

**Санкт-Петербургский государственный электротехнический университет
“ЛЭТИ” им. В. И. Ульянова (Ленина)
(СПбГЭТУ “ЛЭТИ”)**

Направление подготовки: 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: “Анализ алгоритмов оптимизации размещения функций
для LLVM”**

Студент

Н. О. Сластухин

Руководитель

к. т. н., доцент

А. А. Пазников

Консультант от кафедры

к. т. н., доцент, с. н. с.

И. С. Зуев

Руководитель по расчёту

эффективности

к. э. н., доцент

В. А. Ваганова

Консультант от АО “НИИ ОЭП”

к. т. н., начальник лаборатории

Н. Г. Мотылёв

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)
“___” _____ 2023г.

**КАЛЕНДАРНЫЙ ПЛАН
выполнения выпускной квалификационной работы**

Тема Анализ алгоритмов оптимизации размещения функций для LLVM.

Студент Н. О. Сластухин

Группа № 9305

№ этапа	Наименование работ	Срок выполнения
1	Обзор литературы по теме работы	10.02.2023–25.02.2023
2	Анализ LLVM, BOLT, а также алгоритмов оптимизации размещения функции	01.03.2023–05.04.2023
3	Исследование инструмента LLVM Bolt, использующегося для оптимизаций размещения функций	05.04.2023–20.04.2023
4	Составление тестовых программ	21.04.2023–04.05.2023
5	Сравнительный анализ до и после оптимизации размещения функций	05.05.2023–20.05.2023
6	Оформление пояснительной записки	21.05.2023–25.05.2023
7	Предварительное рассмотрение работы	27.05.2023–31.05.2023
8	Представление работы к защите	02.06.2023

Руководитель

к. т. н., доцент

Студент

А. А. Пазников

Н. О. Сластухин

РЕФЕРАТ

Пояснительная записка стр. 51, рис. 18, табл. 2, ист. 16.

В данной работе будет проведено исследование с целью повышения производительности программ путем применения алгоритмов оптимизации размещения функций после предварительного профилирования. Данные алгоритмы может использовать любая ИТ – компания, занимающаяся оптимизацией кода, языками программирования, разработкой компиляторов, виртуальными машинами и интерпретаторами. Для достижения этой цели, будет разработан ряд тестовых программ, и бинарные файлы этих программ будут анализироваться и оптимизироваться для достижения наилучшего распределения функций внутри них.

В процессе исследования были исследованы различные инструменты для оптимизации программ, произведены замеры до и после модернизации, с целью выявления полезности алгоритмов оптимизации размещения, а также проведено создание тестовых программ для тестирования различных аспектов и особенностей. Для дальнейших исследований они были обработаны при помощи инструментов LLVM –Bolt и Clang.

Целью выпускной квалификационной работы является анализ алгоритмов оптимизации размещения функций в LLVM.

Объектом исследования являются алгоритмы оптимизации размещения функций в LLVM.

Предметом исследования являются увеличение производительности программ при применении к ним алгоритмов оптимизации после предварительного профилирования.

ABSTRACT

In this work we will study how to improve program performance by using optimization algorithms for function allocation after pre – profiling. These algorithms can be used by any IT company dealing with code optimization, programming languages, compiler development, virtual machines and interpreters. To achieve this, a number of test programs will be developed and the binaries of these programs will be analyzed and optimized to achieve the best distribution of functions within them.

During the research process, various tools for program optimization were investigated, measurements were taken before and after upgrading to determine the usefulness of placement optimization algorithms, and test programs were created to test various aspects and features. For further research, they were processed using the LLVM Bolt and Slang tools.

The object of research is optimization algorithms for function placement in LLVM.

The subject of research is the increase of program performance when applying the optimization algorithms to them after pre – profiling.

The purpose of this thesis is to analyze LLVM function allocation optimization algorithms.

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ	6
ВВЕДЕНИЕ	8
1 Алгоритмы оптимизации размещения функций в Low – Level Virtual Machine.....	11
1.1 Базовых блоки.....	11
1.2 Размещение функции	13
1.3 Разделение функций на горячие и холодные	17
1.4 Добавление промежутков между функциями	19
2 Профилирование	21
2.1 Low – Level Virtual Machine	22
2.2 LLVM – Bolt.....	24
2.3 Time.....	25
3 Исследование и разработка тестовых файлов	27
3.1 Настройка и работа с LLVM – Bolt	27
3.2 Тестирование алгоритмов оптимизации для улучшения производительности	32
4 Оценка влияния предложенных технических и организационных изменений проекта, программы, предприятия	36
4.1 Организационные изменения в компаниях	36
4. 2 Эффективность организационных изменений	43
ЗАКЛЮЧЕНИЕ.....	45
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ.....	47
ПРИЛОЖЕНИЕ А. Порядок выполнения команд для оптимизации	49
ПРИЛОЖЕНИЕ В. Скриншоты выполнения программ.....	50

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

LLVM (Low – Level Virtual Machine) – это инфраструктура компилятора с исходным кодом, которая предоставляет набор инструментов для разработки компиляторов и другие инструменты для обработки программного кода;

BOLT (Binary Optimization and Layout Tool) – это инструмент оптимизации бинарных файлов для ускорения работы программ.

API (Application Programming Interface) – набор функций, классов и протоколов, предоставляемых LLVM для взаимодействия с его функциональностью и возможностями;

Группировка функций — это оптимизация, которая изменяет порядок функций в бинарном файле, чтобы повысить производительность;

Кэш – промах – это событие, когда запрошенные данные отсутствуют в кэше, и процессор должен обратиться к оперативной памяти для получения этих данных.

Горячие функции – это функции или участки кода, которые вызываются часто или являются критически важными для работы программы.

Холодные функции – это функции или участки кода, которые вызываются редко или вообще не вызываются в процессе выполнения программы; LLVM – Bolt – инструмент для различных оптимизаций, в основном направлен на методы оптимизаций с учётом размещения инструкций.

Профилирование – это сбор данных о том, как программа работает во время ее выполнения.

Профиль – это файл, который содержит информацию о результатах профилирования, такие как время выполнения различных частей программы.

Базовый блок – это группа машинных инструкций, объединенных в один блок для анализа выполнения кода.

IR (Intermediate Representation) – это промежуточное представление программы на низком уровне, используемое компиляторами LLVM.

Пайнлайн файл – конфигурационный файл, который определяет структуру и последовательность операций в пайплайне обработки данных или выполнения задач.

ВВЕДЕНИЕ

Для оптимизации размещения функций в программном коде зачастую применяются инструменты LLVM и BOLT. Эти инструменты предоставляют мощные возможности для улучшения производительности и оптимизации исполнения программ.

Актуальность использования алгоритмов оптимизации размещения функций заключается в том, что они могут значительно улучшить производительность программы. Эти алгоритмы определяют оптимальное расположение функций в памяти на основе анализа частоты вызовов функций и других факторов, которые могут влиять на производительность.

LLVM – это инфраструктура компилятора с исходным кодом, которая предоставляет набор инструментов для разработки компиляторов и другие инструменты для обработки программного кода. LLVM использует промежуточное представление IR для представления исходного кода и оптимизирует этот IR для создания машинного кода.

Группировка функций — это оптимизация, которая изменяет порядок функций в бинарном файле, чтобы повысить производительность. В частности, количество групповых функций может помочь уменьшить кэш – промах, повысить скорость ссылок и увеличить запуск программ. При применении происходит следующее:

1. Уменьшение кэш – промахов: Когда программа загружается в память, она разбивается на различные сегменты, включая сегменты функций. Если функции, используемые вместе, располагаются рядом в памяти, это уменьшает количество кэш – промахов, что может привести к улучшению производительности.
2. Улучшение локальности ссылок: Если функции, использующие общие данные, расположены рядом в памяти, это может улучшить локальность ссылок. Локальность ссылок – это свойство программы, которое описы-

вает, насколько близко находятся ссылки на данные друг к другу в памяти. Чем более локальны ссылки, тем меньше вероятность, что данные будут выброшены из кэша, что улучшает производительность.

3. Ускорение запуска программы: Группировка функций может уменьшить время, необходимое для загрузки и инициализации программы. Если функции, которые используются раньше, расположены в начале бинарного файла, это может ускорить запуск программы.
4. Уменьшение размера бинарного файла: Поскольку группировка функций может привести к более эффективной упаковке кода в бинарном файле, она может привести к уменьшению размера файла. Это может быть особенно полезно для встраиваемых систем с ограниченной памятью.

Кроме того, группировка функций может помочь снизить накладные расходы, связанные с выполнением программы, и улучшить ее масштабируемость.

BOLT – это инструмент оптимизации бинарных файлов для ускорения работы программ. В основном он используется для оптимизации производительности программ, особенно в больших проектах с множеством функций. Его цель – уменьшить время выполнения программы, уменьшить количество кэш – промахов и уменьшить размер бинарного файла. Для оптимизации размещения функций, BOLT использует несколько алгоритмов, которые позволяют улучшить локальность выполнения кода и уменьшить количество промахов кэша.

BOLT работает в два этапа. Первый этап – это процесс профилирования, во время которого BOLT анализирует производительность программы и определяет наиболее часто используемые функции и блоки кода. Второй этап – это процесс оптимизации, во время которого BOLT перестраивает бинарный файл, размещая функции и блоки кода более эффективно.

Целью выпускной квалификационной работы является анализ алгоритмов оптимизации размещения функций в LLVM.

Объектом исследования являются алгоритмы оптимизации размещения функций в LLVM.

Предметом исследования являются увеличение производительности программ при применении к ним алгоритмов оптимизации после предварительного профилирования.

LLVM и BOLT предоставляют различные алгоритмы оптимизации размещения функций, которые можно выбрать в соответствии с конкретными требованиями. Например, в LLVM доступны алгоритмы bottom – up и top – down, которые учитывают различные аспекты, такие как размер функций, частота их вызовов и другие факторы. BOLT, в свою очередь, предлагает свой собственный алгоритм размещения функций, который оптимизирует производительность, сосредоточиваясь на минимизации переходов и улучшении кэш – попаданий.

В первом разделе рассматривается несколько алгоритмов оптимизации размещения функций в LLVM. Самыми распространёнными являются такие алгоритмы, как Function Layout, Basic blocks, Hot and Cold Function Separation и Function Padding.

Во втором разделе рассматриваются инструменты профилирования. С их помощью мы подготавливаем бинарный файл перед оптимизацией, в нашем случае это инструменты LLVM – bolt и time.

В третьем разделе проводятся тесты применённых алгоритмов оптимизации, замеряется время работы и рост производительности.

В четвёртом разделе рассчитывается готовность IT – компании к организационным и техническим изменениям.

1 Алгоритмы оптимизации размещения функций в LLVM

Всего существует несколько основных алгоритмов для размещения функций:

- базовые блоки;
- размещение функции;
- разделение функций на горячие и холодные;
- добавление промежутков между функциями;

1.1 Базовых блоки

Алгоритм оптимизации размещения базовых блоков (Basic blocks) в LLVM [7] является важной частью процесса компиляции, направленной на улучшение производительности программы. Он включает в себя ряд шагов, которые позволяют эффективно разместить базовые блоки в функции для оптимального выполнения.

Первый шаг – анализ зависимостей. В этом шаге производится анализ зависимостей между базовыми блоками, как контрольными, так и данных. Зависимости контроля связаны с переходами между блоками, а зависимости данных связаны с использованием и определением переменных. Анализ зависимостей позволяет определить, какие блоки должны быть выполнены в определенном порядке и какие зависимости можно разрешить [4].

Далее строится граф зависимостей, где каждый базовый блок представлен узлом, а зависимости – ребрами между узлами. Граф зависимостей служит основой для определения порядка выполнения базовых блоков.

Затем выбирается начальный базовый блок для размещения. Обычно выбираются блоки с наименьшим количеством зависимостей, чтобы уменьшить конфликты и улучшить параллелизм выполнения [10].

Следующий шаг – размещение базовых блоков. Базовые блоки размещаются в функции с учетом их зависимостей и статистической информации о выполнении. Основная идея состоит в том, чтобы поместить блоки с высокой

вероятностью выполнения ближе к началу функции, чтобы уменьшить затраты на переходы и кэш – промахи. Блоки с низкой вероятностью выполнения могут быть размещены ближе к концу функции.

После размещения базовых блоков производится оптимизация и перемещение. Это может включать удаление недостижимых блоков, объединение последовательных блоков и другие оптимизации. Затем алгоритм может повторять процесс размещения и оптимизации несколько раз для достижения лучшего результата, учитывая изменения в графе зависимостей.

Конечный результат алгоритма – функция, в которой базовые блоки размещены оптимальным образом, учитывая зависимости и статистическую информацию о выполнении. Это позволяет улучшить производительность программы, уменьшив затраты на переходы и улучшив кэш – попадания.

Алгоритм оптимизации размещения базовых блоков в LLVM [1] может быть более сложным и содержать дополнительные шаги и техники в зависимости от конкретной реализации и используемых оптимизаций. Он является важным компонентом компилятора LLVM, который способствует улучшению производительности и эффективности генерируемого кода, а также обладает рядом преимуществ, которые делают его привлекательным для разработчиков.

Применение данного алгоритма позволяет снизить затраты на переходы между базовыми блоками и повысить локальность данных, что способствует ускорению выполнения программы и улучшению ее производительности. Кроме того, оптимизация размещения базовых блоков способствует повышению вероятности попадания данных и инструкций в кэш памяти, что уменьшает задержки в памяти и улучшает общую производительность.

Однако стоит учитывать некоторые недостатки данного алгоритма. Реализация его может быть сложной и требовать значительных вычислительных ресурсов. Также результаты оптимизации могут зависеть от точности статистической информации о выполнении программы, и неправильные данные мо-

гут привести к нежелательным последствиям. Некорректное размещение базовых блоков или неточные статистические данные могут ухудшить производительность программы, вместо ее улучшения.

В целом, алгоритм оптимизации размещения базовых блоков в LLVM предоставляет значительные преимущества в производительности и эффективности программы. Однако его применение требует внимательного анализа и тщательного подхода, чтобы достичь наилучших результатов и избежать возможных недостатков.

1.2 Размещение функции

Размещение функции (Function Layout) – это оптимизация размещения функций в LLVM, которая определяет порядок функций в бинарном файле. Она основана на алгоритме размещения функций, описанном в BOLT.

В LLVM Function Layout может быть выполнен как часть процесса компиляции с помощью опции командной строки `-function-sections` [9]. Эта опция приводит к тому, что каждая функция помещается в свой собственный раздел в бинарном файле, и функции, которые часто вызываются вместе, могут быть размещены рядом друг с другом.

Кроме того, LLVM поддерживает оптимизацию размещения функций на уровне линкера с помощью опции командной строки `-function-order`. Эта опция позволяет задать порядок функций в бинарном файле вручную. Это может быть полезно, если заранее известно, какие функции будут часто вызываться вместе, и нужно оптимизировать размещение функций в памяти.

Однако, в отличие от BOLT, LLVM не выполняет анализ графа вызовов функций для определения порядка функций. Вместо этого он полагается на оптимизации на уровне линкера, такие как функциональный порядок и разделение функций на разделы.

Алгоритм Function Layout работает в два этапа. На первом этапе происходит анализ графа вызовов функций. Граф вызовов функций показывает, какие функции вызывают другие функции внутри программы. Этот анализ позволяет определить, какие функции наиболее часто вызываются вместе.

На втором этапе происходит оптимизация размещения функций в памяти. Функции, которые часто вызываются вместе, размещаются близко друг к другу в памяти. Это позволяет снизить количество промахов кэша, так как данные, используемые этими функциями, будут храниться в близлежащих областях памяти.

Function Layout также учитывает размер функций и использует различные эвристики для оптимизации размещения функций в памяти [15]. Он является одним из многих алгоритмов оптимизации размещения функций и может быть применен в различных контекстах, таких как оптимизация компиляторов, оптимизация загрузчиков и оптимизация динамической компоновки.

Алгоритм Function Layout состоит из следующих шагов:

1. Извлечение графа потока управления из функции. Для каждой функции LLVM создает граф потока управления, который представляет собой ориентированный граф, в котором вершины представляют блоки кода, а ребра указывают на порядок выполнения блоков.
2. Вычисление веса каждой функции. Вес функции определяется на основе количества инструкций, которые она содержит. Чем больше инструкций в функции, тем больше ее вес.
3. Сортировка функций в порядке убывания веса. Функции с большим весом должны быть выполнены раньше, чтобы уменьшить время выполнения программы.

4. Определение порядка функций, который минимизирует время выполнения программы. Порядок функций определяется путем перебора всех возможных комбинаций порядка и выбора того, который минимизирует время выполнения программы.
5. Размещение функций в новом порядке. Функции переупорядочиваются в соответствии с оптимальным порядком.

Алгоритм оптимизации размещения функции может выполняться с помощью алгоритмов bottom – up и top – down.

В LLVM bottom – up направлен на улучшение производительности кода. Он основан на оптимизации порядка функций в графе потока.

Преимущества алгоритма bottom – up в LLVM:

1. Уменьшение кэш – промахов: функции, вызываемые друг за другом, размещаются близко друг к другу в памяти, что уменьшает время доступа к памяти и увеличивает производительность.
2. Улучшение локальности данных: при размещении функций в памяти с учетом порядка вызовов уменьшается вероятность того, что данные, используемые в одной функции, будут находиться далеко от данных, используемых в другой функции. Это улучшает локальность данных и, следовательно, производительность.
3. Меньше переключений страниц: когда функции размещаются близко друг к другу, меньше вероятность, что они будут располагаться на разных страницах памяти. Это уменьшает количество переключений страниц, что также улучшает производительность.
4. Уменьшение размера исполняемого файла: при использовании алгоритма bottom – up LLVM сначала компилирует все функции в объектные файлы, а затем объединяет их в один исполняемый

файл. Это позволяет избежать дублирования кода и уменьшить размер исполняемого файла.

Недостатки алгоритма bottom – up в LLVM:

1. Большое количество пересчетов: при перемещении функций LLVM пересчитывает позиции всех функций в списке, что может быть очень затратно по времени.
2. Ограничение на размер функций: если функции слишком большие, перемещение их в другое место может занять слишком много времени и ресурсов.

Недостаточная гибкость: алгоритм bottom – up не позволяет задавать пользовательские правила для размещения функций, что может ограничивать возможности оптимизации.

Алгоритм оптимизации размещения функций top – down в LLVM является частью процесса оптимизации кода и заключается в размещении функций в определенном порядке, чтобы уменьшить время выполнения программы и использовать память более эффективно. Этот алгоритм применяется в LLVM [7] на этапе компиляции программы.

Опишу подробнее, как работает этот алгоритм:

1. Сбор информации о функциях: LLVM анализирует код программы и собирает информацию о каждой функции, включая ее размер, количество вызовов, зависимости от других функций и другие параметры. Эта информация используется для определения наиболее эффективного порядка размещения функций.
2. Ранжирование функций: на основе информации, собранной в первом шаге, LLVM ранжирует функции в порядке от наиболее эффективной до наименее эффективной. Функции, которые часто вызываются и не зависят от других функций, будут располагаться ближе к началу списка, а функции, которые редко вызываются и зависят от других

функций, будут располагаться ближе к концу списка.

3. Размещение функций: LLVM размещает функции в порядке, определенном на втором шаге. Он начинает с наиболее эффективной функции и размещает ее в памяти. Затем он размещает следующую функцию в порядке списка и так далее. Каждая функция размещается в памяти с учетом зависимостей от других функций и доступности свободного места.
4. Оптимизация кода: после размещения функций LLVM применяет дополнительные оптимизации кода, которые могут улучшить его производительность и эффективность. Это может включать в себя удаление неиспользуемого кода, устранение дубликатов, улучшение алгоритмов и другие техники оптимизации.

В целом, алгоритм оптимизации размещения функций top – down в LLVM позволяет улучшить производительность программы, распределяя функции в памяти более эффективно и уменьшая время выполнения кода.

1.3 Разделение функций на горячие и холодные

Разделение функций на горячие и холодные (Hot and Cold Function Separation) – это оптимизация, которая разделяет функции на две категории: горячие и холодные, и помещает их в разные секции памяти.

Горячие функции – это те функции, которые часто вызываются в программе, и потому имеют высокую вероятность быть кэшированными. Холодные функции – то те функции, которые вызываются реже и не нуждаются в кэшировании.

Обычно мы хотим размещать горячие функции вместе так, чтобы они соприкасались друг с другом в одной и той же строке кэша.

На рисунке 1 приведён пример размещения горячих и холодных функций. Здесь мы пытаемся улучшить использование I – кэша и DSB – кэша.

Разделение функций на горячие и холодные позволяет улучшить локальность выполнения кода и снизить количество промахов кэша. Горячие функции могут быть помещены в быстросействующий кэш, тогда как холодные функции могут быть размещены в более медленной памяти [6].

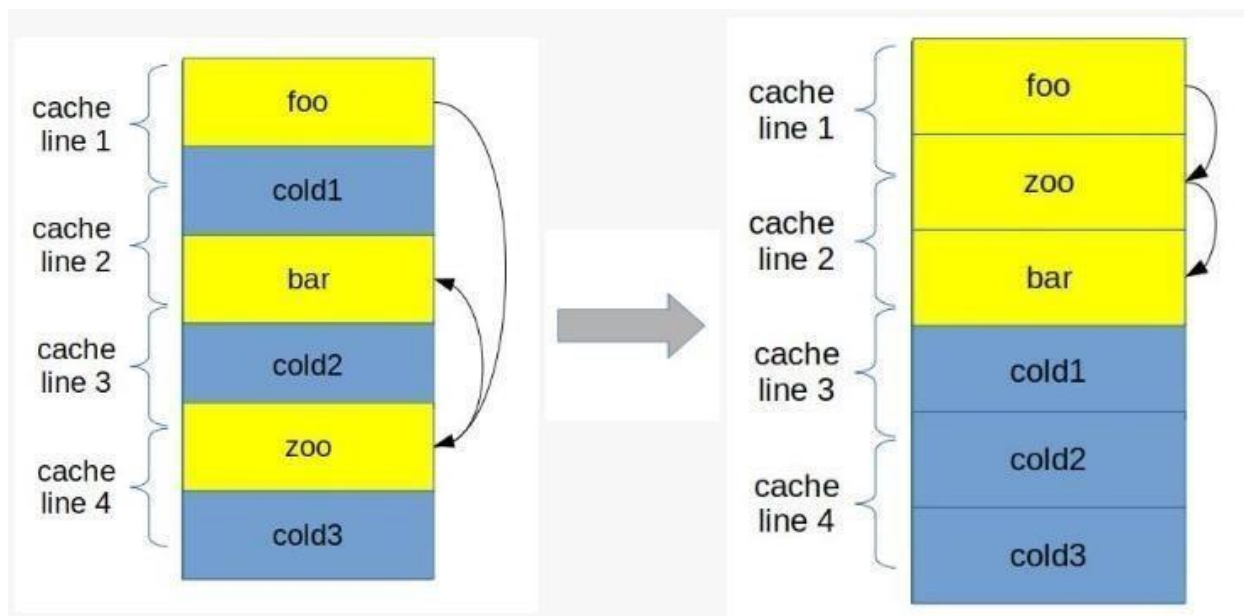


Рисунок 1 – Принцип размещения функций

В LLVM оптимизация Hot and Cold Function Separation может быть выполнена с помощью опции командной строки `-fhot-cold-split`. Эта опция приводит к тому, что функции, которые вызываются реже, помещаются в отдельный раздел `".text.cold"`, а горячие функции остаются в разделе `".text"`. Кроме того, компилятор может пометить функции как горячие или холодные с помощью атрибутов функций.

Hot and Cold Function Separation является эффективной оптимизацией для снижения затрат на кэш и улучшения производительности программы. Однако, она может потребовать дополнительной памяти для хранения дополнительных разделов, и может привести к небольшому увеличению размера бинарного файла.

1.4 Добавление промежутков между функциями

Добавление промежутков между функциями (Function Padding) – это техника оптимизации производительности программного обеспечения, которая заключается в добавлении промежутков между функциями в исполняемом файле. Эти промежутки позволяют улучшить локальность выполнения кода, уменьшить количество промахов кэша и ускорить выполнение программы.

Добавление промежутков между функциями может быть реализовано в LLVM путем использования опции `-function-sections` при компиляции кода. Эта опция заставляет компилятор помещать каждую функцию в отдельный раздел исполняемого файла, что позволяет добавлять промежутки между этими разделами. После компиляции и сборки исполняемого файла, BOLT [2] может использоваться для оптимизации размещения этих разделов и добавления промежутков между ними.

Добавление промежутков между функциями может привести к улучшению производительности программного обеспечения путем уменьшения количества промахов кэша и улучшения локальности выполнения кода. Однако недостатком этой техники является то, что добавление промежутков может увеличить размер исполняемого файла, что может привести к дополнительным затратам на память и увеличению времени загрузки программы.

Также следует учитывать, что эффективность использования Function Padding [11] может зависеть от архитектуры процессора и от специфики конкретной программы. Поэтому рекомендуется проводить тестирование производительности после применения этой техники, чтобы оценить ее эффективность в конкретном случае.

Основные преимущества данного алгоритма:

1. Уменьшение промахов кэша. Добавление промежутков между функциями позволяет снизить количество промахов кэша, так как области кода, используемые вместе, будут находиться ближе друг

к другу.

2. Улучшение локальности выполнения кода. Промежутки между функциями позволяют увеличить вероятность того, что код, связанный с одной функцией, будет находиться в одной кэш – линии, что также улучшает производительность.
3. Низкие затраты на реализацию. Функция Padding не требует сложных алгоритмов оптимизации или дополнительных данных, поэтому ее реализация относительно проста.

Недостатки данного алгоритма:

1. Добавление промежутков между функциями может увеличить размер исполняемого файла, что может привести к замедлению загрузки программы.
2. Улучшение производительности может быть не существенным в некоторых случаях, например, если функции в исполняемом файле уже достаточно разнесены в памяти.
3. Возможны проблемы с виртуальной памятью. При добавлении промежутков между функциями может возникнуть необходимость выделять больше памяти для исполнения программы, что может привести к проблемам с виртуальной памятью, особенно на устройствах с ограниченной памятью.

Добавление промежутков между функциями может повлиять на производительность программы в различных условиях, поэтому необходимо провести тестирование программы после внесения изменений.

2 Профилирование

Профилирование – это процесс анализа и сбора информации о выполнении программы с целью выявления узких мест и оптимизации производительности. Он позволяет разработчикам получить представление о том, как программа выполняется в реальном мире.

Профилирование предоставляет информацию о различных аспектах выполнения программы, таких как частота вызова функций, время выполнения, использование памяти и обращения к кэшу. Анализируя эти данные, разработчики могут идентифицировать участки кода, которые занимают больше времени и ресурсов, и сосредоточиться на их оптимизации [14].

Цель профилирования заключается в выявлении реальных проблем производительности в программе и предоставлении возможностей для их исправления. Оно помогает сосредоточить усилия на узких местах, повышает эффективность оптимизации и сокращает время разработки. Профилирование также помогает выявить проблемы с памятью, такие как утечки или неэффективное использование, и способствует их исправлению.

Однако профилирование может потреблять дополнительные ресурсы и влиять на производительность программы. Результаты профилирования могут зависеть от набора данных или окружения, поэтому важно проводить его на репрезентативных наборах данных и окружении, чтобы получить точные результаты.

Наиболее распространенными профилировщиками являются:

- LLVM – Bolt;
- time.

Все они осуществляют динамический сбор информации в процессе выполнения программы.

2.1 LLVM

Для полного понимания инструмента LLVM – Bolt необходимо ознакомиться с LLVM [1]. Изначально LLVM был разработан как инструмент для создания компиляторов и оптимизаций. Хотя изначально его аббревиатура расшифровывалась как "Low Level Virtual Machine", с течением времени он стал гораздо больше, чем просто виртуальная машина.

LLVM представляет собой промежуточное представление программы, которое позволяет оптимизировать приложения в процессе их жизни. Это представление может быть использовано как для статической, так и для динамической компиляции. Оно обеспечивает гибкость и возможность создания быстрого машинного кода на основе собираемых профилей.

С течением времени LLVM развился и стал использоваться в различных проектах. Он стал бэк – эндом для компилятора GCC и базой для проекта Clang. Комбинация Clang/LLVM [4] стала одним из самых популярных компиляторов в мире, предоставляя множество преимуществ, таких как легкость добавления статических проверок и поддержка операционной системы Windows.

LLVM также имеет множество инструментов, таких как LLVM – Bolt [2], которые используют его промежуточное представление для анализа, оптимизации и преобразования кода. Использование LLVM позволяет регулировать различные параметры программы и применять профилирование для оптимизации кода.

Механизм оптимизации, преобразования и анализа в LLVM основан на проходах. Они позволяют перебирать функции, циклы и модули, выполнять необходимые действия и оптимизации. Диспетчер проходов отвечает за планировку и регистрацию проходов, а также за управление их выполнением. LLVM предоставляет API, которое позволяет разработчикам регистрировать свои собственные проходы, что демонстрирует гибкость и преимущества этой технологии. На рисунке 2 показан процесс оптимизации через LLVM.

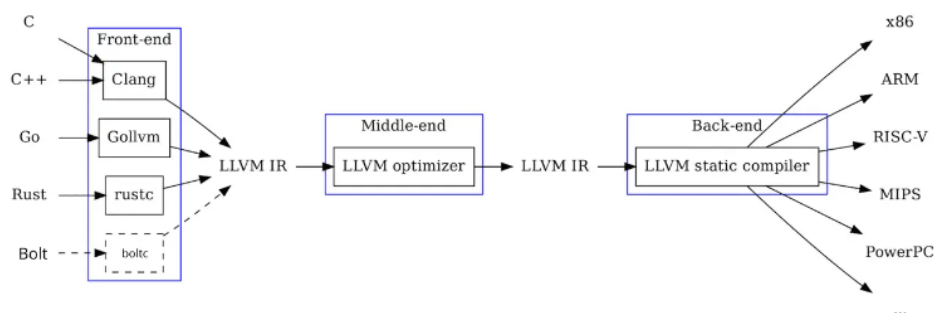


Рисунок 2 – Оптимизация через LLVM

Кроме своей основной функции промежуточного представления и создания компиляторов, LLVM предлагает ряд других возможностей и инструментов.

Одним из ключевых преимуществ LLVM является его архитектура, разделенная на несколько компонентов. Это позволяет разработчикам использовать отдельные части LLVM по отдельности или комбинировать их для достижения конкретных целей. Например, LLVM может использоваться для создания новых компиляторов, оптимизации существующего кода, генерации машинного кода для различных архитектур и даже для разработки языков программирования.

Еще одной важной особенностью LLVM является его поддержка множества языков программирования. Он не привязан к конкретному языку и может быть использован для обработки кода на различных языках, таких как C, C++, Rust, Python и многих других. Благодаря этому разработчики могут использовать LLVM в своих проектах независимо от выбранного языка программирования.

Еще одним важным аспектом LLVM является его поддержка различных операционных систем, включая Linux, macOS и Windows [10]. Это делает LLVM универсальным инструментом, который может быть использован на разных платформах и операционных системах.

Благодаря своей гибкости и модульной структуре, LLVM предлагает разработчикам возможность создания собственных оптимизаций, анализаторов и инструментов на основе его фреймворка. Это позволяет адаптировать

LLVM под конкретные потребности проекта и расширять его функциональность.

LLVM также активно поддерживается и развивается сообществом разработчиков. Это означает, что новые функции, улучшения и исправления ошибок регулярно включаются в новые версии LLVM. Разработчики могут быть уверены в том, что LLVM остается актуальным и надежным инструментом для своих проектов.

2.2 LLVM – Bolt

LLVM – Bolt – это инструмент оптимизации размещения функций, который разработан для повышения производительности исполняемых файлов. Он работает на основе фреймворка LLVM и предоставляет возможности для анализа и оптимизации порядка вызова функций в программе [4].

История LLVM – Bolt начинается в 2018 году, когда он был разработан компанией Facebook. Изначально основной целью инструмента было улучшение производительности путем изменения порядка вызова функций. Обычные компиляторы не занимаются такими оптимизациями, поскольку изменение порядка вызова функций может как улучшить, так и ухудшить производительность программы. Чтобы предотвратить непредсказуемость таких оптимизаций, рекомендуется профилирование программы, которая затем будет подвергнута обработке с помощью LLVM – Bolt.

Одним из ключевых преимуществ LLVM – Bolt [2] является его способность работать с различными проектами и приложениями. Он может использоваться для оптимизации проектов с открытым исходным кодом, а также коммерческих проектов. Это связано с тем, что LLVM – Bolt легко интегрируется в существующие процессы разработки и является гибким инструментом для оптимизации приложений.

2.3 Time

Инструмент `time` имеет долгую историю развития, начиная с ранних дней операционных систем UNIX. Оригинальная версия `time` была разработана в Bell Laboratories в начале 1970 – х годов и была включена в ранние версии UNIX.

Первоначальная реализация `time` была достаточно простой и предоставляла основную информацию о времени выполнения программы. Она измеряла общее время, затраченное на выполнение программы, и предоставляла информацию о процессорном времени и использовании памяти.

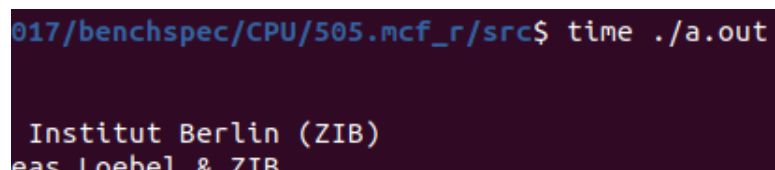
В течение десятилетий различные разработчики и сообщества внесли свой вклад в развитие инструмента `time`. Были добавлены новые функции и улучшения, чтобы сделать его более полезным и информативным.

В операционной системе Linux и его дистрибутивах, таких как Ubuntu [5], инструмент `time` был интегрирован как встроенная команда оболочки. Он предоставляет удобный способ измерения времени выполнения программы и получения основной информации о производительности.

В различных версиях UNIX и Linux разработчики добавляли дополнительные функции и усовершенствования в `time`. Например, в более новых версиях Ubuntu была представлена расширенная реализация `time` под названием `time.real`, которая предоставляет дополнительную информацию о процессорном времени, использовании памяти и других параметрах.

Инструмент `time` остается важным компонентом для разработчиков и системных администраторов, позволяя измерять производительность программы, оптимизировать ее и анализировать использование ресурсов.

На рисунке 4 приведён пример использования инструмента `time`:



```
017/benchspec/CPU/505.mcf_r/src$ time ./a.out
./a.out
0.00s user 0.00s system 0% cpu 0.00s elapsed 0.00s real 0.00s
Institut Berlin (ZIB)
eas Loebel & ZIB
```

Рисунок 4 – Использование инструмента `time`

Преимущества `time` заключаются в его простоте использования и доступности. Он встроен в операционную систему Ubuntu, поэтому не требует дополнительной установки. Просто вызовите `time` из командной строки, чтобы измерить время выполнения программы.

`time` предоставляет базовую информацию о времени выполнения программы, такую как общее время выполнения, процессорное время и использование памяти. Это может быть полезно для оценки производительности программы и оптимизации ее работы.

Однако у `time` также есть некоторые недостатки. Он предоставляет ограниченный объем информации о производительности программы, и не подходит для сложных сценариев или более подробного анализа. Он не обладает гибкими опциями настройки и функциональностью, которые могут быть необходимы для продвинутого профилирования программы.

3 Исследование и разработка тестовых файлов

В данном разделе будут проведены тесты, по результатам которых будут выделены наиболее оптимальные алгоритмы оптимизации размещения функций. Используется бенчмарк `mcg`, который моделирует транспортную систему Берлина.

3.1 Настройка и работа с LLVM – Bolt

LLVM – Bolt является предпочтительным инструментом для оптимизации размещения функций по нескольким причинам. Во – первых, он обладает мощными возможностями оптимизации, которые позволяют улучшить производительность программ. Он использует анализ профиля выполнения и другие техники оптимизации, чтобы определить наиболее часто вызываемые функции и оптимально расположить их в бинарных файлах.

Во-вторых, LLVM – Bolt интегрируется с фреймворком LLVM, который широко используется в различных проектах разработки программного обеспечения. Это обеспечивает совместимость и легкую интеграцию с существующими рабочими процессами разработки.

В данной работе был выбран вариант установки LLVM bolt с помощью инструмента контейнеризации Docker.

Установка LLVM – Bolt через Dockerfile [8] - это удобный способ настроить и запустить LLVM – Bolt в контейнеризованной среде. Dockerfile – это текстовый файл, который содержит инструкции для создания Docker – образа.

После создания Dockerfile вы можете использовать команду `Docker build` для создания Docker – образа, основанного на указанном Dockerfile. Затем вы можете запустить контейнер из этого образа, чтобы использовать LLVM – Bolt.

Использование Docker для установки и запуска LLVM – Bolt позволяет упростить процесс настройки и обеспечить однородную среду выполнения

для LLVM – Bolt на разных системах без необходимости устанавливать и настраивать все зависимости вручную.

На рисунках 5 и 6 изображен, как выглядят скаченный образ Docker и контейнер Docker.

```
test@PCC:~$ sudo docker images
[sudo] пароль для test:
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
<none>        <none>    51f163472794   6 days ago    134MB
<none>        <none>    6a084eebd9ba   6 days ago    3.81GB
ubuntu        20.04     88bd68917189   8 weeks ago    72.8MB
```

Рисунок 5 – Скаченный образ Docker

```
test@PCC:~$ sudo docker ps
[sudo] пароль для test:
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS   NAMES
5f59d638eaa8   51f163472794   "/bin/bash"             56 seconds ago Up 55 seconds          gallant_lamarr
```

Рисунок 6 – Контейнер Docker

Чтобы убедиться, что всё установлено верно, на рисунке 7 показана команда, с помощью которой можно проверить всё необходимое для установки LLVM – Bolt.

```
root@5f59d638eaa8:/# llvm-bolt --version
LLVM (http://llvm.org/):
  LLVM version 14.0.0git
  Optimized build with assertions.
  Default target: x86_64-unknown-linux-gnu
  Host CPU: skylake

BOLT revision 56ff67ccd90702d87a44c7e60abe3c4986855493
Registered Targets:
  aarch64      - AArch64 (little endian)
  aarch64_32   - AArch64 (little endian ILP32)
  aarch64_be   - AArch64 (big endian)
  arm64        - ARM64 (little endian)
  arm64_32     - ARM64 (little endian ILP32)
  x86          - 32-bit X86: Pentium-Pro and above
  x86-64       - 64-bit X86: EM64T and AMD64
```

Рисунок 7 – Информация о версии LLVM bolt

Так как в работе я использую бенчмарк mcf, который является довольно объёмным, нам необходимо его собрать, чтобы на выходе получить бинарный файл [3]. Сделать это можно с помощью команды изображённой на рисунке 8.

Нам показано, что мы собираем бенчмарк без оптимизации, но с флагом `-O2`, на выходе получая бинарный файл `a.out`.

```
test@PCC:~/Рабочий стол/CPU2017/benchspec/CPU/505.mcf_r/src$ clang-16 -O2 *.c spec_qsort/*.c -Ispec_qsort -DSPEC -WL,-emit-reloc
```

Рисунок 8 – Сборка бенчмарка mcf

Далее необходимо создать инструментированную версию бинарного файла `a.out`, это нужно для создания профиля программы перед оптимизацией. Этот профиль будет содержать информацию о данных и работе программы. Это необходимо сделать с помощью команды изображённой на рисунке 9.

```
test@PCC:~/Рабочий стол/CPU2017/benchspec/CPU/505.mcf_r/src$ llvm-bolt-16 -instrument a.out -o a.out-instr
BOLT-INFO: shared object or position-independent executable detected
BOLT-INFO: Target architecture: x86_64
BOLT-INFO: BOLT version: <unknown>
BOLT-INFO: first alloc address is 0x0
BOLT-INFO: creating new program header table at address 0x200000, offset 0x200000
BOLT-INFO: enabling relocation mode
BOLT-INFO: forcing -jump-tables=move for instrumentation
BOLT-INFO: enabling -align-macro-fusion=all since no profile was specified
BOLT-INFO: enabling lite mode
BOLT-INFO: 0 out of 48 functions in the binary (0.0%) have non-empty execution profile
BOLT-INFO: the input contains 10 (dynamic count : 0) opportunities for macro-fusion optimization that are going to be fixed
BOLT-INSTRUMENTER: Number of indirect call site descriptors: 31
BOLT-INSTRUMENTER: Number of indirect call target descriptors: 45
BOLT-INSTRUMENTER: Number of function descriptors: 45
BOLT-INSTRUMENTER: Number of branch counters: 702
```

Рисунок 9 – Создание инструментированного бинарного файла

После этого можно приступать к оптимизации программ. В процессе оптимизации LLVM bolt будет выводить различные данные, которые оптимизируются. На выходе будет получен файл с расширением `.bolt` [12].

Пример оптимизации изображён на рисунке 10.

```
test@PCC:~/Рабочий стол/CPU2017/benchspec/CPU/505.mcf_r/src$ llvm-bolt-16 a.out -o a.out.bolt -data=/tmp/prof.fdata -reorder-blocks=cache+ -reorder-functions=hfsort -split-functions=2 -split-all-cold -split-eh -dyno-stats
BOLT-WARNING: '-reorder-blocks=cache+' is deprecated, please use '-reorder-blocks=ext-tsp' instead
BOLT-WARNING: specifying non-boolean value "2" for option -split-functions is deprecated
BOLT-INFO: shared object or position-independent executable detected
BOLT-INFO: Target architecture: x86_64
BOLT-INFO: BOLT version: <unknown>
BOLT-INFO: first alloc address is 0x0
BOLT-INFO: creating new program header table at address 0x200000, offset 0x200000
BOLT-INFO: enabling relocation mode
BOLT-INFO: enabling lite mode
```

Рисунок 10 – Процесс оптимизации

Для алгоритмов оптимизации размещения функций в LLVM таких, как Basic blocks, Function Layout, Hot and Cold Function Separation используются различные флаги для оптимизации.

Для алгоритма Basic blocks используется флаг командной строки:

- `-reorder-blocks`: эта опция переупорядочивает базовые блоки внутри функций для улучшения локальности данных и уменьшения переходов между блоками.

Параметры `-reorder-blocks`:

- `cache+` или `cache-`: управление стратегией размещения базовых блоков с учетом кэша процессора.

Для алгоритма Function Layout используются флаги командной строки:

- `-function-order=name`: размещение функций в порядке их появления в коде;
- `-function-order=reorder`: упорядочивание функций для улучшения локальности данных;
- `-reorder-functions=hfsort`: упорядочивание функций с учетом профиля исполнения.

HFSort (Hot First Sort) – это стратегия упорядочивания функций, которая помещает "горячие" функции (часто вызываемые или критически важные функции) перед "холодными" функциями (реже вызываемыми функциями или функциями с меньшей важностью). Горячие функции размещаются ближе к началу исполняемого файла или секции кода, чтобы улучшить локальность данных и снизить количество переходов между различными частями кода.

Для алгоритма Hot and Cold Function Separation используется флаг командной строки:

- `-split-functions`: эта опция разделяет функции на более мелкие части для лучшего размещения кода и снижения влияния холодных кодовых путей на горячие пути.
- `-split-all-cold`: с помощью этой опции можно разделить все "холодные" блоки в отдельные функции.

На рисунке 11 изображён пример применения оптимизации.

```
test@PCC:~/Рабочий стол/CPU2017/benchspec/CPU/505.mcf_r/src$ llvm-bolt-16 a.out -o a.out.bolt3 -data=/tmp/prof.fdata -reorder-functions=hfsort -function-order=reorder -function-order=name -function-order=layout -split-eh -dyno-stats
BOLT-INFO: shared object or position-independent executable detected
BOLT-INFO: Target architecture: x86_64
BOLT-INFO: BOLT version: <unknown>
BOLT-INFO: first alloc address is 0x0
BOLT-INFO: creating new program header table at address 0x200000, offset 0x200000
BOLT-INFO: enabling relocation mode
```

Рисунок 11 – Применения оптимизации Function Layout

Также в каждом алгоритме оптимизации будут использоваться флаг командой строки и опция `-split-eh`, `-dyno-stats` [13]. Флаг `-split-eh` и инструмент `-dyno-stats` являются дополнительными возможностями LLVM – Bolt, которые могут помочь в оптимизации размещения функций.

Флаг `-split-eh` позволяет разделить обработку исключений на отдельные блоки кода. Исключения - это механизм обработки ошибок, который может повлиять на производительность программы. Разделение обработки исключений позволяет LLVM – Bolt проанализировать и оптимизировать каждый блок кода независимо, что может привести к улучшению производительности путем более эффективного размещения функций.

Инструмент `-dyno-stats` предоставляет подробную информацию о производительности программы после применения оптимизаций с помощью LLVM – Bolt. Он собирает статистику о времени выполнения функций, использовании кэша и других метриках производительности. Эта информация позволяет разработчикам анализировать эффект оптимизаций и принимать решения о дополнительных шагах для улучшения производительности.

Флаг `-split-eh` и инструмент `-dyno-stats` являются частью мощного инструментария LLVM – Bolt, который помогает в оптимизации размещения функций и повышении производительности программы.

3.2 Тестирование алгоритмов оптимизации для улучшения производительности

Для тестирования на улучшение производительности был взят бенчмарк mcf, который моделирует транспортную систему Берлина.

Он содержит файлы для разной нагрузки при тестировании:

1. test – минимальная нагрузка;
2. train – средняя нагрузка;
3. refrate – высокая нагрузка;
4. refspeed – очень высокая нагрузка;

В работе я использовал две нагрузки, это train и refrate. Также произвели один при очень высокой нагрузке refspeed. Test нагрузка не использовалась, так как в результате её работы выводится небольшие значения, при просмотре которых не так наглядно виден рост производительности, как при других нагрузках. Для получения информации о времени работы программы была использована опция time, которая замеряла время и выводила соответствующие данные.

Было произведено по 5 замеров времени для каждого алгоритма, и занесено среднее значение по этим данным в таблицу 1

Таблица 1 – Результаты исследований алгоритмом на бенчмарке mcf

Нагрузка	Варианты оптимизации				
	Без оптимизации	Basic blocks	Function Layout	Hot and Cold Function Separation	Все алгоритмы
train	81 сек.	80 сек.	79 сек.	81 сек.	78 сек.
refrate	336 сек.	325 сек.	324 сек.	328 сек.	320 сек.
refspeed	1065сек.	1048 сек.	1043 сек.	1054 сек.	1040 сек.

По результатам таблицы 1 были составлены графики, содержащие сравнение вариантов оптимизации для каждой из нагрузок.

На рисунке 12 изображено гистограмма сравнения времени работы оптимизаций при нагрузке train.

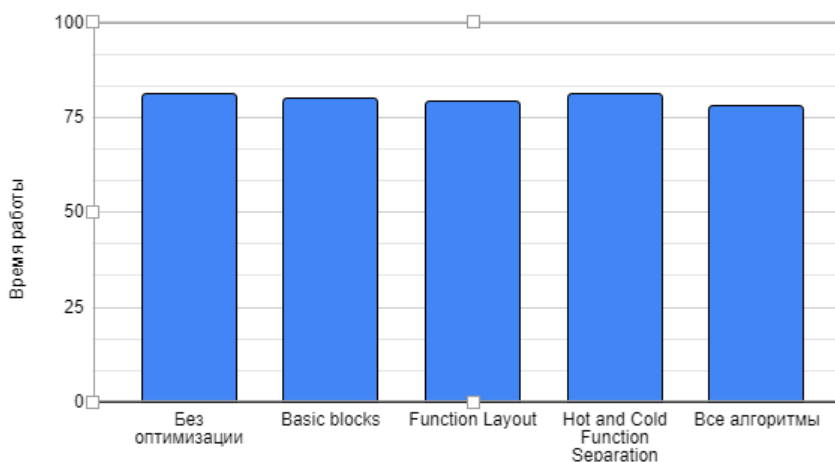


Рисунок 12 – Гистограмма к таблице 1 нагрузки train

И этой гистограммы можно сделать вывод, что при средней нагрузке повышение производительности происходит не сильно заметно. При использовании всех алгоритмов размещения функции происходит самый большой прирост производительности, который составляет 4%. Самым плохим в данном примере алгоритмов оказался Hot and Cold Function Separation, он вовсе не дал никакого прироста. Это может быть связано с тем, что при данной нагрузке использует не слишком много горячих и холодных функций, в следствии чего алгоритм показал такие плохие результаты.

На рисунке 13 изображена гистограмма сравнения времени работы программы при нагрузке refrate.

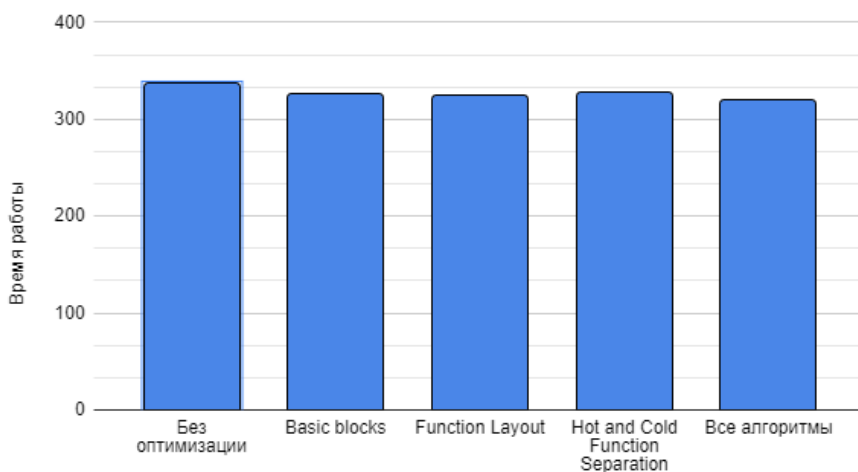


Рисунок 13 – Гистограмма к таблице 1 при нагрузке refrate

По данной гистограмме уже легче заметить разницу при применении оптимизации, так как результат составляет несколько секунд. В данном варианте, все алгоритмы вновь оказывают самый большой прирост производительности. Если рассматривать алгоритмы по отдельности, можно выделить алгоритм Function Layout, который даёт прирост на 4%, что является хорошим показателем, так как все алгоритмы дали прирост на 5%.

При refspeed используется самая большая нагрузка, это изображено на рисунке 14.

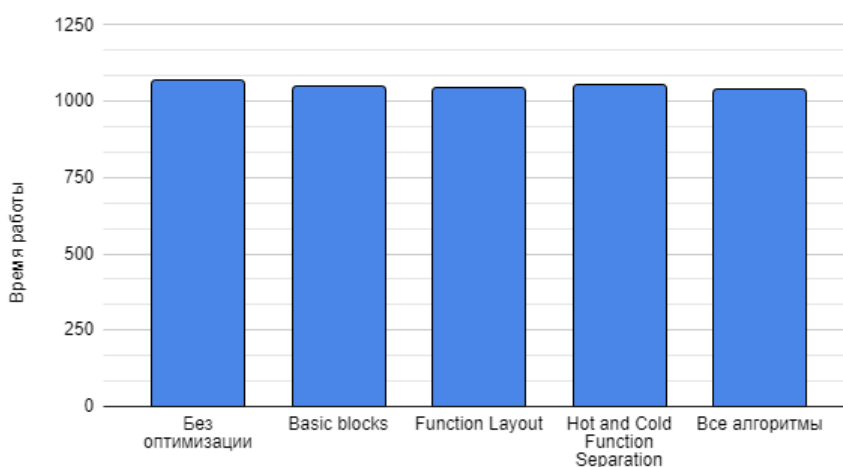


Рисунок 14 – Гистограмма к таблице 1 при нагрузке refspeed

По данной гистограмме можно заметить, что все алгоритмы дали прирост производительности, это говорит о том, что на программах с большим объёмом каждый алгоритм может давать прирост производительности. В совокупности все алгоритмы дали наибольший эффект. Из алгоритмов можно опять выделить алгоритм Function Layout, так как он улучшил программу на практически на уровне со всеми алгоритмами. На данных тестах хуже всего себя проявил алгоритм Hot and Cold Function Separation, это подтверждает слова, сказанные выше о том, что конкретно для данного теста алгоритм не подходит. Что касается алгоритма Basic blocks, на протяжении тестов он показывал средние показатели прироста производительности. Из чего можно сделать вывод, что данные алгоритмы лучше всего использовать в паре с другими алгоритмами, так как они могут дать дополнительный прирост и ускорить время работы программы.

В приложении А представлены команды и описание к ним, для работы в LLVM, а также для применения алгоритмов оптимизации размещения функции в LLVM.

4 Оценка влияния предложенных технических и организационных изменений проекта, программы, предприятия

В данном разделе будет рассмотрена готовность к изменениям ИТ – компаний, а также рассчитана успешность этих изменений.

Применение технических и организационных изменений в проекте, программе или предприятии может иметь значительное влияние на его результативность и успех. Они могут привести к улучшению процессов, повышению эффективности и общей производительности, а также созданию новых возможностей и преимуществ.

Однако, при внедрении таких изменений возникают определенные проблемы и недостатки, с которыми необходимо справиться. Некоторые сотрудники или заинтересованные стороны могут сопротивляться изменениям из – за неопределенности или опасений относительно их влияния на их работу или позицию. Кроме того, внедрение изменений может потребовать значительных инвестиций и ресурсов, а также вызвать неожиданные проблемы или риски.

Тем не менее, правильно спланированные и эффективно реализованные изменения могут принести значительные преимущества. Они могут улучшить производительность, качество и пользовательский опыт, оптимизировать использование ресурсов и создать новые возможности для инноваций и развития. Понимание и учет этих факторов помогают организациям принять взвешенные решения и достичь желаемых результатов при внедрении технических и организационных изменений.

4.1 Организационные изменения в компаниях

Организационные изменения являются неотъемлемой частью развития и успеха организации. Они необходимы для адаптации к изменяющемуся окружению, такому как рыночные требования и технологические инновации.

Организационные изменения помогают повысить эффективность и производительность организации, оптимизируя рабочие процессы, улучшая коммуникацию и сотрудничество, а также снижая издержки. Они также помогают организации реализовывать свои стратегические цели, перераспределяя роли и ответственность, развивая новые компетенции и привлекая новых талантов. Организационные изменения могут быть направлены на улучшение клиентского опыта и обслуживания, что делает организацию более конкурентоспособной. Кроме того, они помогают эффективно управлять переменами внутри организации, снимая сопротивление к изменениям и повышая мотивацию сотрудников.

Руководители играют ключевую роль в достижении хороших организационных изменений. Они должны создавать ясное видение изменений, устанавливать четкие цели и ожидания, а также поддерживать и мотивировать сотрудников. Важно вовлекать сотрудников в процесс изменений. Руководители должны эффективно управлять изменениями, быть гибкими и адаптироваться к новым обстоятельствам. Они должны обеспечивать непрерывное обучение и развитие сотрудников, а также предоставлять оценку и обратную связь.

Организация, особенно IT – компания, представляет собой динамичную структуру, объединяющую специалистов в области информационных технологий, совместно работающих для достижения определенных задач и целей. Организационные изменения в IT – компаниях играют значительную роль, пройдя через этапы анализа, планирования, вовлечения, внедрения и оценки результатов. Важно отметить, что такие изменения необходимы для адаптации к быстро меняющейся технологической среде, оптимизации рабочих процессов, повышения эффективности и обеспечения конкурентоспособности компании [16].

Так, данные из подтверждают необходимость проводить оценку целесообразности, готовности и эффективности планируемых изменений в компании. На рисунке 15 представлен подход к анализу и оценке нововведений на этапе их планирования.

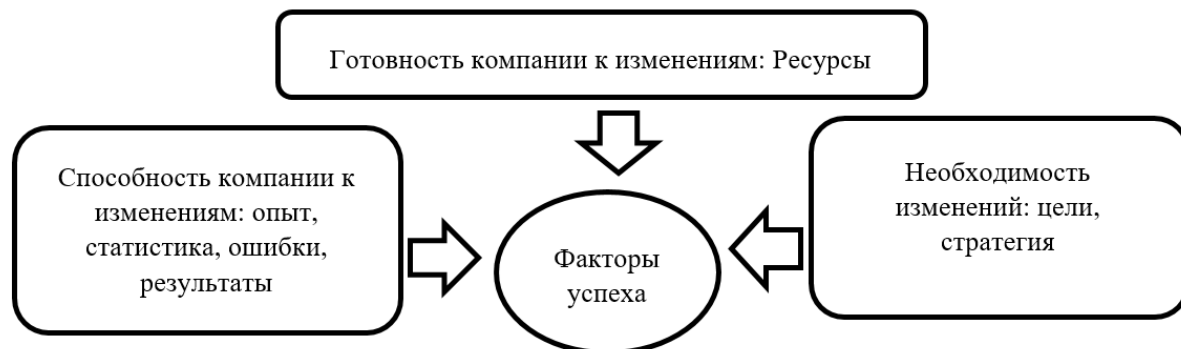


Рисунок 15 – Факторы успеха изменений в компании

Объектом данного исследования является любая компания в IT – индустрии, занимающаяся разработкой программного обеспечения и проводящая исследования в этой области. Она стремится использовать LLVM для повышения производительности своих продуктов и создания передовых решений, которые устанавливают новые стандарты в отрасли. Использование LLVM охватывает различные аспекты, включая языки программирования, компиляторы, оптимизацию кода, инструменты разработки, виртуальные машины, интерпретаторы и исследовательскую деятельность.

LLVM и связанные с ним проекты выделяются своей моделью открытого исходного кода. Это означает, что их исходный код доступен публично и открыт для свободного использования, изучения, изменения и распространения. Такая открытая модель создает уникальные возможности для разработчиков и компаний, позволяя им внести свой вклад в проект, улучшить его функциональность и надежность, а также создавать собственные проекты и продукты на основе LLVM.

Благодаря доступности исходного кода, сообщество разработчиков может активно сотрудничать, обмениваться знаниями и инновационными идеями, способствуя развитию и прогрессу в области компиляции, оптимизации

и программирования. Такая коллективная работа способствует развитию новых технологий, расширению возможностей и постоянному совершенствованию LLVM и связанных проектов.

Задачей, исследуемой в ВКР, является анализ алгоритмов оптимизации размещения функции для LLVM. Применение концепции размещения функции на практике открывает перед нами возможности оптимизации программного кода и повышения производительности в реальных проектах. С помощью LLVM в качестве инструмента для оптимизации, мы можем внедрить соответствующие алгоритмы размещения функции в реальные программные приложения. Размещение функции в оптимальных местах в коде позволяет сократить задержки при выполнении программы, улучшить использование ресурсов памяти и процессора, а также увеличить прогнозируемость работы программы. Это может привести к значительному ускорению выполнения программы и повышению отзывчивости приложения. Для применения данных алгоритмов в реальных проектах требуется глубокое понимание архитектуры и требований конкретного приложения. Разработчики должны изучить код, проанализировать вызовы функций, оценить потребление ресурсов и принять решение о наилучшем размещении функций в программе.

Далее проводится оценка целесообразности изменений в организации в условиях предложенного проекта по выбранным в соответствии с методическими указаниями по приоритетности значимости критериям с использованием пятибалльной шкалы. Данные представлены в таблице 2.

Таблица 2 – Оценка целесообразности изменений

<i>1. Готовность организации к изменениям</i>			
1.1. Персонал	Готовность персонала к изменениям, наличие лидера, квалификация персонала	4	Готовность персонала к изменениям и успешная реализация оптимизации размещения функций в LLVM требует команды высококвалифицированных разработчиков с глубоким пониманием принципов оптимизации, лидером, который обладает глубоким пониманием проблематики оптимизации кода и может обеспечить координацию работы команды. Он принимает стратегические решения, определяет приоритеты и обеспечивает согласованность работы команды. Важно также учитывать, что оптимизация размещения функций в LLVM является продвинутой и специализированной областью. Поэтому персонал, занимающийся этим, должен постоянно обновлять свои знания и следить за последними тенденциями и разработками в области оптимизации и компиляции.
1.2. Материально технические факторы	Наличие необходимых ресурсов, доступность материалов, уровень износа техники	4	В компаниях, занимающихся алгоритмами оптимизации размещения функций в LLVM, имеются необходимые ресурсы для успешной работы. Компании обеспечивают доступность материалов и информации, необходимых для изучения и применения алгоритмов оптимизации размещения функций. У сотрудников есть доступ к документации LLVM, исходному коду и справочным материалам, которые помогают им углубить свои знания и разобраться в тонкостях размещения функций. Также компании должны обеспечивать персонал необходимой технической инфраструктурой для работы с LLVM. Компьютеры и серверы оборудованы достаточной вычислительной мощностью, чтобы эффективно выполнять задачи оптимизации и размещения функций в LLVM. Относительно уровня износа техники, компании следят за состоянием своего оборудования и регулярно проводят техническое обслуживание и замену устаревших компонентов. Это позволяет сотрудникам работать с надежным и актуальным оборудованием, минимизируя проблемы, связанные с неисправностями или устаревшими компонентами.
	Достаточность финансовых средств для реализации из-	5	Компания, специализирующаяся на алгоритмах оптимизации размещения функций в LLVM, обладает достаточными финансовыми ресурсами для реализации необходимых изменений. Благодаря своей финансовой устойчивости, компания имеет возможность инвестировать в исследования, разработки и улучшение алгоритмов оптимизации размещения

	менения, потребность во внешнем финансировании		<p>функций в LLVM. Она может привлекать высококвалифицированных специалистов, обеспечивать необходимое оборудование и инфраструктуру для проведения исследований, а также проводить тестирование и оптимизацию своих алгоритмов.</p> <p>Однако, в зависимости от масштаба и амбиций компании, может возникнуть потребность во внешнем финансировании. В таких случаях, компания может рассмотреть возможности сотрудничества с инвесторами, получение грантов от научных и академических организаций, а также другие источники финансирования, которые позволят усилить их возможности и расширить сферу исследований.</p>
1.4. Информация	Наличие соответствующего программного обеспечения, разработанной системы документооборота, отчетности	4	<p>В компании, применяющей алгоритмы оптимизации размещения функций в LLVM, имеется соответствующее программное обеспечение и разработанная система документооборота и отчетности.</p> <p>Для успешной реализации и применения алгоритмов оптимизации размещения функций в LLVM, компания использует специализированное программное обеспечение. Это включает в себя инструменты и библиотеки, предназначенные для анализа и оптимизации кода, выполнения различных видов оптимизаций и улучшения производительности приложений. Это программное обеспечение обеспечивает надежную и эффективную работу с алгоритмами размещения функций в LLVM.</p> <p>Кроме того, компания также разработала систему документооборота и отчетности, которая облегчает процессы управления и контроля в проектах, связанных с оптимизацией размещения функций. Система документооборота позволяет управлять документами, отслеживать изменения, контролировать версии и обеспечивать эффективное взаимодействие между членами команды и заинтересованными сторонами. Система отчетности предоставляет информацию о прогрессе проектов, результаты оптимизации и другую необходимую статистику, которая помогает в принятии решений и оценке эффективности применяемых алгоритмов оптимизации размещения функций.</p>
2. Необходимость изменений в организации			
2.1 Внутренние факторы	Актуальность решаемых проблем, соответствие целям и стратегии развития компании	5	<p>Использование оптимизации размещения функций имеет большое значение для компаний, так как оно помогает достичь нескольких актуальных проблем, соответствует целям и стратегии развития компании.</p> <p>Оптимизация размещения функций в LLVM позволяет значительно повысить производительность и эффективность программного приложения. Это достигается путем более</p>

			<p>эффективного использования ресурсов и оптимального размещения функций в памяти. Улучшение производительности программного приложения является актуальной проблемой, так как позволяет снизить временные задержки, повысить отзывчивость системы и улучшить общее пользовательское впечатление. Также оптимизация размещения функций в LLVM помогает управлять потреблением ресурсов, таких как память и процессорное время. Это особенно важно для компаний, разрабатывающих ресурсоемкие приложения, где эффективное использование ресурсов может привести к экономии стоимости оборудования и повышению энергоэффективности.</p> <p>Кроме того, оптимизация размещения функций в LLVM способствует улучшению масштабируемости приложений. При разработке больших проектов, где множество функций и модулей взаимодействуют друг с другом, правильное размещение функций может снизить накладные расходы на связывание и упростить процесс сопровождения и расширения приложения.</p>
<p>3. Уровень риска предлагаемого изменения</p>	<p>Вероятность возникновения различных угроз – умеренна.</p>	3	<p>При оптимизации кода с помощью алгоритмов размещения функции в компании необходимо учитывать возможные угрозы, которые могут возникнуть. Одной из таких угроз является риск нарушения функциональности программы, если оптимизация произведена неправильно. Также существует потенциальная уязвимость безопасности, если функции размещены некорректно, что может быть использовано злоумышленниками для атак. Кроме того, неправильная оптимизация может привести к проблемам совместимости программы с другими модулями или платформами. Повышенная сложность оптимизированного кода может снизить его читаемость и поддерживаемость другими разработчиками. Чтобы справиться с этими угрозами, компания должна иметь квалифицированных специалистов, следить за безопасностью и совместимостью, а также обеспечить баланс между производительностью и читаемостью кода.</p>

4.2 Эффективность организационных изменений

В результате проставления баллов для ключевых факторов изменений и их эффективности производится сверстка оценок для вывода единого показателя. Для этого используются баллы, которые были расставлены в таблице, и аддитивная модель, которая позволит суммировать баллы.

Сумма баллов получилась следующей:

$$4 + 4 + 5 + 4 + 5 + 3 = 25$$

Актуальность данного подхода заключается в том, что полученная сумма баллов предоставляет объективную оценку и возможность сравнения привлекательности данных изменений для компаний. В быстро меняющейся среде бизнеса и технологий, компаниям необходимо принимать информированные решения, основанные на актуальной оценке. Сумма баллов служит важным инструментом для определения, насколько данные изменения соответствуют современным требованиям рынка, обеспечивают конкурентное преимущество и могут привлечь внимание потенциальных клиентов или инвесторов. Поэтому анализ и сравнение привлекательности изменений на основе суммы баллов является актуальным и важным в контексте принятия решений о развитии компании.

Коэффициенты линейной комбинации сверстки определяются исходя из приоритетности изменений для организации и в условиях рассматриваемого проекта получились следующими:

$$\begin{aligned}k_{\text{персонал}} &= \frac{4}{25} = 0.16, & k_{\text{мат-тех факторы}} &= \frac{4}{25} = 0.16, \\k_{\text{финансы}} &= \frac{5}{25} = 0.2, & k_{\text{информация}} &= \frac{4}{25} = 0.16, \\k_{\text{необходимость(внутр.)}} &= \frac{5}{25} = 0.2, & k_{\text{риски}} &= \frac{3}{25} = 0.12. \\&& \sum k &= 1\end{aligned}$$

Эффективность изменения предполагает оценку финансовой эффективности с учетом возможных доходов и затрат, и может выполняться с использованием различных методов. Учет рисков изменения предполагает оценку потенциальных угроз реализации преобразования и анализ успешных практик реализации подобных изменений другими компаниями.

В результате общая оценка изменений всех критериев получилась следующей:

$$0.12 * 3 + 0.2 * 5 + 0.16 * 4 + 0.2 * 5 + 0.16 * 4 + 0.16 * 4 = 4.28$$

Компании получают расширенную информацию о факторах успеха отдельных преобразований с точки зрения субъекта управления. Успешность предполагаемых проектом изменений в данном случае оцениваются по пяти-балльной шкале, как **4.28**, что говорит о высокой эффективности изменений. Необходимо иметь в виду, что существует риск получения неблагоприятных результатов при использовании данного подхода. Этот риск связан с наличием критериев, которые могут иметь свои собственные проблемы и ограничения. При оценке и сравнении привлекательности изменений для компаний, важно учитывать возможные проблемы и ограничения, связанные с выбранными критериями. Это поможет избежать ошибочных выводов и принять информированные решения на основе реалистичной оценки.

ЗАКЛЮЧЕНИЕ

Результатом выпускной квалификационной работы является подробный анализ по применению алгоритмов оптимизации размещения функции в LLVM к тестовым программам с разной нагрузкой. Для этого были применены инструменты для оптимизации, такие как LLVM, BOLT и опции, которые они используют.

Всего было выделено 4 алгоритма оптимизации размещения функций:

- Basic blocks
- Function Layout
- Hot and Cold Function Separation
- Function Padding

В результате тестов были сделаны выводы, что алгоритмы оптимизации при работе с большими по объёму программами дают значительный прирост производительности.

В ходе исследований худший результат показал алгоритм Hot and Cold Function Separation на всех 3 вариантах нагрузки.

Basic block показал средние значения, из чего был сделан вывод, что его лучше использовать с кем – то в паре для получения максимального прироста производительности.

Лучшим среди алгоритмов оказался Function Layout, он давал прирост производительности наравне с использованием всех алгоритмов сразу.

При проведении данных тестов можно сказать, что использование данных алгоритмов не приводит к приросту производительности для всех программ, необходимо тщательно изучить программу и проанализировать какие алгоритмы могут подойти лучше в конкретном случае для достижения хорошего результата.

Так же данная работа имеет дальнейшее развитие. Данные алгоритмы оптимизации размещения функций могут быть использованы любыми IT –

компаниями, занимающимися оптимизацией кода, языками программирования, разработкой компиляторов, виртуальными машинами и интерпретаторами, для достижения наилучшего распределения функций внутри них.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Документация LLVM. Writing an LLVM Pass URL: <https://llvm.org/docs/WritingAnLLVMPass.html> (дата обращения 15.04.23)
2. Репозиторий LLVM Bolt URL: <https://github.com/facebookincubator/BOLT> (дата обращения 17.04.23)
3. Advanced Build Configurations URL: <https://llvm.org/docs/AdvancedBuilds.html> (дата обращения 25.04.23)
4. Clang: a C language family frontend for LLVM URL: <https://clang.llvm.org/> (дата обращения 29.04.23)
5. Herlihy, M. The Art of Multiprocessor Programming / Maurice Herlihy, Nir Shavit, – 2012
6. German Gorelkin. Выравнивание и заполнение структур URL: <https://medium.com/german-gorelkin/go-alignment-a359ff54f272> (дата обращения 16.05.23)
7. Habr. Что такое LLVM и зачем он нужен? / Андрей Боханко URL: <https://habr.com/ru/company/huawei/blog/511854/> (дата обращения 05.05.23)
8. Docker URL: <https://www.docker.com/> (дата обращения 28.03.23)
9. LLVM Command Guide URL: <https://llvm.org/docs/CommandGuide/> (дата обращения 28.05.23)
10. Denis Bakhvalov. Improving performance by better code locality. URL: <https://easypref.net/blog/2018/07/09/Improving-performance-by-better-code-locality> (дата обращения 10.05.23)
11. Denis Bakhvalov. Machine code layout optimizations URL: <https://easypref.net/blog/2019/03/27/Machine-code-layout-optimizations#profile-guided-optimizations-pgo> (дата обращения 13.04.23)
12. Lattner, C. LLVM and Clang: Next Generation Compiler Technology / Chris Lattner, – 2008
13. Репозиторий LLVM project URL: <https://github.com/llvm/llvm-project> (дата обращения 18.05.23)

- 14.LLVM Language Reference Manual URL: <https://llvm.org/docs/LangRef.html>
(дата обращения 02.05.23)
- 15.Habr. Мой первый компилятор на LLVM: <https://habr.com/ru/articles/342456/>
(дата обращения 23.05.23)
- 16.Лопес, Б. К. LLVM: инфраструктура для разработки компиляторов / Бруно Кардос Лопес, Рафаэль Аулер / пер. с англ. Киселев А. Н. – М.: ДМК Пресс, – 2015

ПРИЛОЖЕНИЕ А

Порядок выполнения команд для оптимизации

Клонирование репозитория:

```
git clone https://github.com/chfeng-cs/CPU2017
```

Идём в каталог с бенчмарком:

```
cd CPU2017/benchspec/CPU/505.mcf/src
```

Собираем бенчмарк без оптимизаций BOLT, но с -O2, получаем бинарный файл a.out на выходе:

```
clang-16 -O2 *.c spec_qsort/*.c -Ispec_qsort -DSPEC -Wl,-emit-reloc
```

Создаём инструментированную версию бинарного файла:

```
llvm-bolt-16 -instrument a.out -o a.out-instr
```

Делаем прогон инструментации, чтобы получить профиль, по которому оптимизировать:

```
./a.out-instr ../data/test/input/inp.in
```

Проводим саму оптимизацию с BOLT

```
llvm-bolt-16 a.out -o a.out.bolt -data=/tmp/prof.fdata -reorder-blocks=cache+ -reorder-functions=hfsort -split-functions=2 -split-all-cold -split-eh -dyno-stats
```

Делаем замер оригинальной версии с помощью опции time:

```
time ./a.out ../data/test/input/inp.in
```

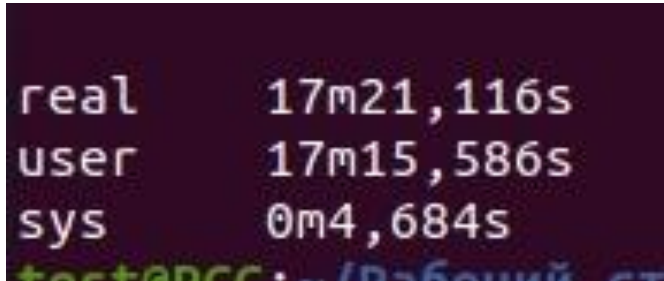
Делаем замер оптимизированной версии с помощью опции time:

```
time ./a.out.bolt ../data/test/input/inp.in
```

ПРИЛОЖЕНИЕ В

Скриншоты выполнения программы

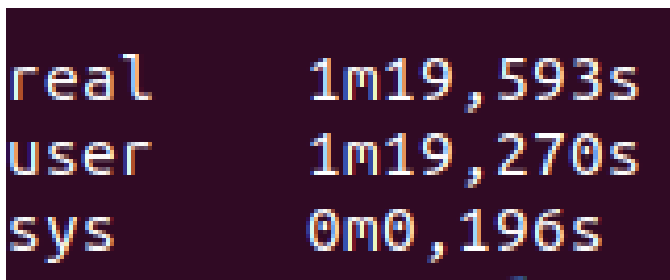
На рисунке 16 проводится результаты работы тестовой программы при нагрузке refspeed.



```
real    17m21,116s
user    17m15,586s
sys     0m4,684s
```

Рисунок 16 – Время работы программы при нагрузке refspeed

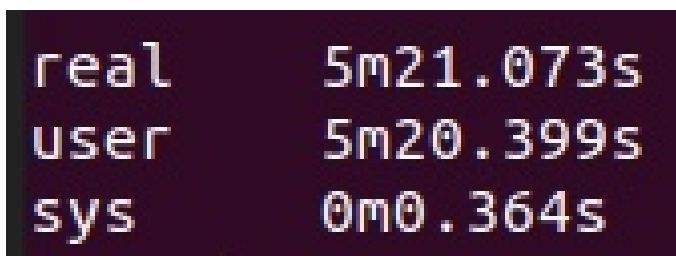
На рисунке 17 проводится результаты работы тестовой программы при нагрузке train.



```
real    1m19,593s
user    1m19,270s
sys     0m0,196s
```

Рисунок 17 – Время работы программы при нагрузке train

На рисунке 18 проводится результаты работы тестовой программы при нагрузке refrate.



```
real    5m21.073s
user    5m20.399s
sys     0m0.364s
```

Рисунок 18 – Время работы программы при нагрузке refrate