

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

Направление подготовки: 09.03.01 “Информатика и вычислительная техника”
Профиль: “Вычислительные машины, комплексы, системы и сети”

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: “Средства трансформации исходного кода на основе
LLVM с учетом микроархитектурных свойств
вычислительных систем”**

Студент

А. В. Костин

Руководитель

к. т. н., доцент

А. А. Пазников

Консультант по экономическому
обоснованию

Н. И. Олехова

Консультант от кафедры

к. т. н., доцент, с. н. с.

И. С. Зуев

Санкт-Петербург
2022 г

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

Направление 09.03.01 “Информатика и
вычислительная техника”
Профиль “Вычислительные машины,
комплексы, системы и сети”
Факультет компьютерных технологий
и информатики
Кафедра вычислительной техники

УТВЕРЖДАЮ
Заведующий кафедрой ВТ
д. т. н., профессор
(М. С. Куприянов)
“___” _____ 202__ г.

**ЗАДАНИЕ
на выпускную квалификационную работу**

Студент А. В. Костин

Группа № 8305

1. Тема: Средства трансформации исходного кода на основе LLVM с
учетом микроархитектурных свойств вычислительных систем

(утверждена приказом № _____ от _____)

Место выполнения ВКР: Санкт-Петербургский государственный
электротехнический университет «ЛЭТИ» им. В. И. Ульянова (Ленина)

2. Объект исследования: средства трансформации исходного кода на
основе LLVM.

3. Предмет исследования: разделение и слияние циклов с помощью
средств трансформации LLVM.

4. Цель: исследование средств трансформации исходного кода на
основе инструментов LLVM для разделения и слияния циклов.

5. Исходные данные: исходные тексты и документация LLVM,
англоязычные книги и статьи в сети Интернет.

6. Содержание

- Анализ оптимизаций путём трансформации исходного кода.
- Ознакомление с инструментами LLVM.
- Исследование трансформаций разделения и слияния циклов.
- Ознакомление с трансформациями исходного кода с помощью Clang.

7. Дополнительный раздел: экономическое обоснование выпускной квалификационной работы.

8. Результаты: пояснительная записка, презентация, реферат.

Дата выдачи задания
«28» февраля 2022 г.

Дата представления ВКР к защите
«20» июня 2022 г.

Руководитель
к. т. н., доцент

А. А. Пазников

Студент

А. В. Костин

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

**Направление 09.03.01 «Информатика и
вычислительная техника»**

**Профиль «Вычислительные машины,
комплексы, системы и сети»**

**Факультет компьютерных технологий
и информатики**

Кафедра вычислительной техники

УТВЕРЖДАЮ
Заведующий кафедрой ВТ
д. т. н., профессор
(М. С. Куприянов)
“ ____ ” _____ 202__ г.

**КАЛЕНДАРНЫЙ ПЛАН
выполнения выпускной квалификационной работы**

Тема **Средства трансформации исходного кода на основе LLVM с
учетом микроархитектурных свойств вычислительных систем**

Студент А. В. Костин

Группа № **8305**

№ этапа	Наименование работ	Срок выполнения
1	Анализ предметной области, изучение литературы	08.01 – 31.01
2	Установка и изучение используемых технологий, обзор документации и примеров.	01.02 – 24.02
3	Окончательный выбор темы работы	25.02 – 28.02
4	Разбор модуля с трансформацией структуры	08.04 – 20.04
5	Написание собственного простого прохода LLVM	21.04 – 28.04
6	Изучение документации и примеров трансформаций с помощью Clang	29.04 – 12.05
7	Написание собственного инструмента с помощью Clang	13.05 – 16.05
8	Исследование инструментов LLVM для разделения и слияния циклов, проверка на простых тестах и формулирование результатов	17.05 – 25.05
9	Тестирование модулей	26.05 – 28.06
10	Оформление пояснительной записки	29.06 – 06.06
11	Предварительное рассмотрение работы	07.06
12	Представление работы к защите	08.06

Руководитель

к. т. н., доцент

А. А. Пазников

Студент

А. В. Костин

РЕФЕРАТ

Пояснительная записка содержит: 66 страниц, 26 рисунков, 6 таблиц, 1 приложение, 27 источников литературы.

Целью выпускной квалификационной работы ставится исследование средств трансформации исходного кода на основе инструментов LLVM для разделения и слияния циклов.

В выпускной квалификационной работе производится анализ предметной области, в котором рассматриваются изученные оптимизации путём трансформации кода, описываются используемые технологии и инструменты, процесс исследования и практические эксперименты, а также приводится описание трансформаций с помощью Clang AST.

Предмет исследования – трансформации разделения и слияния циклов. Оптимизации на их основе могут довольно значительно повлиять на производительность программы, тем не менее они не включены по умолчанию, что вызывает потребность в их анализе.

ABSTRACT

The purpose of the final qualification work is to study the means of transforming the source code based on the LLVM tools for separating and merging cycles.

The final qualification work provides an analysis of the subject area, which examines the studied optimizations by code transformation, describes the technologies and tools used, the research process and practical experiments, and also provides a description of transformations using Clang AST.

The subject of the study was the transformation of the division and merging of cycles. Optimizations based on them can significantly affect the performance of the program, however, they are not enabled by default, which necessitates their analysis.

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ	9
ВВЕДЕНИЕ	11
1 Оптимизации на основе трансформации исходного кода	13
1.1 Оптимизации доступа к памяти	13
1.2 Оптимизации компоновки машинного кода	15
1.3 Оптимизация структур	16
1.3.1 Разделение структур.....	17
1.3.2 Оптимизация выравнивания структур	17
1.4 Оптимизация циклов	18
1.4.1 Развёртывание циклов.....	19
1.4.2 Разделение циклов.....	20
1.4.3 Слияние циклов	21
Вывод	23
2 Исследование инструментов LLVM для разделения и слияния циклов ..	24
2.1 Проект LLVM	24
2.2 Промежуточное представление LLVM IR	25
2.3 Условия разделения и слияния циклов.....	27
2.4 Разделение циклов	30
2.4.1 Практические эксперименты	32
2.5 Слияние циклов	38
2.5.1 Трансформация слияния циклов.....	39
2.5.1 Практические эксперименты	41
Вывод	47
3 Трансформации с помощью Clang AST.....	49
3.1 Дерево AST	49
3.2 Собственный модуль на основе Clang AST	50
Вывод	52

4 Экономическое обоснование ВКР	54
4.1 Составление плана-графика выполнения работ.....	55
4.2 Расчет расходы на оплату труда исполнителей	55
4.2.1 Расчет основной заработной платы исполнителей.....	56
4.2.2 Расчет дополнительной заработной платы	56
4.2.3 Итоговые расходы на оплату труда.....	57
4.3 Расчет отчислений на социальные нужды	57
4.4 Расчет затрат на сырье и материалы.....	58
4.5 Расчет амортизационных отчислений	59
4.6 Расчет накладных расходов	60
4.7 Расчет совокупной величины затрат	60
Вывод	61
ЗАКЛЮЧЕНИЕ.....	62
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ.....	64
ПРИЛОЖЕНИЕ А – Код модуля Clang.....	67

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

AST – абстрактное синтаксическое дерево, древовидное представление абстрактной синтаксической структуры исходного кода.

Clang – фронтенд LLVM, работающий с языками семейства C.

GCC – GNU Compiler Collection, набор компиляторов.

IR – низкоуровневое промежуточное представление, используемое компиляторами LLVM.

LLVM – Low Level Virtual Machine – это компилятор и набор инструментов для создания компиляторов, представляющие собой программы, преобразующие инструкции в форму, которую может прочитать и выполнить компьютер.

SIMD – single instruction, multiple data, принцип вычислений и наборы операций, которые одновременно применяют одну и ту же инструкцию к двум, четырем или более частям данных.

Бэкенд – компилятор, преобразующий промежуточное представление программы в окончательный машинный код для нужной аппаратной архитектуры и содержащий необходимые инструменты для оптимизации.

Временная локализация – повторное обращение к одним и тем же адресам через небольшие промежутки времени.

ВКР – выпускная квалификационная работа.

Задержка кэша – время доступа к данным внутри кэш-памяти.

Локальность ссылок – свойство программы относительно часто обращаться к одним адресам памяти.

Матчер – выражение-сопоставитель, использующееся для описания определённых узлов Clang AST дерева.

Пропускная способность кэша – количество байт, передаваемых между кэш-памятью и вычислительным ядром процессора за такт.

Пространственная локализация – повторное обращение к адресам, расположенным в памяти близко друг к другу, в течение короткого промежутка времени.

Проход – программа и процесс, осуществляющий анализ преобразования или оптимизацию кода с использованием LLVM.

Фронтенд – компилятор, транслирующий код программ на языке высокого уровня в промежуточное представление и содержащий функции для анализа исходного кода.

Частота попаданий в кэш– отношение числа успешных поисков и чтений данных из кэш-памяти (попаданий кэша) к общему числу обращений.

ВВЕДЕНИЕ

Для некоторых программ производительность является ключевым фактором. Поэтому многие программисты, в том числе во всемирно известных компаниях, исследуют и работают над оптимизациями на совершенно разных уровнях и различными средствами. В частности, описанное в работе улучшение локальности ссылок может дать значительный прирост в производительности. Многие из таких оптимизаций выполняются путём преобразований исходного кода.

Проект LLVM имеет множество инструментов для оптимизации, в том числе и трансформации. Многие из них работают с архитектурно-независимым промежуточным кодом, создаваемым «фронтендом». Проект развивающийся, постоянно обновляется и дорабатывается, что придаёт актуальности изучению его инструментов.

Основное исследование направлено на реализованные в LLVM инструменты для разделения и объединения циклов. Эти трансформации имеют большой потенциал для оптимизации, особенно при работе с большими данными. Но они также имеют свои недостатки, которые при неправильном использовании могут вылиться в падение производительности программы. В LLVM данные трансформации не включены в стандартные конвейеры оптимизации и имеют недоработки. Исследования поможет выяснить особенности работы с разделением и объединением циклов и определить дальнейшее направление их развития.

Целью данной работы является исследование средств трансформации исходного кода на основе инструментов LLVM для разделения и слияния циклов.

Объектом исследования выпускной квалификационной работы являются средства трансформации исходного кода на основе LLVM. Предмет исследования — разделение и слияние циклов с помощью средств трансформации LLVM.

Для достижения поставленной цели были выделены следующие задачи:

- анализ предметной области, включающий анализ оптимизаций на основе понятия локальности ссылок, реализующихся трансформацией кода;
- изучение средств работы с LLVM;
- анализ реализованных модулей для трансформации кода;
- исследование инструментов LLVM для разделения и слияния циклов и выполнение практических экспериментов;
- изучение трансформаций clang и разработка своего модуля;

В первом разделе работы описывается проблема доступа к памяти, разъясняются понятия локальности ссылок и рассматриваются различные оптимизации на их основе с обоснованием эффективности.

Во втором разделе после определения используемых технологий подробно описываются процесс и результаты основного исследования.

В третьем разделе описываются трансформации с помощью clang и представлен собственно разработанный пример.

В четвертом разделе представлен дополнительный раздел по теме «Экономическое обоснование выпускной квалификационной работы».

1 Оптимизации на основе трансформации исходного кода

В данном разделе обзревается различные оптимизации на уровне исходного кода, их польза и недостатки. Среди прочего приведено описание разделения и слияния цикла, на дальнейшее исследование и тестирование которых направлена работа.

1.1 Оптимизации доступа к памяти

Существует множество путей оптимизации кода программ [1] – от сокращения вызовов процедур до улучшения параллелизма [2]. Многие из них перспективны и открыты для исследования и создания инструментов. Одно из таких направлений основано на доступе к памяти и включает множество различных оптимизаций [3].

В иерархической (многоуровневой) организации памяти (рисунок 1.1) поиск и загрузка данных в регистры процессора происходит в первую очередь из кэш-памяти верхнего уровня. Если запись там не найдена, то поиск происходит на следующих уровнях кэша, далее в оперативной памяти, и т. д. В случае, когда запись не найдена в кэш-памяти, происходит её запись в кэш или замещение, если в нём не осталось свободного места.

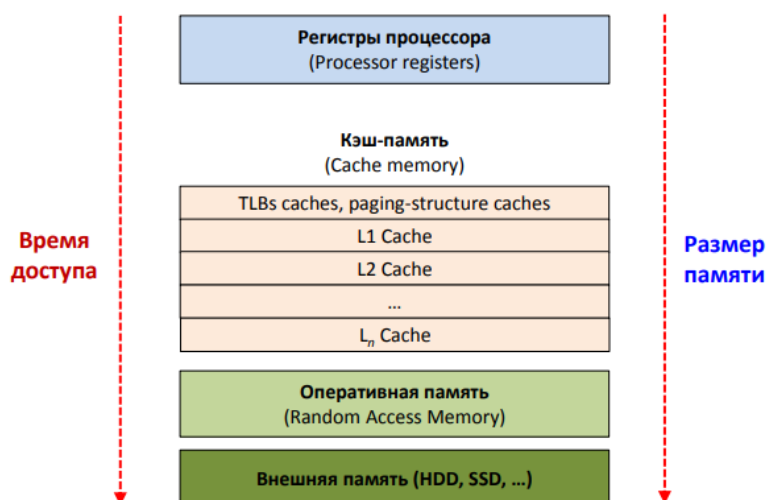


Рисунок 1.1 – Иерархическая организации памяти

Также стоит отметить, что обмен данными между оперативной памятью и кэш-памятью передаются строками фиксированного размера, называемыми кэш линиями, а не отдельными полями.

Существует несколько показателей эффективности кэш-памяти:

Задержка кэша (cache latency) – время доступа к данным внутри кэш-памяти.

Пропускная способность кэша (cache bandwidth) – количество байт, передаваемых между кэш-памятью и вычислительным ядром процессора за такт.

Частота попаданий в кэш (cache hit rate) – отношение числа успешных поисков и чтений данных из кэш-памяти (попаданий кэша) к общему числу обращений.

Эти показатели зависят от структурной организации кэш-памяти [4], алгоритмов записи и замещения записей, и прочего. Но также существует множество оптимизаций самих программ путём трансформации кода, то есть его преобразования, которые увеличивают количество попаданий кэша, и, соответственно, уменьшают число промахов, тем самым сокращая время выполнения программы и повышая производительность. Часто подобные оптимизации основываются на понятии локальности ссылок.

Локальность ссылок (locality of reference) – свойство программы относительно часто обращаться к одним адресам памяти. Существует две её формы (рисунок 1.2):

Пространственная локализация (spatial locality) – повторное обращение к адресам, расположенным в памяти близко друг к другу, в течение короткого промежутка времени.

Временная локализация (temporal locality) – повторное обращение к одним и тем же адресам через небольшие промежутки времени.

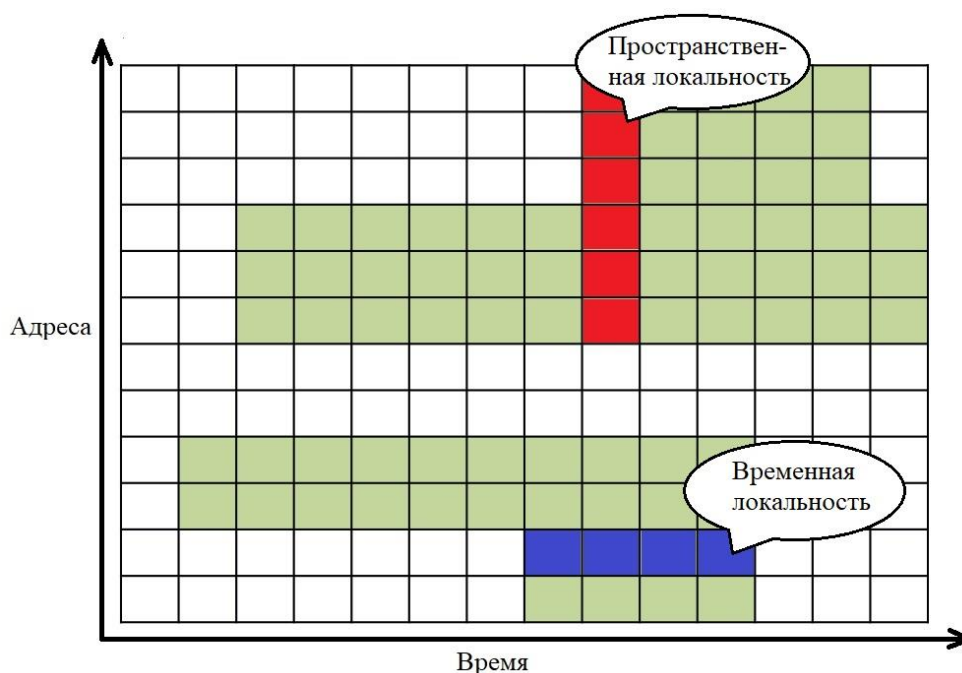


Рисунок 1.2 – Локальность ссылок

Таким образом, локальность ссылок приводит к тому, что происходит частое обращение к одним и тем же или смежным адресам, которые недавно использовались и уже размещены в кэш-памяти. Из-за этого реже возникают промахи кэша и увеличивается производительность программы.

Далее приведены два направления оптимизации, основанные на доступе к памяти.

1.2 Оптимизации компоновки машинного кода

В процессе компиляции компилятор транслирует программу, генерируя машинно-ориентируемый код из исходного. Его инструкции будут закодированы некоторым количеством байт и последовательно расположены в памяти. Это называется компоновкой машинного кода.

Оптимизации компоновки основаны на разделении горячего кода — ветвей кода, которые выполняются относительно часто, и холодного кода — ветвей, которые выполняются значительно редко.

Выгода расположением рядом блоков горячего кода в лучшем использовании локальности ссылок. Так как инструкции загружаются из

оперативной памяти в кэш-память кэш линиями, при таком раскладе есть большая вероятность поместить туда сразу несколько инструкций, которые часто будут использоваться в скором времени [5].

Самым простым примером является оптимизация размещения базовых блоков. Базовый блок – последовательность инструкций с одним входом и выходом, то есть которые обязательно выполняются подряд. На рисунке 1.3 изображён пример, в котором простое инвертирование условия ветвления с веткой холодного кода на противоположное приводит к смежному размещению горячего кода при компоновке.

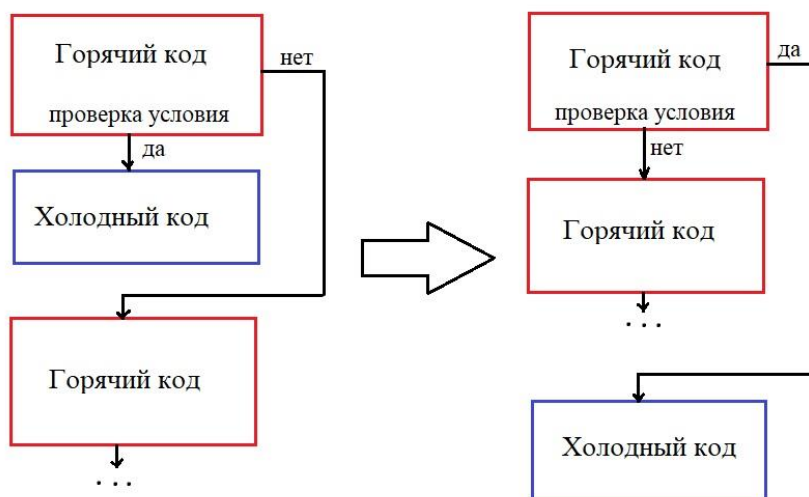


Рисунок 1.3 – Размещение базовых блоков

Но большая часть подобных оптимизаций является скорее работой компоновщика, поэтому в работе на них не заостряется внимание.

1.3 Оптимизация структур

Другим большим направлением является оптимизация структур. Большинство решений этого направления решаются трансформацией кода [3]. Рассмотрим два примера.

1.3.1 Разделение структур

Одна из проблем при работе со структурами заключается в том, что при загрузке нужного поля структуры, в кэш-линию так же загружаются следующие поля, которые могут не использоваться в ближайшее время. Это же относится и к массивам.

Таким образом, эта оптимизация также связана с разделением горячего и холодного кода либо с пространственной локализацией цикла. Её простой пример приведён на рисунке 1.4.

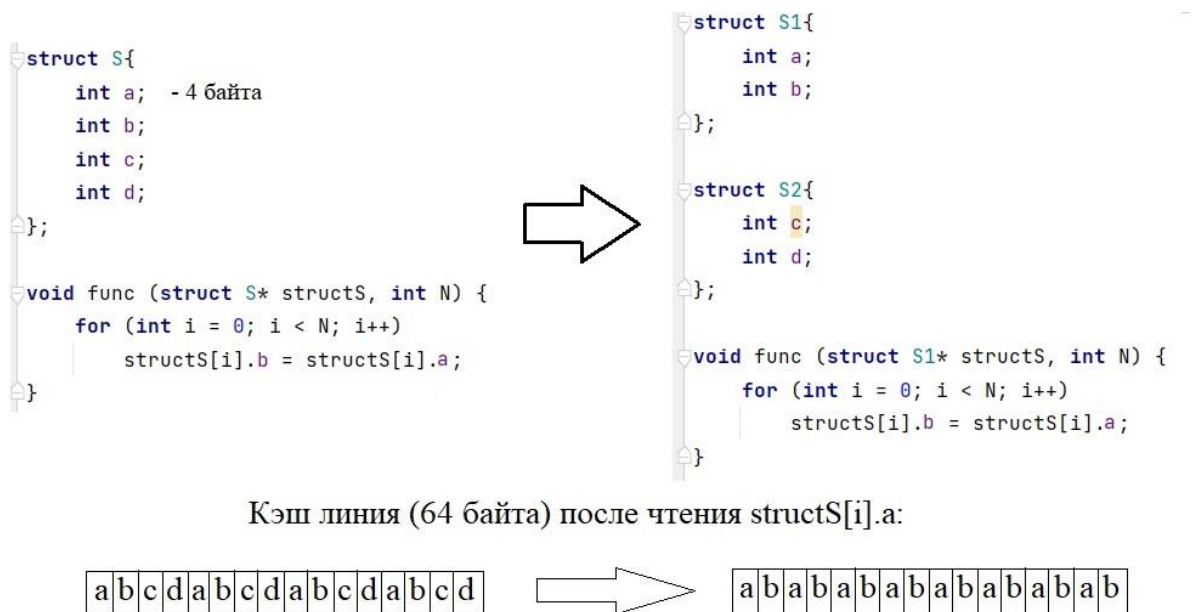


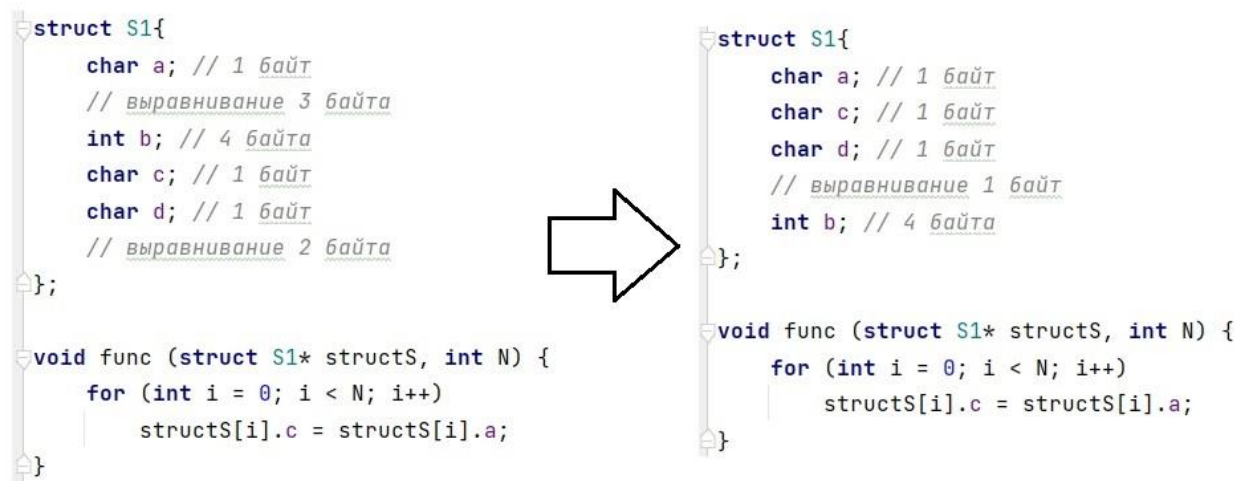
Рисунок 1.4 – Разделение структур

После разделения структуры на две части в цикле функции `func` улучшается использование пространственной локальности ссылок. В линию кэша сразу загружаются в два раза больше полей, которые потребуются в ближайшее время.

1.3.2 Оптимизация выравнивания структур

Другая проблема оптимизации структур – их выравнивание [6]. Дело в том, что компилятор выравнивает экземпляр структуры так, что он становится равен самому длинному её полю. Происходит это для того, чтобы убедиться, что все поля имеют быстрый доступ. Пример такого

выравнивания и оптимизации путём трансформации структуры приведён на рисунке 1.5.



Кэш линия (первые 16 байт) после чтения structS[0].i:

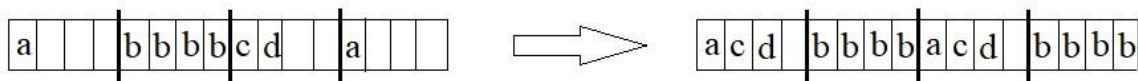


Рисунок 1.5 – Оптимизация выравнивания структур

В результате такой трансформации уменьшится её длина, в том числе и при размещении в кэш-памяти, что повышает её вместительность. В данном примере это приводит к тому, что большее количество полей, необходимых в ближайшее время, будут уже находиться в линии кэша, и количество промахов кэша сократится.

1.4 Оптимизация циклов

Существует также много оптимизаций, направленных на циклы. Их большая часть тоже осуществляется с помощью трансформации исходного кода. Циклы могут выполнять много итераций и занимать относительно долгое время, следовательно их оптимизация может повлиять на скорость выполнения сильнее, чем другие. Это делает данное направление более привлекательным, чем, например, оптимизацию ветвлений [7].

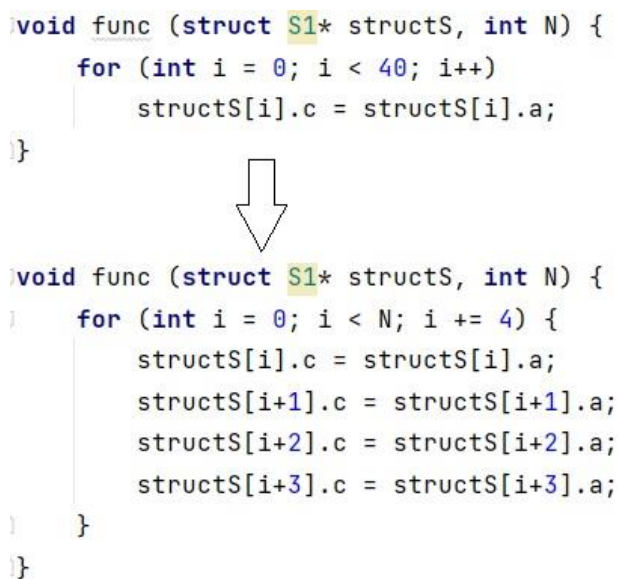
Трансформаций циклов существует довольно много [8]. Некоторые из них распространены, но могут иметь разные интерпретации и улучшения. Для исследования рамках работы были предложены две из них – разделение и

слияние циклов. Они не так распространены, как некоторые другие, и имеют некоторые дополнительные условия и проблемы, что повышает интерес.

Далее представлено описание одной из самых распространённых преобразований циклов, а также двух исследуемых в работе трансформаций.

1.4.1 Развёртывание циклов

Развёртывание циклов (loop unrolling) – хорошо известная трансформация цикла [9]. Изменение заключается в увеличении шага итерации цикла в K раз, где K – коэффициент развёртывания, и соответствующее повторение инструкций в теле цикла для каждой исходной итерации. Таким образом, размер тела цикла в большинстве случаев увеличивается в K раз. Пример развёртывания цикла приведён на рисунке 1.6.



```
void func (struct S1* structS, int N) {  
    for (int i = 0; i < 40; i++)  
        structS[i].c = structS[i].a;  
}  
  
↓  
  
void func (struct S1* structS, int N) {  
    for (int i = 0; i < N; i += 4) {  
        structS[i].c = structS[i].a;  
        structS[i+1].c = structS[i+1].a;  
        structS[i+2].c = structS[i+2].a;  
        structS[i+3].c = structS[i+3].a;  
    }  
}
```

Рисунок 1.6 – Развёртывание цикла

Потенциал увеличения производительности связан с уменьшением накладных расходов на итерации. Также развёртывание цикла может позволить осуществить другие оптимизации цикла или позволить процессору распараллелить тело цикла на уровне инструкций.

Но также это является компромиссом между скоростью и размером кода, и большой коэффициент развёртывания может привести к снижению производительности. Также стоит учитывать, что для такой трансформации

может потребоваться дополнительный код. Во-первых, если число итераций исходного цикла может неизвестно заранее и может меньше, чем коэффициент развёртывания, необходимо включить соответствующую проверку. Во-вторых, если число повторений не кратно коэффициенту, то необходим ещё один цикл до или после основного, который сделает остаточное количество итераций.

Согласно исследованиям [10], при подходящем для такой оптимизации примере увеличение скорости прохождения цикла может достигать практически двух раз.

1.4.2 Разделение циклов

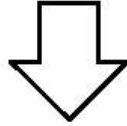
Разделение циклов (loop fission или loop distribution) – это трансформация, который разбивает исходный цикл на несколько производных с одними и теми же диапазонами итераций, каждый из которых включает часть тела исходного цикла [11]. Основная цель такой оптимизации – улучшение локальности ссылок. Если разные производные циклы обращаются к разным полям или массивам, то при выполнении одного этого цикла в линии кэша не помещаются данные, необходимые другому, что и повышает локализацию.

На рисунке 1.7 приведён пример такой оптимизации. При считывании первых элементов массива или элементов, которых нет в кэш-памяти, в линию кэша помимо них записываются следующие ячейки памяти, пока она не заполнится. Т.к. элементы массивов хранятся в памяти последовательно, а точнее построчно, в линию будут помещены следующие значения из того же массива. В примере при выполнении исходного цикла будут загружаться такие строки для каждого из четырёх массивов. В производных циклах же их количество равняется двум. Следовательно, при их выполнении понижается вероятность вытеснения линий из верхних уровней кэш-памяти с ещё нужными данными при записи новых, что повышает производительность.

```

void func (int* a, int* b, int* c, int* d, int k, int N) {
    for (int i = 1; i < N; i++) {
        a[i] = a[i - 1] + b[i - 1];
        b[i] = k * a[i];
        c[i] = d[i-1] + 1;
    }
}

```



```

void func (int* a, int* b, int* c, int* d, int k, int N) {
    for (int i = 1; i < N; i++) {
        a[i] = a[i - 1] + b[i - 1];
        b[i] = k * a[i];
    }

    for (int i = 1; i < N; i++) {
        c[i] = d[i-1] + 1;
    }
}

```

Рисунок 1.7 – Разделение цикла

Стоит также отметить, что в данном примере мы не можем разделить две строки в начале тела исходного цикла, т.к. первая строка имеет зависимость от результата второй на предыдущей итерации. К тому же в качестве разных аргументов в функцию может передаваться один и тот же массив или его части, что может привести к невозможности деления цикла. Например, если бы в третьей строке исходного цикла из примера выше вместо `d[i-1]` было `b[i+1]` и ссылки `b` и `d` указывали на одну и ту же ячейку, то подобное разделение было невозможно. В связи с этим для разделения циклов могут потребоваться дополнительные проверки и хранение обоих вариантов цикла, что может сделать такую оптимизацию неэффективной.

1.4.3 Слияние циклов

Слияние циклов (loop fusion) – трансформация, обратная разделению. Она заключается в объединении двух или более последовательных циклов в

один так, чтобы каждое использование и определение данных в образованном цикле было таким же, как в исходных [9].

Учитывая то, что в прошлом подразделе мы рассматривали обратное преобразование, слияние цикла не всегда оказывает положительный эффект на производительность. Тем не менее, оно позволяет уменьшить накладные расходы на управление циклом и позволить процессору совершить распараллеливание на уровне инструкций подобно развёртыванию. Также слияние получившихся внутренних циклов может быть полезно после развёртывания внешнего. Но, помимо этого, объединение в некоторых примерах улучшает использование локальности ссылок. Такой пример приведён на рисунке 1.8.

```
void func(int* a, int* b, int* c, int k, int N) {  
    for (int i = 1; i < N; i++) {  
        a[i] = a[i - 1] + b[i - 1];  
    }  
  
    for (int i = 1; i < N; i++) {  
        c[i] = k * a[i];  
    }  
}  
  
↓  
  
void func(int* a, int* b, int* c, int k, int N) {  
    for (int i = 1; i < N; i++) {  
        a[i] = a[i - 1] + b[i - 1];  
        c[i] = k * a[i];  
    }  
}
```

Рисунок 1.8 – Слияние циклов

Так как в обоих выражениях тел циклов происходит обращение к $a[i]$, после слияния они выполняются почти подряд, что повышает временную локализацию. Но, как и в случае с объединением циклов, нужно быть осторожным с зависимостями. Если в такую функцию передаётся одна и та же ссылка в аргументы b и c , это нарушает условия и делает преобразование не

допустимым. В сложных программах это также может потребовать дополнительных проверок и хранения обоих вариантов, что даёт дополнительные накладные расходы и может сделать такую трансформацию неэффективной.

Вывод

В данном разделе был произведён анализ предметной области, а именно разбор проблемы доступа к памяти, понятий локальности данных и различных оптимизаций на их основе. К каждой был приведён пример трансформации кода на языке C.

В подразделах 1.4.2 и 1.4.3 были описаны разделение и слияние циклов, которые кроме теоретического увеличения производительности имеют свои проблемы и могут быть вовсе неэффективны. Именно они были выбраны для дальнейшего исследования инструментов трансформации. Таким образом, описанные здесь определения и выводы будут использованы в следующем разделе для объяснения результатов экспериментов.

2 Исследование инструментов LLVM для разделения и слияния циклов

В данном разделе после краткого описания используемых технологий подробно описан процесс исследования трансформаций разделения и слияния циклов с помощью найденных инструментов, включая практические эксперименты и разбор результатов.

2.1 Проект LLVM

Название LLVM [12] изначально было акронимом для «Low Level Virtual Machine», но проект имеет мало общего с другими виртуальными машинами, поэтому от такого значения и теперь LLVM – полное его имя и не является аббревиатурой. Изначальной целью было создание максимально независимого от архитектуры промежуточного представления (intermediate representation, IR), которое можно сравнить с байт-кодом Java, более подробно представленного в подразделе 2.2. Возможность сохранения такого представления на диске – также важная отличительная особенность проекта.

LLVM привычно сравнивают с набором компиляторов GNU Compiler Collection (GCC) [13]. Первое отличие между ними состоит в том, что GCC поддерживает несколько языков программирования, а LLVM не является компилятором ни для одного из языков. Также есть разница в политике лицензирования библиотек. Более либеральное лицензирование LLVM и сложность расширяемости GCC привлекло к себе крупные компании, в том числе Apple [14] и Google [15], и в результате такой поддержки развитие второго начало отставать от первого.

Чтобы понять суть проекта, нужно ознакомиться со следующими частями инфраструктуры:

Фронтенд (frontend) или анализатор исходного кода – компилятор, транслирующий код программ на языке высокого уровня в промежуточное представление и содержащий функции для анализа исходного кода.

Бэкенд (backend) или генератор выполняемого кода – компилятор, преобразующий промежуточное представление программы в окончательный машинный код для нужной аппаратной архитектуры и содержащий необходимые инструменты для оптимизации.

Сначала Крис Латтнер, один из создателей проекта, делал LLVM как бэкенд для GCC, но затем начал работать над новым фронтендом для языков C, C++ и Objective-C, названным clang [16]. Более подробное его описание приведено в разделе 3. Сейчас LLVM – это сборник множества различных подпроектов, фреймворков и инструментов, являющимися лидерами в своей области. Основной подпроект, с которого и начался проект – ядро LLVM, независимый от источника и цели оптимизатор, построенный на основе промежуточного представления IR и работающий на его уровне. Именно его библиотеки исследуются в этой работе.

2.2 Промежуточное представление LLVM IR

Как уже сказано, промежуточное представление – одно из самых важных понятий при работе с LLVM. Оно связывает анализаторы исходного кода, которые и создают IR, и генераторы выполняемого кода и тем самым позволяет проекту анализировать программы на разных языках программирования и генерировать код для разных архитектур. Промежуточное представление может храниться на диске в двух форматах: двоичный формат (.bc) и текстовая сборка (.ll).

Для наглядности создадим текстовую сборку IR без анализа и оптимизаций для простого файла test.c. Для этого можно использовать команду «\$ clang sum.c -emit-llvm -S -c -o sum.ll», где -emit-llvm указывает на то, что нам нужно получить именно промежуточное представление, а не исполняемый файл, а -S – что это должен быть читаемый файл, а не двоичный. Исходный и получившийся файл с пометками, обозначающие соответствие инструкций IR строкам файла C, показаны на рисунке 2.1

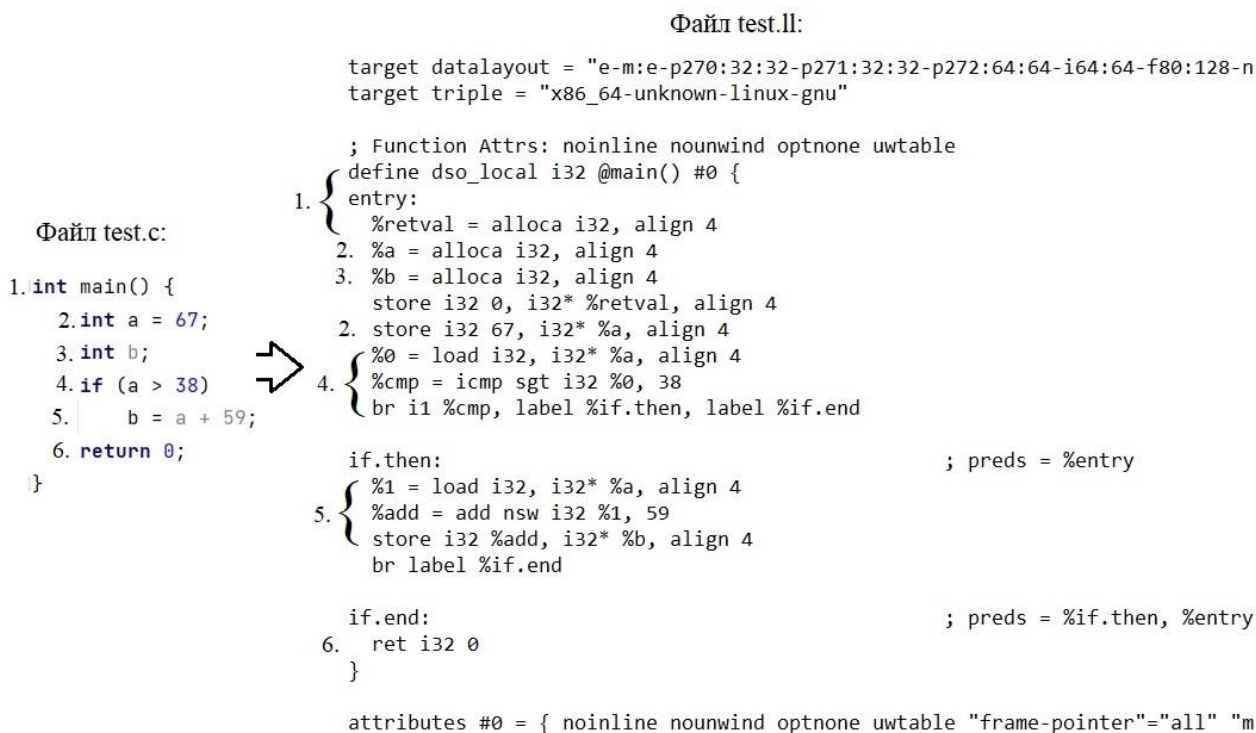


Рисунок 2.1 – Пример промежуточного представления

Один такой файл представляет собой так называемый модуль. Модуль содержит последовательность функций, которые содержат группы базовых блоков, в свою очередь содержащие инструкции. Кроме этого, в модуль включены дополнительные объекты поддержки. Например, на рисунке 2.1 конструкции `target datalayout` и `target triple` содержат информацию для определения целевого формата данных, а `attributes #0` – множество атрибутов 0-й функции.

Модуль на примере содержит всего одну функцию `main`, которую легко идентифицировать в IR. Она содержит три базовых блока, явно разделённых пустой строкой. Базовый блок – это набор последовательных инструкций, не разделённых ветвлениями и выполняющихся друг за другом в любом случае. Таким образом, условный оператор в исходном коде разделил IR на три базовых блока, два из которых и имеют имя «`if.then`» и «`if.end`». Если один базовый блок ведёт в другой, то последняя его инструкция, называемая терминатором, указывает на этот переход. В данном случае первый базовый блок в зависимости от выполнения условия ветвления может вести в два

разных базовых блока, поэтому включает проверку и оба перехода. По соответствию инструкций строкам кода несложно определить их значение. Например, `alloca` выделяет память в стеке, `store` перезаписывает переменную, `load` считывает ячейку стека, а `add nsw` совершает операцию сложения. Фактически инструкции `store` и `load` являются избыточными и убираются при компиляции со стандартным флагом оптимизации `-O1` [17].

Преобразования, оптимизации и анализ над промежуточными представлениями выполняются с помощью так называемых проходов, которые по очереди перебирают нужные им части (модули, функции, циклы, и др.) и совершают необходимые действия. Регистрация и планирование проходов, а также регистрация зависимостей между ними выполняются с помощью диспетчера проходов. Запустить их с выводом получившегося файла IR можно с помощью команды `opt`, которая описывается как модульный оптимизатор и анализатор LLVM. Также LLVM API позволяет регистрировать собственные проходы, что является очередной полезной особенностью проекта [18].

2.3 Условия разделения и слияния циклов

При поиске реализации нужных трансформаций в исходниках LLVM были обнаружены уже готовые проходы разделения и слияния циклов, представленные файлами `LoopDistribute.cpp` и `LoopFuse.cpp` соответственно. Подробно каждый из них описан в подразделах 2.4 и 2.5, здесь же описаны общие нюансы при работе с ними, связанные с требованиями для запуска этих проходов. Эти условия также необходимы для большинства других преобразований циклов [19][20].

Во-первых, оба прохода работают только со стилем кода, которое называют статическое одиночное присваивание (*static single assignment, SSA*). Форма SSA требует, чтобы каждая переменная присваивалась ровно один раз, и позволяет по минимуму использовать операции загрузки или сохранения в

память и использовать вместо этого, когда это возможно, регистры. Такая форма упрощает большое число оптимизаций компилятора. Поддерживают её инструкции Фи. Когда в коде присутствуют ветвления или другие непредсказуемые переходы, в них для одной и той же переменной исходного кода могут использоваться разные регистры. И после соединения веток при использовании этой переменной соответствующий ей регистр зависит от потока управления, который заранее неизвестен. Здесь и приходят на помощь инструкции Фи, которые вставляют регистр в зависимости от предыдущего выполняемого базового блока. Кроме поддержки формы SSA, функции Фи помогают проанализировать зависимости между базовыми блоками. Перевести промежуточное представление в такую форму позволяет проход - mem2reg, который переводит ссылки на память в ссылки на временные регистры, попутно выполняя очевидные оптимизации. Пример IR после такого прохода с одной инструкцией Фи на выходе показан на рисунке 2.2.

```

; Function Attrs: noline nounwind uwtable
define dso_local i32 @main() #0 {
entry:
    %cmp = icmp sgt i32 67, 38
    br i1 %cmp, label %if.then, label %if.end

if.then:                                ; preds = %entry
    %add = add nsw i32 67, 59
    br label %if.end

if.end:                                  ; preds = %if.then, %entry
    %b.0 = phi i32 [ %add, %if.then ], [ 81, %entry ]
    %sub = sub nsw i32 %b.0, 4
    ret i32 0
}

```

int main() {
 int a = 67;
 int b = 81;
 if (a > 38)
 b = a + 59;
 int c = b - 4;
 return 0;
}

Рисунок 2.2 – Форма SSA

Во-вторых, циклы должны иметь каноническую (simplify) форму. Это означает, что у каждого из них есть предзаголовок, то есть единственный блок, предшествующий заголовку цикла; каждый цикл имеет единственный переход из конца к заголовку (backedge) и защёлку (latch), из которого осуществляется переход; а также собственный выходной блок, который не имеет предшественников вне цикла. Последнее означает, что цикл доминирует над выходным блоком, то есть второй может выполняться только после выполнения первого. Схема типичного варианта каноничной формы цикла

изображена на рисунке 2.3. Чтобы выполнить это условие, нужно выполнить проход -loop-simplify.

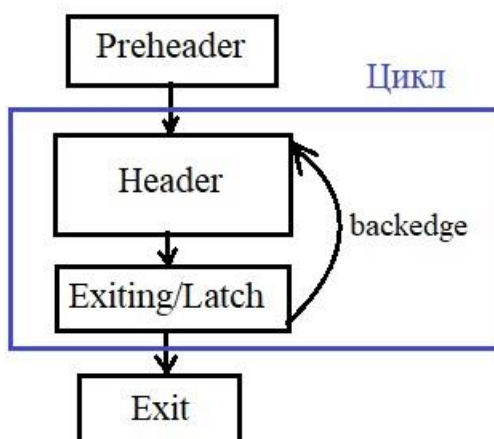


Рисунок 2.3 – Схема канонической формы цикла

В-третьих, циклы должны быть повёрнутым, то есть с постусловием (стиля do/while). При этом если цикл защищённый, то есть может при определённых условиях не выполнить ни одной итерации, перед ним добавляется блок, который позволяет обойти цикл при соответствующем результате проверки. Преобразование в такую форму осуществляется с помощью прохода -loop-rotate.

В-четвёртых, обе трансформации сильно зависят от управления циклом, поэтому им требуется анализ индуктивных переменных, который предоставляет проход -indvars. Также, согласно документации, он попутно, если это возможно, приводит индуктивные переменные к канонической форме, которая имеет следующие характеристики:

- цикл имеет одну каноническую переменную индукции, которая начинает счёт с нуля и делает шаг по единице;
- переменная индукции будет первой инструкцией Фи в блоке заголовка цикла;
- любые арифметические операции с указателями преобразуются в использования индексов массива.

Но, к сожалению, на практике первое и третье преобразование после проходов ни разу не было мной замечено, но без предоставленных этим

анализатором данных работа с проходами разделения и слияния цикла невозможна.

Также обе трансформации используют автоматически выполняющиеся анализаторы, например Loop Info, предоставляющий информацию о важных блоках цикла, описанных ранее. Попробуем проделать указанные выше четыре прохода над простой функцией на языке C с одним циклом. Для этого сначала сгенерируем файл сборки командой «\$ clang -S -emit-llvm -O0 -Xclang -disable-O0-optnone hello.c», после чего запустим нужные проходы командой «opt -mem2reg -indvars -loop-simplify -loop-rotate hello.ll -S -o out.ll». Исходный текст hello.c и результат out.ll представлены на рисунке 2.4.

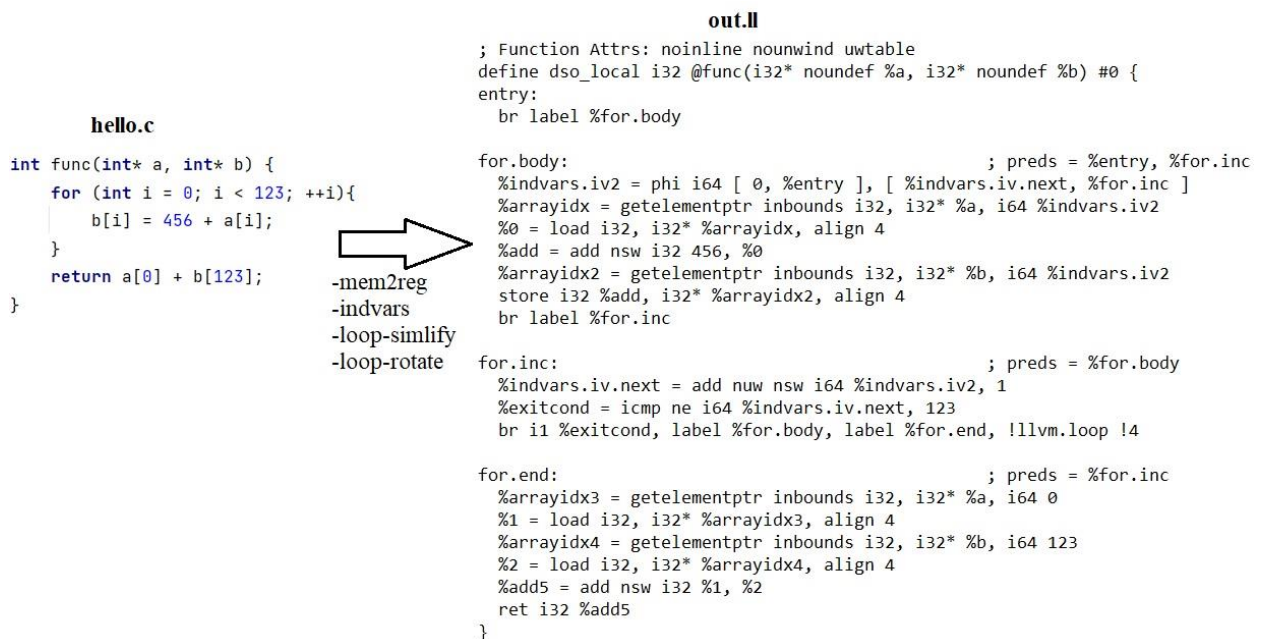


Рисунок 2.4 – Пример работы предварительных проходов

Далее перейдём к индивидуальному разбору трансформаций и практическим экспериментам.

2.4 Разделение циклов

Для начала стоит отметить, что проход разделения циклов (-loop-distribute) выделяет только «наиболее внутренние» циклы и идёт по ним. При переборе циклов для каждого выполняется проверка условий, по большей части описанных во втором пункте. Далее просматривается каждая

инструкция, связанная с памятью, и тело цикла разбивается на последовательный разделы. При этом рассматриваются зависимости между инструкциями. Те из них, которые имеют небезопасные зависимости или «охватываются» ими, не могут быть разбиты. Инструкции, не имеющие таких зависимостей, помещаются в другие отдельные разделы, которые затем при необходимости объединяются. Основная работа самого разбития цикла на получившиеся в итоге части происходит с помощью функции `cloneLoopWithPreheader` другого класса `CloneFunction.cpp`.

Также для работы с разделением циклов стоит отметить очередные детали. В пункте 1.4.2 упоминались, что бывают случаи, когда в качестве разных аргументов в функцию может передаваться один и тот же массив или его части, что может привести к невозможности деления цикла и для чего могут понадобиться проверки. Здесь эта ответственность передана анализатору `Scalar Evolution`, который создаёт для цикла предикат с «run-time» проверками. Как и было сказано, к IR после этого хранится как исходный цикл, так и производные, и терминатор базового блока предиката может осуществить переход на один из вариантов. Проверок может быть довольно много, что может сделать трансформацию неэффективной. Поэтому проход имеет специальный флаг `loop-distribute-scev-check-threshold`, в котором можно указать их максимальное количество. Также если пользователь уверен, что указатели не будут пересекаться, он может добавить к аргументам функции указывающее на это ключевое слово `restrict`. На практике проверено, что в таком случае предикат с проверками не создаётся.

Проход `-loop-distribute` включен в стандартные конвейеры проходов, но разделение циклов выключено по умолчанию, так как эффективность этой трансформации не доказано. Чтобы его включить, можно добавить к команде флаг `-enable-loop-distribute`, либо в исходном файле локально над каждым циклом указать директиву «`#pragma clang loop distribute(enable)`».

Также разделение циклов применяется, в основном, не для собственного повышения производительности, а для упрощения или возможности последующей векторизации. Об этом говорит ряд проверок, связанных с векторизацией, в файле прохода разделения. Если векторизация памяти уже доступна, то цикл разбит не будет и выведется соответствующее сообщение.

Векторизация циклов – оптимизация, похожая на развёртывание из пункта 1.4.1, но направлена на улучшение использования SIMD (single instruction, multiple data – одиночный поток команд, множественный поток данных) – принцип вычислений и наборы операций, которые одновременно применяют одну и ту же инструкцию к двум, четырем или более частям данных [21]. То есть, в отличие от развёртывания, сами инструкции как бы не разделяются. Большинство современных компиляторов могут автоматически векторизовать простые циклы, но проход `-loop-vectorize` анализирует также и более сложные. Как и у разделения, в исходном коде над циклом можно указать директиву «`#pragma clang loop vectorize(enable)`» при выключенной векторизации [22].

Таким образом, связка разделения и векторизации позволяет добиться большей эффективности. Тем не менее, `loop-distribute` отключен по умолчанию в стандартных проходах, так как, по объяснению разработчиков, эта комбинация работает редко, хоть и позволяет достичь большого роста производительности в подходящих примерах.

2.4.1 Практические эксперименты

Перейдём к тестированию и временным измерениям. Так как проход не рассматривает циклы в которые вложены другие циклы, что подтверждено на практике, в тестах нет смысла строить сложных программ. Поэтому для наглядности код будет небольшим, с одним циклом, в котором сосредоточена нагрузка. Так для измерения времени можно будет использовать простейший

внешний метод – утилиту time. Для сравнения к каждому тесту будет сгенерировано три IR с разными проходами:

- со всеми проходами, которые необходимы для разделения цикла, но без самого прохода loop-distribute;
- с проходами из предыдущего пункта и разделением цикла (loop-distribute);
- с проходами из предыдущего прохода и векторизацией (loop-vectorize).

В процессе тестирования был выявлен ряд примеров, в которых разделение циклов могло бы улучшить локальность ссылок, но трансформация не была выполнена, так как либо в этом нет необходимости для векторизации, либо найдены опасные зависимости, хотя не всегда они на самом деле есть. Эти примеры в отчёт включены не были, но мнение разработчиков о том, что на данном этапе проход пропускает большинство циклов, подтверждено. В результате изначальный круг тестов сузился до трёх. На рисунке 2.5 представлены исходные файлы для тестирования.

Пример 1	Пример 2	Пример 3
<pre> struct S{ int a, b; }; struct L{ int c, d, e; }; int main(){ struct S ab[100]; struct L cde[100]; int sum [100]; cde[0].c = 1; for(int i = 0; i < 99; i++) { ab[i].a = i; ab[i].b = (i%10); // разделение 1 cde[i+1].c = cde[i].c*(i/2); // разделение 2 cde[i].d = (cde[i].c+i)*i; cde[i].e = cde[i].d/2; sum[i] = ab[i].a + ab[i].b; } return 0; } </pre>	<pre> int main(){ int A[1000],B[1000],C[1000], D[1000],E[1000],F[1000]; int k = 0; A[0] = 1; E[0] = 1; for(int i = 0; i < 999; i++) { D[i] = i%10; E[i] = E[i] * i; B[i] = i; // разделение 1 A[i + 1] = A[i] + B[i]; // разделение 2 k += A[i+1]; C[i] = D[i] * E[i]; F[i] = C[i] - k; } return 0; } </pre>	<pre> int main(){ int A[1000],B[1000],C[1000]; int k = 0; int n = 0; for(int i = 0; i < 999; i++) { A[i] = i%10; A[i+1] = A[i] * i; // разделение k += i; B[i] = k; C[i] = B[i]/i; n += C[i]; } return n; } </pre>

Рисунок 2.5 – Исходные файлы для проведения тестов

Хоть для тестирования планировалось использовать больше примеров, разделить цикл получилось только на трёх. На рисунке в теле цикла комментариями разделены строки так, как разбил его проход. Разберём подробнее каждый пример.

Первый пример самый сложный, здесь цикл работает с массивом структур. В теории для улучшения локальности ссылок нужно сделать два разделения: после второй строки тела и перед последней. Первое разбиение проход сделал верно, но с целью сделать доступным для векторизации первый производный цикл. Последняя строка от остального массива не отделилась. Таким образом, с точки зрения локальности ссылок мы имеем спорную ситуацию: улучшающее первое разделение и ухудшающее последнее. Такое разбиение произошло потому, что единственная строка второго производного массива не пригодна для векторизации, т.к. вычисление на текущей итерации зависит от предыдущей. Но проход векторизации не разделил и третий цикл, т.к. посчитал это невыгодным.

Во втором примере цикл работает с простыми целочисленными массивами. Для локализации ссылок можно было бы обособить первую, вторую и последние две строки. А совершённые разделения этому как раз противоречат. Сделаны же они по тому же принципу, что в предыдущем примере, отделяя строку, которую невозможно векторизовать. Тем не менее, векторизован был тоже только первый производный цикл.

В третьем примере планировалось максимально упростить как векторизацию, так и разделение для улучшения локальности ссылок. Изначально во второй строке тела вместо $A[i+1]$ было $A[i]$. Но цикл не разделился, так как векторизация уже была возможна. Поэтому пришлось добавить строку, которая не векторизуется. В отличие от других тестов, цикл разделился всего на две части. Это произошло потому, вычисление первой строки зависит от второй на предыдущей итерации, и разбивать их нельзя. Хоть для локальности ссылок цикл и разбит идеально, целью была снова

возможность векторизации. Но, к сожалению, проход векторизации пропустил и второй производный цикл из-за функции Фи, появившейся из-за переменной k, что делает этот тест неполноценным.

На основании этого можно сделать некоторые выводы о работе прохода. Его основная цель – разделить цикл так, чтобы отделить инструкции в теле, которые мешают векторизации, при этом принимая во внимание зависимости памяти. В примерах такая инструкция всего одна, но на практике проверено, что это не предел. При этом затраты на накладные расходы при разделении не рассчитываются, а локальность ссылок не учитывается вовсе.

Для определения изменения производительности после трансформации получившиеся промежуточные представления были переведены в машинный код, а затем в исполняемые файлы с помощью команд «\$ llc test.ll -o test.s» и «\$ clang test.s -o test». Утилитой time было измерено среднее время его выполнения, при этом процессу было задано сходство только с одним процессором. Чтобы создать условия, где локальность ссылок имеет достаточно большое влияние, число итераций циклов было увеличено до 1 000 000. Для этого массивы были сделаны динамическими. Также для более удобного и дополнительно усреднённого результата основная часть программы дополнительно будет повторяться 1 000 раз. В итоге погрешность измерений получилась небольшой. По результатам тестирования была составлена сводная гистограмма (рисунок 2.6).

Все три теста показали схожую ситуацию. Разделение цикла значительно увеличило время выполнения. Векторизация же, где она была, увеличила производительность, но недостаточно даже для того, чтобы покрыть замедление в результате предыдущего прохода. Также немного удивило, что наибольший рост времени наблюдается в первом примере – порядка 82%. Но два разделения в нём создают спорную ситуацию, которую не так просто просчитать. В третьем примере разделение было всего одно, при этом улучшающее локальность ссылок, из-за чего падение

производительности самое незначительное. Возможно, в подобном примере с возможностью векторизации в одном из производных циклов, проход `-loop-vectorize` мог бы увеличить скорость даже относительно первого значения. Но стоит также отметить, что ни в одном из этих примеров не требовались дополнительные проверки, которые дополнительно увеличили бы расходы.

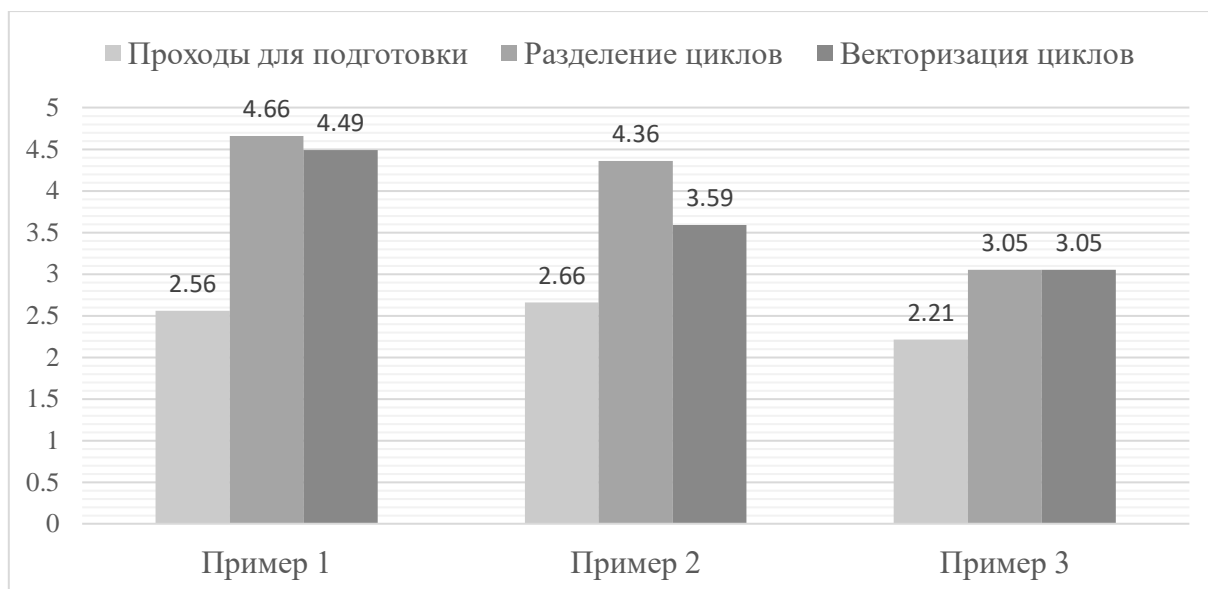


Рисунок 2.6 – Результаты тестирования

Таким образом, данное тестирование поставило вопрос, имеет ли этот проход какую-то пользу. Для его разрешения был создан пример, в котором не векторизуемая строка делит цикл так, что локальность ссылок очевидно улучшается. Его исходный код с пометкой места разделения приведён на рисунке 2.7.

```
for (int j = 0; j < 1000; j++) {
    A[0] = 1;
    for(int i = 0; i < 999999; i++) {
        B[i] = 1000 - i;
        A[i] = B[i] * i;
        A[i + 1] = A[i] + i;
        // разделение
        C[i] = D[i] * E[i];
        D[i] = D[i]/2;
    }
}
```

Рисунок 2.7 – Исходный код для итогового теста

Разделение произошло как ожидалось, при этом вторая часть цикла была успешно векторизована следующим проходом. Измерения времени проводились тем же способом, что и в предыдущих тестах. Его результат в виде гистограммы можно увидеть на рисунке 2.8.

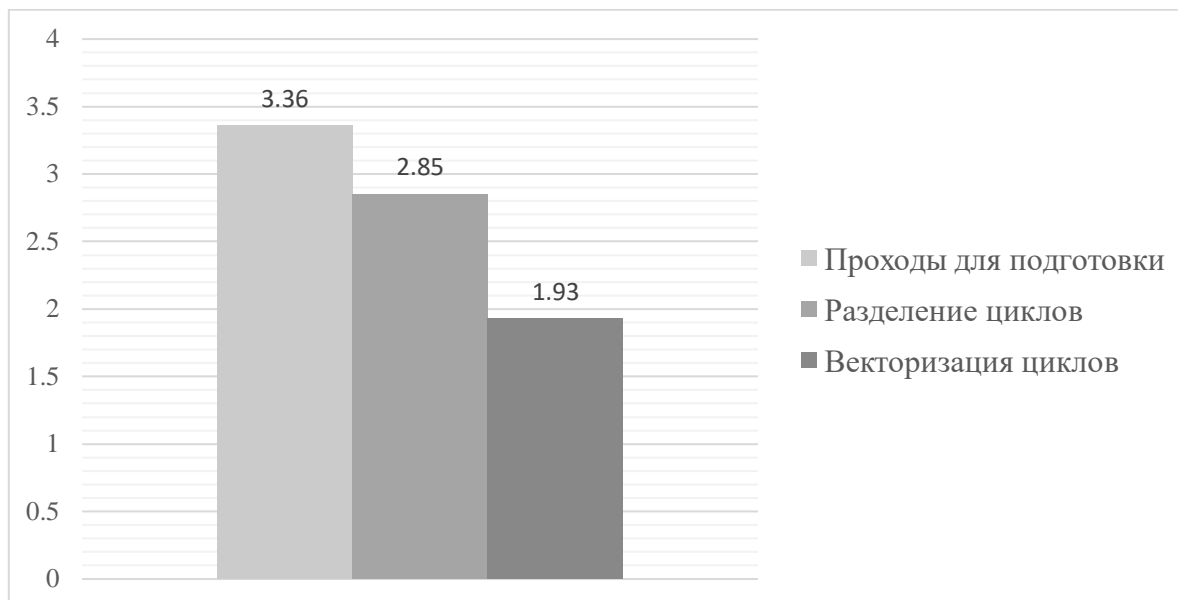


Рисунок 2.8 – Результат итогового теста

Результат получился совершенно другим. Здесь проход разделения циклов дал положительный эффект на производительность – увеличил скорость выполнения примерно на 15%. А вместе с векторизацией время снизилось чуть больше, чем на 57%. Это хороший результат, особенно с учётом предыдущих. Такое снижение показывает, что могут существовать как менее очевидные примеры, в которых улучшение использования локальности ссылок влияет сильнее, чем накладные расходы на управление ссылок, так и более удачные, в которых скорость увеличивается ещё больше.

Тем не менее, в целом оценка полученных результатов подтверждает мнение разработчиков: хоть и существуют программы, для которых данный проход может значительно повысить производительность, они довольно редки. Для большинства циклов разделение не выполняется вовсе, а для другой части далеко не всегда результат имеет положительный эффект. Решение не включать этот проход в стандартные конвейеры понятен и логичен. Но разработка для алгоритма разделения циклов модели, с помощью

которой при анализе можно будет также оценивать изменение локальности ссылок, имеет большие перспективы.

2.5 Слияние циклов

Слияние циклов (loop-fusion) делает проход по функциям и получает нужную информацию от ряда других анализаторов. Далее с помощью полученного от одного из них, Loop Info, программа получает так называемое дерево глубины циклов, содержащее наборы циклов на одном уровне, после чего перебирает их по очереди, начиная с самого внешнего уровня, и, если в таком наборе более одного цикла, помещает их в коллекцию кандидатов на слияние и пытается объединить. Для этого кандидаты должны удовлетворять следующим условиям:

- циклы должны быть смежными, между ними не должно операций;
- иметь одинаковое количество итераций;
- быть эквивалентными потоку управления, то есть второй цикл гарантированно выполняется при выполнении первого, что называется доминированием;
- у защищённых циклов также должны быть эквивалентные условия защиты.;
- не должно быть негативных зависимостей расстояния между циклами.

При этом второе условие может нарушаться при указании максимальной разницы между числом итераций флагом `loop-fusion-peel-max-count`. Для ликвидации этой разницы выполняется функция `peelFusionCandidate`, которая выносит часть итераций цикла с большим диапазоном счётчика выносятся за его тело. Также среди проверок есть вызов функции `isBeneficialFusion`, которая должна проверять прибыльность такой трансформации, но на данный момент она не реализована и всегда возвращает `true`. Это означает, что проход объединяет все циклы, которые возможно.

2.5.1 Трансформация слияния циклов

После успешного прохождения проверок выполняется сама трансформация. Для полного понимания процесса описание будет подкреплено картинками, на которых показаны строки кода, выполняющие трансформацию, и схематическое изображение изменений на примере типичной структуры циклов. Все операции в функции сопровождаются соответствующими обновлениями в анализаторе DominatorTree, которые будут опущены в этом описании. Большинство трансформаций осуществляется на уровне базовых блоков, функциями класса BasicBlock. Итак, порядок операций над циклами:

1. Инструкции и функции Фи перемещаются из предзаголовка второго цикла (которой также является выходным блоком первого) в конец предзаголовка первого. Аналогичное перемещение происходит с узлами Фи. После чего инструкции Фи из защёлки первого цикла переносятся в защёлку второго (рисунок 2.9).

```
moveInstructionsToTheEnd(*FC1.Preheader, *FC0.Preheader, DT, PDT, DI);  
FC1.Preheader->replaceSuccessorsPhiUsesWith(FC0.Preheader);  
FC0.Latch->replaceSuccessorsPhiUsesWith(FC1.Latch);
```

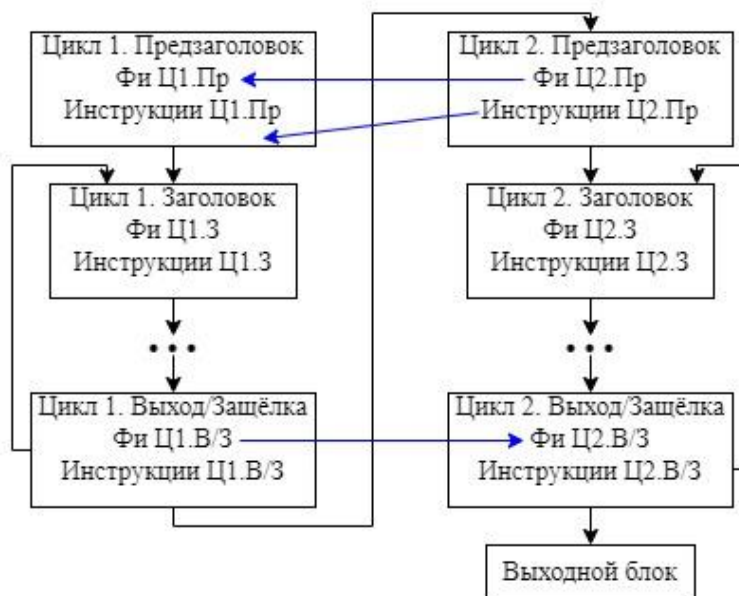


Рисунок 2.9 – Первый шаг трансформации слияния

2. Удаляется переход из выхода первого цикла в предзаголовок второго цикла, который затем удаляется. Если часть итераций первого цикла была вынесена функцией `peelFusionCandidate`, то также удаляется сгенерированный при этом выходной блок (рисунок 2.10).

```
if (!FC0.Peeled) {
    FC0.ExitingBlock->getTerminator()->replaceUsesOfWith(FC1.Preheader,
                                                         FC1.Header);
} else {
    FC0.ExitingBlock->getTerminator()->replaceUsesOfWith(FC0.ExitBlock,
                                                         FC1.Header);
    FC0.ExitBlock->getTerminator()->eraseFromParent();
}
FC1.Preheader->getTerminator()->eraseFromParent();
```

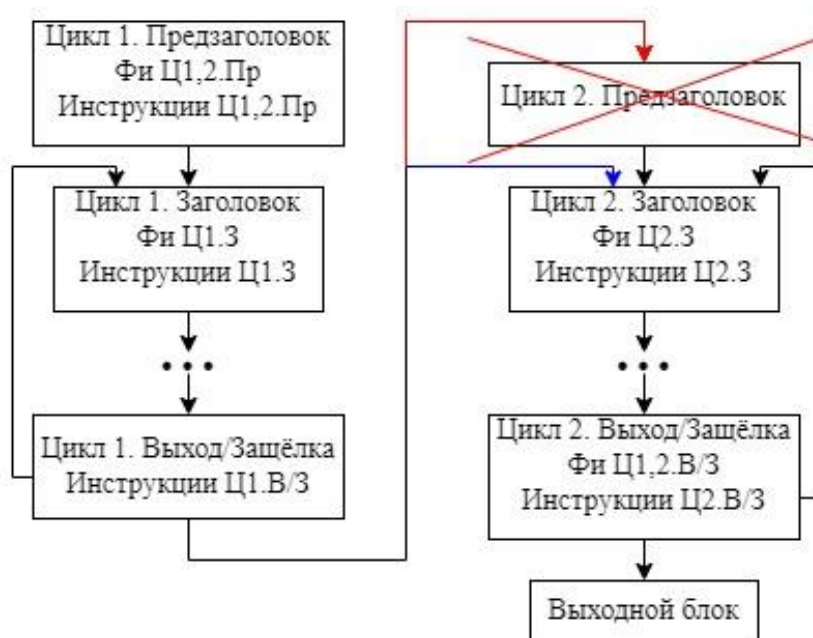


Рисунок 2.10 – Второй шаг трансформации слияния

3. Узлы Фи из заголовка второго цикла также перемещаются в заголовок первого.

4. Защёлка первого цикла меняет переход в собственный заголовок на заголовок второго цикла. Так как её терминатор теперь имеет два одинаковых перехода, они объединяются. Над защёлкой второго цикла осуществляется противоположная операция (рисунок 2.11).


```

FC0.Latch->getTerminator()->replaceUsesOfWith(FC0.Header, FC1.Header);
FC1.Latch->getTerminator()->replaceUsesOfWith(FC1.Header, FC0.Header);
simplifyLatchBranch(FC0);

```

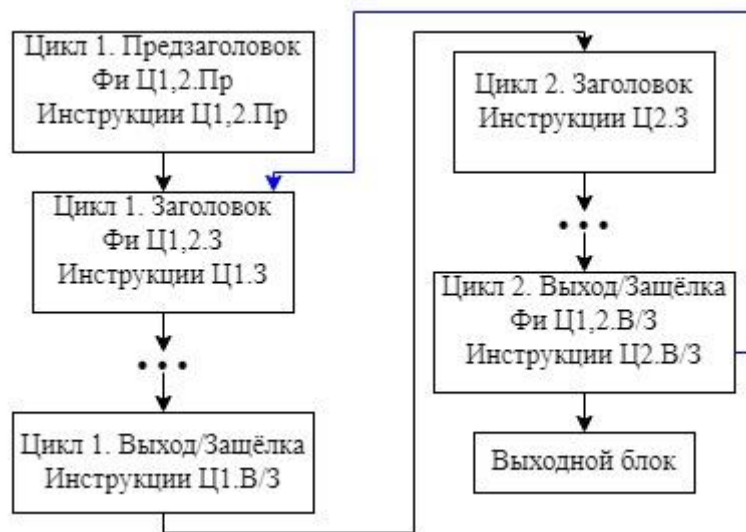


Рисунок 2.11 – Четвёртый шаг трансформации слияния

5. После этого все блоки второго цикла переносятся в первый, после чего пустой второй удаляется.

2.5.1 Проверка функционала на практике

Перейдём к практическим экспериментам. Для начала следует на практике проверить функционал прохода. Для этого создан довольно сложный пример с вложенными циклами, где на обоих уровнях возможно слияние. При этом есть случаи с операцией между циклами, разными названиями и границами итераторов и зависимостями, при которых объединение циклов невозможно. Исходный код приведён на рисунке 2.12. Для удобства каждый цикл в комментарии обозначен своим номером.

Первый список кандидатов на объединение состоит из всех внешних циклов. Между первым двумя были обнаружены зависимости, не позволяющие объединять их, и это действительно так. Внешние циклы 2 и 3 также не были соединены. При отладке была написана причина – не идентичные счётчики цикла. На практике было замечено, что проход, выполняющий приведение итератора к канонической форме, использует для

него название `indvars` в промежуточном представлении. Для цикла 3 счётчик переименован не был, поэтому это несоответствие можно сослать на анализирующий проход `-indvars`. Соответственно, аналогичное сообщение было получено и при попытке присоединить четвёртый цикл.

```
int main(){
    long A[1000], B[1000], C[1000], n = 0;
    for (int i = 0; i < 999; i++){ //1
        A[i] = 0; B[i] = 0; C[i] = 0; }
    for (int i = 0; i < 999; i++){ //2
        for (int j = 0; j < 999; j++){ //2.1
            A[j] += i - j; }
        for (int j = 0; j < 999; j++){ //2.2
            B[j] += j%i; }
        for (int j = 0; j < 999; j++){ //2.3
            C[i] += j/i; }
        }
    for (int i = 0; i < 999; i++){ //3
        for (int j = 0; j < 999; j++){ //3.1
            C[j] = C[j] - B[j]; }
        for (int j = 1; j < 1000; j++){ //3.2
            A[j] = A[j-1] - j; }
        }
    for (int l = 0; l < 999; l++){ //4
        for (int j = 0; j < 999; j++){ //4.1
            B[j] = B[j-1] / 2; }
        for (int j = 0; j < 999; j++){ //4.2
            B[j] += A[l]; }
        n = B[l];
        for (int j = 0; j < 999; j++){ //4.3
            n += C[j]; }
        }
    return 0;
}
```

Рисунок 2.12 – Исходный тест для теста функционала

Также было замечено, что проверяются условия объединения каждой пары циклов, даже не располагающиеся рядом, что влечёт лишние действия. Это обусловлено алгоритмом перебора, не проверяющим, были ли объединены предыдущие циклы. Но при проверке на последовательное расположение циклов такие соединения отбрасываются, так что единственное

последствие – увеличение времени компиляции. В целом, из-за ошибки прохода анализа итераторов объединение внешних циклов не оправдало никаких ожиданий.

Перейдём к вложенным циклам. Первые кандидаты на объединение – три внутренних цикла 2.1 – 2.3. Первые два из них успешно объединились, а вот у третьего снова возникла проблема со счётчиком.

В следующем наборе необоснованно были сразу четыре кандидата – все подциклы второго и третьего внешнего цикла. Напомню, что их наборы также предоставляет другой анализатор LoopInfo. Но проверка на последовательное расположение циклов отсеивает такие случаи, так что единственная проблема здесь – увеличенное время компиляции. Подциклы второго цикла же успешно объединились несмотря на отличие в стартовой позиции итератора. Это означает, что проход -indvars успешно преобразовал управление цикла, чтобы изначальное значение счётчика было равно нулю.

Последний набор включает все циклы на глубине 2. Нет смысла рассматривать уже пройденные, перейдём к кандидатам из четвёртого внешнего цикла. Как и ожидалось, объединение первых двух не произошло в связи с зависимостью памяти, а последних – из-за небезопасной инструкции в предзаголовке цикла 4.3. Итоговую статистику прохода можно увидеть на рисунке 2.13.

```
2 loop-fusion      - Loops fused
4 loop-fusion      - Dependencies prevent fusion
12 loop-fusion     - Loops are not adjacent
1 loop-fusion      - Loop has a non-empty preheader with instructions that cannot be moved
12 loop-fusion     - Loop trip counts are not the same
12 loop-fusion     - Number of candidates for loop fusion
```

Рисунок 2.13 – Статистика прохода

Из-за ошибок прохода -indvars на этом примере не удалось показать, что различные названия переменной индукции никак не влияют на работу модулей, но на других тестах подтверждено, что это так. Также в примере ни разу не получилось объединить более двух циклов, но отдельно тестировался код с девятью, и все они были объединены в один.

В итоге проход работает так, как описано при анализе кода, за некоторыми исключениями. Первое – пропуск некоторых циклов проходом анализа итераторов по непонятным причинам. Второе – лишние проверки возможности объединения кандидатов, очевидно не являющимися соседними. При этом второе исключение не влияет на результат работы, только на время компиляции.

2.5.2 Эксперименты для оценки эффективности

Теперь перейдём к тестам с измерением времени выполнения программы, которые позволят оценить эффективность трансформации объединения циклов. Для полной картины нужно рассмотреть изменение производительности в разных ситуациях. Чтобы верно понять результаты, исходные программы должны быть простыми и состоять из двух циклов, которые возможно объединить. С учётом этих требований были написаны примеры, представленные на рисунке 2.14.

Разберём каждый из них. В первом примере циклы работают всего с несколькими ячейками памяти. Соответственно, единственное, что должно повлиять на производительность при трансформации – уменьшение накладных расходов на управление циклом, что и ожидается измерить опытным путём с помощью этого теста.

Разберём каждый из них. В первом примере циклы работают всего с несколькими ячейками памяти. Соответственно, единственное, что должно повлиять на производительность при трансформации – уменьшение накладных расходов на управление циклом, что и ожидается измерить опытным путём с помощью этого теста.

<p>Пример 1</p> <pre> for (int i = 0; i < 1000000; i++) { A[i%4] = A[i%4] + i%31; } for (int i = 0; i < 1000000; i++) { B[i%3] = A[i%4]; } </pre>	<p>Пример 2</p> <pre> int k = 0; for (int i = 0; i < 1000000; i++){ A[i] = (i*71)/39; } for (int i = 0; i < 1000000; i++) { k += A[i]; A[i] = k; } </pre>
<p>Пример 3</p> <pre> for(int i = 0; i < 999999; i++) { B[i] = 1000 - i; A[i] = B[i] * i; A[i + 1] = A[i] + i; } for(int i = 0; i < 999999; i++) { C[i] = D[i] * E[i]; D[i] = D[i]/2; } </pre>	<p>Пример 4</p> <pre> int j = 1, k = 1; do{ if (cnt[j]) run[cur++] = j - 1; j++; } while(j <= numbuckets); do{ cnt[k] += cnt[k - 1]; k++; } while(k <= numbuckets); </pre>

Рисунок 2.14 – Исходные файлы для проведения тестов

Во втором примере циклы работают с одним и тем же целочисленным массивом, причём с теми же индексами. Таким образом, их объединение кроме уменьшения накладных расходов должно улучшить использование локальности ссылок, что влечёт дополнительное увеличение производительности.

В третьем примере в разных циклах идёт работа с не пересекающимися множествами массивов. В теории их объединение может замедлить работу программы, в которой до него была лучшая локальность ссылок.

В четвёртом же примере была попытка опробовать трансформацию на реальном примере. Для этого была взята функция блочной сортировки массива из статьи про алгоритмы сортировки [23]. Но с одним из двух циклов, которые можно было объединить, снова возникла проблема с итератором, поэтому их код был слегка изменён. Ещё стоит отметить следующее: если один из циклов типа for, а другой – do/while, и при этом верхняя граница итератора не определена постоянным числом, при их объединении также может возникнуть проблема из-за разного представления счётчиков. На

рисунке показана только небольшая часть изменённого кода с указанными двумя циклами. В таком виде они были успешно объединены.

Измерения были проведены точно так же, как при тестировании разделения циклов, но, в отличие от него, каждый пример имел только две версии – с проходом -loop-fusion и без него. Также в связи с тем, что последний тест основан на реальной программе, было принято решение не повторять его, а измерить время одного выполнения. Результаты тестирования в виде сводной гистограммы приведены на рисунке 2.15.

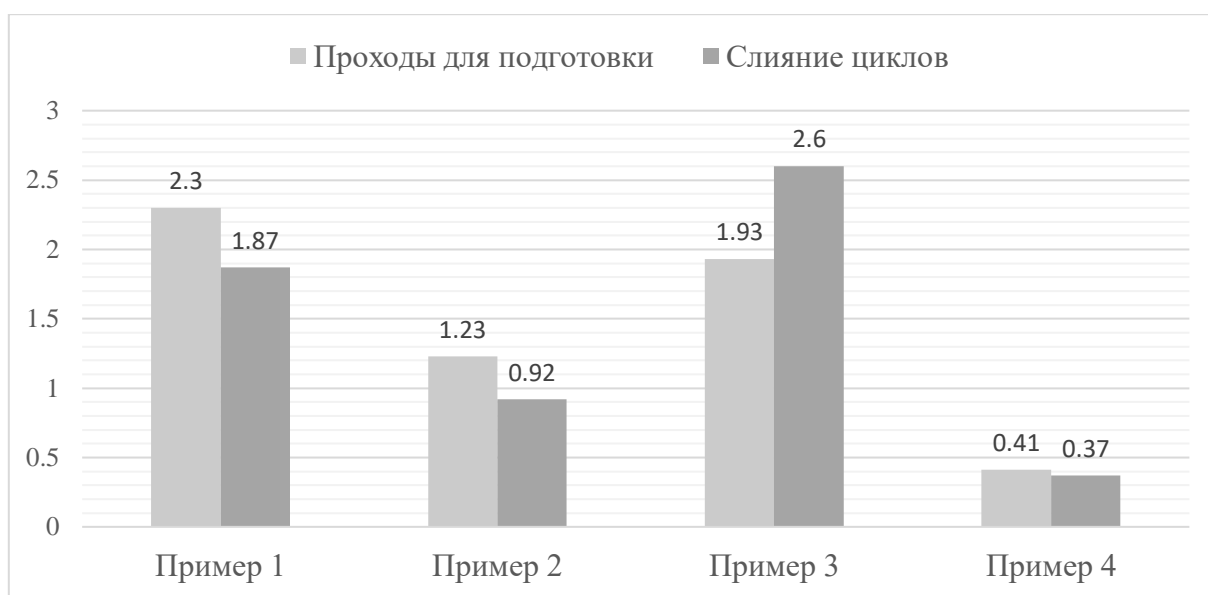


Рисунок 2.15 –Результаты тестирования

Полученные результаты приблизительно соответствуют ожиданиям. В первом примере несмотря на то, что ускорять выполнение должно только уменьшение накладных расходов, снижение времени выполнения довольно значительное – почти 19%. Такое большое изменение подкреплено тем, что циклы экспериментальны, и в каждом из них только одно выражение, из-за чего операции управления циклом занимают значительную долю времени.

Как и ожидалось, во втором примере рост скорости ещё выше – около 25%. Это обеспечивается улучшением использования локальности ссылок путём объединения циклов, где происходит обращение к одним и тем же ячейкам памяти.

В третьем же примере обратная ситуация – в телах соединённых циклов не пересекающиеся множества используемых ячеек памяти. Следовательно, после выполнения проходов локальность ссылок ухудшается, причём настолько, что скорость выполнения программы снизилось почти на 35%.

Четвёртый же тест показывает влияние объединения циклов на реальную функцию. Её примером выступает блочная сортировка массивов. Объединение циклов также влечёт за собой улучшение локальности ссылок, но объединяемые циклы здесь – лишь малая часть программы, и не так сильно влияют на время её выполнения. Тем не менее, рост скорости более чем заметен – он составляет 10% от выполнения исходной программы. Это ещё раз доказывает актуальность трансформации объединения циклов.

Тем не менее, с данным преобразованием тоже следует обращаться осторожно. Хотя оно и даёт ускорение даже в нейтральных случаях, на определённых примерах объединение циклов может замедлять программу. Так что следующий шаг в развитии этой трансформации – анализ её выгоды на основе модели, оценивающей дистанцию между ссылками на память.

Вывод

В данной главе проводилось исследование трансформаций разделения и слияния массивов с помощью готовых проходов LLVM. Для этого были рассмотрены важные понятия, в том числе описание и особенности проекта LLVM, а также вкратце обоснована актуальность его использования. В ходе исследования самих трансформаций был произведён анализ кода соответствующих файлов, разработаны тесты и проведены практические эксперименты, показывающие функционал проходов и их эффективность.

Цель прохода разделения циклов – отделить строки, которые мешают дальнейшей векторизации. При этом срабатывает он на редких примерах, и довольно часто отрицательно влияет на производительность. Даже в тестах, где, казалось бы, скорость выполнения должна увеличиваться за счёт

локальности ссылок, время в итоге возрастало даже после прохода векторизации. Поэтому несмотря на то, что данный проход включен в стандартные конвейеры проходов для оптимизации, он всегда выключен по умолчанию, и включается либо глобально флагом `-enable-loop-distribute`, либо отдельно для каждого цикла с помощью специальной директивы. Исходя из этого, дальнейшие работы с трансформацией разделения циклов надо проводить крайне осторожно.

В проходе объединения циклов не реализована функция, отвечающая за определение выгоды данной трансформации, поэтому он пытается соединить все возможные циклы, составляя для этого списки кандидатов. Это неэффективно для оптимизации, но открывает пространство для экспериментов. Тем не менее, не все возможные объединения осуществляются из-за ошибок при анализе счётчиков. В процессе тестирования было доказано, что данная трансформация может значительно увеличить скорость выполнения циклов, но также может и, наоборот, повлиять отрицательно. Чтобы этого избежать, необходимо анализировать локальность ссылок. Также было проведено тестирование трансформации объединения циклов на примере реальной программы, в результате которого был обнаружен заметный рост производительности, что ещё раз доказывает актуальность этого направления.

3 Трансформации с помощью Clang AST

В ходе исследования трансформаций также были изучены преобразования с помощью Clang на уровне AST. Как уже упоминалось, Clang – это фронтенд LLVM, транслирующий код на языках семейства C в IR, попутно выполняя анализ и оптимизацию. При этом он превращает исходник в абстрактное синтаксическое дерево (AST), где каждая конструкция имеет ссылку на положение в исходном тексте, что подробнее описано в следующем подразделе.

3.1 Дерево AST

Дерево AST представлено в виде иерархии узлов, по большей части наследующихся от основных гибких классов: Decl – объявление, Stmt – оператор, Type – тип [24]. Они включают себя более конкретные классы, например: CallExpr – вызов функции, BinaryOperator – бинарный оператор, например сложение, FunctionDecl – объявление или прототип функции. Чтобы проще понять дерево, можно вывести его для простого исходного кода в терминал с помощью команды «\$ clang -Xclang -ast-dump -fsyntax-only» (рисунок 3.1).

```

Исходный файл
int func(int a) {
    int b = 4;
    b = a - b;
    return b;
}

```

AST дерево

```

FunctionDecl 0x5582a0dd1750 <hello.c:1:1, line:5:1> line:1:5 func 'int (int)'
| -ParmVarDecl 0x5582a0dd1680 <col:10, col:14> col:14 used a 'int'
| -CompoundStmt 0x5582a0dd1a10 <col:17, line:5:1>
| | -DeclStmt 0x5582a0dd18e0 <line:2:5, col:14>
| | | -VarDecl 0x5582a0dd1858 <col:5, col:13> col:9 used b 'int' cinit
| | | | -IntegerLiteral 0x5582a0dd18c0 <col:13> 'int' 4
| | | -BinaryOperator 0x5582a0dd19a8 <line:3:5, col:13> 'int' '='
| | | | -DeclRefExpr 0x5582a0dd18f8 <col:5> 'int' lvalue Var 0x5582a0dd1858 'b' 'int'
| | | | -BinaryOperator 0x5582a0dd1988 <col:9, col:13> 'int' '-'
| | | | | -ImplicitCastExpr 0x5582a0dd1958 <col:9> 'int' <LValueToRValue>
| | | | | | -DeclRefExpr 0x5582a0dd1918 <col:9> 'int' lvalue ParmVar 0x5582a0dd1680 'a' 'int'
| | | | | -ImplicitCastExpr 0x5582a0dd1970 <col:13> 'int' <LValueToRValue>
| | | | | | -DeclRefExpr 0x5582a0dd1938 <col:13> 'int' lvalue Var 0x5582a0dd1858 'b' 'int'
| | -ReturnStmt 0x5582a0dd1a00 <line:4:5, col:12>
| | | -ImplicitCastExpr 0x5582a0dd19e8 <col:12> 'int' <LValueToRValue>
| | | | -DeclRefExpr 0x5582a0dd19c8 <col:12> 'int' lvalue Var 0x5582a0dd1858 'b' 'int'

```

Рисунок 3.1 – AST дерево

На примере выведенного дерева мы видим объявление функции (FunctionDecl) и её параметра (ParmVarDecl); выражение объявления (DeclStmt), включающее в себя объявление переменной (VarDecl); бинарный оператор (BinaryOperator) и его участники; оператор return (ReturnStmt).

Clang также открывает возможности для написания собственного модуля для анализа и трансформации с помощью дерева AST. При этом преобразования будут происходить на уровне исходного кода, который будет как на входе, так и на выходе (source-to-source).

3.2 Собственный модуль на основе Clang AST

Для создания собственного модуля для трансформации на основе дерева AST используются выражения, сопоставляющиеся с узлами и позволяющие уточнять нужные, называемые матчерами [25]. Для их освоения в ходе работы был изучен набор справочных плагинов руководства «clang-tutor» [26] и на его основе написан собственный модуль.

Созданный плагин позволяет переименовать переменную в исходном коде, при этом принимает из командной строки в качестве аргументов старое (OldName) и новое (NewName) имя. Основная часть кода программы приведена в приложении А, при этом часть с обработкой принимаемых параметров опущена. На рисунке 3.2 приведены матчеры, с помощью которых отслеживаются все использования переменной с названием, прописанном в параметре OldName.

```
const auto MatcherForVarDecl = varDecl(namedDecl(hasName(OldName)))
    .bind("VarDecl");
Finder.addMatcher(MatcherForVarDecl, &CodeRefactorHandler);

const auto MatcherForVarStmt = declRefExpr(to(varDecl(namedDecl(
    hasName(OldName))))).bind("VarStmt");
Finder.addMatcher(MatcherForVarStmt, &CodeRefactorHandler);
```

Рисунок 3.2 – Матчеры собственного модуля

Первый матчер находит само объявление переменной (varDecl) с нужным названием (hasName). Выражение nameDecl нужна скорее для приведения к нужному типу, проверяя, что переменная именная. При этом объявлению устанавливается в соответствие ключевое слово VarDecl, и затем матчер добавляется в объект Finder класса MatchFinder, что даёт сигнал другой функции, которая осуществляет саму замену имени. Последняя функция также приведена в приложении А.

Второй матчер находит в выражениях переменные, которые ссылаются (declRefExpr) на объявление, описанное до этого. Поэтому можно заметить, что второй матчер включает в себя первый. Далее матчер также добавляется в объект Finder.

Пример работы модуля можно увидеть на рисунке 3.3, где слева приведён исходный код программы, сверху – команда, справа – результат в консоли.

```

akostin@LAPTOP-5LNLDH91:~/dir/clang-tutor/build$ clang -c -load ~/dir/clang-tutor/build
/lib/libCodeRefactor.so -plugin CodeRefactor -plugin-arg-CodeRefactor -old-name -plugin-arg-CodeRefactor
nline -plugin-arg-CodeRefactor -new-name -plugin-arg-CodeRefactor lineNumber ~/dir/clang-tutor/test/1.
c
/* Треугольник из звездочек */
#include <stdio.h>
/* Печать n символов с */
printn(c, n){
    while( --n >= 0 )
        putchar(c);
}
int lines = 10; /* число строк треугольника */
void main(argc, argv) char *argv[];
{
    register int nline; /* номер строки */
    register int naster; /* количество звездочек */
    register int i;
    if( argc > 1 )
        lines = atoi( argv[1] );
    for( nline=0; nline < lines ; nline++){
        naster = 1 + 2 * nline;
        /* лидирующие пробелы */
        printn(' ', lines-1 - nline);
        /* звездочки */
        printn('*', naster);
        /* перевод строки */
        putchar( '\n' );
    }
    exit(0); /* завершение программы */
}

CodeRefactor arg = -old-name
CodeRefactor arg = -new-name
/home/akostin/dir/clang-tutor/test/1.c:2:14: fatal error: 'stdio.h' f
#include <stdio.h>
/* Треугольник из звездочек */
#include <stdio.h>
/* Печать n символов с */
printn(c, n){
    while( --n >= 0 )
        putchar(c);
}
int lines = 10; /* число строк треугольника */
void main(argc, argv) char *argv[];
{
    register int lineNumber; /* номер строки */
    register int naster; /* количество звездочек в строке */
    register int i;
    if( argc > 1 )
        lines = atoi( argv[1] );
    for( lineNumber=0; lineNumber < lines ; lineNumber++){
        naster = 1 + 2 * lineNumber;
        /* лидирующие пробелы */
        printn(' ', lines-1 - lineNumber);
        /* звездочки */
        printn('*', naster);
        /* перевод строки */
        putchar( '\n' );
    }
    exit(0); /* завершение программы */
}

```

Рисунок 3.3 – Пример работы модуля

Сама команда получается длинной, но можно заметить, что параметром `old-name` введено «`nline`», а `new-name` – «`lineNumber`». Далее в консоль выводится текст программы слева, в которой переменная `nline` успешно переименована в `lineNumber` в объявлении и всех использованиях. Помимо вывода в консоль получившийся код можно записать в файл, либо применить изменения прямо в исходном.

Вывод

В данном разделе были разобраны особенности трансформации кода с помощью Clang AST дерева, при которой и входом, и выходом является исходный файл на языке семейства C. При этом частично разобраны матчеры, с помощью которых происходит большая часть преобразований. Также разработан собственный модуль на основе описанных свойств дерева AST, приведён пример его работы. Этот модуль позволяет переименовать переменную во всём коде, взаимодействуя с деревом AST с помощью

матчеров. Хотя подобное преобразование не влечёт за собой роста производительности, написание собственной программы наиболее чётко показывает понимание темы, а также может сократить время при работе с большим проектом.

4 Экономическое обоснование ВКР

Для некоторых программ производительность является ключевым фактором. Трансформация исходного кода может позволить ускорить выполнение метода в несколько раз путем изменения всего лишь нескольких его строк, а LLVM предоставляет множество средств для преобразования исходного кода, что предоставляет много пространства для их исследования и практического применения.

Среди прочих трансформаций существует трансформации циклов, которые также могут значительно увеличивать производительность программы, но не все оптимизации доказали свою эффективность и не были включены в стандартные конвейеры проходов LLVM. Поэтому их исследование может открыть новое направление для оптимизаций.

В рамках экономического обоснования ВКР были выполнены следующие расчеты [27]:

- составлен детализированный план-график выполнения работ, позволяющий определить совокупную трудоемкость проведения им своего исследования;
- оценена величина заработной платы и социальных отчислений участников исследования;
- оценены затраты, связанные с приобретением необходимого сырья, материалов, комплектующих, полуфабрикатов;
- определена величина амортизационных отчислений используемых основных средств;
- оценены накладные расходы;
- рассчитана совокупная величина затрат, связанных с проведением исследования.

4.1 Составление плана-графика выполнения работ

Первым этапом при расчете полных затрат на выполнение разработки является расчет полных затрат при выполнении исследования начинается с составления детализированного плана работ, которые необходимо выполнить на каждом этапе проектирования. Для расчета затрат на этапе проектирования необходимо определить продолжительность каждой работы. В данном обосновании единица измерения трудоемкости работ – человеко-дни. План-график с указанием исполнителей и трудоемкости работ представлен в таблице 4.1.

Таблица 4.1 – План-график выполнения работ

№ работ ы	Наименование работы	Исполнитель	Длительность работы, человеко-дни
1	Консультации с научным руководителем	Руководитель	2
		Студент	2
2	Исследование области работы	Студент	12
3	Постановка задачи, выдача технического задания	Руководитель	1
4	Получение и изучение задания	Студент	2
5	Сборка и установка LLVM.	Студент	1
6	Изучение документации и примеров	Студент	5
7	Обзор статей и работ	Студент	5
8	Составление и анализ кода	Студент	10
9	Выполнение тестов и экспериментов	Студент	7
10	Подведение итогов	Руководитель	1
		Студент	1
11	Оформление пояснительной записки	Студент	12

Таким образом, на выполнение всех работ было затрачено:

– студентом: 57 ч. дн.

– руководителем: 4 ч. дн.

4.2 Расчет расходы на оплату труда исполнителей

На основе данных о трудоемкости выполняемых работ и ставки за день соответствующих исполнителей необходимо определить расходы на основную и дополнительную заработную плату исполнителей.

4.2.1 Расчет основной заработной платы исполнителей

На основе данных о месячных заработных платах руководителя и студента был произведен расчет оплаты одного человеко-дня каждого исполнителя. Месячный оклад руководителя составляет 85000 рублей, студента – 2200 рублей. Стоимость человеко-дня вычисляется путем деления месячного оклада исполнителя на количество рабочих дней в месяце, в данной работе принято равным 21.

Таким образом, стоимость человеко-дня составляет:

$$- \text{ для руководителя: } \text{СЧД}_{\text{рук}} = \frac{85000}{21} = 4\,047,62 \text{ руб/день};$$

$$- \text{ для студента: } \text{СЧД}_{\text{студ}} = \frac{2200}{21} = 104,76 \text{ руб/день};$$

Используя полученные ставки за день соответствующих исполнителей и трудоемкость выполняемых работ из пункта 4.1 можно рассчитать основную заработную плату исполнителей, рассчитываемую путем умножения затраченных на работу человеко-дней на стоимость человеко-дня. Тогда основная заработная плата составляет:

$$- \text{ для руководителя: } \text{З}_{\text{осн.з./пл рук}} = 4\,047,62 \cdot 4 = 16\,190,48 \text{ руб.};$$

$$- \text{ для студента: } \text{З}_{\text{осн.з./пл}} = 104,76 \cdot 57 = 5\,971,32 \text{ руб.}$$

Таким образом, общие затраты на основную заработную плату исполнителей составляет 22 161,8 рублей.

4.2.2 Расчет дополнительной заработной платы

После определения основной заработной платы производится расчет дополнительной заработной платы исполнителей по следующей формуле:

$$\text{З}_{\text{доп.з./пл}} = \text{З}_{\text{осн.з./пл}} \cdot \frac{H_{\text{доп}}}{100}, \quad (4.1)$$

где $Z_{\text{доп.з/пл}}$ – дополнительная заработная плата исполнителей, руб., $Z_{\text{осн.з/пл}}$ – основная заработная плата исполнителей, руб., $H_{\text{доп}}$ – норматив дополнительной заработной платы.

Норматив дополнительной заработной платы в СПбГЭТУ принят равным 8,3 %.

В соответствии с формулой 4.1 и данными из пункта 4.2.1 дополнительная заработная плата составляет:

– для руководителя: $Z_{\text{доп.з/пл рук.}} = 16\,190,48 \cdot 0,083 = 1\,343,81$ руб.;

– для студента: $Z_{\text{доп.з/пл студ.}} = 9\,771,51 \cdot 0,083 = 811,04$ руб.

Таким образом, общие затраты на дополнительную заработную плату исполнителей составляет 2 154,85 рублей.

4.2.3 Итоговые расходы на оплату труда

Исходя из пунктов 4.2.1 и 4.2.2 в таблице 4.2 представлены итоги по расходам на оплату труда исполнителей.

Таблица 4.2 – Расходы на оплату труда исполнителей

Исполнитель	Основная заработная плата, руб.	Дополнительная заработная плата, руб.	Итого, руб
Руководитель	16 190,48	1 343,81	17 534,29
Студент	5 971,32	811,04	6 782,36
Общие затраты на оплату труда			24 316,65

Итого затраты по статье «Расходы на оплату труда» составили 24 316,65 рублей.

4.3 Расчет отчислений на социальные нужды

Отчисления в сторону страховых взносов на обязательное страхование (медицинское, социальное и пенсионное) вычисляется с основной и дополнительной заработной платы по формуле:

$$Z_{\text{соц}} = (Z_{\text{осн.з/пл}} + Z_{\text{доп.з/пл}}) \cdot \frac{H_{\text{соц}}}{100}, \quad (4.2)$$

где $Z_{\text{соц}}$ – отчисления на социальные нужды с заработной платы, руб., $H_{\text{соц}}$ – норматив отчислений на страховые взносы на обязательное страхование, $Z_{\text{доп.з/пл}}$ – дополнительная заработная плата исполнителей, руб., $Z_{\text{осн.з/пл}}$ – основная заработная плата исполнителей, руб.

Норматив отчислений страховых взносов на обязательное социальное, пенсионное и медицинское страхование на 2022 год составляет 30 %.

В соответствии с формулой 4.2 и данными из пунктов 4.2.1 и 4.2.2 отчисления на социальные нужды составляют:

– для руководителя: $Z_{\text{соц рук.}} = (16\,190,48 + 1\,343,81) \cdot 0,3 = 5\,260,29$ руб.;

– для студента: $Z_{\text{соц студ.}} = (5\,971,32 + 811,04) \cdot 0,3 = 2\,034,71$ руб.;

Итого затраты по статье «Отчисления на социальные нужды» составляют 7 295 рублей.

4.4 Расчет затрат на сырье и материалы

Расчет затрат на сырье и материалы производится по формуле:

$$Z_{\text{м}} = \sum_{i=1}^L G_i C_i \left(1 + \frac{H_{\text{т.з.}}}{100}\right), \quad (4.3)$$

где $Z_{\text{м}}$ – затраты на сырье и материалы, руб., i – индекс вида сырья или материала, G_i – норма расхода i -го материала на единицу продукции, ед., C_i – цена приобретения единицы i -го материала, руб. за единицу, $H_{\text{т.з.}}$ – норма транспортных затрат.

Норма транспортных затрат в данной ВКР принята за 10%.

Список расходных материалов и затрат на их приобретение представлен в таблице 4.3.

Таблица 4.3 – Затраты на сырье и материалы

Изделие	Тип	Норма расхода изделия, ед.	Цена единицу изделия, руб./ед.	Транспортные расходы, руб.	Итого, руб.
Бумага офисная	A4, 500 листов	1	270	27	297

Картридж для принтера	Лазерный, черный	1	615	61,5	676,5
Общие затраты на сырье и материалы, руб.					973,5

Общие расходы по статье «Сырье и материалы» составили 973,5 рублей.

4.5 Расчет амортизационных отчислений

Амортизационные отчисления по i -му основному средству рассчитываются по формуле:

$$A_i = Ц_{п.н.i} \cdot \frac{H_{ai}}{100}, \quad (4.4)$$

где A_i – амортизационные отчисления за год по i -му основному средству (руб.); $Ц_{п.н.i}$ – первоначальная стоимость i -го основного средства, руб.; H_{ai} – годовая норма амортизации i -го основного средства, %; Норма амортизации рассчитывается как:

$$H_{ai} = \frac{1}{T_{п.и.}} \cdot 100\%, \quad (4.5)$$

где $T_{п.и.}$ – срок полезного использования основного средства.

Амортизационные отчисления по i -му основному средству, используемому при работе над ВКР, определяются как:

$$A_{iВКР} = A_i \cdot \frac{T_{iВКР}}{12}, \quad (4.6)$$

где $A_{iВКР}$ – амортизационные отчисления по i -му основному средству, используемому при работе над ВКР, руб.; A_i – амортизационные отчисления по i -му основному средству за год, руб.; $T_{iВКР}$ – время, в течение которого используется i -е основное средство, мес.

Список и расчет амортизационных отчислений по основным средствам, используемым в ВКР, приведены в таблицах 4.4 и 4.5.

Таблица 4.4 – Амортизационные отчисления за год

Основное средство	Первоначальная стоимость, руб.	Срок полезного использования, лет	Годовая норма амортизации, %	Амортизационные отчисления за год, руб
-------------------	--------------------------------	-----------------------------------	------------------------------	--

Ноутбук HONOR MagicBook 15 Intel Core i5 1135G7	60 000	3	33%	19 800
Принтер Canon LBP2900	4 000	3	33%	1 320

Таблица 4.5 – Амортизационные отчисления за период выполнения ВКР

Основное средство	Амортизационные отчисления за год, руб	Время выполнения ВКР, месяцев	Амортизационные отчисления, руб
Ноутбук HONOR MagicBook 15 Intel Core i5 1135G7	19 800	3	4 950
Принтер Canon LBP2900	1 320	3	330

Итого общие затраты по статье «Амортизационные отчисления» составляют 5 280 рублей.

4.6 Расчет накладных расходов

В рамках данной работы сумма накладных расходов рассчитывается как доля в 20% от общей стоимости разработки. Её размер вычисляется следующим образом:

$$З_{\text{накл}} = (24\,316,65 + 7\,295 + 973,5 + 5\,280) \cdot 0,2 = 7\,573,03 \text{ руб.}$$

Таким образом, затраты по статье «Накладные расходы» составляют 7 573,03 рублей.

4.7 Расчет совокупной величины затрат

Итоги по статьям расходов, а также расчет совокупной величины затрат, связанных с проведением исследования представлен в таблице 4.6.

Таблица 4.6 – Совокупная величина затрат

Наименование статьи	Сумма, руб.	Доля в общей стоимости, %
Расходы на оплату труда	24 316,65	53,52
Отчисления на социальные нужды	7 295	16,05
Сырье и материалы	973,5	2,14
Амортизационные отчисления	5 280	11,62
Накладные расходы	7 573,03	16,67

Итого	45 438,18	100
-------	-----------	-----

Общая себестоимость работы составила 45 438,18 рублей.

Вывод

В данном разделе для экономического обоснования ВКР были по статьям приведены затраты на исследование и оценена его полная себестоимость.

Совокупная величина затрат составляет 45 438 рублей и 18 копеек. Статьями расходов с наибольшей долей расходов стали: оплата труда (59,14%); отчисления на социальные нужды (17,74%); накладные расходы (16,67%).

Оптимизация циклов – одно из самых актуальных направлений трансформации кода с целью оптимизации. При этом незначительные изменения циклов могут значительно повлиять на производительность программы. Данное исследование может внести значительный вклад в его развитие и помочь в дальнейшем создании автоматизированного инструмента оптимизации путём трансформации, что оправдывает такую себестоимость.

ЗАКЛЮЧЕНИЕ

Целью данной работы являлось исследование средств трансформации исходного кода на основе инструментов LLVM для разделения и слияния циклов. Для этого были описаны проблемы доступа к памяти, понятие локальности ссылок и различных оптимизаций на их основе. Далее с их использованием было проведено исследование кода соответствующих файлов исходников LLVM, проведены практические эксперименты, на основе которых сделаны выводы о функционале и эффективности трансформаций разделения и слияния циклов. Кроме этого, был проведён вводный анализ другого средства трансформации, на основе Clang AST, а также написан собственный модуль, выполняющий простое преобразование.

Слияние циклов оказалось более эффективным чем разделение за счёт того, что при первой трансформации накладные расходы на управление уменьшаются, а при второй – увеличиваются. Но в циклах, работающих с большими данными, этот показатель уходит на второй план, а основное изменение в производительности происходит из-за изменения локальности ссылок. При этом оба преобразования на разных примерах могут как значительно повысить, так и заметно снизить скорость выполнения программы.

Готовые реализации данных проходов в LLVM никак не учитывают локальность ссылок. Разделение циклов пытается отделить строки тела цикла, не позволяющие дальнейшую его реализацию, а в слиянии цикла вовсе не реализована функция, проверяющая выгоду трансформации. Таким образом, для использования в текущем виде эти проходы не годятся, но это позволило провести различные тесты, в том числе и снижающие производительность, и с помощью графиков на основе результатов тестирования проанализировать эффективность исследуемых преобразований. В качестве одного из тестов было проведение трансформации объединения на примере реальной функции,

и как результат было заметное увеличение скорости выполнения после соединения всего двух циклов.

Совершённые в данной работе исследование, анализ и тесты трансформаций разделения и слияния циклов доказывают актуальность этого направления, а также открывают дальнейшие возможности как для доработки рассматриваемых проходов LLVM, так и для написания новых инструментов. Следующий очевидный, но не самый простой шаг для обеих трансформаций – разработка модели, анализирующей дистанцию между используемыми ячейками памяти и оценивающей на её основе выгоду преобразования с точки зрения локальности ссылок. Другой вариант развития – разработка программы, выполняющей поочерёдные трансформации и проверяющие изменения скорости выполнения, но такой подход влечёт за собой долгое время оптимизации. Могут быть выбраны и другие пути развития, но опыт данного исследования сократит время большинству из них.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Bryant, R. E. Computer Systems: A Programmer's Perspective = Компьютерные системы: взгляд программиста / Randal E. Bryant, David R. O'Hallaron, 2010. – 1128 с.
2. Herlihy, M. The Art of Multiprocessor Programming = Искусство многопроцессорного программирования / Maurice Herlihy, Nir Shavit, 2012–536 с.
3. Курносов, М. Г. Оптимизация доступа к памяти (memory access optimization) / Курносов Михаил Георгиевич // Курс «Высокопроизводительные вычислительные системы» – Сибирский государственный университет телекоммуникаций и информатики – Новосибирск, 2015.
4. Compiler construction / by Wikipedians, 2011 – 688 с.
5. Denis Bakhvalov. Machine code layout optimizations = Оптимизация компоновки машинного кода – <https://easyperf.net/blog/2019/03/27/Machine-code-layout-optimizatoins#profile-guided-optimizations-pgo>.
6. German Gorelkin. Выравнивание и заполнение структур – <https://medium.com/german-gorelkin/go-alignment-a359ff54f272>.
7. Курносов, М. Г. Оптимизация ветвлений и циклов (branch prediction & loop optimization) / Курносов Михаил Георгиевич // Курс «Высокопроизводительные вычислительные системы» – Сибирский государственный университет телекоммуникаций и информатики – Новосибирск, 2015.
8. Paredes, J. C++ Programming: Good Principles For Excellent Endings = Программирование на C++: хорошие принципы для отличного завершения / João Paredes, 2011 – 202 с.
9. Cardoso, J. M. P. Embedded Computing for High Performance = Встроенные вычисления для высокой производительности / João M.P. Cardoso, José Gabriel F. Coutinho, Pedro C. Diniz, 2017 – 297 с.

10. Huang, J. C. Generalized Loop-Unrolling: a Method for Program Speed-Up = Обобщенное разворачивание цикла: метод ускорения программы / J. C. Huang, T. Leng // Department of Computer Science – The University of Houston – 9 с.
11. Srikant, Y. N. The Compiler Design Handbook: Optimizations and Machine Code Generation = Справочник по проектированию компиляторов: оптимизация и генерация машинного кода / Y. N. Srikant, Priti Shankar, 2018 – 784 с.
12. Habr. Что такое LLVM и зачем он нужен? / Андрей Боханко – <https://habr.com/ru/company/huawei/blog/511854/>.
13. Википедия. GNU Compiler Collection – https://ru.wikipedia.org/wiki/GNU_Compiler_Collection.
14. Википедия. Apple – <https://ru.wikipedia.org/wiki/Apple>.
15. Википедия. Google (компания) – [https://ru.wikipedia.org/wiki/Google_\(компания\)](https://ru.wikipedia.org/wiki/Google_(компания)).
16. Lattner, C. LLVM and Clang: Next Generation Compiler Technology = LLVM и Clang: технология компиляции следующего поколения / Chris Lattner, 2008 – 33 с.
17. Лопес, Б. К. LLVM: инфраструктура для разработки компиляторов / Бруно Кардос Лопес, Рафаэль Аулер / пер. с англ. Киселев А. Н. – М.: ДМК Пресс, 2015. – 342 с.: ил.
18. Документация LLVM. Writing an LLVM Pass = Написание прохода LLVM – <https://llvm.org/docs/WritingAnLLVMPass.html>.
19. Документация LLVM. LLVM Loop Terminology (and Canonical Forms) = Терминология циклов LLVM (и канонические формы) – <https://llvm.org/docs/LoopTerminology.html>.
20. Документация LLVM. LLVM's Analysis and Transform Passes = Проходы анализа и преобразования LLVM – <https://llvm.org/docs/Passes.html>.

21. Schmidt, B. Parallel Programming: Concepts and Practice = Параллельное программирование: концепции и практика / Bertil Schmidt, Jorge González-Domínguez, Christian Hundt, Moritz Schlarb, 2018 – 403с.
22. Документация LLVM. Auto-Vectorization in LLVM = Автоматическая векторизация в LLVM – <https://llvm.org/docs/Vectorizers.html>.
23. Habr. Описание алгоритмов сортировки и сравнение их производительности / Михаил Опанасенко – <https://habr.com/ru/post/335920/>.
24. Документация Clang. Introduction to the Clang AST = Введение в Clang AST – <https://clang.llvm.org/docs/IntroductionToTheClangAST.html>.
25. Документация Clang. AST Matcher Reference = Справочник по матчерам AST – <https://clang.llvm.org/docs/LibASTMatchersReference.html>.
26. GitHub. Example Clang plugins for C and C++ – based on Clang 13 = Примеры подключаемых модулей Clang для C и C++ – на основе Clang 13 / Andrzej Warzyński (banach-space) – <https://github.com/banach-space/clang-tutor>.
27. Алексеева О. Г. Выполнение дополнительного раздела ВКР бакалаврами технических направлений: Учебно-методическое пособие / О.Г. Алексеева // СПб.: Изд-во СПбГЭТУ “ЛЭТИ”, 2018 – 15 с.

ПРИЛОЖЕНИЕ А

Код модуля Clang

```
// Переопределение функции, запускающей при добавлении узла в Finder
void CodeRefactorMatcher::run(const MatchFinder::MatchResult &Result) {

    // Получение узла, связанного с VarStmt (использование переменной в
    // выражениях)
    const DeclRefExpr *VarStmt =
        Result.Nodes.getNodeAs<clang::DeclRefExpr>("VarStmt");

    // Получение расположения самой переменной в исходном коде и замена
    // этого участка кода введённым в параметрах новым именем
    if (VarStmt) {
        SourceRange CallExprSrcRange = VarStmt->getLocation();
        CodeRefactorRewriter.ReplaceText(CallExprSrcRange, NewName);
    }

    // Получение узла, связанного с VarDecl (объявлением переменной)
    const NamedDecl *VarDecl =
        Result.Nodes.getNodeAs<clang::NamedDecl>("VarDecl");

    // Получение расположения имени переменной в исходном коде и замена
    // этого участка кода введённым в параметрах новым именем
    if (VarDecl) {
        SourceRange VarDeclSrcRange = VarDecl->getLocation();
        CodeRefactorRewriter.ReplaceText(
            CharSourceRange::getTokenRange(VarDeclSrcRange), NewName);
    }
}

// ASTConsumer – большой класс, использующийся клиентами, читающими
// дерево AST. Он позволяет перебирать узлы и находить нужные с
// помощью матчеров, что и реализовано в данной функции
CodeRefactorASTConsumer::CodeRefactorASTConsumer(Rewriter &R,
    std::string OldName, std::string NewName)
    :CodeRefactorHandler(R, NewName),
    OldName(OldName), NewName(NewName) {

    // Матчер, который находит объявление с именем, а именно названием,
    // записанном в параметре OldName. При этом происходит связывание с
    // ключевым словом VarDecl.
    // Затем матчер добавляется в Finder класса MatchFinder, что даёт
    // сигнал к вызову функции CodeRefactorMatcher::run.
    const auto MatcherForVarDecl = varDecl(namedDecl(hasName(OldName))
        .bind("VarDecl"));
    Finder.addMatcher(MatcherForVarDecl, &CodeRefactorHandler);

    // Аналогичное действие, но матчер находит в выражениях переменную,
    // которая ссылается на объявление, описанное в предыдущем матчере.
    const auto MatcherForVarStmt = declRefExpr(to(varDecl(namedDecl(
        hasName(OldName))))).bind("VarStmt");
    Finder.addMatcher(MatcherForVarStmt, &CodeRefactorHandler);
}
```