**«Санкт-Петербургский государственный электротехнический университет**

**«ЛЭТИ» им. В.И.Ульянова (Ленина)»**

**(СПбГЭТУ «ЛЭТИ»)**

|  |  |  |
| --- | --- | --- |
| **Направление** | 09.04.01 – Информатика и вычислительная техника | |
| **Программа** | Распределенные интеллектуальные системы и технологии | |
| **Факультет** | КТИ | |
| **Кафедра** | ВТ | |
| *К защите допустить* |  | |
| Зав. кафедрой, д. т. н., профессор |  | М. С. Куприянов |

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

магистра

Тема: ДИНАМИЧЕСКАЯ ОПТИМИЗАЦИЯ МАШИННОГО КОДА ДЛЯ RISC-V

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Студент |  |  |  | С. А. Репин |
|  |  |  |  |  |
| Руководитель | к. т. н., доцент |  |  | А. А. Пазников |
|  |  |  |  |  |
| Консультанты | к. э. н., доцент |  |  | М. Н. Магомедов |
|  |  |  |  |  |
|  |  |  |  | С. Е. Ильин |
|  |  |  |  |  |

Санкт-Петербург

2024**ЗАДАНИЕ**

**на выпускную квалификационную работу**

|  |  |
| --- | --- |
|  | Утверждаю |
|  | Зав. кафедрой ВТ  д. т. н., профессор |
|  | \_\_\_\_\_\_\_\_\_\_\_\_ М. С. Куприянов |
|  | «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_ г. |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Студент | С. А. Репин | | | |  | Группа | 8307 | |
| Тема работы: Динамическая оптимизация машинного кода для RISC-V. | | | | | | | | |
| Место выполнения ВКР: СПбГЭТУ «ЛЭТИ»  Исходные данные: спецификация архитектуры RISC-V, документация к оптимизациям LLVM, GNU GCC, RyuJIT. | | | | | | | | |
| Технические требования: использовать спецификацию архитектуры RISC-V RV64GC; применять языки C++20, C11, язык ассемблера RISC-V; целевая платформа – Linux; обеспечить поддержку произвольных оптимизаций. | | | | | | | | |
| Содержание ВКР: основы оптимизации программного кода, анализ и выбор JIT-оптимизаций МК, инфраструктура JIT-компилятора МК. | | | | | | | | |
| Перечень отчетных материалов: пояснительная записка, иллюстративный материал. | | | | | | | | |
| Дополнительные разделы: Оценка и защита результатов интеллектуальной деятельности. | | | | | | | | |
|  | | | | | | | | |
| Дата выдачи задания | | | Дата представления ВКР к защите | | | | | |
| «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_ г. | | | «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_ г. | | | | | |
|  | | |  | | | | | |
| Студент | |  | | С. А. Репин | | | |
| Руководитель к. т. н., доцент | |  | | А. А. Пазников | | | |

**календарный план выполнения**

**выпускной квалификационной работы**

|  |  |
| --- | --- |
|  | Утверждаю |
|  | Зав. кафедрой ВТ  д. т. н., профессор |
|  | \_\_\_\_\_\_\_\_\_\_\_\_ М. С. Куприянов |
|  | «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_ г. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Студент | С. А. Репин |  | Группа | 8307 |
| Тема работы: Динамическая оптимизация машинного кода для RISC-V | | | | |

|  |  |  |
| --- | --- | --- |
| № п/п | Наименование работ | Срок выполнения |
| 1 | Анализ существующих решений по оптимизации программ | 05.02–25.02 |
| 2 | Формирование требований к проекту | 26.02–01.03 |
| 3 | Анализ программных оптимизаций, применимых для машинного кода | 02.03–16.03 |
| 4 | Проектирование архитектуры инфраструктурного кода | 17.03–24.03 |
| 5 | Реализация инфраструктурного кода | 25.03–22.04 |
| 6 | Тестирование, измерение производительности | 23.04–30.04 |
| 7 | Оформление пояснительной записки | 01.05–13.05 |
| 8 | Подготовка иллюстративного материала | 14.05–17.05 |
| 9 | Предварительное рассмотрение работы | 18.05–28.05 |
| 10 | Представление работы к защите | 31.05 |

|  |  |  |
| --- | --- | --- |
| Студент |  | С. А. Репин |
| Руководитель к.т.н., доцент |  | А. А. Пазников |
|  |  |  |

**РЕФЕРАТ**

Пояснительная записка содержит: 74 с., 10 рис., 11 табл., 32 источн., 1 прил.

КОМПИЛЯТОРЫ, ОПТИМИЗАЦИЯ ПРОГРАММ, СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ, АРХИТЕКТУРА RISC-V, ДИНАМИЧЕСКАЯ КОМПИЛЯЦИЯ

Цель работы представляет собой исследование динамических оптимизаций для машинного кода, а также разработку библиотеки на языке C++, позволяющую выполнять динамическую оптимизацию исполняемых файлов в операционной системе GNU/Linux.

Для достижения исследовательских целей использовались такие методы, как анализ, классификация и эксперимент.

Дипломная работа содержит обзор существующих подходов в области оптимизации программного обеспечения, введение в архитектуру RISC-V. Представлен анализ этих подходов и сформулированы недостатки каждого из них, которые в совокупности приводят к целесообразности выполнения оптимизаций для машинного кода во время работы оптимизируемой программы.

В ходе работы выполнено проектирование и реализация библиотеки, которая предоставляет возможности для создания и применения различных динамических оптимизаций над машинным кодом. В качестве демонстрации и подтверждения работы была реализована одна из оптимизаций, в результате тестирования которой было показано ее положительное влияние на скорость работы тестовой программы. Результат дипломной работы может быть использован в самых разных областях для оптимизации работы произвольных прикладных программ, а также для построения полнофункционального JIT-компилятора машинного кода любых архитектур.

**ABSTRACT**

The purpose of this work is to study dynamic optimizations of machine code and to develop a C++ library to perform dynamic optimization of executables for GNU/Linux.

Such methods as analysis, classification and experimentation were used to achieve the research goals.

The thesis contains a review of existing approaches in the field of software optimization, an introduction to the RISC-V architecture. The analysis of these approaches is presented and disadvantages of each of them are formulated, which together lead to the expediency of performing optimizations of machine code while the optimized program is running.

The work involves the design and implementation of a library that provides facilities for creating and applying various dynamic optimizations for machine code. As a demonstration and confirmation of the work one of the optimizations was implemented, as a result of testing it was shown its positive impact on the speed of a test program. The result of the thesis can be used in various fields to optimize the work of arbitrary application programs, as well as to build a full-featured JIT-compiler of machine code of any architecture.

**СОДЕРЖАНИЕ**

[ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ 8](#_Toc167746781)

[ВВЕДЕНИЕ 9](#_Toc167746782)

[1 Основы оптимизации программного кода 13](#_Toc167746783)

[1.1 Сущность оптимизации программного кода 13](#_Toc167746784)

[1.2 Статические оптимизации компиляторов 15](#_Toc167746785)

[1.3 Оптимизации на стадии линковки 18](#_Toc167746786)

[1.4 Оптимизации на основе профилирования 18](#_Toc167746787)

[1.5 Оптимизация машинного кода на основе профилирования 20](#_Toc167746788)

[1.6 Динамические оптимизации (JIT-компиляция) 20](#_Toc167746789)

[1.7 Система динамической инструментации кода DynamoRIO 24](#_Toc167746790)

[1.8 Динамические оптимизации машинного кода 26](#_Toc167746791)

[2 Анализ и выбор JIT-оптимизаций МК 28](#_Toc167746792)

[2.1 Сведения об архитектуре RISC-V 28](#_Toc167746793)

[2.2 Выравнивание кода 30](#_Toc167746794)

[2.3 Векторизация кода 32](#_Toc167746795)

[2.4 Встраивание функций 34](#_Toc167746796)

[2.5 Оптимизация расположения кода 35](#_Toc167746797)

[3 Инфраструктура JIT-компилятора МК 38](#_Toc167746798)

[3.1 Алгоритм работы оптимизатора 38](#_Toc167746799)

[3.2 Исполнение динамического кода 41](#_Toc167746800)

[3.3 Трамплин и выход из оптимизированного кода 43](#_Toc167746801)

[3.4 Декодер и энкодер инструкций 46](#_Toc167746802)

[3.5 Процесс построения графа потока управления 49](#_Toc167746803)

[3.6 Релокация кода 50](#_Toc167746804)

[3.7 Описание архитектуры 52](#_Toc167746805)

[3.8 Автоматизирование динамических оптимизаций 55](#_Toc167746806)

[3.9 Python-скрипт для выполнения бенчмарков 56](#_Toc167746807)

[3.10 Пример работы динамической оптимизации 57](#_Toc167746808)

[4 Оценка и защита результатов интеллектуальной деятельности 61](#_Toc167746809)

[4.1 Описание результата интеллектуальной деятельности 61](#_Toc167746810)

[4.2 Оценка рыночной стоимости результата интеллектуальной деятельности 62](#_Toc167746811)

[4.3 Правовая защита результатов интеллектуальной деятельности 67](#_Toc167746812)

[ЗАКЛЮЧЕНИЕ 71](#_Toc167746813)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 72](#_Toc167746814)

[ПРИЛОЖЕНИЕ А Фрагменты исходного кода проекта 75](#_Toc167746815)

# ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

В настоящей пояснительной записке применяются следующие термины, обозначения и сокращения с соответствующими определениями:

Ассемблер – представление команд процессора (машинного языка) в понятном для человека виде

ББ – базовый блок

Кэш-память процессора – высокоскоростная память, применяющаяся внутри процессора для уменьшения времени доступа к оперативной памяти

МК – машинный код

Опкод инструкции – число, составная часть инструкции, характеризующая операцию, которая должна быть выполнена процессором

Application binary interface (ABI) – описание способов взаимодействия между двумя бинарными (скомпилированными) программами (или частями одной)

Application programming interface (API) – описание способов взаимодействий одной компьютерной программы (или части компьютерной программы) с другой

Data Plane Development Kit (DPDK) – фреймворк для ускорения процессов обработки сетевых пакетов

JIT-компиляция – just-in-time компиляция (динамическая компиляция).

PGO – profile guided optimization

# ВВЕДЕНИЕ

Эффективность (производительность) работы программы, которая обычно представляется двумя показателями: скорость работы и объем потребляемой памяти [1], было и остается одним из ключевых нефункциональных требований для программного обеспечения [2].

С одной стороны, эффективность важна для обеспечения высокого уровня пользовательского опыта при взаимодействии с программой (отзывчивость интерфейса, возможность параллельной работы с другими программами и другое). Так, например, компания Google проводило исследование, в ходе которого установила, что увеличение длительности загрузки страницы c 1 секунды до 3-5 секунд повышает шанс того, что пользователь покинет сайт, на 32-90% [3].

С другой стороны, ряд задач вообще не решаются без внимательного и трудозатратного подхода к оптимизации ПО. В данном контексте интересна так называемая проблема C10k [4], которая проявилась в конце 1990-х годов. Она заключается в невозможности без специальной оптимизации алгоритмов обрабатывать 10 тысяч соединений параллельно, хотя оборудование того времени теоретически обеспечивало должную производительность. Решением проблемы является применение асинхронного программирования или пулов потоков, в то время как распространенные программы той эпохи выделяли на каждое соединение новый поток и обслуживали его синхронными системными вызовами. Сейчас эта задача трансформировалась в проблему C10M, т.е. обработку 10 миллионов параллельных соединений. Другим примером является фреймворк DPDK [5], который создан компанией Intel для выполнения сетевых операций в обход ядра операционной системы. Оказывается, что генерация UDP трафика пакетами 64 байта на максимальной скорости (linerate) 10-гигабитного Ethernet с использованием сетевого стека операционной системы невозможна в принципе. Для подобных целей DPDK оказывается незаменим.

Обходить проблему неэффективной работы программы можно двумя способами: масштабированием системы или оптимизацией кода [6]. Первое заключается в расширении аппаратных возможностей (установка более мощных процессоров, большей оперативной памяти, переход на SSD накопители и т.д.), а вторая — в развитии алгоритмов и оптимизаций программного обеспечения. Выбор способа зависит от возможностей бизнеса: оправданы ли больше инвестиции в оборудование или инвестиции в разработчиков. В данной работе будет рассмотрен второй подход, который предполагает улучшение программного обеспечения с целью повышения скорости его работы и уменьшения потребляемой оперативной памяти.

Оптимизацию программы можно разделить на четыре этапа [7]:

1. Выбор наиболее подходящего алгоритма для решения задачи (например, использовать протокол UDP вместо TCP, когда скорость передачи важнее надежности).
2. Оптимизации существующих алгоритмов и структур данных (small string optimization для std::string в стандартной библиотеки C++; copy on write для быстрого создания потоков в UNIX).
3. Применение оптимизаций компилятора, которые выполняются с минимальным участием программиста (флаги -O3 в GCC и Clang; встроенной функции \_\_builtin\_expect() в GCC и Clang для подсказок компилятору о наиболее вероятных ветках в конструкции if/else).
4. Ручная оптимизация исходного кода программы (переписывание участков программы на ассемблере; использование интринсиков векторных инструкций).

Список выше ранжирован, т.е. с каждым шагом повышается сложность разработки и требования к программисту. Чаще всего в стандартных невысоконагруженных приложениях останавливаются на первом или третьем этапах, оставляя остальные оптимизации за компилятором и разработчиками стандартных библиотек языков программирования, сторонних библиотек и фреймворков. Кроме того, этап 4 весьма специализирован: компиляторы становятся все «умнее» и реализуют все больше оптимизаций. Доходит до того, что код на ассемблере написанный вручную чаще будет работать медленнее, чем сгенерированный компилятором.

Но даже высоконагруженные приложения редко используют глубокие оптимизации (уровня 4) и для этого есть ряд причин. Во-первых, как сказано выше, весьма нетривиально написание кода эффективнее компилятора. Во-вторых, такие оптимизации требуют изначальной разработки программы на достаточно низкоуровневых языках типа C, C++, Rust, что может сильно затянуть разработку и потребует больших инвестиций в проект. В-третьих, применения оптимизаций 4-ого этапа ведет к зависимости от аппаратного обеспечения: фиксируется архитектура процессора и семейство процессоров, для которого разрабатывается ассемблерный код. Например, если программа использует векторные команды AVX-512 архитектуры x86-64 (Intel/AMD), то она не сможет работать на тех процессорах x86-64, которые не имеют расширения AVX. Это ситуация особенно интересна, потому что сказывается и на компиляторных оптимизациях: компиляция все равно выполняется для конкретного семейства процессоров, поэтому программа не будет работать если компилятор применял те инструкции, которых нет на пользовательском процессоре. При распространении программ нужно опираться на те семейства процессоров, которые наиболее распространены на рынке, из-за чего пользователи с современными процессорами могут не получать всей той производительности, которую их процессор мог бы дать.

С другой стороны, существуют языки программирования, которые компилируются в промежуточный переносимый код (C#, Java, Go, JavaScript). Во время выполнения специальная виртуальная машина, выступая в роли JIT-компилятора (just in time компилятор, компилятор времени исполнения), генерирует и тут же исполняет машинный код на основе промежуточного. Тем самым такие языки, с одной стороны, могут генерировать код под конкретный процессор пользователя, а с другой, применять информацию времени исполнения для оптимизации, получая еще более эффективный код, чем мог бы даже в теории сгенерировать AOT-компилятор (ahead of time компилятор; применяется в C, C++, D, Rust).

Из всего вышеперечисленного выходит следующий вопрос: возможно ли реализовать JIT-компилятор для машинного кода — результата работы обычных компиляторов, т.е. создать такой компилятор, который во время исполнения целевой программы автоматически бы применял ряд оптимизаций для нее, учитывая окружение, в которой происходит исполнение?

Проработка этого вопроса представлена в данной работе. Проводится анализ и предлагается решение для следующих задач:

* реализация инфраструктуры для динамической оптимизации произвольного машинного кода;
* выбор и исследование оптимизаций, которые имеет смысл производить во время выполнения программы. Эти оптимизации не должны быть заранее применены статическим компилятором;
* разработка эскизного проекта, который сможет выполнять такие оптимизации для архитектуры RISC-V (проект назван binboost);

Объектом работы является машинный код для RISC-V.

Предмет работы – динамическая оптимизация машинного кода RISC-V.

Цель работы представляет собой исследование динамических оптимизаций для машинного кода RISC-V и разработку динамического оптимизатора машинного кода.

В первой главе приводится анализ различных методов оптимизации программного обеспечения, приводятся доводы в пользу динамической оптимизации машинного кода. Во второй главе производится обзор оптимизаций, пригодных для динамического применения к машинному коду. В третьей главе приводится описание архитектуры библиотеки binboost, представляющей инструменты для создания динамических оптимизаций.

# 1 Основы оптимизации программного кода

В разделе содержится обзор и анализ существующих решений для оптимизации программного кода. Приводятся недостатки каждого из решений. Формулируются принципы и способы динамической оптимизации машинного кода, включая требования к разрабатываемому в дипломной работе проекту.

## 1.1 Сущность оптимизации программного кода

Как уже было сказано выше, кроме корректности к программам выдвигаются и другие требования, в которые входит такое немаловажное, как скорость работы и потребление памяти. Так как большинство современных программ разрабатывается на языках высокого уровня, то важным элементом, влияющим на их производительность, является то, на сколько эффективный машинный код генерируется компилятором во время этапа компиляции. Но кроме компиляции есть и другие стадии, на которых тоже возникают задачи, от решения которых зависит производительность. В частности, это этапы непосредственного написания кода (программист должен выбрать наилучшие алгоритмы, структуры данных и другие решения в тот самый момент, когда он создает исходный код программы), линковки (создания итогового исполняемого файла после преобразования исходного текста в машинный промежуточный код), JIT-компиляции (для языков интерпретируемых и/или имеющих виртуальную машину).

Обычно под оптимизацией кода подразумевают применение различных преобразований кода для улучшения его свойств, повышения эффективности работы. Оптимизации не должны сказываться на корректности программы, т.е. методы оптимизации должны обладать свойством изоморфности.

Программист может высокоуровнево оптимизировать программы. Для этого он выбирает наиболее подходящие для данной задачи структуры данных и алгоритмы работы с ними с учетом архитектуры ЭВМ, сложности алгоритмов, подходов многопоточности и т. д. В дальнейшем оптимизируемый компилятор и различные сторонние инструменты в свою очередь попытаются применить низкоуровневые оптимизации, связанные с удалением «мертвого» кода, перестановками кода, встраиванием функций, упрощением математических операций (например, сдвиг вместо умножения) и другим, более сложным.

Высокоуровневые оптимизации часто сопряжены с решением компромиссов, т.к. высокооптимизированный код нередко сильно теряет в таких характеристиках, как поддерживаемость и читаемость [8]. Например, приходится снижать число непрямых вызовов функций, что сводит на нет преимущества объектно-ориентированного кода. Или приходится использовать более сложные структуры данных, такие как lock-free коллекции, вместо стандартных. Или применять сложнопонимаемые неподготовленным разработчиком структурные паттерны, типа устройства Даффа (пример его использования в фреймворке DPDK приведен ниже). Как видно, ручная оптимизация требует также и высоких компетенций от программиста.

// Функция аллокации памяти для структур rte\_mbuf.

static inline int rte\_pktmbuf\_alloc\_bulk(

struct rte\_mempool \*pool,

struct rte\_mbuf \*\*mbufs, unsigned count)

{

unsigned idx = 0;

int rc;

// Память для rte\_mbuf выделяется в пуле памяти.

rc = rte\_mempool\_get\_bulk(pool, (void \*\*)mbufs, count);

if (unlikely(rc))

return rc;

// На основе логики конструкции switch, будет изменено

// место, с которого начнется выполняться тело цикла

// while.

//

// Внутри цикла while для каждой созданной структуры

// выполняется ее инициализация по умолчанию (сброс).

switch (count % 4) {

case 0:

while (idx != count) {

\_\_rte\_mbuf\_raw\_sanity\_check(mbufs[idx]);

rte\_pktmbuf\_reset(mbufs[idx]);

idx++;

/\* fall-through \*/

case 3:

\_\_rte\_mbuf\_raw\_sanity\_check(mbufs[idx]);

rte\_pktmbuf\_reset(mbufs[idx]);

idx++;

/\* fall-through \*/

case 2:

\_\_rte\_mbuf\_raw\_sanity\_check(mbufs[idx]);

rte\_pktmbuf\_reset(mbufs[idx]);

idx++;

/\* fall-through \*/

case 1:

\_\_rte\_mbuf\_raw\_sanity\_check(mbufs[idx]);

rte\_pktmbuf\_reset(mbufs[idx]);

idx++;

/\* fall-through \*/

}

}

return 0;

}

Оптимизируя программу, нельзя обойтись и без детального представления того, на каком оборудовании будет она работать. На практике оказывается, что для каждой архитектуры (а иногда даже отдельном семействе процессоров одной архитектуры) существуют свои специфичные особенности, которые важно учитывать. Так, например, нужно учитывать размеры кэшей данных и инструкций, размеры и устройство конвейера, ограничения отдельных ядер процессора (E- и P-ядра современных процессоров Intel, технология TurboBoost и Hyper-threading).

По этим причинам все большее развитие получают автоматические оптимизации: они позволяют снизить когнитивную нагрузку на разработчиков и сконцентрировать их внимание на решении бизнес-задач.

## 1.2 Статические оптимизации компиляторов

Оптимизирующий компилятор содержит так называемые проходы (passes, в терминологии LLVM [9]), которые выполняют обход по некоторой части программы и либо собирают полезную для других проходов информацию, либо выполняют трансформации программы. Проходы могут выполняться как для абстрактного синтаксического дерева (АСТ) исходного программы (т.е. на фронтенде), так и для АСТ промежуточного кода – кода, генерируемого на основе исходного, из которого в дальнейшем получается машинный код (бэкэнд компилятора).

В список стандартных оптимизаций входят: разворачивание циклов (дублирование тела циклов, чтобы снизить число прыжков), удаление общего подвыражения, вынесение неизменного кода из цикла, переупорядочивание кода, удаление «мертвого» кода, вычисление константных выражений, автоматическая векторизация и другие.

Распространено мнение [8], что ряд оптимизаций, которые поддерживает компилятор, на практике невозможно реализовать программисту самостоятельно за счет их сложности. В примере ниже демонстрируется, как компиляторы умеют оптимизировать деления, заменяя их на умножения и сдвиги: выполнив сложный математически расчет, компилятор может найти такие константы, что умножение и сдвиг с ней даст тот же результат, что и деление (см. таблицу 1.1).

Таблица 1.1 – Оптимизация деления. Вместо деления сгенерировано умножение на 0x17d81b1, несколько сдвигов, вычитание и сложение

|  |  |
| --- | --- |
| Исходный код | Оптимизированный код |
| unsigned int  foo(unsigned int x)  {  return x / 3747;  } | foo(unsigned int):  mov %edi,%eax  imul $0x17d819b1,%rax,%rax  shr $0x20,%rax  sub %eax,%edi  shr %edi  add %edi,%eax  shr $0xb,%eax  ret |

А учитывая специфику целевой платформы, компилятор даже может вернуть в исходное состояние код, оптимизированный руками, потому что умножение может быть на данной платформе быстрее, чем десяток инструкций с умножением, сложениями, вычитаниями и сдвигами.

Но какими бы умными не были компиляторы, они не в силах предугадать все варианты использования кода и, что важно, весь спектр оборудования, которое будет применяться у пользователей (чаще всего разработчики программ стремятся к тому, чтобы эта программа могла работать у как можно большего числа пользователей). С другой стороны, ряд оптимизаций можно применять при наличии информации, которая доступна только во время выполнения (например, сколько итераций будет у цикла в реальности).

Крупные программы состоят из множества отдельных модулей (translation unit в терминологии стандарта C++), которые компилируются раздельно. Из-за этого компилятор не сможет пользоваться информацией об использовании кода из другого модуля. Например, компилятор не сможет встроить функцию, тело которой находится в отдельном translation unit (т.е. которая не реализована в заголовочном файле, если говорить об языках C или C++).

Но компиляторы не могут быть и идеальными. Огромные коллективы программистов десятилетиями работают над компиляторами, но все же не все возможные оптимизации еще реализованы, а с учетом постоянного развития процессоров, компиляторы не всегда за ними успевают. Как было установлено экспериментальным путем, компиляторы всегда генерируют наиболее эффективный код для используемой платформы, программист может вручную соптимизировать лучше: учесть выравнивание кода циклов, применить интринсики для векторных инструкций, встроить инструкции условного переноса вместо ветвления и т.п. [7]. Это приводит к тому, что некоторые модули программ, особенно чувствительных к производительности, оптимизируют вручную (например, подобного кода много в фреймфорке DPDK, в медиаконверторе ffmpeg, ядрах операционных систем).

## 1.3 Оптимизации на стадии линковки

Для решения упомянутой выше проблемы кроссмодульной оптимизации в компиляторы добавляется так называемая link-time optimization (LTO) – оптимизация на этапе линковки [10]. При этом компиляторы не требуют от разработчика-пользователя дополнительных действий – оптимизации абсолютно прозрачны для них. Компиляторы в этом случае опираются на поддержку линкера или отдельной подсистемы.

Компилятор во время обработки каждого модуля сохраняет специальную информацию, которой затем может воспользоваться на этапе линковки. Затем специальная подсистема компилятора может выполнить удаление ненужного кода с учетом использования его разными частями программы, удаление дублирований кода (например, повторные вызовы «чистых» функций), встраивание функций, места использования и определения которых находятся в разных модулях, оптимизировать функции на основе знания о их параметрах (включая constant propagation). Немаловажно, что компилятор теперь может опираться на анализ потоков данных всей программы целиком.

К сожалению, этот вид оптимизаций не решает проблем, озвученных в предыдущем разделе. В сущности, здесь не появляется новых оптимизаций, а только существующие оптимизации получают возможность быть исполненными с учетом знания обо всей программе, а не только ее части (модуля). Если компилятор до этого генерировал неэффективный код для отдельных функций или циклов, то включение link-time оптимизации эффекта не даст.

## 1.4 Оптимизации на основе профилирования

Выше упоминалось, что компилятор в своих оптимизациях лишен информации о конкретных использованиях программы. В частности, он может только гадать, какие участки кода будут «горячими» (часто используемыми), какие условия в конструкциях if/else будут выполняться чаще, сколько итераций чаще все будет в циклах и т.п.

В целях оптимизации на основе данных о реальном применении программы многие современные компиляторы поддерживают возможность своей работы на основе профиля программы. Программист во время тестирования программы с помощью сторонних средств (возможно, встроенных в компилятор) собирает перечисленную выше статистику во время непосредственной работы программы. После получения профиля программист выполняет перекомпиляцию программы, во время которой компилятор будет использовать переданный ему профиль.

Создание профиля программы (например, утилитой perf) заключается в сборе различных событий от ядра операционной системы и самого процессора (в процессорах Intel, к примеру, существует специальная подсистема для этого – Performance Monitoring Unit) во время работы программы. От операционной системы собираются события переключения контекста, отказ страниц (page fault), миграции между процессорами и другие. Процессор предоставляет информацию о микроархитектурных событиях: число циклов работы отдельных функций и даже блоков кода, промахи по кэшу, ошибки предсказателя переходов, число выполненных инструкций, полноту утилизации конвейера и другие.

В исследовании [11] указывается, что PGO позволяет увеличить производительность до 10-20% по сравнению с методами статического анализа.

Известной проблемой PGO является то, что профиль собирается при решении программой конкретной задачи. То есть в общем случае PGO позволит лишь соптимизировать ту часть программы (или те программы), нагрузка которых стабильна и не зависит от ввода пользователя или другой внешней, динамически меняющейся информации. Если в зависимости от обстоятельств, частота использования компонентов программы будет различаться, то PGO не сумеет дать большого роста производительности. Впрочем, профилирование стало уже важным этапом сборки крупных проектов: браузеров, игр, веб-сервисов, мощных графических приложений и других.

## 1.5 Оптимизация машинного кода на основе профилирования

В составе проекта компиляторной инфраструктуры LLVM есть утилита LLVM Bolt [12] – оптимизатор кода, работающий после линкера. Он применяется для ускорения крупных программ, изменяя расположение инструкций машинного кода на более оптимальное на основе профиля. Профиль собирается с помощью Linux-утилиты perf.

Bolt выполняет дизассемблирование функций из бинарного файла, построение графа управления, оптимизацию кода, генерацию (включая релокацию средствами линкера) видоизмененного машинного кода. По этим причинам Bolt не требует наличия доступа к исходному коду программы (может применяться для оптимизации сторонних библиотек) и при своей работе учитывает всю программу целиком. В состав поддерживаемых оптимизаций входит переупорядочивание базовых блоков на основе видоизмененного алгоритма Петтиса-Хансена.

На данный момент Bolt требует некоторых действий со стороны разработчика, а именно включения флага --emit-relocs и отключение удаления таблицы символов исполняемого файла.

Bolt ограничен в количестве применяемых оптимизаций (они в основном заключаются только в изменении порядка инструкций) и, также как LTO, требует сбора профиля работы программы, что снижает его эффективность в случае, когда программа функционирует по разному в зависимости от конкретных потребностей пользователя. В дальнейшем разработчики планируют реализовать встраивание, девиртуализацию, оптимизацию условных хвостовых вызовов, удаление ненужных загрузок данных, переупорядочивание данных программы.

## 1.6 Динамические оптимизации (JIT-компиляция)

Отдельную нишу занимают динамические оптимизации, используемые в языках высокого уровня (C#, Java). Компиляторы этих языков не производят машинный код, а вместо этого конвертируют (транслируют) исходный код в некоторый промежуточный платформонезависимый низкоуровневый язык (CIL, Java bytecode), предназначенный для выполнения на виртуальной машине (CLR, JVM). Таким образом, прямо во время выполнения программы специальная среда – виртуальная машина – преобразует (исполняет) промежуточный язык, компилируя его в машинный код. Этот процесс называется JIT-компиляция (just in time, «во время выполнения») – динамической компиляцией кода во время исполнения программы (пример работы среды выполнения .NET показан на рисунке 1.1).

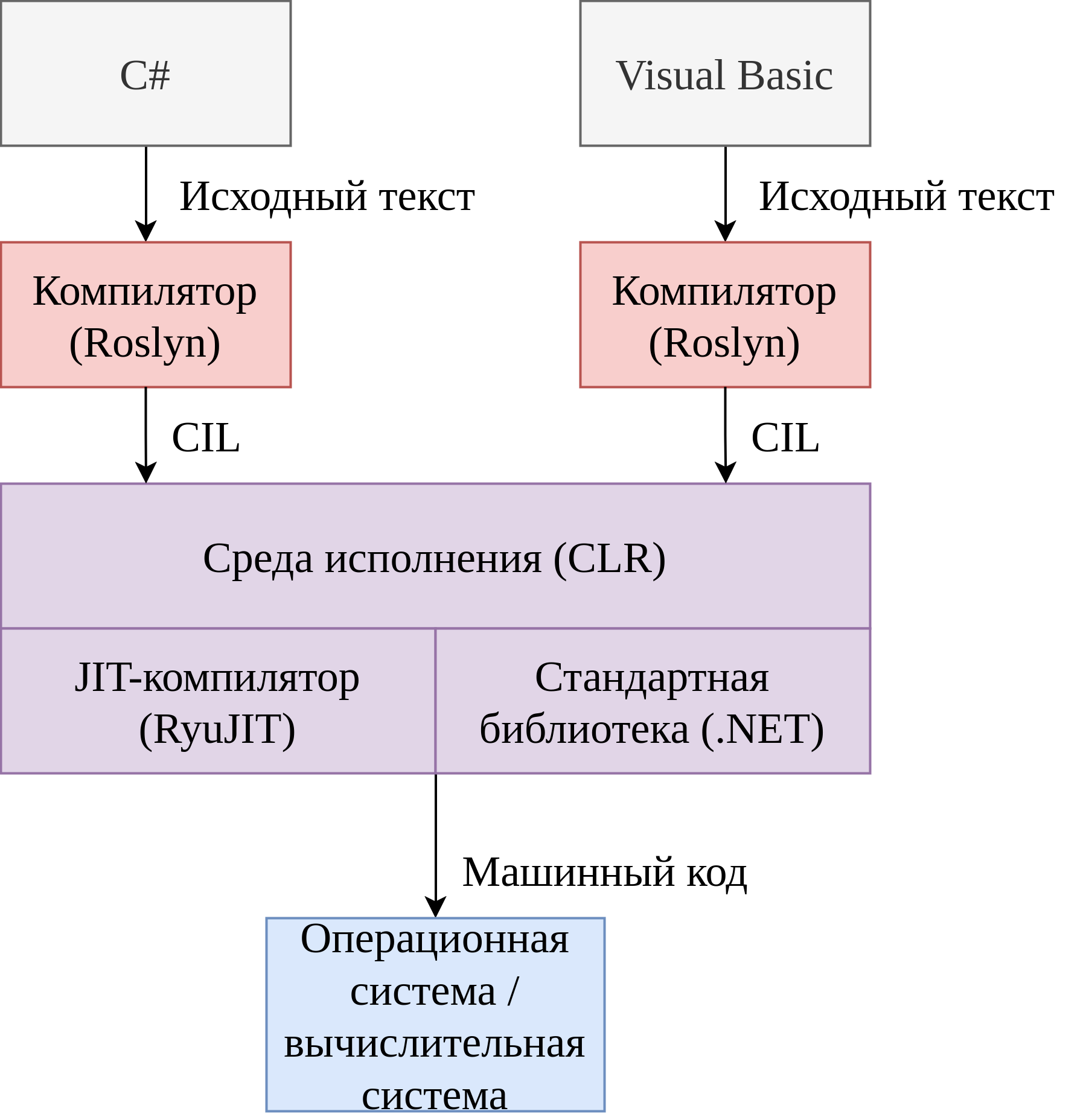


Рисунок 1.1 – Принцип работы среды выполнения .NET

Такой подход дает большие преимущества по сравнению со статической компиляцией (в контексте JIT она часто называется AOT-компиляция – ahead of time, «досрочная» компиляция).

Остановимся на рассмотрении JIT-компилятора языка C# (RyuJIT) [13], предполагая, что другие среды выполнения реализованы аналогично. Диаграмма работы всего RyuJIT представлена на рисунке 1.2.

На вход JIT-компилятора поступает заранее транслированный из поддерживаемых языков высокого уровня байт-код. Сперва – в фазе инициализации – RyuJIT создает и нормализует промежуточное представление для байт-кода и связанных с ним структур данных, строит граф потока управления, применяет простые оптимизации (такие, как встраивание функций). Дальше выполняется фаза оптимизации. И в завершении процесса динамической компиляции выполняет фаза бэкэнда, которые выполняет трансформации, необходимые для генерации машинного (нативного) кода. Для этого промежуточный код трансформируется в новый вид, удобный для кодогенерации. Затем производится аллокация регистров (т.е. назначение реальных регистров, число которых конечно, всем переменным, встречающимся в коде; если регистров не хватает, то часть переменных, наиболее редко использующихся, помещается на стек).

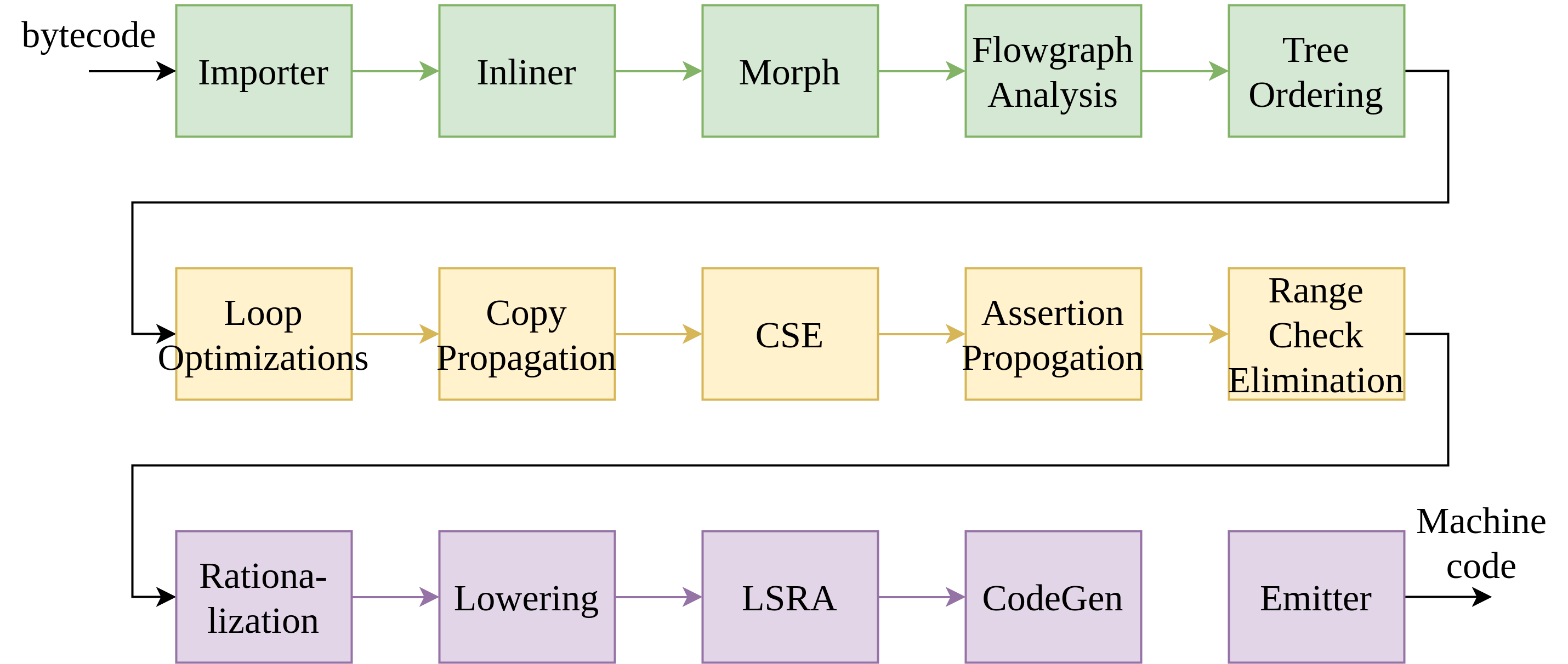


Рисунок 1.2 – Фазы работы RyuJIT

В соответствии с документацией фаза оптимизации состоит из следующих этапов (на диаграмме отмечены желтым цветом, расположены в середине):

* анализ потока данных, построение SSA-форм (single static assignment form, представление программы в виде инструкций, которые присваивают значения только переменным с уникальным именем, т.е. представление, в котором переменные нельзя перезаписывать);
* вынос инвариантов циклов;
* удаление копирований;
* удаление общих подвыражений;
* распространение проверок (распространение по коду различных инвариантов, таких как проверки размеров массивов и указателей на null);
* удаление повторяющихся или лишних проверок индексов массивов.

Как видно, перечисленные оптимизации RyuJIT полностью соответствуют статическим оптимизациям, применяемым для языков C и C++ без всяких JIT. Дело здесь в том, что JIT-компилятор способен автоматически без участия программиста компилировать входной промежуточный код для разных платформ: x86-64, x86, ARM64, ARM32. При этом большинство оптимизаций также платформонезависимы, переносить на новые архитектуры необходимо только бэкэнд – генератор машинного кода.

Важной особенностью является также и способность RyuJIT получать информацию от сборщика мусора и среды исполнения, включая данные о конкретном типе исполняющего процессора. Эта способность также называется адаптивной оптимизацией. Так, оптимизация hot/cold splitting (разделение «горячего» и «холодного» кода), которая улучшает локальность кода и уменьшает количество отказов страниц (page faults).

Еще одним примером будет технология Dynamic PGO (динамическая оптимизация на основе профиля), которая сводится к тому, что CLR во время работы программы собирает профиль, содержащий информацию о частоте вызова конкретных функций, блоков кода, веток условных конструкций, а затем оптимизирует (или «переоптимизирует») код с учетом этих данных о текущем использовании программы.

Аналогичный подход применяется в Java. Самая популярная виртуальная машина HotSpot JVM [14] в своем названии указывает на основной принцип работы. JIT-компилятор компилирует только участки кода, которые признает «горячими», остальной код интерпретирует. За время работы программы статистика накапливается, и компилятор может скомпилировать код, который до этого только интерпретировался. Также поддерживается «деоптимизация»: при компиляции кода JIT использует жадную стратегию, работая спекулятивно и делая предположения, которые могут не соответствовать действительности, поэтому, когда эти предположения не сбываются, приходится отбрасывать скомпилированный код и переходить (fallback) на интерпретацию. Рост производительности в виде отношения AOT-компиляции к JIT-компиляции для различных программ показан в [15].

В исследовании [16] приводится сравнение производительности различных высокоуровневых языков в реальной низкоуровневой задаче. На языках, включая C, C#, Java, разработан драйвер ixgbe для 10-гигабитных сетевых карт Intel и произведены измерения частоты передачи пакетов (packet rate) в зависимости от языка и размера «пачки» отправляемых пакетов (в целях оптимизации есть смысл передавать данные в сетевую карту по PCI не по одному пакету, а группами, называемыми burst или batch). Самыми производительными являются языки C и Rust. Но любопытно, что C# на достаточно большом размере batch-а приближается к этим языкам, а Java стабильно проигрывает в 2 раза. В то же время, Python, в котором отсутствует JIT-компиляция, хуже многократно. Таким образом, в случае C# даже применение сборщика мусора не сильно ограничивает эффективность программы, а динамически скомпилированный код показывает хорошие результаты даже в сравнении со статически скомпилированными C и Rust.

## 1.7 Система динамической инструментации кода DynamoRIO

DynamoRIO [17] представляет собой систему, позволяющую выполнять анализ и модификацию машинного кода программы для x86, x86-64 и ARM (включая Aarch64) прямо во время работы программы (т.е. динамически). Фреймворк может применяться для оптимизации, профилирования, трансляции, фаззинга и понимания работы произвольного машинного кода.

DynamoRIO отслеживает работу каждой инструкции кода, а в целях повышения производительности (чтобы не сводить выполнение программы к интерпретации) умеет кэшировать уже обработанные инструкции в отдельной памяти и прямо исполнять их в дальнейшем. Для корректной работы адресов внутри кэшей (ведь оригинальное местоположение кода изменяется, когда он перемещается в кэш) DynamoRIO транслирует исходные адреса в адреса из кэша с помощью хэш-таблицы. В целях повышения производительности «горячие» базовые блоки объединяются вместе в «след» (trace) и помещаются в памяти последовательно.

Функции обратного вызова в код, анализирующий инструкции, исполняются в конце базовых блоков, причем DynamoRIO самостоятельно производит декодирование кода. Возможно устанавливать обработчики для таких ситуаций, как создание новых потоков, подключение динамических библиотек, возникновение системных вызовов и сигналов. Внутри обработчиков можно без ограничений редактировать ассемблерный код или вставлять новый, например, для выполнения мониторинга или вызова других функций из инструментирующей библиотеки. Состояние должно сохраняться с помощью специальных функций и типов фреймворка, чтобы не нарушать консистентность состояния инструментируемой программы.

К недостаткам DynamoRIO можно отнести то, что он не поддерживает RISC-V, имеет значительные накладные расходы (в том числе из-за того, что DynamoRIO нацелено на более широкие возможности, чем только оптимизация кода). В фреймворке отсутствует поддержка профилей и статистик процессора для определения «горячего» кода.

## 1.8 Динамические оптимизации машинного кода

Проанализировав различные подходы в оптимизации кода, напрашивается рассмотрение и некоторого комбинированного метода, который бы заключался в применении динамической адаптивной компиляции (как CLR или JVM) для произвольного машинного кода (как Bolt, PGO или частично LTO). В разработке такого решения и заключается данная дипломная работа. В ней предлагается реализовать платформу, выполняющую загрузку машинного кода, его анализ (в частности, построение графа управления), применение некоторых доступных для низкоуровневого кода оптимизаций и исполнение преобразованного машинного кода.

Требования, которые следует учесть при проектировании и разработки системы, следующие:

1. Не нарушать корректности кода. Вносимые оптимизации не должны нарушать логики работы программы, которую предполагает программист.
2. Инфраструктурный код должен минимально нагружать систему. Смысл имеют только те динамические оптимизации, сам процесс применения, которых не потребует больше времени, чем оптимизация позволит сэкономить.
3. Прозрачность. Использование разработанной системы должно требовать минимального участия от пользователя (стороннего разработчика). В идеале оптимизации должны применять автоматически.
4. Должна быть предусмотрена подсистема для сбора статистик использования кода, чтобы автоматически выбирать участки кода, которые есть смысл оптимизировать.
5. Оптимизации должны преимущественно подвергаться циклы, как наиболее затратные по времени процессора единицы программы. Такого мнения придерживаются разработчики HotSpot JVM, оптимизируя код функциями и циклами.

Подобная система была разработана в [18] на базе фреймворка DynamoRIO для архитектуры x86 и поддерживает оптимизации удаления избыточной загрузки данных и кэширования непрямых вызовов. В работе не реализовано профилирование и другие автоматизации. В [19] и [20] реализуется динамическая оптимизация для произвольных последовательностей базовых блоков, но они не предоставляют возможностей для их модификации после. Некоторые оптимизации, которые можно назвать динамическими, умеют выполнять суперскалярные процессоры [21] (например, предсказание переходов для увеличения производительности непрямых переходов и автоматическое перемещение инструкций). Пионером в деле программных оптимизаций времени исполнения был фреймворк Dynamo [22] для архитектуры PA-RISC, который выполнял интерпретацию кода вкупе с определением и динамической компиляцией «горячего» кода. В [23] предлагается описание работы фреймворка Mojo для динамической оптимизации бинарного кода от Microsoft Research, показывающего хорошие результаты для маленьких по размеру программ и специализированных бенчмарков (SPEC), но приводящего к деградации производительности на больших проектах.

# 2 Анализ и выбор JIT-оптимизаций МК

В этой главе производится краткий обзор архитектуры RISC-V и компиляторных оптимизаций, которые подходят для динамического применения.

## 2.1 Сведения об архитектуре RISC-V

Архитектура RISC-V появилась в 2010 году в академических целях в Калифорнийском университете Беркли. Главной особенностью архитектуры стала ее полная открытость – использование спецификаций архитектуры свободно и бесплатна для любых целей. Разрешена разработка, коммерциализация процессоров и конфигураций ПЛИС без всяких денежных отчислений.

Набор инструкций следует модульному подходу – в спецификации определена возможность добавления в набор проприетарных расширений [24]. Кодирование устроено так, чтобы упростить декодирование инструкций в процессоре. Размер инструкций имеет произвольную длину, кратную 16 (впрочем, существует простая методика определения длины инструкции по первому байту). В базовом наборе инструкции имеют длину 32 байта. Построение процессоров для архитектуры очень легко (в сравнении с x86), RISC-V предполагает упрощенную микроархитектуру. На архитектуру портировано ядро Linux, компилятор GCC, бэкэнд LLVM и многие другие программы.

Набор инструкций состоит из базового (Base) и расширений, обозначаемых латинскими буквами или сочетаниями букв. Процессоры RISC-V обязаны поддерживать как минимум базовый набор. Существуют спецификации базового набора для 32-, 64-, 128-битных систем. Отдельно есть варианты базовых наборов для встраиваемых систем. К расширениям, например, относятся операции умножения и деления, атомики, операции с плавающей точной, поддержка сжатых (16-битных) инструкций и векторные инструкции.

Базовый набор состоит из 32 регистров x0-x32, причем регистр x0 всегда равен 0, а запись в него не имеет эффектов. Регистры также имеют «человекопонятные» символьные имена (оно же имя в ABI). Большая часть инструкций оперирует регистрами, а чтение и запись памяти выполняет специальными load- и store- инструкциями. Порядок байтов от младшего к старшему (little-endian). Архитектура не содержит регистра флагов, а все инструкции условного перехода принимают в качестве аргумента адрес перехода относительно инструкции перехода и непосредственно имена регистров, которые сравниваются.

Вызов процедур выполняется с помощью инструкций jal и jalr, которые сохраняют адрес возврата в регистре ra (x1).

Стандартизировано также и соглашение о вызове [25] – порядок вызова процедур, передачи в них параметров, получения результата, сохранения контекста и т.д. В таблице 2.1 перечислены регистры x0-x32 вместе с их ABI-именами, описанием и стороной, сохраняющей регистр при вызове процедуры.

Таблица 2.1 – Соглашение о вызове RISC-V

|  |  |  |  |
| --- | --- | --- | --- |
| Имя регистра в  RISC-V | Имя регистра в ABI | Описание | Сторона, сохраняющая регистр при вызове |
| x0 | zero | Всегда равен нулю, запись не имеет эффекта | – |
| x1 | ra | Адрес возврата из текущей процедуры | Вызывающий |
| x2 | sp | Указатель стека | Вызываемый |
| x3 | gp | Глобальный указатель | – |
| x4 | tp | Указатель потока | – |
| x5-x7 | t0-t2 | Регистры временных переменных | Вызывающий |
| x8 | s0/fp | Сохраняемый регистр (или указатель на фрейм) | Вызываемый |
| x9 | s1 | Сохраняемый регистр | Вызываемый |
| x10-x11 | a0-a1 | Аргументы функции (или возвращаемые значения функции) | Вызывающий |
| x12-x17 | a2-a7 | Аргументы функции | Вызывающий |
| x18-x27 | s2-s11 | Сохраняемые регистры (или указатель на фрейм) | Вызываемый |
| x28-x31 | t3-t6 | Регистры временных переменных | Вызывающий |

Получается, что часть регистров должна сохранять вызывающая функция (если ей необходимо), а часть регистров – вызываемая функция (если ей необходимо изменить эти регистры). Понимание соглашения о вызове является важным при написании кода на ассемблере, а также играет большую роль в осознании устройства binboost.

Все тесты кода RISV-V, представленные в работе, выполнялись на реальном железе – одноплатном компьютере StarFive VisionFive 2 с установленной операционной системой Gentoo Linux, в качестве компилятора используется clang-17. Характеристики платформы представлены в таблице 2.2.

Таблица 2.2 – Характеристики платформы StarFive VisionFive 2

|  |  |
| --- | --- |
| Название | Значение |
| Архитектура процессора | RISC-V 64 (RV64GC) |
| Частота процессора | 1,5 ГГц |
| Число ядер | 4 |
| Оперативная память | LPDDR4 8 ГБ |
| Постоянная память | SSD M.2 250 ГБ |
| Интерфейсы | USB Type-C (1 шт.), разъем 8P8C для Gigabit Ethernet (2 шт.), USB 2.0 (2 шт.), USB 3.0 (2 шт.), HDMI 2.0 (1 шт) и другие |

## 2.2 Выравнивание кода

Выравниванием кода называют встраивание в машинный код и данные программы дополнительных байтов неиспользуемых данных с целью сделать так, чтобы указатель на эти данные был кратным определенному числу (обычно степени 2). Это необходимо для ускорения чтения и записи данных в памяти компьютера, а для некоторых компьютерных архитектур вообще является обязательным требованием корректности. Код выравнивается с помощью вставки NOP-инструкций (в RISC-V их роль играет инструкция addi x0, x0, 0), т. е. инструкций, которые процессор игнорирует и не исполняет.

Компилятор самостоятельно выбирает выравнивание для всех типов, используемых в программе (для примитивных типов выравнивание равно размеру этого типа) на основе стандарта языка или общих соображений по оптимизации кода. В некоторых случаях программисту важно самому определять выравнивание для структур данных. Например, если это требует протокол (так, в заголовках сетевых пакетов не должно быть «пустых» пространств, которые обязательно появятся при выравнивании) или из-за необходимости повышения производительности.

Чаще всего вручную код и данные выравнивают по границам кэш-линии – единицы памяти, обслуживаемой кэшом процессора (64 байта для RISC-V). Расположив данные эффективно по кэш-линиям (т.е. минимизировав количество используемых кэш-линий), можно снизить время по загрузки данных из основной памяти в кэш. В [26] производятся различные тесты, демонстрирующие резкий рост производительности при корректном выравнивании.

Но, как было замечено, выравнивать нужно не только данные программы, но и ее исходный код. Ведь он тоже загружается из основной памяти и тоже использует собственный кэш. Как показывает исследование Microsoft [27], выравнивание кода особенно актуально для методов и циклов. Каждый раз, когда программа исполняет метод или цикл, она должна загрузить код из основной памяти. Если этот код не будет выравнен по кэш-линии, то процессору придется сделать больше обращений к памяти, снижая тем самым скорость работы программы, что может сказаться и в целом на работе программы, если неэффективно выравненный код будет «горячим».

Современные версии компиляторов автоматически выравнивают код методов на 64 байта, а код циклов на 16. Программист через флаги компилятора может изменить это, и даже указать компилятору выравнивать все циклы. Но этот подход имеет ограничения: размер выравнивания всегда одинаковый для всех циклов, а также влияет на всю кодовую базу, сильно увеличивая размер всего бинарного файла.

Это простая оптимизация может выполняться динамически. Необходимо определить два параметра алгоритма: какие циклы выравнивать и на сколько выравнивать.

Выравнивание циклов актуально только для небольших внутренних циклов (меньше 96 байтов; для циклов большего размера эффект от выравниваний практически не заметен). По умолчанию циклы выравниваются на границу 32 байтов (возможен автоматический переход на выравнивание по 16 байтов). Если размер выравнивания слишком большой, то предпринимается попытка выравнять его на границу 16 байтов (в таком случае, тоже циклы со слишком большим выравниванием игнорируются). Также игнорируются циклы, которые попадают на кэш-линию (т.е. начало и конец цикла лежит в пределах двух адресов, кратных 32). Не выравниваются и циклы, содержащие вызовы процедур – накладные расходы на вызов и исполнение процедуры сведут на нет выгоды от выравнивания.

Вставлять слишком большое количество NOP-инструкций неразумно потому, что они небесплатны: процессору все равно придется их считать из памяти и дешифровать.

Для циклов, которые содержатся в 1, 2, 3 32-байтных блоках максимальное выравнивание равно соответственно 15, 7, 3, 1 байтам (при выравнивании на 32 байта) и 7, 3, 1 (при выравнивании на 16 байтов).

Вставку NOP-инструкций стоит производить не сразу же перед циклом, в определенных «холодного» кода над ним, чтобы гарантировать, что вставленные инструкции выравнивания не исполняются процессором или исполняются редко.

Динамическое выравнивание циклов реализовано в дипломном проекте, а результаты представлены в разделе 3.10.

## 2.3 Векторизация кода

Векторные инструкции оказываются полезными, если в коде приходится выполнять множество простых вычислений параллельно. Такие инструкции также называют SIMD-инструкциями (single instruction, multiple data; одиночный поток команд, множественный поток данных), они обеспечивают параллелизм на уровне данных. Вычисления выполняются над векторами данных. В RISC-V существует векторное расширение (V) [28], отличающееся от привычных векторных инструкций популярных архитектур. В RISC-V векторы данных, к которым применяются инструкции, могут иметь произвольный размер (в определенных пределах), а сами инструкции полиморфны по отношению к размеру векторов – одни и те же инструкции работают с разными по размеру векторами.

Пример активного использования векторизации можно найти в упоминавшемся фреймворке DPDK. Практически все драйвера сетевых устройств в этом фреймворке поддерживают векторизованные функции, выполняющие отправку и прием пакетов и написанные вручную. Если процессор поддерживает использованные векторные инструкции, то при инициализации драйвера автоматически выбирается векторизованный вариант.

Для векторизации кода в компиляторах чаще всего используется алгоритм SLP-стратегия [29]. Компилятор выбирает набор одинаковых скалярных инструкций, которые могут быть выполнены параллельно и объединяет их в одну векторную (см. рисунок 2.1). Сама задача SLP является NP-полной, поэтому требует применять эвристики в реализациях.

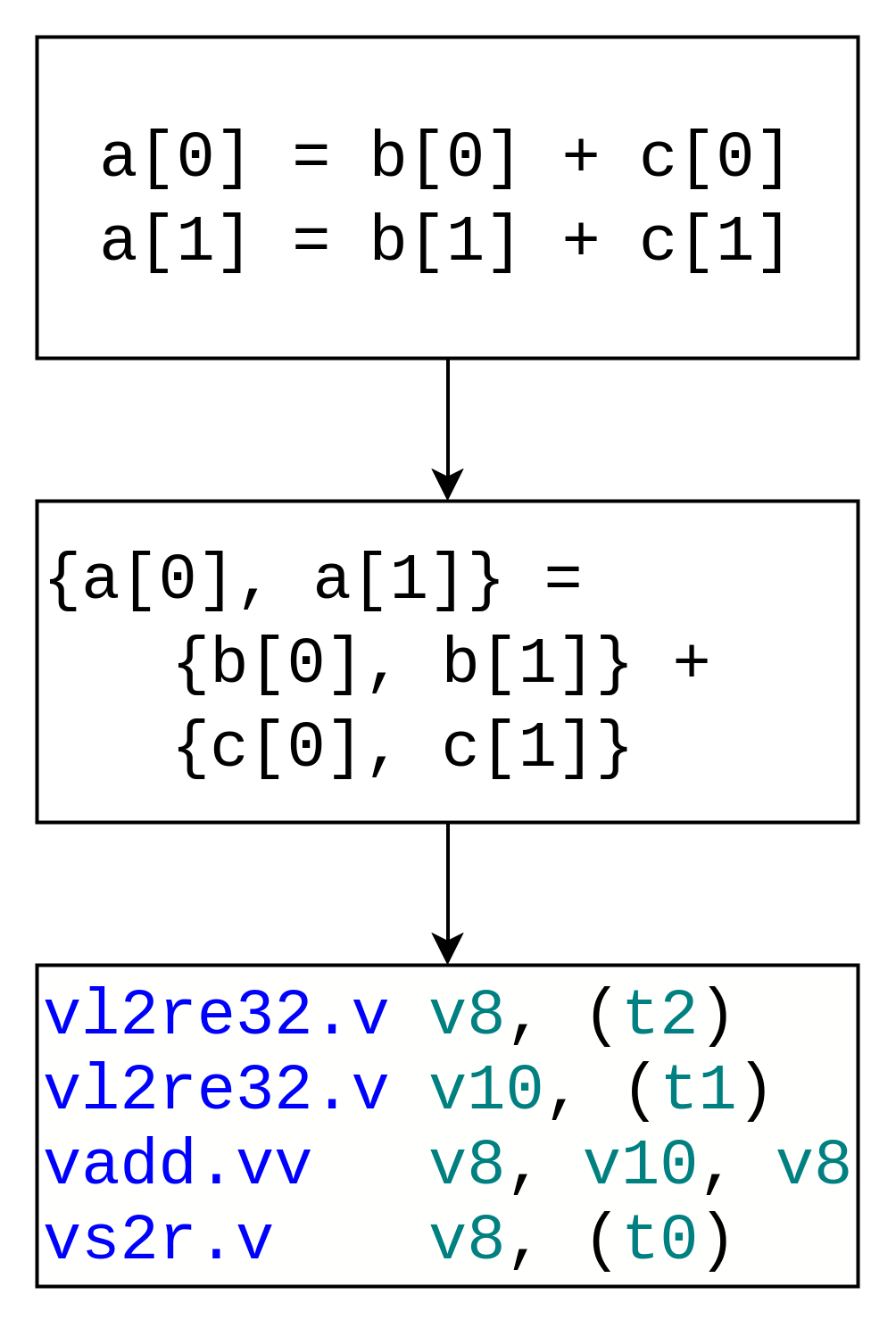


Рисунок 2.1 – Пример конвертации последовательности скалярных инструкций в векторные

## 2.4 Встраивание функций

Встраивание функций (inlining) считается [7] одной из ключевых компиляторных оптимизаций, предоставляющий реальный рост производительности практически для любых кодовых баз, а кроме того, расширяющий возможности компилятора в применении других оптимизаций в районе кода, куда функция встраивается. Встраивание функций заключается в том, что вместо генерации инструкции вызова функции, а также выполнения всех предписанных соглашением о вызовах действиях (сохранении регистров, передачи аргументов в функцию, создание пролога и эпилога функций и т.д.), компилятор в место вызова функции целиком скопирует тело функции, тем самым сокращая количество действий по вызову функции. Встроить можно не любую функцию по соображениям как физической реализуемости (например, если на функцию есть указатели, то в коде программы должна целиком присутствовать эта функция), так и эффективности работы программы (если функция имеет большой размер или вызывается в большом количестве мест, то ее встраивание приведет к чрезмерному увеличению размера бинарного файла).

Отдельным видом встраивания является девиртуализация функций. Объектно-ориентированный подход способствует к росту случаев, когда функция вызывается не статически (т.е. по конкретному, известному заранее адресу), а динамически – по указателю. Такая ситуация, во-первых, приводит к сложностям предсказания переходов процессором, и, во-вторых, убирает возможность встраивания функций, так как значение указателя может однозначно определяться лишь во время работы программы.

В этом случае динамическая компиляция может резко поспособствовать повышению производительности. Во время выполнения программы оказывается возможным определять адрес вызова функции, а также частоты вызовов. Если одна и таже функция, вызывается, например, в цикле (т.е. адрес вызова неизменен на протяжении работы цикла), то виртуальный вызов можно заменить на прямой вызов или вообще выполнить встраивание функции.

## 2.5 Оптимизация расположения кода

Современные процессоры общего назначения в своей работе применяют конвейер – каждая стадия исполнения инструкции может выполняться параллельно. Таким образом, процессор за один такт в действительности выполняет несколько (до десятков) разных стадий конвейера для разных инструкций параллельно. Выделяют обычно следующие стадии:

* загрузка инструкции из оперативной памяти;
* декодирование инструкции (определение ее опкода и операндов);
* загрузка из памяти или регистров данных, выступающих аргументами команды (операндов);
* выполнение действий, предписанных опкодом инструкции;
* запись результатов выполнения в память или регистры.

В общем случае каждая стадия не зависит от предыдущей, поэтому они могут исполняться вместе, но для разных последовательно расположенных инструкций, как показано на рисунке 2.2. Цветом выделено состояние, когда конвейер утилизируется полностью.

В то же время, процессоры поддерживают так называемое внеочередное исполнение команд (out-of-order execution, OoO), когда они умеют автоматически определять, что некоторые инструкции, не расположенные подряд, не имеют зависимости между собой и могут быть переставлены без влияния на логику работы программы. С помощью этого подхода процессор может более эффективно исполнять инструкции, не простаивая, если какая-то инструкция в конвейере займет больше времени, чем остальные.

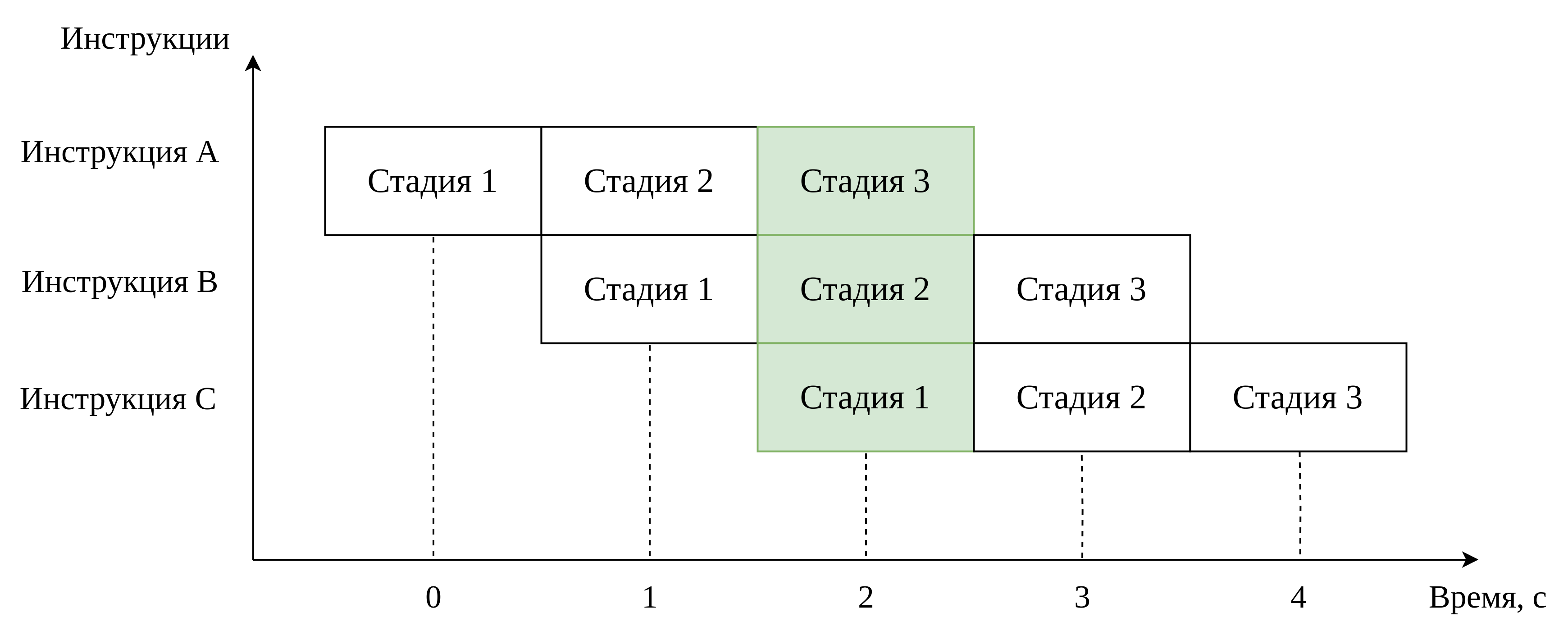


Рисунок 2.2 – Пример работы простого трёхуровневого конвейера

Обе эти особенности устройства процессоров приводят к тому, что расположение инструкций в памяти важно для производительности, ведь процессор стал «заглядывать» в будущее, за инструкцию, исполняющуюся в данный момент времени. Если для линейного кода сложностей не возникает, то при наличии ветвлений процессор может положить в конвейер и начать исполнять те инструкции, которые окажутся в действительности невыполненными, если в ветвлении будет выбран другой участок кода. Таким образом, код, который будет выполнен вероятнее, необходимо располагать вверху, а тот, который выполняется реже – внизу.

Оптимизация, которой пользуется компилятор для выбора более удачного положения машинного кода при его генерации, называется оптимизация расположения кода или code layout optimization. Например, большой упор делается на нее в оптимизации на основе профилирования и в Bolt, о которых рассказано выше. Так, в Bolt удается достичь более 20% прироста производительности.

Но и PGO компиляторов, и Bolt используют профили работы программы, собранные заранее разработчиками. Они могут стать неактуальными в условиях реального применения программы с учетом конкретных нужд пользователя. По этой причине, выигрышным подходом было бы динамически, во время работы программы определять «горячий» код и наиболее вероятно исполняющиеся ветки в условных конструкциях, чтобы тут же перемещать участки кода на более выгодные позиции. Примерно по такому принципу работает виртуальная машина Java – HotSpot. В ней для «холодного» кода применяется обычное интерпретирование, а для «горячих» мест программы прямо во время ее исполнения вызывается компилятор, который преобразует байт-код в машинный.

# 3 Инфраструктура JIT-компилятора МК

В данной главе выполняется проектирование библиотеки, предоставляющей средства для создания динамических оптимизаций машинного кода для RISC-V, а также непосредственного выполнения этих оптимизаций при работе целевой программы.

## 3.1 Алгоритм работы оптимизатора

Итак, разрабатываемое решение должно определять места в коде, которые следует во время выполнения программы рассматривать для оптимизации. Объектом оптимизации будет машинный код на языке ассемблера RISC-V.

В начале рассмотрен инструмент, в котором пользователь самостоятельно расставит в своем коде вызовы функций bb\_jit\_begin() и bb\_jit\_end(), тем самым отмечая диапазон кода (обычно, некоторый цикл или ряд вложенных циклов), который он ожидает будет «горячим» и который требуется оптимизировать. Пример для алгоритма сортировки пузырьком приведен ниже (kBound определена как константа равная 750).

// Выполняет сортировку элементов массива `x` алгоритмом

// пузырька.

static void Inner(int\* x) {

auto limit1 = kBound - 1;

BB\_JIT\_BEGIN(); // Начинаем динамическую оптимизацию.

for (int i = 1; i <= limit1; ++i) {

// В зависимости от значения элемента x[i] сдвигаем его

// в самый конец массива, чтобы часть массива [0, i]

// была отсортированой.

for (int j = i; j <= kBound; ++j) {

if (x[i] > x[j]) {

int temp = x[j];

x[j] = x[i];

x[i] = temp;

}

}

}

BB\_JIT\_END(); // Прерываем динамическую оптимизацию.

}

Видно, что вместо прямых вызовов использованы макросы – это сделано для возможности отключения на этапе компиляции JIT-компиляции.

Сокращенный код на ассемблере, который получается в результате компиляции (используется clang 17 с флагом -O3), находится ниже.

Inner(int\*):

addi sp,sp,-16 // (1)

sd s0,0(sp)

sd ra,8(sp) // (2)

mv s0,a0 // (3)

call bb\_jit\_begin() // (4)

li a5,4096

addi a5,a5,-1092

addi a1,s0,4

li a0,1

add a2,s0,a5

li a6,750

.L2: // (5)

lw a3,0(a1)

mv a5,a1

.L4: // (6)

lw a4,0(a5)

ble a3,a4,.L3

sw a3,0(a5)

sw a4,0(a1)

mv a3,a4

.L3: // (7)

addi a5,a5,4

bne a5,a2,.L4

addi a0,a0,1

addi a1,a1,4

bne a0,a6,.L2

ld s0,0(sp) // (8)

ld ra,8(sp)

addi sp,sp,16

tail bb\_jit\_end // (9)

В диапазоне (1)-(2) представлен пролог функции (в основном, работа с указателем на стек в регистре sp). В (4) компилятор сгенерировал call-инструкцию на функцию bb\_jit\_begin(). Обратим внимание, что вызов этой функции ничем не отличается от вызова любой функции, поэтому компилятор корректно обработает все случаи соглашения о вызовах, и консистентность данных (состояние стека и регистров) не нарушится. Так, компилятор использует в (3) сохраняемые регистры, чтобы их значение не изменилось после вызова bb\_jit\_begin().

Два вложенных цикла и конструкция if представлены 3-мя метками, причем if-у соответствует (6) (а загрузка x[i] вынесена в тело первого, внешнего цикла), а оба цикла обрабатываются по метке (7). В зависимости от условий циклов, исполняется либо if (6), либо происходит перезапуск внутреннего цикла (5). Выход из циклов выполняет в (8), где начинается эпилог функции.

Дальше компилятор применил оптимизацию хвостовой рекурсии: сгенерировал инструкции эпилога функции и вместо инструкции ret (возврата из функции) – безусловный переход на функцию bb\_jit\_end(). Тут не будет проблем: ret уже в функции bb\_jit\_end() выполнит возврат в функцию, вызвавшую Inner(). Но это приводит к важному замечанию, заключающемуся в том, что вызов bb\_jit\_end() пропущен не может быть и обязан будет находиться в оптимизированном машинном коде.

Таким образом, для выполнения различных оптимизаций необходимо:

* выполнить декодирование необходимых инструкций (в частности, переходов),
* построить дерево базовых блоков,
* произвести подходящие оптимизации,
* скопировать оригинальный код в новую область памяти с учетом внесенных модификаций,
* выполнить переход из bb\_jit\_begin() новый код,
* после окончания выполнения нового кода вернуться ровно на инструкцию, производящую вызов bb\_jit\_end().

## 3.2 Исполнение динамического кода

Вычислительные системы, для которых разрабатывается продукт, имеют архитектуру фон Неймана. Этот архитектурный принцип предписывает хранить команды и данные машины в одной и той же памяти компьютера. Таким образом, программа, исполняющаяся на машине, может иметь прямой доступ и к данным, и к командам [22]. В частности, это позволяет перезаписывать свой собственный код, так как доступ к нему ничем не отличается от доступа к обычной памяти данных. Но, кроме перезаписывания, процессор получает возможность интерпретировать произвольную область памяти как команды. Кстати, с этим связано большое число уязвимостей, когда злоумышленник, пользуясь различными стратегиями, загружает зловредную программу в память и указывает процессору ее выполнить.

В контексте дипломной работы архитектура фон Неймана позволяет сгенерировать во время работы программы произвольный машинный код, записать его в память и, переместив счетчик команд процессора на начало нового кода, начать выполнять код.

В процессорах RISC-V (как и в подавляющем большинстве других процессоров, имеющих модуль управления памятью MMU) страницам виртуальной памяти с помощью специальных флагов устанавливаются права. Так, только явно отмеченные флагом X (executable) страницы памяти могут содержать инструкции процессора (в противном случае возникнет исключение процессора). В Linux есть средства, позволяющие устанавливать флаги страницам.

Выделение памяти для целей выполнения в них кода, должно производиться в единицах страниц (иначе не получится установить нужные флаги). Linux предоставляет системный вызов mmap, присутствующий в виде одноименной функции в библиотеки API, заголовочный файл sys/mman.h. В функцию передается желаемый виртуальный адрес начала выделяемой памяти (NULL, если неважно), размер памяти (кратный размеру страницы, иначе будет округление), флаги защиты памяти (в нашем случае, используем PROT\_READ | PROT\_WRITE | PROT\_EXEC) и еще некоторые параметры, которые в данном случае не представляют интереса.

Функция mmap() выделит в оперативной памяти (обычно, в верхней ее части, когда код сам располагается в нижней) область памяти, размером кратным размеру страницы, с указанными флагами. Таким образом, скопировав оптимизированный код в эту память, его будет возможно исполнять процессором.

Также, необходимо обратить внимание, что весь код будет исполняться в одном пространстве виртуальной памяти, в одном процессе. Если в сгенерированном коде будут оставаться абсолютные вызовы по прежним адресам, то их релоцировать (т.е. редактировать, изменять) будет не нужно – они отработают корректно.

Для обеспечения автоматизации применения оптимизаций (т.е. чтобы убрать необходимость расставлять bb\_jit\_begin() и bb\_jit\_end()) во время загрузки кода необходимо статически выявлять все возможные точки оптимизации и вставлять в них вызовы функций bb\_jit\_begin(), bb\_jit\_end(). Внутри этих функций можно реализовать контроль и анализ аппаратных счетчиков производительности, чтобы оптимизировать только в действительности «горячий» код.

## 3.3 Трамплин и выход из оптимизированного кода

Проект написан на языке C++20, но в двух местах используется рукописный код на ассемблере.

При входе в библиотеку внутри функции bb\_jit\_begin() необходимо в самом конце перейти на новый сгенерированный код. Переход будет выполнен на первую же инструкцию, идущую после вызова bb\_jit\_begin(). Скорее всего, эта инструкция будет являться частью кода по восстановлению стека и регистров после вызова. Таким образом, оригинальный код самомстоятельно сохранит (до вызова bb\_jit\_begin()), а потом восстановит контекст (после вызова). Но в процессе сохранения и восстановления контекста участвуют обе стороны вызова: вызывающая функция (Inner из кода подраздела 3.1) и вызываемая фукнция (bb\_jit\_begin()), часть операций выполняется в одном месте, а часть в другом. Поэтому прежде, чем выполнять вызов нового кода необходимо произвести свою работу по восстановлению контекста в bb\_jit\_begin(), т.е. переход следует выполнять только после эпилога функции. А сам переход (j по абсолютному адресу) будет располагаться вместо инструкции ret. Функция bb\_jit\_begin() должна выглядеть полностью как обычная, но последней своей инструкцией будет иметь инструкцию безусловного перехода.

Достичь такого стандартными средствами языка C++ нельзя (нет подходящей возможности влиять на генерацию кода компилятора в прологе и эпилоге). Поэтому была написана самостоятельно функция-обертка вокруг bb\_jit\_begin\_impl() – реальной функции, являющейся точкой входа в оптимизатор (см. код ниже).В ней сохраняется на стеке исходное значение указателя на стек sp (пролог), затем в регистр a1 (временный, он сохраняется внешним кодом) записывается указатель на адрес возврата из bb\_jit\_begin(), производится вызов bb\_jit\_begin\_impl(), затем восстановление sp и безусловный переход по адресу, который возвращает bb\_jit\_begin\_impl(), из регистра a0.

.align 6

.globl bb\_jit\_begin

bb\_jit\_begin:

add sp, sp, -16 // Пролог

sd ra, 0(sp)

sd s0, 8(sp)

mv a0, ra // Передача ra в параметре

call bb\_jit\_begin\_impl

ld ra, 0(sp) // Эпилог

ld s0, 8(sp)

add sp, sp, 16

jr a0 // Переход на адрес,

// по которому расположен

// новый код.

Второе место, содержащее рукописный ассемблер, находится в самом завершении генерируемого кода. После выполнения оптимизированного кода следует вернуться на инструкцию в оригинальном коде, где завершается оптимизируемый диапазон кода. В RISC-V есть две возможности реализации безусловного перехода: по смещению относительно адреса инструкции перехода (jal) или по абсолютному адресу из регистра (jalr). Так как в binboost генерируемый код помещается в отдельной странице памяти, выделенной системным вызовом map(), то расстояние между оригинальным кодом и динамически сгенерированным может быть больше 2 МБ (под смещение выделено 11 битов + знаковый бит), что не позволит использовать относительную адресацию – смещение попросту не поместится в инструкцию. Поэтому единственным вариантом является использование абсолютной адресации.

Инструкция jalr принимает три параметра: регистр, в который сохраняется адрес возврата (в данном случае он не нужен, поэтому указывается x0), регистр, из которого берется адрес перехода и произвольное число-смещение, которое дополняется до 2-х и прибавляется к адресу перехода.

В RISC-V все инструкции имеют размер максимум 4 байта, поэтому задача поместить 64-битное число (которым является адрес) в регистр нетривиальна. Обычно в программах для этих целей применяются различные макроинструкции (%lo/%hi, li, la), которые обрабатывает ассемблер или линкер. Но в случае генерации динамического кода никакими сторонними утилитами пользоваться нельзя и приходится генерировать непосредственно машинный код без синтаксического сахара. Кроме того, задача усложняется тем, что целевой адрес не известен заранее.

Код с этим переход является составной частью генерируемого кода, поэтому нет необходимости его записывать статически, а можно отдельно сгенерировать в процессе оптимизации кода. По всем этим причинам, загрузка адреса в регистр производится с помощью инструкции lui (загрузка 12 битов в старшие разряды регистра), четырех инструкций ori (побитовое ИЛИ содержимого регистра и константы размером 12 битов) и четырех инструкций slli побитового сдвига. Число загружается в цикле по 11 битов за итерацию. Алгоритм показан на рисунке 3.1.

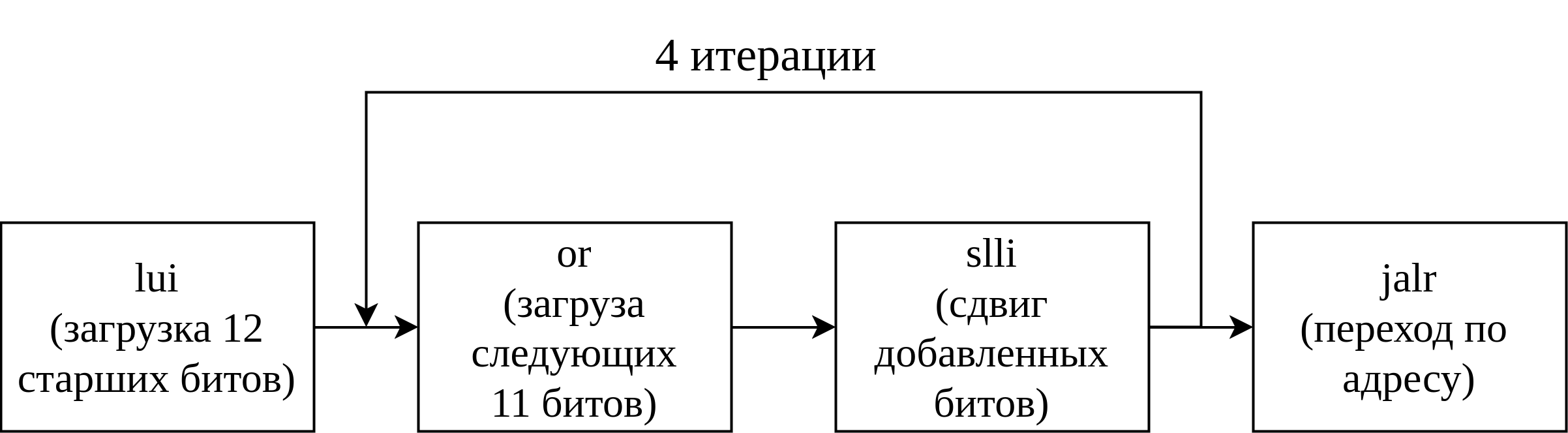


Рисунок 3.1 – Алгоритм загрузки 64-битного числа в регистр

После прыжка в оригинальный код выполнение программы продолжается без изменений. По этой причине необходимо гарантировать, что все регистры и стек будут находиться в том же состоянии, что и до оптимизаций: нельзя переименовывать регистры или изменять данные на стеке, а в случае, если такое все-таки произойдет, нужно обязательно вернуть все в изначальный вид.

## 3.4 Декодер и энкодер инструкций

Для того, чтобы определить, где заканчивается диапазон оптимизируемого кода, необходимо найти в оригинальном коде инструкцию вызова функции bb\_jit\_end(). Функция отмечена как noinline, поэтому компилятор всегда сгенерирует вызов функции и не выполнит встраивание функции. А так как binboost подключается статически, то, получив однажды адрес функции стандартными средствами С++ (оператором «&»), следует только обнаружить любую jump-инструкцию на адрес &bb\_jit\_end, причем рассматривать нужно как переходы по абсолютному, так и переходы по относительному адресам.

Определение jump-инструкций требуется и в компоненте, ответственном за построение базовых блоков, ведь такие инструкции являются маркерами границ блоков. Но в таком случае рассматривать нужно и все условные инструкции, так как в RISC-V отсутствует регистр флагов, и переход в условных конструкциях делается непосредственно инструкцией сравнения.

При выполнении оптимизацией часто окажется необходимым понимать, какие именно инструкции находятся в базовом блоке (например, для замены одних инструкций на другие или определения регистра, в котором хранится номер итерации цикла).

С другой стороны, при выполнении релокаций и вообще изменении машинного кода, потребуется формировать нужные машинные инструкции по заданным частям.

Таким образом, появляется очевидная необходимость в отдельном компоненте, который был бы ответственен за декодирование (декодер) и кодирование (энкодер) инструкций. Декодированием называется процесс чтения битовых данных, представляющих инструкцию в памяти ЭВМ, в определенные семантически цельные единицы (например, смещения адреса, номера используемых регистров, номер опкода инструкции). В свою очередь, кодирование означает обратный процесс, т.е. преобразование информации о команде процессору в бинарное представление инструкции.

В RISC-V инструкции одновременно легки для декодирования и кодирования (имеют строго определенный размер, четко разбиты на типы) и сложны (отдельные числовые данные порой записываются как отдельные биты чуть ли не в случайном порядке). Декодировать полный исходный текст оптимизируемого диапазона не имеет смысла: эта несложно по времени, но позволяет сэкономить на памяти (хранить закодированные инструкции много проще, чем полностью «распаршенные»). По этой причине декодер и энкодер выделены в отдельные классы, методы которых используются каждый раз по требованию. Кроме того, инструкции одного класса выделены в специальные группы со своей специфичной информацией. Например, декодер одинаково декодирует все инструкции, которые изменяют счетчик команд процессора, однообразно и возвращает структуру, которая хранит только адрес перехода.

Оба класса построены на идеи метапрограммирования C++, чтобы снизить нагрузку на процессор и память при работе и предоставить наиболее гибкое API. Для каждой инструкции (или группы инструкций, или типа инструкций) определен шаблонный класс Format, который содержит информацию о том, как нужно интерпретировать каждый бит памяти, чтобы извлечь нужные данные инструкции. Затем, с помощью шаблонного метода Decode на основе переданного Format данные читаются из указателя на память с инструкцией и ими заполняется временная структура, содержащая все, что нужно для парсера.

Декодирование рассматривается на примере инструкции c.j – безусловный переход из расширения сжатых инструкций – представление которой изображено на рисунке 3.2 [15].

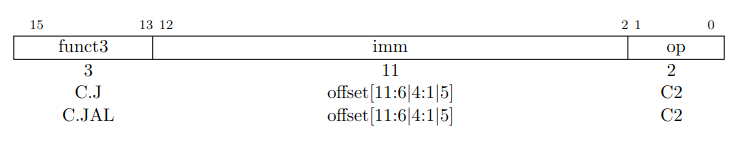


Рисунок 3.2 – Представление инструкции c.j

Структура для декодирования показана в коде ниже.

// Структура, предоставляющая методы для декодирования

// инструкции c.j.

struct CjFormat {

// Выполняет чтение опкода из `instr` (биты [1:0]).

static constexpr auto ReadOp(std::uint32\_t instr) {

return Format<B<1, 0>>::Decode(instr);

}

// Выполняет чтение funct3 из `instr` (биты [15:13]).

static constexpr auto ReadFunct3(std::uint32\_t instr) {

return Format<B<2, 0>, S<13>>::Decode(instr);

}

// Выполняет чтение imm из `instr` (биты [11:6|4:1|5]).

static constexpr auto ReadImm(std::uint32\_t instr) {

return SignExtend<

Format<B<11>, B<4>, B<9, 8>, B<10>,

B<6>, B<7>, B<3, 1>, B<5>, S<2>>>

::Decode(instr);

}

// Проверяет, что в `instr` в дейсвительности хранится

// инструкция c.j.

static constexpr bool IsValid(std::uint32\_t instr) {

return ReadOp(instr) == 0x1 && ReadFunct3(instr) == 0xa;

}

};

То есть, биты [1:0], [11:6|4:1|5], [15:13] с помощью Format разделены на три блока соответственно Op, Imm и Funct3. Декодер на вход получается CjFormat и в зависимости от задачи, вызывая функции ReadOp, ReadFunct3, ReadImm, IsValid, интерпретирует инструкцию.

Кодировщик (энкодер) построен аналогично. Но вместо Read-методов, для кодировщика Format содержит похожие Write-методы (WriteOp, WriteFunct3, WriteImm и т.д.).

## 3.5 Процесс построения графа потока управления

Оптимизирующим компиляторам, выполняющим различные преобразования над кодом, необходимо иметь удобное для алгоритмов представление исходного кода, работать напрямую с конструкциями языка или даже абстрактным синтаксическим деревом неудобно и неэффективно. Обычно программа представляется в виде графа, который называется граф потока управления или control flow graph (CFG) [30].

Представление в виде CFG получается следующим образом:

1. Промежуточный код (в данном случае машинный) разделяется на базовые блоки, каждый из которых содержит последовательность инструкций, удовлетворяющую следующим условиям:
   1. Поток управления может входить в блок только через первую инструкцию (никакая jump-инструкция программы не может указывать на инструкции в базовом блоке, кроме первой).
   2. Поток управления может покидать блок только через последнюю инструкцию (т.е. из каждого ББ может выходить не больше двух ребер).
2. Базовые блоки формируют узлы CFG, ребра графа показывают движения потока управления программы.

В случае binboost CFG строится напрямую из машинного кода. Каждый ББ представлен в виде структуры BasicBlock, содержащий 4 поля: указатель на начало в коде программы, размер в байтах и 2 указателя на BasicBlock (один для случая, когда последняя инструкция исполняется, а второй – когда нет). Первый указатель может быть NULL, если последняя инструкция безусловно переходит к следующему блоку.

На рисунке 3.3 показан CFG, полученный для оптимизируемого участка программы из кода подраздела 2.1. Переходы между базовыми блоками при выполнении условия показаны штриховой линией. Безусловные переходы или переходы при невыполненном условии – основной линией.

Построение графа несложно. Оно состоит из цикла, который выполняется ровно до тех пор, пока не встретится любая jump-инструкция на адрес &bb\_jit\_end. В цикле код программы, начиная с вызова bb\_jit\_begin(), интерпретируется с помощью декодера, определяя jump-инструкции. Если текущая инструкция не переход, то размер текущего ББ увеличивается на размер инструкции (2 или 4 байта) и декодируется следующая инструкция. В противном случае, текущий базовый блок добавляется в CFG, создается новый ББ, цикл продолжается уже для него. Важна аккуратная работа с указателями базовых блоков, чтобы граф в итоге получился корректным.

Все jump-инструкции и метки, на которые выполняется ими переход, помещаются в JumpTable – таблицу, хранящую тип инструкции или метрики, ее положение и указатель на память, куда она ссылается. После окончания этапа декодирования, в узлах графа проставлены только ребра, по которым поток управления идет безусловно. Выполняя перебор по JumpTable и на основе сохраненных в ней адресов, разрешаются оставшиеся ссылки между BasicBlock.

## 3.6 Релокация кода

Как говорилось ранее, переходы в коде могут быть абсолютными и относительными и особую сложность представляют именно последние, так как значение адреса перехода прямо зависит от расположения инструкции, выполняющей переход. Но в то же время, большинство оптимизаций кода, который binboost будет выполнять (включая даже выравнивание кода циклов), приводят к смещению инструкций с прежних позиций. При кодогенерации необходимо учитывать это обстоятельство.

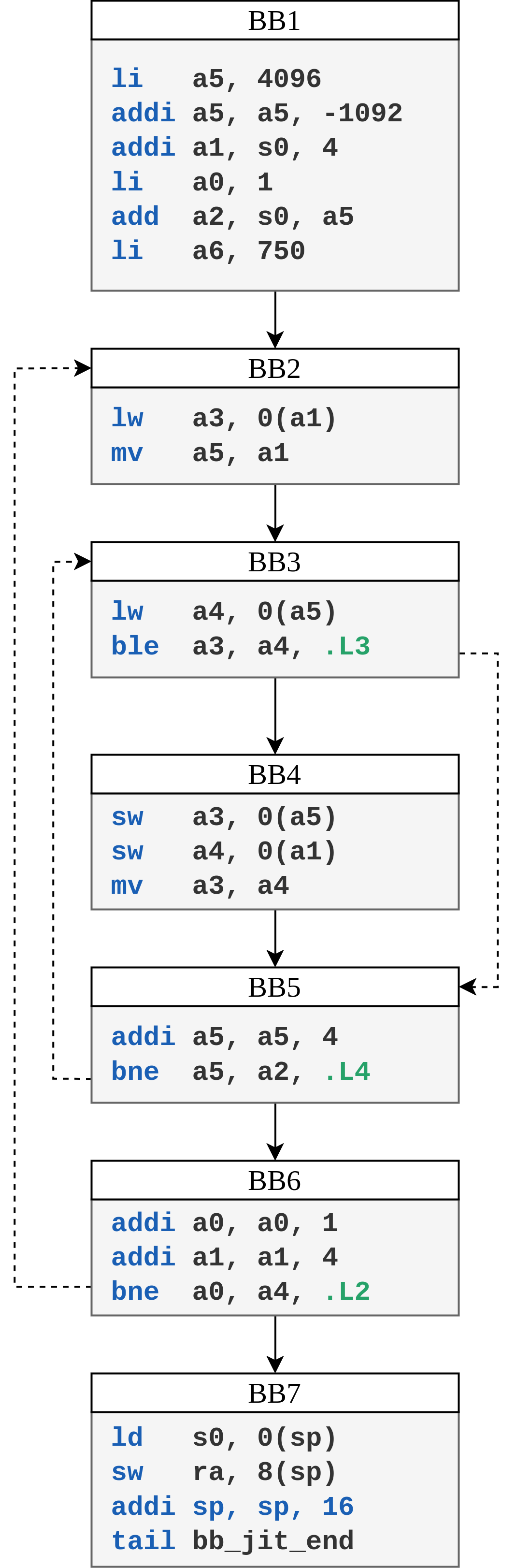


Рисунок 3.3 – Граф потока управления для программы для кода   
из разделов 2.1 и 2.2

Все базовые блоки хранятся в бинарном дереве и отсортированы по адресам их местоположений в коде, в тому же известно, что инструкции перехода (условного и безусловного) могут быть расположены только в конце базового блока. Для алгоритмов оптимизации размер базовых блоков всегда константен, новый код вставляется только как новые базовые блоки. Поэтому достаточно для каждой jump-инструкции посчитать размеры блоков перед ней (для переход наверх) или после нее (для переходов вниз). Зная этот размер, можно исправить адрес перехода и при генерации кода указывать этот новый адрес.

## 3.7 Описание архитектуры

Рассмотрев устройство и работу основных компонентов binboost, можно перейти к обзору архитектуры проекта целиком. Основные классы и зависимости между ними представлены на рисунке 3.4.

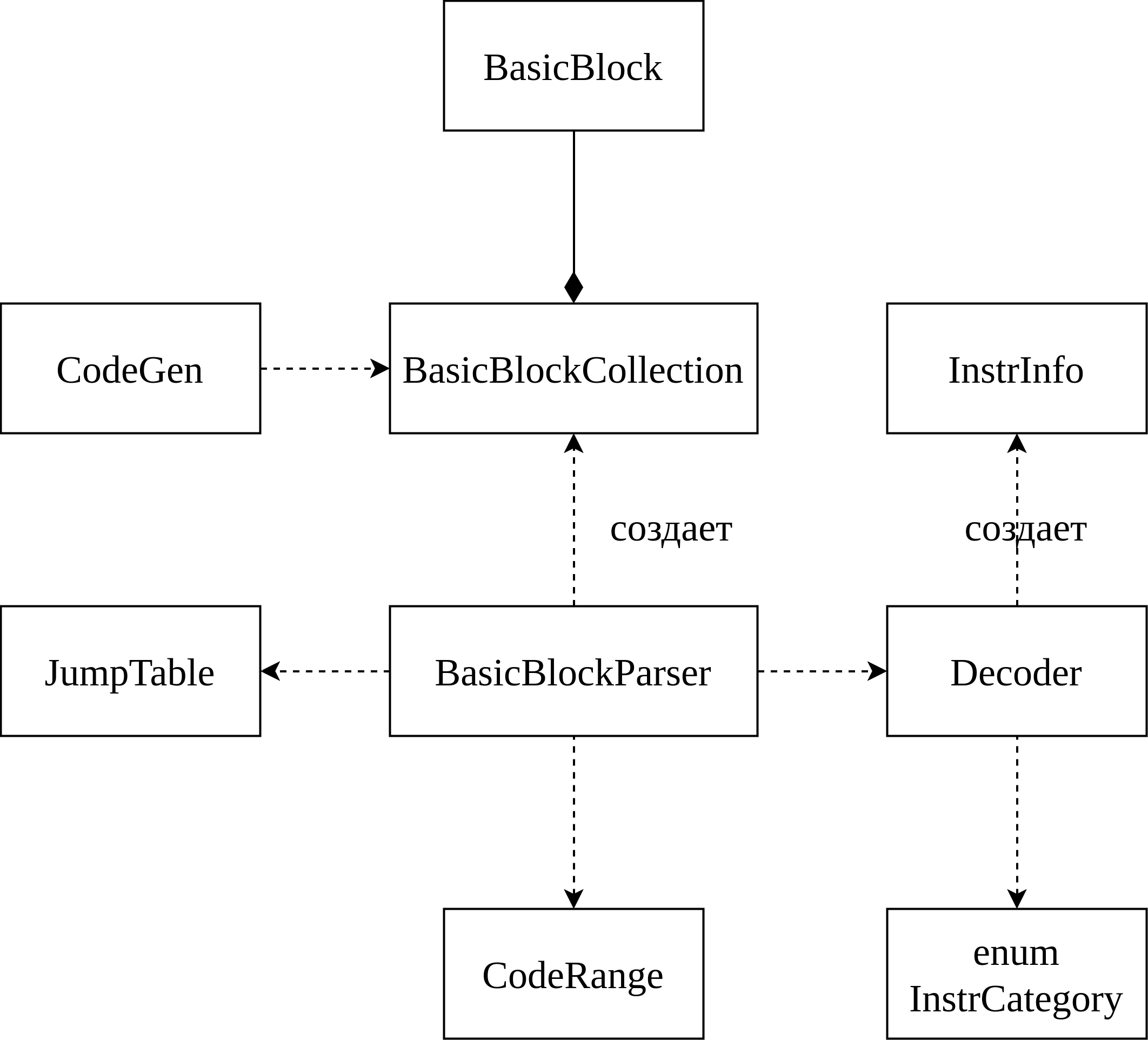


Рисунок 3.4 – Основные классы проекта

Сторонний оптимизированный код вызывает функцию bb\_jit\_begin(), являющуюся по существу оберткой вокруг bb\_jit\_begin\_imp(), которая в свою очередь запускает весь конвейер по решению задачи оптимизации. С помощью класса BasicBlockParser исходный машинный код раскладывается в граф объектов типа BasicBlock, начиная от следующей инструкции после вызова bb\_jit\_begin() и заканчивая включительно инструкцией вызова функции bb\_jit\_end(). Каждый BasicBlock ссылается на непрерывный диапазон машинных инструкций и заканчивается некоторой условной или безусловной jump-инструкцией. BasicBlock кроме того содержит два указателя на другие BasicBlock, один из которых может быть NULL, если BasicBlock не имеет условного перехода. Вся совокупность базовых блоков содержится внутри класса BasicBlockCollection, предоставляющего единообразный интерфейс по добавлению и поиску элементов, а также он предоставляет пару итераторов для обхода графа. Код, использующий BasicBlock, может либо добавлять новые базовые блоки в дерево, либо изменять содержимое старого (без изменения общего размера блока в байтах). Если модификация базового блока приводит к увеличению или уменьшению числа инструкций, то необходимо пересоздать базовый блок целиком.

Каждая оптимизация представляется отдельным классом, получающим на вход BasicBlockCollection. Таким образом, каждая оптимизация работает независимо, каждый раз заново обходя CFG, по тому же принципу, который применяется в LLVM.

В целях дальнейшей поддержки дополнительных архитектур весь исходный код проект разбит на две части: платформонезависимый и платформозависимый. Все файлы второй часть находятся в папке src/arch. Средствами CMake во время компиляции автоматически определяется платформа, после чего выбирается соответствующая платформозависимая часть, и она статически линкуется с платформонезависимой. Все поддерживаемые платформы должны иметь одинаковый интерфейс на уровне подключаемых заголовочных файлов. Заголовочные файлы имеют одинаковые названия и содержат классы, методы, алиасы типов с одинаковыми названиями. Таким образом, платформонезависимый код подключает все заголовочные файлы по одному и тому же пути, а система сборки с помощью параметров компилятора выбирает уже конкретные файлы платформы. Все это обеспечивает статический полиморфизм без необходимости использовать медленное наследование. Например, платформонезавимый код подключается файл arch/instr.hpp, а CMake, передавая компилятору путь до папки, в которой необходимо искать этот файл, может регулировать платформу, под которую собирается код (указанный заголовочный файл может находиться по пути src/arch/amd64/include/arch/instr.hpp или src/arch/riscv/include/arch/instr.hpp).

В таблице 3.1 перечислены все файлы проекта и дано их описание.

Таблица 3.1 – Структура проекта

|  |  |
| --- | --- |
| Файл | Описание |
| include/bb.hpp | Содержит интерфейс всего проекта – объявления функций bb\_jit\_begin() и bb\_jit\_end(). |
| src/binboost.(cpp,hpp) | Содержит определение «главной» функции проекта – bb\_jit\_begin\_impl(), а также bb\_jit\_end(). |
| src/basic\_block\_parser.(cpp,hpp) | Создает CFG на основе |
| src/code\_gen.(cpp,hpp) | Выполняет кодогенерацию, выделяет новую страницу памяти, копирует в нее оптимизированный код и выполняет. |
| src/arch/riscv/include/arch/instr.hpp | Типы и утилитарные функции для работы с отдельными инструкциями кода |
| src/arch/riscv/include/arch/decoder.hpp | Декодер инструкций |
| src/arch/riscv/include/arch/instr\_gen.hpp | Кодогенератор |
| src/arch/riscv/include/arch/instrinsics.hpp | Низкоуровневые функции над машинным кодом |
| src/arch/riscv/trampoline.S | Реализация функции bb\_jit\_begin() на ассемблере |

## 3.8 Автоматизирование динамических оптимизаций

Разработанная в ходе дипломной работе инфраструктура позволяет без особого труда реализовать и автоматическое применение оптимизаций. На данный момент ожидается, что пользователь продукта самостоятельно расставит в исходном коде вызовы bb\_jit\_begin()/bb\_jit\_end(). Автоматизация позволит выполнять действия по оптимизации вообще без участия разработчика.

В операционной системе с ядром Linux исполняемые файлы имеют сложную структуру, а не просто содержат машинный код. Формат бинарных исполняемых файлов называется ELF. Он содержит внутри большое количество разнообразной информации о том, что и как необходимо сделать, чтобы запустить исполняемый файл. Файл формата ELF состоит из заголовка, хранящего различную метаинформацию о фале (версию, целевую платформу, тип, информацию о динамических зависимостях), и данных (описание и содержимое сегментов или секций).

Запуск ELF файла выполняет так называемый динамический компоновщик (загрузчик). Он считывает с диска содержимое файла, интерпретирует информацию из него, выделяет оперативную память и загружает в нужные ее места данные секций файла, обрабатывает ссылки на динамические библиотеки. Затем, непосредственно создает новый процесс и исполняет в нем машинный код.

Таким образом, используя уже созданные средства для переданного пользователем произвольного исполняемого файла формата ELF необходимо построить CFG, определить места, потенциальные для оптимизации, вставить вызовы bb\_jit\_begin()/bb\_jit\_end() и исполнить этот файл. То есть полностью повторить работу компоновщика, но только добавить в него функционал по внедрению вызовов функций. Для разработки можно использовать свободные компоновщики, например, стандартный компоновщик Linux ld-linux.so, лишь адаптировав его для целей проекта, как было рассказано выше.

## 3.9 Python-скрипт для выполнения бенчмарков

В ходе разработки и исследований в рамках проекта выяснилось, что требуется также иметь специальную программу, автоматизирующую действия по выполнению бенчмарков различных программ на различном оборудовании и отображающую результаты в удобной для анализа человеком и машиной виде.

Были сформулированы некоторые требования:

1. Должна быть возможность простого добавления новых бенчмарков без существенной модификации кода программы.
2. Должна быть возможность указывать различные параметры компиляции кода бенчмарков.
3. Время работы программы должно измеряться от начала и до конца ее работы с помощью программы perf.
4. Каждый бенчмарк должен проверять, что он отработал функционально правильно. Это делается с помощью возврата из функции main() числа: если число равно нулю, то считается, что бенчмарк выполнился верно.
5. Должна поддерживаться возможность исполнения как одного конкретного бенчмарка по его имени, так и группы бенчмарков (включая и возможность запуска всех возможных бенчмарков).
6. Для каждого выполненного бенчмарка необходимо отобразить: его стандартный вывод, код возврата и время выполнения. Если бенчмарк выполнился неудачно, необходимо сигнализировать пользователю об этом.

В результате было решено написать программу на Python на основе фреймворка тестирования pytest. Pytest позволит из коробки получить возможность легкого добавления новых бенчмарков, их централизированного запуска и симпатичного отображения результатов.

Каждый бенчмарк представляется отдельным тест-кейсом (в терминологии pytest). Используя утилитарные функции каждый тест-кейс сможет без труда вызывать внешние программы с произвольными аргументами. Кроме того, для perf написана обертка, которая позволяет измерять время выполнения произвольной программы.

Так как часть тест-кейсов выполняет сборку бенчмарков, которые являются частью binboost, т.е. их сборка прописана в CMake, то есть смысл создать фикстуру, выполняющую запуск cmake –build . перед каждым тест-кейсом. Фикстурой в pytest называется функция, которая будет автоматически вызвана средствами pytest перед запуском каждого тест-кейса, который с ней связан.

## 3.10 Пример работы динамической оптимизации

В качестве примера работы проекта представлена оптимизация, заключающаяся в выравнивание циклов. Программой, выступающей в качестве бенчмарка (т.е. кода, на котором тестируется производительность binboost), является обычная сортировка пузырьком массива псевдослучайных чисел размера 37500. Исходный код сортировки приведен в разделе 3.1.

В конце работы программа проверяет правильность выполненной сортировки и выводит 1, если массив был отсортирован, и 0 в противном случае.

Код в диапазоне между вызовами bb\_jit\_begin() и bb\_jit\_end() (т.е. внешний и внутренний циклы сортировки) выравнивается на 32 байта. Измеряется время полной работы программы: от запуска до проверки корректности выполнения сортировки включительно. Произведены запуски:

1. С флагом оптимизации -O3 (вариант А).
2. С включенным binboost, флагом -O3, но отключенными оптимизациями binboost (вариант Б)
3. С включенным binboost и флагом -O3 (вариант В).

В таблице 3.2 приведены усредненные измерения и погрешность. Произведено 100 измерений, полученная выборка проверена на соответствие нормальному распределению, а затем обработана как прямые измерения. Оценка истинного значения определяется как среднее арифметическое элементов выборки, а погрешность рассчитана по Стьюденту (P = 95%).

Таблица 3.2 – Измерения времени выполнения программы

|  |  |
| --- | --- |
| Вариант запуска | Время работы, с |
| А |  |
| Б |  |
| В |  |

На рисунке 3.5 показан отладочный вывод программы. В частности, показаны адреса инструкций перехода и целей перехода в оригинальном коде, перечислены обнаруженные базовые блоки вместе с адресами их начала, конца, ссылками на следующие блоки. В последней строке выведен текст, сгенерированный оптимизируемой программой, который подтверждает, что в ходе оптимизации логика работы не нарушена (сортировка проведена успешно). Также присутствуют адреса начала и конца оптимизирующего кода, размер этого блока, содержимое таблицы JumpTable и содержимое базовых блоков.

На рисунке 3.6 содержатся примеры работы отладчика gdb для конечного участка сгенерированного динамически кода. Этот код показывает окончание исходного оптимизируемого кода, а также динамически сгенерированный машинный код для загрузки адреса возврата и выполнения возврата в оригинальный код.

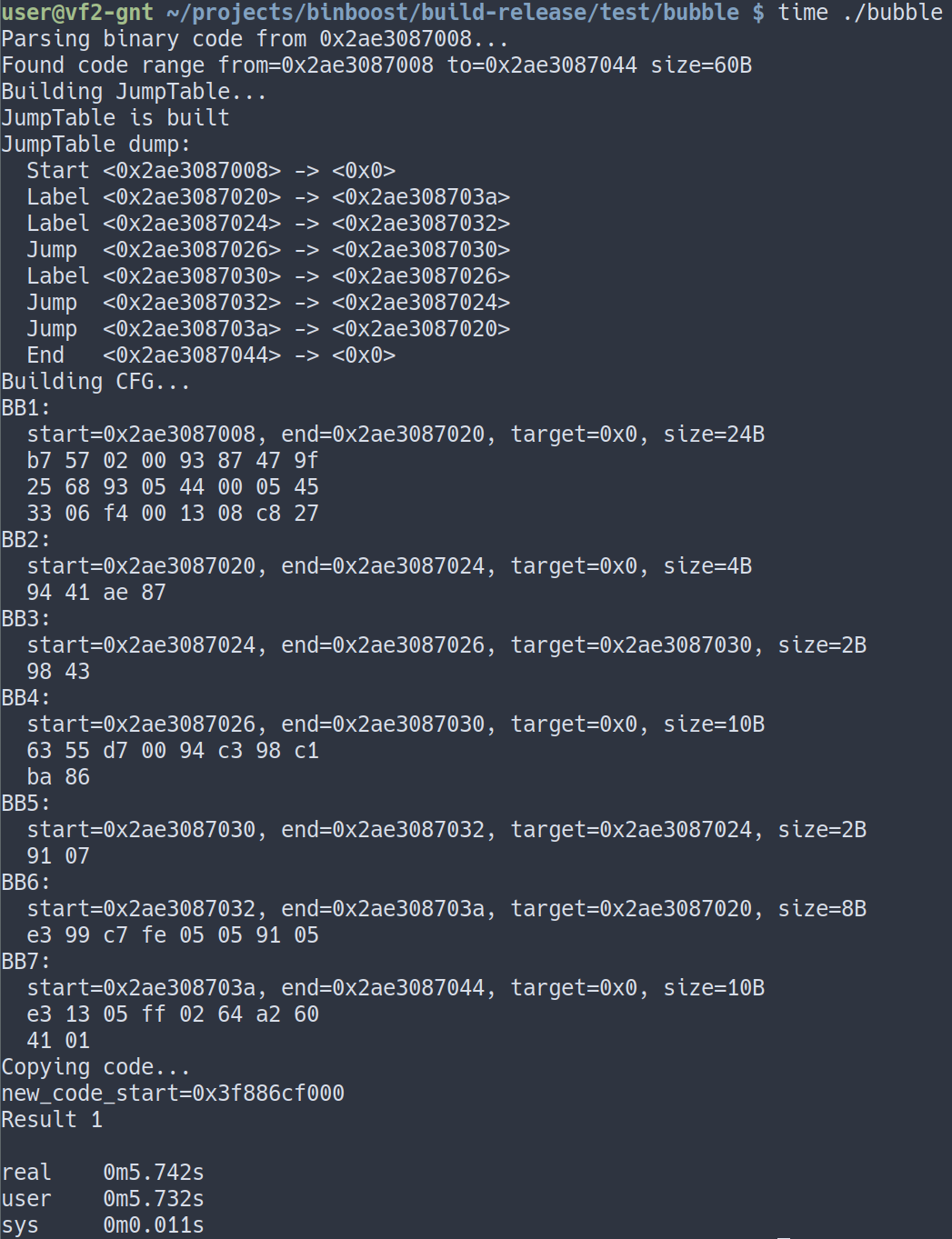


Рисунок 3.5 – Пример работы программы binboost

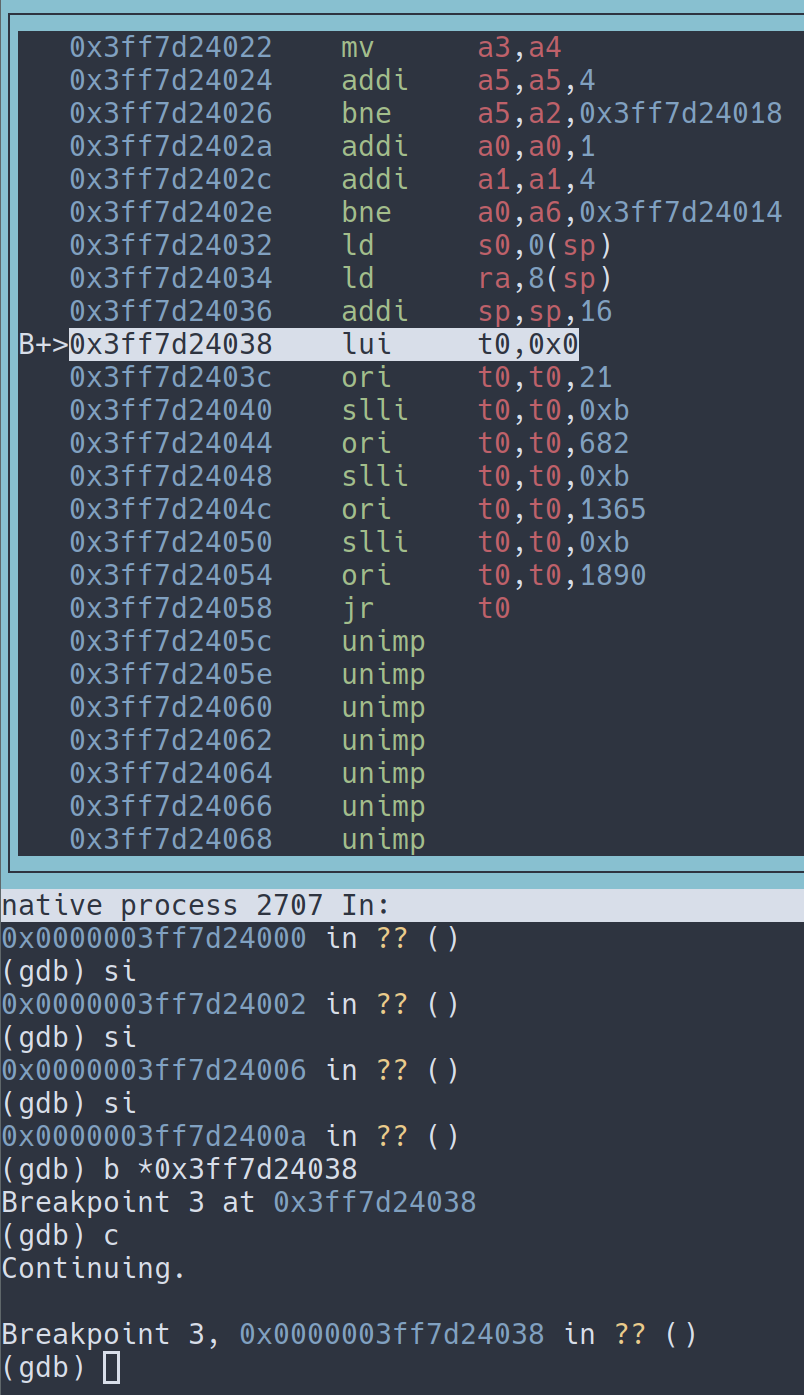


Рисунок 3.6 – Пример работы сгенерированного кода в GDB

# 4 Оценка и защита результатов интеллектуальной деятельности

Для созданного в работе продукта рассмотрим вопросы, касающиеся оценки и защиты интеллектуальной деятельности. Данный раздел посвящен определению объекта исследования в терминах Гражданского Кодекса Российской Федерации, оценке вероятностной стоимости объекта исследования доходным методом и описание принципов защиты интеллектуальных прав на объект исследования в соответствии с нормативно-правовыми актами Российской Федерации.

## 4.1 Описание результата интеллектуальной деятельности

Законодательство Российской Федерации [31] предоставляет физическим и юридическим лицам возможность правовой охраны результатов их научно-интеллектуальной деятельности. Таким образом, обеспечивается база для создания безопасных и честных условий при обороте результатов деятельности на конкурентном рынке.

Прежде чем исследовать вопрос защиты деятельности, необходимо сформулировать объект исследования: понять, в чем заключается суть интеллектуальной деятельности в соответствии с законодательством РФ. Рассматриваемый вопрос регулируется частью четвертой Гражданского Кодекса РФ. Статья 1225 Кодекса перечисляет 16 видов объектов интеллектуальной собственности. Разработанный в контексте дипломной работы проект – средство динамической оптимизации машинного кода – соответствует подпункту 2 «Программы для электронных вычислительных машин (программы для ЭВМ)».

Инструмент, разработанный в дипломном проекте, предназначен для конечных пользователей разнообразного программного обеспечения, желающих с минимальными усилиями адаптировать его к собственному аппаратному обеспечению, ускорить и раскрыть его потенциал, снизить сопровождаемую нагрузку на вычислительную систему. Для применения инструмента нет необходимости иметь доступ к исходному коду оптимизируемого продукта, поэтому ограничений на множество доступных продуктов нет. Кроме того, инструмент будет интересен и корпоративным клиентам, разворачивающим свои собственные решения и желающим добиться лучших показателей производительности, а значит снизить расходы, с одной стороны, на аппаратное обеспечение (инфраструктуру), и, с другой стороны, на процесс разработки, сократить объем работы, связанный с оптимизацией ПО.

## 4.2 Оценка рыночной стоимости результата интеллектуальной деятельности

Рыночная цена – это цена продукта, по которой права за него могут быть отчуждены на открытом и конкурентном рынке при действии сторон сделки разумно, с учетом всей информации. На цену сделки не отражаются любые чрезвычайные обстоятельства.

Разрабатываемый в дипломной работе проект является программой для ЭВМ, то есть нематериальным активом – он проявляет свои экономические свойства, при этом не имея материально-вещественной формы. Как показано в предыдущем подразделе, объект оценки способен удовлетворять определенные потребности при использовании, и на него есть спрос.

Рыночная стоимость проекта рассчитывается с помощью затратного и доходного подходов. Сперва применяется затратный.

Разработка инструмента займет 12 месяцев. В создании проекта принимает участие 4 человека, их заработная плата из расчета 100 000 рублей на человека в месяц в сумме с учетом налогов составляет рублей.

Для эффективной разработки необходим также парк техники, на которой можно было бы выполнять отладку и тестирование. В парк должны входить процессоры различных архитектур (x86-64, ARM, RISC-V) и семейств. Процессоры, которые планируется иметь (на основе популярности и целесообразности для проекта), представлены в таблице 4.1.

Таблица 4.1 — Расходы на техническое обеспечение

|  |  |
| --- | --- |
| Название процессора (платформы) | Стоимость, рублей |
| Intel Core i5-14400 (x86-64), материнская плата | 29498 |
| Intel Core i5-11400 (x86-64), материнская плата | 16298 |
| AMD Ryzen 5 5600X (x86-64), материнская плата | 20739 |
| ARM Cortex-A76 (ARM) в комплекте Raspberry PI 5 | 24899 |
| Apple M1 (ARM) в комплекте MacBook Air M1 | 94499 |
| JH7110 (RISC-V) в комплекте StarFive VisionFive 2 | 10597 |
| Итого | 196530 |

Работа сотрудников предполагается удаленная, поэтому расходов на помещение и сопутствующие траты нет.

Таким образом, за один год работы расходы составят 6 446 130 рублей.

В качестве альтернативного подхода для оценки стоимости применяется доходный подход (метод капитализации доходов), так как есть все возможности спрогнозировать будущие доходы на основе распространения подписочной лицензии на продукт и выполнить анализ рисков.

Используемый метод основывается на применении коэффициента капитализации *К*, представляющего разнице между ставкой дисконтирования и прогнозируемым приростом прибыли, для получения текущей стоимости будущих доходов *Стек* на основе прогнозируемого дохода *Д*:

Коэффициент К рассчитывает по следующей формуле:

где *R* – ставка дисконтирования,

*g* – прогнозируемый прирост дохода,

*Rб* – ключевая ставка Центрального Банка.

В соответствии с формулой ставка дисконтирования учитывает дополнительные риски к безрисковой ставке дисконтирования. Дополнительные риски состоят из:

* риска внедрения в производство продукта (*Rп*);
* риска неудачного продвижения продукта (*Rн*);
* риска низкой влиятельности при распределении совокупных выгод от реализации (*Rв*);
* риска высоких расходов и неэффективных инвестиций (*Rи*).

Прогнозируемый доход определяется на основе анализа потенциального рынка с целью оценки реалистичного количества проданных лицензий, т.е. оценки будущего дохода. Прямых аналогов разрабатываемому в дипломе проекту нет, а попытки найти информацию по продажам схожих продуктов (анализаторы кода, компиляторы, среды разработки) не принесли успеха по причине коммерческой тайны. Поэтому дальше в качестве прогнозируемого дохода берется пессимистичный ожидаемый доход, обеспечивающий полную окупаемость и 10-процетную чистую выручку.

Распространять продукт планируется в виде продажи лицензии на пользование инструментом сроком в 1 год. Для физических лиц стоимость лицензии будет составлять 10 000 рублей, а для юридических 500 000 рублей. Целью является покрытие расходов на разработку и получение около 10% чистой прибыли, т.е. с учетом расходов необходимо заработать, округляя вверх до десятков тысяч, 7 100 000 рублей.

Это означает, что необходимо продать 710 лицензию для физических лиц или 15 лицензий для юридических лиц для получения планируемого дохода в 7 100 000 рублей. Как следует из подраздела 4.1, обеспечить продажу 15 лицензий в течение года вполне реалистично: емкость рынка для разрабатываемого продукта большая, так как он может применяться в самых разных условиях и не накладывает ограничений на сферу. Продукт возможно использовать как для ускорения обычного программного обеспечения (например, веб-серверов), так и специализированного (например, математические пакеты, нейросетевые вычисления).

Ниже представлен анализ рисков с помощью таблиц 4.1-4.4 и определена величина каждого из них.

Таблица 4.1 — Риски внедрения в производство

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Риск | 1% | 2% | 3% | 4% | 5% |
| Известность имени в нескольких отраслях | Общеизвестное имя | Известное в отрасли | Узкопрофильный специалист | Малоизвестное имя | Неизвестное имя |
| Опыт применения ОИС в производстве | Многократный в различных областях | Многократный отраслевой | Несколько раз в одной отрасли | Один раз в одной отрасли | Нет |
| Наличие научной известности | Общемировая | Государственная | Есть научная степень | Есть  научные  труды | Нет |

Таблица 4.2 — Риски неудачного продвижения продукта

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Риск | 1% | 2% | 3% | 4% | 5% |
| Степень разработки | Комерциализация в производстве | Опытное производство | Разработана технология на базе НИОКР | НИР, НИОКР | Регистрация ОИС |
| Наличие и объемы рынка | Несколько рынков с большими объемами | Несколько рынков | Несколько сегментов рынка | Один сегмент рынка | Рынок не сформирован |
| Рекламная компания | Многолетняя плановая | Годовая плановая | Отработанная | В начале | Не ведется |

Таблица 4.3 — Риски низкой влиятельности при распределении совокупных выгод от реализации

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Риск | 1% | 2% | 3% | 4% | 5% |
| Степень  использования  ОИС в  отдельных  элементах  товарной  продукции | Использование  во всем  изделии | Использование  в основных  элементах | Использование  в одном  основном  элементе | Использование  в нескольких  второстепенны  х элементах | Использование  во  второстепенно  м элементе |
| Стратегия  ценообразован  ия продукта  ОИС | Полностью  разработана и  апробирована  многократно | Разработана и  апробирована | В стадии  апробации | В стадии  разработки | Нет |
| Менеджмент | Имеющий  большую  практику  выпуска и  реализации  продукции | Имеющий  небольшую  практику  выпуска и  реализации  продукции | В стадии  разработки  реализации  продукции | В стадии  набора и  обучения | Отсутствие  специализированного  менеджмента |

Таблица 4.4 — Риски высоких расходов и неэффективных инвестиций

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Риск | 1% | 2% | 3% | 4% | 5% |
| Опыт  применения  аналогичных  ОИС на  предприятии | Многократный | Однократный | В процессе отработки | Есть опыт  применения на  аналогичных  предприятиях | Нет |
| Наличие на  рынке  конкурентного  контрафакта | Нет | Возможно появление | Есть в небольшом количестве | Растущая динамика роста | Есть в большом количестве |
| Наличие на  рынке  конкурентных  ОИС | Нет | Возможно появление | Есть в небольшом количестве | Среднеконкурентные ОИС | Высококонкурентные ОИС |

На май 2024 года ключевая ставка Центрального Банка равна 16,0%. Примем прогнозируемый прирост дохода равным текущему уровню инфляции (7,72 на март 2024 года). Тогда с учетом определенных выше рисков получим

Таким образом, рыночная стоимость затратным подходом получилась равной 6 446 130 рублей, а доходным – 36 261 492 рублей. Для приведение полученных оценок к одной стоимости применяется взвешенное среднеарифметическое. По сравнению с затратным доходный подход имеет более глубокую проработку, так как учитывает инвестиционные риски, связанные с разработкой проекта. С другой стороны, ввиду существенных преимуществ, которые предоставляет продукт пользователям, и отсутствия прямых конкурентов, влияние рисков может быть завышенным и не сказываться столь сильно на рыночную стоимость. Тем самым, будет разумным взять веса для оценок в соотношении 3:7. Тогда итоговая оценка равна

## 4.3 Правовая защита результатов интеллектуальной деятельности

Правовая охрана может обеспечиваться патентным законодательством, правом на коммерческую тайну, авторским правом или законодательством о средствах индивидуализации. Нарушение прав интеллектуальной собственности влечет гражданскую, административную и уголовную ответственность.

Так как результатом дипломной работы является компьютерная программа, то защита прав определяется авторским правом, суть которого сформулирована в статье 1261 главы 70 ГК РФ «Программы для ЭВМ».

Коротко, авторское право в контексте программ для ЭВМ сводится к тому, что программы, выраженные в любой форме и на любом языке (исходные тексты и объектный код) охраняются по тем же принципам, что и авторские права на произведения литературы. В законе программой считается совокупность данных и команд, представленная в объективной форме и предназначенная для функционирования как на ЭВМ, так и на других компьютерных устройствах в целях получения результата. Сюда относится и сама программа, и порождаемые ею аудиовизуальные отображения.

Авторское право распространяется вне зависимости от коммерческой реализации программы. Оно имеет силу в течение всей жизни автора и дополнительно в течение 50 лет после его смерти.

Товарный знак охраняется после его государственной регистрации.

Возможна передача права использования объекта интеллектуальной собственности. Для этого должен быть сформирован лицензионный договор (или договор об уступке прав), содержащий следующие условия:

* предмет договора;
* объем передаваемых прав;
* срок своего действия;
* территорию своего действия;
* цену одной лицензии;
* гарантии и ответственности сторон;
* обеспечение конфиденциальности;
* условия разрешения спора и расторжения договора.

Хотя право авторства предоставляется автору программы безусловно по факту создания этой программы, иногда имеет смысл государственная регистрация программы в соответствии со статьей 1262 ГК РФ «Государственная регистрация программ для ЭВМ и баз данных». Она не только формально подтверждает авторство, но также повышает степень его защищенности в случае появления различных споров по вопросам собственности и нарушения авторских прав: защита от несанкционированного использования (копирование, распространение, применение, заимствование исходных кодов и другое), упрощение доказательства авторства, возможность распоряжаться объектом авторского права.

Регистрация программы выполняется путем получения специального свидетельства в органах власти. Для этого необходимо предоставить ряд документов:

* заявление о регистрации программы для ЭВМ, где указывается правообладатель и автор;
* депонируемые материалы для идентификации программы (исходный код);
* согласие автора на указание сведений о нем в заявлении;
* подтверждение полномочий заявителя;
* подтверждение уплаты государственной пошлины;
* согласие на обработку персональных данных;

Охрана и защита результатов интеллектуальной собственности предусмотрена пунктом 1 статьи 44 Конституции Российской Федерации («интеллектуальная собственность охраняется законом»). Законным актом, реализующим охрану, является Гражданский Кодекс Российской Федерации, а точнее главы 69-77 ее четвертой части раздела VII. Перечень объектов, на которые распространяется охрана результатов интеллектуальной деятельности, приведен в статье 1225 ГК РФ. Содержание понятия «интеллектуальные права» раскрывается в статьях 1226, 1227 ГК РФ: оно состоит из исключительных прав, личных неимущественных прав и иных прав и существует независимо от права на материальный носитель. Статья 1228 ГК РФ устанавливает сущность автора результата и его права. Сущность исключительных прав приведена в статьях 1229, 1230, 1230.1 ГК РФ.

Защита предусмотрена мерами ГК РФ гражданско-правовой части. В соответствии со статьями 12 и 1252 обладатель исключительных прав на результаты вправе требовать от нарушителя 1) признания прав (в том числе публичным объявлением); 2) восстановления положения до нарушения и пресечения действий, которые нарушают или создают угрозу нарушения; 3) возмещения убытков (в том числе упущенную выгоду); 4) взыскания доходов нарушителя вследствие нарушения авторских прав; 5) компенсации морального вреда (только если нарушены личные неимущественные права).

Кроме гражданско-правовых мер защиты существуют меры административного-правового характера, а именно статья 7.12 Кодекса Российской Федерации об административных правонарушениях предусматривает наложение штрафов от 1 тыс. до 40 тыс. рублей (в зависимости от тяжести и вида лица) и конфискацию контрафактных экземпляров в случае «нарушения авторских прав путем ввоза, продажи, сдачи в прокат или иного незаконного использования экземпляров произведений или фонограмм в коммерческих целях».

В случае, если нарушение авторских прав причинили крупный ущерб и совершении такого деяния группой лиц по предварительному сговору, в соответствии с Уголовным кодексом Российской Федерации наказывается штрафом, обязательными работами, принудительными работами или лишением свободы (статьи 146, 147 УК РФ).

# ЗАКЛЮЧЕНИЕ

В результате выполнения дипломной работы были выполнены все поставленные задачи: создан фреймворк для обеспечения динамической оптимизации машинного кода для архитектуры RISC-V под управлением операционной системы GNU/Linux, проведен анализ методов оптимизации программ во время их выполнения, реализована и протестирована оптимизация выравниванием кода циклов.

Исследование и тестирование показало целесообразность динамической оптимизации, так как скорость работы бенчмарка выросла в сравнении со статическими оптимизациями кода, а проведенный анализ указал на другие оптимизации, которые будут эффективны для машинного кода. Таким образом, накладные расходы применения динамических оптимизаций не перекрывают выгоды от самих оптимизаций.

Гибкость разработанной библиотеки позволяет в дальнейшем расширить количество применяемых оптимизаций, добавить поддержку новых компьютерных архитектур (в частности, x86-64 и ARM) и обеспечить автоматическое применение оптимизаций без участия в процессе разработчика.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Скиена С. Алгоритмы. Руководство по разработке / Пер. с англ. – 2-е изд. – СПб: БХВ-Петербург, 2011. – 720 с.
2. ГОСТ 28195-89. Оценка качества программных продуктов. – М.: ИПК издательство стандартов, 2001. – 31 с.
3. Think with Google. Find out how you stack up to new industry benchmarks for mobile page speed [Электронный ресурс]. – URL: <https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks/> (дата обращения 10.05.2024).
4. Kegel D. The C10K problem [Электронный ресурс]. – URL: <http://www.kegel.com/c10k.html> (дата обращения 10.05.2024).
5. Zhu H. Data Plane Development KIT (DPDK). A Software Optimization Guide to the User Space-based Network Applications. – Boca Raton: CRC Press, 2021. – 336 p.
6. Kleppmann M. Designing Data-Intensive Applications. – Sebastopol: O’Reilly Media, 2017. – 613 p.
7. Fog A. Optimizing software in C++. – Lyngby: Technical University of Denmark, 2024. – 179 p.
8. Fog A. Optimizing subroutines in assembly language. – Lyngby: Technical University of Denmark, 2023. – 155 p.
9. LLVM Documentation. LLVM’s Analysis and Transform Passes [Электронный ресурс]. – URL: <https://llvm.org/docs/Passes.html> (дата обращения 10.05.2024).
10. GNU GCC Documentation. Link-Time Optimization in GCC: Requirements and High-Level Design [Электронный ресурс]. – URL: <https://gcc.gnu.org/projects/lto/lto.pdf> (дата обращения 10.05.2024).
11. Go User Manual. Profile-guided optimization [Электронный ресурс]. – URL: <https://go.dev/doc/pgo> (дата обращения 10.05.2024).
12. Panchenko M., Auler R., Nell B. BOLT: a practical binary optimizer for data centers and beyond // CGO 2019: Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization. – Piscataway: IEEE Press, 2019 – P. 2-14.
13. Ahmed O. M. C# 7 and .NET Core 2.0 High Performance. – Birmingham: Packt Publishing, 2018. – 300 p.
14. Beckwith M. JVM Performance Engineering: Inside OpenJDK and the HotSpot Java Virtual Machine (Developer's Library). – Independence: Oracle Press, 2024. – 400 p.
15. Wade A., Kulkarni P. Exploring Impact of Profile Data on Code Quality in the HotSpot JVM // ACM Transactions on Embedded Computing Systems. – 2020. – Vol. 19, №6. – P. 1-26.
16. The Case for Writing Network Drivers in High-Level Programming Languages / P. Emerich, E. Ellmann, F. Bonk et al. – Munich: Technical University of Munich, 2019. – 16 p.
17. DynamoRIO Official Site. Overview [Электронный ресурс]. – <https://dynamorio.org/page_home.html> (дата обращения 10.05.2024).
18. Bruening D., Garnett T. An Infrastructure for Adaptive Dynamic Optimization. – Cambridge: Massachusetts Institute of Technology, 2003. – 11 p.
19. Desoli G., Mateev N., Duesterwald E. DELI: A new run-time control point // 35th Annual International Symposium on Microarchitecture (MI-CRO’02). – Piscataway: IEEE Press, 2002. – P. 257-268.
20. Davidson S., Davidson J. Strata: A software dynamic translation infrastructure // Proceedings of the IEEE 2001 Workshop on Binary Translation. – Piscataway: IEEE Press, 2001. – P. 1-10.
21. Fog A. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. – Lyngby: Technical University of Denmark, 2024. – 179 p.
22. Bala V., Duesterwald E., Banerjia S. Dynamo: A transparent runtime optimization system // Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’00). – New York: Association for Computing Machinery, 2000. – P. 1-12.
23. Chen W., Lerner S. Mojo: A Dynamic Optimization System. – Redmond: Microsoft Research, 2000. – 10 p.
24. Waterman A., Asanović K. The RISC-V Instruction Set Manual, Volume 1: User-Level ISA. – Berkley: RISC-V International, 2019. – 236 p.
25. Cheng K., Clarke J. RISC-V ABIs Specification. – Berkley: RISC-V International, 2022 – 56 p.
26. Drepper U. What Every Programmer Should Know About Memory. Red Hat [Электронный ресурс]. – URL: <https://people.freebsd.org/~lstewart/articles/cpumemory.pdf> (дата обращения 10.05.2024).
27. Pathak K. Loop alignment in .NET 6 [Электронный ресурс]. – URL: <https://devblogs.microsoft.com/dotnet/loop-alignment-in-net-6/> (дата обращения 10.05.2024).
28. RISC-V Specifications Repository. RISC-V "V" Vector Extension [Электронный ресурс]. – URL: <https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc> (дата обращения 10.05.2024).
29. Mendis C., Amarasighe S. goSLP: Globally Optimized Superword Level Parallelism Framework // Proc. ACM Program. Lang. 2, OOPSLA. – New York: Association for Computing Machinery, 2018. – P. 1-28.
30. Ахо А., Лам М., Сети Р., Ульман Д. Компиляторы: принципы, технологии и инструментарий. – 2 изд. – М.: Вильямс, 2008. – 1175 с.
31. Гражданский кодекс Российской Федерации (часть четвертая) от 18.12.2006, № 230-ФЗ (ред. от 30.01.2024) // Собрание законодательства РФ. – 05.02.2024 – № 6. – Ст. 1225-1230.1.
32. Магомедов М. Н., Чигирь М. В. Оценка и защита результатов интеллектуальной деятельности: учебно-методическое пособие по выполнению дополнительного раздела выпускных квалификационных работ магистров. – СПб.: Изд-во СПбГЭТУ “ЛЭТИ”, 2019 – 26 с.

# ПРИЛОЖЕНИЕ А Фрагменты исходного кода проекта

// src/binboost.cpp

// Точка входа в JIT-компилятор. Реализация фукнций bb\_jit\_begin\_impl() и  
// bb\_jit\_end().

#include "arch/intrinsics.hpp"

#include "binboost/bb.h"

#include <cstdio>

#include <cstdlib>

#include "arch/intrinsics.hpp"

#include "basic\_block\_parser.hpp"

#include "code\_gen.hpp"

#include "code\_analyzer.hpp"

extern "C" {

// Реализация точки входа в оптимизатор

std::uintptr\_t bb\_jit\_begin\_impl(std::uintptr\_t opt\_code\_start) {

using namespace bb;

// Если bb\_jit\_begin() не передал адрес возврата в первом параметре,  
// определяем адрес возврата сами.

if (opt\_code\_start == 0) {

opt\_code\_start = bb::arch::GetReturnAddr();

}

std::printf("Parsing binary code from 0x%lx...\n", opt\_code\_start);

// Выполняет построение CFG.

BasicBlockParser parser(opt\_code\_start);

auto bbs = parser.Parse();

if (!bbs.IsValid()) {

// Прекращаем работу, если не удалось построить CFG.

std::fprintf(stderr,

"Panic: failed to find bb\_jit\_end() call (from=0x%02lx)\n",

opt\_code\_start);

std::exit(1);

}

// Отладочный вывод списка получившихся BasicBlock.

bbs.Dump(stdout);

// Применяем оптимизацию выравнивания циклов.

CodeAligner aligner(&bbs);

aligner.DoAlign();

// Выполняет копирование нового кода, представленного CFG, в специальную   
 // для него область памяти.

CodeGen code\_gen(&bbs);

std::printf("Copying code...\n");

auto new\_code\_start = code\_gen.CopyCode();

std::printf("new\_code\_start=0x%lx\n", new\_code\_start);

// Возвращаем адрес начала скопированного выше кода. Bb\_jit\_begin() в конце   
 // работы произведет переход по этому адресу.

return new\_code\_start;

}

// Функция, отмечающая завешение диапазона кода, который нужно   
// оптимизировать. Вызов этой функции ищется внутри исходного машинного кода.

void bb\_jit\_end() {

// Чтобы компилятор не соптимизировал функцию.

\_\_asm\_\_ volatile("");

}

// Определяем пустую функцию bb\_jit\_begin(), если параметрами компиляции  
// binboost был отключен.

#ifdef BB\_EXCLUDE\_TRAMPOLINE

void bb\_jit\_begin() {

fprintf(

stderr,

"Panic: BB\_EXCLUDE\_TRAMPOLINE enabled, program launch is imposible\n");

std::exit(1);

}

#endif // BB\_EXCLUDE\_TRAMPLINE

}

// src/arch/riscv/decoder.cpp

// Фрагменты реализации декодера инструкций RISC-V.

#include "arch/decoder.hpp"

#include <cstdio>

#include "arch/instr.hpp"

namespace bb::arch {

namespace {

// Представление часть данных инструкции, над которыми должно быть выполнено расширение знака.

// T – структура с интерфейсом Format.

template <typename T>

struct SignExtend {

// Число битов в данных.

static constexpr auto kWidth = T::kWidth - 1;

// Выполняет получение данных и расширение их знака.

static constexpr std::int64\_t Decode(std::uint32\_t instr) {

struct {

std::int64\_t x : kWidth;

} s;

return s.x = T::Decode(instr);

}

};

// Декодирует данные, как набор битов, расположенных в инструкции между битами MsbOff (старший) и LsbOff (младший).

template <int MsbOff, int LsbOff>

struct Bit {

static\_assert(MsbOff >= LsbOff);

static constexpr int kMsbOffset = MsbOff;

static constexpr int kLsbOffset = LsbOff;

// Количество битов, которые будут декодированы.

static constexpr int kWidth = MsbOff - LsbOff + 1;

// Вынимает из val kWidth младших битов.

static constexpr std::uint32\_t Decode(std::uint32\_t val) {

return val & ((1 << kWidth) - 1);

}

};

// Пропускает в инструкции следующие Count битов (ничего с ними не делает).

template <int Count>

struct Skip {

static constexpr int kWidth = Count;

static constexpr int kLsbOffset = 0;

static constexpr std::uint32\_t Decode(std::uint32\_t) { return 0; }

};

// Формат инструкции представляется в виде последовательности структур Bit  
// или Skip. Относительное положение этих структур в списке Bits будет   
// соответствовать положению битов, описанных Bits, в декодированных данных.

template <typename... Bits>

struct Format {

static constexpr int kWidth = 0;

static constexpr std::uint32\_t Decode(std::uint32\_t) { return 0; }

};

template <typename BitHead, typename... Bits>

struct Format<BitHead, Bits...> {

using BitTail = Format<Bits...>;

static constexpr int kWidth = BitHead::kWidth + BitTail::kWidth;

static constexpr std::uint32\_t Decode(std::uint32\_t val) {

// Вытаскиваем из val бит, соответствующий порядковому BitHead.

auto bits = BitHead::Decode(val >> BitTail::kWidth);

// Передвигаем бит на нужную позицию и добавляем значения

// остальных битов.

return (bits << BitHead::kLsbOffset) | BitTail::Decode(val);

}

};

// Вспомогательные псевдонимы типов для упрощения.

template <int MsbOff, int LsbOff = MsbOff>

using B = Bit<MsbOff, LsbOff>;

template <int Count>

using S = Skip<Count>;

// Описание декодера инструкций перехода (j-type).

//

// 31 30 20 19 11 6

// J imm[20] imm[10:1] imm[11] imm[19:12] rd opcode

//

// opcode – опкод инструкции.

// rd – номер регистра, в который записывается адрес перехода.

// imm – относительный адрес перехода.

struct JFormat {

// Декодирует часть инструкции, обозначаемую opcode.

static constexpr auto ReadOp(std::uint32\_t instr) {

return Format<B<6, 0>>::Decode(instr);

}

// Декодирует часть инструкции, обозначаемую rd .

static constexpr auto ReadRd(std::uint32\_t instr) {

return Format<B<4, 0>, S<7>>::Decode(instr);

}

// Декодирует часть инструкции, обозначаемую imm.

static constexpr auto ReadImm(std::uint32\_t instr) {

return SignExtend<Format<B<20>, B<10, 1>, B<11>, B<19, 12>, S<12>>> //

::Decode(instr);

}

// То же, что ReadImm().

static constexpr auto ReadOffset(std::uint32\_t instr) {

return ReadImm(instr);

}

// Проверяет, что данные instr соответствуют инструкции перехода, т.е.  
 // имеют опкод 0x6f или 0x67.

static constexpr bool IsValid(std::uint32\_t instr) {

auto op = ReadOp(instr);

return op == 0x6f || op == 0x67;

}

};

} // namespace

// Пробует декодировать данные по адресу instr\_, как инструкцию условного или   
// безусловного перехода.

//

// В случае удачи возвращает true и записывает в переменную instr информацию   
// о переходе. Иначе возвращает false.

template <>

bool Decoder::TryDecode<InstrCategory::Jump>(JumpInstr\* instr) const {

// Интерпретируем память как инструкцию размером 16 или 32 бита.

std::uint16\_t instr16 = \*instr\_;

std::uint32\_t instr32 = (\*(instr\_ + 1) << 16) | (instr16);

// Вспомогательная функция, пытающая декодировать instr\_data с помощью де  
 // кодера Format.

auto try\_decode = [this, instr]<typename Format>(auto instr\_data) {

if (!Format::IsValid(instr\_data)) {

return false;

}

instr->data = instr\_;

instr->target = instr\_ + (Format::ReadOffset(instr\_data) / 2);

return true;

};

// Последовательно пытаемся распознать инструкцию как сжатую безусловного   
 // перехода, обычную безусловного или обычную условного.

return //

try\_decode.operator()<CjFormat>(instr16) || //

try\_decode.operator()<JFormat>(instr32) || //

try\_decode.operator()<BFormat>(instr32);

}

// Проверяет, что инструкция instr\_ является сжатой.

bool Decoder::IsCompressed() const {

// 32-битные инструкции начинаются с 0b11 в младших битах

// 16-битные -- с 0b00, 0b01, 0b10

return (\*instr\_ & 0x3) != 0b11;

}

// Возвращает размер инструкции по адресу instr\_ в 16-битных словах.

int Decoder::GetInstrSizeMultiplier() const { return IsCompressed() ? 1 : 2; }

// Тесты времени компиляции для декодера.

namespace test {

struct JFormatTest {

// Пытаемся выполнить декодирование инструкции jal x1, 23866

// Или в численном значении:

// 0x53b050ef = 0b0101'0011'1011'0000'0101'0000'1110'1111

static\_assert(JFormat::ReadOp(0x53b050ef) == 0b1101111);

static\_assert(JFormat::ReadRd(0x53b050ef) == 1);

static\_assert(JFormat::ReadOffset(0x53b050ef) == 23866);

static\_assert(JFormat::IsValid(0x53b050ef));

};

} // namespace test

} // namespace bb::arch

// src/basic\_block\_parser.hpp

// Интерфейс классов для создание CFG.

#pragma once

#include <cstddef>

#include <cstdint>

#include <cstdio>

#include <list>

#include "arch/instr.hpp"

namespace bb {

// Диапазон кода.

struct CodeRange {

/// Виртуальный адрес начала диапазона (включительно).

std::uintptr\_t start;

/// Виртуальный адрес конца диапазона (включительно).

std::uintptr\_t end;

/// Размер диапазона.

std::size\_t GetSize() const { return end - start; }

// Представляет ли структура валидный диапазон.

bool IsValid() const { return end > 0 && start <= end; }

// Возвращает указатель на начало диапазона.

arch::InstrData\* GetStartPtr() const {

return arch::AddrToPtr<arch::InstrData\*>(start);

}

// Возвращает указатель на конец диапазона.

arch::InstrData\* GetEndPtr() const {

return arch::AddrToPtr<arch::InstrData\*>(end);

}

};

// Динамическое хранилище памяти для инструкций, которые создаются в процессе

// оптимизации кода.

class CodeStore {

public:

CodeStore();

// Выделяет диапазон инструкций размера size.

CodeRange AllocRange(std::size\_t size);

private:

static constexpr auto kMaxSize = 512;

std::size\_t end\_ = 0;

std::vector<std::byte> code\_;

};

// Базовый блок.

//

// Последовательность инструкций, непрерывающихся инструкциями перехода или

// метками.

//

// Может иметь 0 или больше родителей и 1 или 2 дочерних узла.

//

// Может ссылать на инструкции в оптимизируемой программы или на инструкции из

// CodeStore. В любом случае, всегда иммутабелен.

class BasicBlock {

public:

BasicBlock() = default;

// Создает новый ББ, представляющий инструкции с адреса s до адреса e.

BasicBlock(const arch::InstrData\* s, const arch::InstrData\* e)

: start\_(s), size\_(CalculateSizeFromStartAndEnd(s, e)) {}

// Возвращает адрес на первую инструкцию.

const arch::InstrData\* GetStartPtr() const { return start\_; }

// Возвращает адрес на инструкцию, идущую за последней.

const arch::InstrData\* GetEndPtr() const { return start\_ + size\_; }

// Возвращает размер блока.

std::size\_t GetSize() const { return size\_ \* sizeof(arch::InstrData); }

// Возвращает указатель на следующий ББ.

BasicBlock\* GetNext() { return next\_; }

// Возвращает константный указатель на следующий ББ.

const BasicBlock\* GetNext() const { return next\_; }

// Возвращает указатель на ББ, в который совершается переход из данного.

BasicBlock\* GetJumpTarget() { return jmp\_target\_; }

// Устанавливает следующий ББ.

void SetNext(BasicBlock\* bb) { next\_ = bb; }

// Устанавливает ББ, в который совершается переход из данного.

void SetJumpTarget(BasicBlock\* bb) { jmp\_target\_ = bb; }

private:

std::size\_t CalculateSizeFromStartAndEnd(const arch::InstrData\* s,

const arch::InstrData\* e) {

return e - s;

}

const arch::InstrData\* start\_;

std::size\_t size\_;

BasicBlock\* jmp\_target\_ = nullptr;

BasicBlock\* next\_ = nullptr;

};

// Представляет CFG -- граф BasicBlock

class BasicBlockCollection {

public:

// Контейнер, в котором хранятся BasicBlock.

using Container =

std::list<BasicBlock>;

// Итераторы CFG.

using iterator = Container::iterator;

using const\_iterator = Container::const\_iterator;

BasicBlockCollection() = default;

// Создает CFG, представляющий код по указанному диапазону.

explicit BasicBlockCollection(std::uintptr\_t original\_start,

std::uintptr\_t original\_end)

: original\_start\_(original\_start), original\_end\_(original\_end) {}

// Вставляет в CFG новый ББ.

BasicBlock& Insert(arch::InstrData\* start, arch::InstrData\* end);

// Возвращает корень CFG.

BasicBlock& GetFirst() { return \*blocks\_.begin(); }

// Выводит отладочную информацию о CFG.

void Dump(FILE\* f);

// Возвращает число ББ в CFG.

std::size\_t Size() const { return blocks\_.size(); }

// Проверяет корректен ли CFG.

bool IsValid() const { return blocks\_.size() > 0; }

// Возвращает конец исходного кода, для которого построен CFG.

std::uintptr\_t GetOriginalEnd() const { return original\_end\_; }

// Возвращает итераторы на начало и конец CFG.

auto begin() const { return blocks\_.begin(); }

auto end() const { return blocks\_.end(); }

// Возвращает хранилище инструкций.

CodeStore& GetCodeStore() { return code\_store\_; }

private:

std::uintptr\_t original\_start\_;

std::uintptr\_t original\_end\_;

Container blocks\_;

CodeStore code\_store\_;

};

// Выполняет построение CFG.

class BasicBlockParser {

public:

explicit BasicBlockParser(std::uintptr\_t start) : start\_(start) {}

// Строит CFG.

BasicBlockCollection Parse();

private:

CodeRange ParseRange();

std::uintptr\_t start\_;

};

} // namespace bb

// test/bubble/bubble.cpp

// Бенчмарк работы выравнивания кода.

#include <cstdio>

#include "binboost/bb.h"

// Выполняет сортировку массива размера kBound методом пузырька.

class BubbleSort2 {

public:

// Число элементов в массиве.

static constexpr int kBound = 50 \* 15;

// Непосредственно функция сортировки.

\_\_attribute\_\_((noinline)) static void Inner(int\* x) {

auto limit1 = kBound - 1;

BB\_JIT\_BEGIN();

for (int i = 1; i <= limit1; ++i) {

for (int j = i; j <= kBound; ++j) {

if (x[i] > x[j]) {

int temp = x[j];

x[j] = x[i];

x[i] = temp;

}

}

}

BB\_JIT\_END();

}

// Генерирует массив, сортируем его и проверяет, корректно ли он отсортирован.

static bool Test() {

// Выделение памяти под массив.

int\* x = new int[kBound + 1];

int j = 99999;

int limit = kBound - 2;

int i = 1;

// Псевдослучайная генерация элементов массива.

do {

x[i] = j & 32767;

x[i + 1] = (j + 11111) & 32767;

x[i + 2] = (j + 22222) & 32767;

j = j + 33333;

i = i + 3;

} while (i <= limit);

x[kBound - 1] = j;

x[kBound] = j;

// Сортировка.

Inner(x);

// Проверка, что массив отсортирован.

auto is\_sorted = true;

for (i = 0; i < kBound - 1; ++i) {

if (x[i] > x[i + 1]) {

is\_sorted = false;

break;

}

}

delete[] x;

return is\_sorted;

}

};

int main() {

auto res = BubbleSort2::Test();

printf("Result %d\n", static\_cast<int>(res));

return 0;

}