

РЕФЕРАТ

Расчетно-пояснительная записка 51 с., 9 рис., 3 табл., X ист., X прил.

КЛЮЧЕВЫЕ СЛОВА

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	8
1 Аналитическая часть	9
1.1 Постановка задачи?	9
1.2 Методы трансляции	9
1.2.1 Интерпретация	9
1.2.2 Рекомпиляция	9
1.2.3 Эмуляция высокого уровня	10
1.2.4 Сравнение методов трансляции	10
1.3 Динамические трансляторы	10
1.3.1 QEMU	10
1.3.2 box64	11
1.3.3 FEX	12
1.3.4 Сравнение динамических трансляторов	14
1.4 Существующие оптимизации трансляции	16
1.4.1 Использование блоков трансляции	16
1.4.2 Поддержка специфичных для архитектуры библиотек	18
1.4.3 Оптимизации кода	19
1.4.4 Распространение констант	20
1.4.5 Устранение мертвого кода	20
1.4.6 Устранение загрузок контекста	20
1.4.7 Устранение хранения	20
1.4.8 Сжатие инструкций	21
1.4.9 Устранение временных регистров	21
1.4.10 Анализ живости	21
1.4.11 Сравнение оптимизаций трансляции	22
1.5 Поддержка самомодифицирующегося кода	23
1.6 Поддержка TSO	23
1.6.1 Регистр RBP	26
1.7 Оценка скорости работы трансляторов	27
1.8 Вывод	28

2	Конструкторская часть	29
2.1	Архитектура программного обеспечения	29
2.2	Используемые структуры данных	29
2.3	Алгоритм оптимизации динамической трансляции доступа к памяти	29
2.4	Алгоритм построения графа выполнения блоков	31
2.5	Алгоритм распространения состояния регистра	31
2.6	Вывод	32
3	Технологическая часть	33
3.1	Выбор средств разработки	33
3.1.1	Выбор языка программирования	33
3.1.2	Сборка программного обеспечения	33
3.2	Требования к вычислительной системе	34
3.3	Структура программного обеспечения	35
3.4	Распространение программного обеспечения	35
3.5	Дополнительные утилиты	35
3.6	Вывод	37
4	Исследовательская часть	38
4.1	Описание используемых данных	38
4.2	Результаты исследования	39
4.3	Тестирование на надежность	43
4.4	Вывод	43
	ЗАКЛЮЧЕНИЕ	44
	СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	45
	ПРИЛОЖЕНИЕ А	47

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

Транслятор — программа или техническое средство, выполняющие трансляцию программы. [1]

Трансляция программы — преобразование программы, представленной на одном языке программирования, в программу на другом языке и в определенном смысле равносильную первой. [1]

Статическая трансляция (Ahead-of-time (AOT)) — трансляция проводящаяся до начала выполнения программы.

Динамическая трансляция (Just-in-time (JIT)) — трансляция выполняемая непосредственно во время выполнения программы.

Интерпретация — трансляция и выполнения каждого предложения исходного языка машинной программы перед трансляцией и исполнением следующего предложения. [2]

Блок трансляции — непрерывная последовательность инструкций программы выделенная для трансляции.

Промежуточное представление — это специальный код используемый внутри компилятора или виртуальной машины для представления исходного кода. Предназначен для дальнейшей обработки, такой как оптимизация и трансляция в машинный код.

Статическое единственное присваивание (СЕП) — промежуточное представление в котором каждой переменной значение присваивается один раз, переменные исходной программы разбиваются на версии, обычно с помощью добавления суффикса, таким образом, что каждое присваивание осуществляется уникальной версии переменной.

Эмуляция системы — это совокупность логических и технических средств и ресурсов, направленных на полную имитацию технического устройства выбранной пользователем системы для максимально точного воспроизведения всех процессов, происходящих внутри эмулируемой системы.

Эмуляция режима пользователя — режим эмуляции поддерживающий запуск пользовательских приложений, не эмулирует полноценную систему, а использует ядро запускающей системы для выполнения необходимых системных задач.

RISC — это вычислительная машина с упрощенной системой команд, которая обеспечивает увеличение скорости декодирования команд. [3]

Пролог — машинный код в начале функции (процедуры, подпрограммы), выполняющий предваряющие действия по подготовке стека потока и регистров с целью их дальнейшего использования в теле функции. [2]

Эпилог — машинный код в конце функции (процедуры, подпрограммы), восстанавливающий резервирование пространства стека до начального состояния; восстанавливающий регистры до состояния, предшествовавшего вызову функции и производящий возврат управления из функции. [2]

ВВЕДЕНИЕ

В 2019 году процессоры архитектуры ARM составляли 34% от рынка процессоров, при этом занимая 90% рынка мобильных процессоров [4]. С появлением процессоров M1 компании Apple большое число людей стали пользоваться компьютерами на основе архитектуры ARM в качестве персональных компьютеров. (чето написать типа им нужно использовать проприетарный софт который нельзя самому для себя пересобрать)

Программы собранные под архитектуру x86 не работают на таких компьютерах, необходим или статический транслятор, такой как Rosetta 2, или виртуальная машина поддерживающая необходимую архитектуру. Rosetta 2 не транслирует программы не предназначенные для macOS, для запуска программ созданных для Windows или Linux нужно использовать виртуальную машину. Еще одно ограничение статической трансляции — наличие самомодифицирующегося кода и динамических библиотек, таким образом использование только статической трансляции не запустит любую программу. [5]

Использование трансляции по своей природе несет дополнительные издержки при выполнении программы, как в памяти, так и в скорости выполнения. Цель этой работы — исследование причин дополнительных издержек и попытка нахождения пути устранения этих издержек.

Для достижения поставленной цели необходимо решить следующие задачи:

- бла;
- бла;
- бла;
- бла;
- бла.

1 Аналитическая часть

Блоки трансляции до того как описал

В данном разделе производится анализ предметной области, приводятся методы трансляции, рассматриваются существующие оптимизации трансляции и рассматриваются существующие проблемы.

1.1 Постановка задачи?

Никакой из существующих трансляторов не достигает стопроцентной скорости выполнения программ, хотя теоретически такая скорость достижима.

а че еще сказать.....

Было бы неплохо ее достигнуть.

еще написать про ир пассы

write

1.2 Методы трансляции

write

Надо написать почему он тупит... И все методы типа тупят почему.

1.2.1 Интерпретация

Под интерпретацией понимается реализация каждой команды процессора в виде кода на языке высокого уровня. Такой подход обеспечивает высокую точность исполнения, так как обычно описывает все, даже самые малые, изменения в состоянии процессора. Из рассматриваемых методов имеет самую маленькую скорость выполнения, так как исполняющему код процессору приходится постоянно разбирать транслируемый код и выполнять его при помощи вызовов функций.

1.2.2 Рекомпиляция

Под рекомпиляцией понимается преобразование машинного кода одной архитектуры в другую. Таким образом после рекомпиляции процессор выполняет родной для себя код без необходимости вызова функций. Не все машинные инструкции имеют точные аналоги на другой архитектуре, так что необходимо

хранить дополнительные данные, например ARM не рассчитывает все флаги архитектуры x86 и их необходимо сохранять отдельно. По скорости рекомпиляция быстрее интерпретации.

1.2.3 Эмуляция высокого уровня

Под эмуляцией высокого уровня понимается подмена каких-либо высокоуровневых функций на их эквиваленты для архитектуры хоста. Например, при эмуляции приложения использующего библиотеку `libc` можно заменить все вызовы функций этой библиотеки на вызовы функций библиотеки `libc` хоста, видимый результат функций одинаков, при этом не нужно транслировать или рекомпилировать эти функции. Данный метод не подходит для эмуляции всей системы, однако может сильно повысить производительность если таких вызовов много.

1.2.4 Сравнение методов трансляции

Эмуляция высокого уровня сама по себе не может обеспечить эмуляцию всей системы, однако в связке с рекомпиляцией или интерпретацией повышает производительность последних. Интерпретация используется в случаях когда нужно точно исполнять команды процессора, рекомпиляция также может достигнуть такую точность, но для этого нужно проделать намного больше работы.

1.3 Динамические трансляторы

Как уже было упомянуто выше для запуска приложений собранных под иную платформу нужно использовать трансляцию. В данном разделе рассматриваются поддерживающие ARM динамические трансляторы QEMU, `box64` и `FEX`. Достоинство динамических трансляторов — скорость запуска программы, так как трансляция происходит во время выполнения, нет необходимости ждать окончания, например, рекомпиляции.

1.3.1 QEMU

QEMU поддерживает эмуляцию как пользовательского режима, так и эмуляцию всей системы. QEMU транслирует машинные инструкции при помощи

промежуточного представления, в качестве промежуточного представления используются «TCG ops», по организации похожие на инструкции архитектур RISC, эти операции сначала оптимизируются, а затем выполняются на хосте, таким образом проще портировать TCG на различные платформы. [7]

В QEMU реализованы инструкции x86, x86-64, x87, MMX и MMXEXT.

Пример операции TCG представлен на листинге 1:

Листинг 1: Пример операции TCG

```
1 add_i32 t0, t1, t2 (t0 <- t1 + t2)
```

1.3.2 box64

box64 — эмулятор пользовательского режима, в отличие от QEMU box64 не может эмулировать полную систему, однако может запускать linux-приложения собранные под x86_64. box64 проводит динамическую трансляцию без использования промежуточного представления, таким образом при динамической рекомпиляции поддерживаемые инструкции, при помощи макросов C, транслируются в инструкции ARM. box64 также включает в себя эмулятор, написанный на C который работает как интерпретатор, используемый при запуске на архитектуре отличной от ARM. [?]

В box64 реализованы инструкции x86, x86-64, x87 и, частично, MMX, MMXEXT, SSE, SSE2, SSE3.

Каждый блок транслируется в 4 прохода:

- Первый проход считает все количество транслируемых x86 инструкций. Для каждой инструкции выделяется память.
- На втором проходе обрабатываются все инструкции ветвления, проверяется где находится адрес: в этом же блоке или в каком-либо другом. В случае если адрес находится в другом блоке его необходимо связать с текущим. Также рассчитываются флаги процессора, не нуж-

ные флаги предлагается не рассчитывать, так как инструкция может использовать/выставлять не все флаги.

- На третьем проходе считается количество необходимых ARM инструкций в блоке.
- На четвертом проходе генерируются необходимые инструкции.

После генерации блока он записывается в таблицу сгенерированных блоков, в которой также содержится отображение эмулированных адресов в физические. [6]

1.3.3 FEX

FEX, как и box64, является эмулятором пользовательского режима, в отличие от box64 FEX используется промежуточное представление вида СЕП. FEX поддерживает не оптимизированную трансляцию в x86 и оптимизированную трансляцию в ARM. В FEX реализованы инструкции x86, x86-64, x87, MMX, SSE1, SSE2, SSE3, SSSE3 и BMI.

Трансляция происходит в 4 этапа:

- Frontend: Декодирование инструкций, при декодировании также определяются границы блоков трансляции и функций.
- OpDispatcher: Перевод инструкций во внутренние (SSA IR, IR.json) инструкции.
- IR/Passes: Оптимизация внутренних инструкций.
- JIT: Трансляция внутренних инструкций в машинные инструкции платформы хоста.

x86 инструкции декодируются в более простые для обработки, например инструкция add имеет большое количество опкодов, хотя по сути там меняются размеры и типы операндов, а не сама операция. Тогда операции add, представленные на листинге 2, попадают в одну категорию.

Листинг 2: Пример декодирования инструкций в FEX

```
1 00 C0: add al, al
2 04 01: add al, 0x1
```

Использование СЕП облегчает оптимизацию кода, например, при устранении бессмысленных присвоений, которые показаны в листинге 3:

Листинг 3: Пример лишнего присваивания

```
1 y := 1
2 y := 2
3 x := y
```

Не сложно понять, что первое присвоение переменной не нужно, однако для транслятора это далеко не так очевидно. Пример использования СЕП на листинге 4:

Листинг 4: Использование СЕП для определения бессмысленных присвоений

```
1 y1 := 1
2 y2 := 2
3 x1 := y2
```

СЕП решает следующие задачи:

- свёртка констант;
- удаление мёртвого кода;
- частичное устранение избыточности;
- снижение стоимости операций;
- распределение регистров.

После трансляции в IR количество инструкций больше изначального в 10-30 раз, так как, например, в соответствии с СЕП для каждого присваивания генерируются новые переменные. 32-х битные инструкции расширяются до 64 бит.

write (write what?)

1.3.4 Сравнение динамических трансляторов

Рассмотрим два аспекта динамических трансляторов, скорость и полноту трансляции.

Для замеров скорости использовался однопоточный бенчмарк nbench.

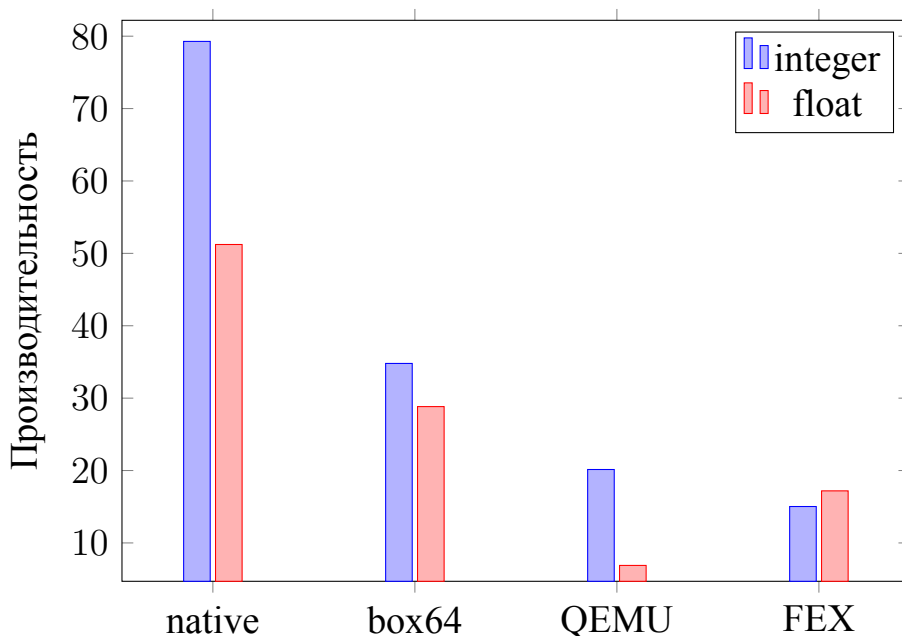


Рисунок 1 – Результаты замеров

Таким образом видно что динамический транслятор box64 достигает 44% скорости целочисленных вычислений и 56% вычислений с плавающей точкой. Используя только критерий скорости, можно сделать вывод что он является лучшим решением для динамической трансляции. Однако, по полноте трансляции он сильно проигрывает двум другим трансляторам: box64 не может транслировать все загружаемые библиотеки, то есть для корректной работы он почти обязательно будет использовать родные для архитектуры библиотеки, что целиком обходит проблему именно трансляции. Также, box64 не может рекомпи-

лизовать бинарные файлы созданные только на языке ассемблера, он их интерпретирует. Наконец, boх64 не поддерживает многие SIMD-операции и не умеет правильно докладывать о доступных расширениях процессора через CPUID. Таким образом boх64 достаточно далек от полноценного решения для трансляции. (адский главред)

QEMU достигает 25% скорости целочисленных вычислений и 13% вычислений с плавающей точкой. Низшая производительность по сравнению с boх64 связана, в том числе, с тем что QEMU не оптимизирован по ARM, а является универсальным решением для разных платформ. Также стоит заметить что QEMU — единственный из рассмотренных трансляторов который умеет эмулировать систему.

FEX достигает 18% скорости целочисленных вычислений и 33% вычислений с плавающей точкой. FEX разработан для трансляции на архитектуру ARM и предоставляет хорошую поддержку различных операций и умеет правильно докладывать о доступных расширениях процессора через CPUID. Так как внутри используется СЕП-представление он лучше остальных трансляторов подходит для проведения оптимизаций.

Рассмотрим результаты индивидуальных тестов, представленных в таблице ??.

Таблица 1 – Таблица результатов отдельных бенчмарков.

Бенчмарк	native	box64	QEMU	FEX
NUMERIC SORT	831.84	421.07	300.83	211.29
STRING SORT	413.19	209.33	57.525	25.112
BITFIELD	258910000	216760000	92293000	56201000
FP EMULATION	370.51	111.52	102.89	41.368
FOURIER	56443	33655	1946.3	9898.4
ASSIGNMENT	22.622	12.22	7.1951	9.7578
IDEA	7196.3	2016.5	1228.8	1660.5
HUFFMAN	2396.6	859.58	625.73	566.98
LU DECOMPOSITION	1261.4	401.87	91.377	280.59

Судя по результатам таблицы можно заметить что минимальная производительность для FEX происходит на тесте «STRING SORT» который производит много работы с памятью. Для QEMU таким тестом оказался «FOURIER» — тест зависящий от производительности FPU. Для box64 — «IDEA», тест измеряющий общую производительность.

1.4 Существующие оптимизации трансляции

write

rewrite с оверхедом? да?... а надо?...)

1.4.1 Использование блоков трансляции

Блоки трансляции используются в каждом из рассматриваемых трансляторов.

В QEMU как блок трансляции выделяется часть кода до момента изменения состояния процессора которое нельзя выяснить на этапе трансляции, например, некоторое ветвление. [8]

В box64, так же как и в QEMU, код разбивается на блоки трансляции, блок заканчивается когда после нее нет других инструкций, например jump, call или

ret, и когда на последнюю инструкцию не ссылается ветвление из этого блока. Исключением являются многобайтовые NOP инструкции, которые обрабатываются отдельно, а прочтение неизвестной инструкции останавливает выполнение блока. [?]

В FEX Frontend.cpp разбивает код на блоки трансляции. Блок так же заканчивается при изменении порядка выполнения, однако рассматривается ситуация когда блоки трансляции находятся в одном месте. Некоторые трансляторы заканчивают трансляцию при изменении порядка выполнения, хотя в скомпилированном коде часто встречаются конструкции похожие на листинг 5:

Листинг 5: Пример кода, сгенерированного компилятором

```
1  test eax, eax
2  jne .Continue
3  ret          <--- Можно продолжать трансляцию после
4                безусловного окончания функции
5  .Continue:
```

Если можно определить адрес условного перехода, то есть возможность продолжить трансляцию. [9]

Одной из важных оптимизаций является связывание различных блоков трансляции. Если не связывать блоки необходимо выходить из цикла выполнения кода, проходить эпилог процедуры и затем искать следующий, необходимый блок.

Например, после выполнения одного блока трансляции QEMU ищет следующий блок, для этого используется PC (эмулируемый program counter) и другая информация о статусе процессора (например регистр CS). Сначала блок ищется в кэше трансляции, если он там не найден он достается из хэш-таблицы и добавляется в кэш. Для поиска нужно выйти из цикла выполнения блока,

пройти через эпилог процедуры, найти следующий блок, запустить его, пройдя через пролог процедуры. В качестве оптимизации предлагается связывать несколько блоков трансляции напрямую.

Для этого в конце блока вызывается `tcg_gen_lookup_and_goto_ptr()`, он в свою очередь вызывает `helper_lookup_tb_ptr` который ищет нужный блок и генерирует инструкцию `goto_ptr`, которая либо продолжит управление в нужном блоке, либо вернется в основной цикл трансляции. Похожая оптимизация используется при ветвлении, если ветвление происходит напрямую, в пределах одной страницы QEMU выполняет поиск блока трансляции на который будет произведено ветвление, а затем сохраняет его адрес в транслированном коде, использование данного метода повышает скорость выполнения на 15%. Таким образом при следующем выполнении этого блока нет необходимости в поиске следующего блока. [8]

В `box64` и `FEX` применяется похожая идея, таким образом сильно снижается время поиска следующего блока при безусловном ветвлении, например в `FEX` при выполнении 500 миллионов ветвлений поиск блока выполнялся 100 миллионов раз. [10]

1.4.2 Поддержка специфичных для архитектуры библиотек

Одним из важных механизмов, позволяющим добиться хорошей производительности, является использование специфичных для архитектуры библиотек. Такой подход используется в `box64` и `FEX`. Например, в `box64` в графически требовательных приложениях производительность близка к 100%, однако в приложениях не использующие сторонние библиотеки (то есть полностью транслируемые) производительность около 50%.

При эмуляции `Quake 3` в `box64` (графического приложения с повторно вызываемыми функциями) производительность была около 85%.

Необходимые для работы приложения библиотеки либо транслируются, либо используется специфичная для архитектуры библиотека (в таком случае производительность выше). Вызовы функций перехватываются для вызова спе-

цифичных для архитектуры функций.

В ELF содержатся специальные символы называемые перемещениями, они используются при линковке для установки адреса необходимой функции. При вызове родной функции в качестве адреса выставляется адрес заранее подготовленного кода, он состоит из последовательности байт `CC 53 43` и следующими за ней двух указателей. Первый указатель рассматривается как указатель на оберточную функции, а второй как указатель на обертываемую (родную) функцию. Оберточной функции передается структура с состоянием процессора и указатель на вызываемую функцию, эта структура распаковывается и вызывается переданная функция. По завершению работы оберточная функции завершается через `ret` или `retl`.

Поиск необходимой функции осуществляется при помощи файлов находящихся в `src/wrapped`, их необходимо определять в ручную, так как названия функций не сохраняются после компиляции программы. Сигнатура функции состоит из символов, определяющих тип возвращаемого значения и типов аргументов, разделенных буквой `F`. Пример сигнатуры показан на рисунке 4.

```
G0(memcpy, pFppL) void* memcpy( void *dest, const void *src, size_t count );
```

Рисунок 2 – Пример сигнатуры функции `memcpy`

После определения функция заносится в таблицу функций библиотеки, а затем находится при помощи разных методов. [11]

В FEX так же существует поддержка специфичных для архитектуры библиотек, но, например, используется другая последовательность байт — `0F 36`, реализация похожа на реализацию в `box64`.

1.4.3 Оптимизации кода

В FEX и QEMU реализованы некоторые оптимизации, свойственные оптимизирующим компиляторам, критерием выбора оптимизаций является скорость работы (так как трансляция динамическая) и эффективность.

box64 является меньшим проектом, поэтому в нем реализовано меньше оптимизаций кода.

1.4.4 Распространение констант

При этом проходе заменяется выражение, которое при выполнении всегда возвращает одну и ту же константу, самой этой константой. Это значение прописывается в инструкцию.

1.4.5 Устранение мертвого кода

Устраняется «бессмысленный», не вызываемый код. Например, на листинге 6 представлена бессмысленная операция:

Листинг 6: Пример бессмысленной инструкции

```
1 and_i32 t0, t0, $0xffffffff
```

При выполнении побитовой операции И, в которой один из аргументов равен максимально возможному для регистра числу, второй аргумент не изменит значение.

В FEX так же устраняются бессмысленные и ненужные операции.

1.4.6 Устранение загрузок контекста

В FEX устраняются бесполезные загрузки контекста, например, если идет сохранение контекста, а затем сразу же его загрузка, такая загрузка выполняться не будет.

1.4.7 Устранение хранения

В QEMU удаляются неиспользуемые перемещения данных, например, в листинге 7 представлен пример не оптимизированного хранения:

Листинг 7: Пример не оптимизированных TCG инструкций

```
1 add_i32 t0, t1, t2
2 add_i32 t0, t0, $1
```

```
3 | mov_i32 t0, $1
```

После оптимизации выполнится только последняя инструкция.

В FEX устраняются бесполезные хранения, не высчитываются неиспользуемые флаги (например, те что будут перезаписаны следующей инструкцией: при операции умножения или деления используют несколько регистров и один из этих регистров будет перезаписан следующей инструкцией), если после ветвления в любом случае перезаписывается какое-либо значение — его можно не хранить.

1.4.8 Сжатие инструкций

Многие x86 инструкции читают или записывают регистры подряд, можно объединять их в пары (и использовать `ld2` или `st1` и подобные).

Некоторые MMX операции (то есть с операндами в 64 бита) можно объединить в операции с операндами в 128 бит которые смогут использовать целый SIMD-регистр архитектуры ARM.

1.4.9 Устранение временных регистров

В FEX, если транслируется блок, который включает в себя весь код функции, и известен используемый двоичный интерфейс, есть возможность исключить временные регистры при сохранении контекста. Также при трансляции целой функции можно убрать загрузки и сохранения в контекст посреди функции и выполнять одно сохранение в конце функции и загрузку в начале.

1.4.10 Анализ живости

В QEMU каждый регистр проверяется на используемость в определенном блоке трансляции. Если регистр не используется в блоке трансляции связанные с ним операции оптимизируются, так как значения этих регистров (и связанное с ними состояние процессора) за блок не могут измениться. Если состояние виртуального процессора меняется, такой блок не будет выполняться пока состояние процессора не будет соответствовать необходимому для блока

(например, другой уровень привилегий).

Например, если на x86 регистры SS, DS и SS содержат в себе 0, транслятор не будет генерировать для них смещение.

1.4.11 Сравнение оптимизаций трансляции

В таблице 1 перечислены выделенные методы трансляции.

Таблица 2 – Таблица методов трансляции.

Методы	QEMU	box64	FEX
Промежуточное представление	+	-	+
Блоки трансляции	+	+	+
Связывание блоков трансляции	+	+	+
Поддержка самомодифицирующегося кода	+	+	±
Поддержка родных библиотек	-	+	+
Распространение констант	+	-	+
Устранение мертвого кода	+	-	+
Устранение загрузок контекста	-	-	+
Устранение хранения	+	-	+
Сжатие инструкций	+	-	+
Устранение временных регистров	-	-	+
Анализ живости	+	-	+

1.5 Поддержка саомодифицирующегося кода

Саомодифицирующийся код на x86 представляет особую проблему, так как нет механизма оповещения об изменении кода, в отличие, например, от ARM.

При запуске в режиме пользователя QEMU помечает все страницы с транслированным кодом как защищенные от записи, при попытке записи в них поднимается сигнал SEGV, допускается запись, транслированная страница и связанные с ней блоки помечаются как не валидные. При запуске эмуляции системы программный MMU выполняет защиту от записи и инвалидацию страниц. [8]

В box64 используется похожий механизм, все транслированные страницы помечаются защищенными на запись, при попытке записи страница в которую произведена запись и все последующие помечаются как "грязные". Однако, они не обязательно являются невалидными, при попытке выполнения кода с такой страницы считается CRC32, полученный результат сравнивается с контрольной суммой посчитанной при создании блока, если суммы совпадают блок объявляется валидным и опять защищается от записи, иначе генерируется новый блок. [?]

В FEX не реализована полноценная поддержка саомодифицирующегося кода, по мнению разработчиков подход QEMU и box64 является единственным приемлемым по скорости и планируется использовать его. [?]

1.6 Поддержка TSO

При трансляции между двоичного кода между платформами следует также учитывать различные модели доступа к памяти, их делят на две группы: слабую и сильную. К слабой модели относятся архитектуры ARM, PowerPC, RISC-V и т.д; к сильной относятся x86/64, специальные режимы работы процессоров архитектуры SPARC и RISC-V и другие. В таблице ?? представлены оптимизаций доступа к памяти на стадии выполнения.

Таблица 3 – Таблица переупорядочиваний обращений к памяти [12]

	ARMv7-A/R	RISC-V (WMO)	RISC-V (TSO)	x86
Чтение может быть переставлено после чтения	Да	Да	-	-
Чтение может быть переставлено после записи	Да	Да	-	-
Запись может быть переставлена после записи	Да	Да	-	-
Запись может быть переставлена после чтения	Да	Да	Да	Да
Атомарные операции могут быть переставлены с чтением	Да	Да	-	-
Атомарные операции могут быть переставлены с записью	Да	Да	-	-

Как видно из таблицы, слабые модели памяти включают в себя большее количество перестановок операций. Таким образом дословная трансляция машинных операций многопоточного приложения с большой вероятностью приведёт к некорректным результатам.

Простейшее решением этой проблемы — использование инструкций с барьером. Такие инструкции устанавливают строгую последовательность между обращениями к памяти до и после барьера. Это означает, что все обращения к памяти перед барьером выполнятся до первого обращения к памяти после барьера, если все инструкции доступа к памяти будут выполнятся как инструкции с барьером, невозможна будет ситуация с перестановкой данных.

Использование инструкций с барьером оказывает сильное влияние на производительность ядер с развитой системой внеочередного исполнения, рассмотрим пример программы на ассемблере ARM:

Листинг 8: Простая программа

```

1  .globl _start
2  _start:
3      mov     x0, 58368
4      movk    x0, 0x540b, lsl 16
5      movk    x0, 0x2, lsl 32
6  loop:
7      mov     x21, #0xfffffffffffffec

```

8	add	x21, sp, x21
9	str	w20, [x21]
10		
11	sub	x0, x0, #1
12	cmp	x0, 0
13	bne	loop
14		
15	mov	x0, #0
16	mov	w8, #93
17	svc	#0

Эта программа 10000000000 раз производит сохранение регистра w20 в область памяти стека. Таким образом на время выполнения программы в сильнее всего влияет операция `str w20, [x21]`. Проведём замеры выполнения программы с операцией `str w20, [x21]` и с операцией `stlr w20, [x21]` — доступом к памяти с барьером. Результаты замеров времени выполнения программы представлены на рисунке 8.

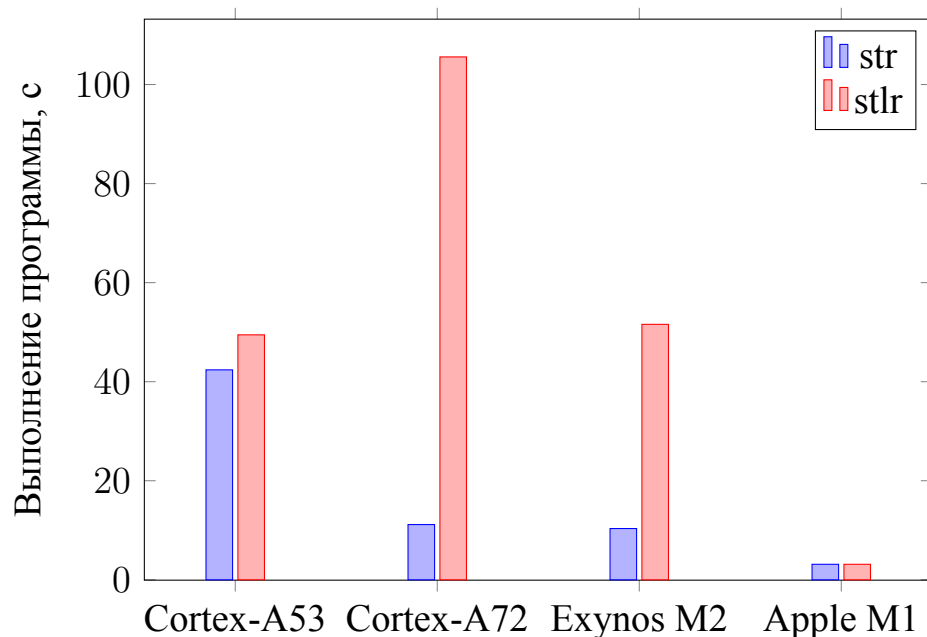


Рисунок 3 – Замеры времени выполнения программы

На рисунке видно что ядро Cortex-A53 не сильно пострадало из-за замены

операции чтения на операцию чтения с барьером, это связано с тем что его выпустили в 2012 году как энергоэффективное ядро, таким образом оно не включает в себя полноценное внеочередное исполнение, только имея возможность одновременного исполнения некоторых команд. Так же видно что барьерный доступ к памяти для процессоров Cortex-A72 и Exynos M2 является затратной операцией, время выполнения программы вырастает в 10 и в 5 раз соответственно. Для процессора Apple M1 время выполнения этой программы не изменилось, однако барьерный доступ к памяти в нем также связан с потерями в производительности. Отношение времени выполнения барьерных инструкций к обычным как правило падает в новых ядрах.

Несмотря на большую потерю в производительности FEX обрабатывает любой доступ к памяти, кроме доступа к памяти через регистр стека, как доступ к памяти с барьером, это простое, но не производительное решение.

Инженеры компании Apple столкнулись с такой же проблемой и решили ее при помощи специального режима работы своего процессора. Этот режим работы не описан в спецификации ARM, в нем процессор придерживается строгой модели памяти, однако этот режим невозможно включить со стороны пользователя без расширений ядра, которые Apple не рекомендует использовать.

1.6.1 Регистр RBP

В скомпилированном под x86 с -O0 флагом коде часто для доступа к данным на стеке используется регистр RBP. По изначальной задумке этот регистр предназначен для работы со стеком, однако современные оптимизирующие компиляторы используют его как дополнительный регистр общего назначения. Рассмотрим программу на языке C:

Листинг 9: Цикл на C

```
1 int main() {  
2     const unsigned long upto = 10000000000;
```



```

3      int j = 2;
4      for (unsigned long i = 1; i < upto; i++) {
5          j *= i;
6      }
7
8      return j;
9  }

```

Скомпилированный компилятором gcc без оптимизаций код:

Листинг 10: Фрагмент скомпилированного кода

```

1  main:
2  endbr64
3  pushq    %rbp
4  movq     %rsp, %rbp
5  movabsq   $100000000000, %rax
6  movq     %rax, -8(%rbp)
7  movl     $2, -20(%rbp)
8  movq     $1, -16(%rbp)

```

Как видно из скомпилированного фрагмента, доступ к стеку производится через RBP, такая организация доступа к памяти используется и далее по коду программы. Несмотря на то что доступ производится к памяти стека FEX не считает такой доступ TSO-безопасным и транслирует его в инструкции доступа к памяти с барьером.

Однако нельзя считать любой доступ через RBP безопасным, так как он часто используется как регистр общего назначения.

1.7 Оценка скорости работы трансляторов

write (или уже written? :))

1.8 Вывод

В этом разделе рассмотрены некоторые методы трансляции применяемые в среде открытого программного обеспечения. По результатам анализа FEX является наиболее полным решением для трансляции кода, однако он достигает только 18% скорости целочисленных вычислений и 33% вычислений с плавающей точкой, крупные потери в скорости связаны с организацией доступа к памяти.

(нужно же еще как-то вкрутить что я решил доступ к памяти править? а то несвязно с конструкторской получается)

2 Конструкторская часть

В данном разделе описываются используемые структуры данных, проводится подробное описание алгоритма оптимизации динамической трансляции доступа к памяти и сопутствующих ему алгоритмов. Проводится проектирование программной реализации алгоритма оптимизации динамической трансляции доступа к памяти.

Алгоритм рассчитан на организацию кода в блоки из которых одна точка выхода в конце блока и одна точка входа — в начале блока.

2.1 Архитектура программного обеспечения

ну вообще не обязательно ?

или обязательно.....

2.2 Используемые структуры данных

BlockInfo — структура хранящая информацию об отдельном блоке транслированного кода. Включает в себя:

- *State* — рассчитанное состояние блока, свидетельствует о состоянии регистра RBP, либо регистр не менялся, либо в нем адрес связанный со стеком, либо в нем адрес не связанный со стеком.
- *StackNodes* — множество операций загружающих стековое значение;
- *UnStackNodes* — множество операций загружающих не стековое значение;
- *Predecessors* — множество блоков трансляции предшествующих этому;
- *Visited* — флаг, показывает обработан ли блок.

2.3 Алгоритм оптимизации динамической трансляции доступа к памяти

Входные данные: Множество блоков транслированного кода *IRBlocks*.

Выходные данные: Множество блоков транслированного кода *IRBlocks* с оптимизированным доступом к памяти.

Алгоритм 1 Алгоритм оптимизации динамической трансляции доступа к памяти

```
1:  $in \leftarrow$  множество блоков транслированного кода
2:  $out \leftarrow$  множество блоков транслированного кода с оптимизированным доступом к памяти
3:  $ControlFlow \leftarrow$  построить граф потока выполнения блоков
4: for  $i$  in  $in$  do
5:    $StackStatus \leftarrow$  STACK
6:   for  $j$  in предшественники  $i$  do
7:      $PredecessorState \leftarrow$  рассчитать состояние блока с учетом предшественников
8:     if  $CodeNode \neq$  STACK then
9:        $StackStatus \leftarrow PredecessorState$ 
10:    end if
11:  end for
12:  for  $CodeNode$  in строки кода  $i$  do
13:    if  $CodeNode =$  инструкция загрузки адреса стека в RBP then
14:       $StackStatus \leftarrow$  STACK
15:    end if
16:    if  $CodeNode =$  инструкция загрузки любого значения, кроме стека в RBP then
17:       $StackStatus \leftarrow$  NOT_STACK
18:    end if
19:    if  $StackStatus ==$  STACK and  $CodeNode =$  инструкция записи или чтения по адресу
    регистра RBP then
20:       $out \leftarrow$  заменить барьерную инструкцию на обычную
21:    end if
22:  end for
23: end for
```

2.4 Алгоритм построения графа выполнения блоков

Алгоритм 2 Алгоритм построения графа выполнения блоков

```
1:  $in \leftarrow$  множество блоков транслированного кода
2:  $Predecessors \leftarrow$  ориентированный граф потока выполнения блоков
3:  $State \leftarrow$  множество состояний блоков
4:  $StackNodes \leftarrow$  множество строк изменяющих содержимое регистра RBP на значение связанное с RSP
5:  $UnStackNodes \leftarrow$  множество строк изменяющих содержимое регистра RBP на значение не связанное с RSP
6: for  $i$  in  $in$  do
7:   for  $j$  in строки  $i$  do
8:     if  $j$  — условный переход then
9:        $Predecessors \leftarrow i$  — предшественник блоков куда передается управление
10:    end if
11:    if  $j$  — безусловный переход then
12:       $Predecessors \leftarrow i$  — предшественник блока куда передается управление
13:    end if
14:    if  $j$  — запись регистра RBP then
15:      if записывается связанное с RSP значение then
16:         $State \leftarrow$  в блоке стековое значение
17:         $StackNodes \leftarrow$  строка записи стекового значения
18:      else
19:         $State \leftarrow$  в блоке не стековое значение
20:         $UnStackNodes \leftarrow$  строка записи не стекового значения
21:      end if
22:    end if
23:  end for
24: end for
```

2.5 Алгоритм расчета распространенного состояния стека

Этот алгоритм используется для определения состояния регистра в последующих блоках. Если в блоке 1 была произведена загрузка стекового значения в регистр RBP, а в блоке 2, для которого предок только блок 1, этот регистр не

меняется, то для всех потомков блока 2 следует считать что регистр содержит стековое значение.

Алгоритм 3 Алгоритм расчета распространенного состояния стека

```
1: TargetBlock  $\leftarrow$  блок для которого рассчитывается состояние стека
2: ControlFlow  $\leftarrow$  граф потока выполнения блоков
3: ResultingState  $\leftarrow$  состояния стека для блока с учетом предыдущих
4: out STACK
5: for i in предшественники TargetBlock do
6:   PredecessorState  $\leftarrow$  рассчитать распространенное состояние стека для i
7:   if PredecessorState = NOT_CHANGED then
8:     ResultingState  $\leftarrow$  NOT_CHANGED
9:   end if
10:  if PredecessorState = NOT_STACK then
11:    ResultingState  $\leftarrow$  NOT_STACK
12:    break
13:  end if
14: end for
```

2.6 Вывод

Были описаны используемые структуры данных, дано описание алгоритмов используемых.. Был спроектирован алгоритм оптимизации динамической трансляции доступа к памяти.?

3 Технологическая часть

В данном разделе описываются средства разработки программного обеспечения, требования к вычислительной системе. Приводится структура разработанного приложения.

3.1 Выбор средств разработки

Из рассмотренных в аналитическом разделе трансляторов FEX лучше прочих подходит для реализации алгоритма, так как в нем присутствует СЕП.

3.1.1 Выбор языка программирования

Динамический транслятор FEX написан на языке C++, в нем используются разные библиотеки, например ядро транслятора FEXCore это библиотека реализующая непосредственно динамическую трансляцию. FEXCore также написан на C++ и не предусматривает расширений с помощью внешних модулей, поэтому для того чтобы ее модифицировать надо писать код на C++.

3.1.2 Сборка программного обеспечения

Для сборки проекта используется система сборки CMake. Для компиляции реализации алгоритма необходимо добавить имя файла в соответствующий CMakeLists.txt.

Для сборки всего проекта используется ninja. На листинге 13 указана конфигурация сборки проекта.

Листинг 11: Конфигурация сборки проекта

```
1 CC=clang CXX=clang++ cmake -DCMAKE_INSTALL_PREFIX=/usr \  
2 -DCMAKE_BUILD_TYPE=Release -DENABLE_LTO=True \  
3 -DBUILD_TESTS=False -DCMAKE_CXX_COMPILER_LAUNCHER=ccache \  
4 -DENABLE_CLANG_FORMAT=False -G Ninja ..
```

Оптимизирующий проход создан на основе версии 2204, коммит 37f1e55ed5dc7a35ba9bf875e250de0b75581a22.

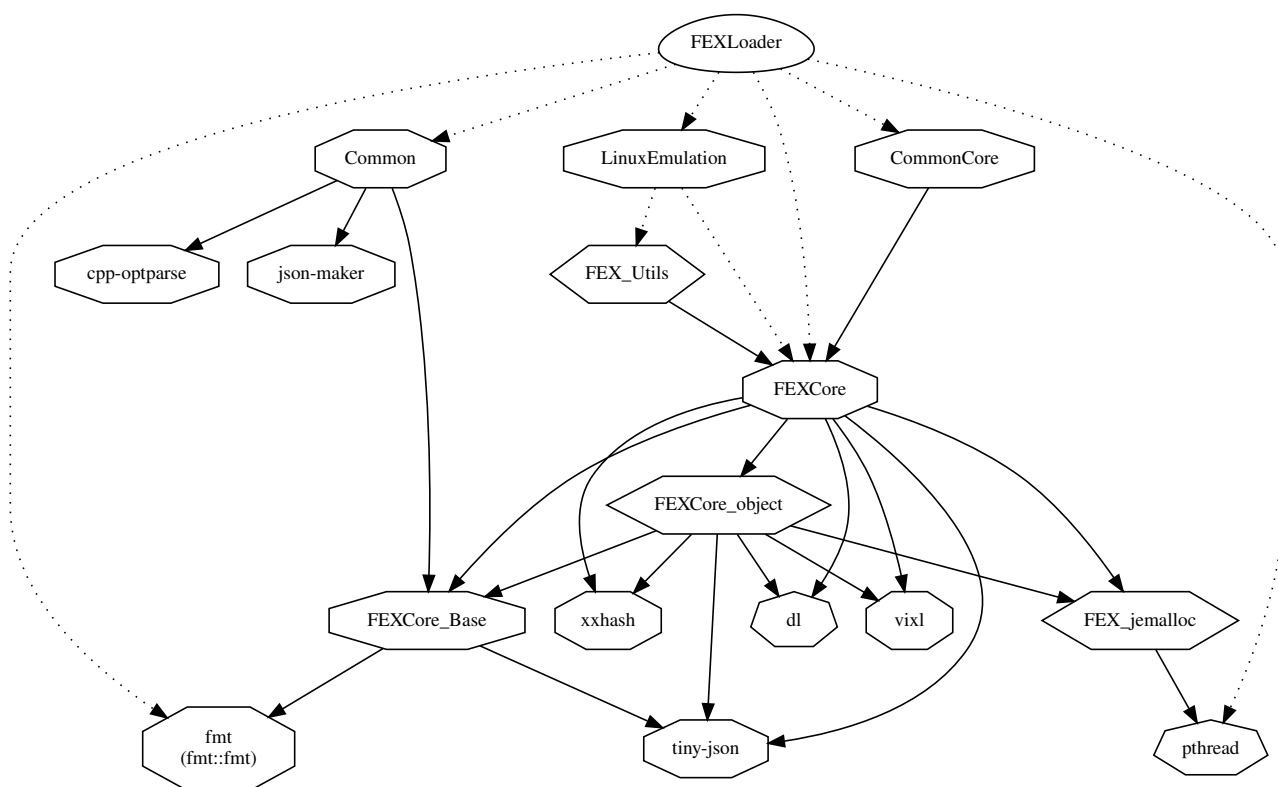


Рисунок 4 – Диаграмма зависимостей userspace эмулятора FEXLoader

Вносимые в проект изменения относятся к FEXCore_Base.

3.2 Требования к вычислительной системе

Для запуска программного обеспечения требуется исходный код динамического транслятора FEX. Так как алгоритм реализует оптимизацию доступа к памяти с сохранением корректного выполнения многопоточных приложений для прироста в скорости требуется процессор с поддерживаемой архитектурой со слабым доступом к памяти и несколькими ядрами, то есть оптимизирующий проход будет приводить к росту производительности не только на ARM, но и, например, RISC-V.

Для разработки и сборки проекта использовался ноутбук на основе процессора RK3399, архитектуры ARM. Несмотря на это разработка не привязана к конкретной архитектуре, теоретически разработку можно вести на любой архитектуре имеющий набор инструментов для кросс-компиляции под архитектуру ARM.

3.3 Структура программного обеспечения

Разработанное ПО реализовано как оптимизирующий проход над промежуточным представлением.

При инициализации транслятора вызывается функция `AddDefaultPasses`, внутри которой вызываются функции `InsertPass` регистрирующие оптимизирующие проходы в определенном порядке. Функция `InsertPass` принимает на вход `std::unique_ptr<Pass>` — указатель на экземпляр класса прохода. У класса обязательно должен быть метод `Run` принимающий на вход `IREmitter` и возвращающий `bool` — был ли изменен код во время прохода.

Разработанный проход зависит от `StaticRegisterAllocationPass`, во время этого прохода статически выделяются регистры, на основе этих регистров и идет оптимизация доступа памяти.

Отключить все оптимизирующие проходы можно с помощью аргумента `-O` или `-o0`.

3.4 Распространение программного обеспечения

Программное обеспечение распространяется в виде патча сформированного командой `git format-patch`. Для применения патча к проекту можно, например, использовать `git am`.

3.5 Дополнительные утилиты

Для анализа генерируемого кода нужно анализировать скомпилированный код, такой возможности у FEX нет, поэтому была проведена модификация функции `Context::CompileBlock` которая компилирует и запускает блок. Перед запуском вся область запускаемой памяти сохраняется в бинарный файл.

Листинг 12: Сохранение скомпилированного блока

```
1  --- a/External/FEXCore/Source/Interface/Core/Core.cpp
2  +++ b/External/FEXCore/Source/Interface/Core/Core.cpp
3  @@ -1001,6 +1001,15 @@ namespace FEXCore::Context {
4  }
5  }
6
7  +
8  +  FILE *fp;
9  +
10 +  std::string filename = "Core_Block_other_" + \
11 +    std::to_string((uint64_t)CodePtr) + ".cdmp";
12 +  fp = fopen(filename.c_str(), "wb");
13 +  fwrite((void*)CodePtr, DebugData->HostCodeSize, 1, fp);
14 +  fclose(fp);
15 +  printf("block dumped %s\n", filename.c_str());
16 +
17 +  if (IRCaptureCache.PostCompileCode(
18 +    Thread,
19 +    CodePtr,
20 +    --
21 +    2.36.0
22
```

Обработка сохраненной памяти затем выполнялась при помощи `objdump` — в итоге получался код на ассемблере.

После анализа скомпилированного кода были сделаны выводы по основным факторам замедляющим выполнение скомпилированного кода, таких как барьерный доступ к памяти.

3.6 Вывод

В данном разделе были описаны средства разработки программного обеспечения, требования к вычислительной системе. Была дана структура разработанного приложения.

4 Исследовательская часть

В рамках дипломной работы было проведено исследование изменения результатов бенчмарка nbench с разработанным проходом. Результаты исследования представлены в этом разделе.

4.1 Описание используемых данных

Для проведения исследования используется бенчмарк nbench скомпилированный при помощи gcc в два разных бинарных файла: с флагом -O0 и с флагом -O3.

В бенчмарке nbench реализовано 9 тестов:

- Numeric sort — сортировка массива 32-х битных чисел. Оценивается производительность работы с целыми числами.
- String sort — сортировка массива символов. Проверяется скорость работы с памятью.
- Bitfield — выполняет различные битовые операции. Например: очистить или выставить n бит, инвертировать число. Оценивается скорость выполнения таких операций.
- Emulated floating-point — небольшой эмулятор для работы с числами с плавающей запятой. Хорошо оценивает общую производительность.
- Fourier coefficients — рассчитывает коэффициенты Фурье. Оценивает производительность FPU, при этом память использует не активно.
- Assignment algorithm — решение задачи о назначениях. Работает с массивом, на результат влияет скорость последовательного доступа к памяти.
- Huffman compression — алгоритм Хаффмана. Операции с байтами, битами и с целыми числами. Считается хорошей оценкой общей производительности.
- IDEA encryption — алгоритм шифрования. Оценивает скорость исполнения кода.

— LU Decomposition — алгоритм решения линейных уравнений. Работает с числами с плавающей запятой, использует только фундаментальные операции (+, -, *, /).

В итоге получаются INTEGER INDEX и FLOATING-POINT INDEX — комбинированный результат бенчмарков.

Каждый бенчмарк запускается 5 раз, затем рассчитывается среднее квадратическое отклонение и доверительный интервал с уровнем доверия 0.95. Если длина полуинтервала меньше 1% от абсолютного значения среднего, то замер времени считается успешным, иначе проводятся дополнительные замеры.

Динамический транслятор FEX запускается на SOC Rockchip RK3399, Exynos 8895 и Apple M1. Система работающая на RK3399 работает под управлением Linux; Exynos — Android, сам запуск производится через chroot в окружение Debian; Apple M1 — виртуальная машина с Ubuntu.

Из-за того что SOC RK3399 и Exynos 8895 используют архитектуру big.LITTLE у них одинаковые «малые» ядра — Cortex A53. Больших различий между этими ядрами нет, поэтому в тестах использовались результаты работы этого ядра на RK3399. Раздельно представлены результаты бенчмарков для их «больших» ядер — Cortex A72 для RK3399 и Mongoose M2 для Exynos 8895.

Замеры времени выполнения проводились на версии FEX-2204.

4.2 Результаты исследования

На рисунках 5-9 представлены результаты замеров скорости выполнения nbench с использованием алгоритма и без.

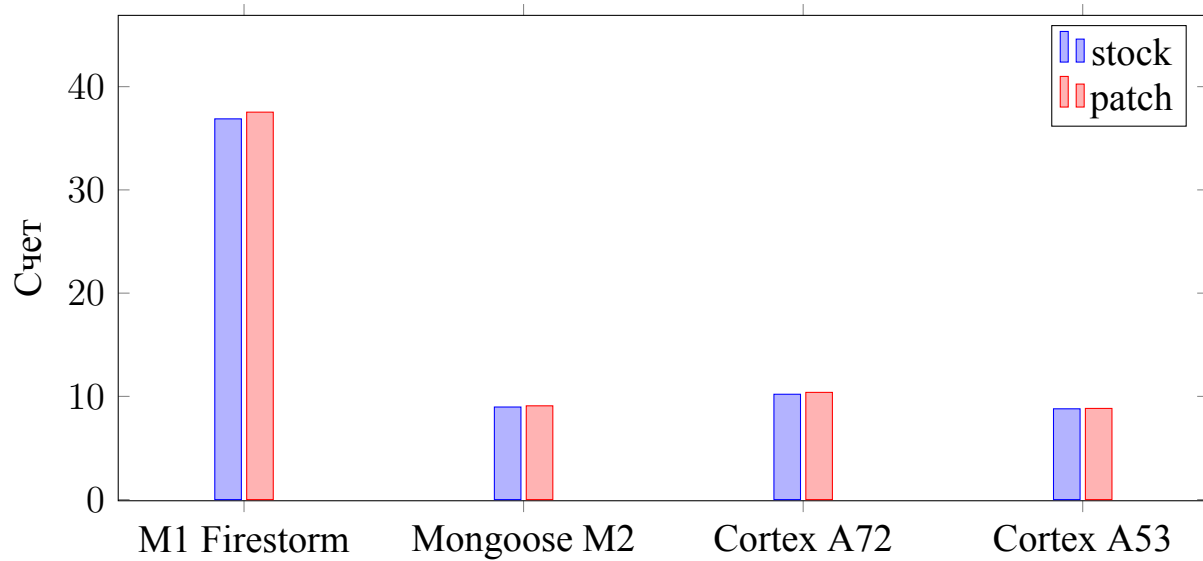


Рисунок 5 – nbench O0

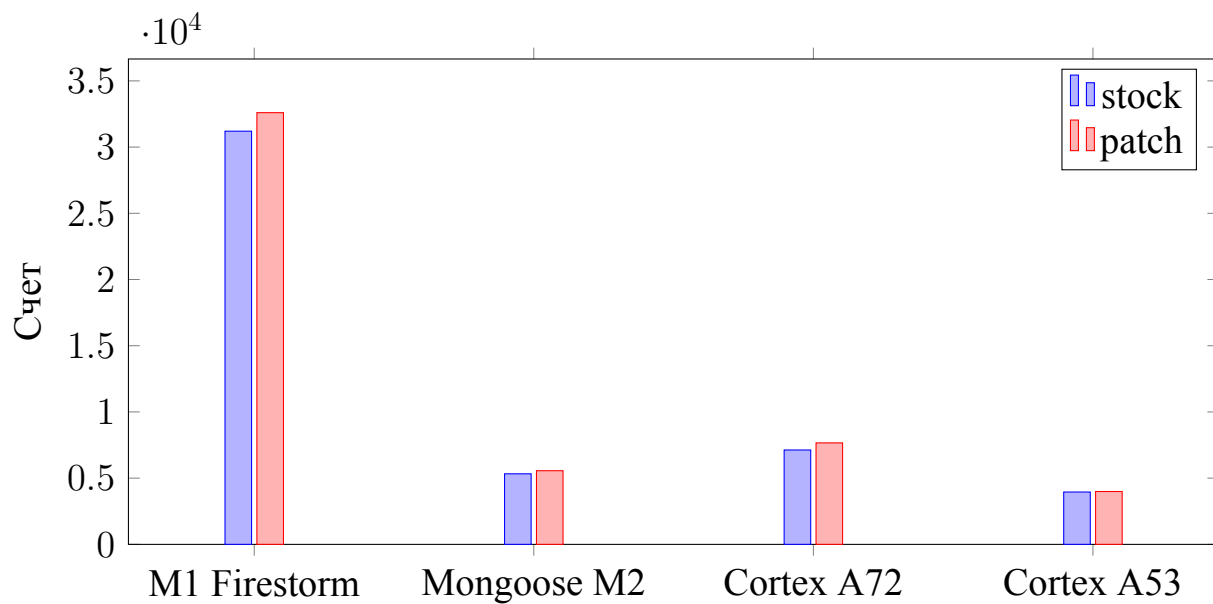


Рисунок 6 – nbench FOURIER O0

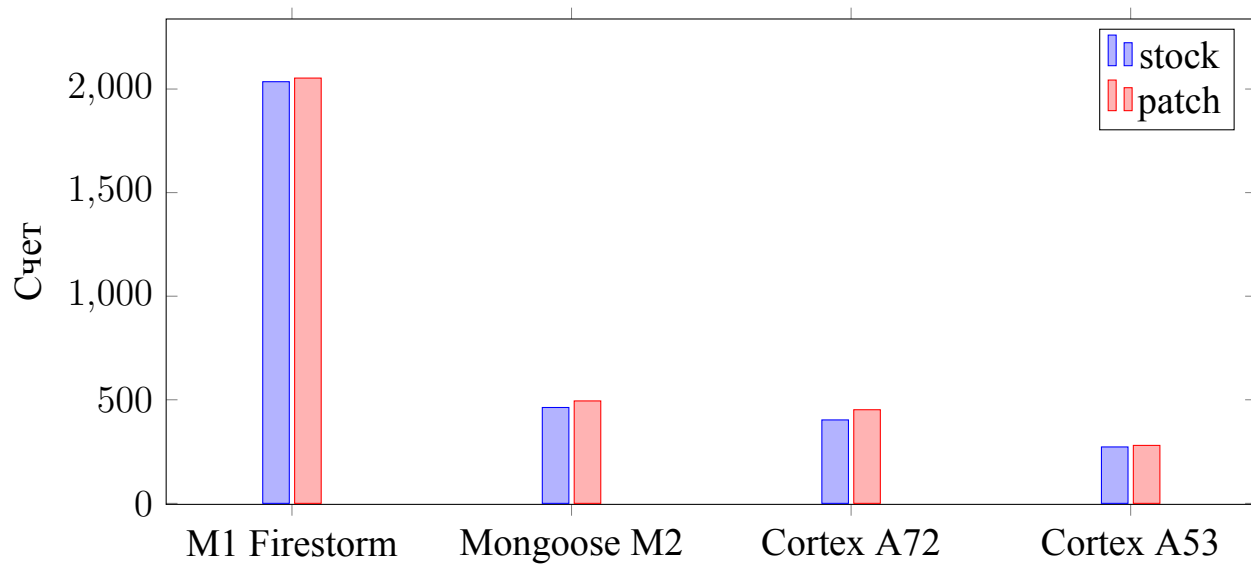


Рисунок 7 – nbench IDEA O0

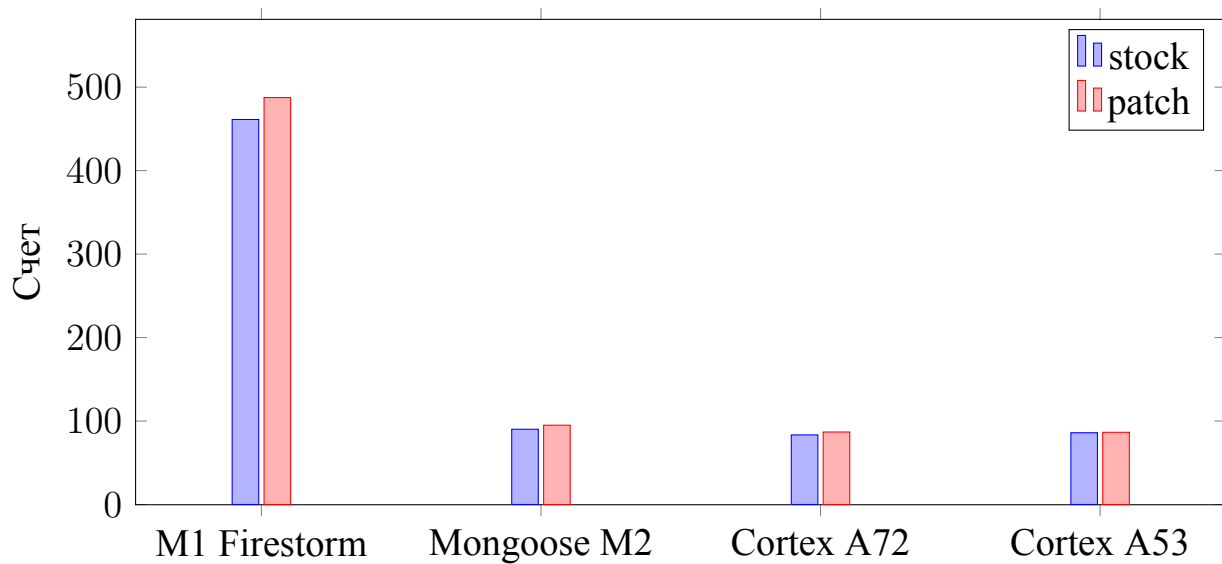


Рисунок 8 – nbench HUFFMAN O0

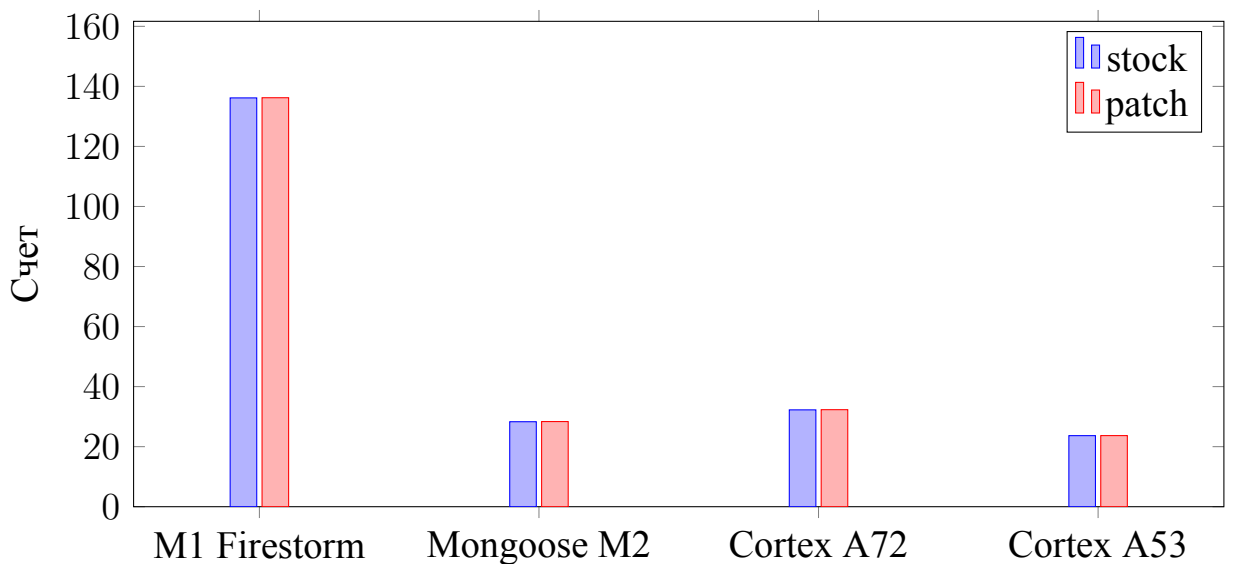


Рисунок 9 – nbench O3

Таким образом по результатам замеров видно что проход работает только на не оптимизированном бенчмарке (собранным с флагом -O0), также стоит заметить что для Apple M1 рост в производительности на выделенных бенчмарках меньше.

На O3 рост производительности незначителен, это связано с тем что регистр RBP, при оптимизации, используется в качестве регистра общего назначения.

Рост в производительности составил:

- Cortex A53 — производительность в среднем выше в 0.42 процента, для FOURIER — прирост 1 процент, IDEA — 2.7 процента, HUFFMAN — меньше процента;
- Cortex A72 — производительность в среднем выше в 1.77 процента, для FOURIER — прирост 7.5 процента, IDEA — 12.2 процента, HUFFMAN — 4.2 процента;
- Mongoose M2 — производительность в среднем выше в 1.29 процента, для FOURIER — прирост 4.4 процента, IDEA — 6.8 процента, HUFFMAN — 5.3 процента;
- M1 Firestorm — производительность в среднем выше в 1.76 процента,

для FOURIER — прирост 4.5 процента, IDEA — меньше процента, HUFFMAN — 5.7 процента;

Более высокий рост производительности для старых ядер связан с затратностью барьерных операций с памятью, новые поколения ядер уменьшают затратность таких операций. У Cortex A53, как было выяснено в аналитическом разделе, разница в скорости выполнения программ с барьерным доступом к памяти и без него мала, поэтому для этого ядра результаты меньше.

4.3 Тестирование на надежность

Главной причиной разработки алгоритма по оптимизации доступа к памяти являлась слабая модель архитектуры ARM, что при трансляции многопоточных приложений приводило к ошибкам.

Для проверки корректной работы прохода использовался `tst-cond16.c` — тест библиотеки `libc`. Этот тест создает 8 потоков которые проводят операции с мьютексами. Стандартная версия транслятора и версия с оптимизирующим проходом этот тест проходит, без барьерных инструкций и с безусловной заменой барьерных операций связанных с регистром RBP тест не проходит.

4.4 Вывод

В результате исследования было установлено, что проход увеличивает производительность отдельных бенчмарков и в целом повышает скорость выполнения транслированного кода, но только если исходный код был собран без оптимизаций.

Также было частично доказано что барьерные инструкции доступа к памяти сильнее влияют на ранее выпущенные ядра ARM.

ЗАКЛЮЧЕНИЕ

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. ГОСТ 19781-83. Вычислительная техника. Терминология: Справочное пособие. Выпуск 1 / Рецензент канд. техн. наук Ю. П. Селиванов. Москва: Издательство стандартов, 1989. – 168 с.
2. Першиков В. И. Толковый словарь по информатике / В. И. Першиков, В. М. Савинков – Москва: Финансы и статистика, 1991. – 543 с.
3. Толковый словарь по вычислительным системам / Под ред. В. Иллинуорта и др.: Пер. с англ. А. К. Белоцкого и др.; Под ред. Е. К. Масловского. – Москва: Машиностроение, 1990. – 560 с.
4. Arm Limited Roadshow Slides Q1 2020 [Электронный ресурс]. – Режим доступа: https://group.softbank/system/files/pdf/ir/presentations/2020/arm-roadshow-slides_q1fy2020_01_en.pdf, свободный – (24.11.2021)
5. Probst, Mark. Fast Machine-Adaptable Dynamic Binary Translation. 2001.
6. box64: Inner workings [Электронный ресурс]. – Режим доступа: <https://box86.org/2021/07/inner-workings-a-high%e2%80%91level-view-of-box86-and-a-low%e2%80%91level-view-of-the-dynarec/>, свободный – (11.12.2021)
7. tcg/README [Электронный ресурс]. – Режим доступа: <https://gitlab.com/qemu-project/qemu/-/blob/master/tcg/README>, свободный – (26.12.2021)
8. QEMU: Translator Internals [Электронный ресурс]. – Режим доступа: <https://qemu.readthedocs.io/en/latest/devel/tcg.html>, свободный – (11.12.2021)
9. FEXCore Frontend [Электронный ресурс]. – Режим доступа: <https://github.com/FEX-Emu/FEX/blob/main/External/FEXCore/docs/Frontend.md>, свободный – (26.12.2021)

10. FEX-Emu: Internals in High Level @ ungleich.ch [Электронный ресурс]. – Режим доступа: <https://www.youtube.com/watch?v=BYIIwaHfl3E>, свободный – (26.12.2021)
11. box64: A deep dive into library wrapping [Электронный ресурс]. – Режим доступа: <https://box86.org/2021/08/a-deep-dive-into-library-wrapping/>, свободный – (11.12.2021)
12. Paul E. McKenney. Memory Barriers: a Hardware View for Software Hackers. 2010. <http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2010.06.07c.pdf>
13. Memory Reordering Caught in the Act [Электронный ресурс]. – Режим доступа: <https://preshing.com/20120515/memory-reordering-caught-in-the-act/>, свободный – (29.04.2022)

ПРИЛОЖЕНИЕ А

Листинг 13: Проход оптимизации доступа к памяти

```
1  #include <FEXCore/Core/X86Enums.h>
2  #include <FEXCore/IR/IR.h>
3  #include <FEXCore/IR/IREmitter.h>
4  #include <FEXCore/IR/IntrusiveIRList.h>
5
6  #include <array>
7  #include <iostream>
8
9  #include "Interface/Core/OpcodeDispatcher.h"
10 #include "Interface/IR/PassManager.h"
11
12 namespace FEXCore::IR {
13
14     enum RBP_STATE {
15         NOT_CHANGED = (0b000 << 0),
16         STACK = (0b001 << 0),
17         NOT_STACK = (0b010 << 0),
18     };
19
20     struct BlockInfo {
21         int State = NOT_CHANGED;
22         std::set<OrderedNode*> StackNodes;
23         std::set<OrderedNode*> UnStackNodes;
24         std::vector<OrderedNode*> Predecessors;
25         bool Visited = false;
26     };
27
28     class StackAccessTSORemovalPass final : public Pass {
29     public:
30         bool Run(IREmitter* IREmit) override;
31
32     private:
33         void CalculateControlFlowInfo(IREmitter* IREmit);
34         int CalculateComplexState(const OrderedNode* TargetNode,
35                                 const IRListView& CurrentIR);
36         std::unordered_map<NodeID, BlockInfo> OffsetToBlockMap;
37     };
38
39     static bool HitsRegister(const int Register, IREmitter* IREmit,
40                             IROp_Header* Op) {
41         std::queue<IROp_Header*> values{};
42         values.push(Op);
43
44         while (!values.empty()) {
```

```

45         IROp_Header* ValOp = values.front();
46         values.pop();
47
48         switch (ValOp->Op) {
49             case IR::OP_ADD:
50             case IR::OP_SUB:
51             case IR::OP_OR:
52             case IR::OP_AND: {
53                 values.push(IREmit->GetOpHeader(ValOp->Args[0]));
54                 values.push(IREmit->GetOpHeader(ValOp->Args[1]));
55
56                 break;
57             }
58             case IR::OP_LOADREGISTER: {
59                 auto LocalOp = ValOp->CW<IR::IROp_LoadRegister>();
60
61                 if (LocalOp->Offset ==
62                     offsetof(FEXCore::Core::CPUState, gregs[Register]) &&
63                     LocalOp->Class == GPRClass) {
64                     return true;
65                 }
66
67                 break;
68             }
69             default: {
70                 break;
71             }
72         }
73     }
74
75     /* never hit the RSP register, so it's not a stack value being
76     * loaded. */
77     return false;
78 }
79
80 void StackAccessTSORemovalPass::CalculateControlFlowInfo(IREmitter* IREmit) {
81     IRListView CurrentIR = IREmit->ViewIR();
82
83     for (auto [BlockNode, BlockHeader] : CurrentIR.GetBlocks()) {
84         BlockInfo* CurrentBlock =
85             &OffsetToBlockMap.try_emplace(CurrentIR.GetID(BlockNode)).first->second;
86         for (auto [CodeNode, IROp] : CurrentIR.GetCode(BlockNode)) {
87             switch (IROp->Op) {
88                 case IR::OP_CONDJUMP: {
89                     auto Op = IROp->CW<IR::IROp_CondJump>();
90
91                     {
92                         auto Block =
93                             &OffsetToBlockMap.try_emplace(Op->TrueBlock.ID()).first->second;
94                         Block->Predecessors.emplace_back(BlockNode);
95                     }

```

```

96
97         {
98             auto Block = &OffsetToBlockMap.try_emplace(Op->FalseBlock.ID())
99                 .first->second;
100             Block->Predecessors.emplace_back(BlockNode);
101         }
102
103         break;
104     }
105     case IR::OP_JUMP: {
106         auto Op = IROp->CW<IR::IROp_Jump>();
107
108         {
109             auto Block =
110                 OffsetToBlockMap.try_emplace(Op->Header.Args[0].ID()).first;
111             Block->second.Predecessors.emplace_back(BlockNode);
112         }
113
114         break;
115     }
116     case IR::OP_STOREREGISTER: {
117         IROp_StoreRegister* Op = IROp->CW<IR::IROp_StoreRegister>();
118         if (Op->Offset == offsetof(FEXCore::Core::CPUState,
119             gregs[FEXCore::X86State::REG_RBP]) &&
120             Op->Class == GPRClass) {
121             if (HitsRegister(FEXCore::X86State::REG_RSP, IREmit,
122                 IREmit->GetOpHeader(Op->Value))) {
123                 CurrentBlock->State = STACK;
124                 CurrentBlock->StackNodes.emplace(CodeNode);
125             } else {
126                 CurrentBlock->State = NOT_STACK;
127                 CurrentBlock->UnStackNodes.emplace(CodeNode);
128             }
129         }
130         break;
131     }
132     default:
133         break;
134 }
135 }
136 }
137 }
138
139 // returns the combined state of all the predecessors and the block itself
140 int StackAccessTSORemovalPass::CalculateComplexState(
141     const OrderedNode* TargetNode, const IRListView& CurrentIR) {
142     auto TargetPair = OffsetToBlockMap.find(CurrentIR.GetID(TargetNode));
143     if (TargetPair == OffsetToBlockMap.end()) {
144         std::cout << "TargetPair not found in offsets!" << std::endl;
145         return NOT_STACK;
146     }

```

```

147
148     BlockInfo* TargetBlock = &TargetPair->second;
149
150     if (TargetBlock->Visited || TargetBlock->Predecessors.size() == 0 ||
151     TargetBlock->State != NOT_CHANGED) {
152         return TargetBlock->State;
153     }
154
155     TargetBlock->Visited = true;
156     int ResultingState = STACK;
157     for (auto i : TargetBlock->Predecessors) {
158         // only call if predecessor is not visited?
159         if (i == TargetNode) {
160             continue;
161         }
162         int PredecessorState = CalculateComplexState(i, CurrentIR);
163
164         if (PredecessorState == NOT_CHANGED) {
165             ResultingState = NOT_CHANGED;
166         } else if (PredecessorState == NOT_STACK) {
167             ResultingState = NOT_STACK;
168             break;
169         }
170     }
171
172     TargetBlock->State = ResultingState;
173
174     return TargetBlock->State;
175 }
176
177 bool StackAccessTSORemovalPass::Run(IREmitter* IREmit) {
178     CalculateControlFlowInfo(IREmit);
179
180     IRListView CurrentIR = IREmit->ViewIR();
181     bool Changed = false;
182
183     for (auto [BlockNode, BlockHeader] : CurrentIR.GetBlocks()) {
184         auto CurrentBlockPair = OffsetToBlockMap.find(CurrentIR.GetID(BlockNode));
185         if (CurrentBlockPair == OffsetToBlockMap.end()) {
186             std::cout << "Block not found in offsets" << std::endl;
187             continue;
188         }
189
190         int StackStatus = STACK;
191         BlockInfo* CurrentBlock = &CurrentBlockPair->second;
192         for (auto CodeNode : CurrentBlock->Predecessors) {
193             int PredecessorState = CalculateComplexState(CodeNode, CurrentIR);
194
195             if (PredecessorState != STACK) {
196                 StackStatus = PredecessorState;
197                 break;

```



```

198         }
199     }
200
201     for (auto [CodeNode, IROp] : CurrentIR.GetCode(BlockNode)) {
202         if (CurrentBlock->StackNodes.contains(CodeNode)) {
203             StackStatus = STACK;
204         }
205         if (CurrentBlock->UnStackNodes.contains(CodeNode)) {
206             StackStatus = NOT_STACK;
207         }
208         if (IROp->Op == IR::OP_LOADMEMTSO && StackStatus == STACK) {
209             if (HitsRegister(FEXCore::X86State::REG_RBP, IREmit,
210                             IREmit->GetOpHeader(IROp->Args[0])) ||
211                 HitsRegister(FEXCore::X86State::REG_RSP, IREmit,
212                             IREmit->GetOpHeader(IROp->Args[0]))) {
213                 IROp->Op = IR::OP_LOADMEM;
214                 Changed = true;
215             }
216         }
217         if (IROp->Op == IR::OP_STOREMEMTSO && StackStatus == STACK) {
218             if (HitsRegister(FEXCore::X86State::REG_RBP, IREmit,
219                             IREmit->GetOpHeader(IROp->Args[1])) ||
220                 HitsRegister(FEXCore::X86State::REG_RSP, IREmit,
221                             IREmit->GetOpHeader(IROp->Args[1]))) {
222                 IROp->Op = IR::OP_STOREMEM;
223                 Changed = true;
224             }
225         }
226     }
227
228     CurrentBlock->Visited = true;
229     CurrentBlock->State = StackStatus;
230 }
231
232 return Changed;
233 }
234
235 std::unique_ptr<Pass> CreateStackAccessTSORemovalPass() {
236     return std::make_unique<StackAccessTSORemovalPass>();
237 }
238 } // namespace FEXCore::IR

```