больше гетерогенных процессоров, сочетающих в себе ядра с различной микроархитектурой, создание подобного приложения имеет ещё больший смысл.

Помимо того, такой инструмент мог бы оптимизировать программы прозрачно, не требуя вмешательства пользователя или разработчика, что особенно актуально для старых программ, не использующих современные расширения процессоров.

Главной проблемой при создании такого оптимизатора является правильный выбор точек возврата управления и минимизация времени трансформации. Некоторые идеи, призванные их решить, описаны в разделе 3.

3 Алгоритмы и способы оптимизации

Существует множество методов оптимизации кода программ, каждый из которых имеет свою область применимости.

3.1 Подходы

Различные подходы позволяют улучшить знание особенностей работы программы, что также способствует генерации кода с лучшим быстродействием.

3.1.1 Статическая оптимизация

Классический метод оптимизации, заключающийся в анализе и трансформации программы до её запуска. Среди оптимизаций подобного рода можно выделить:

1. Устранение избыточных вычислений.
2. Оптимизацию использования регистров.
3. Улучшение работы с кеш-памятью.
4. Векторизацию.

Именно статическая оптимизация вносит большую долю в ускорение оригинальной программы.

3.1.2 Статическая оптимизация с предпросчётом

При оптимизации непосредственно машинного кода известно ещё меньше, чем при компиляции исходного. Однако, при возможности запустить оптимизатор на целевой машине можно выполнить определённый предпросчёт, «изучив» подобным образом компьютер, а затем применив полученные сведения при оптимизации реальной программы. Схожий подход используется в платформе .NET – при обновлении она может некоторое время загружать компьютер интенсивными предварительными вычислениями.

3.1.3 Профилирование

При компиляции исходного кода многие оптимизации компилятор выполняет, основываясь лишь на предположениях – зачастую он не знает,

какими будут возможные значения той или иной переменной, по какой ветви будет чаще всего выполняться условный код, как будут использоваться циклы, и множество других вещей, известных во время непосредственной работы программы.

Одним из способов дать компилятору больше информации является запуск особым образом модифицированной (инструментированной) версии исполняемого файла. Например, GCC позволяет выполнить это следующим образом:

```
gcc -O2 -fprofile-generate input1.c input2.c ...
./a.out
gcc -O2 -fprofile-use input1.c input2.c ...
```

В результате первой компиляции появится инструментированный файл, вторая же воспользуется результатами запуска инструментированного файла, чтобы осуществить оптимизацию лучше.

Однако, это всё ещё не идеальное решение проблемы, так как входные данные программы могут измениться, что может привести к снижению эффективности оптимизаций. Альтернативный подход, используемый в средах выполнения для некоторых языков программирования – JIT-компиляция (Just in Time, «на лету»).

3.1.4 Оптимизация «на лету»

Нашедший применение в языках вроде Java и Python, а также в платформе .NET, метод генерации исполняемого кода «на лету» позволяет осуществлять сбор профилирующей информации непосредственно во время работы программы, и тогда же, при необходимости, её трансформировать.

Преимуществом данного подхода является возможность нивелировать нехватку знаний о программе при статической компиляции, недостатком же являются накладные расходы на сбор статистики и трансформацию кода.

В данной работе будут рассмотрены методы снижения негативного влияния профилирования и трансформаций, благодаря чему предполагается создание более эффективного оптимизатора.

3.2 Гибридный способ

В данной работе предлагается создать среду выполнения, которая объединит три различных подхода – статическую оптимизацию с предпросчётом, профилирование и JIT, применительно непосредственно к исполняемым файлам.

В пользу исследований именно в этой области можно назвать несколько причин:

- применение альтернативных способов расстановки точек анализа и трансформации, позволяющих уменьшить накладные расходы от JIT;
- ориентированность оптимизаций на конкретный компьютер, что позволяет лучше раскрыть его возможности;
- возможность оптимизации старых программ, для которых не осталось исходного кода или сборка которого проблематична, за счёт использования определённых эвристик и машинных инструкций, появившихся в новых процессорах;
- относительно неглубокая изученность темы – использование предварительного просчёта практически не встречается;
- до сих пор не существует инструмента, который бы оптимизировал код подобным образом.

Подход подобного рода вполне можно сочетать с различными существующими методами оптимизации. Так, авторы статьи [16] предлагают улучшение алгоритмов поиска блоков трансляции и машинно-независимые трансформации. В [17] рассмотрен подход, при котором оптимизация выполняется в отдельном потоке, который также можно взять на вооружение, однако он не решает проблему частых остановов. В работе [18] предлагаются ценные идеи по использованию PGO (оптимизации с использованием профиля).

Различные способы ускорения в контексте DynamoRIO и движка V8, используемого для запуска и оптимизации JavaScript предложены в [19]. Авторы [20] делают акцент на максимизации переиспользования ранее сгенерированного кода, что также имеет место в контексте предлагаемого способа оптимизации. Достаточно давний, но всё ещё актуальный труд [21] демонстрирует пример создания среды динамической оптимизации, работающей прозрачно для пользователя. Идеи уменьшения накладных расходов представлены в контексте платформы Odin [22]. В [23] можно найти описание системы динамической оптимизации Mojo, некогда разрабатывавшейся компанией Microsoft.

Интересные идеи, касающиеся использования аппаратных счётчиков производительности, рассмотрены в [24]. Также в одном из трудов разработчиков GCC представлены теоретические наработки по PGO [25].

3.3 Возможные проблемы и способы решения

Чтобы раскрыть весь потенциал гибридного подхода, необходимо решить несколько проблем.

3.3.1 Принцип «не навреди»

Большую часть времени работы программы, как правило, занимают циклы. Именно модификация циклов оказывает наибольшее влияние на время работы программы, а значит, имеет смысл свести число точек возврата управления во время работы цикла к минимуму, либо исключить вообще. Альтернативный способ заключается в том, чтобы создать несколько различных трансформаций, опробовать их на малом числе итераций цикла (считая, что одна итерация при одной трансформации выполняется примерно за одно и то же время), и для оставшейся части цикла запустить наиболее быстрый способ.

То есть вместо кода:

```
for (int i = 0; i < 100000; i++) {<default>}
```

будет выполнен:

```
measure_start();
for (int i = 0; i < 1000; i++) {<transform1>}
measure_reset();
for (int i = 0; i < 1000; i++) {<transform2>}
measure_reset();
...
measure_reset();
for (int i = 0; i < 1000; i++) {<transformN>}
measure_end();

for (int i = 0; i < 100000 - N * 1000; i++) {<transformBest>}
```

Подобный подход позволяет найти наиболее эффективную реализацию трансформации, основываясь, к тому же, на данных о регистрах, что может обеспечить ускорение решения определённого круга задач.

3.3.2 Выбор мест для точек возврата управления

Как уже упоминалось, одной из наиболее значимых задач, которые важно решить оптимально, является грамотный выбор точек возврата управления, то есть мест запускаемой программы, в которых будет возвращаться управление оптимизатору для дальнейшей трансформации.

В наиболее простом случае можно избегать возврата управления внутри циклов. Однако, такой подход не всегда является оптимальным – что, если программа содержит несколько вложенных циклов, и внешние циклы работают достаточно медленно? В таком случае есть риск упустить оптимизации, которые оказались бы возможными в случае прерывания работы внутри цикла.

В данной работе рассматривается случай, при котором точки останова выставляются вне каких-либо циклов.

3.3.3 Методика измерения времени работы

Так как число итераций тестируемого цикла может оказаться достаточно малым, необходимо выбрать наиболее точную методику замера времени работы каждой из реализаций.

Это может быть осуществлено с помощью машинной инструкции RDTSC в ассемблере x86, возвращающей в регистрах EDX и EAX число тактов с момента запуска процессора. Не совсем очевидным подводным камнем в данном случае может оказаться механизм переупорядочивания инструкций, способный повлиять на корректность измерений. Исправить ситуацию можно, вставив перед ней другую инструкцию – CPUID, что позволяет выполнить сброс конвейера. Однако использование подобной комбинации может быть чревато большими накладными расходами. Начиная с Intel Nehalem и с процессоров AMD серии 0x0F и выше (Athlon, Opteron, Turion) есть поддержка инструкции RDTSCP, выполняющей операцию, аналогичную использованию RDTSC + CPUID за гораздо меньшее время.

3.3.4 Выявление циклов

Не самой тривиальной задачей может оказаться и выявление самих циклов. В ассемблере x86 есть множество способов задания цикла: инструкцией LOOP, условными и безусловными переходами (наиболее сложный случай) и с помощью префикса REP.

В контексте данной работы рассматриваются циклы с использованием инструкции LOOP и условными переходами.

3.3.5 Глубина трансформации

Другим открытым вопросом, требующим решения, является выбор оптимальной глубины трансформации – то есть числа байт, которые будет брать в рассмотрение оптимизатор, чтобы трансформировать код.

Если выбрать слишком большое значение, это может привести к слишком долгому анализу, и, как следствие, падению производительности.

Слишком малое значение может не дать оптимизатору полной картины, что ограничит его возможности.

В качестве временного оптимального решения предлагается ограничиться кодом до конца цикла.

3.3.6 Техническая реализация

Так как трансформация исполняемого кода непосредственно во время работы программы не предусмотрена классическими средами их запуска, необходимо выяснить, каким образом эта задача будет воплощена в жизнь технически.

В контексте GNU/Linux добиться динамического изменения кода можно, выделив с помощью системного вызова `mmap()` страницы памяти, пометив их как записываемые и исполняемые, и далее передавая управление непосредственно по адресу, относящемуся к ним, возвращая управление вызвавшей программе путём вставки соответствующих инструкций.

Однако, при таком подходе может возникнуть ситуация, когда разметка памяти вызывающего процесса пересекается с разметкой вызываемого – получится конфликт. Во избежание такого случая можно перемещать код оптимизатора по случайному адресу в виртуальной памяти, что, однако, может оказаться сложной задачей. Для упрощения можно проводить эксперименты на простых ассемблерных программах, компилируемых с указанием линковщику располагать их по адресу, который гарантированно не достанется оптимизатору (наиболее остро эта проблема может проявиться в случае отключения ASLR – механизма рандомизации разметки программы, внедрённого для улучшения защиты от эксплойтов).

3.3.7 Реализация предпросчёта

С точки зрения эффективности финального решения наиболее предпочтительным вариантом является полный перебор всевозможных комбинаций инструкций с учётом всех значений регистров. Данный вариант если и применим, то лишь на крайне ограниченных наборах инструкций и регистров (из-за

долгого выполнения самого перебора), однако от него можно отталкиваться, разрабатывая другие алгоритмы.

Более эффективный предпросчёт можно осуществить по шаблонам (или через генерацию самих шаблонов). Например, можно создать генератор различных арифметических выражений, которые могут встретиться в теле цикла, сгенерировать и запустить все варианты оптимизации, и сохранить их в базу данных оптимизатора с некоторым отпечатком (хешем), пометив наиболее оптимальный для целевой микроархитектуры. При работе самого оптимизатора можно быстро искать встречающиеся паттерны по хешу и заменять их на наиболее эффективные.

3.3.8 Порядок трансформаций и многопроходность

Так же, как и в компиляторах, в динамическом оптимизаторе важную роль играет порядок применения различных трансформаций.

Например, если для эксперимента с инструкциями ADD и INC сначала выполнить трансформацию со сворачиванием цикла, то это сделает бессмысленной оптимизацию с помощью возможной замены инкрементов на одну команду суммирования – в некоторых случаях такая трансформация может оказаться более предпочтительной. Хорошая новость заключается в том, что оптимизатор имеет возможность выполнить несколько трансформаций с различным порядком и опробовать их все, в то время как у компилятора такой возможности нет.

Кроме того, некоторые трансформации имеет смысл применять многократно – например, удаление бессмысленного кода. Количество проходов, опять-таки, зависит от порядка оптимизаций, поэтому необходимо выделить оптимальную стратегию их осуществления.

В контексте данной работы вопрос не рассматривается, так как число реализуемых оптимизаций достаточно мало, и может быть опробовано оптимизатором в любых вариациях без существенных накладных расходов.

4 Техническое описание платформы для оптимизаций

Разработка полноценного универсального приложения для трансформации исполняемого кода в реальном времени – достаточно трудозатратный процесс, поэтому в контексте данной работы рассматривается крайне упрощённый его вариант, ориентированный на работу в 64-битных операционных системах на базе GNU/Linux.

Данный раздел посвящён техническим подробностям реализации рассматриваемой платформы для оптимизаций. Наиболее значимые фрагменты исходного кода помещены в приложении Б.

4.1 Разбор ELF-заголовка и загрузка программы в память

Исполняемые файлы, как правило, характеризуются метаданными, необходимыми для корректного их распознавания операционной системой. В случае 64-битной GNU/Linux-системы речь идёт о формате ELF64, описание и руководство по работе с которым можно найти в [26].

ELF64 представлен в виде структуры, состоящей из множества полей. Наибольший интерес представляют поля, отвечающие за число заголовков и секций программы (phnum и shnum), а также смещение заголовков секций в исполняемом файле в байтах (shstrndx).

Заголовки программы отвечают за загрузку различных сегментов памяти по корректным виртуальным адресам и с правильными настройками режима доступа (чтение, запись и исполнение). Например, сегмент .text содержит код и обычно загружается только с правами на выполнение; .bss резервирует место для неинициализированных данных и является неисполняемым, но с доступом на чтение и запись; .data имеет те же права доступа, что и .bss, но содержит инициализированные данные.

Задачей средства для трансформации является загрузка описанных секций по корректным адресам, а также инициализация стека. Ради упрощения в программе не реализовано перемещение собственных сегментов, в точности с предложенной в подразделе 3.3.6 идеей – поэтому предусматривается работа

только с простыми ассемблерными программами, не предполагающими какой-либо внешней линковки и загружаемыми гарантированно по адресу, гарантированно не пересекающемуся ни с каким сегментом вызывающей программы.

Каноническим способом выделения виртуальной памяти является использование системного вызова `mmap()` – с помощью него и производится выделение памяти с нужными правами доступа путём установки соответствующих флагов. Далее производится первичный анализ исходного кода программы, запускаются базовые трансформации, не требующие апробации (то есть гарантированно не вредящие) и интеграция точек возврата управления, после чего итоговый фрагмент копируется в память и получает контроль.

Таким образом, запуск оптимизатора выполняется в несколько шагов:

1. Проверка существования запускаемого файла.
2. Разбор метаданных (ELF64).
3. Выделение места в памяти для необходимых сегментов.
4. Загрузка начала сегмента .text, содержащего исполняемый код.
5. Анализ загруженного машинного кода и базовые трансформации.
6. Копирование трансформированного кода.
7. Передача управления.

Все дальнейшие манипуляции производятся уже после возврата управления вызванным фрагментом.

4.2 Поиск циклов

В системе команд x86 цикл может быть реализован несколькими способами – командой LOOP и производными (LOOPNZ, LOOPZ, LOOP – опкоды 0xE0, 0xE1 и 0xE2, соответственно), а также командами условных переходов (опкоды 0x70 – 0x7F).

Реализован следующий алгоритм поиска циклов:

1. В загруженном коде выполняется поиск всех инструкций, влияющих на поток выполнения программы.

2. Из операндов каждой инструкции извлекается абсолютный адрес, на который производится прыжок; если же указан не адрес, а относительное смещение, абсолютный адрес вычисляется.
3. Из полученного массива абсолютных адресов извлекается наименьший – он и считается началом цикла.

В действительности логика поиска циклов может оказаться гораздо более сложной, однако в рамках работы подобные допущения считаются позволительными.

4.3 Интеграция точек возврата управления

Для того, чтобы корректно восстановить работу после возврата управления оптимизатору, необходимо сохранить состояние регистров. В системе команд x86 это возможно сделать с помощью инструкции PUSHA для 16-битных регистров. К сожалению, для 64-битных регистров соответствующей команды не предусмотрено, поэтому запись регистров производится отдельными вызовами команды PUSH.

4.4 Перезапись адресов переходов

Размер результирующего кода после трансформации вовсе не обязательно будет совпадать с тем, что был до неё. Это означает, что необходимо поддерживать в актуальном состоянии относительные и абсолютные смещения, хранящиеся в условных, безусловных переходах и циклах.

Если размер кода после трансформации уменьшился, задачу можно решить тривиально – просто расставив однобайтные инструкции NOP (опкод 0x90). Однако, это может привести к снижению производительности и к более интенсивному использованию кэша инструкций, поэтому подобный подход не является желательным.

Алгоритм изменения смещений достаточно прост:

1. Машинный код делится на три части – тот, что находится до трансформированного участка, сам трансформированный код и код, идущий после него.

2. Для всех переходов, находящихся до трансформированного участка, выполняется проверка, производится ли прыжок через в него или через него. Если же прыжок приходится на трансформированный участок, ответственность за предоставление информации о метках ложится на трансформирующий проход. В противном случае вычисляется разница между размером трансформированного кода до и после и прибавляется к значению адреса/смещения, на которые приходится переход.
3. Противоположная процедура проводится для кода, находящегося после трансформации.

4.5 Оптимизации

Трансформации можно поделить на две категории – требующие апробации и гарантирующие улучшение или сохранение производительности.

4.5.1 Удаление избыточного кода

Элементарный проход, выполняющий поиск бессмысленных паттернов вроде того, что был применён в самом первом эксперименте, подразумевавшем запуск цикла с NOP-инструкциями.

К числу таких паттернов можно отнести не только NOP: бессмысленными также являются операции `XCHG EBX, EBX`, `XCHG ECX, ECX` и подобные. Также не имеют смысла циклы без тела, созданные с помощью инструкций LOOP и их условных переходов-альтернатив.

Реализуется путём последовательных запусков, работающих до тех пор, пока встречался хотя бы один бессмысленный участок.

Не является проходом, требующим апробации, так как удаление несущественного кода считается всегда выигрышной оптимизацией.

4.5.2 Выравнивание циклов

Для реализации прохода сначала производится поиск самих циклов, описанный ранее. Затем, если тело цикла не помещается в кеш-линию, производится выравнивание на её границу с помощью инструкций NOP.

Более оптимально этот проход можно реализовать, вставляя безусловный переход на начало цикла, если число выравнивающих инструкций оказалось слишком большим – такой подход используется ассемблером NASM при компиляции с флагом O2.

Так же, как и первый проход, не считается требующим апробации, поскольку эксперименты с тестовыми образцами не выявили какого-либо ухудшения производительности в случае выравнивания. Однако, не на всех процессорах выравнивание имеет смысл – например, оно не даёт какого-либо значимого эффекта на энергоэффективном ядре Intel Core i7-13700H и на Intel Celeron J4105. Отключение подобных оптимизаций можно сделать частью настройки оптимизатора (в том числе по данным предпросчёта).

4.5.3 Адаптивная замена ADD на INC и наоборот

Данная трансформация рассмотрена во втором эксперименте и показывает эффективность альтернативной реализации (с инкрементом) на одном из протестированных процессоров.

Алгоритм достаточно прост: найти все места, где встречается инструкция ADD с относительно небольшим числом в пределах 4 и заменить их на несколько вызовов инструкции INC. Аналогичная операция может быть произведена при замене инкрементирования на общую инструкцию для сложения.

Вариантов реализации этого прохода гораздо больше, чем может показаться на первый взгляд: оптимизации подобного рода можно применить и к вычитанию, и к умножению с делением, и к некоторым битовым манипуляциям – именно из-за большого числа всевозможных замен предлагается

использовать технику предварительного просчёта, которая позволит существенно ускорить как поиск вариантов для трансформации, так и саму трансформацию.

Данный проход требует апробации, так как не всегда может давать положительный результат.

5 Экономическое обоснование ВКР

Одной из важнейших вех при создании нового программного продукта является экономическое обоснование.

5.1 Экономическая целесообразность работы

В рамках выполнения дипломного проекта производится исследование существующих программ на предмет возможности применения общих оптимизаций, разработка новых способов ускорения приложений без исходных текстов, а также создание платформы для реализации полученных алгоритмов.

Благодаря полученным результатам появится возможность положительным образом влиять на производительность существующих приложений – в частности, тех, для которых исходные тексты по тем или иным причинам недоступны. Кроме того, созданный базис позволит реализовать кроссплатформенную виртуальную машину, оптимизирующую машинный код непосредственно для целевой аппаратной конфигурации.

5.2 План работ

Для успешной реализации любого проекта важно составление плана работ (что, в частности, описано в [7]). Это позволяет определить совокупную трудоёмкость создания продукта и оценить его экономическую целесообразность. Чтобы составить соответствующее расписание, нужно вычислить часовую ставку заработной платы по формуле:

$$S_h = \frac{S_m}{d \cdot h},$$

где S_h – часовая ставка, руб.; S_m – зарплата в месяц, руб.; d – количество рабочих дней в месяце, дни; h – количество рабочих часов, часы. Будем считать, что месячная заработная плата студента равна 7200 руб. (социальная стипендия), руководителя – 90000 руб.; рабочее время в день – 8 часов; количество рабочих дней в месяце составляет 21 день. В результате вычислений получим следующие значения:

1. Студент – $7200 \div (21 \cdot 8) = 7200 \div 168 = 42{,}86$ руб./час.

2. Руководитель – $90000 \div (21 \cdot 8) = 90000 \div 168 = 535{,}71$ руб./час.

После расчёта часовых ставок исполнителей составлен план разработки, представленный в таблице 5.1. Следует отметить, что трудоёмкость определена по факту выполнения каждого из этапов работы.

Таблица 5.1 – Исполнители и трудоёмкость этапов разработки (начало)

№ этапа	Этапы и содержание выполняемых работ	Исполнитель	Трудоёмкость t_0, час	Ставка, руб./час
1	Разработка технического задания	Студент	2	42,86
2	Разработка технического задания	Руководитель	2	535,71
3	Обзор литературы по теме работы	Студент	8	42,86
4	Проведение экспериментов на простых программах	Студент	10	42,86
5	Разработка и отладка контрольных примеров	Студент	5	42,86
6	Объяснение аномалий производительности в исследуемых программах	Руководитель	2	535,71
7	Выбор инструментов для анализа производительности	Студент	2	42,86
8	Консультация по применению инструментов для анализа производительности	Руководитель	1	535,71
9	Разработка новых подходов к оптимизации на основе экспериментальных данных	Студент	8	42,86
10	Реализация и отладка платформы для применения полученных алгоритмов	Студент	70	42,86
11	Консультация по созданию платформы для применения алгоритмов	Руководитель	3	535,71
12	Выполнение дополнительного раздела «Экономическое обоснование ВКР»	Студент	5	42,86
13	Оформление пояснительной записки	Студент	32	42,86
14	Консультация по выполнению ВКР	Руководитель	2	535,71
15	Оформление иллюстративного материала	Студент	8	42,86

Общие трудозатраты студента составили 150 часов или 19 дней, руководителя – 10 часов.

5.3 Затраты на заработную плату

После вычисления часовой ставки и общего количества затраченных часов можно вычислить расходы на основную и дополнительную заработную платы.

Основная заработная плата рассчитывается по следующей формуле:

$$S_m = \sum_{i=1}^{k} T_i \cdot C_i,$$

где S_m – затраты на основную зарплату исполнителей, руб.; k – число исполнителей; T_i – время, затраченное i-м исполнителем на работу, часы; C_i – ставка i-го исполнителя, руб./час. В соответствии с формулой расходы на основную заработную плату составят:

$$S_m = 150 \cdot 42{,}86 + 10 \cdot 535{,}71 = 6429 + 5357{,}1 = 11786{,}1 \text{ руб.}$$

Кроме того, необходимо учесть затраты на дополнительную заработную плату. Вычислить её можно по следующей формуле:

$$S_a = S_m \cdot \frac{H_a}{100},$$

где S_a – расходы на дополнительную зарплату исполнителей, руб.; H_a – норматив дополнительной заработной платы, %. Подставим ранее вычисленные значения в формулу и получим:

$$S_a = 11786{,}1 \cdot \frac{8{,}3}{100} = 978{,}25 \text{ руб.}$$

Таким образом, итоговая сумма расходов на заработную плату составляет 12764,35 руб.

5.4 Затраты на социальные отчисления и накладные расходы

Ещё одной статьёй расходов являются отчисления на социальные нужды. В 2023 году работодатели будут перечислять страховые взносы одним

платежом по единому тарифу 30%. Федеральное казначейство будет самостоятельно распределять взносы по видам страхования; 72,8% от тарифа пойдут на пенсионное страхование, 18,3% – на медицинское, 8,9% – на социальное. Вычисление отчислений производится по формуле:

$$S_s = (S_m + S_a) \cdot \frac{H_s}{100},$$

где S_s – отчисления на социальные нужды с заработной платы, руб.; H_s – норматив отчислений страховых взносов на обязательные социальное, пенсионное и медицинское страхование, %. Вычислим в соответствии с формулой:

$$S_s = (11786{,}1 + 978{,}25) \cdot \frac{30}{100} = 3829{,}31 \text{ руб.}$$

Также необходимо учесть, что в СПбГЭТУ «ЛЭТИ» на накладные расходы действует ставка 20%. Для расчёта этой суммы воспользуемся формулой:

$$S_o = (S_m + S_a) \cdot \frac{H_o}{100},$$

где S_o – накладные расходы, руб.; H_o – норматив отчислений накладных расходов, %. Подставим ранее полученные значения и получим:

$$S_o = (11786{,}1 + 978{,}25) \cdot \frac{20}{100} = 2552{,}87 \text{ руб.}$$

Таким образом, итоговые суммарные расходы на социальные отчисления и накладные расходы составили 6382,18 руб.

5.5 Затраты на сырьё и материалы

В процессе выполнения ВКР было произведено несколько покупок. Расходы на купленные материалы можно вычислить по формуле:

$$З_м = \sum_{l=1}^{L} G_l Ц_l \left(1 + \frac{Н_{т.з.}}{100}\right),$$

причём транспортные затраты принимаются равными 10% [27]. Результаты вычисления расходов представлены в таблице 5.2.

Таблица 5.2 – Затраты на материалы

№	Наименование	Количество, шт.	Цена за единицу, руб.
1	Канцелярские принадлежности	–	100
2	Бумага SvetoCopy Classic A4, 80 г/м^2	1	425
3	Чернила для принтера	1	500
	Транспортные расходы		102,5
	ИТОГО		1127,5

Суммарно на сырьё и материалы потрачено 1127,5 рублей.

5.6 Затраты на услуги сторонних организаций

Для доступа в сеть Интернет использовались услуги компании-провайдера «Ростелеком» с тарификацией в виде ежемесячной платы в размере 500 руб./мес вместе с НДС по ставке 20/120. Цена услуги без НДС составила:

$$500 - 500 \cdot \frac{0{,}2}{1 + 0{,}2} = 416{,}67 \text{ руб.}$$

5.7 Затраты на содержание и эксплуатацию оборудования

Помимо услуг сторонних организаций также необходимо учитывать стоимость электроэнергии. В процессе выполнения работы использовались два устройства, потребляющие электроэнергию:

- ноутбук Lenovo IdeaPad Gaming 3 15ACH6 – в среднем 0,5 кВт/час;
- принтер HP LaserJet 3390 – в среднем 0,25 кВт/час.

Время использования ноутбука составило 150 рабочих часов. Принтер же использовался 1 час. В дневное время стоимость электроэнергии в Санкт-Петербурге составляет 6,51 руб./кВт · ч. Вычислим суммарный расход электроэнергии:

$$6{,}51 \cdot (0{,}5 \cdot 150 + 0{,}25 \cdot 1) = 489{,}88 \text{ руб.}$$

5.8 Издержки на амортизационные отчисления

В процессе выполнения работы применялись устройства, представленные в таблице 5.3.

Таблица 5.3 – Список использованных устройств

№	Номенклатура	Цена, руб.
1	Ноутбук Lenovo IdeaPad Gaming 3 15ACH6	60599,00
2	Принтер HP LaserJet 3390	17000,00
	ИТОГО	77599,00

Применим формулу для вычисления амортизационных отчислений:

$$A_i = \text{Ц}_{\text{п.н.}i} \frac{H_{ai}}{100},$$

где A_i – ежегодная сумма амортизационных отчислений i-го объекта, $\text{Ц}_{\text{п.н.}i}$ – первоначальная цена объекта, H_a – годовая норма амортизационных отчислений, вычисляется по формуле:

$$H_a = \frac{1}{T_{\text{Н}}} \cdot 100,$$

где $T_{\text{Н}}$ – нормативный срок службы объекта (в годах). Для техники составляет 3 года.

Суммарная стоимость ежегодных амортизационных отчислений:

$$A = \sum_{j=0}^{n} Aj,$$

где A – общая стоимость амортизационных отчислений, n – число рассматриваемых объектов. Вычислим общие амортизационные отчисления за год:

$$A = 77599 \cdot 0{,}33 = 25866{,}34 \text{ руб.}$$

Объём амортизационных отчислений по основным средствам, применяемым в работе за период выполнения, вычисляется по формуле:

$$A_{i\text{ВКР}} = A_i \cdot \frac{T_{i\text{ВКР}}}{365},$$

где A_i – отчисления за год по основному средству (руб.), $T_{i\text{ВКР}}$ – время, в течение которого используется основное средство (дни). Таким образом, амортизационные отчисления по основным средствам за период выполнения работы составят:

$$A_{i\text{ВКР}} = 25866{,}34 \cdot \frac{19}{365} = 1346{,}46 \text{ руб.}$$

5.9 Вычисление затрат на ВКР

Итоговые затраты на реализацию ВКР представлены в таблице 5.4.

Таблица 5.4 – Вычисление расходов на ВКР

№	Наименование статьи	Сумма, руб.
1	Оплата труда	12764,35
2	Социальные отчисления	3829,31
3	Накладные расходы	2552,87
4	Материалы	1127,5
5	Услуги сторонних организаций	416,67
6	Содержание и эксплуатация оборудования	489,88
7	Амортизационные отчисления	1346,46
ИТОГО затрат		22527,04

Суммарно расходы на выполнение ВКР составили 22527,04 рубля.

5.10 Выводы

В результате вычислений расходы на ВКР составили 22527,04 рубля. За данную сумму были проведены исследования и эксперименты, разработаны подходы и алгоритмы, а также создана платформа для оптимизации выполнения программ в реальном времени. Полученные результаты позволят ускорить исполнение существующих программ (в том числе программ без исходных текстов, реверс-инжиниринг и переписывание которых может оказаться экономически неоправданным), что снизит расходы на эксплуатацию, а также затраты на электроэнергию. Также разработанная платформа позволит упростить профилирование и ручную оптимизацию программ, что

расширит возможности ускорения и сократит необходимое на оптимизацию число человеко-часов.

Кроме того, созданная платформа может служить базисом не только для интеграции новых оптимизаций в дальнейшем, но и для реализации кроссплатформенного исполнения программ, что особенно актуально в свете продвижения новых процессорных архитектур.

ЗАКЛЮЧЕНИЕ

В выпускной квалификационной работе рассмотрены примеры программ, демонстрирующих применимость различных оптимизаций в зависимости от микроархитектуры процессора, а также отражающих недостатки современных компиляторов и программ, выполняющих запуск и модификацию исполняемого кода в реальном времени.

На основе полученных результатов был сформирован новый гибридный подход, включающий в себя трансформацию кода непосредственно во время работы программы (динамическую оптимизацию) с оптимальным выбором точек возврата управления. Предложен метод апробации различных версий машинного кода путём проведения экспериментов на части итераций цикла. Также рассмотрены возможные проблемы и сложности, возникающие при реализации подобного метода ускорения программ.

По результатам разработки оптимальных моделей динамической оптимизации была реализована платформа, осуществляющая запуск и модификацию программ с минимальными накладными расходами, а также выполняющая простейшие оптимизирующие трансформации.

Полученные наработки могут быть использованы в качестве базиса для создания более совершенных динамических оптимизаторов, а также могут лечь в основу приложения, способного эффективно запускать машинный код для различных архитектур.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Таненбаум Э. Архитектура компьютера. 6-е изд. / Э. Таненбаум, Т. Остин. – СПб.: Питер, 2013. – 816 с.
2. Юров В. И. Assembler. Учебник для вузов. 2-е изд. / В. И. Юров. – СПб.: Питер, 2003. – 637 с.
3. Хорошевский В. Г. Архитектура вычислительных систем. 2-е изд., перераб. и доп. / В. Г. Хорошевский. – М.: Изд-во МГТУ Н. Э. Баумана, 2008. – 520 с.
4. Drepper U. What Every Programmer Should Know About Memory. Red Hat – 2007. – 114 p.
5. Intel® 64 and IA-32 Architectures Software Developer’s Manual. Intel, 2023. – 2522 p.
6. AMD64 Architecture Programmer’s Manual Volume 1: Application Programming. AMD, 2020. – 392 p.
7. AMD64 Architecture Programmer’s Manual Volume 4: 128-Bit and 256-Bit Media Instructions. AMD, 2023. – 1049 p.
8. Fog A. Instructions tables. Technical University of Denmark – 2022. – 429 p.
9. Fog A. The microarchitecture of Intel, AMD, and VIA CPUs. Technical University of Denmark – 2023. – 271 p.
10. Yue A., Mehta S. An Application-Oriented Approach to Designing Hybrid CPU Architectures. 2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2023. – pp. 92-102.
11. Slechta B., Crowe D., Fahs B., Fertig M., Muthler G., Quek J., Spadini F., Patel S. J., Lumetta S. S. Dynamic Optimization of Micro-Operations. – 12 p.
12. Bellard F. QEMU, a Fast and Portable Dynamic Translator. – 2005. – 6 p.
13. Nethercote N. Dynamic Binary Analysis and Instrumentation. – 2004. – 177 p.
14. Nethercote N., Seward J. Valgrind: A Framework for Heavylight Dynamic Binary Instrumentation. Proceedings of ACM SIGPLAN 2007 Conference on Programming Languages Design and Implementation, 2007. – 12 p.

15. Carvalho A. The New Linux ‘perf’ Tools. – 2010. – 11 p.
16. Батузов К. Оптимизация динамической двоичной трансляции / К. Батузов, А. Меркулов. // Труды Института системного программирования. – 2011. Т. 20 – с. 37-49.
17. Соколов Р. А. Фоновая оптимизация в системе динамической двоичной трансляции / Р. А. Соколов, А. В. Ермолович // Программирование. – 2012. – № 3. – с. 45-56.
18. Vaswani K., Srikant Y. N. Dynamic recompilation and profile-guided optimisations for a .NET JIT compiler. IEE Proc.-Softw., Vol. 150, No. 5, 2023. – pp. 296-302.
19. Hawkins B., Demsky B., Bruening D., Zhao Q. Optimizing Binary Translation of Dynamically Generated Code. – 11 p.
20. Wang W., Wu J., Gong X., Li T., Yew P.-C. Improving Dynamically-Generated Code Performance on Dynamic Binary Translators. International Conference on Virtual Execution Environments, 2018. – pp. 17-30.
21. Bala V., Duesterwald E., Banerjia S. Dynamo: A Transparent Dynamic Optimization System. – 12 p.
22. Wang M., Liang J., Zhou C., Wu Z., Xu X., Jiang Y. Odin: On-Demand Instrumentation with On-the-Fly Recompilation. PLDI ’22, 2022. – 15 p.
23. Chen W.-K., Lerner S., Chaiken R., Gillies D. M. Mojo: A Dynamic Optimization System. – 10 p.
24. Wicht B., Vitillo R. A., Chen D., Levinthal D. Hardware Counted Profile-Guided Optimizaion. – 2014. 10 p.
25. Hubicka J. Profile driven optimisations in GCC. Proceedings of the GCC Developers’ Summit, 2005. – pp. 107-124.
26. Drepper U. How To Write Shared Libraries. Red Hat – 2002. – 29 p.
27. Алексеева О. Г. Выполнение дополнительного раздела ВКР бакалаврами технических направлений: учеб. пособие. / О. Г. Алексеева. – СПб.: Изд-во СПбГЭТУ «ЛЭТИ», 2018. – 16 с.

ПРИЛОЖЕНИЕ А

Экспериментальные образцы

Для всех примеров на языке ассемблера необходим NASM версии 2.10 и выше, формат elf64. Примеры на языке Си могут быть собраны с помощью любого компилятора, поддерживающего стандарт C99.

```
; loop-alignment.asm
; Демонстрация работы хорошо и плохо выровненного машинного кода
; Реализовал: А. А. Когутенко
; Дата: 04.05.2023

global _start

%ifndef ITERATIONS
ITERATIONS  equ 1000000000
%endif

%ifndef INNER_NOPS
INNER_NOPS  equ 8
%endif

%ifndef ALIGNMENT
ALIGNMENT   equ 64
%endif


_start:

    ; Iterations count
    mov rcx, ITERATIONS

    ; Loop align - placing NOPs
    times ALIGNMENT - ($ - _start) nop

    ; Loop itself
    loop_begin:
        times INNER_NOPS nop
        loop loop_begin

    ; Terminate
    mov rax, 60         ; exit() syscall code
    xor rdi, rdi        ; Return code
    syscall
```

```
; loop-unrolling.asm
; Демонстрация работы инструкций ADD и INC
; Реализовал: А. А. Когутенко
; Дата: 02.05.2023

global _start
ARRAY_SIZE equ 100000000

section .bss

array resq ARRAY_SIZE
sum resq 1

section .text

_start:

    ; Initialize Xorshift64
    mov rax, 1

    ; Fill the array
    mov rcx, ARRAY_SIZE

    align 64
    fill:

        mov rbx, rax
        shl rax, 13
        xor rax, rbx

        mov rbx, rax
        shr rax, 7
        xor rax, rbx

        mov rbx, rax
        shl rax, 17
        xor rax, rbx

        mov [array + rcx * 8], rax
        loop fill

    ; Prepare to summarize
    xor rax, rax        ; Future sum of items

    ; Outer loop iterations
    mov rcx, 10

    outer_loop:

        xor rdx, rdx        ; Items count

        align 64

        iterate:

            add rax, [array + rdx * 8]
            add rax, [array + rdx * 8 +  8]
            add rax, [array + rdx * 8 + 16]
            add rax, [array + rdx * 8 + 24]

        %ifdef INC
```

```
        inc rdx
        inc rdx
        inc rdx
        inc rdx
    %else
        add rdx, 4
    %endif

        cmp rdx, ARRAY_SIZE     ; Check if the end of array
        jne iterate             ; Next iteration

    loop outer_loop

; Save sum to memory
bswap rax           ; Change bytes order
mov [sum], rax      ; Save to memory

; Print the result
mov rax, 1          ; write() syscall index
mov rdi, 1          ; stdout index
mov rsi, sum        ; Set buffer address
mov rdx, 8          ; Sum is 8 bytes length (64-bit register)
syscall

; Terminate
mov rax, 60         ; exit() syscall code
xor rdi, rdi        ; Return code
syscall
```

```
; branches.asm
; Демонстрация программы для сравнения ветвления и CMOV
; Реализовал: А. А. Когутенко
; Дата: 05.05.2023

global _start

%ifndef ARRAY_SIZE
ARRAY_SIZE equ 100000000
%endif

section .bss

array resq ARRAY_SIZE
count resq 1

section .text

_start:

    ; Initialize Xorshift32
    mov rax, 1

    ; Fill the array
    mov rcx, ARRAY_SIZE
    fill:

        mov rbx, rax
        shl rax, 13
        xor rax, rbx

        mov rbx, rax
        shr rax, 7
        xor rax, rbx

        mov rbx, rax
        shl rax, 17
        xor rax, rbx

        mov [array + rcx * 8], rax
        loop fill

    ; Prepare to summarize
    xor rax, rax        ; Count of items
    mov rbx, 0          ; Threshold
    xor rcx, rcx        ; Items count

%ifdef BRANCHLESS
    mov rdi, 1
%endif

    iterate:

    %ifdef BRANCHLESS

        xor rdx, rdx

        cmp [array + rcx * 8], rbx
        cmovg rdx, rdi

        add rax, rdx
```

```
%else

    cmp [array + rcx * 8], rbx
    jle continue

    inc rax

%endif

    continue:

        inc rcx

        cmp rcx, ARRAY_SIZE
        jne iterate

; Save count to memory
bswap rax           ; Change bytes order
mov [count], rax    ; Save to memory

; Print the result
mov rax, 1          ; write() syscall index
mov rdi, 1          ; stdout index
mov rsi, count      ; Set buffer address
mov rdx, 8          ; Sum is 8 bytes length (64-bit register)
syscall

; Terminate
mov rax, 60         ; exit() syscall code
xor rdi, rdi        ; Return code
syscall
```

```
// collatz.c
// Демонстрация работы компиляторов на примере проверки гипотезы Коллатца
// Реализовал: А. А. Когутенко
// Дата: 15.05.2023

#include <stdio.h>
#include <stdint.h>

int main()
{
    for (uint64_t i = 1; i <= 10000000; ++i)
    {
        uint64_t x = i;
        uint64_t iterations = 0;

        while (x != 1 && iterations++ <= 1000)
            if (x % 2 == 0)
                x /= 2;
            else
                x = x * 3 + 1;

        if (iterations == 1000)
            return 1;
    }

    return 0;
}
```

```
// data-dependence.c
// Демонстрация оптимизации в зависимости от входных данных
// Реализовал: А. А. Когутенко
// Дата: 12.05.2023

#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>

#ifndef ITEMS_COUNT
#define ITEMS_COUNT 1000000000
#endif

// Xorshift32 implementation
// A simple pseudorandom numbers generator
uint32_t xorshift32()
{
    static uint32_t x = 1;

    x ^= x << 13;
    x ^= x >> 17;
    x ^= x << 5;

    return x;
}


int main(int argc, char *argv[])
{
    uint32_t sum = 0;
#ifndef DIV
    uint32_t div = strtoul(argv[1], NULL, 10);
#endif

    // Generate random items with Xorshift32
    for (size_t i = 0; i < ITEMS_COUNT; ++i)
    #ifdef DIV
        sum += xorshift32() / DIV;
    #else
        sum += xorshift32() / div;
    #endif

    return sum;
}
```

```
// bubble_sort.c
// Демонстрация работы компилятора на примере пузырьковой сортировки
// Реализовал: А. А. Когутенко
// Дата: 08.05.2023

#include <stdio.h>
#include <inttypes.h>

#ifndef ITEMS_COUNT
#define ITEMS_COUNT 50000
#endif

uint32_t array[ITEMS_COUNT];

// Xorshift32 implementation
// A simple pseudorandom numbers generator
uint32_t xorshift32()
{
    static uint32_t x = 1;

    x ^= x << 13;
    x ^= x >> 17;
    x ^= x << 5;

    return x;
}

// Print all array items
void print_array()
{
    for (size_t i = 0; i < ITEMS_COUNT; ++i)
        printf("%" PRIu32 " ", array[i]);

    putchar('\n');
}

int main()
{
    // Generate random items with Xorshift32
    for (size_t i = 0; i < ITEMS_COUNT; ++i)
        array[i] = xorshift32();

    // Print items before sorting
    print_array();

    // In-place sorting
    for (size_t i = 1; i < ITEMS_COUNT; ++i)
        for (size_t j = 1; j <= ITEMS_COUNT - i; ++j)
            if (array[j - 1] > array[j])
            {
                int temp = array[j - 1];
                array[j - 1] = array[j];
                array[j] = temp;
            }

    // Print items after sorting
    print_array();

    return 0;
}
```

```
; bubble_sort.asm
; Ручная реализация пузырьковой сортировки на языке Ассемблера
; Реализовал: А. А. Когутенко
; Дата: 20.05.2023

global bubble_sort

bubble_sort:

    mov r8, rsi         ; R8 - i

    align 64

    outer_loop:

        mov r9, 1       ; R9 - j

        inner_loop:

            mov edx, [rdi + r9 * 4]         ; arr[j]
            mov eax, [rdi + r9 * 4 - 4]     ; arr[j - 1]

            mov r10d, edx

            cmp eax, edx                    ; Comparing

            cmovg edx, eax                  ; Swapping
            cmovg eax, r10d

            mov [rdi + r9 * 4], edx         ; Push arr[j - 1] to arr[j]
            mov [rdi + r9 * 4 - 4], eax     ; Push arr[j] to arr[j - 1]

            inc r9              ; Increase j

            cmp r9, r8          ; Compare with top index
            jl inner_loop       ; If not equal, go to inner
                                ; loop beginning

        dec r8              ; Decrease top

        test r8, r8         ; Check if zero
        jnz outer_loop      ; Go to outer loop beginning

    ret
```

```
# benchmark-loop-alignment.sh
# Скрипт для запуска тестирования различных выравниваний цикла
# Реализовал: А. А. Когутенко
# Дата: 04.05.2023

#!/bin/bash

export LC_ALL=en_US.UTF-8
TIMEFORMAT="%E"

ALIGNMENT_RANGE="59 60 61 62 63 64"
INNER_NOPS_RANGE="0 1 2 3 4 5"

echo "$INNER_NOPS_RANGE"

for ALIGNMENT in $ALIGNMENT_RANGE
do
    echo -n "$ALIGNMENT "

    for INNER_NOPS in $INNER_NOPS_RANGE
    do
        rm -f loop-alignment
        make loop-alignment \
            DEFINES="-D ALIGNMENT=$ALIGNMENT –D INNER_NOPS=$INNER_NOPS" > /dev/null
        { time ./loop-alignment; } 2>&1 | tr '\n' ' '
    done
    echo
done
```

```
# benchmark-all.sh
# Скрипт для запуска всех экспериментальных образцов
# Реализовал: А. А. Когутенко
# Дата: 30.05.2023

#!/bin/bash

DYNAMORIO_PATH="/home/$(whoami)/DynamoRIO-Linux-9.0.1"
ICC_PATH="/home/$(whoami)/intel/oneapi/compiler/latest"

TIMEFORMAT="%E"
DRRUN="$DYNAMORIO_PATH/bin64/drrun"
ICC="$ICC_PATH/linux/bin/icx"

function bench
{
    echo -n "Source     "; time "./$1" "$2" > /dev/null
    echo -n "QEMU       "; time qemu-x86_64 "./$1" "$2" > /dev/null
    echo -n "DynamoRIO  "; time "$DRRUN" "./$1" "$2" > /dev/null
    echo -n "Valgrind   "; time valgrind "./$1" "$2" 2> /dev/null > /dev/null
}



echo "loop-alignment"

# Building non-optimized version
nasm -f elf64 loop-alignment.asm -D ALIGNMENT=63 -D INNER_NOPS=4
ld -o loop-alignment loop-alignment.o
bench "loop-alignment"

# Building optimized version
nasm -f elf64 loop-alignment.asm -D ALIGNMENT=64 -D INNER_NOPS=4
ld -o loop-alignment-optimized loop-alignment.o
rm loop-alignment.o

echo -n "Optimized  "; time ./loop-alignment-optimized > /dev/null

echo



echo "loop-unrolling"

# Building non-optimized version
nasm -f elf64 loop-unrolling.asm
ld -o loop-unrolling loop-unrolling.o
bench "loop-unrolling"

# Building optimized version
nasm -f elf64 loop-unrolling.asm -D INC
ld -o loop-unrolling-optimized loop-unrolling.o
rm loop-unrolling.o

echo -n "Optimized  "; time ./loop-unrolling-optimized > /dev/null

echo



echo "branches"
```

```
# Building non-optimized version
nasm -f elf64 branches.asm
ld -o branches branches.o
bench "branches"

# Building optimized version
nasm -f elf64 branches.asm -D BRANCHLESS
ld -o branches-optimized branches.o
rm branches.o

echo -n "Optimized  "; time ./branches-optimized > /dev/null

echo



echo "collatz"

# Version with variable divisor
echo "GCC O2"
gcc -O2 -o "collatz-gcc" collatz.c
bench "collatz-gcc" 16

echo "Clang O2"
clang -O2 -o "collatz-clang" collatz.c
bench "collatz-clang" 16

echo "ICC O2"
"$ICC" -O2 -o "collatz-icc" collatz.c
bench "collatz-icc" 16

# Version with constant divisor
echo "GCC Ofast march=native"
gcc -Ofast -march=native -o "collatz-gcc" collatz.c
bench "collatz-gcc" 16

echo "Clang Ofast march=native"
clang -Ofast -march=native -o "collatz-clang" collatz.c
bench "collatz-clang" 16

echo "ICC Ofast march=native"
clang -Ofast -march=native -o "collatz-icc" collatz.c
bench "collatz-icc" 16

echo



echo "data-dependence"

# Version with variable divisor
echo "GCC"
gcc -O2 -o "data-dependence-gcc" data-dependence.c
bench "data-dependence-gcc" 16

echo "Clang"
clang -O2 -o "data-dependence-clang" data-dependence.c
bench "data-dependence-clang" 16

echo "ICC"
```

```
"$ICC" -O2 -o "data-dependence-icc" data-dependence.c
bench "data-dependence-icc" 16

# Version with constant divisor
echo "GCC const"
gcc -O2 -o "data-dependence-gcc" data-dependence.c -D DIV=16
bench "data-dependence-gcc" 16

echo "Clang const"
clang -O2 -o "data-dependence-clang" data-dependence.c -D DIV=16
bench "data-dependence-clang" 16

echo "ICC const"
clang -O2 -o "data-dependence-icc" data-dependence.c -D DIV=16
bench "data-dependence-icc" 16

echo



echo "bubble-sort"

# Without optimizations
echo "GCC O0"
gcc -O0 -o "bubble-sort-gcc" bubble-sort.c
bench "bubble-sort-gcc"

echo "Clang O0"
clang -O0 -o "bubble-sort-clang" bubble-sort.c
bench "bubble-sort-clang"

echo "ICC O0"
"$ICC" -O0 -o "bubble-sort-icc" bubble-sort.c
bench "bubble-sort-icc"

# With optimizations
echo "GCC O2"
gcc -O2 -o "bubble-sort-gcc" bubble-sort.c
bench "bubble-sort-gcc"

echo "Clang O2"
clang -O2 -o "bubble-sort-clang" bubble-sort.c
bench "bubble-sort-clang"

echo "ICC O2"
"$ICC" -O2 -o "bubble-sort-icc" bubble-sort.c
bench "bubble-sort-icc"

# With optimizations without vectorization
echo "GCC O2 fno-tree-vectorize"
gcc -O2 -fno-tree-vectorize -o "bubble-sort-gcc" bubble-sort.c
bench "bubble-sort-gcc"

echo "Clang O2 fno-tree-vectorize"
clang -O2 -fno-tree-vectorize -o "bubble-sort-clang" bubble-sort.c
bench "bubble-sort-clang"

echo "ICC O2 fno-tree-vectorize"
"$ICC" -O2 -fno-tree-vectorize -o "bubble-sort-icc" bubble-sort.c
bench "bubble-sort-icc"
```

ПРИЛОЖЕНИЕ Б

Фрагменты программы для реализации трансформаций

```
// main.c
// Фрагменты программы для запуска трансформаций
// Реализовал: А. А. Когутенко
// Дата: 05.06.2023

/**
 * Searches for a binary, including PATH directories (like shell).
 *
 * @param binary_name Binary file name
 * @return Real absolute path, if found. NULL otherwise.
 *     Should be deallocated manually with free()
 */
char *find_binary(const char *binary_name)
{
    char *env_path = getenv("PATH");

    if (strchr(binary_name, '/') || env_path == NULL)
    {
        // Slash in the name or PATH is not set

        if (access(binary_name, F_OK) == 0)
        {
            // File exists, returning full path
            return realpath(binary_name, NULL);
        }
    }
    else
    {
        // Searching in PATH directories

        char *env_path_copy = strdup(env_path);
        char *env_separate_path = strtok(env_path_copy, ":");

        while (env_separate_path)
        {
            // Handling one path from PATH
            char *assumed_binary_path = malloc(strlen(env_separate_path) +
                                               strlen(binary_name) + 2);
            sprintf(assumed_binary_path, "%s/%s", env_separate_path, binary_name);

            if (access(assumed_binary_path, F_OK) == 0)
            {
                // Binary found
                char *real_binary_path = realpath(assumed_binary_path, NULL);

                free(assumed_binary_path);
                free(env_path_copy);

                return real_binary_path;
            }

            free(assumed_binary_path);

            // Getting next path
            env_separate_path = strtok(NULL, ":");
```

```
        }

        free(env_path_copy);
    }

    return NULL;
}


/**
 * Checks for file read and execute permissions.
 *
 * @param binary_path Path to binary file
 * @return true, if OK, false otherwise
 */
bool check_file_permissions(const char *binary_path)
{
    if (access(binary_path, R_OK | X_OK))
        return false;
    else return true;
}


void parse_header(const char *binary_path)
{
    FILE *binary_stream = fopen(binary_path, "rb");

    if (!binary_stream)
        error(RCODE_OTHER, "failed to fopen()");

    char signature[4];

    if (fread(signature, sizeof(char), sizeof(signature), binary_stream) !=
        sizeof(signature))
        error(RCODE_OTHER, "unknown format");

    char elf_signature[] = {'\x7f', 'E', 'L', 'F'};

    if (strncmp(signature, elf_signature, sizeof(elf_signature)) == 0)
    {
        // Parsing as ELF

        // Read EI_CLASS to detect ELF bitness and use appropriate format
        unsigned char class = fgetc(binary_stream);

        if (feof(binary_stream))
            error(RCODE_PARSING_ERROR, "unexpected end of binary");

        if (class == 2)
        {
            // 64-bit ELF

            // Just the beginning
            rewind(binary_stream);

            struct elf64_header header;

            // Read header
            if (fread(&header, sizeof(header), 1, binary_stream) != 1)
                error(RCODE_PARSING_ERROR, "unexpected end of header");
```

```
            // Jump to the program sections
            if (fseek(binary_stream, (long) header.phoff, SEEK_SET) == -1)
                error(RCODE_PARSING_ERROR, "failed to parse program sections");

            for (size_t i = 0; i < header.phnum; ++i)
            {
                struct elf64_program_header program_header;

                // Read section
                if(fread(&program_header,sizeof(program_header),1,binary_stream) !=1)
                    error(RCODE_PARSING_ERROR, "unexpected end of program sections");

                if (program_header.type == 1)
                {
                    // PT_LOAD
                    printf("%p\n", (void *) program_header.vaddr);
                }
            }
            extern char _start;
            printf("%p\n", &_start);
        }
        else error(RCODE_NOT_SUPPORTED, "32-bit ELF does not supported");
    }
    else error(RCODE_OTHER, "unknown signature");

    fclose(binary_stream);
}
```