РЕФЕРАТ

Расчетно-пояснительная записка 33 с., 2 рис., 2 табл., X ист., X прил.

КЛЮЧЕВЫЕ СЛОВА

СОДЕРЖАНИЕ

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

Процессоры архитектуры ARM занимают большую долю рынка, в 2019 году они составляли 34% от рынка процессоров, занимая 90% рынка мобильных процессоров [4]. С появлением процессоров M1 от компании Apple большое число людей начало пользоваться компьютерами на основе архитектуры ARM в качестве персональных компьютеров.

Программы собранные под архитектуру x86 не смогут работать на таких компьютерах, им необходим или транслятор, например Rosetta 2, или виртуальная машина поддерживающая необходимую архитектуру. Rosetta 2 не может транслировать программы не предназначенные для macOS, для запуска программ созданных для Windows или Linux необходимо использовать виртуальную машину. Еще одним ограничением статической трансляции является наличие самомодифицирующегося кода и динамических библиотек, таким образом использование только статической трансляции не сможет запустить любую программу. [5]

Виртуальные машины в основном используют динамическую трансляцию, поэтому они более чувствительны к ее скорости.

Цель данной работы – провести обзор методов применяемых в трансляции машинного кода на x86 в машинный код архитектуры ARM.

1 Аналитическая часть

1.1 Постановка задачи?

Трансляция машинных кодов решает задачу запуска программ собранных под иную, по отношению к запускающей машине, архитектуру. При динамической трансляции существуют следующие проблемы:

— выделение регистров;

— поддержка самоизменяемого кода;

— корректная трансляция;

— быстрая трансляция;

— сохранение скорости выполнения кода.

Последние два пункта при динамической трансляции представляют собой смежные проблемы. Действительно, если применить все возможные оптимизации оптимизирующего компилятора LLVM к блоку трансляции этот блок будет выполнятся на скорости близкой к скорости выполнения кода собранного под необходимую архитектуру. Однако проходы оптимизации займут большое время, что в итоге приведет к замедлению выполнения кода, нахождение этого баланса и является главной проблемой динамической трансляции.

1.2 Методы трансляции

нам надо с лешей какой-то прикол на защиту придумать

write

1.2.1 Интерпретация

write

1.2.2 Рекомпиляция

write

1.2.3 Эмуляция высокого уровня

write

1.2.4 Сравнение методов трансляции

write

1.3 Динамические трансляторы

В данном разделе рассматриваются динамические трансляторы QEMU, box64 и FEX.

1.3.1 QEMU

move

1.3.2 box64

move

1.3.3 FEX

move

дальше не по структуре

QEMU поддерживает эмуляцию как пользовательского режима, так и системы. box64 и FEX являются эмуляторами пользовательского режима, то есть в отличие от QEMU box64 и FEX не могут эмулировать полную систему, однако могут запускать linux-приложения собранные под x86_64.

Все рассматриваемые проекты поддерживают динамическую трансляцию для архитектуры ARM. Два из них — QEMU и FEX — поддерживают промежуточное представление. FEX поддерживает не оптимизированную трансляцию в x86 (для отладки) и оптимизированную трансляцию в ARM. box64 включает в себя эмулятор, написанный на C который можно воспринимать как интерпретатор, используемый при запуске на любой архитектуре кроме ARM.

В box64 не используется промежуточное представление, таким образом при динамической рекомпиляции поддерживаемые инструкции, при помощи макросов C, транслируются в инструкции ARM. [6]

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 дру-

гом. В случае если адрес находится в другом блоке его необходимо связать с текущим. Также рассчитываются флаги процессора, 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. [7]

В QEMU 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 разные платформы. Операции поддерживаются над 32 и 64 битными целыми числами, указатели определены как псевдоним над соответствующим целым числом. [8]

Пример операции TCG представлен на листинге 1:

Листинг 1: Пример операции TCG

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

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 разных опкодов, хотя по своей 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

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

Затем в качестве промежуточного представления используются СЕП инструкции напоминающие ARM. Использование СЕП облегчает оптимизацию кода, например, при устранении бессмысленных присвоений, которые показаны в листинге 3:

Листинг 3: Пример лишнего присваивания

```
y := 1
y := 2
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.

1.3.4 Сравнение динамических трансляторов

write

1.4 Существующие оптимизации трансляции

write

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). [9]

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 какое-либо ветвление из

этого блока. Исключением являются многобайтовые NOP инструкции, которые обрабатываются отдельно, а прочтение неизвестной инструкции останавливает процесс выполнения блока. [6]

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 любой 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. [10]

Одной из важных отпимизаций 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, прой-

дя 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. [9]

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. [11]

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 специфичных 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 retn.

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 1.

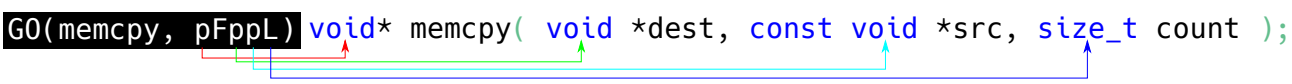

Рисунок 1 – Пример сигнатуры функции memcpy

После определения функция заносится в таблицу функций библиотеки, а затем находится при помощи разных методов. [13]

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 8 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 выполнятся 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 1 – 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 самомодифицирующегося кода

Самомодифицирующийся 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 систе-

мы программный MMU выполняет защиту от записи и инвалидацию страниц. [9]

В box64 используется похожий механизм, все транслированные страницы помечаются защищенными на запись, при попытке 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. [6]

В FEX не реализована полноценная поддержка самомодифицирующегося кода, по мнению разработчиков подход QEMU и box64 является единственным приемлемым по скорости и планируется использовать его. [12]

1.6 Поддержка TSO

При трансляции между бинарного кода между различными платформами следует также обращать внимание на разные модели доступа к памяти, их делят на две группы: слабую и сильную. К слабой модели относятся такие архитектуры как ARM, PowerPC, RISC-V; к сильной относятся, например, x86/64 и специальные режимы работы процессоров архитектуры SPARC и RISC-V. В таблице **??** представлено сравнение аспектов оптимизаций доступа к памяти на стадии выполнения.

Таблица 2 – Таблица переупорядочиваний обращений к памяти [14]

	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 может привести к некорректным результатам.

Простейшим решением данной проблемы является использование инструкций с барьером, данные инструкции устанавливают строгую последовательность между обращениями к памяти до и после барьера. Это означает, что все обращения к памяти перед барьером будут гарантированно выполнены 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.

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

Листинг 8: Простая программа

```
^^I^^I.globl _start
^^I^^I_start:
^^I^^Imov     x0, 58368
^^I^^Imovk    x0, 0x540b, lsl 16
^^I^^Imovk    x0, 0x2, lsl 32
^^I^^Iloop:
^^I^^Imov     x21, #0xfffffffffffffec
^^I^^Iadd     x21, sp, x21
^^I^^Istr     w20, [x21]
^^I^^I
^^I^^Isub     x0, x0, #1
^^I^^Icmp     x0, 0
^^I^^Ibne     loop
^^I^^I
^^I^^Imov     x0, #0
```

```
16    ^^I^^Imov     w8, #93
17    ^^I^^Isvc     #0
```

Данная программа 10000000000 раз производит сохранение регистра w20 в область памяти стека. Таким образом на время выполнения программы в основном влияет операция `str w20, [x21]`. Проведем 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 2.

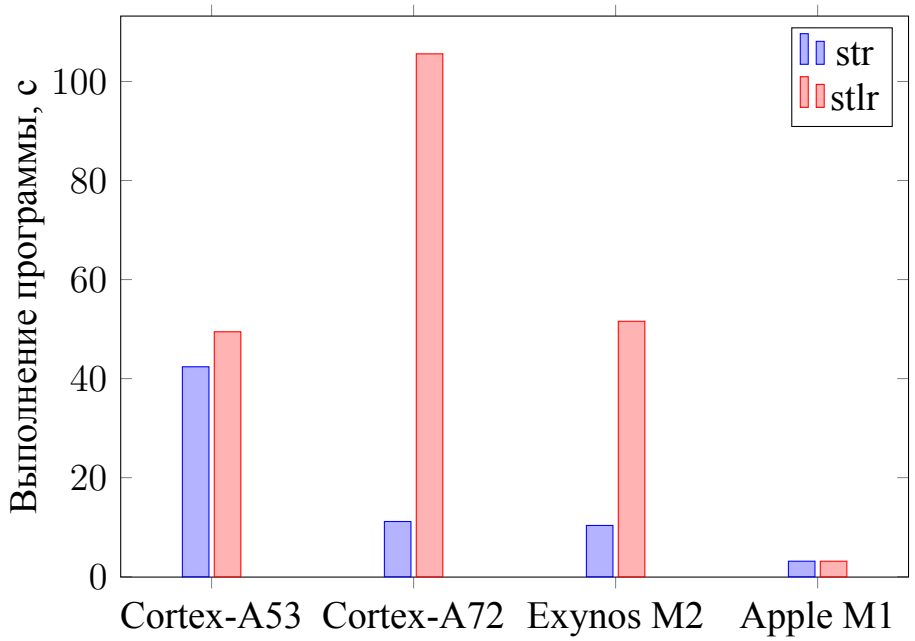

Рисунок 2 – Пример рисунка

На рисунке заметно что ядро Cortex-A53 практически не выполняло перестановки, это связано с тем что оно было выпущено 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 следует заметить 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 соответственной. Для процессора Apple M1 время выполнения именно 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, что является простым, но не самым производительным решением.

1.7 Оценка скорости работы трансляторов

write

1.8 Вывод

В данном разделе были рассмотрены некоторые методы трансляции применяемые в среде открытого программного обеспечения. По результатам анализа FEX является наиболее полным решением для трансляции кода, однако его производительность является самой низкой из рассмотренных трансляторов.

2 Конструкторская часть

В данном разделе описываются используемые структуры данных, проводится подробное описание алгоритма (алгоритмов? а если я хочу написать про те что я не реализовал еще ??)) оптимизации динамической трансляции доступа к памяти. Проводится проектирование программной реализации алгоритма оптимизации динамической трансляции доступа к памяти.

2.1 Архитектура программного обеспечения

а написать про дампы + скрипт для дампов, объясняю как проблемы решал?

2.2 Используемые структуры данных

2.3 Алгоритм оптимизации динамической трансляции доступа к памяти

2.4 Вывод

3 Технологическая часть

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

3.1 Выбор средств разработки

3.1.1 Выбор языка программирования

3.1.2 Сборка программного обеспечения?

я лично ничего не организовывал, так что не пишем? или пишем как я в смаке прописал свой файл

3.2 Требования к вычислительной системе

поставьте фекс с зависимостями

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. Иллингуорта 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. Переписка с разработчиком box64. См. приложение №1.

7. 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)

8. tcg/README [Электронный ресурс]. – Режим доступа: https://gitlab.com/qemu-project/qemu/-/blob/master/tcg/README, свободный – (26.12.2021)

9. QEMU: Translator Internals [Электронный ресурс]. – Режим доступа: https://qemu.readthedocs.io/en/latest/devel/tcg.html, свободный – (11.12.2021)

10. FEXCore Frontend [Электронный ресурс]. – Режим доступа: https://github.com/FEX-Emu/FEX/blob/main/External/FEXCore/docs/Frontend.md, свободный – (26.12.2021)

11. FEX-Emu: Internals in High Level @ ungleich.ch [Электронный ресурс]. – Режим доступа: https://www.youtube.com/watch?v=BYIIwaHfl3E, свободный – (26.12.2021)

12. Переписка с разработчиком FEX. См. приложение №2.

13. 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)

14. 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

15. 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

Hi. Thanks for your interest in Box! I have answered your question below, to make things easier to read.
As a general design principle for Box, I try to get the Dynarec not too complex, but still trying to have decent translated code. Also, I try to be able to get back to an x86 instruction from an ARM generated code to be able to handle Signal properly.

Sebastien.

Le mer. 8 d=C3=A9c. 2021 =C3=A0 13:34, Mikhail Nitenko <mnitenko@gmail.com>= a =C3=A9crit :

> Hello!
>
> I=E2=80=99m a final-year student interested in dynamic binary translation=
,
> specifically in x86 to ARM. I am writing a bachelor thesis about it
> and wanted to explore the current optimizations related to
> translation and wanted to write to you since your project seems to be
> quite advanced.
>
> I read the =C2=ABInner workings=C2=BB and =C2=ABA deep dive into library =
wrapping=C2=BB
> articles and wondered about some things, specifically:
> * How do you determine where the translation block ends? You said
> =C2=ABthere are a lot of possibilities here=C2=BB, from reading QEMU docs=
I
> would imagine that one of them would be a CPU state change that is
> impossible to determine ahead of time.
>
A block is a contiguous array of x86 instructions. A block ends when the last instruction has no "next" instruction (like a jump, call or ret) , and when that instruction is not referenced by a jump from the current block. There are a few exception: multi-bytes NOP are handled, and an unknown instruction stop the block unconditionally

* How much memory is allocated for each x86 instruction during pass 0?
> Is there some kind of table that determines that?

>
The memory is dynamically allocated in pass 0. So it will use as much memory as needed.

* JITs (mono) and self-modifying code. I know that emulating
> self-modifying code in x86 is difficult, QEMU for example detects
> writes to pages and invalidates translated code at that page and all
> pages that follow, are you doing the same? Is there some document that
> I can read to learn about problems related to JIT (mono, which I
> assume is open-source .NET)?
>
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)

* Am I correct in understanding that dynarec is used to translate x86
> instructions into ARM instructions and x86 emulation is code in C that
> emulates an instruction?
>
Box86 and Box64 include an Interpreter, coded in C, that can interpret x86 code on any architecture, and a Dynarec, actually only available for ARM. The Dynarec is coded in C also, with very little assembly file (just the prolog / epilog of a function call basically). The Dynarec will use "Emitter", (ab)using C macro, to generate actual arm opcodes.

>
>
> Also, maybe you would be able to point me to some other resource
> where I learn more about translation optimizations? Any help would be
> greatly appreciated!
>
You should have a look at FEX project. This project has a huge emphasis on the dynarec and code translation optimisation

ПРИЛОЖЕНИЕ Б

Oh, hey there.

For your specific questions. The pass manager is purely a construct that manages some pass classes and ensures optimization passes are run over the IR in correct order.
This can be found here: https://github.com/FEX-Emu/FEX/blob/main/External/FEXCore/Source/Interface/IR/PassManager.cpp
The same equivalent can be found in other compiler projects like LLVM.

Self-modifying code is currently semi-nonworking under FEX.
We have three modes of operation:
No SMC - Assumes zero code invalidation, highly dangerous.
mmap based invalidation - On mmap and munmap, ensure any overlapping executable regions get code invalidated. Fixes issues where libraries are loaded and need invalidation
per-instruction validation - This is the worst case situation, we emit an instruction check at every.single. emulated instruction. Very /very/ slow, only for debugging purposes.

We have plans to implement write protecting based invalidation but haven't had the time to get around to it. It's pretty much the only good way of detecting SMC.

The full list of passes can be seen in the PassManager link with `AddDefaultPasses`, `AddDefaultValidationPasses`, and `InsertRegisterAllocationPass`.
With implementations living at: https://github.com/FEX-Emu/FEX/tree/main/External/FEXCore/Source/Interface/IR/Passes

Unlike a compiler, we've got some heavy deadline constraints for how long we can take optimizing the code, and luckily the code will have run through a compiler that does most of that anyway.
Some of the more "hidden" optimizations come from around the IR optimization. Our IR is designed to look like AArch64 to some extent, which means we don't really need something like a high level IR then a machine level IR to get close to "optimal".

Additionally some of the memory allocations and container functions around the IR optimize around code size limits, forward only temporary allocator for smart CPU cache behaviour, linked lists that the elements are packed tightly in most cases so forward data fetching that the CPU does will be optimal.
Probably some other various bits.

I should also say that I'm not fully happy about FEX's codegen yet either. There's a /lot/ of room for improvement since we miss a bunch of optimization opportunities.
One of the big things that can help is our AOT compilation. We have some minor work towards both x86->IR AOT and I'm in the process of writing IR->Code AOT.
With AOT we can throw heavier optimizations at the IR that we can't do in JIT time. Which will improve long term performance of the application.

Currently our code generates a /bunch/ of redundant register moves for example. ARM CPUs that have really good renaming support will be inordinately faster than expected.

Some of the things to learn more about translation optimizations mostly looks like reading compiler optimization techniques and picking optimizations that don't take quadratic time complexity like that.
So anything compiler theory can help out JITs. There are of course domain specific optimizations, which are what would help JITs the most.
For applications using x87, you can optimize that to not use a softfloat emulation for example.