



Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ
НА ТЕМУ:

«Метод трансляции машинного кода из x86 в ARM»

Студент группы **ИУ7-83Б**

(Подпись, дата)

М. Ю. Нитенко

(И.О. Фамилия)

Руководитель ВКР

(Подпись, дата)

А. А. Оленев

(И.О. Фамилия)

Нормоконтролер

(Подпись, дата)

Ю. В. Строганов

(И.О. Фамилия)

2022 г.

РЕФЕРАТ

Расчетно-пояснительная записка 25 с., 1 рис., 0 табл., X ист., X прил.

КЛЮЧЕВЫЕ СЛОВА

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	9
1 Аналитическая часть	10
1.1 QEMU	10
1.1.1 Сворачивание и оптимизация тривиальных выражений	10
1.1.2 Состояние процессора и блоки трансляции	11
1.1.3 Связывание блоков трансляции	11
1.1.4 Поддержка саомодифицирующегося кода	12
1.1.5 Эмуляция MMU	12
1.2 box64	12
1.2.1 Трансляция	12
1.2.2 Поддержка родных библиотек	13
1.2.3 Поддержка саомодифицирующегося кода	15
1.3 FEX	15
1.4 Frontend	15
1.5 OpDispatcher	16
1.5.1 IR SSA	16
1.6 IR/Passes	17
1.6.1 Распространение констант (ConstProp)	17
1.6.2 Устранение мертвого кода (DeadCodeElimination)	17
1.6.3 Устранение загрузок контекста (DeadContextStoreElimination)	17
1.6.4 Устранение хранения (DeadStoreElimination)	18
1.6.5 Сжатие инструкций (IRCompaction)	18
1.6.6 Long divide removal pass	18
1.6.7 RedundantFlagCalculationElimination	18
1.6.8 Static register allocation	18
1.6.9 Устранение временных регистров	18
1.6.10 Block chaining	19
1.7 Трансляция IR в инструкции хоста, JIT	19
1.8 Поддержка саомодифицирующегося кода.	19
2 Конструкторская часть	20

3	Технологическая часть	21
4	Исследовательская часть	22
	ЗАКЛЮЧЕНИЕ	23
	СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	24
	ПРИЛОЖЕНИЕ А	25

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

ВВЕДЕНИЕ

Процессоры архитектуры ARM занимают большую долю рынка, еще в 2015 году они составляли 35% от рынка процессоров, однако в основном они использовались в портативных устройствах [1]. С появлением процессоров M1 от компании Apple большое число людей начало пользоваться компьютерами на основе архитектуры ARM в домашней обстановке (типа personal computers). Однако, программы собранные под архитектуру x86 не смогут работать на таких компьютерах, им необходим или транслятор, такой как Rosetta 2, или виртуальная машина поддерживающая необходимую архитектуру.

Виртуальные машины как правило используют динамическую трансляцию, поэтому они намного более чувствительны к ее скорости.

1 Аналитическая часть

Проблемы эмулятора:

- управление кэшем транслированного кода;
- выделение регистров;
- оптимизация условных блоков;
- direct block chaining???? а по русски;
- управление памятью;
- поддержка самоизменяемого кода;
- поддержка исключений;
- поддержка аппаратных прерываний.

1.1 QEMU

кему это круто написать что это такое [2]

1.1.1 Сворачивание и оптимизация тривиальных выражений

В qemu определены свое промежуточное представление называющееся TCG ops, эти операции сначала оптимизируются, а затем выполняются на хосте, таким образом проще портировать TCG на разные платформы. Операции поддерживаются над 32 и 64 битными целыми числами, указатели определены как алиас над соответствующим целым числом.

Пример операции TCG:

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

Одной из оптимизаций является игнорирование бессмысленных операций, например операция:

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

выполняться не будет

подавляются неиспользуемые перемещения данных, например из трех операций:

```
add_i32 t0, t1, t2
```

```
add_i32 t0, t0, $1
```

```
mov_i32 t0, $1
```

выполнится только последняя.

1.1.2 Состояние процессора и блоки трансляции

Блоком трансляции называется часть кода до момента изменения состояния процессора которое нельзя выяснить на этапе трансляции (например, некоторое ветвление).

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).

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

1.1.3 Связывание блоков трансляции

После выполнения 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.

Еще одной оптимизацией является оптимизация ветвления. Если ветвле-

ние происходит напрямую, в пределах одной страницы QEMU выполняет поиск блока трансляции на который будет произведено ветвление, а затем сохраняет его адрес в транслированном коде. Таким образом при следующем выполнении этого блока нет необходимости в поиске следующего блока.

1.1.4 Поддержка самомодифицирующегося кода

Самомодифицирующийся код на x86 представляет особую проблему, так как нет механизма оповещения о изменении кода. 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, обнуляются все транслированные страницы и связь блоков. При запуске эмуляции системы программный MMU выполняет защиту от записи. Для эффективного выполнения этих операций хранится связанный список всех блоков трансляции и связей блоков.

(вообще говоря это наверное не нужно)

1.1.5 Эмуляция MMU

Каждый доступ к памяти проходит через MMU, в нем адреса преобразуются из виртуальных к реальным. Для ускорения трансляции адреса хранятся TLB-кэш.

Все блоки трансляции в кэше индексируются при помощи их физического адреса, таким образом нет необходимости очищать кэш при смене страницы. [3]

(вообще говоря это вроде не нужно (ну или нужно я не знаю!!))

1.2 box64

box64 это норм написать что такое

В отличии от QEMU box64 не может эмулировать полную систему (то есть на нем нельзя, например, запустить Windows), однако можно запускать linux-приложения собранные под x86_64.

1.2.1 Трансляция

Трансляция в box64 производится при помощи dynarec или при помощи эмулятора процессора. Для ARM реализован полноценный dynarec, для всех

остальных архитектур написан общий эмулятор процессора интерпретирующий инструкции. Dynarec написан на C, на ассемблере реализованы пролог и эпилог.

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 какое-либо ветвление из этого блока. Исключения: многобайтовые NOP обрабатываются отдельно, а неизвестная инструкция останавливает процесс выполнения блока.

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, не нужные 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. [4]

1.2.2 Поддержка родных библиотек

Главным механизмом позволяющим добиться хорошей производительности является использование родных для архитектуры библиотек. Таким образом 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 (графического приложения с большим количеством повторно используемых функций) производительность была около 85%.

Необходимые для работы приложения библиотеки либо транслируются, либо используется родная библиотека (в таком случае производительность выше). Вызовы функций перехватываются для вызова родных функций.

В ELF содержатся специальные символы называемые перемещениями (relocations) они используются при линковке для установки адреса необходимой функции. При вызове родной функции в качестве адреса выставляется адрес заранее подготовленного кода, он состоит из 5 байт и 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 вручную, 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 1.

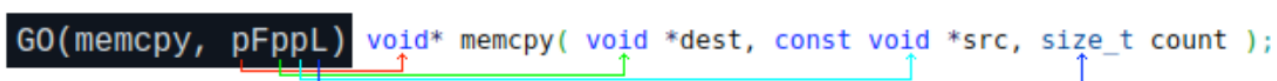


Рисунок 1 – Пример сигнатуры функции `memcpy`

После определения функция заносится в таблицу функций библиотеки, а затем находится при помощи разных методов. (// посмотреть что за методы (а надо?)).

Одной из оптимизаций в `box64`, по сравнению с `box86` (32-х битной версией эмулятора), является определение простых функций, такие функции не нуждаются в оберточной функции и вызываются напрямую, тем самым эконо-

мя время. [5]

1.2.3 Поддержка саомодифицирующегося кода

I'm doing something similar, yes. All pages from which x86 code has been translated are write protected. Once a write occurs, all blocks generated from this page are marked dirty, and the jump table from those blocks are invalidated (to avoid automatic jump from block to block to this one). Once x86 code wants to run a "dirty" block, a crc32 of the memory is runned and compared to the crc computed when creating the block. depending if it's the same or non, the block is marked as clean (and the page protected again) or deleted and a new one is generated (and the page also protected again)

1.3 FEX

FEX это еще круче чем box и почему.

FEX, так же как и box64, является эмулятором пользовательского режима.

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.

1.4 Frontend

Инструкции x86 декодируются в более простые для обработки, например инструкция add имеет большое количество разных опкодов, хотя по своей сути там меняются размеры и типы операндов, а не сама операция. Тогда операции add декодируются как:

```
00 C0: add al ,al
```

```
04 01: add al, 0x1
```

таким образом попадая в одну категорию.

Именно Frontend разбивает код на блоки трансляции. Рассматривается ситуация когда блоки трансляции находятся в одном месте. Некоторые трансляторы заканчивают трансляцию при любой потере управления, хотя в скомпилированном коде часто встречаются такие конструкции:

```
test eax, eax
```

```
jne .Continue
```

```
ret <--- Можно продолжать трансляцию после безусловной потери  
управления
```

```
.Continue:
```

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

1.5 OpDispatcher

Внутри FEX используется промежуточное представление, эти команды напоминают ARM64. Так как набор различных инструкций x86 слишком велик (более 1000) предлагается сначала переводить их в IR, а затем оптимизировать. На этом этапе обрабатываются особенности x86 (например MOVSB и подобные инструкции разворачиваются в циклы). OpDispatcher не выделяет регистры.

1.5.1 IR SSA

SSA (англ. Static single assignment form) — промежуточное представление, используемое компиляторами, в котором каждой переменной значение присваивается лишь единожды. Переменные исходной программы разбиваются на версии, обычно с помощью добавления суффикса, таким образом, что каждое присваивание осуществляется уникальной версии переменной. Использование SSA облегчает оптимизацию кода, пример по устранению бессмысленных присвоений:

```
y := 1
```

```
y := 2
```

$x := y$

Не сложно понять что первое присвоение переменной не нужно, однако для программы это далеко не так очевидно, при использовании SSA:

$y1 := 1$

$y2 := 2$

$x1 := y2$

это проще отследить.

В целом SSA помогает с:

- свёрткой констант;
- удалением мёртвого кода;
- частичным устранением избыточности;
- снижением стоимости операций;
- распределением регистров.

После трансляции в IR количество инструкций больше изначального в 10-30 раз. 32-х битные инструкции расширяются до 64 бит.

1.6 IR/Passes

На этом шаге применяются разные методики для оптимизации инструкций.

1.6.1 Распространение констант (ConstProp)

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

1.6.2 Устранение мертвого кода (DeadCodeElimination)

Устраняется бессмысленный, не вызываемый код.

1.6.3 Устранение загрузок контекста (DeadContextStoreElimination)

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

1.6.4 Устранение хранения (DeadStoreElimination)

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), устраняются бессмысленные и ненужные операции, так же как в QEMU.

Еще пример, если после ветвления в любом случае перезаписывается какое-либо значение — его можно не хранить.

1.6.5 Сжатие инструкций (IRCompaction)

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.

1.6.6 Long divide removal pass

Посоветоваться. (там шок может мне ашот расскажет)

1.6.7 RedundantFlagCalculationElimination

1.6.8 Static register allocation

На ARM можно использовать регистровый файл, если все правильно определить регистры можно заменять store context на store registers (уточнить (может это вообще не оптимизация)).

1.6.9 Устранение временных регистров

Если транслируется блок который включает в себя весь код функции и известен используемый двоичный интерфейс можно 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.

1.6.10 Block chaining

Та же идея что и в qemu, таким образом сильно снижается время поиска следующего блока при безусловном ветвлении.

1.7 Трансляция IR в инструкции хоста, JIT

При трансляции также используется кэш, для каждого потока кэш свой что приводит к избыточности, зато нет проблемы синхронизации. (это шок вообще мне надо писать про организацию кэшей в FEX?) И еще есть LookupCache, это адреса? (вроде) (External/FEXCore/Source/Interface/Core/LookupCache.cpp, используется в Arm64Dispatcher.cpp)

Третий уровень используется для реконструкции второго уровня в случае переполнения. Первый уровень не переполняется (разработчик так говорит).

1.8 Поддержка саомодифицирующегося кода.

Полноценной поддержки нет, автор считает что подход QEMU и box64 является единственным приемлемым по скорости и хочет использовать его.

2 Конструкторская часть

3 Технологическая часть

4 Исследовательская часть

ЗАКЛЮЧЕНИЕ

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Annual Report 2015: Strategic Report [Электронный ресурс]. – Режим доступа: https://media.corporate-ir.net/media_files/IROL/19/197211/2016CustomWork/ARM_Strategic_Report.pdf, свободный – (24.11.2021) (преза арма как оформить??)
2. Bellard F. QEMU, a Fast and Portable Dynamic Translator [Текст] / Bellard F. // FREENIX Track: 2005 USENIX Annual Technical Conference. – 2005. – С. 41-42.
3. QEMU: Translator Internals [Электронный ресурс]. – Режим доступа: <https://qemu.readthedocs.io/en/latest/devel/tcg.html>, свободный – (11.12.2021)
4. 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)
5. 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)

ПРИЛОЖЕНИЕ А