Санкт-Петербургский государственный университет

Белошапкин Михаил Юрьевич

Ускорение эмулятора RISC-V, порожденного спецификацией SAIL

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
инженер исследователь Д. С. Косарев

Санкт-Петербург
2025

Оглавление

1. Введение

В современном мире существует множество различных процессорных архитектур, каждая из которых имеет свою область применения. При этом программа, собранная под одну систему команд, не будет работать на компьютере с другой архитектурой. Для решения этой проблемы существуют так называемые эмуляторы процессоров.

Эмулятор процессора – это программное или программно-аппаратное средство, которое позволяет имитировать работу настоящего физического микропроцессора. На данный момент существует множество таких инструментов, одним из них является Sail. Sail позволяет при помощи специального одноименного языка создавать описание ISA семантики инструкций, и по данному описанию генерировать соответствующие эмуляторы. Таким образом, любой инженер может при помощи этого инструмента описать свою произвольную систему команд и получить готовый симулятор.

На данный момент на Sail реализовано несколько распространенных архитектур, в том числе RISC-V. RISC-V является открытым стандартом, который разработан на базе концепции RISC, и в последние годы он набирает все большую популярность. Причной тому является его открытость и гибкость, позволяющие применять данные процессоры в различных областях, от встроенных систем до высокопроизводительных вычислений.

Sail является перспективным инструментом, поэтому увеличение его быстродействия является актуальной задачей. В рамках данной работы рассматриваются различные методы ускорения эмулятора RISC-V, порожденного спецификацией Sail.

2. Постановка задачи

Целью работы является ускорение эмулятора архитектуры RISC-V, порожденного спецификацией Sail. Для достижения данной цели были сформулированы следующие промежуточные задачи.

- Сделать обзор предметной области:
  - проекта Sail, его особенностей, влияющих на быстродействие моделей
  - существующих проектов, которые решают проблему ускорения Sail
- Выбрать один или несколько подходов к ускорению
- Произвести оценку эффективности примененных методов

3. Обзор

3.1. SAIL

Sail – это инструмент, который позволяет программно эмулировать различные процессорные архитектуры. Основой данного решения является компилятор, который принимает на вход текст на специальном формальном языке Sail, а на выходе генерирует программу на C или OCaml. Sail представляет собой функциональный язык, по синтаксису схожий с Rust. С помощью него пользователь описывает семантику микрокоманд процессора, а полученная в результате трансляции программа и является эмулятором, который в качестве аргумента принимает ELF-файл, собранный под описанную архитектуру. На данный момент на Sail реализованы некоторые наиболее распространенные архитектуры, в том числе RISC-V, MIPS, X86. Также Sail предоставляет пользователям дополнительную функциональность, например: автоматическая генерация тестов, верификация описанной модели.

Сгенерированный эмулятор запускает исполняемые файлы так, как если бы они работали на чистом железе без промежуточного слоя в виде операционной системы, при этом запускаемые бинарные файлы не могут использовать никакие библиотеки, что накладывает некоторые ограничения.

Одним из основных факторов, который негативно влияет на быстродействие сгенерированных эмуляторов, состоит в достаточно неоптимальной схеме работы с представлением данных и памятью. Дело в том, в Sail существует специальный тип данных для битовых векторов, причем этот тип параметризуется размером битового вектора. Например, вектора размером 256 и 512 типизируется по-разному, в случае несоответствия этих типов в тексте программы sail выдаст ошибку компиляции. В компилятор внедрен SMT-решатель, который во время трансляции исходного кода определяет возможную длину каждого битового вектора. Для тех векторов, чей размер составляет не более 64 бит, используется представление uint64_t. Для представления длинных

векторов и для осуществления операций над ними используется библиотека GMP. Проблема состоит в том, очень часто возникает ситуация, при которой sail генерирует неэффективное представление данных, то есть под переменную, для хранения которой достаточно uint64_t, создается GMP-вектор. Происходит это потому, что SMT-решатель не всегда может заранее гарантировать соответствующий размер переменной, это и является причиной избыточных аллокаций памяти. Данная особенность сильно замедляет работу сгенерированных эмуляторов.

3.2. Примененные ранее подходы к ускорению

На данный момент существует несколько работ и проектов, которые решают проблему ускорения Sail. Ниже приведены некоторые из них.

«Оптимизация эмулятора RISC-V, порожденного по спецификации Sail», выпускная квалификационная работа Ильи Балашева. В ней автор описывает способ оптимизации, связанный с подменой стандартных системных функций (например malloc) на их альтернативные реализации. По результатам эксперимента было выявлено, что некоторые альтернативные алгоритмы аллокаций способны существенно улучшить производительность эмулятора (до 50%). Помимо аллокаций, была оптимизирована работа со строками, по результатам тестирования это дало прирост производительности до четырех раз. Также в работе приведены тестовый набор и инструменты профилирования и анализа, которые использовались в исследовании.

«Pydrofoil» – проект с открытым исходным кодом, который призван улучшить быстродействие сгенерированных эмуляторов. Суть решения состоит в том, что промежуточное представление программы на Sail, сгенерированное компилятором, транслируется в RPython, а не в программу на языке C. Такой подход позволяет решить важную проблему: JIT-компилятор определяет размер битовых векторов прямо во время выполнения программы, а следовательно, может выбрать более эффективный способ их представления, таким образом избегая избыточных динамических аллокаций.

3.3. Используемые инструменты

Стоит кратко описать набор инструментов, с помощью которых производилось исследование.

Perf – утилита для анализа производительности программ для ОС Linux. Данный инструмент позволяет производить статистическое профилирование: замерять производительность как отдельной программы, так и ядра системы. В ходе исследования данный инструмент использовался для анализа стека вызовов.

GDB – консольный отладчик, работающий на Unix-подобных операционных системах. С помощью данного инструмента можно исполнять программу пошагово, следить за состоянием переменных и т. д. Данная программа использовалась для изучения внутреннего устройства сгенерированного эмулятора.

time – стандартная Linux-утилита для замера времени работы программ.

4. Исследование

Существуют различные подходы к ускорению и оптимизации программ, однако в случае с эмулятором RISC-V наиболее подходящими являются низкоуровневые и алгоритмические. В рамках первого семестра было принято решение рассмотреть оптимизацию с использованием так называемых интринзик.

4.1. Интринзики

Интринзики – это специальные встроенные функции, поддерживаемые на уровне компилятора. При трансляции исходного текста в месте вызова интринзик компилятор порождает особый код, характерный именно для конкретной функции, иными словами, будет сгенерирована последовательность инструкций, специфичных для определенной архитектуры. Данные встроенные функции используются тогда, когда необходимо ускорить вычисления. Таким образом, интринзики ухудшают переносимость программы, но в то же время повышают ее быстродействие. Компилтор GCC, с помощью которого собирается эмулятор, также поддерживает встроенные функции, полный список которых представлен в документации[1].

Суть оптимизации с применением интринзик заключается в том, что исполнение инструкции будет происходить не напрямую, а с помощью встроенной в компилятор функции. Например, в системе команд эмулируемого процессора есть инструкция, которая по своей семантике полностью совпадает с какой-либо инструкцией физического процессора, на котором запускается программа-симулятор. Тогда логичным решением было бы использовать для вычисления уже существующую команду, а не считать ее программно.

Для проверки эффективности данного метода было принято решение рассмотреть некоторую RISC-V инструкцию, вычисление которой можно было бы заменить одной интринзикой. При этом логика испол-

[1]https://gcc.gnu.org/onlinedocs/gcc/x86-Built- in-Functions.html

нения выбранного оператора должна быть достаточно сложна, чтобы применение встроенной функции могло заметно уменьшить алгоритмическую трудоемкость его вычисления.

4.1.1. Внедрение

В качестве примера такой инструкции был выбран CPOP [3] – данный оператор считает количество единиц в машинном слове. При помощи отладчика исходном тексте эмулятора было найдено место, которое отвечает за его вычисление. На листинге 1 представлен цикл, в котором и происходит вычисление:

Листинг 1: Вычисление CPOP

```
for_start_8849: ;
  {
    if ((zuz32491 > zgsz39259)) goto for_end_8850;
    bool zgaz37141;
    {
      fbits zgaz37140;
      zgaz37140 = (UINT64_C(1) & (zuz32489 >> zuz32491));
      zgaz37141 = eq_bit(zgaz37140, UINT64_C(1));
    }
    if (zgaz37141) {
      {
        sail_int zgsz316231;
        CREATE(sail_int)(&zgsz316231);
        CONVERT_OF(sail_int, mach_int)(&zgsz316231, INT64_C(1));
        add_int(&zuz32490, zuz32490, zgsz316231);
        KILL(sail_int)(&zgsz316231);
      }
      zgsz39261 = UNIT;
    } else {  zgsz39261 = UNIT;  }
    zuz32491 = (zuz32491 + zgsz39260);
    goto for_start_8849;
  }
for_end_8850: ;
```

Здесь в качестве счетчика цикла выступает переменная zuz32491, а количество единиц – результат вычисления CPOP, – хранится в zuz32490. Более того, в случае увеличения счетчика единиц внутри функций CREATE и KILL будут вызваны malloc и free, при том вызва-

ны они будут не более 32 и 64 раз для 32- и 64-битных эмуляторов соответственно. В качестве оптимизации предлагается заменить данный цикл на одну встроенную функцию, которая выполняет ту же задачу.

Компилятор GCC имеет встроенную функцию __builtin_popcount, которая принимает целое число и возвращает количество единиц в двоичном представлении переданного числа. Представленный на листинге цикл и был заменен на эту интринзику.

4.1.2. Описание тестового стенда

Запуск эмулятора осуществлялся на компьютере со следующими характеристиками:

- Процессор Intel i5-7300U
- 16 GB RAM
- Операционная система Ubuntu 22.04 LTS

Также использовался компилятор CC с опциями -O2 и -flto=auto.

4.1.3. Эксперимент

Для оценки эффективности примененной оптимизации необходимо произвести замеры времени на некотором наборе тестов. Для этого можно подобрать такие программы на языке C, при компиляции которых будет сгенерировано достаточно большое количество необходимых инструкций. Но можно пойти другим путем: бинарные файлы с нужным набором инструкций можно генерировать автоматически с помощью Python-скрипта. Так, были получены несколько исполняемых файлов, которые содержат от 10000 до 50000 инструкций CPOP.

На тестовых бинарных файлах были запущены два варианта эмулятора: с оптимизацией и без нее. Время исполнения эмулятора измерялось при помощи утилиты /usr/bin/time. Замеры показали, что внедренная оптимизация практически никак не повлияла на время исполнения, конкретные значения представлены в таблице 1 (представлены усредненные значения за 10 замеров). При прогоне тестов при

помощи утилиты perf было выяснено, что на функцию, вычисляющую инструкции, тратится около одного процента процессорного времени. При этом статистика показывает, что наибольшая часть времени работы программы приходится на системные вызовы malloc и free. Следует заметить, что внедрение встроенных функций уменьшает количество этих системных вызовов, однако сложно оценить заранее, насколько реально эта оптимизация повлияет на быстродействие эмулятора.

	С оптимизацией	Без оптимизации
10000	3.12	3.15
20000	5.96	6.01
30000	8.72	8.70
40000	11.03	11.05
50000	14.34	14.30

Таблица 1: Результаты замеров

4.2. Векторное расширение

Логичным шагом было бы применить встроенные функции в тех инструкциях, которые опрерируют данными довольно большой длины, а также производят над этими данными большое количество операций. Очевидно, что такому условию удовлетворяют векторные инструкции. В архитектуре RISC-V данный тип инструкций реализован в виде векторного расширния, которое называется RVV. Отличительными чертами RVV являются: конфигурируемый вендором размер векторных регистров, а также полиморфизм, иными словами, одна и та же инструкция может производить операции над векторами разных типов.

В современных процессорах Intel реализовано векторное расширение AVX2 с 256-битными регистрами. Большинство компиляторов поддерживают интринсики для удобного использования такого типа инструкий, полный список представлен в документации[2].

[2]https://www.laruence.com/sse/

На первом этаме необходимо выбрать инструкцию, семантика которой эквивалентна для RVV и AVX2, и которая при этом совершает большое количество операций с длинными векторами. Путем анализа исходной RISC-V модели и порожденного кода было выяснено, что этим условиям удовлетворяет инструкция vsadd.vv. Для двух входных векторов она производит поэлементное "сложение с насыщением" для знаковых целых чисел. Эта операция представляет собой обычное сложение, но только в случае выхода за границу диапазона возможных значений в результат записывается крайнее значение.

Как было упомянуто ранее, инструкции RVV являются полиморфными. Для того, чтобы указать нужную конфигурацию векторных регистров, в RVV существует оператор vsetvli, который устанавливает три основных параметра:

- количество элементов
- SEW — размер обрабатываемых элементов в битах
- LMUL — мультипликатор длины векторного регистра, параметр для объединения векторных регистров в группы.

Таким образом, надо подобрать такие параметры, чтобы семантики инструкций vsadd.vv и какой-либо другой AVX2-инструкции совпадали. Было решено рассмотреть следующую конфигурацию: размер одного элемента равен 8 битам, мультипликатор длины равен единице, количество элементов равно 32. Такая конфигурация соответствует AVX2-функции _mm256_adds_epi8. Логика вычисления vsadd.vv была изменена таким образом, чтобы данная интринсика использовалась вместо вычисления результата напрямую.

4.2.1. Эксперимент

Для тестирования эффективности данного метода был применен тот же подход, что и в случае с тестированием CPOP. С помощью скрипта были сгенерированы файлы с разным количеством инструк-

ций vsadd.vv (от 5000 до 25000). Сами вектора конфигурировались при помощи инструкции vsetvli:

```
vsetvli t0, x0, e8, m1, ta, ma
```

Сборка эмулятора производилась при помощи компилятора CC с флагами -mavx512f и -O2. Замер времени осуществлялся утилитой time. В качестве результата было взято среднее время за 10 замеров.

	С оптимизацией	Без оптимизации
5000	0.87	1.12
10000	1.71	2.09
15000	2.49	2.95
20000	3.07	3.55
25000	4.17	4.93

Таблица 2: Результаты замеров

Как видно из замеров (Таблица 2), время выполнения эмулятора сократилось среднем примерно на 20 процентов. После получения данных результатов было принято решение внедрить встроенные AVX2-функции в проект и протестировать прирост быстродействия.

4.2.2. Внедрение встроенных функций в Sail

Для того, чтобы внедрить AVX2-интринсики проект, необходимо определить то, как они будут использоваться. В текущей реализации было принято решение расширить множество встроенных функций языка sail, иными словами, добавить к стандартной библиотеке еще один подключаемый модуль.

В проекте стандартная библиотека реализована следующим образом: в файлах с расширением *.sail прописаны сигнатуры функций, а их имплементация написана на языке C и находится в специальных

файлах, которые линкуются с порожденным кодом эмулятора во время его сборки. Для добавления новых функций был создан еще один модуль: в него входят файл с сигнатурами, а также файл с реализацией.

Сигнатура встроенной функции выглядит следующим образом (листинг 2):

Листинг 2: Сигнатура встроенной функции

```
val avx_mm256_adds_epi8 =
  pure "avx_mm256_adds_epi8" : (bits(256), bits(256)) → bits(256)
```

Данное выражение означает, что функция принимает два битовых вектора длиной 256 бит, возвращает битовый вектор той же длины. В программе на sail она может быть использована следующим образом (листинг 3):

Листинг 3: Пример использования

```
let bv1 : bits(256) =
  0x0101_0101_0202_0303_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0002;
let bv2 : bits(256) =
  0x0202_0002_0303_0404_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0003;
let bv3 : bits(256)= avx_mm256_adds_epi8(bv1, bv2);
```

Вызов данной функции в программе на sail в порожденном коде преобразуется в вызов одноименной функции, определение которой должно находится в подключаемом модуле avx2.c. При трансляции компилятор преобразовывает переменную типа bits(256) в структуру данных lbits (листинг 5). Эта структура содержит два поля: количество битов и указатель на GMP-структуру mpz_t. Именно так представляются битовые вектора, длина которых больше, чем 64. Встроанная функция должна иметь три параметра – указатель на выходной битовый вектор, а также два вектора, над которыми будет произведена соответствующая операция (Листинг 4). В логике самой функции происходит извлечение данных из входных GMP-векторов, для этих данных применяется интринсика _mm256_adds_epi8, полученный результат упаковывается в выходной параметр.

Листинг 4: Объявление встроенной функции

```
void avx_mm256_adds_epi8(lbits *out, lbits vector1, lbits vector2);
```

Листинг 5: Структура lbits

```
typedef struct {
  mp_bitcnt_t len;
  mpz_t *bits;
} lbits;
```

Возможность применения AVX2-интринсик зависит от того, на какой машине производится сборка и запуск скомпилированной модели. Для того, чтобы пользователь мог использовать встроенные AVX2 функции, было принято решение добавить в компилятор еще один ключ -avx2, который подключает заголовочный файл с соответствующими функциями. Команда компиляции RISC-V эмулятора теперь будет выглядеть следующим образом:

```
$ sail --require-version 0.18 -avx2 [ОПЦИИ] <файлы *.sail>
```

После чего сгенерированная программа может быть собрана компилятором CC с использованием флага -mavx512f.

4.2.3. Внедрение встроенных функций в модель

Выбранный способ ускорения подразумевает, что разработчик модели будет сам указывать, в каком месте вызвать встроенную функцию. В случае с RISC-V семантика векторных инструкций может меняться в зависимости от внешних параметров SEW (размер одного элемента), LMUL (мультипликатор длины вектора), vlen (размер одного регистра), vl (количество используемых элементов), а также маски. Таким образом, напрямую заменить логику инструкции в исходной модели не получится, но ускорить ее выполнение можно в том случае, если конфигурация RVV-векторов подходит соответствующей AVX2-функции. В коде модели RISC-V можно добавить проверку (Листинг 6):

Листинг 6: Внедрение встроенных функций в модель

```
if (SEW == 8 & LMUL_pow == 0 & num_elem == 32
            & funct6 == VV_VSADD & is_true_mask(mask)) then {
  let vs2_val: bits(256) = read_vreg_as_bits(vs2);
  let vs1_val: bits(256) = read_vreg_as_bits(vs1);
  let r: bits(256) = avx_mm256_adds_epi8(vs1_val, vs2_val);
  write_single_vreg_bits256(vd, r);
  vstart = zeros();
  RETIRE_SUCCESS
}
else {
  ...
}
```

После внедрения этой оптимизации были произведены замеры времени выполнения на том же наборе тестов, что использовался ранее. В результате время работы эмулятора сократилось примерно на 50 %. Больший прирост быстродействия обусловлен тем, что в код модели были добавлены две новые функции read_vreg_as_bits и write_single_vreg_bits256, которые позволяют выполнить операции чтения и записи в регистры быстрее ввиду того, что размер данных заранее известен. Также был произведен замер среднего времени выполнения оптимизированных инструкций, оно сократилось на 80 %.

4.3. Оптимизация аллокаций на уровне компилятора

Как было показано ранее, использование встроенных функций может существенно ускорить быстродействие эмулятора на некотором множестве синтетических тестов. Однако необходимо учитывать, что использовать данный подход можно в ограниченном количестве случаев, более того, интринсики требуют внесения изменения в код модели. В связи с этим имеет смысл рассмотреть методы сокращения числа аллокаций, которые бы влияли на всю модель и которые бы не требовали изменения ее логики.

4.3.1. Существующее решение

При анализе сгененированного кода, а также при изучении логики компилятора было обнаружено, что в текущей реализации Sail уже существует способ сокращения числа аллокаций. Идея состоит в переиспользовании памяти под GMP-структуры: для каждой нерекурсивной функции f генерируется процедуры startup_f и finish_f, где происходит выделение и очистка памяти под GMP-переменные, сами же переменные объявляются вне функции. Функция startup_f автоматически вызывается во время инициализации модели, функция finish_f вызывается после того, как эмулятор завершил исполнние инструкций. Таким образом выделение памяти происходит только один раз, далее она пересипользуется при каждом вызове функции f. Данное решение позволяет существенно сократить количество аллокаций, а следовательно и улучшить быстродействие. Проблема состоит в том, что данная оптимизация работает только в нерекурсивных функциях и только для единичных GMP-векторов, и не распрастроняется на GMP-вектора внутри структур.

4.3.2. Тип данных vector

В языке Sail существует тип данных vector, котрый представляет собой список фиксированной длины, хранящий в себе значения определенного типа. В описании логики векторных инструкций этот тип используется для удобного представления векторного регистра, потому как позволяет удобно осуществлять поэлементные операции. Для каждого типа vector('n) генерируется соответствующая струкрура, а также специальные функции для работы с этой структурой. Пример такой структуры представлен на Листинге 7. Она хранит в себе массив из GMP-векторов, что соответствует типу vector(bits) в исходном коде. При каждой инициализации вектора необходимо выделять память под массив и под каждое GMP-число, что при достаточно большой длине сильно увеличивает число аллокаций. Особенно это может повлиять на быстродействие в том случае, когда в логике модели необходимо

оперировать несколькими списками из битовых векторов, например, в описании семантики векторного расширения. Для оптимизации работы с памятью имеет смысл воспользоваться подходом, который описан в предыдущем пункте.

Листинг 7: Представление типа vector(bits)

```
struct zz5vecz8z5bvz9 {
  size_t len;
  lbits *data;
};
typedef struct zz5vecz8z5bvz9 zz5vecz8z5bvz9;

static void CREATE(zz5vecz8z5bvz9)(zz5vecz8z5bvz9 *rop);

static void KILL(zz5vecz8z5bvz9)(zz5vecz8z5bvz9 *rop);

static void vector_init_zz5vecz8z5bvz9(zz5vecz8z5bvz9 *vec, sail_int n, lbits elem);

static void RECREATE(zz5vecz8z5bvz9)(zz5vecz8z5bvz9 *rop);

static void COPY(zz5vecz8z5bvz9)(zz5vecz8z5bvz9 *rop, zz5vecz8z5bvz9 op);

...
```

4.3.3. Реализация

При анализе исходного кода компилятора было обнаружено, что существующая оптимизация GMP-векторов реализована в файле c_backend.ml. Данный файл отвечает за генерацию кода на языке C из специального ANF-представления абстрактного синтаксического дерева (типизированное AST заранее преобразуется в ANF для удобства кодогенерации). Данное представление AST содержит всю информацию, необходимую для анализа. Алгоритм оптимизации можно разделить на следующие этапы: обнаружение в AST переменных, содержащих список из битовых векторов, удаление лишних операций создания/удаления, вынесение переменных во внешний контекст, добавления операций создания/удаления в функции startup и finish. При этом, чтобы избежать ненужных аллокаций, выделение памяти под вектора должно происходить не при инициализации модели в функции startup,

а при первом использовании вектора. Для этого необходимо изменить логику работы функций (Листинг 7), отвечающих за выделение памяти вектору, а именно добавить соответствующие проверки.

Как было сказано ранее, оптимизация с переиспользованием памяти работает только для нерекурсивных функций. Следует пояснить, почему данное ограничение является очень существенным. Дело в том, что в языке Sail для удобства описания логики инструкций существует специальный тип функий, которые обозначаются ключевым словом clause (пример приведен на Листинге 8). Данная особенность позвлоляет удобно группировать код в отдельно для каждой инструкции. При декодировании инструкции получается специальная структура, которая передается в функцию execute, и нужный обработчик выбирается автоматически. При компиляции данные функции соединяются в единую процедуру, на вход принимающая union-структуру, которая содежит информацию о типе инструкции, ее операндах, далее с помощью конструкций if-else находится нужный обработчик. Каждый такой обработчик может вызывать внутри себя другой обработчик, как следствие, скомпилированная функция будет содержать в себе рекурсивный вызов, а значит текущая оптимизация работать не будет. Решением данной проблемы является то, что можно рассматривать только те ветви исполнения, которые не содержат в себе рекурсию.

Листинг 8: Обработчик инструкции ADDI

```
union clause ast = ITYPE : (bits(12), regbits, regbits, iop)

function clause decode imm : bits(12) @ rs1 : regbits @ 0b000 @ rd : regbits @ 0b0010011
  = Some(ITYPE(imm, rs1, rd, RISCV_ADDI))

function clause execute (ITYPE (imm, rs1, rd, RISCV_ADDI)) = {
  let rs1_val = X(rs1);
  let imm_ext : xlenbits = EXTS(imm);
  let result = rs1_val + imm_ext;
  X(rd) = result
}
```

4.3.4. Замеры

После внеднения оптимизации в компилятор были произведены замеры времени исполнения на тех же тестах, что и использовались для встроенных функций. После замеров времени выяснилось, что время выполнения сократилось на 10 процентов (Таблица 3). Следует отметить, что оптимизация, направленная на переиспользование памяти, является более перспективной, так как влияет на всю модель в целом а также не требует вмешательства разработчика модели.

	С оптимизацией	Без оптимизации
5000	0.98	1.12
10000	1.82	2.09
15000	2.53	2.95
20000	3.21	3.55
25000	4.46	4.93

Таблица 3: Результаты замеров

5. Заключение

Были получены следующие результаты

- Сделан обзор предметной области, описаны особенности проекта, которые влияют на быстродействие моделей
- Приведены примеры подходов, которые решают проблему ускорения Sail
- Внедрены следующие методы ускорения
  - Использование встроенных функций для векторного расширения
  - Оптимизация памяти для структуры данных vector
- Произведены замеры времени выполнения синтетичесих тестов для каждого подхода
  - Встроенные функций – снижение времени выполнения на 20 %
  - Оптимизация аллокаций – снижение времени выполнения на 10 %

Список литературы

[1] The Sail instruction-set semantics specification language : Rep. / Technical report published by Cambridge University ; executor: Gray Kathryn E, Sewell Peter, Pulte Christopher et al. : 2017

[2] The RISC-V instruction set manual / Waterman Andrew, Lee Yun- sup, Patterson David, Asanovic Krste, level Isa Volume I User, Wa- terman Andrew, Lee Yunsup, and Patterson David // Volume I: User- Level ISA', version.— 2014.—Vol. 2

[3] RISC-V Bitmanip Extension / Claire Wolf // Version 0.94-draft

[4] GCC documentation / https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html (Дата: 10.02.2024)

[5] De Melo Arnaldo Carvalho. The new Linux perf tools // Slides from Linux Kongress.— 2010.—Vol. 18.—P. 1–42.