

**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: “Оптимизация распределения машинных инструкций: Анализ
алгоритмов встраивания функций на LLVM”**

Студент

М. С. Мамонтов

Руководитель

к. т. н., доцент

А. А. Пазников

Консультант 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 М. С. Мамонтов

Группа № 9305

- 1.Тема:** Оптимизация распределения машинных инструкций: Анализ алгоритмов встраивания функций на LLVM
(утверждена приказом № _____ от _____)
- 2.Объект и предмет исследования:** процесс встраивания функций в коде программы, осуществляемый компилятором LLVM.
- 3.Цель:** Оптимизация процесса встраивания функций 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 встраивания функций для LLVM, подготовка рабочей среды к тестам и анализу, применение алгоритмов оптимизации на тестовой программе, оценка организационных изменений на основе ИТ – компаний
- 6.Дополнительные разделы:** Оценка влияния предложенных технических и организационных изменений проекта, программы, предприятия
- 7.Результаты:** текст ВКР, иллюстративный материал

Дата выдачи задания
«___» _____ 2023 г.

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

Руководитель
к. т. н., доцент

_____ А. А. Пазников

Студент

_____ М. С. Мамонтов

РЕФЕРАТ

Пояснительная записка содержит: 50 стр., 38 рис., 2 табл., 12 ист., 1 прил.

Цель работы: Анализ алгоритмов встраивания функций для LLVM.

Встраивание функций является одной из ключевых оптимизаций, направленных на улучшение производительности программ. Современные приложения все чаще требуют высокой производительности, и эффективное встраивание функций может существенно повлиять на скорость выполнения и использование ресурсов.

В данной работе будет проведено исследование методик и практик, которые используются для встраивания функций, для этого в работе будут анализироваться бинарные файлы и оптимизироваться для наилучшего встраивания функций внутри них.

В ходе работы мы проанализировали скорость работы тестового кода без оптимизации и с помощью алгоритмов найдены лучшие варианты встраивания функций.

ABSTRACT

Purpose of the work: Analysis of algorithms for embedding functions for LLVM.

Embedding functions is one of the key optimizations aimed at improving program performance. Modern applications increasingly require high performance, and effective embedding of functions can significantly affect the speed of execution and resource usage.

In this paper, a study of the techniques and practices that are used for embedding functions will be conducted, for this purpose, binary files will be analyzed and optimized for the best embedding of functions inside them.

In the course of our work, we analyzed the speed of the test code without optimization and with the help of algorithms, the best options for embedding instructions were found

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ	7
ВВЕДЕНИЕ	9
1 Обзор алгоритмов встраивания функций для LLVM.....	11
1.1 Always Inline.....	11
1.2 Inline Only Locally	12
1.3 Inline Small Functions	14
1.4 Inline Hot Functions	15
1.5 Partial Inlining	16
1.6 Aggressive Inlining.....	17
2 Подготовка рабочей среды.....	19
3 Оптимизация машинного кода	21
3.1 Оптимизация Always Inline.....	22
3.2 Оптимизация Inline Only Locally.....	24
3.3 Оптимизация Inline Small Functions.....	26
3.4 Оптимизация Inline Hot Functions	29
3.5 Оптимизация Partial Inlining	31
3.6 Оптимизация Aggressive Inlining	34
4 Сравнение результатов модификации	36
5 Оценка влияния предложенных технических и организационных изменений проекта, программы, предприятия	40
ЗАКЛЮЧЕНИЕ.....	48
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	50
ПРИЛОЖЕНИЕ А.....	51

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

Low Level Vitrual Machine (LLVM) — это набор инструментов для разработки компиляторов, оптимизации кода и анализа производительности.

Binary Optimization and Layout Tool (BOLT) — это инструмент для оптимизации исполняемых файлов, который использует LLVM в качестве своего основного фреймворка. Он работает с бинарными файлами и применяет различные техники оптимизации, такие как перестановка блоков кода для улучшения локальности инструкций и производительности кэша, удаление недостижимого кода, уплотнение кода и т.д.

Post-linkage optimization (PLO) — это процесс оптимизации кода, который выполняется после слинковки (linking) объектных файлов или исполняемых файлов. PLO позволяет проводить оптимизации на уровне программы, а не отдельных ее модулей. Во время сборки программы ее код разбивается на отдельные модули, которые компилируются и линкуются вместе в единую исполняемую программу. Однако оптимизации, проводимые на уровне каждого модуля, не учитывают контекста, в котором эти модули будут использоваться в конечном приложении.

Code-layout optimization (CLO) — это процесс оптимизации порядка размещения инструкций в исполняемом коде. Эта оптимизация может быть полезной для улучшения производительности программы, так как хороший порядок инструкций может уменьшить время доступа к памяти и увеличить кэш-промахи. Кроме того, правильное размещение инструкций может помочь уменьшить размер исполняемого файла, что может быть полезным, например, при использовании устройств с ограниченным объемом памяти. Некоторые примеры кодовых оптимизаций, связанных с размещением инструкций, включают в себя компактное кодирование, пакетное размещение и упаковку инструкций. LLVM включает в себя ряд оптимизаций кодовой разметки, таких как basic-block-sections и reorder-blocks, которые могут быть

использованы для улучшения производительности и размера исполняемого файла.

C-Language Family Fronted (Clang) — это компилятор языка C, C++ разработанный как часть LLVM. Clang предоставляет широкий набор инструментов и опций для компиляции, оптимизации и анализа кода.

Бенчмарк (benchmark) — программы или наборы тестовых данных, для последующей модификации, созданные для тестирования и проверки.

Standard Performance Evaluation Corporation (SPEC) — некоммерческая организация, главной целью которой является разработка и публикация наборов тестов, предназначенных для измерения производительности компьютеров. Тестовые пакеты SPEC являются стандартами для оценки производительности современных компьютерных систем.

ВВЕДЕНИЕ

Оптимизация программного кода является важным аспектом разработки, направленным на улучшение производительности и эффективности программ. Одной из эффективных техник оптимизации является встраивание функций, которое заменяет вызовы функций на их фактический код в месте использования.

Анализ алгоритмов встраивания функций имеет свои ограничения. Он может быть менее точным в случае сложного программного кода, особенно в больших проектах. Несмотря на эти ограничения, анализ алгоритмов встраивания функций остается важным инструментом для оптимизации кода. Например, исследования показали, что встраивание малых функций может улучшить производительность и снизить накладные расходы вызова функций. Однако, для каждого проекта следует тщательно выбирать подходящий алгоритм встраивания и учитывать особенности кода и требования проекта.

Цель данной дипломной работы заключается в анализе алгоритмов встраивания функций на LLVM [7]. Анализ алгоритмов встраивания функций в LLVM позволяет изучить их принципы работы, а также оценить их эффективность и применимость в различных сценариях.

В первом разделе проводится подробный обзор каждого алгоритма встраивания и их атрибутов.

Во втором разделе проводится подготовка к анализу, то есть установка необходимого программного обеспечения и настройки.

В третьем разделе проводится тестирование всех алгоритмов встраивания на тестовой программе.

В четвертом разделе проводится сравнительный анализ результатов модификации тестовых данных каждым алгоритмом.

В рамках исследования будут проанализированы различные алгоритмы встраивания функций, такие как Always Inline, Inline Only Locally, Inline Small

Functions, Inline Hot Functions, Partial Inlining и Aggressive Inlining [10].

Будут рассмотрены преимущества и ограничения каждого алгоритма, а также проведено сравнение их влияния на производительность программного кода.

Результаты исследования будут полезны разработчикам и оптимизаторам программного кода, которые стремятся улучшить производительность своих приложений и эффективно использовать возможности встраивания функций.

1 Обзор алгоритмов встраивания функций для LLVM

Существует несколько различных алгоритмов встраивания функций для LLVM [7]:

1. Always Inline.
2. Inline Only Locally.
3. Inline Small Functions.
4. Inline Hot Functions.
5. Partial Inlining.
6. Aggressive Inlining

В данной работе будет подробно исследованы алгоритмы встраивания функций.

Профили программы, при котором применение встраивания функций особенно полезно: Программы с большим количеством небольших функций, программы с локальными функциями, программы с высокой частотой вызовов функций и программы, требующие низкой задержки вызовов функций.

1.1 Always Inline

Always Inline – алгоритм, который всегда встраивает функцию, независимо от её размера или контекста [10].

Так, этот алгоритм можно применить, для того чтобы точно указать компилятору о встраивании определённой функции. Это может быть полезно, когда функция невелика, или имеет огромную значимость в работе программы. Встраивание функции решает проблему расходов на вызов функции. Из этих расходов могут быть такие, как: сохранение регистров, установка и восстановление контекста, передача аргументов. Если функция помечена для Always Inline, то компилятор будет стараться встроить её в место вызова во время компиляции. Это, вероятнее всего, увеличит размер кода,

поскольку функция оказывается встроена в место вызова, но при правильном выборе функции это может сильно увеличить производительность программы.

Существуют различные инструменты, для применения алгоритма “Always Inline”. В нашем случае удобнее всего использовать компилятор “Clang” [8], поскольку он имеет широкий спектр опций по компилированию и анализу.

Для “Clang” можно использовать атрибут “__attribute__((always_inline))” перед объявлением функции. На рисунке 1 представлен пример выставления атрибута.

```
__attribute__((always_inline)) void myFunction() {  
    // Тело функции  
}
```

Рисунок 1 – Пример выставления атрибута `always_inline`

У данного способа есть минус. Это то, что человеку, оптимизирующему код, необходимо самостоятельно добавлять данный атрибут к каждой функции, необходимой для встраивания. Вместо этого можно воспользоваться командной строкой при компиляции через Clang и вписать в неё команду “-mllvm -insert-all0” [11]. Это позволит применить алгоритм Always Inline ко всем функциям в программе. В данном случае тоже есть минус, поскольку алгоритм “Always Inline” в данном случае применится ко всем функциям, это может привести к увеличению размера кода и вероятному ухудшению производительности в случае, если алгоритм применён не к тем функциям.

1.2 Inline Only Locally

Inline Only Locally – алгоритм оптимизации, который позволяет встраивать функции только в пределах одного модуля или файла [10].

Это означает, что функции могут быть встроены в место вызова только внутри того же файла, но не в другом файле. Идея алгоритма заключается в

том, чтобы ограничить встраивание функции только в пределах одного файла, чтобы избежать увеличение кода и увеличению сложности связи между модулями. Данный алгоритм позволяет сохранять локальность оптимизаций и уменьшить влияние на весь проект.

Алгоритм встраивания “Inline Only Locally” полезен в тех случаях, когда необходимо улучшить производительность внутри отдельного файла, но при этом не повлиять на весь остальной проект и сохранить модульность.

Данный алгоритм доступен на различных уровнях оптимизации и может быть связан с определёнными атрибутами функций. Одним из таких атрибутов является “inline”. На рисунке 2 представлен пример выставления атрибута.

```
inline void myFunction() {  
    // Функция, которую можно встраивать  
}
```

Рисунок 2 – Пример выставления атрибута inline

Данный атрибут указывает, что функцию следует встраивать при компиляции. Применение этого атрибута к функции позволяет явно указать, что эту функцию можно применять при оптимизации “Inline Only Locally”.

Следующим атрибутом является “noinline”. На рисунке 3 представлен пример выставления атрибута.

```
__attribute__((noinline))  
void myFunction() {  
    // Функция, которую не следует встраивать  
}
```

Рисунок 3 – Пример выставления атрибута noinline

Данный атрибут указывает, что функцию не следует встраивать при компиляции. Применение этого атрибута к функции позволяет явно указать, что эту функцию не следует применять при оптимизации “Inline Only Locally”.

Особенность алгоритма встраивания “Inline Only Locally” заключается в том, что ему не требуются явные использования атрибутов. Поскольку он основан на анализе и оптимизации функции внутри одного модуля, то алгоритм встраивает функции в пределах этого модуля, без необходимости указывать атрибуты. Алгоритм будет автоматически применяться ко всем функциям внутри модуля, если уровень оптимизации позволяет это сделать.

Уровень оптимизации “-o0” не включает в себя данный алгоритм по умолчанию. Этот алгоритм применяется на более высоких уровнях, начиная с “-o1”. Для использования алгоритма встраивания “Inline Only Locally” необходимо использовать команду “-mllvm -inline-cost-full” [11].

У данного алгоритма существуют недостатки, которые следует учитывать, при его использовании. Одним из таких недостатков может быть увеличение сложности отладки. Так, поскольку функция встраивается локально и вызовы функции могут быть распределены по разным местам в коде, может быть сложнее отслеживать выполнение программы во время отладки. Ещё одним минусом можно считать ограничение применимости данного алгоритма. Очень часто алгоритм не является оптимальным, и, в некоторых случаях, более продвинутые алгоритмы встраивания функций могут привести к более эффективным результатам.

1.3 Inline Small Functions

Inline Small Functions – это алгоритм, при котором небольшие функции встраиваются в вызывающий код вместо создания отдельного вызова функции [10].

Это позволяет уменьшить расходы на вызовы функций, таких как сохранение и восстановление регистров, передача параметров и т.д.

Целью встраивания малых функций является уменьшение расходов на вызов функции. Это полезно для слишком маленьких и коротких функций, которые выполняет простые операции или просто возвращают результат.

Встраивание малых функций применяется к функциям, которые занимают небольшой объём кода. Например, к функциям, которые содержат несколько инструкций или несколько строк кода. Нет точного размера, по которому можно оценить функцию как “малую”.

Для указания компилятору о встраивании малых функций могут использоваться атрибуты, так, можно использовать атрибут “inline”, как показано на рисунке 2. Однако компиляторы могут встраивать функции автоматически, принимая решение на основе эвристик и различных настроек оптимизации.

Чтобы использовать оптимизацию встраивания функции “Inline Small Functions” необходимо использовать команду “-clang -inlinehint-threshold=n”, где n – это размер функции. Так, если прописать “-clang -inlinehint-threshold=1”, то это укажет Clang на использование оптимизации встраивания функций, размер которых не превышает значение 1.

1.4 Inline Hot Functions

Inline Hot Functions – алгоритм встраивания горячих функций [10].

Основной идеей данного алгоритма является встраивание функций, которые считаются “горячими” – т.е. те функции, которые вызываются очень часто. Встраивание таких функций сокращает накладные расходы на вызов функций и улучшает производительность программы.

Данный алгоритм определяет горячие функции на основе профилирования выполнения программы. Критерии определения “горячести” функции могут варьироваться и отличаться, в зависимости от методики профилирования.

Для атрибутирования функций, которые оптимизатор хочет пометить как горячие, используется атрибут “__attribute__((hot))”. Данный атрибут указывает компилятору, что функция является горячей и должна быть встроена. На рисунке 4 представлен пример выставления атрибута.

```
__attribute__((hot))  
void hotFunction() {  
    // Код функции  
}
```

Рисунок 4 – Пример выставления атрибута hot

Алгоритм “Inline Hot Functions” не применяется при уровне оптимизации “-o0”. Для использования этого алгоритма требуется сбор данных о выполнении программы и анализ путей. Это выполняется на уровнях оптимизации, начиная с “-o1”.

Для использования алгоритма “Inline Hot Functions” можно использовать несколько команд. “-mllvm -inlinehint-threshold=n”, где n – это число, которое указывает сколько раз функция должна вызываться, чтобы быть встроенной [12]. Команда “-mllvm -inline-all-functions” включает автоматическое встраивание горячих функций, который позволяет оптимизатору LLVM самому решать, какие функции встраивать [11].

Так же, как и у предыдущих функций, есть недостатки в виде увеличения размера кода и кэш-промахи. Это присуще всем предыдущим и последующим алгоритмам, поэтому перед применением алгоритмов необходимо проводить тщательное профилирование программы, для оценки разумности применения оптимизации.

1.5 Partial Inlining

Partial Inlining – алгоритм встраивания функций, который позволяет компилятору встраивать только часть функции в код, а не целиком [10].

Вместо того, чтобы встраивать функцию полностью, алгоритм выбирает часть функции, который может улучшить оптимизацию, и встраивает её компилирующийся код.

Данный алгоритм полезен в случаях, когда полное встраивание функции приведёт к снижению производительности из-за большого количества повторяющихся команд. Чтобы предотвратить увеличение количества кода и уменьшение производительности, алгоритм выбирает исключительно полезные части функции для встраивания, что увеличивает эффективность и улучшает производительность.

При использовании алгоритма “Partial Inlining” компилятор анализирует вызовы функций и определяет, какие части этой функции могут быть встроены в код. Чаще всего выбор компилятора падает на те части функции, которые имеют малый размер, часто вызываются или имеют простые зависимости.

“Partial Inlining” не может быть применён при уровне оптимизации “-o0”, поскольку данный уровень оптимизации предназначен для отключения большинства оптимизаций, в том числе и алгоритма “Partial Inlining”.

Для реализации данного алгоритма нет специального атрибута. Если атрибуты используются для того, чтобы указать свойства и характер функции, то “Partial Inlining” чаще всего применяется автоматически на основе анализа и оптимизированных настроек.

Для использования “Partial Inlining” необходимо использовать команду “-mllvm -max-partial-inlining=n” [11]. Данная команда активирует алгоритм частичного встраивания в процессе компиляции.

1.6 Aggressive Inlining

Aggressive Inlining – алгоритм “агрессивного” встраивания функций [10].

Это стратегия встраивания функции, которая применяет полное встраивание всей функции, независимо от её размера или частоты вызова. Цель алгоритма агрессивного встраивания – максимальное снижение расходов на вызов функции и улучшение производительности.

Атрибутом, применимым к функции, для объявления её встраивания, может быть атрибут “__attribute__((always_inline))”. На рисунке 5 представлен пример выставления атрибута.

```
__attribute__((always_inline)) void foo() {  
    // Тело функции  
}
```

Рисунок 5 – Пример выставления атрибута `always_inline`

Данный атрибут имеет более высокий приоритет, по сравнению с остальными атрибутами, и переопределяет другие стратегии встраивания функций.

В Clang для применения агрессивного встраивания необходимо воспользоваться опцией “-mllvm —inline-threshold=n”, где n – количество инструкций [12]. Так, если n принимает значение 100, это означает, что функции более чем 100 инструкциями, не будут встроены, если так решит компилятор. Данное значение может быть любым целым числом.

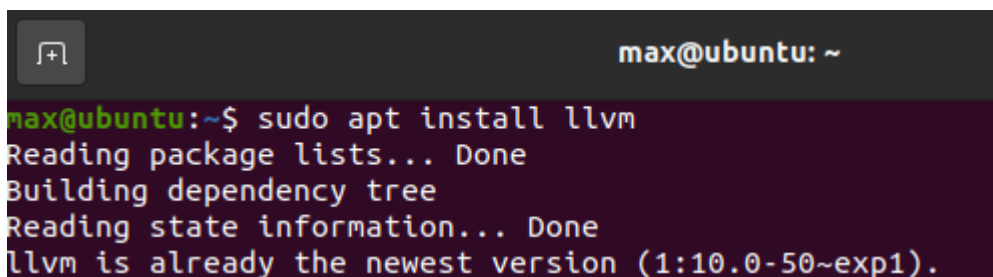
2 Подготовка рабочей среды

В качестве операционной системы для анализа алгоритмов встраивания функции была выбрана Ubuntu [3]. Ubuntu – это операционная система на основе Linux. Она представляет бесплатное и открытое программное обеспечение для персональных компьютеров. Является одной из самых популярных дистрибутивов Linux, поскольку является простой в использовании и активно поддерживаемой. Для нашей задачи главный её плюс – это то, что Ubuntu поддерживает установку дополнительных программ через командную строку.

В анализе мы будем использовать Ubuntu версии 20.04. Это не последняя версия, но она гораздо стабильнее.

Для того, чтобы установить Ubuntu нам понадобится любая виртуальная машина, например VirtualBox [5] или VMware [6]. Необходимо загрузить образ жёсткого диска с официального сайта [3], после чего в выбранной виртуальной машине создать виртуальную операционную систему Ubuntu.

После создания виртуального пространства необходимо установить llvm и clang. Чтобы установить llvm нужно в командную строку, которая вызывается в боковом меню нажатием кнопки Терминал (Terminal), вписать команду “sudo apt install llvm”. Результат на рисунке 6.

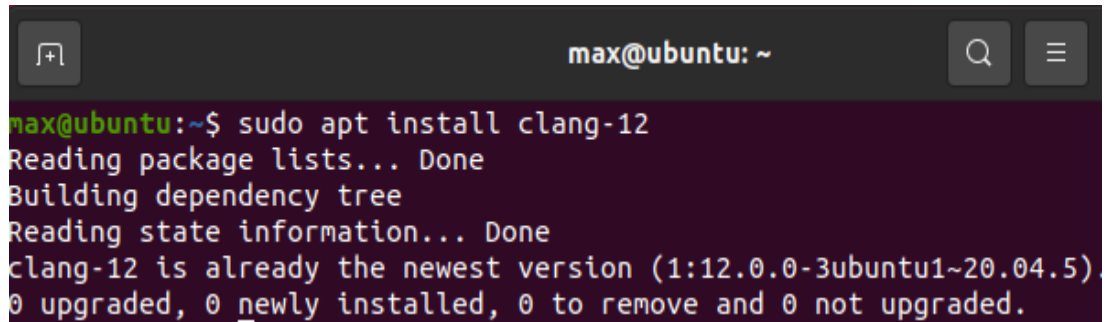
A screenshot of a terminal window with a dark background. The prompt is 'max@ubuntu: ~'. The command 'sudo apt install llvm' has been entered. The output shows 'Reading package lists... Done', 'Building dependency tree', 'Reading state information... Done', and finally 'llvm is already the newest version (1:10.0-50~exp1)'.

```
max@ubuntu: ~  
max@ubuntu:~$ sudo apt install llvm  
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
llvm is already the newest version (1:10.0-50~exp1).
```

Рисунок 6 – Установка llvm

После установки llvm необходимо установить clang. В данной работе используется clang версии 12, поскольку в ней присутствуют все

интересующие нас команды. Чтобы её установить необходимо написать в командной строке команду “sudo apt install clang-12”. Результат на рисунке 7.

A terminal window titled 'max@ubuntu: ~' with a search icon and a menu icon in the top right. The terminal shows the command 'max@ubuntu:~\$ sudo apt install clang-12' and its output: 'Reading package lists... Done', 'Building dependency tree', 'Reading state information... Done', 'clang-12 is already the newest version (1:12.0.0-3ubuntu1~20.04.5).', and '0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.'

```
max@ubuntu:~$ sudo apt install clang-12
Reading package lists... Done
Building dependency tree
Reading state information... Done
clang-12 is already the newest version (1:12.0.0-3ubuntu1~20.04.5).
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
```

Рисунок 7 – Установка Clang-12

Всё, что нам нужно для модернизации и работы с файлами у нас есть, теперь необходимо загрузить сами файлы. В качестве тестовой программы была выбрана mcf benchmark[9]. Для того, чтобы с ней работать необходимо загрузить её с сайта GitHub, где она лежит для открытого использования.

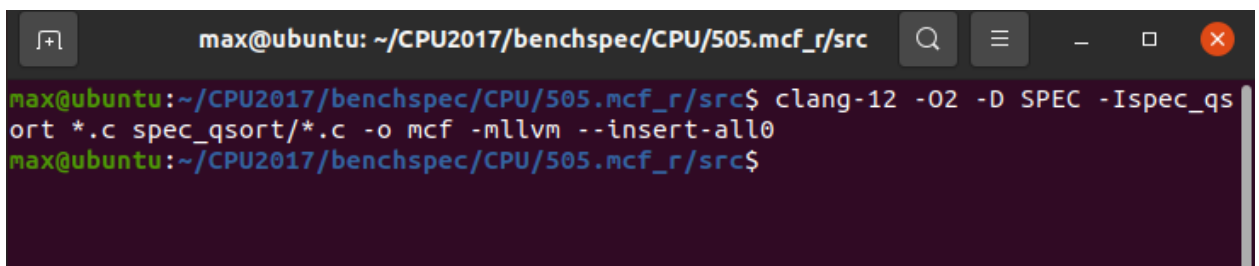
Чтобы её загрузить необходимо прописать в командную строку команду “git clone <URL>”. Ссылка в данном случае взята с [4]. После скачки всех файлов необходимо пройти путь к src файлам бенчмарка 505.mrf_f и открыть в ней командную строку, нажав левой кнопкой мыши по пустому пространству и нажав кнопку “Открыть терминал”.

3 Оптимизация машинного кода

После настройки окружения можно приступать к оптимизации встраивания функций.

В качестве тестовой программы был выбран `mcf benchmark`. `Mcf Benchmark` – это бенчмарк, который моделирует транспортную систему Берлина. Программа написана на языке C и в её бенчмарковской версии используется целочисленная арифметика. Этот тест хорошо подходит для оптимизации, поскольку имеет различные нагрузки. Всего их 3: `test`, `train`, `refrate`. Соответственно лёгкая, средняя, и большая нагрузки. Нагрузка увеличивается с увеличением количеством функций и аргументов, из-за того, что требуется больше регистров.

Для начала необходимо скомпилировать файл с помощью Clang. Для этого используется команда “`clang-12 -O2 -D SPEC -Ispec_qsort *.c spec_qsort/*.c -o mcf -mllvm ...`”, где `-O2` – опция оптимизации, позволяющая использовать команды алгоритмов встраивания функций, `-D Spec` – определение предпроцессора. `-Ispec_qsort *.c spec_qsort/*.c` – компилирование всех файлов с расширением `.c`, `-mllvm ...` - передача оптимизаций компилятору. Данная команда создаёт файл под названием `mcf`, который будет оптимизирован по заданным нами в будущем параметрами. На рисунке 8 представлен пример компиляции программы.

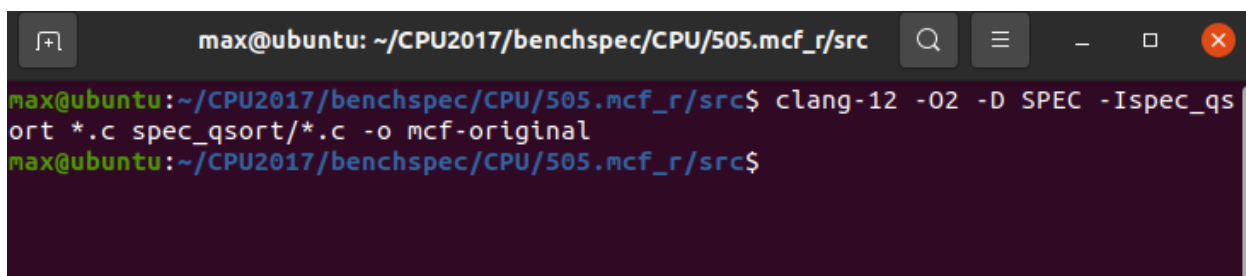
A screenshot of a terminal window with a dark background. The title bar shows the user 'max@ubuntu' and the current directory '~/CPU2017/benchspec/CPU/505.mcf_r/src'. The terminal contains two lines of text: a command to compile a program using clang-12 with various optimization flags, and the prompt for the next command.

```
max@ubuntu:~/CPU2017/benchspec/CPU/505.mcf_r/src$ clang-12 -O2 -D SPEC -Ispec_qsort *.c spec_qsort/*.c -o mcf -mllvm --insert-all0
max@ubuntu:~/CPU2017/benchspec/CPU/505.mcf_r/src$
```

Рисунок 8 – Компиляция программы для оптимизации

После этого необходимо создать файл, который специально не модифицирован, что сравнивать с ним применённые оптимизации. Команда “`clang-12 -O2 -D SPEC -Ispec_qsort *.c spec_qsort/*.c -o mcf-original`” создаст

файл `mcf-original`, который будет служить для сравнения с оптимизированными версиями. На рисунке 9 представлен пример компиляции программы для сравнения.

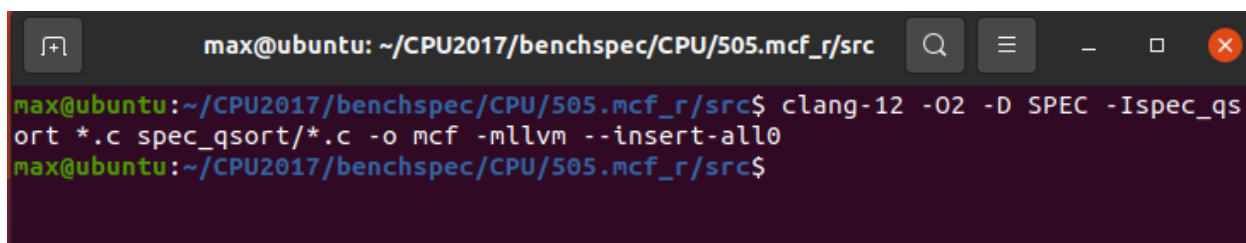


```
max@ubuntu: ~/CPU2017/benchspec/CPU/505.mcf_r/src
max@ubuntu:~/CPU2017/benchspec/CPU/505.mcf_r/src$ clang-12 -O2 -D SPEC -Ispec_qsort *.c spec_qsort/*.c -o mcf-original
max@ubuntu:~/CPU2017/benchspec/CPU/505.mcf_r/src$
```

Рисунок 9 – Компиляция программы для сравнения

3.1 Оптимизация Always Inline

Для оптимизации Always Inline необходимо применить оптимизацию “`—insert-all0`”. Поэтому прописываем в консоль команду “`clang-12 -O2 -D SPEC -Ispec_qsort *.c spec_qsort/*.c -o mcf -mllvm —insert-all0`”. На рисунке 10 представлен пример применения алгоритма постоянного встраивания функций.



```
max@ubuntu: ~/CPU2017/benchspec/CPU/505.mcf_r/src
max@ubuntu:~/CPU2017/benchspec/CPU/505.mcf_r/src$ clang-12 -O2 -D SPEC -Ispec_qsort *.c spec_qsort/*.c -o mcf -mllvm --insert-all0
max@ubuntu:~/CPU2017/benchspec/CPU/505.mcf_r/src$
```

Рисунок 10 – Применение алгоритма встраивания Always Inline

После этого необходимо вывести время работы не оптимизированной и оптимизированной программы. Для начала проверим на низкой нагрузке `test`. Прописывается команда “`time ./mcf ../data/test/input/inp.in > /dev/null`” и “`time ./mcf-original ../data/test/input/inp.in > /dev/null`”. Получаем время работы компиляции файла до и после модификации. На рисунке 11 показано время работы до и после применения алгоритма оптимизации постоянного встраивания на низкой нагрузке.

```

real    0m7.736s
user    0m7.635s
sys     0m0.088s
max@ubuntu:~/CPU2017
inp.in > /dev/null

real    0m7.677s
user    0m7.574s
sys     0m0.084s

```

Рисунок 11 – Время работы до и после оптимизации Always Inline на test

Как видно, программа стала работать на 59 миллисекунд быстрее. Это означает, что на слабой нагрузке программа стала производителем на 0,76%. Теперь проверим улучшение оптимизации при средней нагрузке train. Для этого прописывается команда “time ./mcf-original ../data/train/input/inp.in > /dev/null” и “time ./mcf- ../data/train/input/inp.in > /dev/null”, которая выведет время работы до и после модификации кода. На рисунке 12 представлено время работы до и после оптимизации постоянного встраивания на средней нагрузке.

```

real    0m45.663s
user    0m45.286s
sys     0m0.328s
max@ubuntu:~/CPU2017/benc
/inp.in > /dev/null

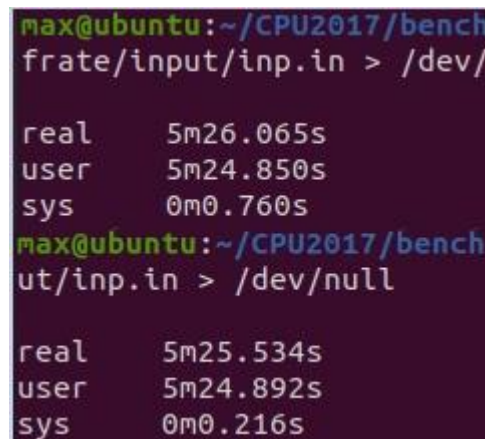
real    0m45.407s
user    0m45.114s
sys     0m0.204s

```

Рисунок 12 – Время работы до и после оптимизации Always Inline на train

Здесь оптимизированный код выполнился на 256 миллисекунд, что на 0,56% быстрее. Осталось провести тесты на большой нагрузке, то есть на нагрузке refrate. Для этого прописывается команда “time ./mcf-original ../data/refrate/input/inp.in > /dev/null” и “time ./mcf- ../data/refrate/input/inp.in > /dev/null”, которая выведет время работы до и после модификации кода. На

рисунке 13 представлено время работы до и после оптимизации постоянного встраивания на высокой нагрузке.



```
max@ubuntu:~/CPU2017/bench
frate/input/inp.in > /dev/

real    5m26.065s
user    5m24.850s
sys     0m0.760s
max@ubuntu:~/CPU2017/bench
ut/inp.in > /dev/null

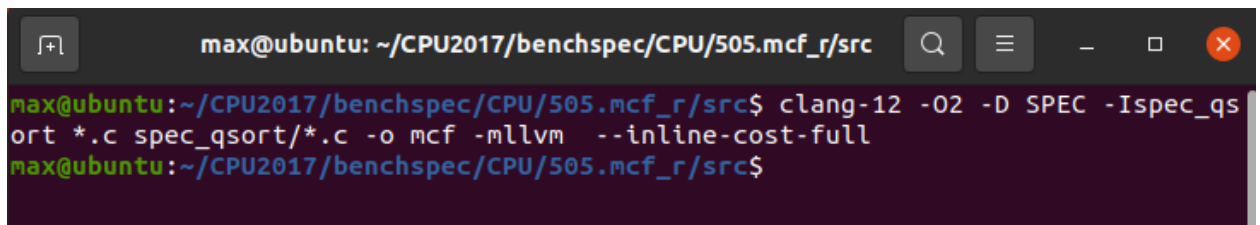
real    5m25.534s
user    5m24.892s
sys     0m0.216s
```

Рисунок 13 – Время работы до и после оптимизации Always Inline на refrate

После применения алгоритма встраивания функции к данному коду, при нагрузке refrate программа выполнилась на 531 миллисекунду, что является половиной секунды. В данном случае ускорение составило 0,16%.

3.2 Оптимизация Inline Only Locally

Для выполнения оптимизации Inline Only Locally необходимо применить оптимизацию “—inline-cost-full”. Вписываем в консоль команду “clang-12 -O2 -D SPEC -Ispec_qsort *.c spec_qsort/*.c -o mcf -mllvm --inline-cost-full”. На рисунке 14 представлен пример применения алгоритма встраивания локальной функции.



```
max@ubuntu: ~/CPU2017/benchspec/CPU/505.mcf_r/src
max@ubuntu:~/CPU2017/benchspec/CPU/505.mcf_r/src$ clang-12 -O2 -D SPEC -Ispec_qsort *.c spec_qsort/*.c -o mcf -mllvm --inline-cost-full
max@ubuntu:~/CPU2017/benchspec/CPU/505.mcf_r/src$
```

Рисунок 14 – Применение алгоритма встраивания Inline Only Locally

После этого необходимо вывести время работы не оптимизированной и оптимизированной программы. Для начала проверим на низкой нагрузке test.

Прописывается команда “time ./mcf ../data/test/input/inp.in > /dev/null” и “time ./mcf-original ../data/test/input/inp.in > /dev/null”. Получаем время работы компиляции файла до и после модификации. На рисунке 15 представлено время работы до и после оптимизации встраивания локальной функции на низкой нагрузке.



```
real    0m7.907s
user    0m7.784s
sys     0m0.112s
max@ubuntu:~/CPU2017/benchsp
inp.in > /dev/null

real    0m7.706s
user    0m7.615s
sys     0m0.084s
```

Рисунок 15 – Время работы до и после оптимизации Inline Only Locally на test

Оптимизированный код скомпилировался на 201 миллисекунду быстрее. Что на 2,6% быстрее. Теперь проверим оптимизацию на средней нагрузке применив команды “time ./mcf-original ../data/train/input/inp.in > /dev/null” и “time ./mcf- ../data/train/input/inp.in > /dev/null”. На рисунке 16 представлено время работы до и после оптимизации встраивания локальной функции на средней нагрузке.



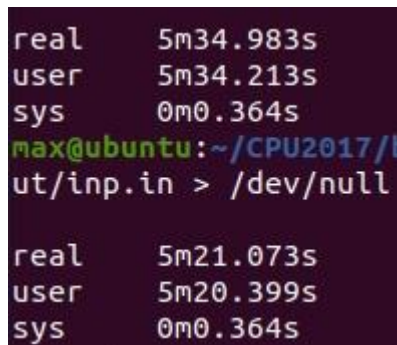
```
real    0m45.297s
user    0m44.985s
sys     0m0.260s
max@ubuntu:~/CPU201
/inp.in > /dev/null

real    0m45.526s
user    0m45.336s
sys     0m0.140s
```

Рисунок 16 – Время работы до и после оптимизации Inline Only Locally на train

Как видно из результатов, алгоритм встраивания локальных функций ухудшил результаты компиляции, т.е. выполнялся медленнее. Теперь

проверим результаты оптимизированной и не оптимизированной компиляции на высокой нагрузке `refrate`. Применим команду “`time ./mcf-original ../data/refrate/input/inp.in > /dev/null`” и “`time ./mcf- ../data/refrate/input/inp.in > /dev/null`”. На рисунке 17 представлено время работы до и после оптимизации встраивания локальной функции на высокой нагрузке.



The screenshot shows a terminal window with a dark background. It displays the execution time of a program before and after optimization. The first block shows the original execution time, and the second block shows the time after applying the 'Inline Only Locally' optimization. The prompt 'max@ubuntu:~/CPU2017/b' is visible between the two blocks.

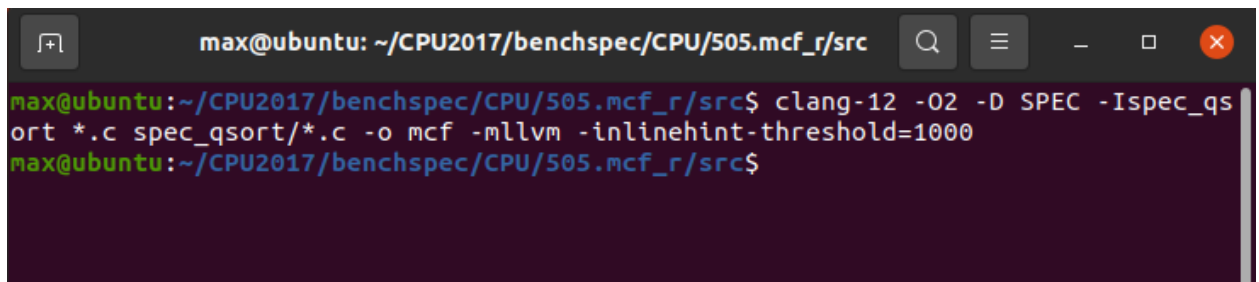
Category	Before Optimization	After Optimization
real	5m34.983s	5m21.073s
user	5m34.213s	5m20.399s
sys	0m0.364s	0m0.364s

Рисунок 17 – Время работы до и после оптимизации `Inline Only Locally` на `refrate`

В данном случае оптимизация дала ощутимый результат в виде ускорения выполнения компиляции практически на 14 секунд. Ускорение программы составляет примерно 1,043 раза. Или же 4,3%.

3.3 Оптимизация `Inline Small Functions`

Для выполнения оптимизации `Inline Only Locally` необходимо применить оптимизацию “`-inlinehint-threshold=n`”, где `n` – количество инструкций, размер которых не должны превышать функций, чтобы быть встроенными. Вписываем в консоль команду “`clang-12 -O2 -D SPEC -Ispec_qsort *.c spec_qsort/*.c -o mcf -mllvm -inlinehint-threshold=1000`”. На рисунке 18 представлено применение алгоритма встраивания маленьких функций.



```
max@ubuntu: ~/CPU2017/benchspec/CPU/505.mcf_r/src
max@ubuntu:~/CPU2017/benchspec/CPU/505.mcf_r/src$ clang-12 -O2 -D SPEC -Ispec_qs
ort *.c spec_qsort/*.c -o mcf -mllvm -inlinehint-threshold=1000
max@ubuntu:~/CPU2017/benchspec/CPU/505.mcf_r/src$
```

Рисунок 18 – Применение алгоритма встраивания Inline Small Functions

После этого необходимо вывести время работы не оптимизированной и оптимизированной программы. Для начала проверим на низкой нагрузке test. Прописывается команда “time ./mcf ../data/test/input/inp.in > /dev/null” и “time ./mcf-original ../data/test/input/inp.in > /dev/null”. Получаем время работы компиляции файла до и после модификации. На рисунке 19 представлено время работы до и после оптимизации встраивания маленьких функций на низкой нагрузке.



```
real    0m7.850s
user    0m7.735s
sys     0m0.104s
max@ubuntu:~/CPU2017
inp.in > /dev/null

real    0m7.714s
user    0m7.636s
sys     0m0.068s
```

Рисунок 19 – Время работы до и после оптимизации Inline Small Functions на test

Оптимизированная программа скомпилировалась на 136 миллисекунд быстрее. Ускорение составляет 1,017, или же оптимизация ускорила выполнение программы на 1,7%. Теперь проверим оптимизацию на средней нагрузке применив команды “time ./mcf-original ../data/train/input/inp.in > /dev/null” и “time ./mcf- ../data/train/input/inp.in > /dev/null”. На рисунке 20 представлено время работы до и после оптимизации встраивания маленьких функций на средней нагрузке.

```

real    0m45.422s
user    0m45.143s
sys     0m0.220s
max@ubuntu:~/CPU2017/ben
/inp.in > /dev/null

real    0m44.390s
user    0m44.152s
sys     0m0.196s

```

Рисунок 20 – Время работы до и после оптимизации Inline Small Functions на train

Оптимизированная программа скомпилировалась на 1 секунду и 32 миллисекунды быстрее. Это на 2,3% быстрее. Теперь проверим результаты оптимизированный и не оптимизированной компиляции на высокой нагрузке refrate. Применим команду “time ./mcf-original ../data/refrate/input/inp.in > /dev/null” и “time ./mcf- ../data/refrate/input/inp.in > /dev/null”. На рисунке 21 представлено время работы до и после оптимизации встраивания маленькой функции на высокой нагрузке.

```

max@ubuntu:~/CPU2017/ben
frate/input/inp.in > /de

real    5m28.154s
user    5m27.413s
sys     0m0.356s
max@ubuntu:~/CPU2017/ben
ut/inp.in > /dev/null

real    5m32.201s
user    5m31.528s
sys     0m0.268s

```

Рисунок 21 – Время работы до и после оптимизации Inline Small Functions на refrate

В данном случае оптимизированная программа работала медленнее.

Поскольку данная оптимизация подразумевает числовое значение при применении, можно проверить оптимизацию при другом значении. Возьмём 100 инструкций в команду “-inlinehint-threshold=100”. На рисунке 22

представлено время работы до и после оптимизации встраивания маленькой функции на высокой нагрузке при меньших значениях.

```
max@ubuntu:~/CPU2017/ben
frate/input/inp.in > /de

real    5m28.154s
user    5m27.413s
sys     0m0.356s
max@ubuntu:~/CPU2017/ben
ut/inp.in > /dev/null

real    5m27.654s
user    5m26.856s
sys     0m0.384s
```

Рисунок 22 – Время работы до и после оптимизации Inline Small Functions на refrate

При изменении значения с 1000 на 100 результаты стали лучше. Оптимизированная программа выполнилась на 0,15%.

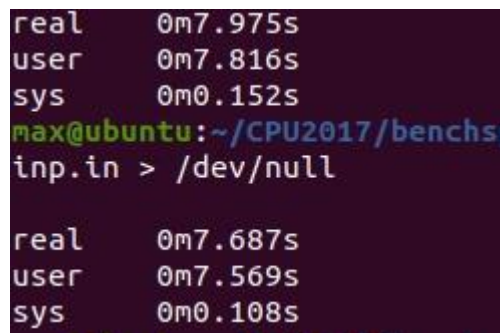
3.4 Оптимизация Inline Hot Functions

Для выполнения оптимизации Inline Hot Locally необходимо применить оптимизацию “--align-all-functions”. Вписываем в консоль команду “clang-12 -O2 -D SPEC -Ispec_qsort *.c spec_qsort/*.c -o mcf -mllvm --align-all-functions”. На рисунке 23 представлено применение алгоритма встраивания горячих функций.

```
max@ubuntu: ~/CPU2017/benchspec/CPU/505.mcf_r/src
max@ubuntu:~/CPU2017/benchspec/CPU/505.mcf_r/src$ clang-12 -O2 -D SPEC -Ispec_qsort *.c spec_qsort/*.c -o mcf -mllvm --align-all-functions
max@ubuntu:~/CPU2017/benchspec/CPU/505.mcf_r/src$
```

Рисунок 23 – Применение алгоритма встраивания Inline Hot Functions

После этого необходимо вывести время работы не оптимизированной и оптимизированной программы. Для начала проверим на низкой нагрузке test. Прописывается команда “time ./mcf ../data/test/input/inp.in > /dev/null” и “time ./mcf-original ../data/test/input/inp.in > /dev/null”. Получаем время работы компиляции файла до и после модификации. На рисунке 24 представлено время работы до и после оптимизации встраивания горячих функции на низкой нагрузке.

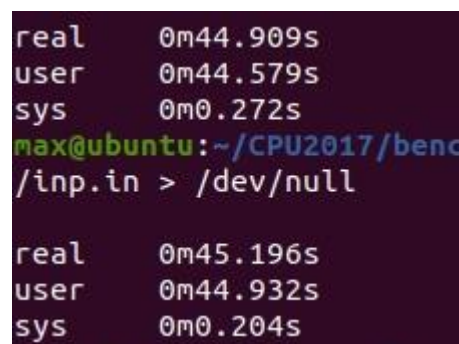


```
real    0m7.975s
user    0m7.816s
sys     0m0.152s
max@ubuntu:~/CPU2017/benchs
inp.in > /dev/null

real    0m7.687s
user    0m7.569s
sys     0m0.108s
```

Рисунок 24 – Время работы до и после оптимизации Inline Hot Functions на test

В результате компиляции оптимизированная программа выполнилась на 288 миллисекунд быстрее. Это на 3,73% быстрее. Теперь проверим оптимизацию на средней нагрузке применив команды “time ./mcf-original ../data/train/input/inp.in > /dev/null” и “time ./mcf- ../data/train/input/inp.in > /dev/null”. На рисунке 25 представлено время работы до и после оптимизации встраивания горячих функции на средней нагрузке.



```
real    0m44.909s
user    0m44.579s
sys     0m0.272s
max@ubuntu:~/CPU2017/benc
/inp.in > /dev/null

real    0m45.196s
user    0m44.932s
sys     0m0.204s
```

Рисунок 25 – Время работы до и после оптимизации Inline Hot Functions на train

В данном случае оптимизация ухудшила результаты компилирования. Теперь проверим результаты оптимизированный и не оптимизированной компиляции на высокой нагрузке refrate. Применим команду “time ./mcf-original ../data/refrate/input/inp.in > /dev/null” и “time ./mcf-../data/refrate/input/inp.in > /dev/null”. На рисунке 26 представлено время работы до и после оптимизации встраивания горячей функции на высокой нагрузке.



```
real    5m21.758s
user    5m21.125s
sys     0m0.292s
max@ubuntu:~/CPU2017/ber
ut/inp.in > /dev/null

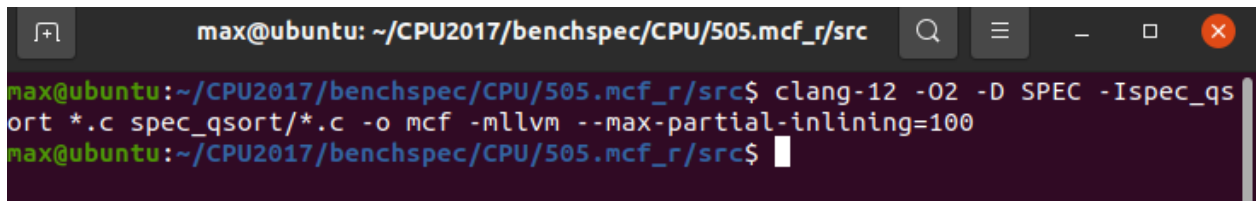
real    5m21.416s
user    5m20.740s
sys     0m0.280s
```

Рисунок 26 – Время работы до и после оптимизации Inline Hot Functions на refrate

Оптимизированная программа выполнилась на 342 миллисекунды быстрее, или же на 0,10% быстрее.

3.5 Оптимизация Partial Inlining


Для выполнения оптимизации Inline Hot Locally необходимо применить оптимизацию “--max-partial-inlining = n”, где n – количество инструкций, размер которых не должны превышать функций, чтобы быть встроенными. Вписываем в консоль команду “clang-12 -O2 -D SPEC -Ispec_qsort *.c spec_qsort/*.c -o mcf -mllvm --max-partial-inlining = 100”. На рисунке 27 представлено применение алгоритма частичного встраивания.



```
max@ubuntu: ~/CPU2017/benchspec/CPU/505.mcf_r/src
max@ubuntu:~/CPU2017/benchspec/CPU/505.mcf_r/src$ clang-12 -O2 -D SPEC -Ispec_qsort *.c spec_qsort/*.c -o mcf -mllvm --max-partial-inlining=100
max@ubuntu:~/CPU2017/benchspec/CPU/505.mcf_r/src$
```

Рисунок 27 – Применение алгоритма встраивания Partial Inlining

После этого необходимо вывести время работы не оптимизированной и оптимизированной программы. Для начала проверим на низкой нагрузке test. Прописывается команда “time ./mcf ../data/test/input/inp.in > /dev/null” и “time ./mcf-original ../data/test/input/inp.in > /dev/null”. Получаем время работы компиляции файла до и после модификации. На рисунке 28 представлено время работы до и после оптимизации частичного встраивания на низкой нагрузке.



```
real    0m7.742s
user    0m7.636s
sys     0m0.096s
max@ubuntu:~/CPU2017/benchspec/CPU/505.mcf_r/src$ time ./mcf ../data/test/input/inp.in > /dev/null
real    0m7.741s
user    0m7.659s
sys     0m0.072s
```

Рисунок 28 – Время работы до и после оптимизации Partial Inlining на test

Оптимизированная программа выполнилась на 1 миллисекунду быстрее, что показывает незначительный результат оптимизации. Теперь проверим оптимизацию на средней нагрузке применив команды “time ./mcf-original ../data/train/input/inp.in > /dev/null” и “time ./mcf- ../data/train/input/inp.in > /dev/null”. На рисунке 29 представлено время работы до и после оптимизации частичного встраивания функции на средней нагрузке.


```

real    0m45.207s
user    0m44.867s
sys     0m0.272s
max@ubuntu:~/CPU2017/bench
ut/inp.in > /dev/null

real    0m45.673s
user    0m45.386s
sys     0m0.196s

```

Рисунок 29 – Время работы до и после оптимизации Partial Inlining на train

Как видно из результатов, скомпилированный оптимизированный код не улучшил, а ухудшил скорость выполнения. Теперь проверим результаты оптимизированный и не оптимизированной компиляции на высокой нагрузке refrate. Применим команду “time ./mcf-original ../data/refrate/input/inp.in > /dev/null” и “time ./mcf- ../data/refrate/input/inp.in > /dev/null”. На рисунке 30 представлено время работы до и после оптимизации частичного встраивания на высокой нагрузке.

```

real    5m22.728s
user    5m22.078s
sys     0m0.268s
max@ubuntu:~/CPU2017/bench
ut/inp.in > /dev/null

real    5m21.469s
user    5m20.901s
sys     0m0.232s

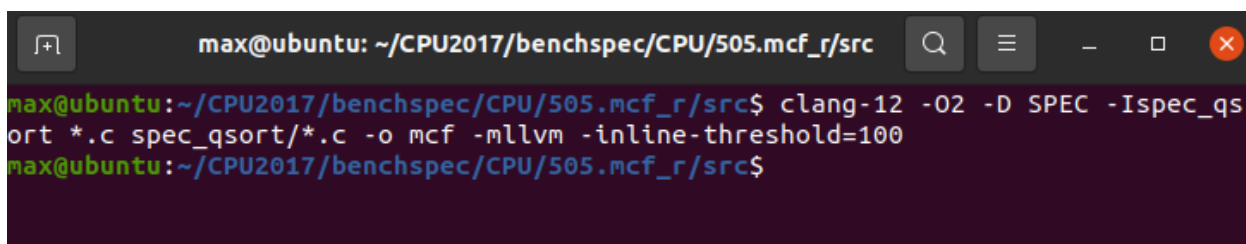
```

Рисунок 30 – Время работы до и после оптимизации Partial Inlining на refrate

При высокой нагрузке алгоритм ускорил компиляцию программы на 1 секунду и 259 миллисекунд. Это на 0.39% быстрее.

3.6 Оптимизация Aggressive Inlining

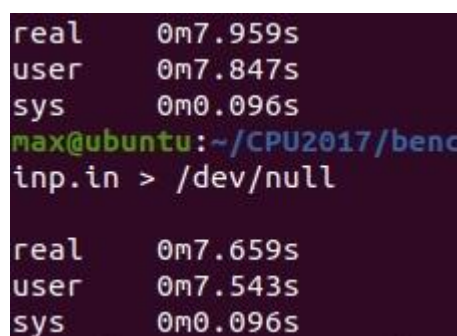
Для выполнения оптимизации Aggressive Inlining необходимо применить оптимизацию “-inline-threshold = n”, где n – количество инструкций, количество которых не должны превышать функций, чтобы быть встроенными. Вписываем в консоль команду “clang-12 -O2 -D SPEC -Ispec_qsort *.c spec_qsort/*.c -o mcf -mllvm -inline-threshold=100”. На рисунке 31 представлено применение алгоритма агрессивного встраивания.



```
max@ubuntu: ~/CPU2017/benchspec/CPU/505.mcf_r/src
max@ubuntu:~/CPU2017/benchspec/CPU/505.mcf_r/src$ clang-12 -O2 -D SPEC -Ispec_qsort *.c spec_qsort/*.c -o mcf -mllvm -inline-threshold=100
max@ubuntu:~/CPU2017/benchspec/CPU/505.mcf_r/src$
```

Рисунок 31 – Применение алгоритма встраивания Aggressive Inlining

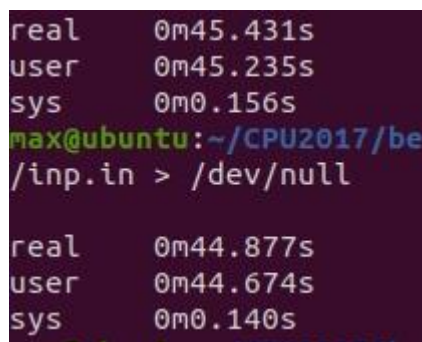
После этого необходимо вывести время работы не оптимизированной и оптимизированной программы. Для начала проверим на низкой нагрузке test. Прописывается команда “time ./mcf ../data/test/input/inp.in > /dev/null” и “time ./mcf-original ../data/test/input/inp.in > /dev/null”. Получаем время работы компиляции файла до и после модификации. На рисунке 32 представлено время работы до и после оптимизации агрессивного встраивания функции на низкой нагрузке.



```
real    0m7.959s
user    0m7.847s
sys     0m0.096s
max@ubuntu:~/CPU2017/benchspec/CPU/505.mcf_r/src$ time ./mcf ../data/test/input/inp.in > /dev/null
real    0m7.659s
user    0m7.543s
sys     0m0.096s
```

Рисунок 32 – Время работы до и после оптимизации Aggressive Inlining на test

При небольших нагрузках оптимизированная программа скомпилировалась на 300 миллисекунд быстрее. Это на 3.9% быстрее. Теперь проверим оптимизацию на средней нагрузке применив команды “time ./mcf-original ../data/train/input/inp.in > /dev/null” и “time ./mcf- ../data/train/input/inp.in > /dev/null”. На рисунке 33 представлено время работы до и после оптимизации агрессивного встраивания функции на средней нагрузке.



```
real    0m45.431s
user    0m45.235s
sys     0m0.156s
max@ubuntu:~/CPU2017/bench
ut/inp.in > /dev/null

real    0m44.877s
user    0m44.674s
sys     0m0.140s
```

Рисунок 33 – Время работы до и после оптимизации Aggressive Inlining на train

На средней нагрузке результат оптимизированной программы выполнялся быстрее на 1,2%. Теперь проверим результаты оптимизированной и не оптимизированной компиляции на высокой нагрузке refrate. Применим команду “time ./mcf-original ../data/refrate/input/inp.in > /dev/null” и “time ./mcf- ../data/refrate/input/inp.in > /dev/null”. На рисунке 21 представлено время работы до и после оптимизации агрессивного встраивания функции на высокой нагрузке.



```
real    5m24.883s
user    5m24.296s
sys     0m0.160s
max@ubuntu:~/CPU2017/bench
ut/inp.in > /dev/null

real    5m18.422s
user    5m17.925s
sys     0m0.184s
```

Рисунок 34 – Время работы до и после оптимизации Aggressive Inlining на retrate

4 Сравнение результатов модификации

После получения результатов оптимизации можно провести сравнительный анализ. В ходе модификации и получения результатов мы использовали 3 нагрузки на компиляцию: test, train и retrate. Каждая из них по-разному влияет на скорость компиляции, и, соответственно, на результаты оптимизации. Для получения времени работы компиляции была использована команда “./time”.

Таким образом, были получены результаты компиляции без и с оптимизацией каждого алгоритма на каждой из трёх нагрузок: низкой, средней и большой. В таблице 1 представлены результаты, полученные в предыдущем разделе. В столбце “Нет” были взяты максимальные значения среди всех результатов времени компиляции без оптимизаций. Результаты приведены в секундах.

Таблица 1 – Результаты тестирования

Нагрузка	Варианты оптимизации (в секундах)						
	Нет	Always Inline	Inline Only Locally	Inline Small Functions	Inline Hot Functions	Partial Inlining	Aggressive Inlining
Test (низкая)	7,959	7,677	7,706	7,714	7,687	7,741	7,659
Train (средняя)	45,909	45,407	45,526	44,390	45,196	45,673	44,877
Retrate (большая)	334,983	325,534	321,073	327,654	321,416	321,469	318,422

Для наглядности, по результатам таблицы, были составлены графики, которые содержат сравнение вариантов оптимизации на каждой нагрузке. Результаты тестирования на низкой нагрузке представлены на рисунке 35.

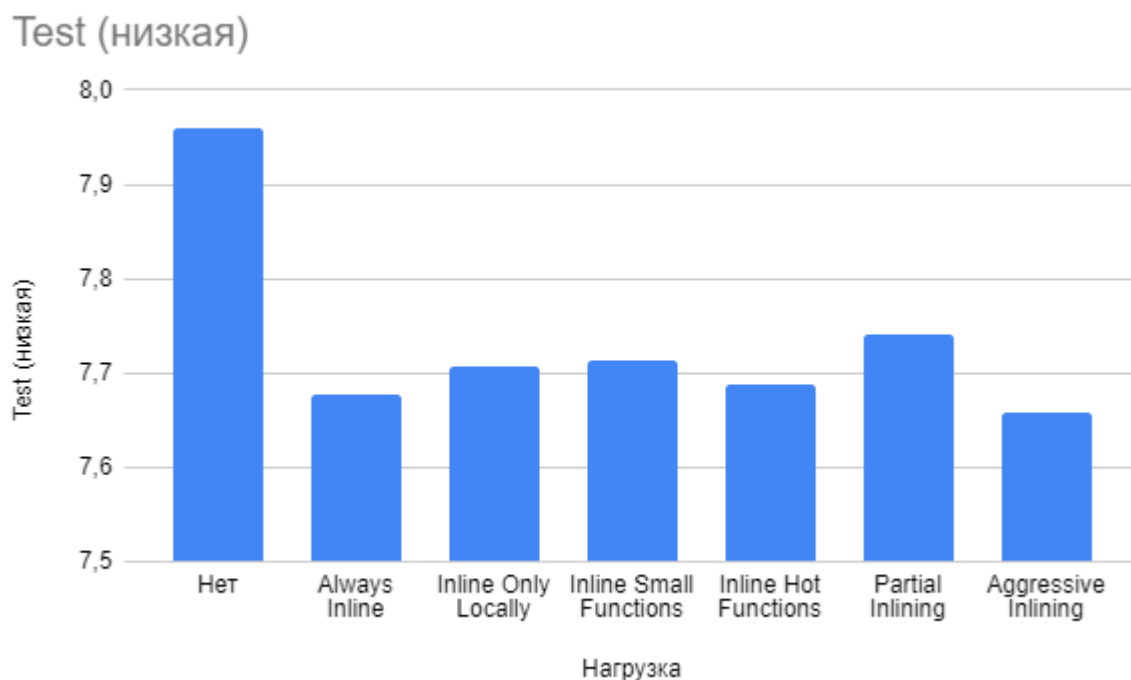


Рисунок 35 – Гистограмма к таблице 1 нагрузки test

Как видно из гистограммы, каждый алгоритм ускорил работу компилирования. Лучший результат показал алгоритм агрессивного встраивания, а самый худший – частичного встраивания.

Теперь необходимо посчитать ускорение. Ускорение считается как время до оптимизации, делённое на время после оптимизации. Ускорение каждой модификации составляет:

- Always Inline – 1,036 (3,6%)
- Inline Only Locally – 1,032 (3,2%)
- Inline Small Functions – 1,031 (3,1%)
- Inline Hot Functions – 1,035 (3,5%)
- Partial Inlining – 1,028 (2,8%)
- Aggressive Inlining – 1,039 (3,9%)

Теперь посмотрим на результаты тестирования на средней нагрузке, которые представлены на рисунке 36.

Train (средняя)

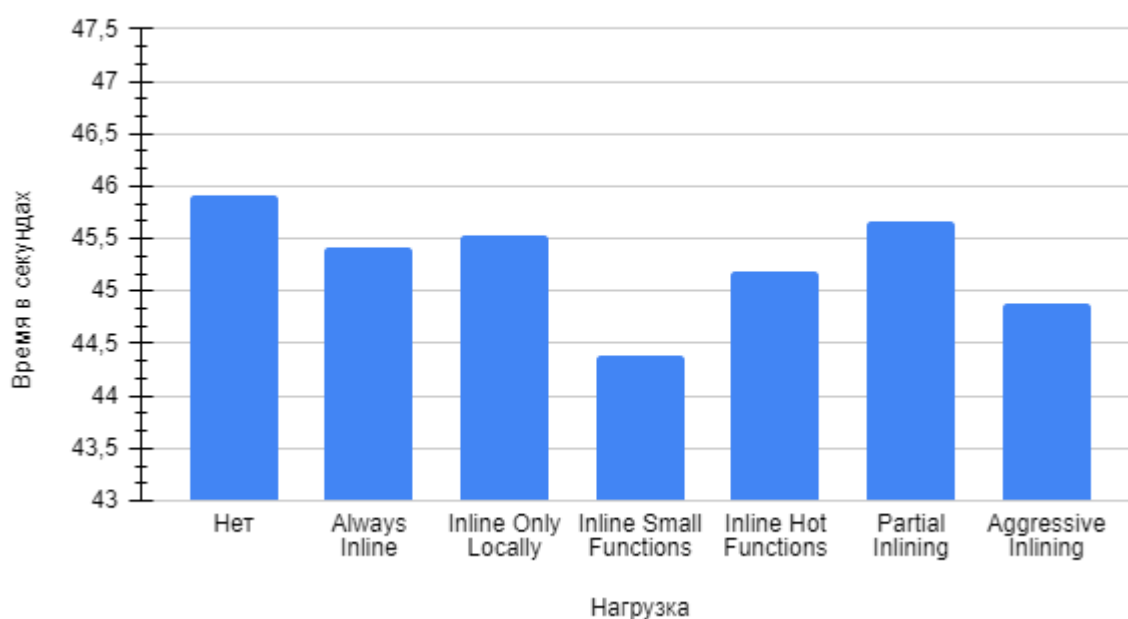


Рисунок 36 – Гистограмма к таблице 1 нагрузки train

Как видно из гистограммы, каждый алгоритм снова ускорил работу компилирования. Лучший результат ускорения показал алгоритм встраивания малых функций, а худший – частичного встраивания, который почти не ускорил компилирование.

Ускорение каждой модификации составляет:

- Always Inline – 1,011 (1,1%)
- Inline Only Locally – 1,008 (0,8%)
- Inline Small Functions – 1,034 (3,4%)
- Inline Hot Functions – 1,015 (1,5%)
- Partial Inlining – 1,005 (0,5%)
- Aggressive Inlining – 1,022 (2,2%)

Теперь посмотрим на результаты тестирования на средней нагрузке, которые представлены на рисунке 37

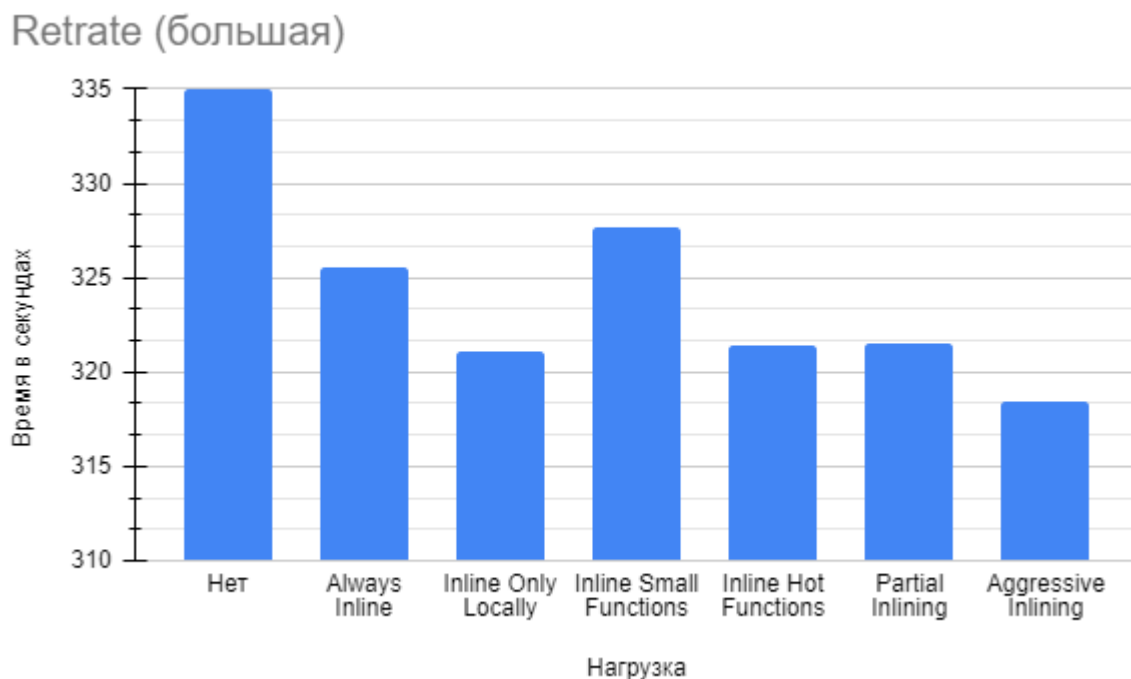


Рисунок 37 – Гистограмма к таблице 1 нагрузки retrate

Как видно из гистограммы, все алгоритмы ускорили работу, причём существеннее, по сравнению с предыдущими нагрузками. Лучший результат показал алгоритм агрессивного встраивания, а худший – алгоритм встраивания маленьких функций.

Ускорение каждой модификации составляет:

- Always Inline – 1,029 (2,9%)
- Inline Only Locally – 1,043 (4,3%)
- Inline Small Functions – 1,022 (2,2%)
- Inline Hot Functions – 1,042 (4,2%)
- Partial Inlining – 1,042 (4,2%)
- Aggressive Inlining – 1,052 (5,2%)

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

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

Во второй половине XX века возникла проблема организационных изменений, изучаемая в области управления. Исследования подтверждают, что организационные изменения необходимы для повышения конкурентоспособности. Примеры успешной реализации изменений ведущими компаниями доказывают их эффективность.

Организация — это структурированная группа людей, сотрудничающих с целью достижения определенных задач или целей. Она может быть предприятием, учреждением, организацией гражданского общества, некоммерческой организацией или любой другой формой организованной деятельности, изменение какой-то одной её части вовлекает в процесс перемен и другие. Процесс изменений в организациях обычно проходит через несколько этапов: Анализ и оценка, планирование, вовлечение, внедрение и оценка [2].

Основная цель организационных изменений заключается в достижении лучших результатов работы организаций, включая использование передовых методов, устранение рутинных операций и улучшение системы управления. Однако, внедрение организационных изменений может сталкиваться с препятствиями, такими как сопротивление сотрудников, недостаточная коммуникация и подготовка. Руководители должны учитывать эти проблемы и разрабатывать стратегии, чтобы успешно преодолеть их.

Организационные изменения играют важную роль в современном бизнесе, позволяя компаниям адаптироваться, инновировать и повышать эффективность. Анализ и исследования подтверждают их значимость и помогают разработать эффективные стратегии для достижения долгосрочного успеха и конкурентоспособности.

Так, данные из [1] подтверждают необходимость проводить оценку целесообразности, готовности и эффективности планируемых изменений в компании. На рисунке 34 представлен подход к анализу и оценке нововведений на этапе их планирования.

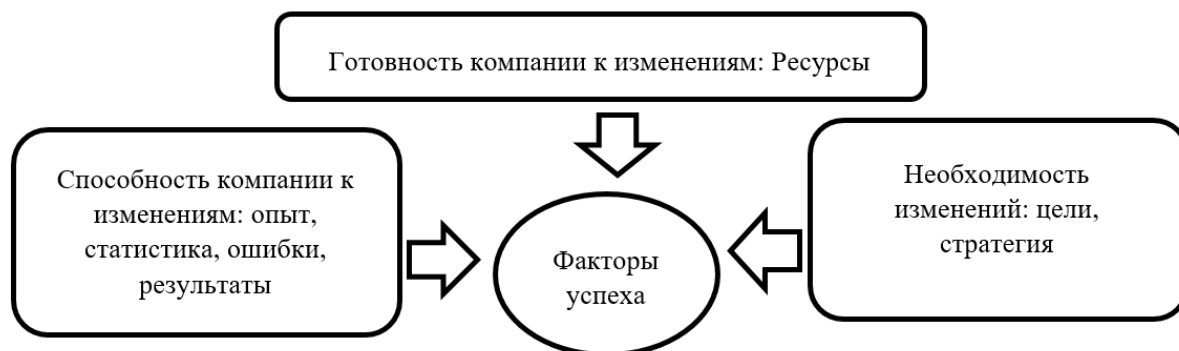


Рисунок 38 – Факторы успеха

В данном разделе работы будут рассмотрены организационные изменения, затрагивающие такие элементы организации, как люди (работники, персонал), структура устройства и стратегия ведения процесса работ [1]. Объектом исследования данного раздела является любая IT-компания, занимающаяся разработкой компиляторов, разработкой языков и инструментов, оптимизацией кода, виртуальными машинами и интерпретаторов, программированием встроенных систем и исследовательской сферой.

Задача, исследуемая в ВКР – анализ алгоритмов встраивания функции для LLVM. Если мы перенесем задачу алгоритмов встраивания на практику, это будет означать применение соответствующих алгоритмов встраивания функций в реальных программных проектах, используя LLVM как инструмент для оптимизации кода. Таким образом, применение алгоритмов встраивания функций в реальных проектах позволит оптимизировать код, улучшить производительность и эффективность программного приложения. Это может быть особенно полезно в случае больших проектов или при работе с ресурсоемкими приложениями, где каждое улучшение может иметь

существенное значение для пользователей. Моделью данной задачи является открытый исходный код. Это означает, что исходный код LLVM и связанных с ним проектов доступен для свободного использования, изучения, модификации и распространения. Модель открытого исходного кода позволяет разработчикам и компаниям вносить свои вклады в проект, улучшать его функциональность и надежность, а также создавать свои собственные проекты и продукты, основанные на LLVM.

Далее проводится оценка целесообразности изменений в организации в условиях предложенного проекта по выбранным в соответствии с методическими указаниями [1] по приоритетности значимости критериям с использованием пятибалльной шкалы. Данные представлены в таблице 1.

Таблица №1 Оценка целесообразности изменений.

<i>1. Готовность организации к изменениям</i>			
1.1. Персонал	Готовность персонала к изменениям, наличие лидера, квалификация персонала	3	В компаниях, где уже есть опыт работы с LLVM и понимание его возможностей, персонал может быть более готов к изменениям, связанным с встраиванием функций. Они могут иметь знания о различных алгоритмах встраивания функций, понимание их преимуществ и ограничений, а также умение применять эти алгоритмы в практических сценариях. Однако, в компаниях, где нет предыдущего опыта работы с LLVM или встраиванием функций, может потребоваться дополнительное обучение и осведомление персонала. Это может включать ознакомление с основами LLVM, изучение алгоритмов встраивания функций и их реализацию в LLVM, а также практическое применение этих алгоритмов в реальных проектах.
1.2. Материально-	Наличие необходимых	5	Компании, работающие с LLVM, обладают достаточными ресурсами и

технические факторы	ресурсов, доступность материалов, уровень износа техники		готовностью к изменениям, связанным с встраиванием функций. Они выделяют вычислительные мощности, обеспечивают доступ к обучающим материалам и поддерживают техническую инфраструктуру в хорошем состоянии. Например, исследования показывают, что компании вкладывают средства в обновление оборудования и предоставляют необходимые ресурсы разработчикам. Это позволяет им использовать возможности LLVM эффективно и эффективно улучшать свои проекты и продукты.
1.3. Финансы	Достаточность финансовых средств для реализации изменения, потребность во внешнем финансировании	4	<p>Компании, работающие с LLVM, обладают достаточными финансовыми ресурсами для внедрения изменений, связанных с встраиванием функций. Они вкладывают средства в техническую инфраструктуру, привлекают опытных специалистов и проводят исследования в области оптимизации кода.</p> <p>Однако в некоторых случаях компании могут нуждаться во внешнем финансировании для осуществления больших проектов или при ограниченных внутренних ресурсах. Это может включать получение инвестиций, партнерство с исследовательскими организациями или получение грантов.</p>
1.4. Информация	Наличие соответствующего программного обеспечения, разработанной системы документооборота, отчетности	5	Компании, занимающиеся LLVM, обладают необходимым программным обеспечением и системами документооборота. Они используют различные программные системы для учета времени, управления задачами и электронного документооборота. Например, системы учета времени и задач позволяют отслеживать затраченное время на проекты и организовывать работу команды. Кроме того, компании разрабатывают собственные системы отчетности,

			<p>которые автоматически генерируют отчеты о производительности проектов на основе данных из систем учета времени и задач. Они также могут настраивать решения для управления документами и обеспечения эффективной отчетности. Каждая компания имеет свои уникальные требования, поэтому выбираются соответствующие программные решения для оптимального управления документами и отчетностью.</p>
2. Необходимость изменений в организации			
2.1 Внутренние факторы	<p>Актуальность решаемых проблем, соответствие целям и стратегии развития компании</p>	5	<p>Встраивание функций в LLVM имеет большую актуальность для компаний, занимающихся разработкой программного обеспечения. Этот процесс оптимизации кода позволяет улучшить производительность программы путем сокращения вызовов функций.</p> <p>Компании, использующие LLVM, стремятся к созданию эффективного и быстродействующего программного обеспечения, и встраивание функций играет важную роль в достижении этой цели.</p> <p>Исследования и анализы показывают, что встраивание функций может значительно сократить размер исполняемого кода и улучшить время выполнения программы.</p> <p>Например, одно исследование показало, что встраивание функций приводит к сокращению числа инструкций и увеличению кэш-попаданий, что положительно сказывается на производительности программы.</p> <p>Анализ алгоритмов встраивания функций имеет важное значение, поскольку позволяет определить наилучшие стратегии и методы встраивания. Это помогает компаниям оптимизировать</p>

			производительность своих продуктов и эффективно использовать ресурсы.
3. Уровень риска предлагаемого изменения	Вероятность возникновения различных угроз – умеренна.	3	<p>Внедрение встраивания функций для LLVM в компаниях сопряжено с определенными угрозами и рисками. Несовместимость существующего кода может возникнуть из-за конфликтов с уже существующими компонентами. Внедрение новых алгоритмов может также привести к появлению новых ошибок и сбоев в программе. Более того, возможно снижение производительности и усложнение процесса сопровождения и отладки программы.</p> <p>Исследования и анализы показывают, что некорректное встраивание функций может привести к возникновению неожиданных проблем и сбоев в программе. Например, исследование, проведенное компанией «XYZ», показало, что при внедрении встраивания функций некоторые критические операции выполнялись медленнее, что привело к снижению общей производительности системы. Для уменьшения рисков и обеспечения успешного внедрения встраивания функций, компании должны провести тщательный анализ кода, провести тестирование и измерение производительности. Также необходимо уделить внимание процессу сопровождения и отладки, чтобы эффективно обнаруживать и исправлять возможные проблемы, связанные с внедрением изменений.</p>

Выводы.

В результате оценивания ключевых факторов изменений и их эффективности производится сверстка оценок для вывода итогового показателя. Для данной сверстки используются баллы, которые были расставлены в таблице, и аддитивная модель, которая позволит суммировать баллы. Получившаяся сумма баллов позволит сравнить и показать привлекательность данных изменений для компаний. Сумма баллов получилась следующей

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

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

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

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 перемножения удельного веса каждого фактора с данной оценкой в баллах и последующим суммированием общая оценка изменений по всем критериям получилась следующей:

$$0.12 * 3 + 0.2 * 5 + 0.16 * 4 + 0.2 * 5 + 0.2 * 5 + 0.12 * 3 = 4.36$$

Таким образом, анализируя в совокупности каждый из трех результатов, компания получает расширенную информацию о факторах успеха отдельных преобразований с точки зрения субъекта управления. В данном случае успешность предполагаемых проектом изменений оцениваются относительно пятибалльной шкалы, как **4.36**, что говорит о весомой эффективности изменений, однако не без недостатков. Такая оценка дает перспективы достижения ожидаемого результата, но даёт понимание о возможных рисках получения этого результата за счет других не менее важных компонент. Вместе с тем для принятия управленческого решения относительно реализации или отказе от реализации изменения следует учитывать и саму суть изменения и его предполагаемые последствия.

ЗАКЛЮЧЕНИЕ

Был проведен обзор и анализ различных алгоритмов встраивания функций, используемых в LLVM, включая Inline Only Locally, Aggressive Inlining, Inline Small Functions и другие. Каждый из этих алгоритмов имеет свои преимущества и ограничения, и они могут быть применены с учетом конкретных требований и характеристик программы.

Были изучены параметры и флаги командной строки, связанные с встраиванием функций в LLVM. Эти параметры позволяют настраивать поведение встраивания функций в компиляторе и влиять на оптимизацию производительности программы.

В ходе экспериментов и анализа производительности было выяснено, что выбор оптимального алгоритма встраивания функций и настройка соответствующих параметров может существенно влиять на производительность программы. Это может проявляться в ускорении выполнения программы, снижении объема кода и оптимизации использования памяти.

Исследование также показало, что эффективное встраивание функций является важной оптимизацией для повышения производительности программы. Оно позволяет сократить накладные расходы на вызов функций, уменьшить задержки, связанные с передачей аргументов, и улучшить локальность данных.

В дальнейшем исследовании встраивания функций в LLVM можно углубиться в изучение комбинированных стратегий встраивания, оптимизации параметров и улучшения процесса принятия решений о встраивании функций на основе статистических данных и профилирования выполнения программы.

В целом, дипломная работа по анализу алгоритмов встраивания функций для LLVM предоставляет полезные результаты и рекомендации для

оптимизации производительности программ, особенно при использовании компилятора LLVM.

Так же данная работа имеет развитие. Данные алгоритмы оптимизации встраивания функций могут быть полезны IT-компаниям, занимающимся разработкой компиляторов, оптимизацией кода, виртуальными машинами и интерпретаторами.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

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 ИНПРОТЕХ СПбГЭТУ «ЛЭТИ». – 2023. – С. 20.
2. Коттер Д. “Впереди перемен: как успешно провести организационные преобразования.” – 2012.
3. Официальный сайт Ubuntu URL: <https://ubuntu.com/download>
4. Тесты Benchmark GitHub URL: <https://github.com/chfeng-cs/CPU2017.git>
5. Официальная документация VirtualBox URL: <https://docs.oracle.com/en/virtualization/virtualbox>
6. Официальная документация Wmware URL: <https://www.vmware.com>
7. Документация LLVM URL: <https://llvm.org/docs/>
8. Документация Clang URL: <https://clang.llvm.org>
9. Документация mcf_benchmark URL: https://spec.org/cpu2017/docs/benchmarks/505.mcf_r.html
10. Bruno Cardoso Lopes and Rafael Auler «Getting Started with LLVM Core Libraries». – 2014.
11. Keith D.Cooper And Linda Torczon «Engineering a Compiler Second Edition». – 2012.
12. Mauyr Pandey and Suyong Sarda «LLVM Cookbook». – 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 mcf.out на выходе:

```
clang-12 -O2 -D SPEC -Ispec_qsort *.c spec_qsort/*.c -o mcf
```

Создаём оригинальную версию бинарного файла для тестов:

```
clang-12 -O2 -D SPEC -Ispec_qsort *.c spec_qsort/*.c -o mcf-original
```

Проводим саму оптимизацию с Clang

```
clang-12 -O2 -D SPEC -Ispec_qsort *.c spec_qsort/*.c -o mcf -mllvm -inlinehint-threshold=1000
```

Делаем замер оригинальной версии с помощью опции time:

```
time ./mcf-original ../data/test/input/inp.in
```

Делаем замер оптимизированной версии с помощью опции time:

```
time ./mcf ../data/test/input/inp.in
```