

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

<b>Направление</b>	09.04.04 - Программная инженерия
<b>Программа</b>	Математическое и программное обеспечение систем искусственного интеллекта
<b>Факультет</b>	КТИ
<b>Кафедра</b>	МО ЭВМ

*К защите допустить*

И.О. зав. кафедрой

А.А. Лисс

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА  
МАГИСТРА**

**Тема: АНАЛИЗ МЕТОДОВ ОПТИМИЗАЦИИ РАЗМЕЩЕНИЯ  
ИНСТРУКЦИЙ МАШИННОГО КОДА**

Студент		<hr/>	А.В. Грибов
		<i>подпись</i>	
Руководитель	к.т.н., доцент	<hr/>	А.А. Пазников
		<i>подпись</i>	
Консультанты	к.э.н., доцент	<hr/>	М.Н. Магомедов
		<i>подпись</i>	
	к.т.н.	<hr/>	М.М. Заславский
		<i>подпись</i>	

Санкт-Петербург  
2023

## ЗАДАНИЕ

Утверждаю

И.О. зав. кафедрой МО ЭВМ

\_\_\_\_\_ А.А. Лисс

«        » 2023 г.

Студент                    Грибов А.В.

Группа 7304

Тема работы: Анализ методов оптимизации размещения инструкций машинного кода.

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

Исходные данные (технические требования):

Алгоритмы методов оптимизации размещения инструкций машинного кода, алгоритмы профилирования, а также инструменты для их применения.

## Содержание ВКР:

«Введение», «Обзор методов оптимизации размещения инструкций машинного кода.», «Профилирование.», «Исследование и разработка тестовых файлов.», «Оценка и защита результатов интеллектуальной деятельности.», «Заключение».

Перечень отчетных материалов: пояснительная записка, презентация.

Дополнительные разделы: Оценка и защита результатов интеллектуальной деятельности.

Дата выдачи задания

Дата представления ВКР к защите

«06» февраля 2023 г.

«31» мая 2023 г.

Студент

А.В. Грибов

Руководитель к.т.н., доцент

А.А. Пазников

## КАЛЕНДАРНЫЙ ПЛАН ВЫПОЛНЕНИЯ ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

Утверждаю

И.О. зав. кафедрой МО ЭВМ

\_\_\_\_\_ А.А. Лисс

«      » 2023 г.

Студент            Грибов А.В.

Група 7304

Тема работы: Анализ методов оптимизации размещения инструкций  
машинного кода

№ п/п	Наименование работ	Срок выполнения
1	Обзор литературы по теме работы	06.02 – 16.03
2	Анализ существующих средств для сбора информации о работе программы	17.03 – 25.03
3	Исследование инструмента LLVM Bolt, использующегося для оптимизаций размещения инструкций	26.03 – 03.04
4	Составление тестовых программ	04.04 – 01.05
5	Оформление пояснительной записки	02.05 – 10.05
6	Оформление иллюстративного материала	11.05 – 12.05

Студент

А.В. Грибов

Руководитель к.т.н., доцент

А.А. Пазников

## РЕФЕРАТ

Пояснительная записка 73 стр., 35 рис., 13 табл., 25 ист.

МЕТОДЫ ОПТИМИЗАЦИИ, БЛОКИ ИНСТРУКЦИЙ, РАЗМЕЩЕНИЕ  
ИНСТРУКЦИЙ.

**Объектом исследования** являются методы оптимизации размещения инструкций машинного кода

**Предметом исследования** увеличение производительности программ при применении к ним методов оптимизации после предварительного профилирования.

**Цель работы:** Анализ методов оптимизации размещения инструкций машинного кода после предварительного профилирования.

В данной работе будет проведено исследование увеличения производительности программ посредством применения к ним методов оптимизации размещения инструкций после предварительного профилирования. Для этого будут разработаны тестовые программы, бинарные файлы которых будут анализироваться и оптимизироваться для наилучшего распределения инструкций внутри них.

В ходе работы были исследованы различные средства для профилирования программ. Затем были разработаны тестовые программы для тестирования различных особенностей. Для дальнейших исследований они были обработаны при помощи инструмента LLVM Bolt.

## **ABSTRACT**

In this paper, we will investigate the increase of program performance by applying code layout optimization techniques to programs after pre-profiling. For this purpose, test programs will be developed whose binary files will be analyzed and optimized for the best distribution of instructions within them.

In the course of this work, various tools for program profiling were investigated. Test programs were then developed to test the various features. For further research, they were processed using the LLVM Bolt tool.

## СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ.....	7
ВВЕДЕНИЕ .....	9
1 Обзор методов оптимизации размещения инструкций машинного кода .	11
1.1 Распределение базовых блоков .....	11
1.2 Выравнивание базовых блоков .....	13
1.3 Разбиение на функции .....	15
1.4 Группировка функций .....	16
2 Профилирование .....	19
2.1 LLVM.....	19
2.2 LLVM Bolt.....	22
2.3 Dynamorio.....	31
2.4 Perf .....	34
3 Исследование и разработка тестовых файлов .....	38
3.1 Настройка и работа с LLVM Bolt.....	38
4.2 Проведение тестирования на улучшение производительности .....	45
4 Оценка и защита результатов интеллектуальной деятельности .....	55
4.1 Описание интеллектуальной деятельности .....	55
4.2 Оценка рыночной стоимости интеллектуальной деятельности .....	55
4.3 Правовая защита результатов интеллектуальной деятельности .....	66
ЗАКЛЮЧЕНИЕ.....	69
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ .....	71
ПРИЛОЖЕНИЕ А.....	74
ПРИЛОЖЕНИЕ В .....	90

## ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

Горячий код – часто используемые куски кода в процессе работы программы;

Холодный код – редко используемые куски кода в процессе работы программы;

Профилирование – процесс сбора информации о программе во время её работы;

Профиль – файл, содержащий информацию о результатах профилирования программы;

ЭВМ – электронно-вычислительная машина;

Базовый блок – набор машинных инструкций, объединённый в единый блок;

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;

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

Clang – фронтэнд LLVM, работающий с языками семейства C;

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

Временная локальность – повторное обращение к одним и тем же адресам через небольшие промежутки времени;

Размещение инструкций – перемещение базовых блоков относительно друг друга;

LLVM Bolt – инструмент для различных оптимизаций, в основном направлен на методы оптимизаций с учётом размещения инструкций;



## ВВЕДЕНИЕ

Большинство программ имеют необходимость в повышении производительности. Это происходит посредством тех или иных оптимизаций. Данная необходимость постоянно возникает в том числе в результате постоянного увеличения количества обрабатываемых данных. Отрасль информационных технологий стремительно развивается, а, значит, растёт и количество её задач. В данный момент уже существует необходимость в обработке больших объёмов данных. Как правило их обработка требует множество однотипных операций небольшой сложности. Это приводит к тому, что программа производит огромное количество действий, которые, даже при небольшой оптимизации, могут значительно увеличить скорость получения результатов.

В связи с этим стоит постоянно искать новые способы оптимизации, ведь это позволит значительно ускорить работу подобных программ. На данный момент уже существует множество возможностей оптимизировать программы тем или иным способом.

Стоит так же упомянуть, что для того, чтобы проследить эффективность проведённых улучшений производительности, так и оптимизации программного обеспечения следует использовать профилирование. Оно собирает информацию, которая характеризует работу программы, такую, как количество функций или количество вызовов функций. Именно от методов профилирования зависит насколько хорошо или плохо сработают рассматриваемые в данной работе методы оптимизации размещения инструкций машинного кода. Без них оптимизации посредством размещения инструкций машинного кода были бы опасными и непредсказуемыми.

Целью работы является проведение анализа методов оптимизации размещения инструкций машинного кода после предварительного профилирования.

В связи с этим были выделены следующие задачи:

- Анализ предметной области, включающий изучение методов оптимизации с перемещением блоков инструкций машинного кода;

- Анализ существующих средств для сбора информации о работе программы;
- Изучение средств работы с LLVM;
- Исследование инструмента LLVM Bolt, использующегося для оптимизаций размещения инструкций;
- Сравнение различных средств для сбора информации о работе программы.
- Разработка тестовых программ для проверки эффективности работы различных методов оптимизации;

Объектом исследования в данной работе являются методы оптимизации размещения инструкций машинного кода

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

Практическая ценность работы заключается в возможности лучше использовать существующие методы оптимизации с размещением инструкций машинного кода в различных ситуациях. Это может позволить улучшить производительность существующих программ вплоть до 20%.

# 1 Обзор методов оптимизации размещения инструкций машинного кода

Всего существует несколько основных методов для распределения машинных инструкций:

- Распределение базовых блоков
- Выравнивание базовых блоков
- Разбиение на функции
- Группировка функций

## 1.1 Распределение базовых блоков

Под базовым блоком в данном случае следует понимать последовательность команд машинного кода с одним входом и одним выходом.

Компиляторы как правило работают на уровне базовых блоков. Это позволяет гарантировать, что каждая инструкция будет выполнена только один раз. Из набора таких блоков, по сути, и составляется программа. Они не имеют ветвлений внутри себя. К примеру, использование операторов, управляющих условным ветвлением, `if` и `else` в языке `c++` разбивает программу на несколько блоков. Пример такого разбиения представлен на рис. 1.

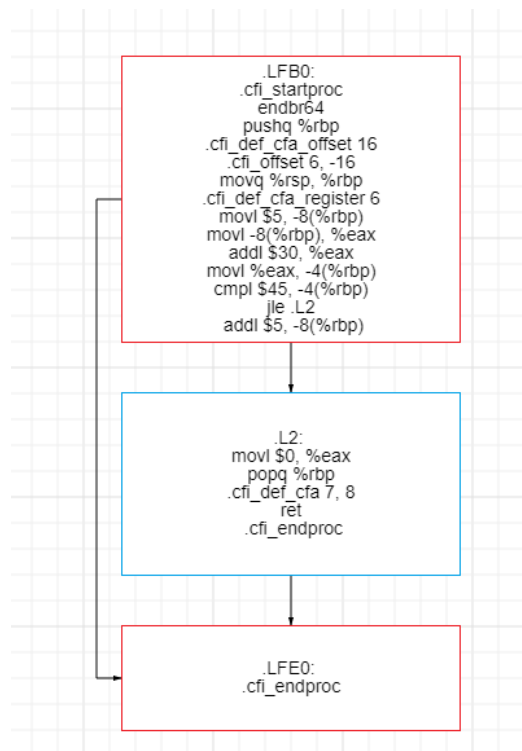


Рисунок 1 – Пример разбиения

Существуют такие понятия как горячий и холодный код [14], первый из них является наиболее часто используемым кодом, второй соответственно используется менее часто. Используя данные понятия и вышеописанное разбиение программы на блоки, можно взглянуть на всё под несколько иным углом. На рис. 2 представлен вариант кода с условным ветвлением, разбитый на блоки. На нём можно увидеть простейший пример из трёх блоков с горячим и с холодным кодом. Оптимизированный при помощи распределения базовых блоков вариант представлен на рис. 3. Здесь можно увидеть, что инструкции горячего кода идут непрерывно, без использования команды `jump`. Первый блок в ней выполняется всегда, поскольку является точкой входа, но смена расположения второго и третьего блока позволяет программе работать быстрее, поскольку третий используется значительно чаще.

```
1 int main()  
2 {  
3     int a=5;  
4     int b=a+30;  
5     if(b>45)  
6     {  
7         a+=5;  
8     }  
9     return 0;  
10 }
```

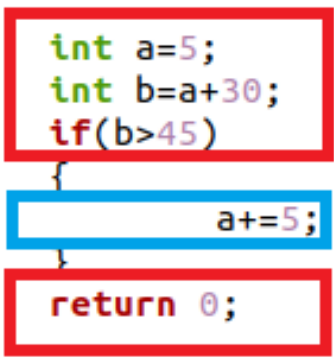


Рисунок 2 – Пример разделения на горячий и холодный код

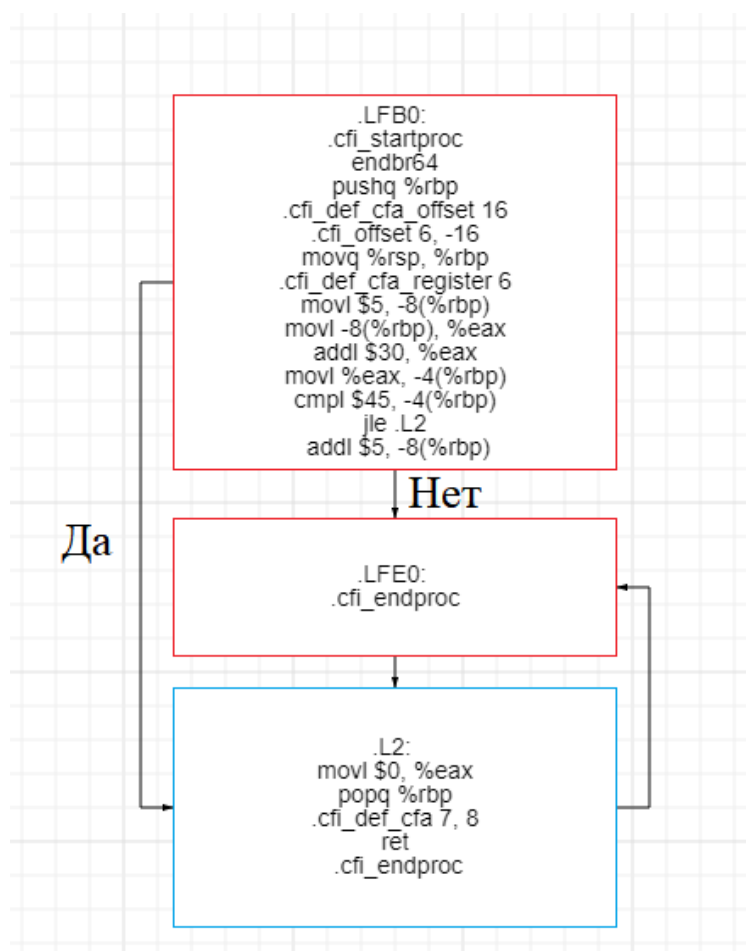


Рисунок 3 – Пример оптимизированных блоков

Конечно, в некоторых случаях данный вид оптимизации может и ухудшить время работы программы в определённых случаях, но в целом, в большинстве случаев программа получит значительный прирост в эффективности времени работы.

Данный вид оптимизации заключается в том, чтобы выставить базовые блоки инструкций с горячим кодом над блоками с холодным кодом. Это позволяет получить более оптимальную по времени исполнения программу.

## 1.2 Выравнивание базовых блоков

Выравнивание блоков относится к кэш-памяти [1]. Основная его идея заключается в том, чтобы собрать горячий код в одну кэш-линию, как на рис. 4. Это происходит за счёт пропуска ячеек в прошлой кэш-линии, где начинается цикл [2], таким образом весь цикл остаётся в одной кэш-линии и выполняется быстрее[9]. Если бы цикл находился в разных кэш-линиях, то процес-

сору пришлось бы тратить время на данные переходы. Это работает, поскольку существует такой блок как DSB(Decoded Stream Buffer), который является своеобразным кэшем микроопераций. Он позволяет найти какой-либо набор инструкций, который недавно выполнялся. В связи с этим можно будет избавиться не только от повторной трансляции данных, но и их чтении из оперативной памяти, что позволяет значительно ускорить работу программы.

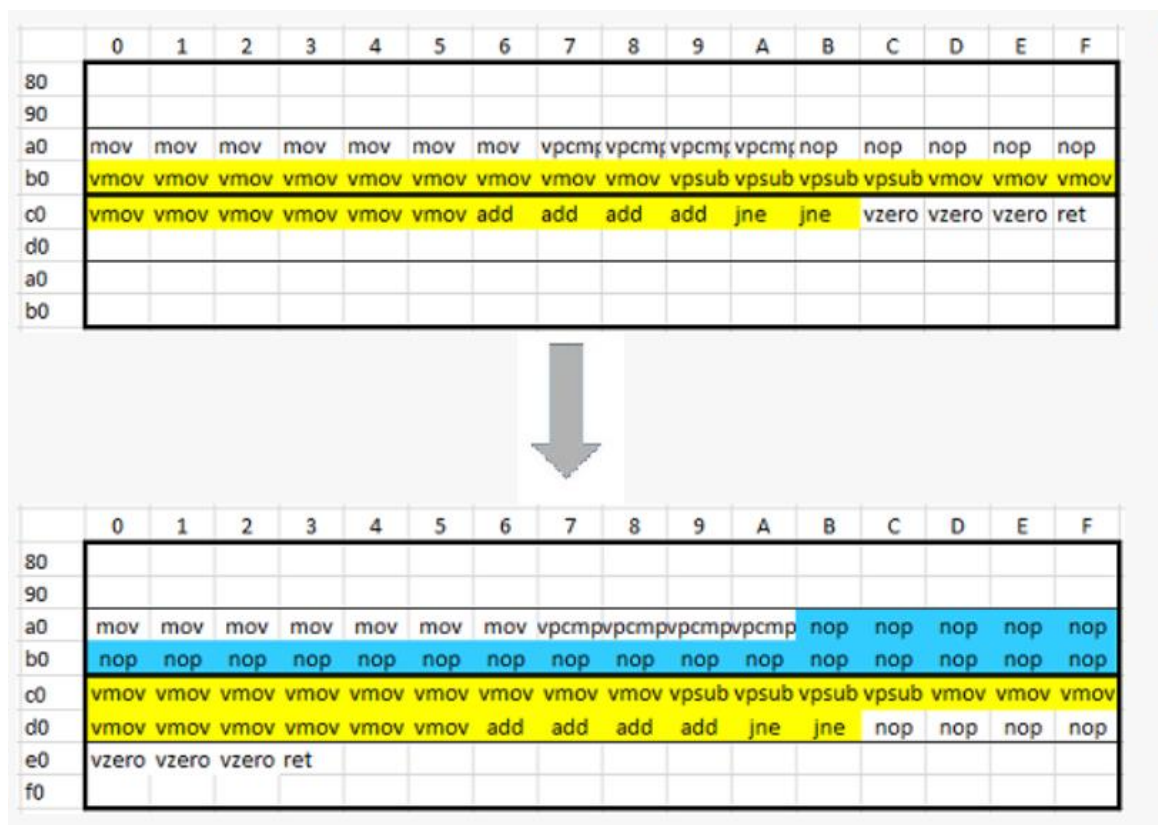


Рисунок 4 – Сбор горячего кода в одну кэш линию

Из минусов данного метода можно перечислить, что компилятор вставляет NOP в код инструкции [14], что приводит к увеличению размера бинарного файла, а также может ухудшить производительность программы, поскольку выполнение данных инструкций не получится абсолютно не затратным в отношении ресурсов машины, ведь их необходимо считывать и декодировать.

Этот метод оптимизации является исключительно микроархитектурной оптимизацией и чаще всего применяется к циклам. Он позволяет улучшить кэш инструкций и работу блока DSB.

### 1.3 Разбиение на функции

Функции так же являются одним из вариантов разделения программы на блоки. Каждая функция создаётся как отдельный блок. Основная идея разбиения функций заключается в разделении горячего и холодного кода. Это позволяет улучшить локальность памяти горячего кода. Этот параметр может значительно повлиять на скорость работы программы.

Локальность бывает двух видов [13], а именно пространственная и временная. Первая отвечает за повторные обращения к адресам, которые находятся близко друг к другу в памяти, за небольшой промежуток времени.

Временная же локальность отвечает за обращение к одному и тому же адресу в памяти спустя короткие промежутки времени. Разбиение на функции по признакам горячего и холодного кода показывает преимущества обоих видов локальности. В частности, отделение горячего кода хорошо отражает временную, поскольку он постоянно находится в одном и том же адресе памяти, который используется наиболее часто.

Пространственная же локальность отображается в размещении кусков такого кода в соседних адресах. Пример можно увидеть на рис. 5.

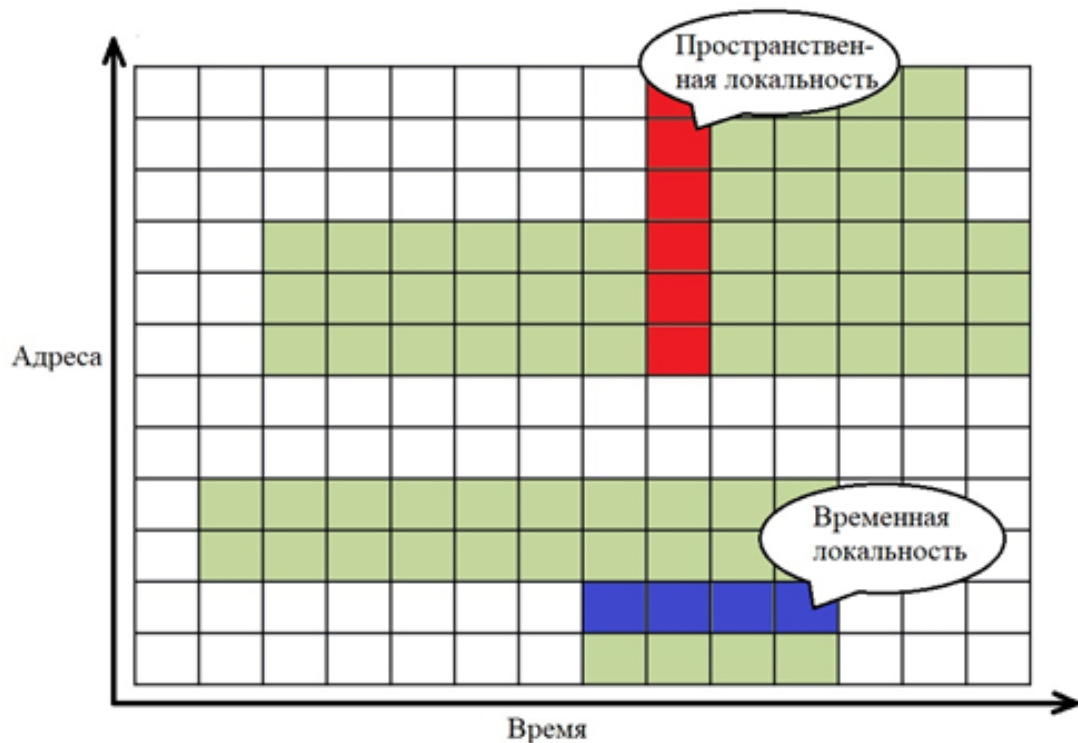


Рисунок 5 – Виды локальности

Из него можно увидеть, что холодные блоки кода были отделены от основного тела функции. На рис. 6 показан код до разбиения.

```
void foo(bool cond1, bool cond2) {  
    // hot path  
    if (cond1)  
        // cold code 1  
    //hot code  
    if (cond2)  
        // cold code 2  
}
```

Рисунок 6 – Код до разбиения

Для разделения кода на холодный и горячий в примере применено условное ветвление, которое, в данном случае, содержит холодный код при соблюдении условия. Его необходимо вынести в отдельные функции, что и происходит на рис. 7, а в ветвлении код заменён на вызов новых функций. В данном случае пример показан с точки зрения изменения кода программы.

```
void foo(bool cond1, bool cond2) {  
    // hot path  
    if (cond1)  
        cold1();  
    //hot code  
    if (cond2)  
        cold2();  
}  
  
void cold1() __attribute__((noinline)) { // cold code 1 }  
void cold2() __attribute__((noinline)) { // cold code 2 }
```

Рисунок 7 – Оптимизированный код

## 1.4 Группировка функций



Данный метод оптимизации является своего рода противоположностью предыдущему, поскольку он заключается уже в объединении функций. Но особенностью является то, что объединяются функции, содержащие в себе горячий код. Принцип работы данной оптимизации можно увидеть на рис. 8.

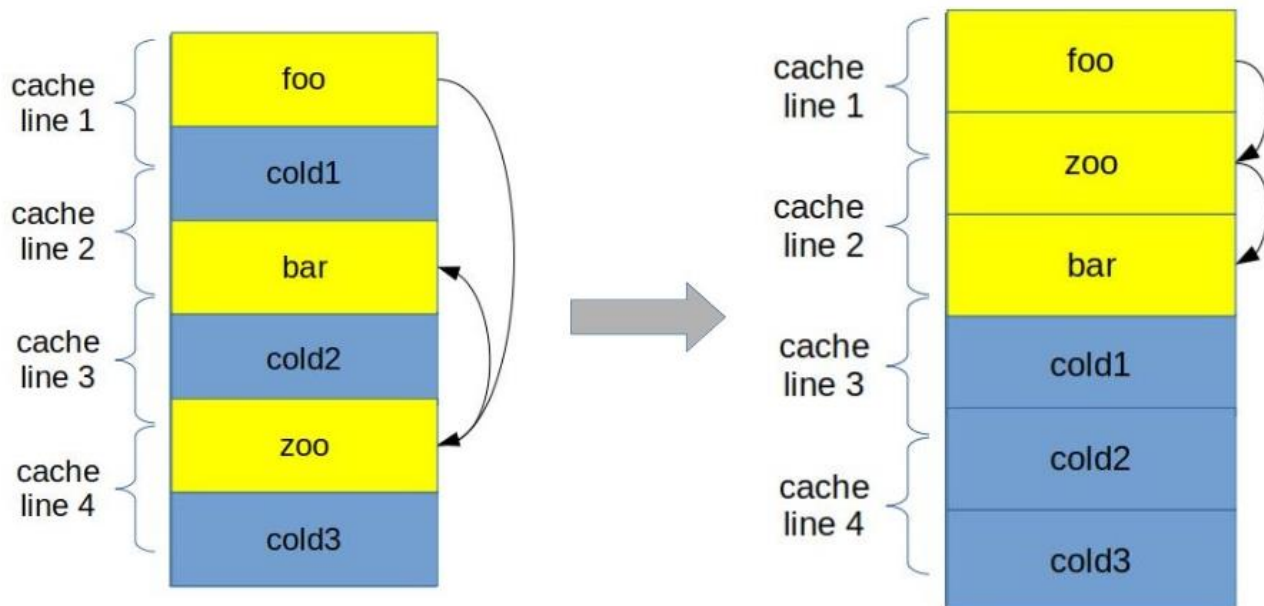


Рисунок 8 – Принцип работы группировки функций

Она применяется преимущественно, когда в коде есть множество мелких функций с горячим кодом, поскольку их объединение позволяет целиком забивать линию кэш-памяти и обрабатывать одновременно несколько функций, не затрачивая время на каждую отдельно. Это схоже с методом выравнивания базовых блоков и позволяет повысить локальность функций.

Данный метод оптимизации может как значительно ускорить работу программы, так и замедлить, поскольку в таких случаях, как, например, многократные вызовы одной и той же функции, вместо того чтобы вызывать только блок с функцией, программе придётся забивать часть линии кэш-памяти куском другой функции. Это будет не оптимально с точки зрения временной и пространственной локальностей, а также в целом с точки зрения рационального использования кэш-памяти [8], так как кэш-линию периодически будет забивать неиспользуемый кусок кода.

Однако этот метод может дать ощутимый прирост, когда используется множество мелких функций, которые могут без труда вместе поместиться в кэш-линию.

## 2 Профилирование

Применение всех вышеперечисленных оптимизаций может стать проблемой в связи с обширным количеством кода в современных программах, возникает новая проблема. Она состоит из вычисления порядка вызова и поведения блоков программы.

Однако, существует возможность предсказать хотя бы частично данные для её решения. Для этого необходимо собирать информацию о программе в процессе её работы. Эти данные называются профилем программы, а сам сбор данных о ней профилированием [23]. Как правило, сбор профиля проводится на бинарных файлах. В качестве примера можно упомянуть различные профилировщики, это программы, которые, соответственно, выполняют профилирование. Самые наиболее часто встречаемые профилировщики это:

- Dynamorio
- Perf
- LLVM-Bolt

Все они занимаются динамическим сбором информации, поскольку делают это в процессе её работы.

Это позволяет сильно упростить понимание использования программой тех или иных блоков кода, а, следовательно, и возможность применения упомянутых выше оптимизаций. В связи с этим целью работы стал анализ методов оптимизации размещения инструкций машинного кода после предварительного профилирования.

### 2.1 LLVM

Для понимания такого инструмента как LLVM-Bolt необходимо сначала разобраться с LLVM [16]. Это тоже неплохой инструмент, хоть и для других видов оптимизаций, на основе которого зародился LLVM-Bolt.

В начале своего пути аббревиатура LLVM расшифровывалась как “Low Level Virtual Machine”. Тем не менее, на данный момент этот проект значительно отошёл от понятия виртуальных машин и на данный момент является инструментом для создания компиляторов. В связи с этим, LLVM уже явля-

ется именем нарицательным. Изначально данный программный продукт создавался с целью разработки промежуточного представления программ, которое, в свою очередь, позволяло бы значительно оптимизировать приложение в процессе его жизни с момента запуска. В основе лежало создание представления, которое хорошо подходило бы как под статическую, так и под динамическую компиляции. В данном случае наиболее интересна будет именно вторая, поскольку она получает на выходе более быстрый машинный код в процессе работы, основываясь на профилях, которые продолжает собирать. На рис. 9 проиллюстрирована архитектура старого LLVM [10].

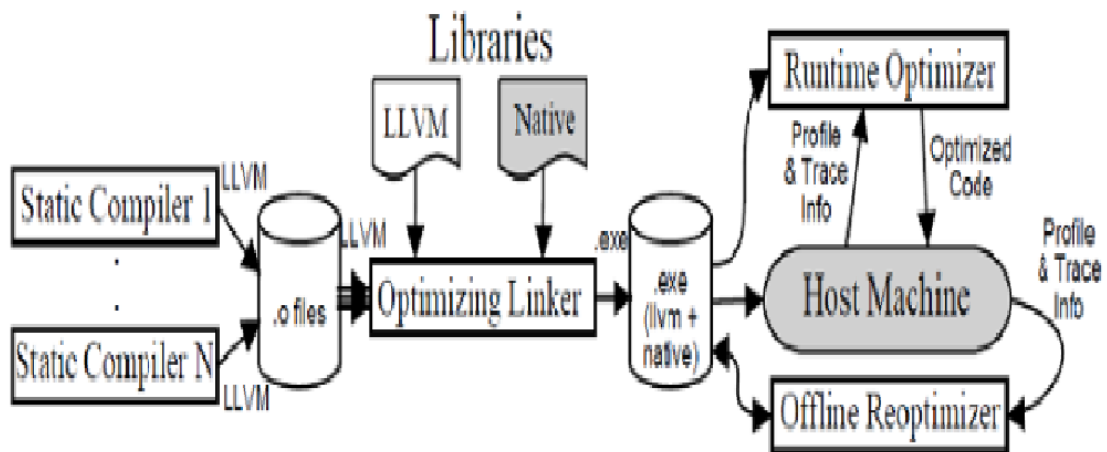


Рисунок 9 – Архитектура изначального LLVM

Тем не менее, на данный момент времени работает несколько не по этому принципу [5]. В результате своего развития LLVM переводился в бэк-энд для компилятора GCC [24]. Таким образом упор был сделан на часть, которая занималась оптимизацией программ и переводила код из промежуточного представления в простой машинный код. Позже появился проект под названием Clang. На его основе получилась комбинация фронт-энда Clang [7] и бэк-энда LLVM, так называемый Clang/LLVM [15]. На данный момент этот компилятор является одним из наиболее популярных в мире. У данного проекта относительно GCC [17] имеется много плюсов. К примеру, в него проще добавлять статические проверки в компилятор для каждой новой ошибки, а ещё он поддерживает операционную систему Windows. В процессе

развития на основе LLVM созданы многие инструменты [19], к примеру LLVM Bolt.

В процессе работы LLVM используется неоднократно [3], поскольку именно этот инструмент даёт возможность регулировать различные параметры программы, в том числе и применять профилирование к программам. Это получается за счёт того, что LLVM обрабатывает промежуточное представление [12]. Это является важной вещью, поскольку именно от него зависит связь анализаторов исходного кода, в результате работы которых можно получить IR [20], и генераторов выполняемого кода, что даёт возможность проекту анализировать программы независимо от языка программирования и создавать код под различные архитектуры. Промежуточное представление как правило бывает двух форматов: бинарный формат с расширением bc и формат текстовой сборки ll. На рис. 10 можно увидеть пример текстовой сборки IR с пометками без анализа и оптимизаций для тестового файла. Данный файл можно получить с помощью компилятора Clang.

```

1. int main() {
2.   int a = 67;
3.   int b;
4.   if (a > 38)
5.     b = a + 59;
6.   return 0;
}

target datalayout = "e-m:e-p270:32:32-
target triple = "x86_64-unknown-linux-

; Function Attrs: noinline nounwind op
define dso_local i32 @main() #0 {
1. { entry:
2.   %retval = alloca i32, align 4
3.   %a = alloca i32, align 4
4.   %b = alloca i32, align 4
5.   store i32 0, i32* %retval, align 4
6.   store i32 67, i32* %a, align 4
7.   %0 = load i32, i32* %a, align 4
8.   %cmp = icmp sgt i32 %0, 38
9.   br i1 %cmp, label %if.then, label %if.end

if.then:
10.  %1 = load i32, i32* %a, align 4
11.  %add = add nsw i32 %1, 59
12.  store i32 %add, i32* %b, align 4
13.  br label %if.end

if.end:
14.  ret i32 0
}

attributes #0 = { noinline nounwind op

```

Рисунок 10 – Текстовая сборка IR

Каждый подобный файл называется модулем. Он содержит в себе последовательность функций. Каждая из них в свою очередь имеет набор групп базовых блоков, а те уже содержат в себе инструкции. В примере на рис. 10 можно увидеть программу с единственной функцией `main`. Так же прекрасно видно что она в себе содержит три базовых блока, помимо основного с именем `@main` имеются так же `if.then` и `if.end`. Переход из одного блока в другой обычно знаменуется так называемым терминатором, инструкцией, которая, находясь в конце блока, непосредственно указывает на него. Поскольку в примере находится условное ветвление, то первый блок может вести как во второй так и в третий, в зависимости от условия, а следовательно там указаны проверка и оба перехода. Сравнивая программу-пример и полученный в результате IR файл можно расшифровать некоторые из инструкций, к примеру `alloc` отвечает за выделение памяти под переменные в стеке, `store` позволяет распоряжаться содержимым этих переменных, `load` является инструкцией для чтения ячеек из стека, а за сложение отвечает `add nsw`. Здесь явно видно, что инструкции `load` и `store` лишние, в связи с чем они убираются даже при стандартной оптимизации.

Механизм преобразований, оптимизации и анализа над промежуточными представлениями реализован при помощи проходов. В результате их работы перебираются все необходимые части, такие как функции, циклы, модули, после чего те совершают необходимые действия. За планировку и регистрацию проходов отвечает диспетчер проходов. Он так же регистрирует зависимости между проходами и их выполнение. В LLVM всё это можно запустить вместе с выводом IR файла при помощи модульного оптимизатора и анализатора. Важной особенностью LLVM API является возможность регистрации собственных проходов, что в очередной раз показывает пользу и плюсы данной технологии.

## 2.2 LLVM Bolt

LLVM-Bolt [4] является основным инструментом, используемым в работе, поскольку он позволяет непосредственно заниматься оптимизациями

размещения машинных инструкций. Его история начинается с разработки компанией Facebook в 2018 году. Основной целью изначально и являлись как раз оптимизации при помощи изменения порядка вызова функций. Обычные компиляторы таким не занимаются, поскольку оптимизированные таким методом программы могут как улучшить свои характеристики, так и сделать их значительно хуже. Чтобы предотвратить непредсказуемость этих оптимизаций, желательно проводить профилирование программ, которые потом подвергнутся обработке. Следовательно, сначала необходимо запустить приложение, затем собрать различную информацию, после чего уже применить подходящие оптимизации.

Всё это возможно, поскольку профилирование позволяет выделить так называемые горячие участки кода. Это такие участки, которые вызываются наиболее часто при работе программы. После их разметки происходит перестановка блоков машинного кода, как правило это бывает по принципу от большего к меньшему, что в результате может приводить к значительному повышению производительности, особенно заметному у больших приложений.

LLVM-Bolt использовался во многих компаниях по разработке программного обеспечения, а так же проектах с открытым исходным кодом для оптимизации каких-либо приложений. В качестве примера можно привести следующие проекты:

1. Chromium
2. Firefox
3. MySQL
4. PostgreSQL

Chromium это довольно известный и много где используемый проект веб браузера с открытым исходным кодом. Он использует LLVM Bolt для оптимизации производительности механизма работы браузера. По данным разработчиков, дополнение проекта данной оптимизацией смогло повысить общую производительность на 10%.

Firefox же использует данный инструмент в браузере Mozilla для оптимизации компонентов в которых использовался язык Rust. Это привело к улучшению скорости загрузки приложения на 17%.

LLVM Bolt может быть полезен не только в разработке браузеров. Ярким примером этого является MySQL. Используемые там оптимизации позволили значительно ускорить работу запросов, использующих сложные соединения. Как и говорилось ранее это работает, поскольку оптимизации размещения машинного кода лучше всего подходят для крупных проектов с большим количеством соединений.

Ещё одна система управления реляционными базами данных PostgreSQL. Однако, в отличии от предыдущей, она смогла уменьшить количество промахов кэша процессора, снизив при этом его загрузку при выполнении запросов на 30%.

Помимо этого, LLVM Bolt ещё используется и в других коммерческих и открытых проектах для оптимизации. Это происходит потому, что его легко применить и интегрировать в существующие рабочие процессы разработки. Так же это является эффективной оптимизацией кода и довольно гибким инструментом в оптимизации приложений.

Работа данного инструмента основана на графе работы программы. Он показывает существующие блоки кода и то, как они связаны между собой. Их связи соответственно имеют вес, который определяется при помощи количества переходов по ним. Эта информация собирается вместе с остальной во время профилирования. Граф же создаётся и изменяется во время оптимизации программы, переставляя блоки наиболее оптимальным из полученной информации способом.

Для его оптимизации используется алгоритм HFSort, содержащий в себе два других алгоритма. Благодаря этому имеется возможность выбрать наилучший из них. Графы состояются из блоков кода, которые считаются вершинами, и связей в виде рёбер между ними. Таким образом составленный граф имеет одну или несколько пронумерованных вершин, не все из которых



должны быть обязательно связаны. Такое представление может быть значительно проще для понимания, ведь его можно визуализировать. Пример такого графа можно увидеть на рис. 11.

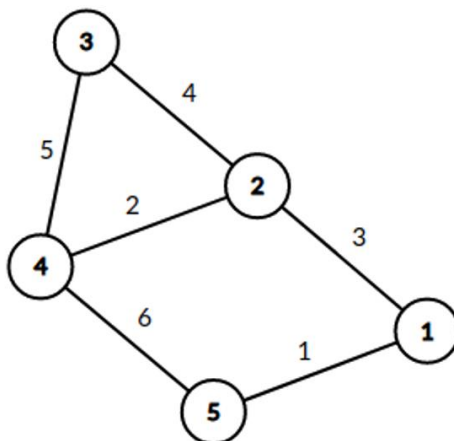


Рисунок 11 – Взвешенный граф.

При создании графа вызовов используется алгоритм Strongly Connected Components, поскольку, помимо возможности генерировать граф вызовов функций в приложении, у него так же имеется возможность выявления у функций рекурсивных зависимостей.

Данный алгоритм работает путём анализа направленного графа, который образуется в результате вызова функций в приложении. Каждая из них будет узлом, а их вызовы являются направленными рёбрами. После чего алгоритм определяет сильно связанные компоненты в полученном графе. В данном случае это группы функций с циклическими зависимостями, которые вызываются рекурсивно. Алгоритм LLVM Bolt использует о них информацию для распределения порядка их расположения в бинарном графе.

Это приводит к тому, что рекурсивные зависимости тоже учитываются в графе, что в свою очередь ведёт к более точной генерации графа вызовов. Всё это позволяет ещё сильнее оптимизировать компоновку кода и повышению производительности за счёт уменьшения промахов кэша и количества обращений к инструкциям для часто используемых функций.

Для сортировки таких компонентов в LLVM Bolt используется алгоритм топологической сортировки [22]. Он сортирует вершины в направленном графе в соответствии с зависимостью между ними так, что вершина появляется исключительно до появления её зависимостей. Такой способ сортировки позволяет сразу влиять на расположение блоков в выходном бинарном файле. Это позволяет улучшить локальность ссылок и уменьшить количество переборов инструкций при выполнении приложения.

Основной задачей топологической сортировки является указание такого линейного порядка на вершинах графа, что любое его ребро ведёт от вершины с меньшим номером к вершине с большим номером [21]. Применительно к LLVM Bolt это можно интерпретировать как выставление блокам кода номеров, которые определяют в первую очередь порядок считывания блока, а затем уже с применением топологической сортировки выстраивают наиболее оптимальный с точки зрения соединения функций маршрут.

LLVM Bolt для оптимизации графов использует HFSort, который применяет 2 алгоритма распределения инструкций по графу для выбора лучшего из них. В таких графах вершинами будут блоки кода, а в качестве рёбер выступают вызовы одним блоком другого. Первым алгоритмом является эвристика Петтиса и Хэнсена, а вторым кластеризация цепи вызова

Суть эвристики Петтиса и Хэнсена сводится к следующим пунктам:

1. Выбрать ребро с наибольшим весом
2. Проверить все возможные соединения вершин, которые соединены этим ребром, путём их переворачивания
3. Объединить выбранные вершины по наиболее нагруженному варианту перебора
4. Повторять до тех пор, пока в графе есть рёбра

Пример работы алгоритма можно увидеть на рис. 12:

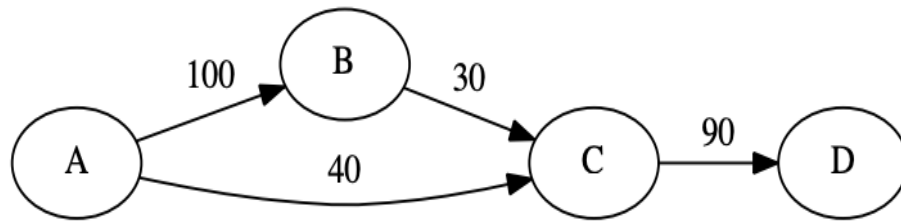


Рисунок 12 – Изначальный граф

По нему явно видно, что самое большое ребро находится между вершинами А и В. Поэтому их необходимо объединить. По итогу существует всего один вариант: АВ соединение, поскольку переворот блоков ничего не меняет. Вес между новой вершиной А;В и вершиной С будет получен из суммы рёбер А-С и В-С.

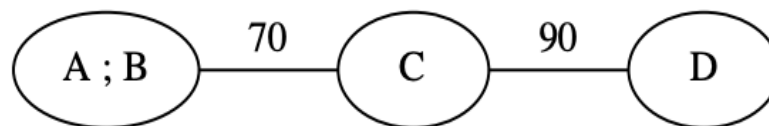


Рисунок 13 – Граф в результате первого шага

В данном случае на рис. 13 наибольшим ребром является C-D, а значит как и в предыдущем шаге, необходимо объединить эти вершины.

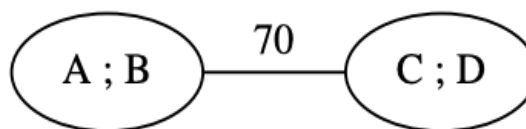


Рисунок 14 – Граф в результате второго шага

Оставшееся ребро на рис. 14 и будет выбрано для дальнейшей работы алгоритма. На основе данных вершин осталось определить, вес какого из соединений взять, всего получается 4 варианта соединений:

AB – CD – соединение В и С, вес ребра 30.

AB – DC – соединение В и D, вес ребра 0 (прямого соединения нет).

BA – CD – соединение А и С, вес ребра 40.

BA – DC – соединение А и D, вес ребра 0.

Здесь опять необходимо действовать по принципу выбора наибольшего веса, в данном случае таковым обладает соединение BA – CD, поэтому оно будет использовано для распределения инструкций. Результат можно увидеть на рис. 15.

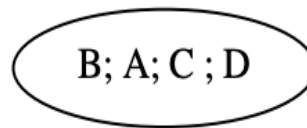


Рисунок 15 – Конечный граф

После вычисления эвристики Пэттиса и Хэнсона, необходимо так же рассмотреть кластеризацию цепи вызова. Отчасти эти два решения могут казаться очень схожими, но алгоритм кластеризации следующий:

1. Разделить все вершины на кластеры.
2. Выбрать вершину, к которой примыкает наибольшее по весу ребро.
3. Соединить кластеры в один.
4. Повторять со второго шага, пока не останется один кластер.

Пример работы алгоритма:

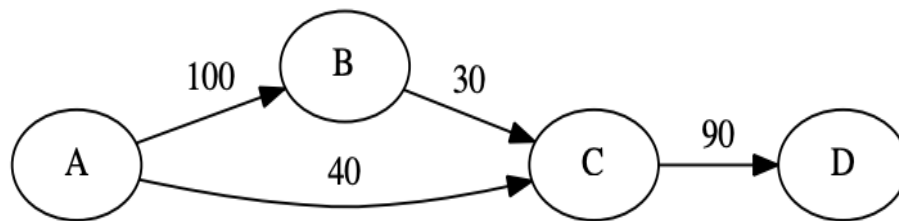


Рисунок 16 – Изначальный граф

Для примера возьмём тот же граф на рис. 16, что и до этого, это позволит показать различие способов более наглядно. Для начала выберем наибольшее входное ребро, в данном случае это ребро с весом 100. После того как оно найдено, необходимо выбрать вершины, с которыми оно соприкасается. Эти вершины, в данном случае A и B будут объединены в кластер, как на рис. 17.

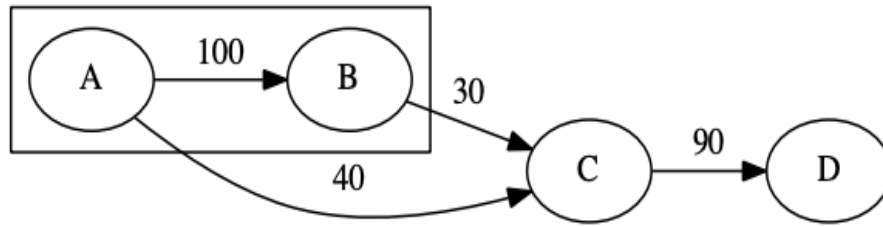


Рисунок 17 – Первый шаг кластеризации

Теперь наибольшее по весу ребро является входным для D, поэтому объединяем в кластер C и D, как на рис. 18.

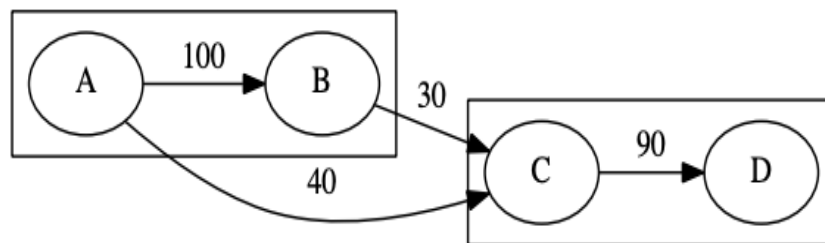


Рисунок 18 – Второй шаг кластеризации

В результате наибольшим входящим ребром является ребро с весом 40 в кластер CD. В связи с этим необходимо соединить его с кластером AB. Таким образом ответом является ABCD

Далее идёт непосредственно выбор лучшего из этих решений.

Поскольку результат алгоритмов различается, необходимо выбрать наилучшее решение. Для этого можно посчитать совокупную стоимость прыжков каждого из вариантов.

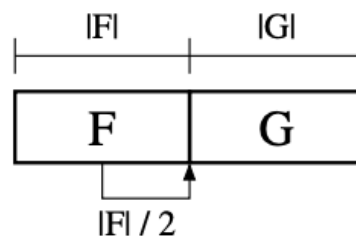


Рисунок 19 – Стоимость соединений

Стоимость прямого перехода в соседний блок будет равна F - размер функции, который у всех является одинаковым. Его необходимо поделить на

2 т.к. изначально принимается то, что вызов происходит где-то в середине функции и ведёт в самое начало, как указано на рис. 19. Далее нужно умножить стоимость такого перехода на количество переходов по нему в результате профилирования по следующей формуле:

$$cost = \frac{F * frequency}{2}$$

Чтобы вычислить стоимость промахов, необходимо провести обратное соединение, оно показано на рис. 20.

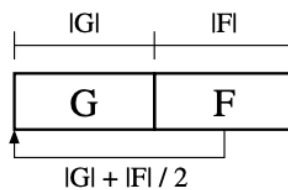


Рисунок 20 – Стоимость промахов

Поскольку размер функций принимается за одинаковый, то он вместе с количеством использований умножается на полтора по следующей формуле:

$$cost = F * frequency * 1.5$$

После этих расчётов нужно посчитать стоимость полученных в их результате распределений. Принципы по которым построены формулы расчётов указаны на рис. 21 и рис. 22.

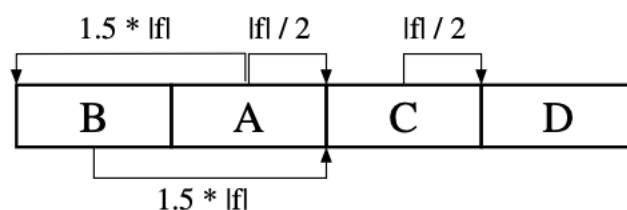
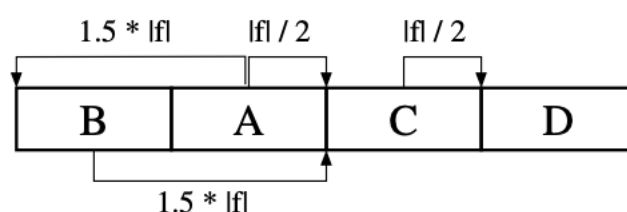


Рисунок 21 – РН соединение

$$cost_1 = 100 * 1.5 * F + 40 * 0.5 * F + 30 * 1.5 * F + 90 * 0.5 * F$$

$$cost_1 = (150 + 20 + 45 + 45) * F = 260 * F$$



### Рисунок 22 – C3 распределение

$$cost_2 = 100 * 0.5 * F + 40 * 1.5 * F + 30 * 0.5 * F + 90 * 0.5 * F$$

$$cost_2 = (50 + 60 + 15 + 45) * F = 170 * F$$

В этом конкретном случае стоимость второго распределения оказалась меньше ( $170 < 260$ ), поэтому выбор падает на алгоритм кластеризации цепи, как на самый эффективный. В зависимости от входных данных, оптимизация исключительно по второму алгоритму не всегда эффективна, поэтому необходимо считать оба варианта и сравнивать их.

Итого весь цикл работы инструмента LLVM Bolt выглядит примерно следующим образом:

1. Сбор данных профиля
2. Генерация файлов компоновки
3. Анализ двоичных файлов
4. Генерация графика
5. Оптимизированная компоновка
6. Перераспределение кода
7. Тестирование

Сбор данных профиля генерируется при помощи любого профилировщика во время выполнения приложения [13]. Далее генерируются файлы компоновки PDL для управления процессом оптимизации на основе полученного профиля. Затем LLVM Bolt анализирует двоичный файл на наиболее влияющие на производительность функции. После чего создаётся и сортируется граф вызовов. Оптимизированная компоновка проходит на основе данных из PDL файла. На следующем этапе алгоритм LLVM Bolt переупорядочивает код не изменяя исходный файл, и на основе этого генерирует новый бинарный файл. На стадии тестирования сравнивается результат изначальной программы и полученной в результате работы алгоритмов.

### 2.3 Dynamorio

Основное назначение проекта Dynamorio [6] это создание виртуальной машины, которая могла бы изменять код в процессе работы программы. Это

очень похоже на начальный концепт LLVM. В процессе изменения кода пока программа работает, Dynamorіo не обязательно изменять весь набор инструкций, поскольку имеется возможность изменять куски кода. Это подразумевает так же необходимость программы для профилирования, ведь данный проект изменяет код программы динамически. Для этого у него имеется арсенал для профилирования и анализа программы, на основе которых он уже может оптимизировать проект путём изменения программы в реальном времени. Оригинальный код при этом не меняется, ведь Dynamorіo создаёт его копию с возможностью модификации непосредственно машинных инструкций.

Для данной работы Dynamorіo представляет интерес в первую очередь в качестве профилировщика. Для профилирования при помощи Dynamorіo использовался Docker.

Используя данную программу, можно получить отдельный лог файл, в котором содержатся операции чтения и записи по порядку в следующем формате:

- Адрес инструкции
- Название операции
- Размер данных
- Адрес памяти

Пример такого файла можно увидеть на рис. 23.



```

root@4a5a69ad694c:/dynamorio# ./bin64/drrun -c samples/bin64/libmemtrace_x86_text.so -- echo
Client memtrace is running
Data file /dynamorio/samples/bin64/memtrace.echo.00034.0000.log created

Instrumentation results:
  saw 64118 memory references

root@4a5a69ad694c:/dynamorio# cat samples/bin64/memtrace.echo.00034.0000.log
Format: <instr address>,<(r)ead/(w)rite>,<data size>,<data address>
0x7f8b8ea51103,w,8,0x7ffcad02a878
0x7f8b8ea51df4,w,8,0x7ffcad02a870
0x7f8b8ea51df8,w,8,0x7ffcad02a868
0x7f8b8ea51dfd,w,8,0x7ffcad02a860
0x7f8b8ea51dff,w,8,0x7ffcad02a858
0x7f8b8ea51e01,w,8,0x7ffcad02a850
0x7f8b8ea51e03,w,8,0x7ffcad02a848
0x7f8b8ea51e18,w,8,0x7f8b8ea7d5e0
0x7f8b8ea51e1f,r,8,0x7f8b8ea7de68
0x7f8b8ea51e29,r,8,0x7f8b8ea7e000
0x7f8b8ea51e30,w,8,0x7f8b8ea7e9f8
0x7f8b8ea51e37,w,8,0x7f8b8ea7e9e8
0x7f8b8ea51ea5,w,8,0x7f8b8ea7ea98
0x7f8b8ea51ea9,r,8,0x7f8b8ea7de78
0x7f8b8ea51ea5,w,8,0x7f8b8ea7ea48
0x7f8b8ea51ea9,r,8,0x7f8b8ea7de88

```

Рисунок 23 – Пример лог файла

Помимо этого, при помощи Dynamorio можно составить профиль для кэша программы. Это позволяет отслеживать промахи и попадания кэша. Его пример можно увидеть на рис. 24.

```

root@4a5a69ad694c:/dynamorio# ./bin64/drrun -t drcachesim -- echo
— <application exited with code 0> —
Cache simulation results:
Core #0 (1 thread(s))
  L1I stats:
    Hits:                215382
    Misses:               1166
    Compulsory misses:    1135
    Invalidations:        0
    Miss rate:            0.54%
  L1D stats:
    Hits:                62107
    Misses:               2052
    Compulsory misses:    2835
    Invalidations:        0
    Prefetch hits:        252
    Prefetch misses:      1800
    Miss rate:            3.20%
Core #1 (0 thread(s))
Core #2 (0 thread(s))
Core #3 (0 thread(s))
LL stats:
  Hits:                  601
  Misses:                 2617
  Compulsory misses:     3970
  Invalidations:          0
  Prefetch hits:          447
  Prefetch misses:       1353
  Local miss rate:        81.32%
  Child hits:             277741
  Total miss rate:        0.93%
root@4a5a69ad694c:/dynamorio#

```

Рисунок 24 – Профиль кэша программы

Ещё одной полезной функцией является возможность обнаруживать утечки памяти. Для этого в Dynamorio используется Dr. Memory. Пример такого профиля изображён на рис. 25.

```
root@4a5a69ad694c:/dynamorio# ./bin64/drrun -t drmemory -- echo
~Dr.M~ Dr. Memory version 2.5.19012
~Dr.M~ ERROR: Failed to find "main" for limiting memory dump
~Dr.M~ WARNING: application is missing line number information.

~Dr.M~
~Dr.M~ NO ERRORS FOUND:
~Dr.M~      0 unique,      0 total unaddressable access(es)
~Dr.M~      0 unique,      0 total uninitialized access(es)
~Dr.M~      0 unique,      0 total invalid heap argument(s)
~Dr.M~      0 unique,      0 total warning(s)
~Dr.M~      0 unique,      0 total,      0 byte(s) of leak(s)
~Dr.M~      0 unique,      0 total,      0 byte(s) of possible leak(s)
~Dr.M~ ERRORS IGNORED:
~Dr.M~      15 unique,      18 total,      6822 byte(s) of still-reachable allocation(s)
~Dr.M~      (re-run with "-show_reachable" for details)
~Dr.M~ Details: /dynamorio/drmemory/drmemory/logs/DrMemory-echo.38.000/results.txt
root@4a5a69ad694c:/dynamorio#
```

Рисунок 25 – Профиль Dr.Memory

Dynamorio является неплохим инструментом для профилирования программ, поскольку позволяет проводить профилирование и получать всю необходимую информацию о её работе. Но несмотря на то, что данный проект изначально разрабатывался для непосредственного вмешательства в код в процессе его выполнения для достижения лучших параметров его работы на основе профилирования, он значительно отличается от задачи дипломной работы и плохо подходит для её реализации, поскольку оптимизации машинных инструкций подразумевают обработку до запуска программы. Однако его профилировщик может быть подходящим для исследований в рамках предварительного профилирования, поскольку с его помощью можно получить отдельно информацию по различным параметрам начиная с операций чтения и записи и заканчивая кэш памятью.

## 2.4 Perf

Ещё одним неплохим инструментом для профилирования является Perf [18]. Этот инструмент существует исключительно на линуксе.

Неплохой особенностью является то, что Perf анализирует помимо проблем производительности каждого потока ещё и проблемы производительности ядра. Начинался он как, своего рода инструмент для счётчиков производительности в операционной системе типа Linux. Позже расширился

и до возможностей трассировки. На данный момент он способен выполнять легковесное профилирование.

Под счётчиками производительности понимаются аппаратные регистры процессора. Они подсвечивают аппаратные события, промахи в кэше и прочую информацию, которая в итоге ложится в основу создаваемого этим инструментом профиля. В итоге это выливается в возможность динамически выявлять и отслеживать точки трассировки, являющиеся “горячими”.

В свою очередь под точками трассировки понимаются места, где код разделяется на логические блоки, к примеру, условные ветвления. Естественно, такая возможность вызывает некоторые накладные расходы, но на практике они оказываются довольно не существенными. При помощи некоторых фреймворков, таких как kprobes и uprobes, perf может создавать точки для динамической трассировки ядра.

Профиль в результате работы perf изначально является несовместимым с вышеуказанным инструментом bolt, но это исправляет его часть perf2bolt, которая bolt в себя включает. В результате её работы можно получить возможность преобразования data файла к формату, который подходит для bolt. Пример такого профиля, который был переведён из perf в формат для bolt можно увидеть на рис. 26.

```

root@366a81aae1c4:/tests# perf record -e cycles:u -m 4 -o perf.data -- echo hi
hi
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.001 MB perf.data (2 samples) ]
root@366a81aae1c4:/tests# perf2bolt -p perf.data -o perf.fdata -ignore-build-id /usr/bin/echo
BOLT-INFO: shared object or position-independent executable detected
PERF2BOLT: Starting data aggregation job for perf.data
PERF2BOLT: spawning perf job to read branch events
PERF2BOLT: spawning perf job to read mem events
PERF2BOLT: spawning perf job to read process events
PERF2BOLT: spawning perf job to read task events
BOLT-INFO: Target architecture: x86_64
BOLT-INFO: BOLT version: 88c70afe9d388ad430cc150cc158641701397f70
BOLT-INFO: first alloc address is 0x0
BOLT-INFO: creating new program header table at address 0x200000, offset 0x200000
BOLT-INFO: disabling -align-macro-fusion in non-relocation mode
BOLT-INFO: enabling lite mode
BOLT-INFO: pre-processing profile using perf data aggregator
BOLT-INFO: binary build-id is: 714b557112010bbcd04b0e5e6efc1b106166733c
PERF2BOLT: spawning perf job to read buildid list
PERF2BOLT-ERROR: failed to match build-id from perf output. This indicates the input binary supplied for data aggregation is not the
same recorded by perf when collecting profiling data, or there were no samples recorded for the binary. Use -ignore-build-id option
to override.
PERF2BOLT: waiting for perf mmap events collection to finish...
PERF2BOLT: parsing perf-script mmap events output
PERF2BOLT: waiting for perf task events collection to finish...
PERF2BOLT: parsing perf-script task events output
PERF2BOLT: input binary is associated with 1 PID(s)
PERF2BOLT: waiting for perf events collection to finish...
PERF2BOLT: parse branch events...
PERF2BOLT: read 2 samples and 0 LBR entries
PERF2BOLT-WARNING: all recorded samples for this binary lack LBR. Record profile with perf record -j any or run perf2bolt in no-LBR
mode with -nl (the performance improvement in -nl mode may be limited)
PERF2BOLT: traces mismatching disassembled function contents: 0
PERF2BOLT: out of range traces involving unknown regions: 0
BOLT-WARNING: Ignored 0 functions due to cold fragments.
BOLT-WARNING: Running parallel work of 0 estimated cost, will switch to trivial scheduling.
PERF2BOLT: processing branch events...
PERF2BOLT: wrote 0 objects and 0 memory objects to perf.fdata
root@366a81aae1c4:/tests# ls
file perf.data perf.fdata
root@366a81aae1c4:/tests#

```

Рисунок 26 – Результат работы perf2bolt

В целом использование данного профилировщика для целей работы подходит уже больше, поскольку имеется возможность взаимодействия с LLVM-Bolt. Основная его часть сосредоточена на сборе информации о процессоре и работы с ней.

К плюсам данного инструмента можно причислить удобство использования, оно выходит из того, что сразу идёт в комплекте с операционными системами типа Linux. Несмотря на это его профили являются довольно подробными, что может пригодиться с учётом использования предварительного профилирования.

Главным минусом данного профилировщика является его сосредоточенность на системе Linux, это создаёт проблемы по его использованию на других системах. Ещё к минусам можно добавить необходимость перевода получаемых профилей в другой формат, что в свою очередь является невыгодным при динамической обработке машинного кода, а, следовательно, и

затормаживает обработку программы при оптимизациях на основе профилирования в целом.

### 3 Исследование и разработка тестовых файлов

Основным инструментом работы был выбран LLVM Bolt, поскольку это единственный удобный инструмент для оптимизаций размещения инструкций машинного кода.

#### 3.1 Настройка и работа с LLVM Bolt

Поскольку LLVM Bolt является проектом с кодом в открытом доступе, а также это единственный инструмент для работы с возможностью перемещать программные блоки, то выбор пал именно на него.

Для его установки есть различные методы, к примеру его можно установить через git репозиторий, затем при помощи make собрать LLVM и LLVM Bolt. Чтобы не ожидать достаточно длительное время, эту сборку можно запустить в несколько параллельных процессов. Наиболее быстро это получится при установке максимального количества доступных ядер. В результате сборки можно будет обнаружить LLVM-Bolt, который имеется возможность добавлять к исполняемым файлам.

Так же существует альтернативный вариант установки при помощи инструмента для контейнеризации Docker [11]. Репозиторий LLVM Bolt [4] содержит так же и Dockerfile, при помощи которого не составит труда собрать готовый образ.

После того, как будет создан образ, как в примере на рис. 27, необходимо создать и запустить рабочий контейнер, как изображено на рис. 28.

```
root@augiro-VirtualBox:/home/augiro/Test/build/bin# docker images
REPOSITORY          TAG             IMAGE ID        CREATED         SIZE
llvm_bolt            latest         c13d853d8191   2 days ago     134MB
llvm-bolt            latest         5a97b23b49c0   2 days ago     134MB
snowstep/llvm        latest         69519471b948   2 days ago     1.16GB
root@augiro-VirtualBox:/home/augiro/Test/build/bin#
```

Рисунок 27 – Скачанный образ в Docker

```
root@augiro-VirtualBox:/home/augiro/Test/build/bin# docker ps
CONTAINER ID   IMAGE          COMMAND         CREATED        STATUS        PORTS
b455fe06fbf2   c13d853d8191   "/bin/bash"    7 hours ago   Up 7 hours
225621fdb19d   5a97b23b49c0   "/bin/bash"    8 hours ago   Up 8 hours
root@augiro-VirtualBox:/home/augiro/Test/build/bin# docker run -it 5a97b23b49c0
root@f543e5a02eb8:/#
```

Рисунок 28 – Контейнер Docker



Далее для работы использовался интерактивный режим.

В итоге пользователь получает доступ к виртуальной среде, которая содержит в себе всё необходимое для использования LLVM-Bolt. В этом можно убедиться, использовав команду, которую можно увидеть на рис. 29. Как понятно из самой команды, она показывает версию самой утилиты, а также дополнительную информацию.

```
root@f543e5a02eb8:/# llvm-bolt --version
LLVM (http://llvm.org/):
  LLVM version 14.0.0git
  Optimized build with assertions.
  Default target: x86_64-unknown-linux-gnu
  Host CPU: skylake

BOLT revision 56ff67ccd90702d87a44c7e60abe3c4986855493
Registered Targets:
  aarch64      - AArch64 (little endian)
  aarch64_32   - AArch64 (little endian ILP32)
  aarch64_be   - AArch64 (big endian)
  arm64        - ARM64 (little endian)
  arm64_32     - ARM64 (little endian ILP32)
  x86          - 32-bit X86: Pentium-Pro and above
  x86-64       - 64-bit X86: EM64T and AMD64
root@f543e5a02eb8:/#
```

Рисунок 29 – Информация о LLVM-Bolt

После того как окружение настроено, необходимо собрать профили тестируемых программ. Для этого необходимо их скомпилировать при помощи clang, чтобы получить бинарные файлы в промежуточном представлении. Затем по ним уже будут выстраиваться профили.

Далее необходимо создать файл с инструментами сбора данных о работе. Это так же создаёт профиль для оптимизируемой программы. После этого можно уже запускать непосредственно оптимизацию при помощи LLVM Bolt. Для этого необходимо передать в команду профиль и бинарный файл, содержащий промежуточное представление. Процесс получения профиля из бинарного файла изображён на рис. 30.

```

root@b455fe06fbf2:/test# llvm-bolt -instrument Source -o Mars1
BOLT-INFO: Target architecture: x86_64
BOLT-INFO: BOLT version: 56ff67ccd90702d87a44c7e60abe3c4986855493
BOLT-INFO: first alloc address is 0x400000
BOLT-INFO: creating new program header table at address 0x600000, offset 0x200000
BOLT-INFO: enabling relocation mode
BOLT-INFO: forcing -jump-tables=move for instrumentation
BOLT-INFO: enabling -align-macro-fusion=all since no profile was specified
BOLT-INFO: enabling lite mode
BOLT-WARNING: Ignored 0 functions due to cold fragments.
BOLT-INSTRUMENTER: Number of indirect call site descriptors: 3
BOLT-INSTRUMENTER: Number of indirect call target descriptors: 14
BOLT-INSTRUMENTER: Number of function descriptors: 14
BOLT-INSTRUMENTER: Number of branch counters: 7
BOLT-INSTRUMENTER: Number of ST leaf node counters: 21
BOLT-INSTRUMENTER: Number of direct call counters: 0
BOLT-INSTRUMENTER: Total number of counters: 28
BOLT-INSTRUMENTER: Total size of counters: 224 bytes (static alloc memory)
BOLT-INSTRUMENTER: Total size of string table emitted: 296 bytes in file
BOLT-INSTRUMENTER: Total size of descriptors: 1956 bytes in file
BOLT-INSTRUMENTER: Profile will be saved to file /tmp/prof.fdata
BOLT-INFO: 0 out of 17 functions in the binary (0.0%) have non-empty execution profile
BOLT-INFO: 62 instructions were shortened
BOLT-INFO: UCE removed 3 blocks and 136 bytes of code.
BOLT-INFO: SCTC: patched 0 tail calls (0 forward) tail calls (0 backward) from a to
is 0 and the number of times CTCs are taken is 0.
BOLT-INFO: output linked against instrumentation runtime library, lib entry point t
BOLT-INFO: clear procedure is 0x80a3e0
BOLT-INFO: setting _end to 0x403df0
BOLT-INFO: patched build-id (flipped last bit)
root@b455fe06fbf2:/test#

```

Рисунок 30 – Получение профиля

В процессе работы LLVM Bolt выводит различные действия, которые производятся для оптимизации. Их можно увидеть на рис. 31.



```

root@b455fe06fbf2:/test# llvm-bolt Source -o Mars1.bolt -data=prof.fdata -reorder-blocks
BOLT-INFO: Target architecture: x86_64
BOLT-INFO: BOLT version: 56ff67ccd90702d87a44c7e60abe3c4986855493
BOLT-INFO: first alloc address is 0x400000
BOLT-INFO: creating new program header table at address 0x600000, offset 0x200000
BOLT-INFO: enabling relocation mode
BOLT-INFO: enabling lite mode
BOLT-INFO: pre-processing profile using branch profile reader
BOLT-WARNING: Ignored 0 functions due to cold fragments.
BOLT-INFO: 8 out of 17 functions in the binary (47.1%) have non-empty execution profile
BOLT-INFO: 2 functions with profile could not be optimized
BOLT-WARNING: 1 (12.5% of all profiled) function have invalid (possibly stale) profile.
BOLT-WARNING: 991 out of 1005 samples in the binary (98.6%) belong to functions with inv
BOLT-INFO: profile for 211 objects was ignored
BOLT-INFO: 0 instructions were shortened
BOLT-INFO: removed 3 empty blocks
BOLT-INFO: basic block reordering modified layout of 1 (4.76%) functions
BOLT-INFO: UCE removed 0 blocks and 0 bytes of code.
BOLT-INFO: 0 Functions were reordered by LoopInversionPass
BOLT-INFO: SCTC: patched 0 tail calls (0 forward) tail calls (0 backward) from a total o
is 0 and the number of times CTCs are taken is 0.
BOLT-INFO: padding code to 0xa00000 to accommodate hot text
BOLT-INFO: setting __end to 0x403df0
BOLT-INFO: setting __hot_start to 0x800000
BOLT-INFO: setting __hot_end to 0x8001fd
BOLT-INFO: patched build-id (flipped last bit)
root@b455fe06fbf2:/test# █

```

Рисунок 31 – Процесс оптимизации

А в результате получится бинарный файл с расширением .bolt. По умолчанию он настроен на наиболее оптимальное размещение программных логических блоков. Это означает что блоки горячего кода стоят перед блоками с холодным кодом.

Сам инструмент имеет некоторые настройки, которые позволяют указать, какие методы оптимизации применять, а какие нет. Это полезно, поскольку в определённых случаях применение некоторых оптимизаций может повлиять в положительную сторону, в то время как другая часть в негативную, что приведёт к меньшему приросту производительности в сумме. Для этого в LLVM Bolt существуют следующие флаги:

- `-reorder-blocks` : Данная опция позволяет переупорядочивать блоки внутри функции на основе заданного критерия. Примером такого использования является флаг `“-reorder-blocks=cache+”`, он упорядочивает блоки машинных инструкций таким образом, что наиболее часто используемые блоки окажутся рядом для мини-

мального перебора инструкций. Иными словами, от него зависит применение метода оптимизации путём распределения базовых блоков, поскольку это и является перестановкой горячего кода.

- `-reorder-functions` : Эта опция даёт возможность переупорядочить функции на основе заданного критерия. Это можно тоже отнести к методу оптимизации распределения базовых блоков, но скорее эта функция необходима для этапа создания графа, поскольку может влиять на приоритет функций. К примеру, `“-reorder-functions=hfsort”` определит приоритет для наиболее часто выполняемых функций для бинарного файла при помощи алгоритма `hfsort`. Всего у неё существует 6 вариантов. Первый из них `none`, при нём данная оптимизация не выполняется, второй `exes-count`, сортирует функции в порядке выполнения, третий уже упомянутый `hfsort`, четвёртый является его улучшенной версией `hfsort+`, пятый `pettis-hansen` напрямую использует упомянутую эвристику, шестой расставляет функции в случайном порядке, а седьмой позволяет установить пользовательский порядок.
- `-split-functions` : Как ясно из названия, эта опция отвечает за разбиение функций на несколько секций. Ярким примером её использования является вариант `“-split-functions=2”`. Он, соответственно, разделяет только крупные функции. Всего данная опция имеет 4 варианта параметра, в случае если она равна 0, то функции не разбиваются, если 1 или 2, то разбиваются только большие функции, по сути выполняются одинаковые действия, это оставлено для исключения ошибок при использовании старой версии, если опция равна 4, то все функции подлежат разбиению.
- `-split-all-cold-functions` : Соответственно этот флаг задаёт необходимость разбиения холодных функций на более мелкие, а так же не имеет параметров.

- `-align-functions` : Следующая опция выравнивает начало каждой функции по заданной границе адресов, это может позволить лучше использовать кэш память, поскольку адреса функций располагаются близко. Это значительно повышает локальность.
- `-layout-order-file` Данная опция позволяет указать любой пользовательский файл порядка раскладки, который указывает как LLVM Bolt должен переупорядочивать функции в двоичном файле. Проще говоря она необходима для пользовательских стилей оптимизации, что даёт возможность расширения функционала без необходимости вмешательства в код оригинального проекта. В качестве аргумента необходимо указать путь к пользовательскому файлу.
- `-instrumentation-file` : Эту опцию LLVM Bolt использует для того, чтобы указать данные профилирования, собранные автономным инструментом профилирования. Это необходимо, чтобы понимать, что необходимо сделать для оптимизации проекта, при этом не снизив изначальную производительность программы. В качестве аргумента у данного флага используется путь к файлу профиля.
- `--function-order` : Данная опция позволяет устанавливать текстовый файл, который содержит в себе упорядоченные функции. Это позволяет производить упорядочивание функций в различных порядках, что может быть полезно для тестирования поиска наиболее оптимального их расположения вручную. Также это можно использовать для генерации порядка функций при помощи алгоритмов, которые не взаимодействуют напрямую с LLVM Bolt.
- `--version` : выводит версию и некоторую информацию о программе.

- `--dyno-stats` : является полезным для получения статистики, которую LLVM Bolt может вывести во время оптимизации.

Статистика содержит в себе следующие пункты:

```

3 : executed forward branches
1 : taken forward branches
2 : executed backward branches
1 : taken backward branches
0 : executed unconditional branches
10 : all function calls
5 : indirect calls
2 : PLT calls
89 : executed instructions
16 : executed load instructions
11 : executed store instructions
0 : taken jump table branches
0 : taken unknown indirect branches
5 : total branches
2 : taken branches
3 : non-taken conditional branches
2 : taken conditional branches
5 : all conditional branches

3 : executed forward branches (=)
0 : taken forward branches (-100.0%)
2 : executed backward branches (=)
1 : taken backward branches (=)
0 : executed unconditional branches (=)
10 : all function calls (=)
5 : indirect calls (=)
2 : PLT calls (=)

```

88 : executed instructions (-1.1%)  
 16 : executed load instructions (=)  
 11 : executed store instructions (=)  
 0 : taken jump table branches (=)  
 0 : taken unknown indirect branches (=)  
 5: total branches (=)  
 1 : taken branches (-50.0%)  
 4 : non-taken conditional branches (+33.3%)  
 1 : taken conditional branches (-50.0%)  
 5 : all conditional branches (=)

Как видно из перечисленных флагов, среди них нет возможности использовать метод оптимизации путём выравнивания базовых блоков.

## **4.2 Проведение тестирования на улучшение производительности**

Для тестирования на улучшение производительности были предварительно разработаны и выбраны следующие программы:

1. Алгоритм шифрования MARS.
2. Рекурсивный алгоритм Фибоначчи в 2 вариантах с простой рекурсией и в 3 функции.

Каждая из них была протестирована не менее чем на 5 вариантах тестовых данных. Для сравнения эффективности различных алгоритмов используются различные флаги инструмента LLVM Bolt. Это позволяет протестировать как каждый метод оптимизации по отдельности для различных случаев, так и при применении всех оптимизаций сразу.

Алгоритм шифрования MARS был выбран потому, что криптографические программы часто работают блоками и используют множество функций. Это позволяет выявить эффективность работы оптимизаций при помощи LLVM-Bolt. Так же именно этот алгоритм довольно прост в понимании и освоении, поскольку является симметричным. Исследование было проведено для его функции шифровки и представлено в таблице 1:

Таблица 1 – Результаты исследований алгоритма шифрования MARS

№ теста	Без оптимизации	Распределение базовых блоков	Разбиение на функции	Группировка функций	Все методы
1	2,39	1,86	1,90	2,05	1,55
2	2,22	1,79	1,93	2,08	1,51
3	1,98	1,80	1,82	2,05	1,56
4	2,23	1,70	2,01	2,06	1,70
5	2,21	1,85	1,94	2,14	1,47

На основе данной таблицы была составлена гистограмма, она представлена на рис. 32.

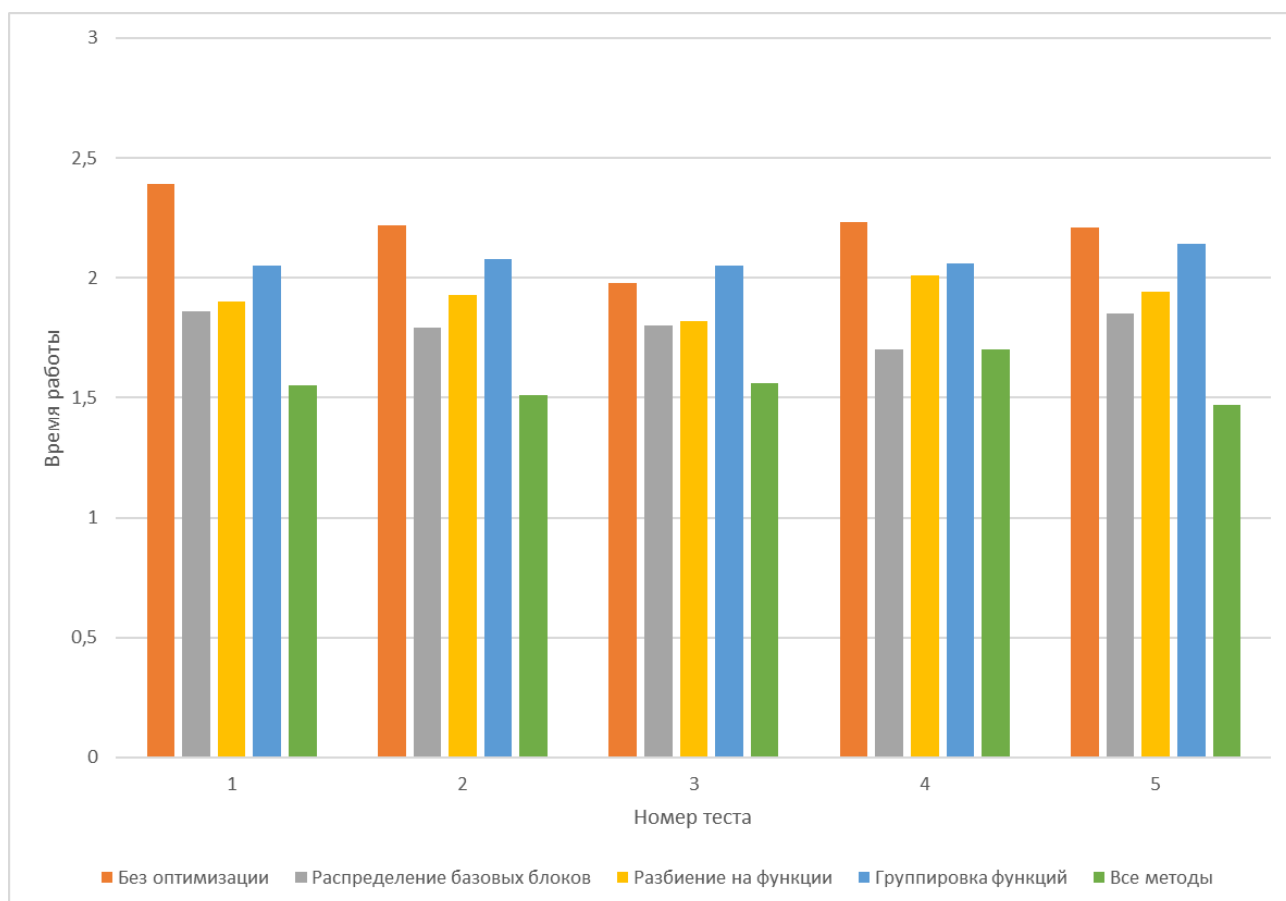


Рисунок 32 – Гистограмма к таблице 1

Из таблицы явно видно, что применение метода распределения базовых блоков принесло увеличение производительности. Это произошло потому, что в оригинальной программе содержится много функций, которые создают

базовые блоки машинных инструкций. Поскольку не все блоки используются одинаково часто, то данная оптимизация является достаточно эффективной. К примеру, использование функций `Code` и `Decode` у класса `MARS` будет равно количеству блоков, подаваемых на шифровку или дешифровку. В то же время количество вызовов функций `ByteToHex` и `HexToByte` тоже напрямую зависит от количества блоков, но уже умножается на 8, поскольку один блок состоит из 4 шестнадцатеричных чисел, каждое из которых в свою очередь состоит из 2 цифр. Количество использований тех или иных функций зависит от того, какой режим используется. Поскольку в режиме шифровки используются одни функции, а при дешифровке частично другие. Обильное количество базовых блоков позволяет проявить себя данному методу оптимизации в полной мере, однако, это всё ещё не даёт максимально возможный результат.

Разбиение на функции дало менее значительный прирост к производительности. Основной причиной этого стало отсутствие в большинстве случаев возможности разбиения блоков машинных инструкций. Поскольку большая часть операций, записанных в функции, не имеет условных ветвлений, то данный метод оптимизации не может показать себя в полную силу. Тем не менее, самым ярким примером, подходящим под данный метод оптимизации здесь является функция `StringToInt`. Она содержит в себе условное ветвление, которое можно записать в отдельную функцию, а далее вызывать её только по необходимости. Наличие небольшого количества таких функций всё же приводит к небольшому увеличению производительности программы.

Группировка функций дала не сильно ощутимый прирост к производительности. Это обусловлено тем, что в программе не так много блоков с горячим кодом являются довольно мелкими функциями, что позволяет с лёгкостью их объединить и обрабатывать в кэше сразу несколько. К примеру, такими функциями в программе могут стать уже упомянутые `ByteToHex` и `HexToByte`. Но в наибольшем выигрыше будут функции вроде `GetBits`, они тоже маленькие и используют циклический сдвиг. В итоге это позволило

ускорить работу программы, но в случае с большими функциями данный метод может значительно замедлить работу. Здесь он был актуален, поскольку имеются подходящие для него функции, однако в сравнении с другими методами показал наихудший результат в отдельности.

Использование одновременно всех методов оптимизации дало меньший прирост, чем суммирование результатов каждой по отдельности. Это обусловлено в первую очередь тем, что данные оптимизации частично мешают друг другу. Но также, в некоторых моментах, они дополняют друг друга. К примеру, группировка функций и распределение блоков вместе позволяют сначала собрать несколько мелких функций с горячим кодом в одну, забывая кэш-линию, после чего переместить новый блок кода выше холодных функций. Таким образом остаются преимущества распределения размещения машинных инструкций, но количество обрабатываемых функций в целом уже значительно меньше. Похожим образом метод распределения взаимодействует с разбиением функций. Хотя это и увеличивает количество функций в программе, оно позволяет расходовать меньше ресурсов на обработку холодного кода. К примеру, если в блоке с горячим кодом находится холодное условное ветвление, как в функции `StringToInt`, то отделение холодной части ускорит выполнение кода. В данном случае это не особо эффективно, поскольку блок при выполнении условного ветвления довольно маленький. В итоге использование всех методов оптимизации является в данном случае оправданным, поскольку в итоге даёт больший прирост производительности, чем любой из них по отдельности. Результат прироста составил 28,4% в среднем.

По результатам, полученным в ходе исследований, можно утверждать, что данные методы оптимизации хорошо подходят для повышения производительности криптографических программ. Во всех тестовых ситуациях наблюдалось ускорение работы программ. Полный текст тестовой программы представлен в приложении А.



Программа была протестирована по несколько раз на шифрование текста с одинаковыми ключом и текстом для шифровки. Длина ключа всегда составляет от 4 до 14 символов. На каждом тесте ключ и текст для шифровки изменялись. Это необходимо чтобы рассмотреть используемую программу при различном количестве поступающих на вход блоков. Используемые пары ключ-текст представлены в таблице 2:

Таблица 2 – Набор пар ключ-текст для тестирования MARS

№ теста	Ключ	Текст для шифровки
1	llvmbolt	The LLVM project has multiple components. The core of the project is itself called "LLVM". This contains all of the tools, libraries, and header files needed to process intermediate representations and convert them into object files. Tools include an assembler, disassembler, bitcode analyzer, and bitcode optimizer.
2	clang	C-like languages use the Clang frontend. This component compiles C, C++, Objective-C, and Objective-C++ code into LLVM bitcode -- and from there into object files, using LLVM.
3	boltoptim	This tool works on already compiled binary. It uses profile information to reorder basic blocks within the function. I think it shouldn't be too hard to integrate it in the build system and enjoy the optimized code layout! The only thing you need to worry about is to have representative and meaningful workload for collecting profiling information, but that a topic for PGO which we will touch later.
4	codelay-out	Compilers like to operate on a basic block level, because it is guaranteed that every instruction in the

		basic block will be executed exactly once. Thus for some problems we can treat all instructions in the basic block as one instruction. This greatly reduces the problem of CFG (control flow graph) analysis and transformations.
5	basicblock	I already wrote a complete article on this topic some time ago: Code alignment issues. This is purely microarchitectural optimization which is usually applied to loops. Figure below is the best brief explanation of the matter

Далее была разработана программа для вычисления чисел Фибоначчи при помощи рекурсивного алгоритма. Это необходимо для проверки описанного ранее процесса формирования графов. Поскольку там указано, что это используется в рекурсии. При больших значениях итераций данный процесс занимает довольно много времени. В связи с этим были разработаны две программы. Первая является обычным рекурсивным алгоритмом Фибоначчи, а вторая вычисляет то же самое, но в виде трёх связанных друг с другом функций. Это сделано для того, чтобы выявить насколько оптимизации смогут повлиять на рекурсию при увеличении количества блоков. Результаты тестов соответственно указаны в таблице 3 и таблице 4:

Таблица 3 – Результаты исследований с рекурсивной функцией

№ теста	Без оптимизации	Распределение базовых блоков	Разбиение на функции	Группировка функций	Все методы
1	150,83	151,69	148,58	154,04	148,04
2	4,62	4,57	4,37	4,61	4,51
3	2697,79	2710,49	2669,70	2741,15	2698,49
4	72,74	72,20	69,94	75,31	72,65
5	24,69	24,97	23,78	25,61	23,81

На основе таблицы 3 была составлена гистограмма, она представлена на рис. 33.

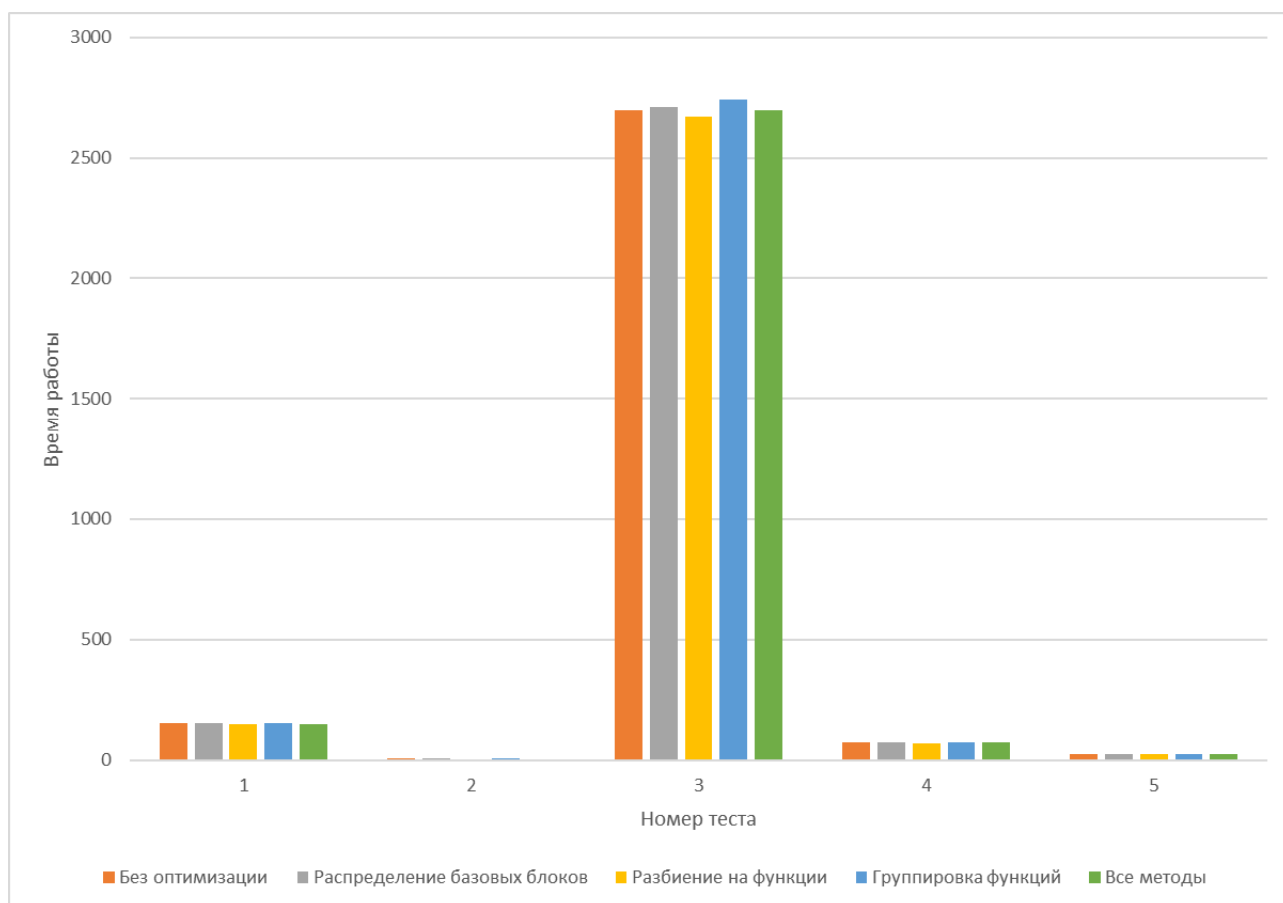


Рисунок 33 – Гистограмма к таблице 3

В случае с обычным рекурсивным алгоритмом Фибоначчи, распределение базовых блоков не приносит никакого увеличения производительности. Это обусловлено малым количеством блоков, ведь их всего 3. В случае со слишком простыми программами метод оптимизации путём распределения базовых блоков неэффективен.

Разбиение на функции в данном случае даёт совсем небольшой прирост, это связано с существованием холодного условного ветвления в рекурсивной функции.

Метод группировки функций в данном случае вообще является неуместным, поскольку имеется всего 1 функция. Это приводит к тому, что программа начинает работать даже медленнее чем без оптимизации.

При применении всех методов оптимизации время работы программы изменяется совершенно незначительно. Это связано с тем, что замедление работы от группировки функций совмещается с практически незаметным ускорением от разбиения функций.

Таблица 4 – Результаты исследований с 3 связанными функциями

№ теста	Без оптимизации	Распределение базовых блоков	Разбиение на функции	Группировка функций	Все методы
1	157,36	155,27	154,63	154,21	153,43
2	4,92	4,81	4,69	4,78	4,57
3	2814,01	2802,52	2757,93	2764,35	2661,44
4	83,24	82,35	73,37	74,25	66,26
5	25,69	24,35	23,54	23,40	23,17

На основе таблицы 4 была составлена гистограмма, она представлена на рис. 34.

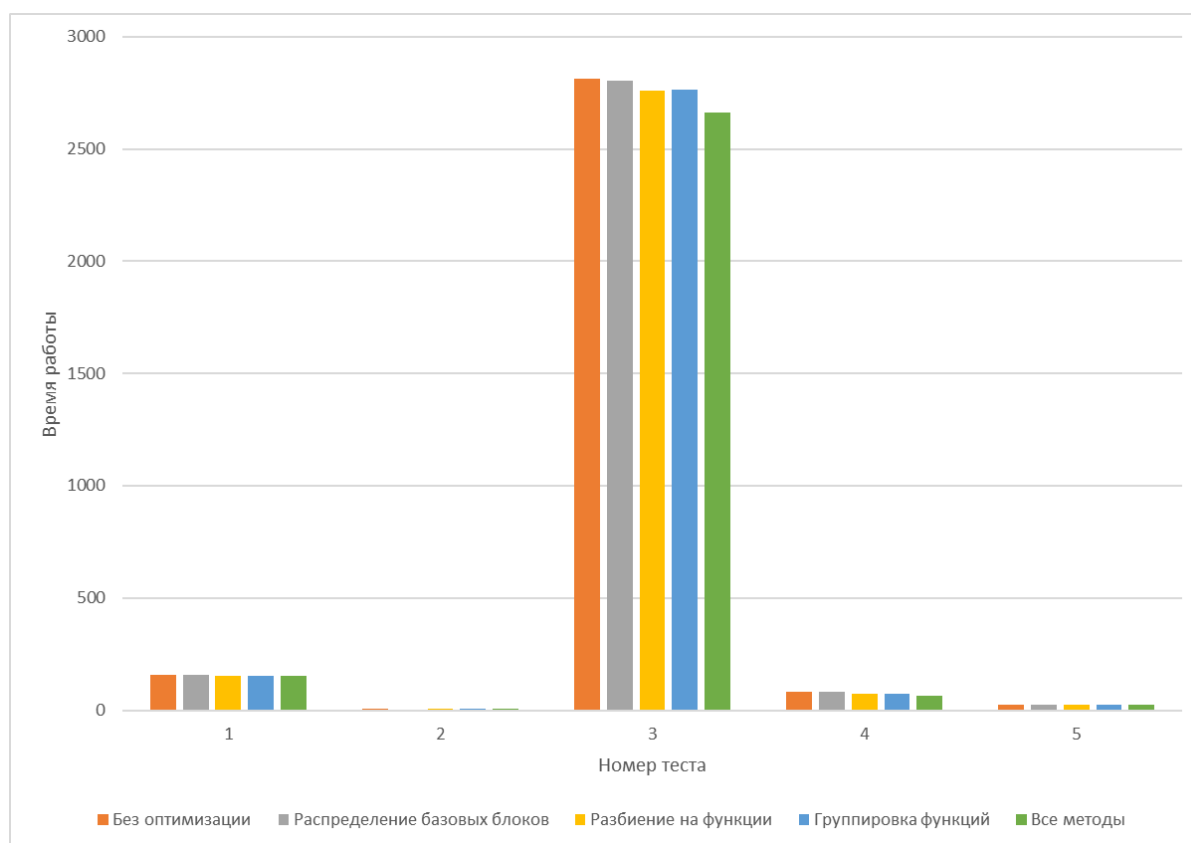


Рисунок 34 – Гистограмма к таблице 4

В случае с несколькими связанными функциями, распределение базовых блоков даёт небольшой прирост производительности. Связано это в данном случае с повышением количества функций, а значит и возможностью их распределения. Данная программа работает медленнее предыдущей, тем не менее здесь метод распределения базовых блоков даёт небольшой прирост относительно времени выполнения программы без оптимизации. Данный прирост можно ощутить только в случаях, когда число итераций не кратно количеству функций.

Разбиение на функции даёт чуть больше, чем в предыдущей версии. Это обусловлено тем, что в данном случае существует большее количество блоков функций, которые можно разбить на горячий и холодный код. В связи с этим прирост производительности в данной тестовой программе стал несколько больше, но он всё ещё мал.

Группировка функций в данном случае уже начинает играть роль, поскольку функции малы и есть возможность исполнять сразу несколько из них за один проход. В отличие от предыдущей программы, в этой такой метод оптимизации имеет смысл.

Когда применяются все виды оптимизаций, данная программа ускоряется значительно, чем прошлая. Это скорее всего связано с повышением количества блоков инструкций в программном коде. Однако это всё ещё довольно маленький результат.

В качестве входных данных для обеих программ использовались данные из таблицы 5:

Таблица 5 – Входные данные

Номер теста	Число итераций
1	35
2	27
3	41
4	33

5	31
---	----

Подводя итоги по результатам обеих программ, можно с точностью сказать, что данные методы оптимизации совершенно не подходят для повышения производительности рекурсивных программ. В первом случае повышения производительности, можно сказать, не было вообще, а во втором оно было куда более незначительным, чем у криптографической программы. В данных методах оптимизации в первую очередь всегда играет роль количество блоков с инструкциями, а рекурсия не способствует их появлению. Однако если рекурсивные функции так или иначе связаны с вызовом других функций, то можно точно утверждать, что данные методы оптимизации могут принести какую-либо пользу. Текст обеих программ поочерёдно представлен в приложении В.

## **4 Оценка и защита результатов интеллектуальной деятельности**

Данное исследование было направлено на исследование методов, подходящих для оптимизации больших программ, что является особенно актуальным в рамках развития современных технологий big data. Оно может помочь улучшить работу крупных проектов путём ускорения их работы вплоть до 20%. Этими технологиями уже пользуются крупные зарубежные компании показывая их эффективность. Разработанные тестовые программы призваны убедиться в эффективности тех или иных оптимизаций путём тестирования различных ситуаций, которые могут возникать в ходе разработки программного обеспечения.

### **4.1 Описание интеллектуальной деятельности**

В данной работе были разработаны и проанализированы тестовые программы для вычисления наиболее успешных методов оптимизации. Согласно Гражданскому кодексу Российской Федерации, программы для электронных вычислительных машин являются результатами интеллектуальной деятельности, а, следовательно и полученные с помощью них результаты. Данное программное обеспечение призвано в первую очередь выявить полезность различных методов оптимизации путём размещения инструкций машинного кода как по отдельности, так и в совокупности. Поскольку данные методы имеют довольно непредсказуемое поведение, то появляется необходимость их сравнения в различных ситуациях. В результате этого тестовые программы нацелены на выявление проблем с такого типа оптимизациями в различных возможных алгоритмах. Это позволяет понять какие из них следует использовать для каких-либо ситуаций, а какие лучше не подключать к процессу оптимизации.

### **4.2 Оценка рыночной стоимости интеллектуальной деятельности**

Поскольку основное направление работы — это тестирование методов оптимизации, то основным его преимуществом является результат, полученный в результате работы тестовых программ. Он позволяет выявить возможности оптимизации уже существующего и работающего программного обес-

печения, что в свою очередь влияет на скорость его работы, что в некоторых ситуациях может напрямую влиять на доход потребителя. При всём этом полученные данные имеют достаточно длительный период времени использования, поскольку использование данных методов оптимизации находятся в процессе развития, а значит они актуальны не только сейчас, но и будут актуальны ещё очень длительное время.

Спрос на возможности оптимизации программного обеспечения всегда высок, потому что каждый разработчик заинтересован в изготовлении наиболее удобного для пользователя продукта, где скорость работы зачастую является одним из ключевых факторов. В результате ускорение работы приложения может привести к различным положительным последствиям, в основном репутационным, таким как повышение количества пользователей.

Результаты данного тестирования могут очень сильно пригодиться для промышленного использования, поскольку изначально данные методы оптимизации рассчитаны на крупные проекты, но основной проблемой при их использовании остаётся неизвестность поведения в тех или иных обстоятельствах, что приводит к необходимости проведения различных тестов. Так же стоит учитывать, что полученные результаты могут превысить затраты преимущественно в промышленном использовании, поскольку эффективность методов оптимизации наилучшим образом видна на них.

Поскольку сложно оценить, сколько в результате принесёт данный метод, а затраты известны, то в дальнейшем для оценки рыночной стоимости будет использоваться затратный подход. В данном случае существует возможность воссоздания подобного объекта оценки путём определения затрат на его воспроизводство. Тем не менее, доход потребителя в этом случае напрямую зависит от программного обеспечения, к которому в итоге будут применяться результаты исследования тестовых программ, что в свою очередь позволяет окупить затраты. Поскольку различных случаев, с которыми могут столкнуться методы оптимизации действительно очень много, то вос-



создание аналогичных по стоимости тестовых программ тоже будет востребовано.

Для повторения результатов данного исследования или создания данных с аналогичными параметрами необходимо вычислить следующие параметры:

- Заработная плата исполнителей
- Длительность проведения работы
- Затраты на расходные материалы
- Амортизационные отчисления
- Накладные расходы

Для вычисления заработной платы исполнителей необходимо для начала определиться с их количеством. В случае данной работы было всего 2 исполнителя, а именно руководитель и студент. В среднем зарплата начинающего программиста на данный момент равняется около 50 000-60 000 Р в месяц, эта сумма причисляется к студенту. Руководитель же в свою очередь получает около 90 000-100 000 Р, как ведущий программист. Для дальнейших вычислений необходимо взять средний показатель зарплаты, 55 000 и 95 000 соответственно. В рабочем месяце 21 день, значит необходимо взять месячную зарплату и поделить её на количество рабочих дней в часах. Полный рабочий день составляет 8 часов, следовательно за месяц человек работает  $8 \cdot 21 = 168$  часов. Далее уже можно определить зарплату студента и преподавателя за час. Зарплата студента  $55\,000 / 168 = 327,38$  Р/час, а преподавателя  $95\,000 / 168 = 565,47$  Р/час. Далее необходимо выяснить длительность проведения работы в часах, для этого можно использовать расчёт среднего времени, которое было потрачено на задачу по формуле

$$t_j^0 = \frac{3t_{min} + 2t_{max}}{5}$$

Где  $t_j^0$  это ожидаемая длительность работы под номером  $j$ ,  $t_{min}$  – наименьшая длительность, а  $t_{max}$  – наибольшая длительность работы. В связи с этим получается таблица 6 по продолжительностям работы:

Таблица 6 – Работы

№ п/п	Наименование работы	Продолжительность работы, ч-ч		
		t <sub>min</sub>	t <sub>max</sub>	t <sub>0</sub>
1	Разработка ТЗ	10	18	13,2
2	Изучение материалов	55	65	59
3	Освоение методов профилирования программ	25	45	33
4	Применение оптимизаций на основе профилирования	20	25	22
5	Изучение методов для проведения оптимизаций блоков машинных инструкций	10	20	14
6	Применение инструментов для проведения оптимизаций блоков машинных инструкций	10	20	14
7	Изучение инструментов для анализа методов оптимизации размещения блоков машинных инструкций	25	35	29
8	Изучение получившихся тестовых программ на производительность	10	15	12
9	Создание тестовых программ	55	95	71
10	Оформление пояснительной записки	70	100	82
11	Создание финальной презентации	20	40	28
Итог		310	478	377,2

На основе данной таблицы уже можно вычислить необходимые затраты на заработную плату исполнителей. За основу можно взять ожидаемую длительность работы и ставку в ас, таким образом получить необходимые расходы на заработную плату студента и руководителя. Для этого была составлена таблица 7:

Таблица 7 – Исполнители

№ п/п	Этапы и содержание выполняемых работ	Исполнитель	Трудоемкость ч-ч	Ставка, руб./час
1	Разработка ТЗ	Руководитель	13,2	565,47
2	Изучение материалов	Студент	59	327,38
3	Освоение методов профилирования программ	Студент	33	327,38
4	Применение оптимизаций на основе профилирования	Студент	22	327,38
5	Изучение методов для проведения оптимизаций блоков машинных инструкций	Студент	14	327,38
6	Применение инструментов для проведения оптимизаций блоков машинных инструкций	Студент	14	327,38
7	Изучение инструментов для анализа методов оптимизации размещения блоков машинных инструкций	Студент	29	327,38
8	Изучение получившихся тестовых программ на производительность	Студент	12	327,38

9	Создание тестовых программ	Студент	71	327,38
10	Оформление пояснительной записки	Студент	82	327,38
11	Создание финальной презентации	Студент	28	327,38

По данным таблицы можно рассчитать размер заработной платы каждого исполнителя:

Студент

$$З_{\text{общ.ст.}} = \sum t_{\text{ст.}} * C_{\text{ст.}} = 365 * 327,38 = 119\,493,7$$

В данной формуле  $З_{\text{общ.ст.}}$  это расходы на заработную плату студента, а  $t_{\text{ст.}}$  это время работы студента, и  $C_{\text{ст.}}$  это ставка студента.

Руководитель

$$З_{\text{общ.р.}} = \sum t_{\text{р.}} * C_{\text{р.}} = 13,2 * 565,47 = 7\,464,2$$

Здесь же будет аналогично предыдущему, но  $З_{\text{общ.р.}}$  - расходы на заработную плату руководителя,  $t_{\text{р.}}$  - время работы руководителя,  $C_{\text{р.}}$  - ставка руководителя.

Так же необходимо учитывать расходы на дополнительную заработную плату для руководителя и студента. Она считается по формуле:

$$З_{\text{доп.}} = З_{\text{осн.ст.}} * \frac{Н_{\text{доп}}}{100}$$

Здесь  $З_{\text{доп.}}$  - дополнительная заработная плата, а  $Н_{\text{доп}}$  - норматив дополнительной заработной платы. В качестве норматива дополнительной заработной платы для студента берётся 13%, для руководителя 8,1% В итоге в результате формулы выглядят следующим образом:

$$З_{\text{доп.ст.}} = 119\,493,7 * \frac{13}{100} = 15\,534,18$$

$$З_{\text{доп.р.}} = 7464,2 * \frac{8.1}{100} = 604,6$$

Где  $З_{\text{доп.ст.}}$  - дополнительная заработная плата студента, а  $З_{\text{доп.р.}}$  - дополнительная заработная плата руководителя.

Ещё необходимо рассчитать страховые взносы. Это нужно, поскольку социальное, пенсионное и медицинское страхование является обязательным. Для этого задействована следующая формула:

$$З_{\text{соц.}} = (З_{\text{осн.}} + З_{\text{доп.}}) * \frac{Н_{\text{соц}}}{100}$$

Здесь  $З_{\text{соц.}}$  Это социальные расходы, а  $Н_{\text{соц}}$  это норматив отчислений страховых взносов на обязательные медицинское и пенсионное страхование, он равен 30%. Результаты вычислений для исполнителей приведены далее:

$$З_{\text{соц.ст.}} = (119\,493,7 + 15\,534,28) \frac{30}{100} = 40\,508,39$$

$$З_{\text{соц.р.}} = (7\,464,2 + 604,6) \frac{30}{100} = 2\,420,64$$

Где  $З_{\text{соц.ст.}}$  – социальные расходы на студента, а  $З_{\text{соц.р.}}$  - социальные расходы на руководителя соответственно.

В итоге получается следующая таблица 8 с полными расходами на заработную плату работников

Таблица 8 – Расходы на работников

Исполнитель	Оплата, руб./час.	Количество часов	Основная оплата руб	Дополнительная оплата руб	социальные расходы	Итоговые затраты
студент	327,38	365	119 493,7	15 534,28	40 508,39	175 536,37
руководитель	565,47	12,2	7 464,2	604,6	2 420,64	10 489,44
итог			126 957,9	16 138,88	42929,03	186 025,81

По этим данным была составлена гистограмма расходов на сотрудников. Она представлена на рис. 35.

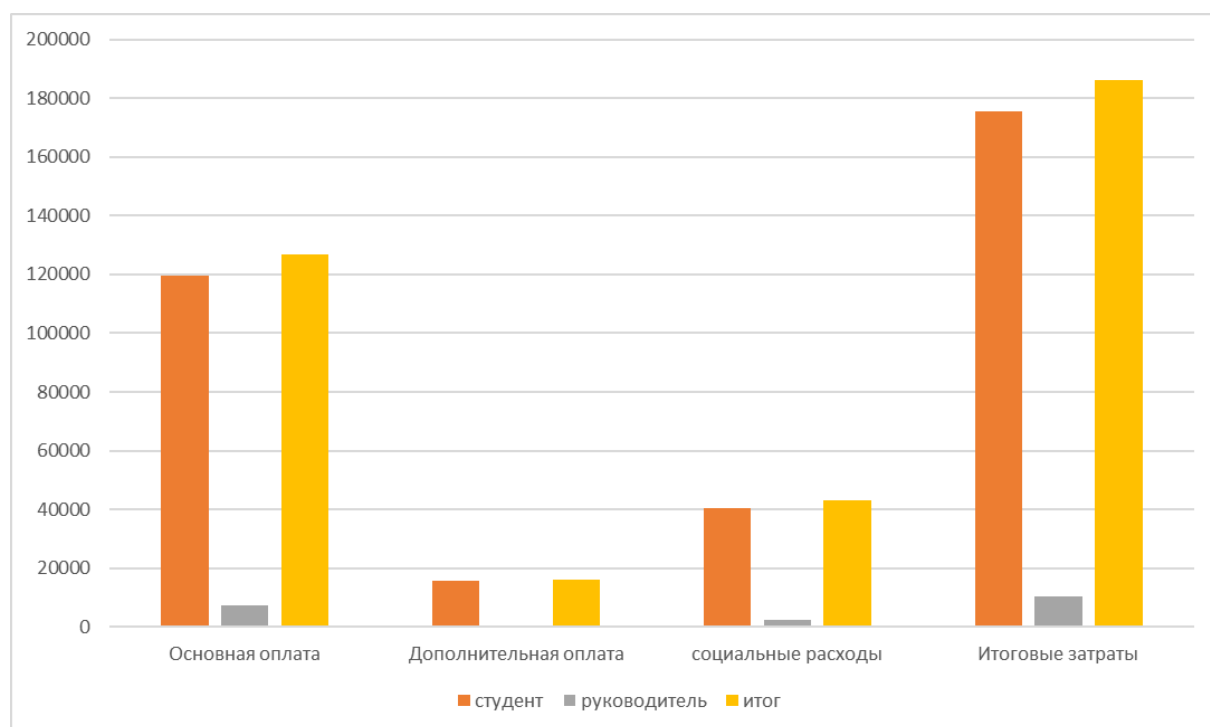


Рисунок 35 – Гистограмма к таблице 8

Помимо зарплаты сотрудников существуют так же и затраты на материалы, они вычисляются по формуле:

$$З_M = \sum_{l=1}^L G_l C_l \left(1 + \frac{H_{т.з.}}{100}\right)$$

Здесь  $З_M$  это затраты на материалы,  $L$  является индексом вида материала, а  $G_l$  нормой расхода  $l$ -го материала.  $H_{т.з.}$  – норма транспортно-заготовительных расходов (10 %). По ним получается следующая таблица 9, являющаяся таблицей расходов.

Таблица 9 – Расходы на материалы

Изделие	Тип	Норма расходов на изделие (ед.)	Цена за единицу (руб./шт.)	Сумма на изделие (руб.)
Бумага	Формат	1	400	440

офисная	A4			
Картридж для прин- тера	Черный	1	700	770
Итого				1 210

Далее необходимо так же учесть амортизационные отчисления, они определяются по формуле:

$$A_i = Ц_{п.н.i} * \frac{H_{ai}}{100}$$

Здесь  $A_i$  – амортизационные отчисления за год по основному средству, а  $Ц_{п.н.i}$  - первоначальная стоимость основного средства.  $H_{ai}$  является годовой нормой амортизации основного средства, её можно рассчитать по следующей формуле:

$$H_{AM} = \frac{t_{и.}}{t_{п.}} \cdot 100\%$$

В ней  $t_{п.}$  это срок полезного использования, а  $t_{и.}$  - срок использования для работы. Первый подразумевает под собой средний срок службы того или иного оборудования, а второй сколько оно непосредственно использовалось для работы.

Исходя из данных формул можно получить таблицу 10, показывающую амортизационные отчисления.

Таблица 10 – амортизационные отчисления

Основное сред- ство	Первона- чальная стоимость, руб.	Срок полез- ного исполь- зования, год	Годовая норма аморти- зации, %	Амортизацион- ные отчисления за год, руб.
Ноутбук Lenovo	100 809	5	20	20 161,8

legion y-520				
Принтер HP LaserJet Professional P1102	23500	10	10	2350

На основе этих данных можно рассчитать амортизационные отчисления по основному средству по следующей формуле:

$$A_{i(\text{вкр})} = T_{i(\text{вкр})} \frac{A_i}{12}$$

В данном случае считаются отчисления, затраченные на время написания дипломной работы, что и обозначает  $A_{i(\text{вкр})}$ . Для того чтобы их получить, необходимо поделить амортизационные вычисления  $A_i$  на количество месяцев в году. Это позволяет узнать амортизационные отчисления за месяц, после его умножив на  $T_{i(\text{вкр})}$ , время, затраченное на исследование, получим итоговые расходы на амортизационные вычисления, представленные в следующей таблице 11.

Таблица 11 – амортизационные отчисления

Устройство	Амортизационные отчисления за год, руб.	Время выполнения ВКР, мес.	Амортизационные отчисления, руб.
Ноутбук Lenovo legion y-520	20 161,8	3	5 040,45
Принтер HP LaserJet Professional P1102	2350	3	587,5
ИТОГ			5 627,95

Осталось добавить ко всему вышеперечисленному накладные расходы. Несмотря на то, что они не являются основными, они нужны для обеспече-



ния работы. В данном случае в качестве таких расходов были выбраны расходы на транспорт и на связь. Каждый из них рассчитывался по следующей формуле:

$$З_{н.р.} = (З + З_{доп.}) * \frac{Н_{н.р.}}{100}$$

Соответственно  $З_{н.р.}$  здесь является затратами на накладные расходы, которые выходят из заработной платы с учётом коэффициента  $Н_{н.р.}$ , т.е. процента накладных расходов. В данном случае он был принят за 10%. Использование данной формулы позволяет получается таблица 12.

Таблица 12 – накладные расходы

Исполнитель	Накладные расходы
студент	13 502, 79
руководитель	806,88
ИТОГ	14309,67

После этого можно составить общую таблицу затрат на выпускную квалификационную работу. В неё входят все вышеперечисленные пункты. Эти данные представлены в таблице 13.

Таблица 13 – затраты на ВКР

№ п/п	Наименование статьи	Сумма, руб
1.	Расходы на оплату труда	143 096,78
2.	Отчисления на социальные нужды	42 929,03
3.	Материалы	1 210
4.	Амортизационные отчисления	6 125
5.	Накладные расходы	34 431
ИТОГ		227 791,81

Как видно из результатов расчёта, затраты на производство таких исследований составляют 227 791,81 рублей. Основная часть денежных средств

уходит на оплату работы студента, примерно 59% без учёта социальных нужд или около 77% включая их. Это получилось в результате того, что не были учтены другие факторы для затрат, такие как аренда помещения и аренда или закупка дорогостоящего оборудования для проведения тестов. Тем не менее разработка программного обеспечения является относительно долгосрочным вложением, поскольку среднее время жизни программного продукта на рынке 10 лет. Результаты данных тестов позволяют продлить время его работы примерно на 20%, поскольку дают возможность ускорения работы программы примерно на такой уровень. Так же здесь имеется возможность оптимизации компилятора для программ. Программисты очень много времени тратят в ожидании компиляции кода, в среднем она занимает 5-10 минут, а за день таких набирается 20-30. Ускоренное на 30% время компиляции позволяет сотруднику сделать большее количество полезной работы за меньшее время. Иными словами программист, работающий полный рабочий день за рабочий месяц наберёт примерно  $7 \cdot 25 \cdot 21 = 3675$  минут на ожидание компиляции, сокращение этого времени на 30% сделает из него 2572,5 минуты, что в совокупности даёт прибыль в 1102,5 минуты, что равняется около 18 часов, в денежном эквиваленте при зарплате сотрудника в 55 000 это будет  $18 \cdot 327,38 = 5\,892,84$  в месяц. Таким образом данное исследование окупится за 38 с половиной месяцев.

В итоге была определена большая часть затрат, сопутствующая разработке проекта, на основе чего можно вычислить некоторую доходность и прибыль. В результате оценки рыночной стоимости результата интеллектуальной деятельности была выявлена возможность воссоздания объекта оценки путём определения затрат на него.

#### **4.3 Правовая защита результатов интеллектуальной деятельности**

Правовая защита объекта исследований, являющегося программой ЭВМ обеспечивается в первую очередь авторским правом. Более подробно об этом рассказано в Гражданском кодексе Российской Федерации [25], в

главе 70 “Авторское право”. В описании авторских прав в статье 1255 перечислены основные права, принадлежащие автору произведения:

- Исключительное право на произведение
- Право авторства
- Право автора на имя
- Право на неприкосновенность произведения
- Право на обнародование произведения

Авторское право распространяется на любые программы ЭВМ и базы данных, вне зависимости от стадии разработки, носителя, назначения и достоинства. По сути, оно приравнивается к авторскому праву в литературе. Так же существует возможность государственной регистрации программ для ЭВМ. Для этого необходимо зарегистрировать такую программу в федеральном органе исполнительной власти по интеллектуальной собственности. Однако, если данная программа содержит государственную тайну, то она не будет допущена к регистрации. Федеральный орган исполнительной власти по интеллектуальной собственности может вносить изменения в Реестр программ для ЭВМ предупреждая правообладателя, в случае если там были найдены и исправлены очевидные и технические ошибки, как по собственной инициативе, так и по просьбе любого лица. Так же в статье Гражданского кодекса Российской Федерации [25] расписаны возможности технических средств защиты авторских прав.

Тем не менее программа может так же быть защищена патентным правом. Оно так же описано в Гражданском кодексе Российской Федерации. В целом в нём указаны варианты распоряжения и передачи прав на получение патента. Их можно передавать по договору, в том числе и трудовому. Защита исключительного права, которое удостоверено патентом, осуществляется только после государственной регистрации изобретения, полезной модели или выдачи патента. Патент может так же быть признан не действительным по ряду пунктов, указанных в статьях Гражданского кодекса Российской Федерации [25] под номерами 1398 и 1349.

Авторское право возникает с момента создания программы и действует в течение всей жизни автора и 50 лет после его смерти, в отличие от авторства. Оно, как и имя автора и неприкосновенность произведения, охраняется бессрочно. В случае составления открытой лицензии, если её срок действия не определён, то для программ ЭВМ будет являться договор считается заключённым на весь срок действия исключительного права, у других видов произведений срок такого договора составляет 5 лет. В случае нарушения исключительного права, штраф может составлять от десяти тысяч рублей до пяти миллионов, что определяется судом исходя из характера нарушения, либо же в двукратном размере стоимости права использования из цены, которая обычно взимается за такое же правомерное использование произведения.

Сроки действия исключительного права на патенты составляют 20 лет для изобретений, 10 для полезных моделей и 5 для промышленных образцов. Тем не менее существует возможность продлить исключительное право. К примеру, срок его действия может быть продлён для промышленного образца и удостоверяющего это право патента по заявлению патентообладателя на 5 лет, но не более чем на 25 лет с момента подачи первоначальной заявки.

В данном разделе были перечислены все основные способы правовой защиты интеллектуальной деятельности, подходящие для объекта исследований, являющемуся набором программ для ЭВМ. В том числе были указаны сроки и объём данной защиты.

## ЗАКЛЮЧЕНИЕ

В результате данной работы были проведены анализ предметной области с методами оптимизации перемещения машинных инструкций и анализ существующих средств для сбора информации о работе программы. На основе этого были выбраны определённые методы оптимизации и инструменты для профилирования. Это так же привело к выбору инструмента LLVM Bolt, который в свою очередь основан на LLVM, для оптимизации размещения блоков с инструкциями. Поскольку на работу данных оптимизаций сильно влияет профилирование, то были найдены и перечислены несколько инструментов, выполняющих его. Среди них был выбран наиболее подходящий для работы совместно с LLVM Bolt. Далее были разработаны тестовые программы для определения эффективности выбранных методов оптимизации в различных условиях.

Тестовые программы были составлены, чтобы выяснить актуальность методов оптимизации после предварительного профилирования для криптографических программ, а также для рекурсивных программ. Основная проблема в использовании данных методов, как правило, заключается в том, что они являются опасными. Это подразумевает, что они могут не только увеличить, но и ухудшить производительность. Тем не менее результаты тестов показали, что такое свойственно в основном небольшим программам с малым количеством блоков. В больших программах данный вид оптимизаций показал себя как вполне эффективный, повысив производительность криптографической программы на 28%.

Выбранные методы показали себя неэффективными так же и в рамках оптимизации рекурсивных программ. Это является вполне логичным, ведь рекурсивные программы не создают новые блоки кода после компиляции.

В итоге можно сказать, что эти методы могут предложить неплохую прибавку к производительности программ, в которых много блоков машинного кода. В других случаях прибавка либо отсутствует, либо является не-

значительной. Тем не менее это не отменяет возможность использовать их со стандартными методами оптимизации.

В дальнейшем предлагается направить исследования применения методов оптимизации размещения машинных инструкций после предварительного профилирования в сторону использования криптографических и веб технологий. Основная причина для этого в том, что они обрабатывают информацию по блочному принципу, а следовательно, имеют множество итераций, успешная оптимизация которых может привести к сильному повышению производительности их тандема.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. 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.
2. 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.
3. Документация LLVM. Writing an LLVM Pass URL: <https://llvm.org/docs/WritingAnLLVMPass.html> (дата обращения 26.03.23)
4. Репозиторий LLVM Bolt URL: <https://github.com/facebookincubator/BOLT> (дата обращения 26.03.23)
5. Advanced Build Configurations URL: <https://llvm.org/docs/AdvancedBuilds.html> (дата обращения 26.03.23)
6. Dynamorio URL: <https://dynamorio.org> (дата обращения 18.03.23)
7. Clang: a C language family frontend for LLVM URL: <https://clang.llvm.org/> (дата обращения 30.03.23)
8. Herlihy, M. The Art of Multiprocessor Programming / Maurice Herlihy, Nir Shavit, – 2012
9. German Gorelkin. Выравнивание и заполнение структур URL: <https://medium.com/german-gorelkin/go-alignment-a359ff54f272> (дата обращения 12.03.23)





[%B0%D0%BD%D0%B8%D0%B5\\_%D0%BE%D0%B1%D1%85%D0%BE%D0%B4%D0%B0\\_%D0%B2\\_%D0%B3%D0%BB%D1%83%D0%B1%D0%B8%D0%BD%D1%83\\_%D0%B4%D0%BB%D1%8F\\_%D1%82%D0%BE%D0%BF%D0%BE%D0%BB%D0%BE%D0%B3%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B9\\_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B8](https://habr.com/ru/articles/100953/) (дата обращения 30.03.23)

22. Habr. Топологическая сортировка URL: <https://habr.com/ru/articles/100953/> (дата обращения)
23. Дерюгин Д. Е. Профилирование операционных систем реального времени / Д. Е. Дерюгин. – СПб – 2014
24. Лопес, Б. К. LLVM: инфраструктура для разработки компиляторов / Бруно Кардос Лопес, Рафаэль Аулер / пер. с англ. Киселев А. Н. – М.: ДМК Пресс, – 2015
25. Гражданский кодекс Российской Федерации (ГК РФ) – АСТ – 2023

## ПРИЛОЖЕНИЕ А

```
#include <vector>
```

```
#include<string>
```

```
const unsigned int B[4] = { 0xa4a8d57b, 0x5b5d193b, 0xc8a8309b, 0x73f9a978 };
```

```
static unsigned int S0[256] = {
```

```
    0x09d0c479, 0x28c8ffe0, 0x84aa6c39, 0x9dad7287, 0x7dff9be3, 0xd4268361,  
    0xc96da1d4, 0x7974cc93, 0x85d0582e, 0x2a4b5705, 0x1ca16a62, 0xc3bd279d,  
    0x0f1f25e5, 0x5160372f, 0xc695c1fb, 0x4d7ff1e4, 0xae5f6bf4, 0x0d72ee46,  
    0xff23de8a, 0xb1cf8e83, 0xf14902e2, 0x3e981e42, 0x8bf53eb6, 0x7f4bf8ac,  
    0x83631f83, 0x25970205, 0x76afe784, 0x3a7931d4, 0x4f846450, 0x5c64c3f6,  
    0x210a5f18, 0xc6986a26, 0x28f4e826, 0x3a60a81c, 0xd340a664, 0x7ea820c4,  
    0x526687c5, 0x7eddd12b, 0x32a11d1d, 0x9c9ef086, 0x80f6e831, 0xab6f04ad,  
    0x56fb9b53, 0x8b2e095c, 0xb68556ae, 0xd2250b0d, 0x294a7721, 0xe21fb253,  
    0xae136749, 0xe82aae86, 0x93365104, 0x99404a66, 0x78a784dc, 0xb69ba84b,  
    0x04046793, 0x23db5c1e, 0x46cae1d6, 0x2fe28134, 0x5a223942, 0x1863cd5b,  
    0xc190c6e3, 0x07dfb846, 0x6eb88816, 0x2d0dcc4a, 0xa4ccae59, 0x3798670d,  
    0xcbfa9493, 0x4f481d45, 0xeafc8ca8, 0xdb1129d6, 0xb0449e20, 0xf5407fb,  
    0x6167d9a8, 0xd1f45763, 0x4daa96c3, 0x3bec5958, 0xababa014, 0xb6ccd201,  
    0x38d6279f, 0x02682215, 0x8f376cd5, 0x092c237e, 0xbfc56593, 0x32889d2c,  
    0x854b3e95, 0x05bb9b43, 0x7dcd5dcd, 0xa02e926c, 0xfae527e5, 0x36a1c330,  
    0x3412e1ae, 0xf257f462, 0x3c4f1d71, 0x30a2e809, 0x68e5f551, 0x9c61ba44,  
    0x5ded0ab8, 0x75ce09c8, 0x9654f93e, 0x698c0cca, 0x243cb3e4, 0x2b062b97,  
    0x0f3b8d9e, 0x00e050df, 0xfc5d6166, 0xe35f9288, 0xc079550d, 0x0591ae8,  
    0x8e531e74, 0x75fe3578, 0x2f6d829a, 0xf60b21ae, 0x95e8eb8d, 0x6699486b,  
    0x901d7d9b, 0xfd6d6e31, 0x1090acef, 0xe0670dd8, 0xdab2e692, 0xcd6d4365,  
    0xe5393514, 0x3af345f0, 0x6241fc4d, 0x460da3a3, 0x7bcf3729, 0x8bfd1d1e0,  
    0x14aac070, 0x1587ed55, 0x3afd7d3e, 0xd2f29e01, 0x29a9d1f6, 0xefb10c53,  
    0xcf3b870f, 0xb414935c, 0x664465ed, 0x024acac7, 0x59a744c1, 0x1d2936a7,  
    0xdc580aa6, 0xcf574ca8, 0x040a7a10, 0x6cd81807, 0x8a98be4c, 0xaccea063,  
    0xc33e92b5, 0xd1e0e03d, 0xb322517e, 0x2092bd13, 0x386b2c4a, 0x52e8dd58,  
    0x58656dfb, 0x50820371, 0x41811896, 0xe337ef7e, 0xd39fb119, 0xc97f0df6,  
    0x68fea01b, 0xa150a6e5, 0x55258962, 0xeb6ff41b, 0xd7c9cd7a, 0xa619cd9e,  
    0xbcf09576, 0x2672c073, 0xf003fb3c, 0x4ab7a50b, 0x1484126a, 0x487ba9b1,  
    0xa64fc9c6, 0xf6957d49, 0x38b06a75, 0xdd805fcd, 0x63d094cf, 0xf51c999e,  
    0x1aa4d343, 0xb8495294, 0xce9f8e99, 0xbffcd770, 0xc7c275cc, 0x378453a7,  
    0x7b21be33, 0x397f41bd, 0x4e94d131, 0x92cc1f98, 0x5915ea51, 0x99f861b7,  
    0xc9980a88, 0x1d74fd5f, 0xb0a495f8, 0x614deed0, 0xb5778eea, 0x5941792d,  
    0xfa90c1f8, 0x33f824b4, 0xc4965372, 0x3ff6d550, 0x4ca5fec0, 0x8630e964,  
    0x5b3fbbd6, 0x7da26a48, 0xb203231a, 0x04297514, 0x2d639306, 0x2eb13149,  
    0x16a45272, 0x532459a0, 0x8e5f4872, 0xf966c7d9, 0x07128dc0, 0xd44db62,  
    0xafc8d52d, 0x06316131, 0xd838e7ce, 0x1bc41d00, 0x3a2e8c0f, 0xea83837e,
```

```
0xb984737d, 0x13ba4891, 0xc4f8b949, 0xa6d6acb3, 0xa215cdce, 0x8359838b,  
0x6bd1aa31, 0xf579dd52, 0x21b93f93, 0xf5176781, 0x187dfdde, 0xe94aeb76,  
0x2b38fd54, 0x431de1da, 0xab394825, 0x9ad3048f, 0xdfea32aa, 0x659473e3,  
0x623f7863, 0xf3346c59, 0xab3ab685, 0x3346a90b, 0x6b56443e, 0xc6de01f8,  
0x8d421fc0, 0x9b0ed10c, 0x88f1a1e9, 0x54c1f029, 0x7dead57b, 0x8d7ba426,  
0x4cf5178a, 0x551a7cca, 0x1a9a5f08, 0xfcd651b9, 0x25605182, 0xe11fc6c3,  
0xb6fd9676, 0x337b3027, 0xb7c8eb14, 0x9e5fd030
```

```
};
```

```
const unsigned int S1[256] = {
```

```
0x6b57e354, 0xad913cf7, 0x7e16688d, 0x58872a69, 0x2c2fc7df, 0xe389ccc6,  
0x30738df1, 0x0824a734, 0xe1797a8b, 0xa4a8d57b, 0x5b5d193b, 0xc8a8309b,  
0x73f9a978, 0x73398d32, 0xf59573e, 0xe9df2b03, 0xe8a5b6c8, 0x848d0704,  
0x98df93c2, 0x720a1dc3, 0x684f259a, 0x943ba848, 0xa6370152, 0x863b5ea3,  
0xd17b978b, 0x6d9b58ef, 0x0a700dd4, 0xa73d36bf, 0xe6a0829, 0x8695bc14,  
0xe35b3447, 0x933ac568, 0x8894b022, 0x2f511c27, 0xddfbcc3c, 0x006662b6,  
0x117c83fe, 0x4e12b414, 0xc2bca766, 0x3a2fec10, 0xf4562420, 0x55792e2a,  
0x46f5d857, 0xcda25ce, 0xc3601d3b, 0x6c00ab46, 0xfac9c28, 0xb3c35047,  
0x611dfee3, 0x257c3207, 0xfdd58482, 0x3b14d84f, 0x23becb64, 0xa075f3a3,  
0x088f8ead, 0x07adf158, 0x7796943c, 0xfacabf3d, 0xc09730cd, 0xf7679969,  
0xda44e9ed, 0x2c854c12, 0x35935fa3, 0x2f057d9f, 0x690624f8, 0x1cb0bafd,  
0x7b0dbdc6, 0x810f23bb, 0xfa929a1a, 0x6d969a17, 0x6742979b, 0x74ac7d05,  
0x010e65c4, 0x86a3d963, 0xf907b5a0, 0xd0042bd3, 0x158d7d03, 0x287a8255,  
0xbba8366f, 0x096edc33, 0x21916a7b, 0x77b56b86, 0x951622f9, 0xa6c5e650,  
0x8cea17d1, 0xcd8c62bc, 0xa3d63433, 0x358a68fd, 0x0f9b9d3c, 0xd6aa295b,  
0xfe33384a, 0xc000738e, 0xcd67eb2f, 0xe2eb6dc2, 0x97338b02, 0x06c9f246,  
0x419cf1ad, 0x2b83c045, 0x3723f18a, 0xcb5b3089, 0x160bead7, 0x5d494656,  
0x35f8a74b, 0x1e4e6c9e, 0x000399bd, 0x67466880, 0xb4174831, 0xacf423b2,  
0xca815ab3, 0x5a6395e7, 0x302a67c5, 0x8bdb446b, 0x108f8fa4, 0x10223eda,  
0x92b8b48b, 0x7f38d0ee, 0xab2701d4, 0x0262d415, 0xaf224a30, 0xb3d88aba,  
0xf8b2c3af, 0xdaf7ef70, 0xcc97d3b7, 0xe9614b6c, 0x2baebff4, 0x70f687cf,  
0x386c9156, 0xce092ee5, 0x01e87da6, 0x6ce91e6a, 0xbb7bcc84, 0xc7922c20,  
0x9d3b71fd, 0x060e41c6, 0xd7590f15, 0x4e03bb47, 0x183c198e, 0x63eeb240,  
0x2ddb49a, 0x6d5cba54, 0x923750af, 0xf9e14236, 0x7838162b, 0x59726c72,  
0x81b66760, 0xbb2926c1, 0x48a0ce0d, 0xa6c0496d, 0xad43507b, 0x718d496a,  
0x9df057af, 0x44b1bde6, 0x054356dc, 0xde7ced35, 0xd51a138b, 0x62088cc9,  
0x35830311, 0xc96efca2, 0x686f86ec, 0x8e77cb68, 0x63e1d6b8, 0xc80f9778,  
0x79c491fd, 0x1b4c67f2, 0x72698d7d, 0x5e368c31, 0xf7d95e2e, 0xa1d3493f,  
0xdcd9433e, 0x896f1552, 0x4bc4ca7a, 0xa6d1baf4, 0xa5a96dcc, 0x0bef8b46,  
0xa169fda7, 0x74df40b7, 0x4e208804, 0x9a756607, 0x038e87c8, 0x20211e44,  
0x8b7ad4bf, 0xc6403f35, 0x1848e36d, 0x80bdb038, 0x1e62891c, 0x643d2107,  
0xbf04d6f8, 0x21092c8c, 0xf644f389, 0x0778404e, 0x7b78adb8, 0xa2c52d53,  
0x42157abe, 0xa2253e2e, 0x7bf3f4ae, 0x80f594f9, 0x953194e7, 0x77eb92ed,
```

```

0xb3816930, 0xda8d9336, 0xbf447469, 0xf26d9483, 0xee6faed5, 0x71371235,
0xde425f73, 0xb4e59f43, 0x7dbe2d4e, 0x2d37b185, 0x49dc9a63, 0x98c39d98,
0x1301c9a2, 0x389b1bbf, 0x0c18588d, 0xa421c1ba, 0x7aa3865c, 0x71e08558,
0x3c5cfcaa, 0x7d239ca4, 0x0297d9dd, 0xd7dc2830, 0x4b37802b, 0x7428ab54,
0xaeee0347, 0x4b3fbb85, 0x692f2f08, 0x134e578e, 0x36d9e0bf, 0xae8b5fcf,
0xedb93ecf, 0x2b27248e, 0x170eb1ef, 0x7dc57fd6, 0x1e760f16, 0xb1136601,
0x864e1b9b, 0xd7ea7319, 0x3ab871bd, 0xcfa4d76f, 0xe31bd782, 0x0dbeb469,
0xabb96061, 0x5370f85d, 0xffb07e37, 0xda30d0fb, 0xebc977b6, 0x0b98b40f,
0x3a4d0fe6, 0xdf4fc26b, 0x159cf22a, 0xc298d6e2, 0x2b78ef6a, 0x61a94ac0,
0xab561187, 0x14eea0f0, 0xdf0d4164, 0x19af70ee
};

```

//Циклический сдвиг влево на countOfbits

```

unsigned int LeftCyclShift(unsigned int countOfbits, unsigned int number)
{
    countOfbits %= 32;
    unsigned int ones = number;
    ones >>= (32 - countOfbits);
    number <<= countOfbits;
    number |= ones;
    return number;
}

```

//Циклический сдвиг вправо на countOfbits

```

unsigned int RightCyclShift(unsigned int countOfbits, unsigned int number)
{
    countOfbits %= 32;
    unsigned int ones = number;
    ones <<= (32 - countOfbits);
    number >>= countOfbits;
    number |= ones;
    return number;
}

```

//Получить байт под номером byteNum(от 1 до 4) из числа number

```

unsigned int GetByte(unsigned int byteNum, unsigned int number)
{
    unsigned int curr = (number >> 8 * byteNum) & 255;
    return curr;
}

```

```
unsigned int GetBits(unsigned int countOfbits, unsigned int number) {
```

```
    countOfbits %= 32;
    unsigned int value = number;
    value << (32 - countOfbits);
    return value >> (32 - countOfbits);
```

```
}
```

```
std::vector<unsigned int> StringToInt(std::string msg) {
```

```
    std::vector<unsigned int> arr;
    int j = 0;
    for (int i = 0; i < msg.length(); i++) {
        if (j == 0) {
            arr.push_back(0);
        }
        arr[arr.size() - 1] |= msg[i] << (j * 8);
        ++j;
        j %= 4;
    }
```

```
    return arr;
```

```
}
```

```
std::string StringFromInt(std::vector<unsigned int> arr) {
```

```
    std::string result;
    for (int j = 0; j < arr.size(); j++) {
        unsigned int temp = arr[j];
        result = result + char(GetByte(0, temp));
        result = result + char(GetByte(1, temp));
        result = result + char(GetByte(2, temp));
        result = result + char(GetByte(3, temp));
    }
```

```
    return result;
```

```
}
```

```
unsigned char ByteToHex(unsigned char byte)
```

```
{
```

```
    switch (byte)
```

```
    {
```

```
        case 10:
```

```
            byte = 'A'; break;
```

```
        case 11:
```

```

        byte = 'B'; break;
case 12:
        byte = 'C'; break;
case 13:
        byte = 'D'; break;
case 14:
        byte = 'E'; break;
case 15:
        byte = 'F'; break;
default:
        byte += '0';
    }
    return byte;
}

```

unsigned int HexToByte(unsigned char byte)

```

{
    switch (byte)
    {
        case 'A':
            byte = 10; break;
        case 'B':
            byte = 11; break;
        case 'C':
            byte = 12; break;
        case 'D':
            byte = 13; break;
        case 'E':
            byte = 14; break;
        case 'F':
            byte = 15; break;
        default:
            byte -= '0';
    }
    return byte;
}

```

//61 12 A4 BB 87 5E 41 81 71 9A A6 E3 F5 FA 79 32

std::string IntToHex(unsigned int number)

```

{
    std::string s = "";
    for (int i = 0; i < 4; i++)
    {

```

```

        s += ByteToHex(GetByte(i, number) / 16);
        s += ByteToHex(GetByte(i, number) % 16);
        s += ' ';
    }
    return s;
}

std::vector<unsigned int> HexToInt(std::string s)
{
    std::vector<unsigned int> vec;
    unsigned int number;
    s += ' ';
    for (int i = 0; i < s.length(); i += 13)
    {
        number = 0;

        number += (HexToByte(s[i+9]) * 16) + HexToByte(s[i + 10]);

        number = number << 8;

        number += ((HexToByte(s[i+6]) * 16) + HexToByte(s[i + 7]));

        number = number << 8;

        number += ((HexToByte(s[i+3]) * 16) + HexToByte(s[i + 4]));

        number = number << 8;

        number += ((HexToByte(s[i]) * 16) + HexToByte(s[i + 1]));

        vec.push_back(number);
    }
    return vec;
}

class Key {
public:
    std::vector<unsigned int> K;
    void input(std::vector<unsigned int> T) {
        for (unsigned int i = 0; i < 40; i++) {
            K.push_back(0);
        }
    }
};

```

```

T.push_back(T.size());
for (unsigned int i = T.size(); i < 15; i++) {
    T.push_back(0);
}
for (unsigned int j = 0; j < 4; j++) {
    for (unsigned int i = 0; i < 15; i++) {
        unsigned int temp = T[(i + 8) % 15] ^ T[(i + 13) % 15];
        temp = LeftCyclShift(3, temp);
        temp ^= 4 * i + j;
        T[i] = temp;
    }
    for (unsigned int k = 0; k < 4; k++) {
        for (unsigned int i = 0; i < 15; i++) {
            unsigned int temp = GetBits(9, T[(i + 14) % 15]);
            if (temp < 256) {
                temp = S0[temp];
            }
            else {
                temp = S1[temp % 256];
            }
            unsigned int temp1 = T[i] + temp;
            temp = LeftCyclShift(9, temp1);
            T[i] = temp1;
        }
        for (unsigned int i = 0; i < 10; i++) {
            K[10 * j + i] = T[(4 * i) % 15];
        }
    }
}
}
unsigned int i = 5;
while (i <= 35) {
    unsigned int jj = K[i] & 3;
    unsigned int w = K[i] | 3;
    unsigned int M = MakeMask(w);
    unsigned int p = B[jj];
    p = LeftCyclShift(GetBits(5, K[i - 1]), p);
    K[i] = w ^ (p & M);
    i += 2;
}
}
unsigned int getElement(unsigned int i) {
    i %= 40;

```



```

        return K[i];
    }
    unsigned int MakeMask(unsigned int w) {
        unsigned int M = 0;
        unsigned int ones = (1 << 10) - 1;
        unsigned int d = 1;
        while (d < 31 - 10) {
            unsigned int zerosOrOnes = (ones << d) & w;
            if (zerosOrOnes == 0 || zerosOrOnes == (ones << d)) {
                ++d;
                bool dd = zerosOrOnes > 0;
                while (d < 32 && ((1 << d) > 0) == dd) {
                    M |= 1 << d;
                    ++d;
                }
                --d;
                M ^= 1 << d;
            }
            ++d;
        }
        return M;
    }
    void show() {
        for (unsigned int i = 0; i < K.size(); i++) {
            std::cout << K[i] << " ";
        }
    }
};

class Block {
public:
    unsigned int subblock[4];
    Key K;
    unsigned int l, m, r;

    void input(unsigned int a, unsigned int b, unsigned int c, unsigned int d, Key k) {
        subblock[0] = a;
        subblock[1] = b;
        subblock[2] = c;

```

```

        subblock[3] = d;
        K = k;
    }

void show() {
    for (unsigned int i = 0; i < 4; i++) {
        std::cout << subblock[i] << " ";
    }
}

std::vector<unsigned int> GetBlock() {
    std::vector<unsigned int> GB;
    for (int i = 0; i < 4; i++) {
        GB.push_back(subblock[i]);
    }
    return GB;
}

void ForwardPass(unsigned int i) {
    unsigned int sub0;
    subblock[1] ^= S0[GetByte(0, subblock[0])];
    subblock[1] += S1[GetByte(1, subblock[0])];
    subblock[2] += S0[GetByte(2, subblock[0])];
    subblock[3] ^= S1[GetByte(3, subblock[0])];
    subblock[0] = RightCyclShift(24, subblock[0]);
    if (i == 0 || i == 4) {
        subblock[0] += subblock[3];
    }
    if (i == 1 || i == 5) {
        subblock[0] += subblock[1];
    }
    sub0 = subblock[0];
    for (unsigned int j = 0; j < 3; j++) {
        subblock[j] = subblock[j + 1];
    }
    subblock[3] = sub0;
}

void DBackwardPass(unsigned int i) {
    unsigned int sub0 = subblock[3];
    for (unsigned int j = 3; j > 0; j--) {
        subblock[j] = subblock[j - 1];
    }
}

```

```

    }
    subblock[0] = sub0;
    if (i == 0 || i == 4) {
        subblock[0] -= subblock[3];
    }
    if (i == 1 || i == 5) {
        subblock[0] -= subblock[1];
    }
    subblock[0] = LeftCyclShift(24, subblock[0]);
    subblock[3] ^= S1[GetByte(3, subblock[0])];
    subblock[2] -= S0[GetByte(2, subblock[0])];
    subblock[1] ^= S1[GetByte(1, subblock[0])];
    subblock[1] ^= S0[GetByte(0, subblock[0])];
}

```

```

void E(unsigned int subblock, unsigned int i) {
    unsigned int M = subblock;
    M += K.getElement(2 * i + 4);
    unsigned int R = subblock;
    R = LeftCyclShift(13, R);
    R *= K.getElement(2 * i + 5);
    unsigned int j = GetBits(9, M);
    unsigned int L;
    if (j < 256)
        L = S0[j];
    else
        L = S1[j % 256];
    M = LeftCyclShift(GetBits(5, R), M);
    L ^= R;
    R = LeftCyclShift(5, R);
    L ^= R;
    L = LeftCyclShift(GetBits(5, R), L);
    l = L;
    r = R;
    m = M;
}

```

```

void Crypto(unsigned int i) {
    E(subblock[0], i);
    unsigned int O1, O2, O3, sub0;
    O1 = l;
    O2 = m;

```

```

O3 = r;
subblock[0] = LeftCyclShift(13, subblock[0]);
subblock[2] += O2;
if (i < 8) {
    subblock[1] += O1;
    subblock[3] ^= O3;
}
else {
    subblock[3] += O1;
    subblock[1] ^= O3;
}
sub0 = subblock[0];
for (unsigned int j = 0; j < 3; j++) {
    subblock[j] = subblock[j + 1];
}
subblock[3] = sub0;
}

void DCrypto(unsigned int i) {
    unsigned int O1, O2, O3;
    unsigned int sub0 = subblock[3];
    for (unsigned int j = 3; j > 0; j--) {
        subblock[j] = subblock[j - 1];
    }
    subblock[0] = sub0;
    subblock[0] = RightCyclShift(13, subblock[0]);
    E(subblock[0], i);
    O1 = l;
    O2 = m;
    O3 = r;
    subblock[2] -= O2;
    if (i < 8) {
        subblock[1] -= O1;
        subblock[3] ^= O3;
    }
    else {
        subblock[3] -= O1;
        subblock[1] ^= O3;
    }
}

void BackwardPass(unsigned int i) {

```

```

        if (i == 2 || i == 6) {
            subblock[0] -= subblock[3];
        }
        if (i == 3 || i == 7) {
            subblock[0] -= subblock[1];
        }
        subblock[1] ^= S1[GetByte(0, subblock[0])];
        subblock[2] -= S0[GetByte(3, subblock[0])];
        subblock[3] -= S1[GetByte(2, subblock[0])];
        subblock[3] ^= S0[GetByte(1, subblock[0])];
        subblock[0] = LeftCyclShift(24, subblock[0]);
        unsigned int sub0 = subblock[0];
        for (unsigned int j = 0; j < 3; j++) {
            subblock[j] = subblock[j + 1];
        }
        subblock[3] = sub0;
    }
    void DForwardPass(unsigned int i) {
        unsigned int sub0 = subblock[3];
        for (unsigned int j = 3; j > 0; j--) {
            subblock[j] = subblock[j - 1];
        }
        subblock[0] = sub0;
        subblock[0] = RightCyclShift(24, subblock[0]);
        subblock[3] ^= S0[GetByte(1, subblock[0])];
        subblock[3] += S1[GetByte(2, subblock[0])];
        subblock[2] += S0[GetByte(3, subblock[0])];
        subblock[1] ^= S1[GetByte(0, subblock[0])];
        if (i == 2 || i == 6) {
            subblock[0] += subblock[3];
        }
        if (i == 3 || i == 7) {
            subblock[0] += subblock[1];
        }
    }
    void AddKey() {
        for (int i = 0; i < 4; i++) {
            subblock[i] += K.getElement(i);
        }
    }

```

```

void DAddKey() {
    for (int i = 0; i < 4; i++) {
        subblock[i] += K.getElement(36 + i);
    }
}

void SubKey() {
    for (int i = 0; i < 4; i++) {
        subblock[i] -= K.getElement(36 + i);
    }
}

void DSubKey() {
    for (int i = 0; i < 4; i++) {
        subblock[i] -= K.getElement(i);
    }
}

std::vector<unsigned int> ToInt32() {
    std::vector<unsigned int> arr;
    for (int i = 0; i < 4; i++) {
        arr.push_back(subblock[i]);
    }
    return arr;
}

};

std::vector<Block> IntToBlock(std::vector<unsigned int> bls, Key k) {
    std::vector<Block> blocks;
    Block block;
    for (int i = 0; i < bls.size() / 4; i++) {
        block.input(bls[i * 4], bls[i * 4 + 1], bls[i * 4 + 2], bls[i * 4 + 3], k);
        blocks.push_back(block);
    }
    return blocks;
}

class MARS {
public:

```

```

Key key;
std::vector<unsigned int> M;
std::vector<Block> blocks;
void inp_key(std::vector<unsigned int> k) {
    key.input(k);
}

void code(std::vector<unsigned int> m) {
    blocks.clear();
    if (m.size() % 4 != 0) {
        int temp = m.size();
        for (int i = 0; i < 4 - (temp % 4); i++) {
            m.push_back(0);
        }
    }
    int i = 0;
    while (i < m.size()) {
        Block block;
        block.input(m[i], m[i + 1], m[i + 2], m[i + 3], key);
        block.AddKey();
        for (int j = 0; j < 8; j++) {
            block.ForwardPass(j);
        }
        for (int j = 0; j < 16; j++) {
            block.Crypto(j);
        }
        for (int j = 0; j < 8; j++) {
            block.BackwardPass(j);
        }
        block.SubKey();
        blocks.push_back(block);
        i += 4;
    }
}

void decode(std::vector<Block> B) {
    blocks = B;
    M.clear();
    for (int i = 0; i < B.size(); i++) {
        blocks[i].DAddKey();
        for (int j = 7; j >= 0; j--) {
            blocks[i].DForwardPass(j);

```

```

        }
        for (int j = 15; j >= 0; j--) {
            blocks[i].DCrypto(j);
        }
        for (int j = 7; j >= 0; j--) {
            blocks[i].DBackwardPass(j);
        }
        blocks[i].DSubKey();
        for (int j = 0; j < 4; j++) {
            M.push_back(blocks[i].subblock[j]);
        }
    }
}

void show() {
    for (int i = 0; i < blocks.size(); i++) {
        std::cout << std::endl << "*****" << std::endl;
        blocks[i].show();
        std::cout << std::endl << "*****" << std::endl;
    }
}

void dshow() {
    std::cout << std::endl << "*****" << std::endl;
    for (int i = 0; i < M.size(); i++) {
        std::cout << M[i] << " ";
    }
    std::cout << std::endl << "*****" << std::endl;
}

std::vector<unsigned int> GetBlocks() {
    std::vector<unsigned int> GB;
    for (int i = 0; i < blocks.size(); i++) {
        for (unsigned int j = 0; j < 4; j++) {
            GB.push_back(blocks[i].subblock[j]);
        }
    }
    return GB;
}

Key getKey() {
    return key;
}

};

```



```

int main()
{
    setlocale(LC_ALL, "Russian");
    MARS mars;
    Block block;
    int c, x;
    unsigned int z;
    std::vector<Block> B1, b11;
    std::vector<unsigned int> V, G;
    std::string s, s1, s2;
    std::vector<unsigned int> T;
    std::cout << "Введите ключ(от 4 до 14 символов)" << std::endl;
    std::cin >> s;
    T = StringToInt(s);
    mars.inp_key(T);
    do {
        std::cout << std::endl << "Выберите действие:\n0-Выход\n1-Зашифровать\n2-
Дешифровать" << std::endl;
        std::cin >> c;
        if (c == 1) {

            std::cout << "Шифровка сообщения" << std::endl << "Введите текст для
шифровки" << std::endl;

            getline(std::cin, s);
            getline(std::cin, s);
            V = StringToInt(s);
            mars.code(V);
            B1 = mars.blocks;
            V = mars.GetBlocks();
            std::cout << "Количество блоков:" << V.size() / 4 << std::endl << "Зашифро-
ванное сообщение:";

            for (int i = 0; i < V.size(); i++) {
                std::cout << IntToHex(V[i]) << " ";
            }
        }
        if (c == 2) {
            std::cout << "Дешифровка сообщения" << std::endl;
            V.clear();
            std::cout << "Введите количество блоков в шифре" << std::endl;
            std::cin >> x;
            std::cout << "Введите зашифрованное сообщение" << std::endl;
            getline(std::cin, s);

```

```

        getline(std::cin, s);
        V = HexToInt(s);
        mars.decode(IntToBlock(V, mars.key));
        std::cout << "Расшифрованное сообщение\n" << StringFromInt(mars.M) <<
std::endl;
    }
    s = "";

    } while (c > 0);
    return 0;
}

```

## ПРИЛОЖЕНИЕ В

```
#include <iostream>
```

```

int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n - 1) + fib(n - 2);
}

```

```

int main()
{
    std::cin>>n;
    std::cout << fib (n);
    return 0;
}

```

```
#include <iostream>
```

```

int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n - 1) + fib(n - 2);
}

```

```

int fib1(int n)
{
    if (n <= 1)
        return n;
    return fib(n - 1) + fib1(n - 2);
}

```

```

int fib2(int n)
{
    if (n <= 1)
        return n;
    return fib1(n - 1) + fib2(n - 2);
}

```

```

int main()
{
    std::cin>>n;
    std::cout << fib2(n);
}

```

```
    return 0;  
}
```