

**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 базовых блоков для  
LLVM”**

Студент \_\_\_\_\_ И. А. Китаев

Руководитель  
к. т. н., доцент \_\_\_\_\_ А. А. Пазников

Руководитель по расчёту  
эффективности  
к. э. н., доцент \_\_\_\_\_ В. А. Ваганова

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 И. А. Китаев

Группа № 9305

- 1. Тема:** Анализ алгоритмов оптимизации размещения базовых блоков для LLVM  
(утверждена приказом № \_\_\_\_\_ от \_\_\_\_\_)
- 2. Объект и предмет исследования:** процесс размещения базовых блоков в коде программы, осуществляемый компилятором LLVM.
- 3. Цель:** Оптимизация процесса размещения базовых блоков с целью улучшения производительности, эффективности и качества кода
- 4. Исходные данные:** исходный код программы, информация о вызовах функций, метрики производительности, ограничения и требования
- 5. Содержание:** анализ алгоритмов оптимизации размещения базовых блоков в LLVM. Обзор методов профилирования. Разработка тестовых программ и применение на них алгоритмов оптимизации. Оценка организационных изменений на основе IT-компаний.
- 6. Дополнительные разделы:** Оценка влияния предложенных технических и организационных изменений проекта, программы, предприятия
- 7. Результаты:** текст ВКР, иллюстративный материал, пояснительная записка.

Дата выдачи задания  
«\_\_\_» \_\_\_\_\_ 2023 г.

Дата представления ВКР к защите  
«\_\_\_» \_\_\_\_\_ 2023 г.

Руководитель  
к. т. н., доцент

\_\_\_\_\_

А. А. Пазников

Студент

\_\_\_\_\_

И. А. Китаев

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

---

**Направление** 09.03.01 “Информатика и  
вычислительная техника”  
**Профиль** “ Организация и программиро-  
вание вычислительных и информацион-  
ных 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 базовых блоков  
для LLVM

---

**Студент** И. А. Китаев

**Группа №** 9305

№ этапа	Наименование работ	Срок выполнения
1	Поиск литературы, научных статей для выполнения выпускной квалификационной работы	10.02.2022 – 25.04.2022
2	Освоение научных материалов, изучение LLVM, BOLT и алгоритмов размещения базовых блоков	01.05.2022 – 05.05.2022
3	Составление тестовых программ	05.05.2022 – 21.05.2022
4	Сравнительный анализ до и после оптимизирования алгоритмов размещения базовых блоков	21.05.2022 – 05.06.2022
5	Оформление пояснительной записки	05.06.2022 – 09.06.2022
6	Предварительное рассмотрение работы	09.06.2022
7	Представление работы к защите	21.06.2022

**Руководитель**  
к. т. н., доцент

\_\_\_\_\_ А. А. Пазников

**Студент**

\_\_\_\_\_ И. А. Китаев

## РЕФЕРАТ

Пояснительная записка содержит: 54 стр., 40 рисунке, 2 табл., 15 ист., 1 прил.

Цель дипломной работы на тему "Анализ алгоритмов оптимизации размещения базовых блоков для LLVM" состоит в исследовании и сравнении различных алгоритмов оптимизации размещения базовых блоков в рамках компилятора LLVM. Главной целью работы является выявление наиболее эффективных алгоритмов, способных повысить производительность и оптимизировать использование ресурсов системы, таких как память и процессорное время.

В результате выполнения данной работы ожидается получение новых знаний о существующих алгоритмах оптимизации размещения базовых блоков и их эффективности в рамках компилятора LLVM. Эти результаты могут быть использованы для улучшения производительности и оптимизации работы компиляторов LLVM, что в свою очередь может привести к повышению эффективности программного обеспечения, работающего на базе LLVM.

Объектом исследования являются алгоритмы оптимизации размещения базовых блоков.

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

## **ABSTRACT**

The aim of the thesis on "Analysis of Optimization Algorithms for Basic Block Placement in LLVM" is to investigate and compare different optimization algorithms for basic block placement within the LLVM compiler. The main objective of the work is to identify the most effective algorithms capable of improving performance and optimizing resource utilization, such as memory and CPU time.

The expected outcome of this work is to gain new knowledge about existing optimization algorithms for basic block placement and their effectiveness within the LLVM compiler. These findings can be used to enhance the performance and optimization of LLVM compilers, leading to increased efficiency of software based on LLVM.

The object of the research is optimization algorithms for basic block placement.

The subject of the research is the improvement of program performance through the application of optimization algorithms after preliminary profiling.

## СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ .....	7
ВВЕДЕНИЕ .....	9
1 Обзор подходов к оптимизации размещения базовых блоков в программном коде .....	11
1.1 Описание алгоритма Reorder-Blocks .....	13
1.2 Описание алгоритма Split-All-Cold .....	15
1.3 Описание алгоритма Align-Blocks .....	16
2 Профилирование .....	17
2.1 LLVM .....	18
2.2 LLVM-BOLT .....	20
3 Исследование и разработка тестовых файлов.....	21
3.1 Подготовка окружения.....	21
3.2 Работа с бенчмарком.....	24
3.3 Утилита time.....	27
3.4 Тестирование алгоритмов оптимизации размещения базовых блоков ..	27
3.5 Измерение временных показателей для алгоритма Reorder-Blocks .....	29
3.6 Измерение временных показателей для алгоритма Split-All-Cold.....	32
3.7 Измерение временных показателей для алгоритма Align-Blocks .....	33
3.8 Измерение временных показателей для трех алгоритмов, запущенных одновременно.....	34
4 Сравнение результатов модификации .....	36
5 Оценка влияния предложенных технических и организационных изменений проекта, программы, предприятия.....	42
ЗАКЛЮЧЕНИЕ.....	51
СПИСОК ЛИТЕРАТУРЫ .....	53
ПРИЛОЖЕНИЕ А .....	54

## ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

**Low Level Vitrual Machine (LLVM)** – это компиляторная инфраструктура с открытым исходным кодом, которая обеспечивает разработку и оптимизацию программного обеспечения. LLVM представляет собой набор инструментов, библиотек и технологий, которые позволяют компилировать программы в промежуточное представление (IR – Intermediate Representation) и далее оптимизировать и генерировать код для различных целевых платформ.

**Binary Optimization and Layout Tool (BOLT)** – это инструмент оптимизации бинарного кода. Предназначен для улучшения производительности исполняемых файлов путем оптимизации размещения инструкций в памяти (code layout optimization) и других оптимизаций, направленных на уменьшение времени выполнения программ.

**Post-linkage optimization (PLO)** – это методика оптимизации программного обеспечения, которая направлена на оптимизацию программы после этапа линковки. Линковка представляет собой процесс объединения объектных файлов или библиотек в один исполняемый или разделяемую библиотеку. Традиционно большинство оптимизаций выполняются на этапе компиляции до процесса линковки.

**Code-layout optimization (CLO)** – Оптимизация расположения кода (Code-layout optimization) – это процесс оптимизации размещения инструкций в памяти с целью улучшения производительности программы. При выполнении программы процессор загружает инструкции из памяти в свой кэш и выполняет их. Оптимизация расположения кода стремится улучшить локальность данных и уменьшить промахи в кэше, что приводит к более эффективной работе процессора.

**C-Language Family Fronted (Clang)** – это компилятор с открытым исходным

кодом, разработанный в рамках проекта LLVM. Он представляет собой семейство компиляторов для различных языков программирования, включая C, C++, Objective-C и Objective-C++.

**Бенчмарк (benchmark)** – это набор тестов для измерения и оценки производительности компьютерных систем, программного обеспечения, аппаратного обеспечения или конкретных компонентов системы.

**IR (Intermediate Representation)** – это промежуточный уровень абстракции в компиляторах и инструментах для анализа программного обеспечения. Он представляет собой форму представления программы, которая обычно находится между исходным кодом программы и машинным кодом.

**Базовый блок (basic block)** – это последовательность инструкций в программе, в которой выполнение начинается с самого начала и продолжается последовательно до точки безусловного перехода (ветвления) или завершения блока.

**Профилирование (profiling)** – это процесс измерения и анализа производительности программы или системы с целью выявления узких мест, оптимизации и улучшения ее работы.



## ВВЕДЕНИЕ

В современном программировании и компиляции, оптимизация играет важную роль в повышении производительности программ и улучшении их эффективности. Одной из ключевых задач оптимизации является размещение базовых блоков в программном коде, которое может существенно влиять на производительность выполнения программы. Оптимальное размещение базовых блоков позволяет минимизировать промахи в кэше процессора, улучшить предсказание переходов и обеспечить более эффективное использование ресурсов системы.

LLVM (Low-Level Virtual Machine) является мощным инструментом для компиляции программного кода, широко применяемым в академических и промышленных сферах. Он использует промежуточное представление (IR), которое представляет программу в форме, удобной для анализа и оптимизации.

Целью данного исследования является изучение и сравнительный анализ различных алгоритмов оптимизации размещения базовых блоков в LLVM.

Объектом исследования являются алгоритмы оптимизации размещения базовых блоков.

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

Наша задача состоит в исследовании существующих алгоритмов переупорядочивания базовых блоков, размещения базовых блоков в холодных (редко используемых) функциях и алгоритмов выравнивания базовых блоков, сравнительном анализе временных данных. Мы рассмотрим влияние различных факторов, таких как размер программы, характеристики аппаратной платформы и особенности кода, на производительность и эффективность оптимизации размещения базовых блоков.

Полученные результаты исследования будут полезны для разработчиков компиляторов и оптимизаторов, которые стремятся улучшить производительность программ и оптимизировать использование ресурсов системы. Они помогут определить наиболее эффективные алгоритмы оптимизации размещения базовых блоков в LLVM и дадут рекомендации по выбору оптимальных стратегий оптимизации для различных сценариев и приложений.

## **1 Обзор подходов к оптимизации размещения базовых блоков в программном коде**

Существует несколько ключевых методов оптимизации размещения базовых блоков [11]. Рассмотрим основные из них:

- Reorder-Blocks;
- Split-All-Cold;
- Align-Blocks.

Для начала разберемся, что такое базовые блоки, для чего нужны и как размещаются. Базовый блок представляет собой последовательность инструкций, которая начинается с входящего перехода и заканчивается исходящим переходом или завершающей инструкцией. Внутри базового блока нет переходов в другие базовые блоки. Он представляет собой фундаментальную структурную единицу программы

Вот общая последовательность действий, которая может быть применена для распределения базовых блоков:

1. Анализ потока управления: В этом шаге строится граф потока управления (Control Flow Graph, CFG), который представляет связи между базовыми блоками и переходами в программе. Этот анализ помогает определить структуру и связи между базовыми блоками.

2. Определение часто выполняемых участков кода: Используя информацию о профилировании [12] или других методах анализа, можно выделить участки кода, которые выполняются наиболее часто.

3. Оптимизация порядка расположения базовых блоков: Существует множество алгоритмов и подходов, которые могут быть использованы для оптимизации распределения базовых блоков. Некоторые из них включают эвристические методы, анализ зависимостей данных и управления, использование информации о профилировании, а также техники перестановки инструкций.

4. Учет аппаратных ограничений: При распределении базовых блоков необходимо учитывать аппаратные ограничения, такие как размер кэш-памяти [] и доступ к регистрам. Некоторые алгоритмы оптимизации могут учитывать эти ограничения, чтобы обеспечить эффективное использование ресурсов.

5. Оценка производительности: После распределения проводится оценка производительности программы. Это может включать измерение времени выполнения, анализ использования ресурсов процессора и другие метрики производительности. Оценка помогает определить эффективность примененных оптимизаций и, при необходимости, внести дополнительные корректировки.

На рисунке 1 представлено два варианта представления физического макета.

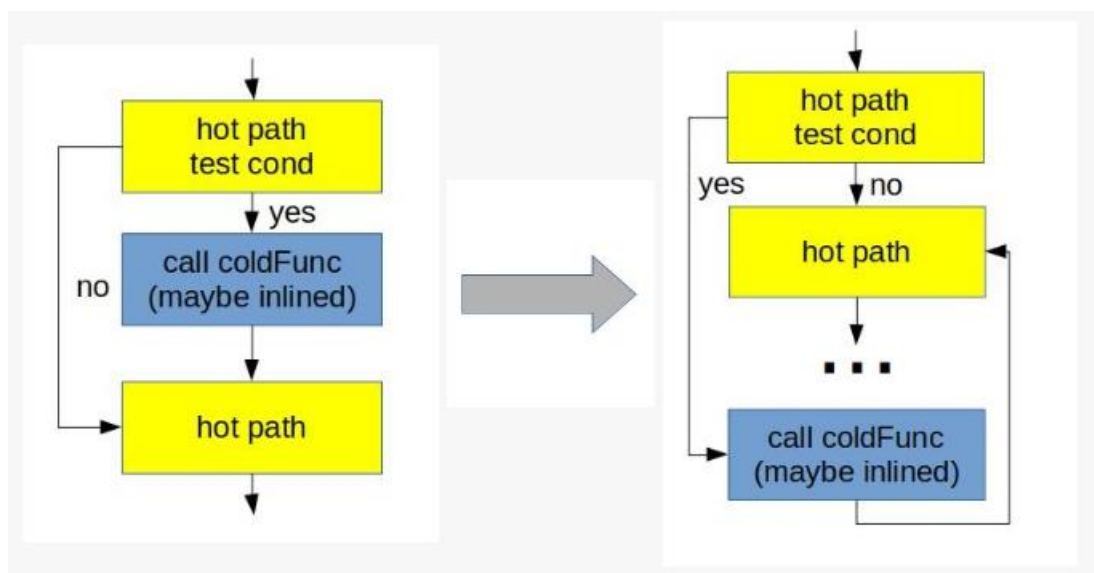


Рисунок 1 – Принцип работы размещения базовых блоков [13]

В нашем случае мы просто инвертировали условие из "if (cond)" в "if (!cond)". Стрелка подразумевает, что версия справа лучше, чем версия слева. И основная причина этого заключается в том, что мы сохраняем последовательность выполнения кода между активными участками. Неисполняемые ветвления в своей сущности более эффективны, чем исполняемые. К тому же, второй вариант обеспечивает более эффективное использование кэша L1 I-cache и uop-cache (DSB).

Распределение базовых блоков в программном коде [13] имеет свои преимущества и недостатки, которые стоит учитывать.

Правильное распределение базовых блоков может привести к улучшению производительности программы. Это может произойти благодаря повышению предсказуемости переходов, сокращению задержек чтения/записи в память, оптимальному использованию регистров и кэш-памяти процессора, а также улучшению локальности кода. Кроме этого оно может помочь улучшить работу с памятью, так как оптимальное размещение инструкций может снизить задержки при доступе к памяти и повысить эффективность использования кэш-памяти.

Однако процесс размещения базовых блоков может быть сложным и требовать значительных вычислительных ресурсов. Алгоритмы и методы оптимизации, используемые для распределения базовых блоков, могут быть сложными в реализации и требовать дополнительного времени компиляции. В некоторых случаях данный метод может привести к увеличению размера исполняемого кода. Это может быть вызвано введением дополнительных переходов или инструкций, необходимых для оптимального размещения базовых блоков. А также немаловажной особенностью является то, что оптимальное распределение базовых блоков может зависеть от конкретных характеристик аппаратной платформы, на которой будет выполняться программа.

### **1.1 Описание алгоритма Reorder-Blocks**

Алгоритм Reorder-Blocks [11] представляет собой стратегию переупорядочивания базовых блоков в исполняемом файле. Он используется для оптимизации размещения базовых блоков с целью улучшения производительности программы.

Алгоритм Reorder-Blocks имеет несколько вариантов, которые можно указывать в качестве аргумента (опции):

- none – Опция указывает на то, что базовые блоки останутся в том порядке, в котором они были в исходном исполняемом файле.

- `cache` – В этом варианте базовые блоки переупорядочиваются с учетом оптимизации кэша. Алгоритм пытается минимизировать кэш-промахи, упорядочивая базовые блоки таким образом, чтобы инструкции, которые часто исполняются вместе, находились близко друг к другу в памяти.
- `cache+` – Этот вариант является улучшенной версией `cache`. Он также учитывает не только кэш-промахи, но и процессорные промахи, чтобы создать еще более эффективное размещение базовых блоков.
- `reverse` – Данная опция выполняет обратное переупорядочивание базовых блоков. Она меняет порядок блоков на противоположный, т.е. последний блок становится первым, предпоследний – вторым и так далее.
- `normal` – Эта опция указывает на использование стандартного алгоритма переупорядочивания базовых блоков в LLVM-Bolt. Блоки переупорядочиваются в соответствии с выбранным алгоритмом оптимизации.
- `branch-predictor` – Данная опция используется для оптимизации порядка базовых блоков с учетом предсказания ветвлений. Она пытается упорядочить блоки таким образом, чтобы минимизировать количество неудачных предсказаний ветвлений.
- `ext-tsp` – Эта опция применяет алгоритм коммивояжера (TSP) для переупорядочивания базовых блоков. Она стремится найти оптимальный порядок блоков, чтобы минимизировать общую стоимость переходов между ними.
- `cluster-shuffle` – Данная опция используется для переупорядочивания базовых блоков с помощью алгоритма кластеризации. Она группирует блоки в кластеры и затем переставляет их, чтобы улучшить кэш-промахи и предсказание ветвлений.

Выбор конкретного варианта Reorder-Blocks зависит от характеристик программы и целей оптимизации. Различные приложения могут иметь разные требования и эффективность при использовании разных стратегий переупорядочивания базовых блоков.

## **1.2 Описание алгоритма Split-All-Cold**

Алгоритм Split-All-Cold [11] используется для оптимизации размещения базовых блоков в холодных (редко используемых) функциях. Этот алгоритм позволяет улучшить кэширование и предсказание ветвлений для холодных функций, что может привести к улучшению производительности программы.

Когда программа выполняется, некоторые функции вызываются редко, и их базовые блоки имеют низкую вероятность попадания в кэш и плохое предсказание ветвлений. Опция Split-All-Cold позволяет разделить холодные функции на меньшие части или базовые блоки, чтобы повысить вероятность попадания в кэш и улучшить предсказание ветвлений.

Алгоритм Split-All-Cold анализирует холодные функции [13] и их базовые блоки, определяет части кода, которые редко исполняются, и разделяет их на более мелкие блоки. Это может быть достигнуто путем вставки дополнительных ветвлений или использования специальных инструкций для разделения блоков. Результатом является более гибкое размещение холодных блоков в памяти, что может привести к лучшей эффективности кэширования и предсказания ветвлений.

На представленном рисунке 2 можно заметить, как мы сгруппировали функции foo, bar и zoo таким образом, что теперь весь их код уместается всего в 3 строки кэша.

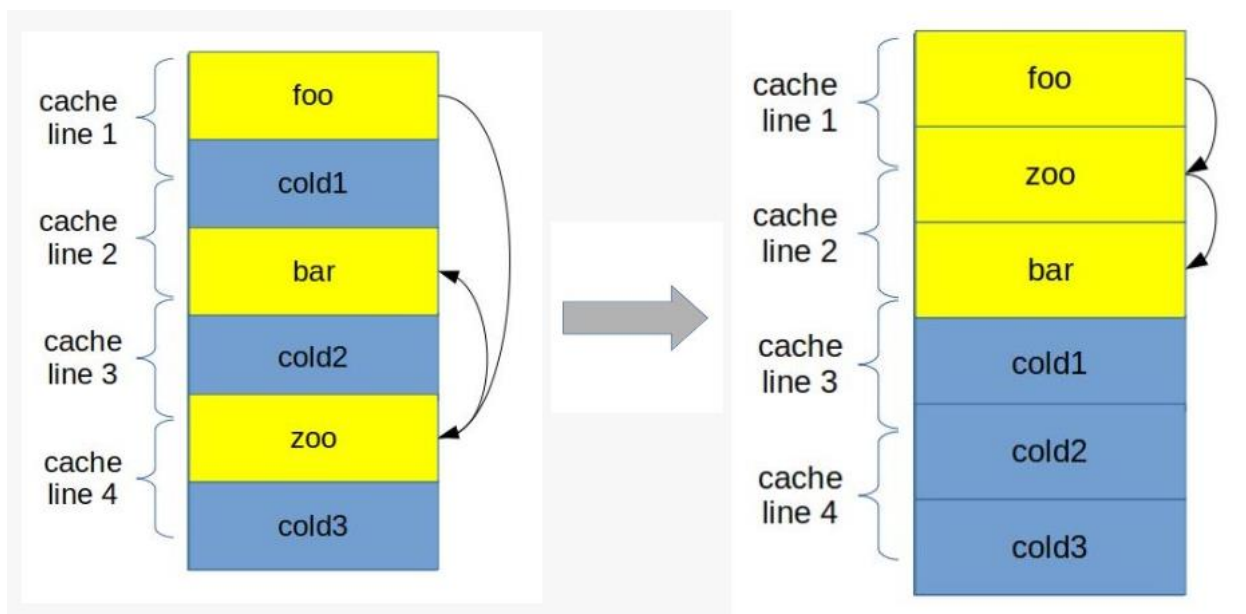


Рисунок 2 – Принцип работы алгоритма Split-All-Cold

Опция Split-All-Cold [11] в LLVM-Bolt полезна в ситуациях, когда имеется большое количество холодных функций или базовых блоков, которые могут быть эффективно разделены и размещены. Это может снизить негативное влияние холодных блоков на производительность программы и повысить общую эффективность выполнения.

### 1.3 Описание алгоритма Align-Blocks

Опция Align-Blocks в LLVM-Bolt [11] позволяет оптимизировать размещение базовых блоков с целью выравнивания их по границам кэш-линий или других оптимальных границ, которые определяются аппаратными особенностями процессора или специфическими требованиями к производительности.

При использовании опции Align-Blocks в LLVM-Bolt, алгоритм будет переразмещать базовые блоки в исполняемом коде таким образом, чтобы они выравнивались в соответствии с указанными оптимальными границами. Это может потребовать перестановки блоков, вставки дополнительных инструкций или других манипуляций с кодом. Целью является создание более эффективной последовательности базовых блоков для улучшения производительности программы.



## 2 Профилирование

Профилирование [12] – это процесс сбора и анализа данных о выполнении программы или системы с целью оптимизации ее производительности или выявления проблемных участков. В процессе профилирования собираются различные метрики, такие как время выполнения, использование ресурсов (память, процессорное время), количество вызовов функций и другие события, связанные с выполнением программы.

Профилирование позволяет определить узкие места в программе, т.е. места, где происходит значительное потребление ресурсов или затраты времени для дальнейшей оптимизации программы. Профилирование также может использоваться для анализа поведения программы в различных сценариях выполнения или при различных входных данных. Информация, собранная о программе в процессе ее работы, называется профилем программы

Существуют различные методы профилирования, включая профилирование времени выполнения, профилирование памяти, профилирование вызовов функций и другие. Каждый метод имеет свои особенности и предоставляет информацию о конкретных аспектах выполнения программы.

В контексте работы с LLVM наиболее распространенными профилировщиками являются:

- LLVM-BOLT;
- Perf;
- Dynamorio.

Все эти инструменты осуществляют сбор информации во время выполнения программы, что называется динамическим сбором данных. В своей работе я использую только LLVM-Bolt [14], поэтому только о нем далее пойдет речь.

## 2.1 LLVM

Для того, чтобы разобраться и понять, что такое LLVM-BOLT, для начала нужно разобраться с более фундаментальной вещью, а именно с LLVM. Изначально LLVM [8] задумывался как мощный инструмент для разработки компиляторов и оптимизаций. Несмотря на то, что его аббревиатура "Low-Level Virtual Machine" указывает на первоначальное предназначение, со временем LLVM превратился в гораздо более универсальное и востребованное решение.

LLVM – это компиляторная инфраструктура с открытым исходным кодом, которая предоставляет набор инструментов и библиотек для разработки компиляторов и других программных средств. Основная цель LLVM заключается в обработке и оптимизации промежуточного представления программы для получения эффективного машинного кода.

С течением времени LLVM прошел значительное развитие и нашел широкое применение в различных проектах. Он стал неотъемлемой частью компилятора GCC и является основой для разработки проекта Clang [9]. Комбинация Clang/LLVM получила широкую известность в мире благодаря своим многочисленным преимуществам, включая легкость внедрения статических проверок и поддержку операционной системы Windows.

LLVM также предлагает обширный набор инструментов, включая, например, LLVM-Bolt, который использует промежуточное представление LLVM для анализа, оптимизации и преобразования кода. Использование LLVM позволяет точно настраивать различные параметры программы и изменять профилирование для достижения максимальной оптимизации кода.

Основой механизма оптимизации, преобразования и анализа в LLVM являются проходы. Они предоставляют возможность обходить функции, циклы

и модули, выполняя необходимые действия и оптимизации. Диспетчер проходов отвечает [9] за планировку, регистрацию и выполнение проходов, а также обеспечивает гибкую настройку и контроль их выполнения.

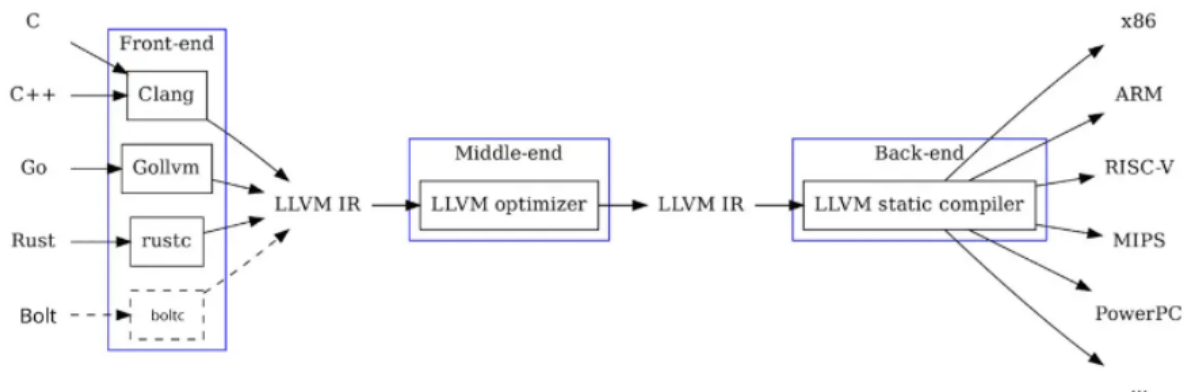


Рисунок 3 – Процесс оптимизации через LLVM

LLVM обладает рядом важных особенностей, которые делают его популярным и мощным инструментом разработки. Кроме универсального промежуточного представления, LLVM предлагает широкий спектр оптимизаций [8], которые могут значительно улучшить производительность программы, поддерживает несколько языков программирования, включая C, C++, Objective-C, Swift, Rust, Python и другие, поддерживает несколько платформ, включая Windows, macOS, Linux, iOS и Android. На рисунке 3 представлена архитектура LLVM [11], которая позволяет лучше понять ее внутреннее устройство и принципы работы.

Помимо этого, LLVM предоставляет API, позволяющий разработчикам создавать собственные проходы, инструменты и расширения, имеет большое и активное сообщество разработчиков, которые вносят вклад в его развитие, поддержку и расширение. Все это позволяет адаптировать LLVM под уникальные требования и задачи проекта, расширяя его функциональность и возможности, обеспечивая постоянное развитие и обновление инструментов и функций LLVM.

## 2.2 LLVM-BOLT

LLVM-Bolt (Binary Optimization and Layout Tool) [15] является инструментом, разработанным на основе LLVM, который предназначен для оптимизации исполняемых файлов (бинарных файлов) на уровне машинного кода. Он был разработан с целью улучшить производительность и эффективность исполняемых файлов, особенно в контексте приложений с большим объемом кода.

История разработки LLVM-Bolt началась в 2015 году, когда разработчики компании Facebook заметили, что оптимизации LLVM на уровне исходного кода не всегда достаточно для достижения наилучшей производительности исполняемых файлов. Они решили исследовать возможности оптимизации бинарных файлов, чтобы улучшить производительность программного обеспечения. И в 2016 году команда разработчиков Facebook анонсировала LLVM-Bolt как открытое программное обеспечение и представила его в качестве проекта LLVM. LLVM-Bolt был включен в набор инструментов LLVM [14] и стал доступен для разработчиков.

Основная цель LLVM-Bolt – это снижение времени выполнения программы и улучшение производительности исполняемых файлов путем оптимизации размещения кода в памяти и преобразования машинного кода. Он выполняет такие операции, как перераспределение функций в памяти для улучшения кэширования, реструктуризацию кода для лучшей предсказуемости ветвлений и другие оптимизации на уровне машинного кода.

За последние годы LLVM-Bolt получил значительное внимание и стал популярным инструментом в сообществе разработчиков. Его использование позволяет значительно улучшить производительность исполняемых файлов, особенно в крупных проектах с обширным объемом кода.

### **3 Исследование и разработка тестовых файлов**

В данном разделе будет проведена серия тестов, чтобы выявить наиболее эффективные алгоритмы оптимизации размещения базовых блоков. Для этого был использован бенчмарк 531.deepsjeng\_r [5], основанный на программе Deep Sjeng WC2008 [10], которая является чемпионом мира по скоростным шахматам на компьютерах в 2008 году. Deep Sjeng – это переработанная версия программы Sjeng-Free, разработанная с целью достижения максимальной игровой мощности.

Был выбран этот бенчмарк, так как он представляет собой сложную и вычислительно интенсивную задачу, которая требует эффективной оптимизации для достижения максимальной производительности. Применение различных алгоритмов оптимизации размещения базовых блоков позволит нам исследовать, какие подходы наиболее эффективны в контексте данной задачи.

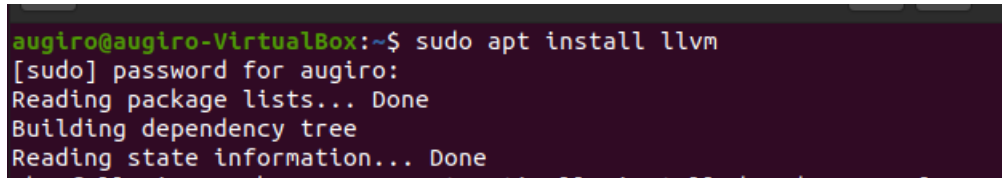
#### **3.1 Подготовка окружения**

Для исследования была выбрана операционная система Ubuntu [4]. Ubuntu – это бесплатная и открытая операционная система на базе Linux, предназначенная для персональных компьютеров. Она является одним из самых популярных дистрибутивов Linux благодаря своей простоте использования и активной поддержке. Одним из преимуществ Ubuntu для нашей задачи является возможность установки дополнительных программ через командную строку.

Для анализа мы используем Ubuntu самой последней версии 22.04.2 LTS, имеющая самый широкий функционал.

Для установки Ubuntu нам потребуется виртуальная машина, такая как VirtualBox [6] или VMware [7]. Мы загружаем официальный образ жесткого диска [4] с официального сайта и создаем виртуальную операционную систему Ubuntu в выбранной виртуальной машине.

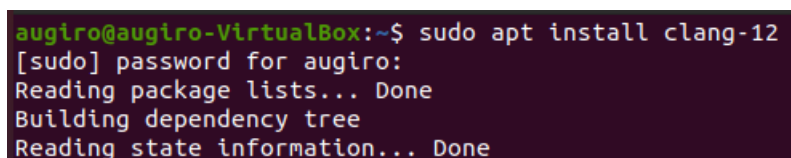
После создания виртуальной среды мы устанавливаем LLVM и Clang [9]. Для установки LLVM мы вводим команду "sudo apt install llvm" [8] в командной строке, которую можно вызвать, нажав кнопку "Терминал" в боковом меню. Результат установки показан на рисунке 4."



```
augiro@augiro-VirtualBox:~$ sudo apt install llvm
[sudo] password for augiro:
Reading package lists... Done
Building dependency tree
Reading state information... Done
```

Рисунок 4 – Установка LLVM

После установки LLVM требуется установить Clang. В нашем исследовании мы выбрали Clang версии 12, так как именно она включает все команды, которые представляют для нас интерес. Чтобы установить Clang-12, выполните следующую команду в командной строке: "sudo apt install clang-12". Результат установки отображен на рисунке 5.



```
augiro@augiro-VirtualBox:~$ sudo apt install clang-12
[sudo] password for augiro:
Reading package lists... Done
Building dependency tree
Reading state information... Done
```

Рисунок 5 –Установка Clang-12

Далее нам потребуется установить профилировщик LLVM-BOLT. В нашей работе мы решили использовать Docker для установки и настройки LLVM-Bolt. Этот инструмент контейнеризации предоставляет удобный и надежный способ запуска LLVM-Bolt в изолированной среде.

Для установки LLVM-Bolt через Dockerfile [6] мы создали текстовый файл, содержащий все необходимые инструкции для сборки Docker-образа. После создания Dockerfile мы использовали команду "Docker build" для создания образа на основе указанного файла.

Получив Docker-образ, мы смогли запустить контейнер, в котором успешно использовали LLVM-Bolt. Преимущество установки таким методом заключается в том, что Docker обеспечивает однородную среду выполнения

LLVM-Bolt на разных системах, без необходимости вручную устанавливать и настраивать все зависимости.

На рисунках 6 и 7 представлены визуальные отображения загруженного Docker-образа и запущенного Docker-контейнера, позволяющие наглядно представить их внешний вид.

```
augiro@augiro-VirtualBox:~/Test/build/bin$ sudo docker images
[sudo] password for augiro:
REPOSITORY          TAG             IMAGE ID        CREATED         SIZE
llvm-bolt            latest          5a97b23b49c0   4 weeks ago    134MB
llvm_bolt            latest          c13d853d8191   4 weeks ago    134MB
<none>               <none>          321c04ad3f77   4 weeks ago    134MB
snowstep/llvm        latest          69519471b948   4 weeks ago    1.16GB
```

Рисунок 6 – Скаченный образ Docker

```
augiro@augiro-VirtualBox:~$ sudo docker ps
[sudo] password for augiro:
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS
82447eef2505   5a97b23b49c0   "/bin/bash"             21 hours ago   Up 21 hours
```

Рисунок 7 – Контейнер Docker

Далее при помощи команды `sudo docker run -it <Docker IMAGE>` запускаем контейнер Docker. После этого вводим новую команду: `llvm-bolt --version`, для того чтобы убедиться, что имеется все необходимое для установки профилировщика [12], рисунок 8.

```
augiro@augiro-VirtualBox:~/Test/build/bin$ sudo docker run -it 5a97b23b49c0
root@82447eef2505:/# 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
root@82447eef2505:/#
```

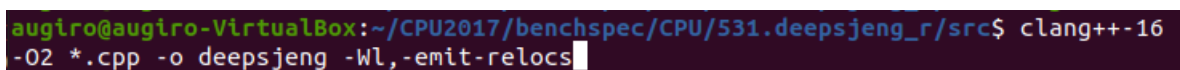
Рисунок 8 – Запуск контейнера Docker и вывод информации о версии LLVM-BOLT

### 3.2 Работа с бенчмарком

Теперь у нас есть все необходимое для модернизации и работы с файлами. Далее нам потребуется загрузить сами файлы и начать работу с ними. Для работы с бенчмарком 531.deepsjeng\_r [10] требуется загрузить его с ресурса GitHub [5], где он доступен для общего использования. Чтобы его получить, нужно выполнить команду "git clone <URL>" в командной строке. После завершения загрузки всех файлов, необходимо перейти в папку src в 531.deepsjeng\_r и открыть командную строку, нажав правой кнопкой мыши на свободном месте и выбрав "Open in Terminal" (Открыть в терминале).

Бенчмарк 531.deepsjeng\_r является достаточно крупным, поэтому перед использованием его необходимо скомпилировать для получения бинарного файла [3]. Для этого мы можем воспользоваться командой, показанной на рисунке 9.

Здесь мы компилируем бенчмарк без оптимизации, но с флагом -O2, который включает оптимизацию компилятора. В результате получаем бинарный файл deepsjeng, который будет использоваться для дальнейших тестов и анализа.



```
augiro@augiro-VirtualBox:~/CPU2017/benchspec/CPU/531.deepsjeng_r/src$ clang++-16  
-O2 *.cpp -o deepsjeng -Wl,-emit-relocs
```

Рисунок 9 – Сборка бенчмарка 531.deepsjeng\_r

Далее мы перейдем к созданию инструментированной версии бинарного файла deepsjeng, что позволит нам собрать профиль программы перед проведением оптимизации. Этот профиль будет содержать информацию о данных и работе программы, что поможет нам в дальнейшем анализе.

Для создания инструментированной версии файла мы можем воспользоваться командой, показанной на рисунке 10.



```

augiro@augiro-VirtualBox:~/CPU2017/benchspec/CPU/531.deepsjeng_r/src$ llvm-bolt-16 -instru
ment -o deepsjeng-instr deepsjeng
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 0xc00000, offset 0xc00000
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 120 functions in the binary (0.0%) have non-empty execution profile
BOLT-INFO: the input contains 17 (dynamic count : 0) opportunities for macro-fusion optimi
zation that are going to be fixed
BOLT-INSTRUMENTER: Number of indirect call site descriptors: 3
BOLT-INSTRUMENTER: Number of indirect call target descriptors: 114

```

Рисунок 10 – Создание инструментального бинарного файла

Делаем прогон инструментации, чтобы получить профиль, по которому можно оптимизировать. Команда приведена на рисунке 11.

```

augiro@augiro-VirtualBox:~/CPU2017/benchspec/CPU/531.deepsjeng_r/src$ ./deepsjeng-instr ../data/test/input/test.txt
Deep Sjeng version 3.2 SPEC, Copyright (C) 2000-2009 Gian-Carlo Pascutto
Allocated 15000000 hash entries, totalling 720000000 bytes.
Analyzing 15 plies...
+-----+
8 | *R |   |   | *Q | *K |   |   | *R |
+-----+
7 | *P | *P | *P |   | *B |   | *P | *P |
+-----+
6 |   |   | *N |   | *P |   |   |   |
+-----+
5 |   |   |   | *P | P |   | *N |   |
+-----+
4 |   |   |   | P |   |   | *B |   |
+-----+
3 |   |   | P | B |   | N | N |   |
+-----+
2 | P | P |   |   |   |   | P | P |
+-----+
1 | R |   | B | Q | K |   |   | R |
+-----+
  a  b  c  d  e  f  g  h

```

Рисунок 11 – Получение профиля

Теперь можно переходить непосредственно к оптимизации программ. В процессе работы профилировщика будут выводиться различные оптимизированные данные. Пример процесса оптимизации представлен на рисунке 12.

```

augiro@augiro-VirtualBox:~/CPU2017/benchspec/CPU/531.deepsjeng_r/src$ llvm-bolt-16 deepsj
eng -o deepsjeng-opt -data=/tmp/prof.fdata -reorder-blocks=cache+ -reorder-functions=hfsor
t -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 0xc00000, offset 0xc00000
BOLT-INFO: enabling relocation mode
BOLT-INFO: enabling lite mode
BOLT-INFO: pre-processing profile using branch profile reader
BOLT-INFO: forcing -jump-tables=move as PIC jump table was detected in function _Z22Setup
PrecalculatedData
BOLT-INFO: 80 out of 120 functions in the binary (66.7%) have non-empty execution profile
BOLT-INFO: 5 functions with profile could not be optimized
BOLT-INFO: profile for 1 objects was ignored
BOLT-INFO: the input contains 20 (dynamic count : 37959619) opportunities for macro-fusio
n optimization. Will fix instances on a hot path.

```

Рисунок 12 – Процесс оптимизации

Функция `llvm-bolt-16` является частью инструмента LLVM-Bolt [15], который используется для оптимизации размещения базовых блоков в исполняемых файлах. Ниже приведено описание каждого аргумента в команде:

- `deepsjeng`: Это входной исполняемый файл, который будет оптимизирован с помощью LLVM-Bolt.
- `deepsjeng-opt`: Этот аргумент указывает на выходной файл, в который будет сохранен оптимизированный исполняемый файл.
- `-data=/tmp/prof.fdata`: Этот аргумент указывает путь к файлу данных профилирования (`prof.fdata`), который содержит информацию о профилировании программы. Эта информация будет использоваться для лучшего размещения базовых блоков.
- `-reorder-blocks=cache+`: Этот аргумент указывает на стратегию переупорядочивания базовых блоков. В данном случае, `cache+` указывает на использование стратегии, оптимизирующей кэш-промахи.
- `-reorder-functions=hfsort`: Этот аргумент указывает на стратегию переупорядочивания функций. В данном случае, `hfsort` указывает на использование стратегии сортировки функций по эвристикам Hot/Cold.

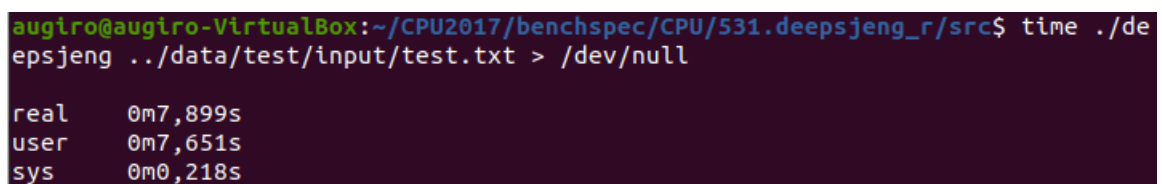
- `-split-functions=2`: Этот аргумент указывает на количество частей, на которые будут разделены функции в процессе оптимизации.
- `-split-all-cold`: Этот аргумент указывает на разделение всех холодных функций на отдельные секции.
- `-split-eh`: Этот аргумент указывает на разделение функций, связанных с обработкой исключений (exception handling).
- `-dyno-stats`: Этот аргумент указывает на сбор и вывод статистики о динамическом поведении программы после оптимизации.

### 3.3 Утилита `time`

В дальнейшем нам потребуется утилита `time` [8]. Она представляет собой инструмент для измерения времени выполнения программы или конкретного кодового фрагмента. Она позволяет определить, сколько времени занимает выполнение определенной операции или функции внутри программы.

`Time` является встроенной утилитой в командной оболочке, которая предоставляет информацию о времени выполнения команды или программы. При ее использовании в LLVM, можно получить информацию о различных метриках времени, таких как общее время выполнения программы, время, затраченное на определенные этапы компиляции или выполнения, а также использование памяти. Это позволяет анализировать производительность программы и оптимизировать ее для улучшения ее работы.

Пример использования утилиты `time` приведен на рисунке 13.



```
auglro@auglro-VirtualBox:~/CPU2017/benchspec/CPU/531.deepsjeng_r/src$ time ./deepsjeng ../data/test/input/test.txt > /dev/null
real    0m7,899s
user    0m7,651s
sys     0m0,218s
```

Рисунок 13 – Утилита `time`

### 3.4 Тестирование алгоритмов оптимизации размещения базовых блоков

В данной работе было решено исследовать и протестировать несколько алгоритмов оптимизации в LLVM-Bolt, включая `Reorder-Blocks`, `Split-All-Cold`

и Align-Blocks [11]. Каждый из этих алгоритмов направлен на улучшение производительности исполняемого кода путем оптимизации размещения базовых блоков.

Алгоритм Reorder-Blocks позволяет переупорядочить базовые блоки в исполняемом коде с целью повышения эффективности работы процессора и использования кэшей. В данной работе были выбраны опции `cache+`, `branch-predictor`, `ext-tsp` и `cluster-shuffle` для алгоритма Reorder-Blocks. Выбор данных опций для исследования был обусловлен их потенциальной способностью улучшить производительность исполняемого кода и особенностями конкретной задачи. Подробнее об этих опциях написано в 1 разделе.

Пример оптимизации при помощи алгоритма Reorder-Blocks с опцией `cache+` приведен на рисунке 14.

A screenshot of a terminal window with a dark background. The prompt is 'augiro@augiro-VirtualBox:~/CPU2017/benchspec/CPU/531.deepsjeng\_r/src\$'. The command entered is 'llvm-bolt-16 deepsjeng -o deepsjeng-opt -data=/tmp/prof.fdata -reorder-blocks=cache+'. The output is not visible.

Рисунок 14 – Применение алгоритма Reorder-Blocks с опцией `cache+`

Алгоритм Split-All-Cold, который также будет исследован в данной работе, направлен на разделение функций и размещение "холодных" (малоиспользуемых) частей кода в отдельные секции.

Пример оптимизации при помощи алгоритма Split-All-Cold приведен на рисунке 15.

A screenshot of a terminal window with a dark background. The prompt is 'augiro@augiro-VirtualBox:~/CPU2017/benchspec/CPU/531.deepsjeng\_r/src\$'. The command entered is 'llvm-bolt-16 deepsjeng -o deepsjeng-opt -data=/tmp/prof.fdata -split-all-cold'. The output is not visible.

Рисунок 15 – Применение алгоритма Split-All-Cold

Алгоритм Align-Blocks, который также будет исследован, осуществляет оптимизацию размещения базовых блоков путем их выравнивания по оптимальным границам, например, границам кэш-линий.

Пример оптимизации при помощи алгоритма Align-Blocks приведен на рисунке 16.

```
augiro@augiro-VirtualBox:~/CPU2017/benchspec/CPU/531.deepsjeng_r/src$ llvm-bo  
lt-16 deepsjeng -o deepsjeng-opt -data=/tmp/prof.fdata -align-blocks
```

Рисунок 16 – Применение алгоритма Align-Blocks

Далее, для оценки эффективности проведенных изменений, необходимо сравнить время работы неоптимизированной и оптимизированной программы для каждого алгоритма. Для этого проведем проверку на разной нагрузке, используя файлы, находящиеся в самом бенчмарке deepsjeng:

- test – минимальная нагрузка;
- train – средняя нагрузка;
- refrate – высокая нагрузка.

Для измерения времени работы на нагрузке test используются следующие команды: "time ./deepsjeng-opt ../data/test/input/test.txt > /dev/null" для оптимизированной программы и "time ./deepsjeng ../data/test/input/test.txt > /dev/null" для неоптимизированной программы. Для остальных нагрузок все аналогично, меняется только папка. Полученные результаты позволят сравнить время компиляции файла до и после внесенных изменений.

### 3.5 Измерение временных показателей для алгоритма Reorder-Blocks

Произведем замеры времени до и после оптимизации для опции cache+ алгоритма Reorder-Blocks на нагрузках test, train и refrate. Результаты тестирования можно пронаблюдать на рисунках 17, 18 и 19.

```
real    0m7,097s  
user    0m6,898s  
sys     0m0,196s  
augiro@augiro-Virtua  
  
real    0m7,249s  
user    0m7,094s  
sys     0m0,153s
```

Рисунок 17 – Время работы опции cache+ алгоритма Reorder-Blocks на нагрузке test до и после оптимизации

```

real    1m0,320s
user    1m0,054s
sys     0m0,242s
augiro@augiro-Virt

real    1m1,251s
user    1m1,031s
sys     0m0,212s
augiro@augiro-Virt

```

Рисунок 18 – Время работы опции cache+ алгоритма Reorder-Blocks на нагрузке train до и после оптимизации

```

real    5m49,879s
user    5m49,673s
sys     0m0,175s
augiro@augiro-Virt

real    5m47,968s
user    5m47,714s
sys     0m0,226s
augiro@augiro-Virt

```

Рисунок 19 – Время работы опции cache+ алгоритма Reorder-Blocks на нагрузке refrate до и после оптимизации

Произведем замеры времени до и после оптимизации для опции branch-predictor алгоритма Reorder-Blocks на нагрузках test, train и refrate. Результаты тестирования можно увидеть на рисунках 20, 21 и 22.

```

real    0m7,204s
user    0m6,987s
sys     0m0,212s
augiro@augiro-Virt

real    0m7,195s
user    0m6,919s
sys     0m0,271s
augiro@augiro-Virt

```

Рисунок 20 – Время работы опции branch-predictor алгоритма Reorder-Blocks на нагрузке test до и после оптимизации

```

real    1m2,357s
user    1m2,110s
sys     0m0,242s
augiro@augiro-Virt

real    1m0,715s
user    1m0,479s
sys     0m0,217s
augiro@augiro-Virt

```

Рисунок 21 – Время работы опции branch-predictor алгоритма Reorder-Blocks на нагрузке train до и после оптимизации

```

real    5m47,811s
user    5m47,549s
sys     0m0,224s
augiro@augiro-Virt

real    5m47,455s
user    5m47,199s
sys     0m0,216s

```

Рисунок 22 – Время работы опции branch-predictor алгоритма Reorder-Blocks на нагрузке refrate до и после оптимизации

Аналогично произведем замеры времени до и после оптимизации для опции ext-tsp алгоритма Reorder-Blocks на нагрузках test, train и refrate. Результаты тестирования можно увидеть на рисунках 23, 24 и 25.

```

real    0m7,188s
user    0m6,896s
sys     0m0,285s
augiro@augiro-Virt

real    0m7,224s
user    0m7,053s
sys     0m0,165s

```

Рисунок 23 – Время работы опции ext-tsp алгоритма Reorder-Blocks на нагрузке test до и после оптимизации

```

real    1m2,937s
user    1m2,741s
sys     0m0,174s
augiro@augiro-Virt

real    1m1,434s
user    1m1,164s
sys     0m0,259s

```

Рисунок 24 – Время работы опции ext-tsp алгоритма Reorder-Blocks на нагрузке train до и после оптимизации

```

real    5m48,668s
user    5m48,436s
sys     0m0,199s
augiro@augiro-Virt

real    5m45,242s
user    5m45,005s
sys     0m0,208s

```

Рисунок 25 – Время работы опции ext-tsp алгоритма Reorder-Blocks на нагрузке refrate до и после оптимизации

Далее произведем замеры времени до и после оптимизации для опции cluster-shuffle алгоритма Reorder-Blocks на нагрузках test, train и refrate. Результаты тестирования можно увидеть на рисунках 26, 27 и 28.

```
real    0m7,276s
user    0m7,061s
sys     0m0,210s
augiro@augiro-Virtu
real    0m7,188s
user    0m6,969s
sys     0m0,214s
```

Рисунок 26 – Время работы опции cluster-shuffle алгоритма Reorder-Blocks на нагрузке test до и после оптимизации

```
real    1m3,177s
user    1m2,962s
sys     0m0,207s
augiro@augiro-Virtu
real    1m3,907s
user    1m3,666s
sys     0m0,229s
```

Рисунок 27 – Время работы опции cluster-shuffle алгоритма Reorder-Blocks на нагрузке train до и после оптимизации

```
real    5m41,683s
user    5m41,384s
sys     0m0,223s
augiro@augiro-Virtu
real    5m48,479s
user    5m48,170s
sys     0m0,260s
```

Рисунок 28 – Время работы опции cluster-shuffle алгоритма Reorder-Blocks на нагрузке refrate до и после оптимизации

### 3.6 Измерение временных показателей для алгоритма Split-All-Cold

Произведем замеры времени до и после оптимизации для алгоритма Split-All-Cold на нагрузках test, train и refrate. Результаты тестирования можно пронаблюдать на рисунках 29, 30 и 31.



```

real    0m7,404s
user    0m7,245s
sys     0m0,157s
augiro@augiro-Virtu

real    0m7,615s
user    0m7,349s
sys     0m0,266s

```

Рисунок 29 – Время работы алгоритма Split-All-Cold на нагрузке test до и после оптимизации

```

real    1m1,974s
user    1m1,740s
sys     0m0,222s
augiro@augiro-Virtu

real    1m1,089s
user    1m0,899s
sys     0m0,180s

```

Рисунок 30 – Время работы алгоритма Split-All-Cold на нагрузке train до и после оптимизации

```

real    5m47,599s
user    5m47,393s
sys     0m0,184s
augiro@augiro-Virtu

real    5m49,446s
user    5m49,182s
sys     0m0,237s

```

Рисунок 31 – Время работы алгоритма Split-All-Cold на нагрузке refrate до и после оптимизации

### 3.7 Измерение временных показателей для алгоритма Align-Blocks

Произведем замеры времени до и после оптимизации для алгоритма Align-Blocks на нагрузках test, train и refrate. Результаты тестирования можно пронаблюдать на рисунках 32, 33 и 34.

```

real    0m7,352s
user    0m7,153s
sys     0m0,194s
augiro@augiro-Virt
real    0m7,464s
user    0m7,234s
sys     0m0,226s

```

Рисунок 32 – Время работы алгоритма Align-Blocks на нагрузке test до и после оптимизации

```

real    1m1,989s
user    1m1,777s
sys     0m0,202s
augiro@augiro-Virt
real    1m1,434s
user    1m1,185s
sys     0m0,232s

```

Рисунок 33 – Время работы алгоритма Align-Blocks на нагрузке train до и после оптимизации

```

real    5m42,535s
user    5m42,320s
sys     0m0,178s
augiro@augiro-VirtualB
real    5m45,031s
user    5m44,784s
sys     0m0,222s

```

Рисунок 34 – Время работы алгоритма Align-Blocks на нагрузке refrate до и после оптимизации

### 3.8 Измерение временных показателей для трех алгоритмов, запущенных одновременно.

Произведем замеры времени до и после оптимизации для алгоритмов Reorder-Blocks и Split-All-Cold, запущенных одновременно, на нагрузках test, train и refrate. В качестве опции для алгоритма Reorder-Blocks будем использовать ext-tsp. Результаты тестирования приведены на рисунках 35, 36 и 37.

```
real    0m7,210s
user    0m7,032s
sys     0m0,172s
augiro@augiro-Virtu

real    0m7,003s
user    0m6,747s
sys     0m0,254s
```

Рисунок 35 – Время работы двух алгоритмов, запущенных одновременно, на нагрузке test до и после оптимизации

```
real    1m2,310s
user    1m2,047s
sys     0m0,226s
augiro@augiro-Virtu

real    0m59,728s
user    0m59,474s
sys     0m0,244s
```

Рисунок 36 – Время работы двух алгоритмов, запущенных одновременно, на нагрузке train до и после оптимизации

```
augiro@augiro-Virtu

real    5m48,271s
user    5m47,977s
sys     0m0,259s
augiro@augiro-Virtu

real    5m43,212s
user    5m42,956s
sys     0m0,229s
```

Рисунок 37 – Время работы двух алгоритмов, запущенных одновременно, на нагрузке refrate до и после оптимизации

## 4 Сравнение результатов модификации

После получения результатов оптимизации можно произвести сравнительный анализ временных показателей. В ходе тестирования были применены три различные нагрузки для компиляции: test, train и retrate [5]. Каждая из этих нагрузок влияет на скорость компиляции и, соответственно, на результаты оптимизации. Для измерения времени была использована команда «./time».

В результате, мы получили временные результаты компиляции как для исходных версий программы, так и для оптимизированных версий с применением каждого из алгоритмов на трех различных нагрузках: низкой, средней и высокой. Результаты, которые были описаны в предыдущем разделе, представлены в таблице 1. В столбце "Нет" указаны максимальные значения времени компиляции без применения оптимизаций. Все результаты представлены в секундах.

Таблица 1 – Результаты тестирования

Нагрузка	Варианты оптимизации (в секундах)							
	Нет	Reorder-Blocks: cache+	Reorder-Blocks: branch-predictor	Reorder-Blocks: ext-tsp	Reorder-Blocks: cluster-shuffle	Split-All-Cold	Align-Blocks	Reorder-Blocks: ext-tsp и Split-All-Cold
Test (низкая)	7,404	7,249	7,195	7,224	7,276	7,615	7,464	7,210
Train (средняя)	63,177	61,251	60,715	61,434	63,907	61,089	61,434	59,728
Refrate (высокая)	349,879	347,968	347,455	345,242	348,479	349,446	345,031	343,212

Для наглядного представления результатов из таблицы были построены графики, на которых произведено сравнение различных вариантов оптимизации для каждой нагрузки. Результаты тестирования на низкой нагрузке представлены на рисунке 38.

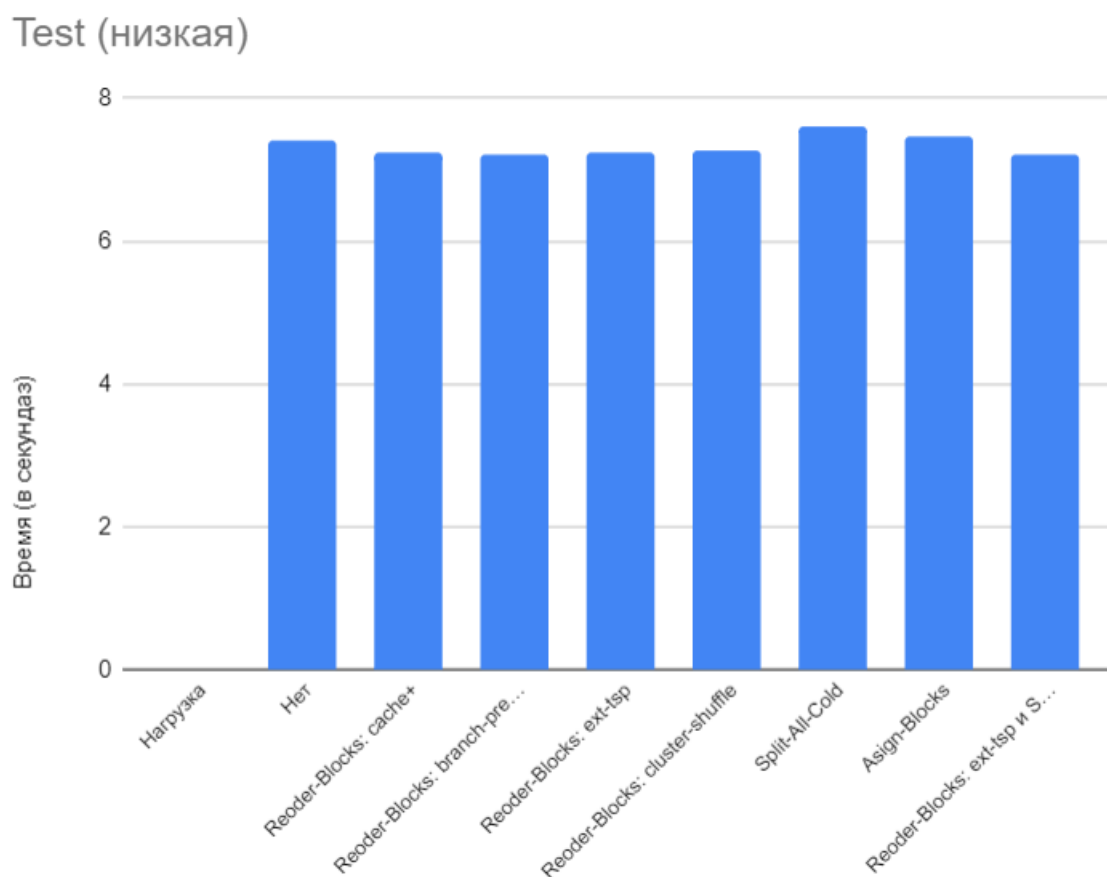


Рисунок 38 – Гистограмма к таблице 1 при нагрузке test

На диаграмме видно, что не все алгоритмы привели к ускорению процесса компиляции. Алгоритмы Split-All-Cold и Align-Blocks показали отрицательный результат, что означает ухудшение производительности. Лучший результат показала опция branch-predictor алгоритма Reorder-Blocks.

Далее необходимо рассчитать ускорение для каждой модификации. Ускорение вычисляется путем деления времени компиляции до оптимизации на время компиляции после оптимизации. Результаты ускорения для каждой модификации представлены ниже:

- Reorder-Blocks: cache+ – 1,021 (2,1%);

- Reorder-Blocks: branch-predictor – 1,029 (2,9%);
- Reorder-Blocks: ext-tsp – 1,025 (2,5%);
- Reorder-Blocks: cluster-shuffle – 1,018 (1,8%);
- Split-All-Cold – 0,972 (ухудшилось на 2,8%);
- Align-Blocks – 0,991 (ухудшилось на 0,9%);
- Reorder-Blocks + Split-All-Cold – 1,027 (2.7%).

Далее на рисунке 39 приведена гистограмма сравнения времени работы алгоритмов при нагрузке train.

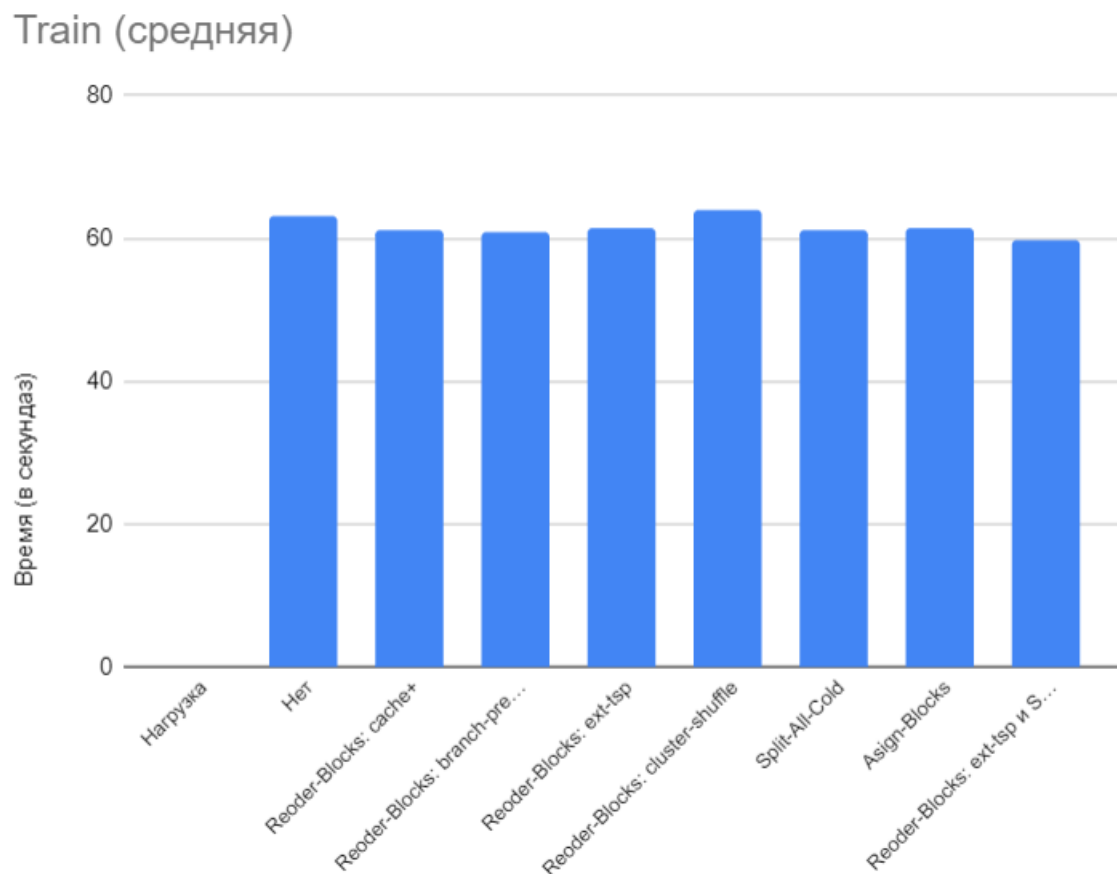


Рисунок 39 – Гистограмма к таблице 1 при нагрузке train

На этот раз практически все алгоритмы показали положительную динамику, отрицательный результат показал только алгоритм Reorder-Blocks с опцией cluster-shuffle. Среди отдельно запущенных алгоритмов лучший результат снова показала опция branch-predictor алгоритма Reorder-Blocks. Однако

среди всех модификаций лучший результат у комбинации Reorder-Blocks + Split-All-Cold.

Рассчитаем ускорение для каждого алгоритма:

- Reorder-Blocks: cache+ – 1,031 (3,1%);
- Reorder-Blocks: branch-predictor – 1,041 (4,1%);
- Reorder-Blocks: ext-tsp – 1,028 (2,8%);
- Reorder-Blocks: cluster-shuffle – 0,989 (ухудшилось на 1,1%);
- Split-All-Cold – 1,034 (3,4%);
- Align-Blocks – 1,028 (2,8%);
- Reorder-Blocks + Split-All-Cold – 1,058 (5.8%).

Теперь проанализируем временные показатели при нагрузке refrate, рисунок 40.

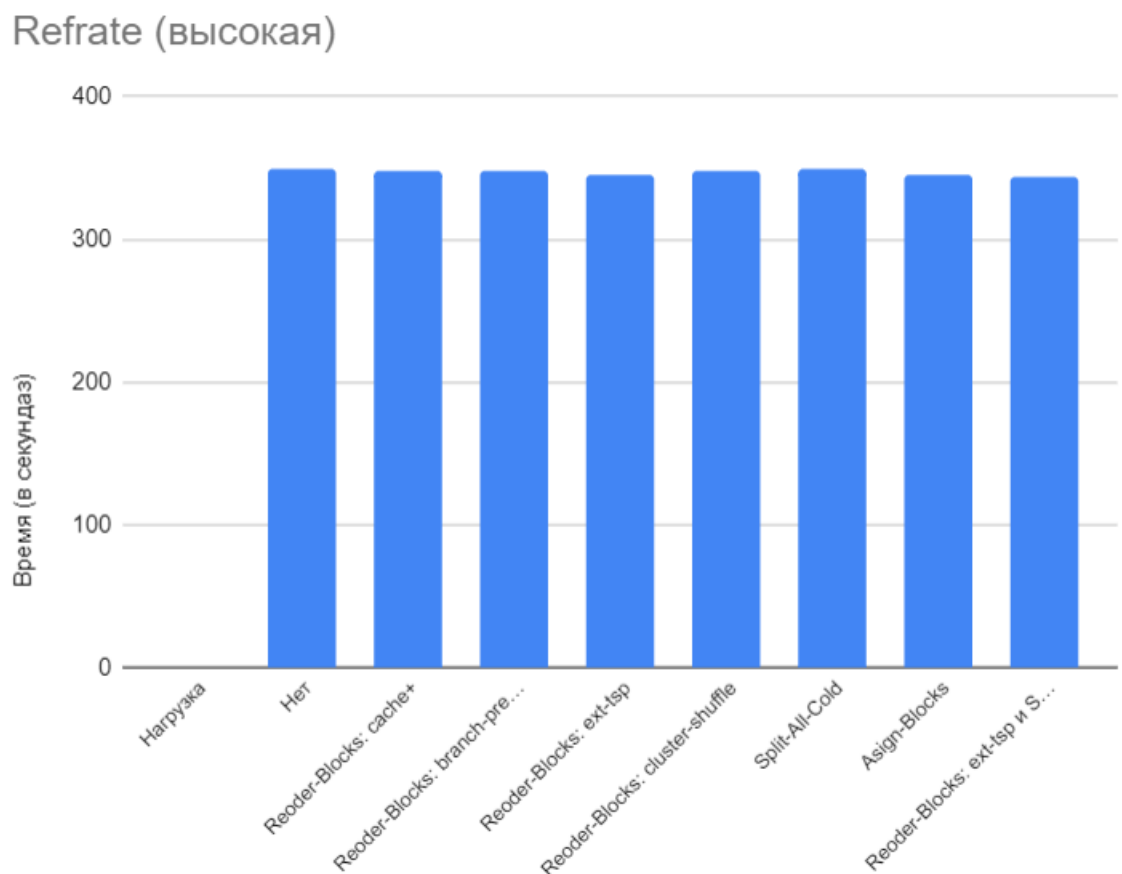


Рисунок 40 – Гистограмма к таблице 1 при нагрузке refrate

На этот раз все алгоритмы показали положительную динамику. Среди отдельно запущенных алгоритмов лучший результат показал алгоритм Align-

Blocks. Но среди всех модификаций лучший результат у комбинации Reorder-Blocks + Split-All-Cold.

Рассчитаем ускорение для каждого алгоритма:

- Reorder-Blocks: cache+ – 1,005 (0,5%);
- Reorder-Blocks: branch-predictor – 1,007 (0,7%);
- Reorder-Blocks: ext-tsp – 1,013 (1,3%);
- Reorder-Blocks: cluster-shuffle – 1,004 (0,4%);
- Split-All-Cold – 1,001 (0,1%);
- Align-Blocks – 1,014 (1,4%);
- Reorder-Blocks + Split-All-Cold – 1,019 (1,9%).

Аккумулируя все полученные данные можно проанализировать каждый отдельный алгоритм, отследить его динамику и понять, как он работает в зависимости от нагрузки.

Итак, опция cache+ алгоритма Reorder-Blocks на всех нагрузках показала положительный результат, лучшее ускорение было продемонстрировано на train (3,1%), худшее на refrete (0,5%).

Опция branch-predictor алгоритма Reorder-Blocks на всех нагрузках показала очень хороший результат. Среди всех отдельно запущенных модификаций она является бесспорным фаворитом, так как показала лучший результат сразу на двух нагрузках. Лучшее ускорение было продемонстрировано на train (4,1%), худшее на refrete (0,7%).

Опция ext-tsp алгоритма Reorder-Blocks также на всех нагрузках показала очень высокий результат, имеет лучший результат среди всех опций алгоритма Reorder-Blocks на нагрузке refrate. Лучшее ускорение было продемонстрировано на train (2,8%), худшее на refrete (1,3%).

Опция cluster-shuffle алгоритма Reorder-Blocks на всех нагрузках показала очень слабый результат, на нагрузке train ухудшило изначальную неоптимизированную программу на 1,1%. Лучшее ускорение было продемонстрировано на test (1,8%).



Алгоритм Split-All-Cold показал не самые выдающиеся результаты, увеличив время работы программы на низкой нагрузке на 2,8%, однако на средней нагрузке он показал один из лучших результатов (3,4%).

Алгоритм Align-Blocks также не показал хороших результатов, но в сравнении с Split-All-Cold оказался более стабильным. Худший результат он показал на низкой нагрузке замедлив работу программы на 0,9%. Лучший результат был достигнут на средней нагрузке (2,8%).

Бесспорным фаворитом является связка алгоритмов Reorder-Blocks с опцией ext-tsp и Split-All-Cold, которая показала наилучший результат на train и refrate, а на test была на втором месте.

## **5 Оценка влияния предложенных технических и организационных изменений проекта, программы, предприятия**

В современном быстро меняющемся мире организации сталкиваются с постоянными вызовами и необходимостью адаптироваться к новым условиям. Именно здесь вступают в игру организационные изменения – процесс изменения структуры, процессов, политик и культуры организации с целью улучшения ее эффективности и способности достигать поставленных целей.

Во второй половине XX века возникла актуальная проблема, связанная с организационными изменениями, которая стала предметом исследований и изучается в области управления. Множество исследований и практических примеров успешной реализации изменений ведущими компаниями демонстрируют их эффективность и значимость.

Организация [1] – это слаженная структурированная группа людей, сотрудничающих с целью достижения определенных задач и целей. Она может представлять собой предприятие, учреждение, организацию гражданского общества, некоммерческую организацию или любую другую форму организованной деятельности. Внедрение изменений в одну часть организации влечет за собой процесс перемен, который охватывает и влияет на другие аспекты деятельности организации.

Организационные изменения [3] проходят через несколько этапов, включая анализ и оценку текущего состояния, планирование изменений, вовлечение сотрудников, внедрение новых подходов и оценку результатов. Целью этих изменений является достижение лучших результатов работы организации, оптимизация системы управления, использование передовых методов и устранение рутинных операций.

Однако, внедрение организационных изменений может столкнуться с различными препятствиями, такими как сопротивление со стороны сотрудни-

ков, недостаточная коммуникация и подготовка. Руководители должны учитывать эти проблемы и разрабатывать стратегии, чтобы успешно преодолеть их.

Организационные изменения в бизнесе могут быть мотивированы различными факторами, включая изменение рыночных условий, появление новых технологий, изменение потребностей клиентов, конкуренцию, стратегические пересмотры и проблемы внутренней эффективности. Целью таких изменений является создание более эффективной и конкурентоспособной организации, способной оперативно адаптироваться к изменениям и использовать новые возможности для достижения успеха.

В данном разделе работы будет проведена оценка готовности организаций к изменениям, включая факторы, такие как персонал (работники), наличие необходимых ресурсов, достаточность финансовых средств для осуществления изменений, наличие подходящего программного обеспечения, актуальность решаемых проблем и уровень риска. Объектом исследования данного раздела является любая ИТ-компания, специализирующаяся на разработке компиляторов, языков и инструментов, оптимизации кода, виртуальных машинах и интерпретаторах, программировании встроенных систем и исследовательской деятельности.

Задача, исследуемая в рамках выпускной квалификационной работы (ВКР) – анализ алгоритмов оптимизации размещения базовых блоков для LLVM [8]. Оптимизация размещения базовых блоков является важным этапом в процессе компиляции, где целью является улучшение производительности и эффективности исполнения программ. Алгоритмы оптимизации размещения базовых блоков [11] направлены на оптимальное расположение блоков в памяти с целью минимизации переходов и улучшения локальности данных. Это позволяет снизить задержки доступа к памяти и улучшить использование кэш-памяти, что в свою очередь приводит к повышению скоро-

сти выполнения программ. Оценка алгоритмов будет проводиться по различным параметрам, включая производительность программы, эффективность использования памяти и степень оптимизации кода.

Использование этих алгоритмов может быть особенно полезным в случаях больших программных проектов с обширным объемом кода, в области высокопроизводительных вычислений, где эффективное использование ресурсов и оптимизация производительности являются критическими, для программ, работающих на встроенных системах и микроконтроллерах с ограниченными ресурсами, и для приложений, работающих в реальном времени, таких как системы управления, автоматическое управление, медицинская техника и другие критические системы.

Проведение оценки целесообразности изменений в организации [2], в соответствии с предложенным проектом, будет осуществляться с учетом заранее выбранных критериев, определенных в методических указаниях. Для оценки будет использоваться пятибалльная шкала, которая позволит определить приоритетность и значимость каждого критерия. Данные приведены в таблице 1.

Таблица 2 – Оценка целесообразности изменений.

<i>1. Готовность организации к изменениям</i>			
<b>1.1. Персонал</b>	Готовность персонала к изменениям, наличие лидера, квалификация персонала	3	Опыт и знания персонала о работе с LLVM могут существенно влиять на их готовность к изменениям. Если в организации уже есть специалисты, обладающие опытом работы с LLVM и пониманием его функциональности, они могут быть более готовыми к анализу алгоритмов оптимизации размещения базовых блоков. Если персонал организации не имеет достаточного

			<p>опыта с LLVM может потребоваться дополнительное обучение и подготовка. Организация может предоставить сотрудникам необходимые ресурсы и обучающие материалы, чтобы они смогли освоить новые концепции [2] и навыки. Важным аспектом готовности персонала является поддержка со стороны руководства и коммуникация внутри организации.</p>
<b>1.2. Материально-технические факторы</b>	Наличие необходимых ресурсов, доступность материалов, уровень износа техники	5	<p>Готовность организации к изменениям [3] связана с наличием соответствующего программного обеспечения. Это включает наличие LLVM и связанных инструментов, которые позволяют проводить анализ и оптимизацию размещения базовых блоков. Организация должна иметь доступ к актуальным версиям программного обеспечения и ресурсы для его установки и поддержки. Готовность организации также зависит от наличия необходимой технической инфраструктуры. Это включает высокопроизводительные компьютеры или серверы, достаточное объем хранилища для обработки и хранения данных, а также средства связи и сетевую инфраструктуру, необходимые для работы с LLVM и проведения анализа.</p>
<b>1.3. Финансы</b>	Достаточность финансовых средств для реализации изменения, потребность во внешнем финансировании	4	<p>Готовность организации к изменениям связана с наличием достаточного бюджета для реализации проекта анализа алгоритмов оптимизации размещения базовых блоков. Это</p>

			<p>включает финансирование приобретения необходимого программного обеспечения, обновления технической инфраструктуры, обучения персонала и других связанных расходов. Организация должна иметь доступ к достаточным финансовым ресурсам, чтобы обеспечить покрытие затрат, связанных с анализом алгоритмов оптимизации размещения базовых блоков. Это включает оплату лицензий на программное обеспечение, оплату услуг специалистов, обеспечение необходимой технической инфраструктуры и обучение персонала.</p> <p>Организация должна провести анализ окупаемости изменений [2]. Это включает оценку потенциальных выгод и улучшений, которые могут быть достигнуты в результате реализации проекта, и сравнение их с затратами. Если ожидаемые выгоды превышают затраты и организация может себе позволить эти затраты, готовность к изменениям будет выше. Готовность организации к изменениям также зависит от ее инвестиционной стратегии и готовности к внедрению новых технологий и подходов. Если организация активно инвестирует в развитие и инновации, она скорее всего будет готова к финансовым затратам.</p>
<b>1.4. Информация</b>	Наличие соответствующего программного обеспечения, разработанной	5	Организация должна иметь доступ к достаточной информации о LLVM и алго-

	системы документооборота, отчетности		<p>ритмах оптимизации размещения базовых блоков. Это включает официальную документацию, исследовательские статьи, руководства, учебные материалы и примеры использования. Чем больше информации имеется, тем лучше организация будет готова к изменениям и принятию соответствующих решений.</p> <p>Готовность организации к изменениям также зависит от наличия каналов обмена информацией и опытом между сотрудниками. Это может включать внутренние коммуникационные системы, форумы, совещания и обучающие программы, которые позволяют обмениваться знаниями и опытом в области анализа алгоритмов оптимизации размещения базовых блоков.</p>
<b>2. Необходимость изменений в организации</b>			
<b>2.1 Внутренние факторы</b>	Актуальность решаемых проблем, соответствие целям и стратегии развития компании	4	<p>Организация должна оценить, насколько актуальна проблема анализа алгоритмов оптимизации размещения базовых блоков для ее текущей деятельности и проектов. Если оптимизация производительности кода и улучшение работы компилятора являются приоритетными задачами для организации, то проблема будет актуальной и готовность к изменениям будет выше.</p> <p>Готовность организации к изменениям также зависит от того, насколько решение проблемы соответствует целям и стратегии</p>

			<p>развития компании. Если это направление соответствует стратегическим целям организации, то она будет более готова внедрить изменения и внести соответствующие ресурсы. Организация должна провести планирование и приоритизацию изменений [3] в рамках своей общей стратегии развития. Это позволит определить, насколько важны эти изменения относительно других проектов и инициатив, а также выделить необходимые ресурсы для их реализации.</p>
<p><b>3. Уровень риска предлагаемого изменения</b></p>	<p>Вероятность возникновения различных угроз – умеренна.</p>	<p>3</p>	<p>Организация должна провести анализ и идентификацию рисков, связанных с внедрением изменений. Это может включать технические риски, связанные с совместимостью существующей инфраструктуры и программного обеспечения, а также организационные риски, связанные с обучением персонала и изменением рабочих процессов.</p> <p>Организация должна оценить вероятность возникновения рисков и их потенциальное воздействие на проект. Некоторые риски могут иметь большую вероятность возникновения и серьезное воздействие на проект, в то время как другие могут быть менее значимыми.</p> <p>Организация может решить внедрять изменения поэтапно или в небольших масштабах, чтобы оценить и снизить риски. Это позволяет организации прово-</p>



			дить тестирование, собирать обратную связь и вносить корректировки на ранних этапах проекта.
--	--	--	----------------------------------------------------------------------------------------------

## Выводы

После анализа ключевых факторов изменений и их эффективности, проводится подведение итогов с использованием оценочной сводки. Для этой сводки используются баллы, которые были присвоены в таблице, и аддитивная модель [1], которая позволяет суммировать эти баллы. Полученная сумма баллов позволяет сравнить и оценить привлекательность данных изменений для компаний. Итоговая сумма баллов составляет следующее значение.

$$3 + 5 + 4 + 5 + 4 + 3 = 24$$

Коэффициенты линейной комбинации в оценочной сводке определены с учетом приоритетности изменений для организации и условий, характерных для рассматриваемого проекта. Результаты вычислений коэффициентов линейной комбинации представлены ниже:

$$\begin{aligned}
 k_{\text{персонал}} &= \frac{3}{25} = 0.12, & k_{\text{мат-тех факторы}} &= \frac{5}{25} = 0.2, \\
 k_{\text{финансы}} &= \frac{4}{25} = 0.16, & k_{\text{информация}} &= \frac{5}{25} = 0.2, \\
 k_{\text{необходимость(внутр.)}} &= \frac{4}{25} = 0.16, & k_{\text{риски}} &= \frac{3}{25} = 0.12. \\
 \sum k &= 0,96
 \end{aligned}$$

Оценка эффективности изменения включает анализ финансовой результативности с учетом возможных доходов и затрат, и может быть выполнена с применением различных методов. Оценка рисков изменения предполагает анализ потенциальных угроз внедрения преобразования и изучение успешных практик реализации аналогичных изменений другими компаниями.

Путем умножения весового коэффициента каждого фактора на соответствующую оценку в баллах и последующего суммирования, общая оценка изменений по всем критериям составляет следующее значение:

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

Анализируя все три результата вместе, компания получает дополнительную информацию о факторах успеха отдельных преобразований с точки зрения управляющего субъекта. В данном случае, предполагаемые изменения оцениваются на пятибалльной шкале с результатом 4, что свидетельствует о значительной эффективности изменений, однако имеются некоторые недостатки. Такая оценка позволяет прогнозировать достижение ожидаемых результатов, но также предоставляет информацию о возможных рисках и компромиссах с другими важными компонентами. При принятии управленческого решения относительно реализации или отказа от реализации изменения следует учитывать суть изменения и его предполагаемые последствия.

## ЗАКЛЮЧЕНИЕ

В ходе данной выпускной квалификационной работы, посвященного анализу алгоритмов размещения базовых блоков в LLVM, мы провели обширный обзор и детальный анализ различных алгоритмов, применяемых в рамках LLVM. В ходе исследования были рассмотрены следующие алгоритмы: Reorder-Blocks с его четырьмя опциями, Split-All-Cold и Align-Blocks. Каждый из этих алгоритмов обладает своими уникальными преимуществами и ограничениями, и их выбор и применение могут зависеть от конкретных требований и характеристик программы.

В рамках исследования было уделено внимание изучению параметров и флагов командной строки, связанных с размещением базовых блоков в LLVM. Эти параметры предоставляют возможность настраивать поведение размещения базовых блоков в компиляторе и оказывать значительное влияние на оптимизацию производительности программы.

Полученные результаты экспериментов и проведенный анализ производительности явно демонстрируют, что выбор оптимального алгоритма размещения базовых блоков и соответствующих параметров может оказать значительное влияние на производительность программы. Это может проявиться в ускорении выполнения программы, сокращении объема генерируемого кода и оптимизации использования памяти.

Полученные результаты также подтверждают, что эффективное размещение базовых блоков является существенной оптимизацией для повышения производительности программы.

В будущих исследованиях в рамках LLVM стоит обратить внимание на изучение комбинированных стратегий размещения, оптимизацию параметров и улучшение процесса принятия решений о размещении базовых блоков на основе статистических данных и профилирования выполнения программы.

В целом, данная дипломная работа по анализу алгоритмов оптимизации размещения базовых блоков в LLVM предоставляет ценные результаты и рекомендации для оптимизации производительности программ, особенно при использовании компилятора LLVM. Данная работа имеет развитие – алгоритмы оптимизации размещения базовых блоков могут быть полезны IT-компаниям, занимающимся разработкой компиляторов, оптимизацией кода, виртуальными машинами и интерпретаторами.

## СПИСОК ЛИТЕРАТУРЫ

1. Ваганова В. А. Методические указания по выполнению дополнительного раздела «Оценка влияния предложенных технических и организационных изменений проекта, программы, предприятия» //для студентов бакалавриата и специалитета всех технических факультетов и ИНПРО-ТЕХ СПбГЭТУ «ЛЭТИ». – 2023. – С. 20.
2. Камерон, Э., & Грин, М. Практическое руководство по управлению изменениями: Модели, инструменты, техники. – 2017.
3. Хэйс, Дж. Теория и практика управления изменениями. Альпина Паблишер -2019.
4. Официальный сайт Ubuntu URL: <https://ubuntu.com/download>
5. ТестыBenchmarkGitHubURL: <https://github.com/chfeng-cs/CPU2017.git>
6. Официальная документация VirtualBox URL: <https://docs.oracle.com/en/virtualization/virtualbox>
7. Официальная документация WMvare URL: <https://www.vmware.com>
8. Документация LLVM URL: <https://llvm.org/docs/Документация>
9. Clang URL: <https://clang.llvm.org>
10. Документация deepsjeng:  
[https://www.spec.org/cpu2017/Docs/benchmarks/531.deepsjeng\\_r.html](https://www.spec.org/cpu2017/Docs/benchmarks/531.deepsjeng_r.html)
11. Adve, V.S., Lattner, C: The LLVM Compiler Infrastructure in Practice / Adve, V.S., Lattner, C – 2012
12. Sarda, S. LLVM Essentials / Suyog Sarda. – 2015
13. Хабр. Что такое LLVM и зачем он нужен? / Андрей Боханко. URL: <https://habr.com/ru/company/huawei/blog/511854/>
14. Репозиторий LLVM Bolt URL: <https://github.com/facebookincubator/BOLT>
15. Andrew Trick, Alexey Kukanov, Sergey Dmitriev и другие. BOLT: Binary Optimization and Layout Tool. URL: <https://github.com/facebookincubator/BOLT>

## ПРИЛОЖЕНИЕ А

### Порядок выполнения команд для оптимизации

1. Клонирование репозитория:

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

2. Переходим в каталог с бенчмарком:

```
cd CPU2017/benchspec/CPU/531.deepsjeng/src
```

3. Собираем бенчмарк без оптимизаций BOLT, но с использованием опции -O2, чтобы получить бинарный файл deepsjeng:

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

4. Создаем инструментированную версию бинарного файла:

```
llvm-bolt-16 -instrument deepsjeng -o deepsjeng -instr
```

5. Выполняем инструментированную версию, чтобы получить профиль, по которому в дальнейшем будет проводиться оптимизация:

```
./ deepsjeng-instr ../data/test/input/test.txt
```

6. Проводим оптимизацию с использованием BOLT:

```
llvm-bolt-16 deepsjeng -o deepsjeng-opt -data=/tmp/prof.fdata -reorder-blocks=cache+ -split-all-cold
```

7. Замеряем время выполнения неоптимизированной программы при помощи утилиты time:

```
time ./ deepsjeng../data/test/input/test.txt
```

8. Замеряем время выполнения оптимизированной программы при помощи утилиты time:

```
time ./ deepsjeng-opt../data/test/input/test.txt
```