Динамическая оптимизация машинного кода для RISC-V

Студент гр. 8307: **Репин Степан Александрович**
Руководитель: к. т. н., доцент **Пазников Алексей Александрович**

- **Цель:** исследование динамических оптимизаций для RISC-V и разработка динамического оптимизатора машинного кода.
- **Объект исследования:** машинный код архитектуры RISC-V
- **Предмет исследования:** динамическая оптимизация машинного кода

Задачи

1. Изучить компиляторные оптимизации, которые можно было бы применять динамически для машинного кода
2. Реализовать фреймворк, предоставляющий возможности для:
   a. динамического анализа и модификации бинарных файлов
   b. реализации произвольных динамических оптимизаций

Результат

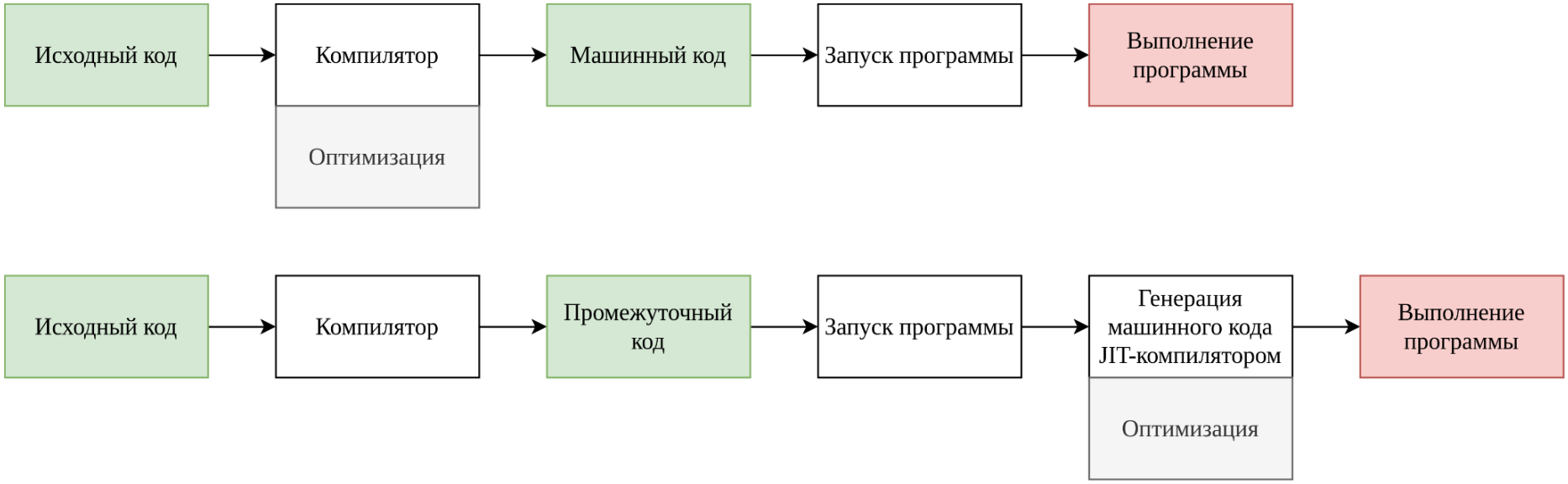
1. Библиотека на C++20, реализующая JIT-компилятор машинного кода, а также предоставляющая средства для создания оптимизаций
2. Динамическая оптимизация выравнивания циклов, подтверждающая работоспособность разработанного решения

Архитектура RISC-V

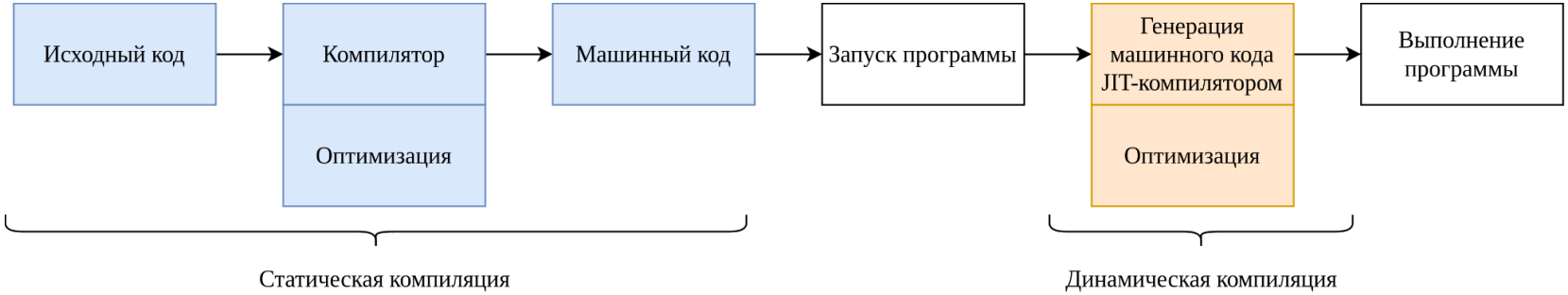
- RISC-V — это свободный и открытый набор команд типа RISC
- Основные характеристики:
  - Простота (существенно меньше по размеру, чем другие ISA)
  - Модульность (заложена возможность расширения и специализации)
    - Небольшой базовый набор и стандартные расширения
  - Стабильность
    - Добавление расширений без изменения базового набора
    - Спецификации доступны для свободного и совершенно бесплатного использования

Имя	ABI псевдоним
x0	zero
x1	ra
x2	sp
x3	gp
x4	tp
x5-x7	t0-t2
x8-x9	s0-s1
x10-x17	a0-a7
x18-x27	s2-s11
x28-x31	t3-t6
f0-f7	ft0-ft7
f8-f9	fs0-fs1
f10-f17	fa0-fa7
f18-f27	fs2-fs11
f28-f31	ft8-ft11

Статическая и динамическая компиляция



Статическая и динамическая компиляция



Какие преимущества это дает?

- Оптимизировать код под конкретное устройство пользователя
  - Статически компиляторы собирают программу под конкретный набор инструкций, заданный разработчиков, а пользователь может иметь оборудование лучше
- Применяя оптимизации можно учитывать runtime состояние программы
- Можно “дооптимизировать” программы, собранные давно и/или для которых недоступны исходники
- Во время оптимизации можно производить бенчмарки вариантов оптимизации и выбирать наилучший

Пример. Исходный код

```c
int sum_arr(int *arr, int num) {
    int i, sum = 0;
    bb_jit_begin();
    for (i = 0; i < num; i++)
        sum += arr[i];
    bb_jit_end();
    return sum;
}
```

```
sum_arr:
        addi    sp, sp, -32
        sd      ra, 24(sp)
        sd      s0, 16(sp)
        sd      s1, 8(sp)
        sd      s2, 0(sp)
        mv      s0, a1
        mv      s1, a0
        call    bb_jit_begin
        li      s2, 0
        blez    s0, .L2
.L1:
        lw      a0, 0(s1)
        addw    s2, a0, s2
        addi    s0, s0, -1
        addi    s1, s1, 4
        bnez    s0, .L1
.L2:
        call    bb_jit_end
        mv      a0, s2
        ld      ra, 24(sp)
        ld      s0, 16(sp)
        ld      s1, 8(sp)
        ld      s2, 0(sp)
        addi    sp, sp, 32
        ret
```

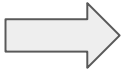
Пропускаем цикл, если num <= 0

Тело цикла

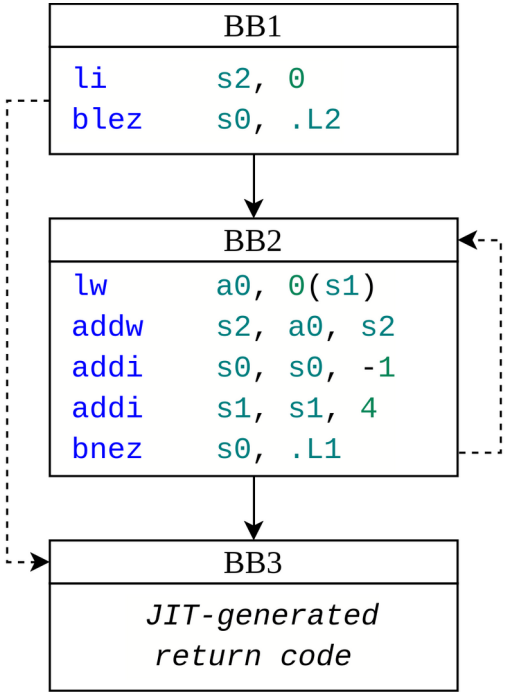
Условие цикла

Пример. Построение CFG*

```
        call    bb_jit_begin
        li      s2, 0
        blez    s0, .L2
.L1:
        lw      a0, 0(s1)
        addw    s2, a0, s2
        addi    s0, s0, -1
        addi    s1, s1, 4
        bnez    s0, .L1
.L2:
        call    bb_jit_end
```

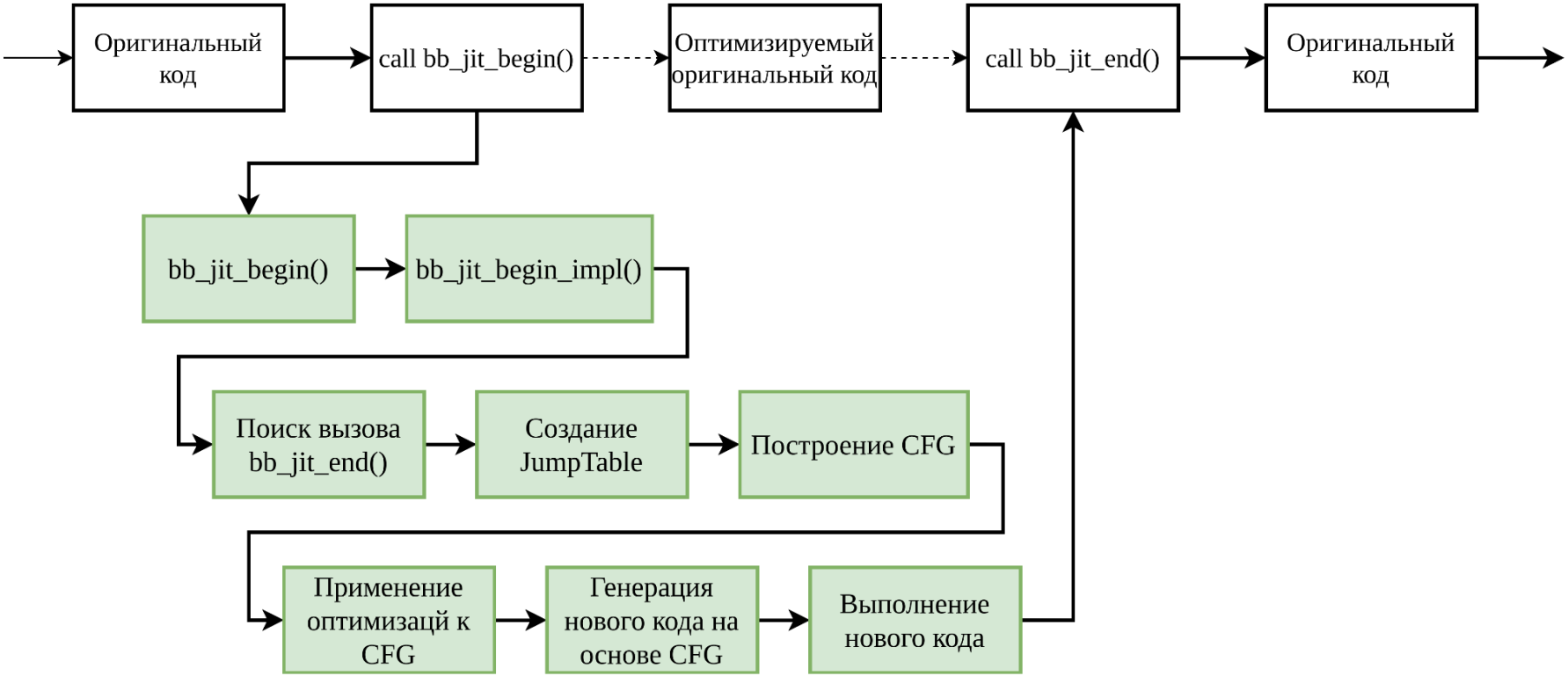


Object code
00 80 60 63 40 88 01 25 09 3b 14 7d 04 91 e0 01

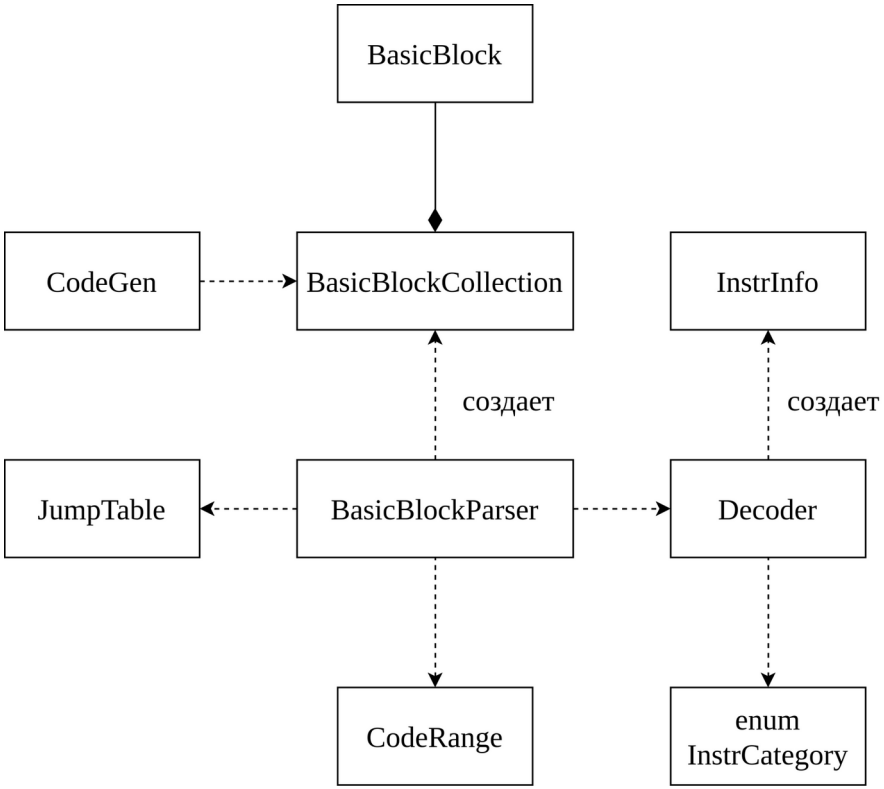


*CFG (control flow graph) – граф потока управления. Каждый узел называется базовым блоком, каждая инструкция внутри блока исполняется ровно один раз, инструкции перехода могут быть только в конце блока

Алгоритм работы оптимизатора



Архитектура оптимизатора



- Функция bb_jit_begin() — единственная написанная вручную на ассемблере
- Она вызывает bb_jit_begin_impl() и передает туда адрес следующей инструкции (B)
- bb_jit_impl() выполняет:
  - Поиск вызова bb_jit_end() — адрес окончания оптимизируемого блока (E)
  - Создание JumpTable — списка всех адресов jump-инструкций и соответствующих им меток
  - Построение CFG (BasicBlockCollection) на ее основе
  - Выполнение оптимизаций
  - Генерацию нового кода на основе CFG
  - Выполнение нового кода, который вернет управление в оригинальный код

Выравнивание исходного кода

- Вставка NOP-инструкций перед “горячими” циклами, чтобы адрес их начала был кратным определенному числу (16 или 32)
- Это позволяет более эффективно использовать I-кэш процессора – минимизирует число чтений памяти для загрузки тела

	0x...06	li s2, 0
	0x...0a	blez s0, .L2
L1:	0x...1e	lw a0, 0(s1)
	0x...22	addw s2, a0, s2
	0x...26	addi s0, s0, -1
	0x...2a	addi s1, s1, 4
	0x...2e	bnez s0, .L1
L2:	0x...32	call bb_jit_end

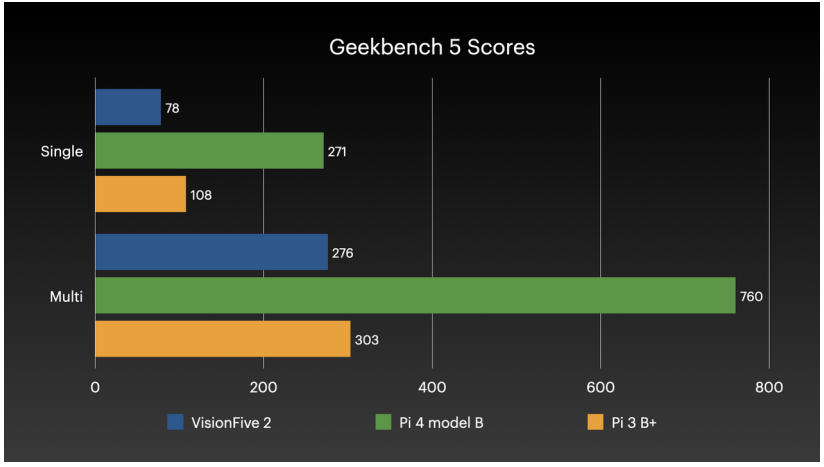
0x…1e % 32 == 30

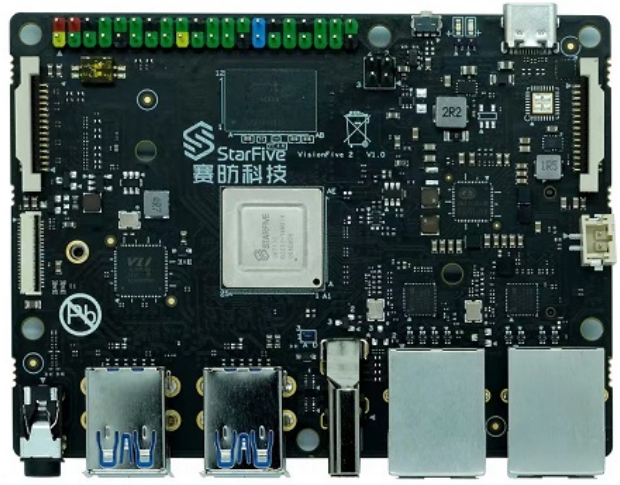
	0x...06	li s2, 0
	0x...0a	blez s0, .L2
	0x...1e	align [2]
L1:	0x...20	lw a0, 0(s1)
	0x...24	addw s2, a0, s2
	0x...28	addi s0, s0, -1
	0x...2c	addi s1, s1, 4
	0x...30	bnez s0, .L1
L2:	0x...34	call bb_jit_end

0x…20 % 32 == 0

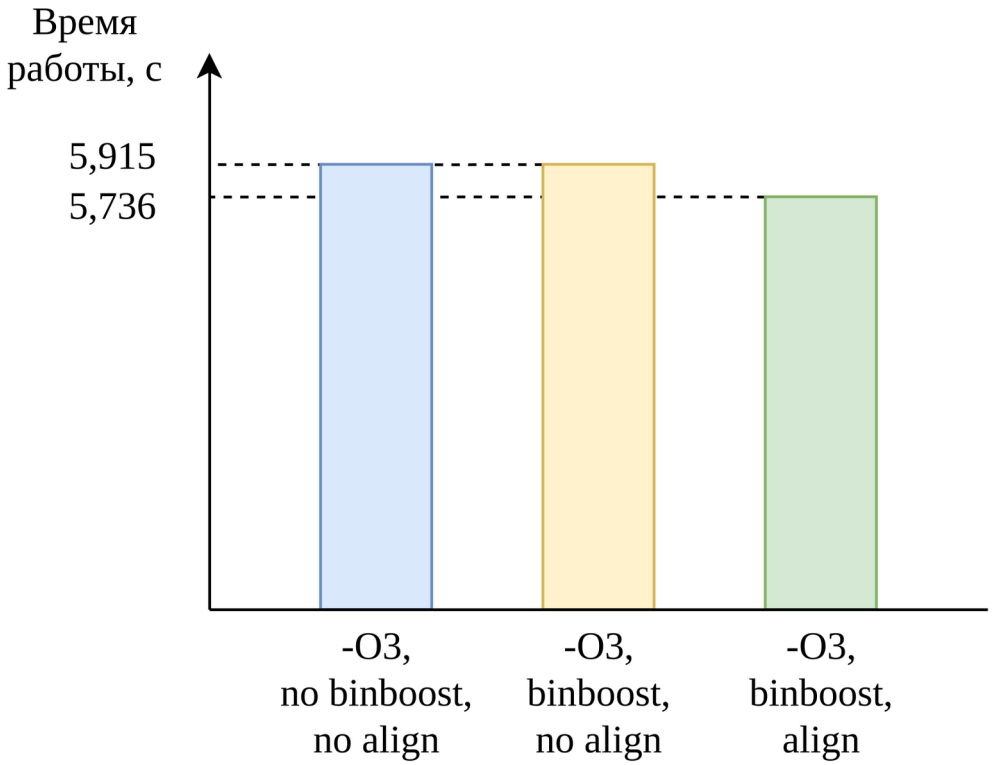
Стенд для тестирования

- Тестирование и отладка в работе выполнялась на реальном оборудовании
- Использовалась платформа StarFive VisionFive 2
- 4 ядра, 16 Гбайт ОЗУ, архитектура RV64IMAFDCZicsr_Zifencei (или RV64GC)
- Операционная система Gentoo Linux.





Измерение производительности



- Сортировка пузырьков 70000 эл.
- Измерение работы всей программы от начала и до конца
- clang 17

Тест	Ускорение
(1)	1
(2)	1
(3)	0,96

Другие динамические оптимизации

- Встраивание функций и девиртуализация
- Разворачивание циклов
- Изменение порядка следования кода (code reordering)
- Векторизация кода

Заключение

- Исследованы различные оптимизации, адаптируемые для динамического применения
- Написан фреймворк для создания и применения динамических оптимизаций
- Фреймворк протестирован с помощью оптимизации выравнивания циклов

Дальнейшее развитие: больше оптимизаций, больше архитектур, использование runtime информации

Спасибо за внимание!