Министерство образования и науки Российской Федерации
МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ
(государственный университет)
ФАКУЛЬТЕТ УПРАВЛЕНИЯ И ПРИКЛАДНОЙ МАТЕМАТИКИ
КАФЕДРА ИНФОРМАТИКИ

Виртуализация исполнения машинного кода процессорной архитектуры ARM в Android-x86 окружении

Выпускная квалификационная работа
студента 176 группы
Лещёва Даниила Сергеевича

Научный руководитель
Тормасов А.Г., д. ф.-м. н., профессор

г. Долгопрудный

2017

Содержание

1. Введение

Данная работа является исследованием, относящимся к области виртуализации, а также архитектурам двух семейств процессоров (ARM и x86). В данной работе рассматривается проблема выполнения машинного кода, скомпилированного под архитектуру ARM на процессорах семейства x86, а также возможное решение с помощью технологии виртуализации.

В настоящее время виртуализация становится всё более и более популярной технологией, применяемой как организациями, так и отдельными пользователями. По своей сути виртуализация является предоставлением дополнительного абстрактного уровня между пользовательскими программами (как ОС, так и приложениями) и аппаратными структурами, на которых данные программы будут выполняться.

Существует огромное множество видов виртуализации, необходимо выделить те их них, которые позволяют исполнять машинный код на архитектуре, чужеродной по отношению к нему (ARM → x86). Их подробное описание будет дано в разделе ниже, а пока стоит сказать, что в качестве отправной точки была взята эмуляция.

Для того, чтобы исследование являлось актуальным, было решено в качестве прикладной части работы использовать реальную проблему, возникающую перед разработчиками ПО. Таковой была выбрана проблема переноса Android приложений, использующих нативный код, с ARM архитектуры на x86 без пересборки. Необходимо сказать, что в данный момент Android является одной из наиболее популярных мобильных ОС, что

влечёт за собой наличие большого числа разработчиков, выпускающих приложения под эту платформу.

Основным языком разработки под Android является Java. Это позволяет разработчикам не задумываться о кроссплатформенности, так как Java-код исполняется так называемой Java-машиной. До недавнего времени на платформе Android использовался Dalvik, но с приходом версии 5.0 его заменил ART. В то же время Java позволяет использовать при написании приложений нативные языки, в первую очередь C/C++ через JNI(Java Native Interface)[1].

На данный момент основная масса (>90%) Android-устройств работают на процессорах архитектуры ARM, однако появляются производители, такие как Lenovo, предоставляющие модели, работающие на процессорах архитектуры x86 [2].

Возникает проблема с проблема с приложениями, использующими нативный машинный код в своем составе, но более не поддерживаемые разработчиками. В большинстве своём это ресурсоемкие приложения, использующие, как правило, трехмерную графику – игры, CAD-пакеты и программы для работы с видео. Так как они не являются кроссплатформенными в отличие от чистых Java-приложений – их невозможно использовать на устройстве с чужеродной архитектурой процессора.

Однако данное ограничение можно обойти, если добавить слой виртуализации, который позволит выполнять машинный код ARM на семействе процессоров x86 без пересборки приложения. Это также позволит

выполнять отладку приложений, использующих нативный код непосредственно на рабочей станции разработчика, без необходимости в физическом устройстве для отладки.

2. Технологии виртуализации

Как уже было сказано выше, в данной работе наибольший интерес представляют те технологии виртуализации, которые позволяют исполнять машинный код ARM на x86, то есть способные предоставлять возможность использовать аппаратную платформу, не связанную архитектурно или как либо ещё с реальным железом, на котором происходит виртуализация. В настоящее время эта тема активно развивается посредством усилий различных компаний из сферы информационных технологий, также написано большое количество литературы, посвященной данному вопросу [3,4,5].

2.1 Существующие технологии виртуализации

В настоящее время среди подобных систем можно выделить три наиболее распространенных технологии [6]:

2.1.1. Эмуляция

Заключается в полном или частичном представлении необходимых аппаратных компонентов (процессор, память, устройства) в виде виртуальных объектов. Для исполнения машинного кода в таком окружении требуется пошаговый разбор и эмуляция инструкций на виртуальном процессоре.

Из плюсов подобного подхода можно выделить независимость от аппаратной начинки хостовой системы, а также широкий простор для кастомизации. К сожалению у такого подхода есть один существенный

минус – эмуляция крайне малопроизводительна, падение быстродействия обычно достигает нескольких порядков.

На сегодняшний день такой подход использует QEMU для целого ряда процессорных архитектур, которые не являются широко распространенными или же имеют сложную для трансляции структуру машинного кода.

2.1.2. Статическая бинарная трансляция

Подразумевает однократную полную трансляцию гостевого машинного кода в машинный код хостовой системы. Основной проблемой данного подхода является его сложность, так как без запуска программы невозможно, к примеру, заранее узнать наиболее часто используемые блоки инструкций, для того чтобы провести оптимизации.

Основным преимуществом такого подхода является скорость полученного кода, так как после трансляции весь код исполняется на реальной аппаратуры. К недостаткам можно отнести практическую сложность реализации.

В настоящий момент такой подход использует ART, однако в его случае происходить трансляция Java байт-кода в машинные инструкции.

2.1.3. Динамическая бинарная трансляция

Данный подход подразумевает трансляцию гостевого машинного кода в машинный код хостовой платформы на лету, во время исполнения. Для этого как правило используется эмулятор, который после единичного прохода по блоку машинных инструкций гостевого кода транслирует его в блок машинных инструкций хостовой системы, который впоследствии исполняется нативно.

Достоинством данного метода является быстродействие, которое

достигается за счёт того, что все или почти все инструкции гостевого кода в конечном счёте исполняются нативно. Также динамический подход позволяет собирать статистику исполняемого кода , выделять наиболее часто используемые участки кода для оптимизации, находить неиспользуемый код.

В данное время таким подходом пользуется отечественный микропроцессор «Эльбрус», предоставляющий таким образом полную совместимость с платформой x86. Также такой подход для отдельных участков кода использует QEMU.

2.2 Устройство QEMU

В данный момент QEMU является относительно быстрым машинным эмулятором, который использует оригинальный динамический транслятор. Он может эмулировать различные архитектуры процессоров (x86, PowerPC, ARM, Sparc), а также является кроссплатформенным. На данный момент QEMU поддерживает полную эмуляцию системы, что позволяет запускать операционную систему без модификаций на виртуальной машине. Также поддерживается так называемый user-mode в Linux окружении, когда процесс исполняемого файла, скомпилированного под одну процессорную архитектуру может быть запущен на другой.

Далее рассмотри подробно реализацию динамического транслятора, который использует QEMU [7]. Данный динамический транслятор позволяет осуществлять преобразование между гостевыми и хостовыми наборами машинных инструкций непосредственно во время исполнения программы. Далее полученный бинарный код хранится в кэше и может быть использован снова. По сравнения с интерпретированием такой подход позволяет

осуществлять выборку и декодирование лишь однажды, а не при каждом проходе по блоку исходного двоичного кода [19,20].

Обычно динамические трансляторы сложно переносить с одной процессорной архитектуры на другую, так как генератор кода должен быть полностью переписан при изменении набора машинных инструкций. Для того, чтобы избежать этого QEMU использует в качестве кодогенератора компилятор C, который представлен для множества процессорных архитектур. QEMU таким образом остается лишь собирать полученные блоки машинных инструкций воедино.

В первую очередь динамический транслятор в QEMU разделяет каждую процессорную инструкцию на некоторое количество более простых инструкций, так называемых микроопераций. Каждая операция реализована в виде небольшого блока кода на C. Далее этот блок компилируется в объектный файл. Количество микроопераций выбрано так, чтобы их общее число было небольшим, и в настоящий момент это количество не превышает двух сотен, что гораздо меньше по сравнению с классическими процессорными архитектурами (к примеру ARM насчитывает более 500 машинных инструкций).

Далее утилита под названием dyngen, используя в качестве входных данных объектный файл, содержащий микрооперации генерирует динамический генератор кода. Он будет вызываться во время работы гостевого кода для того, чтобы сгенерировать конкретную необходимую функцию, прежде разбитую на микрооперации.

Ключевой идеей в данном случае является то, что на момент трансляции не известны значения параметров, которые будут передаваться в функции. Таким образом объектные файлы, сгенерированные компилятором содержат указатели на места, где эти параметры будут использоваться. Это позволяет dyngen определить их и сформировать корректный динамический генератор кода, который в свою очередь в время работы гостевого кода подставив необходимые значения параметров и генерирует корректную последовательность машинных инструкций в архитектуре хостовой системы.

Утилита dyngen является ключевой для процесса трансляции в QEMU. Над каждым из объектных файлов с микрооперациями производятся следующие действия:

1. Объектный файл разбирается для получения таблицы символов, а также для определения вхождения всех параметров, значения которых неизвестны. Этот этап сильно зависит от типа исполняемого файла, а он в свою очередь от гостевой платформы (dyngen поддерживает форматы ELF (Linux), PE-COFF (Windows) и MACH-O (Mac OS X))

2. В объектном файле с помощью таблицы символов выделяется блок машинных инструкций, реализующий транслируемую функцию. Пролог и эпилог функции, как правило, опускается.

3. Далее для каждого параметра, найденного из таблицы символов объектного файла, формируется так называемая релокация – запись особого рода, которая будет заменена на адреса виртуальных регистров при выполнении этого блока кода.

4. Также для некоторых процессорных архитектур, таких как ARM, важно, чтобы параметры находились близко в адресном пространстве относительно кода, который их использует. Это происходит по той причине,

что код получает к ним доступ не по абсолютным адресам, а по относительному смещению счетчика команд

Рассмотрим подробнее как QEMU работает с оттранслированными блоками. При первом вхождении гостевого кода в функцию, эмулятор транслирует её, что позволяет сразу же выполнять инструкции нативно. Каждый такой блок помещается в кэш размером 16Мб, что позволяет далее быстрее находить и выполнять блоки нативного кода, которые используются часто.

Также QEMU использует статически фиксированный регистровый аллокатор. Это значит, что каждый регистр целевой процессорной архитектуры статически отображен на регистр или адрес хостовой системы. В большинстве случаев эмулятор просто отображает целевые регистры на память, так было сделано в угоду простоте и переносимости.

Далее, как только блок машинных инструкций целевой системы был выполнен, эмулятор обращается к виртуальному процессору для нахождения следующего адреса исполнения (например это необходимо когда следующая инструкция – это ветвление). Исходя из этого ищется и исполняется следующий транслированный блок гостевого кода.

3. Постановка задач

Цель работы – предоставить возможность исполнения Android-приложений, содержащих нативный код, скомпилированный под архитектуру ARM, на системах, базирующихся на архитектуре x86 под управлением Android.

Данная работа была выполнена на базе кроссплатформенного эмулятора системы на базе процессора с архитектурой ARMv7. Использованный эмулятор осуществлял виртуализацию не только виртуального процессора, но также памяти и устройств.

Однако для эффективного исполнения Android-приложений, использующие нативный код не требуется эмуляция всей системы. Был сделан вывод о необходимости реализации так называемой user-mode виртуализации – когда используется часть эмулятора, отвечающая за процессор. Она используется только для исполнения нативной части кода Android-приложения, основная часть которого в свою очередь исполняется хостовой Java-машиной.

Таким образом для достижения цели данной работы были выполнены следующие задачи:

1. Преобразование существующего эмулятора с целью позволить запуск отдельных программ, скомпилированных под ARM на процессоре с архитектурой x86. Это позволило удовлетворить цели работы в первом приближении, показать принципиальную возможность используемого подхода.

2. Написание загрузчика, необходимого для корректного запуска исполняемого файла, а также для его связывания (линковки) с внешними библиотеками. За счет данного этапа появилась возможность рассмотреть работу реальных приложений, так как в своей основной массе программы как правило используют внешние библиотеки. В качестве основной библиотеки, на которой производилось тестирование была выбрана стандартная

библиотека языка C, по причине ее широкой распространенности, а также присутствия на всех платформах.

3. Реализация оптимизаций в линкере, позволяющих вместо исполнения кода внешних библиотек через эмулятор сделать так называемый проброс вызовов в хостовую систему, с тем, чтобы эти вызовы исполнить нативно. Этот этап позволил избавиться от необходимости сопровождать исполняемый файл всеми ARM-версиями библиотек, от которых он зависит. Также благодаря этому стало возможным получить прирост в быстродействии.

Несмотря на то, что в целом данная работа выполнена на базе эмулятора, сходного по своей структуре с QEMU, имеется ряд важных отличий, что наделяет использованное решение самостоятельной ценностью:

1. В отличие от QEMU, заточенного под исполнение десктопного бинарного кода, используемый эмулятор изначально разрабатывался таким образом, чтобы иметь возможность быть встроенным в Android на уровне виртуальной машины.

2. Вторым серьёзным отличием является оптимизация, за счёт которой отпала необходимость в поддержании двух видов зависимостей (гостевых и хостовых), а также снизилось время ожидания при старте, так как стало возможным использовать стандартную, отлаженную и оптимизированную утилиту линковки, родную для хостовой системы.

4. Возможные подходы виртуализации

4.1 Структура эмулятора

Данный раздел посвящен общей структуре используемого эмулятора. Здесь представлены его составные части, дано краткое описание каждой из них. Также будет рассмотрен цикл эмуляции на примере одной из команд.

Главной частью пожалуй любого эмулятора является виртуальный процессор или vcpu (virtaul central processor unit). В данной реализации он также стоит во главе угла и состоит из нескольких составных частей:

1. Контекст vcpu.
2. Модуль fetch (выборки) и decode (декодирования) инструкций.
3. Набор функций-обработчиков машинных инструкций ARM

Рассмотрим каждую из приведенных составных частей подробнее.

Контекст vcpu.

В данной работе упор делался на эмуляцию архитектуры процессоров ARMv7, так как именно она на данный момент является наиболее востребованной на рынке мобильных устройств. Это позволяет очертить определенные рамки решаемой задачи, что позволяет упростить некоторые моменты, связанные с внутренней архитектурой, а также интерпретации системы команд.

ARM предоставляет 30 регистров общего назначения разрядностью 32 бит. В зависимости от режима и состояния процессора пользователь имеет доступ только к строго определенному набору регистров. В режиме ARM разработчику постоянно доступны 17 регистров (Рис. 1):

1. 13 регистров общего назначения (r0..r12).
2. Stack Pointer (r13) — содержит указатель стека выполняемой программы.

3. Link register (r14) — содержит адрес возврата в инструкциях ветвления.
4. Program Counter (r15) — биты [31:1] содержат адрес выполняемой инструкции.
5. Current Program Status Register (CPSR) — содержит флаги, описывающие текущее состояние процессора. Модифицируется при выполнении многих инструкций: логических, арифметических, инструкций ветвления и инструкций условного перехода.

Во всех режимах, кроме User mode и System mode, доступен также Saved Program Status Register (SPSR). После возникновения исключения регистр CPSR сохраняется в SPSR. Тем самым фиксируется состояние процессора (режим, состояние; флаги арифметических, логических операций, разрешения прерываний) на момент непосредственно перед прерыванием [8].

Рис. 1. Регистры ARMv7.

Таким образом контекст vcpu содержит в себе все необходимые банки регистров, также подмодули сопроцессоров, необходимые для вычислений с плавающей точкой и simd-инструкций (single instruction multiple data).

Модуль fetch (выборки) и decode (декодирования) инструкций.

Каждая инструкция машинного кода, поданная рассматриваемому эмулятору на вход, проходит так называемый конвейер, повторяющий в

упрощенном виде таковой конвейер исполнения в реальных процессорах. Он может быть представлен следующей краткой схемой:

Fetch → Decode → Emulation

Таким образом первым этапом исполнения машинной инструкции гостевого кода ARM становится её выборка. ARMv7 поддерживает гарвардскую архитектуру, то есть данные и код в программе хранятся раздельно [9]. Это значительно упрощает эмуляцию, позволяя не задумываться о том как интерпретировать входящую последовательность байт – как данные или инструкции.

Также следует отметить, что в настоящий момент архитектура ARM поддерживает несколько наборов команд:

1. ARM. Полноценный набор команд, практически все из них поддерживают условное выполнение, а также все команды имеют размер 4 байта и выровнены в памяти строго на 4 байта (Рис. 2).
2. Thumb. Укороченный набор инструкций, повышающий плотность кода и экономящий память. Размер инструкции равен 2-м байтам, которые выровнены также на 2 байта.
3. Thumb-2. Расширение Thumb. Он расширяет ограниченный 16-битный набор команд Thumb дополнительными 32-битными командами, что позволяет выполнять несколько Thumb инструкций с помощью Thumb-2 инструкции, чтобы повысить плотность кода [10]. Команды могут иметь размер как 4, так и 2 байта с выровнены на 2 байта.

Исходя из наличия различных наборов команд различного размера, на

этапе выборки эмулятору необходимо знать, в каком состоянии находится vcpu (ARM state/Thumb state). Эта информация хранится в CPSR регистре.

При полной аппаратной эмуляции на этапе выборки также проверяется, возможно ли вообще чтение и выполнение кода с используемой страницы памяти, однако в данной работе, как будет показано впоследствии, этот момент является несущественным, по причине того, что вся эмуляция производится в user-mode.

После выборки 2-х или 4-х байтная машинная инструкция попадает на этап декодирования. Декодер сопоставляет паттерн инструкции с функцией-обработчиком, который эту инструкцию будет эмулировать.

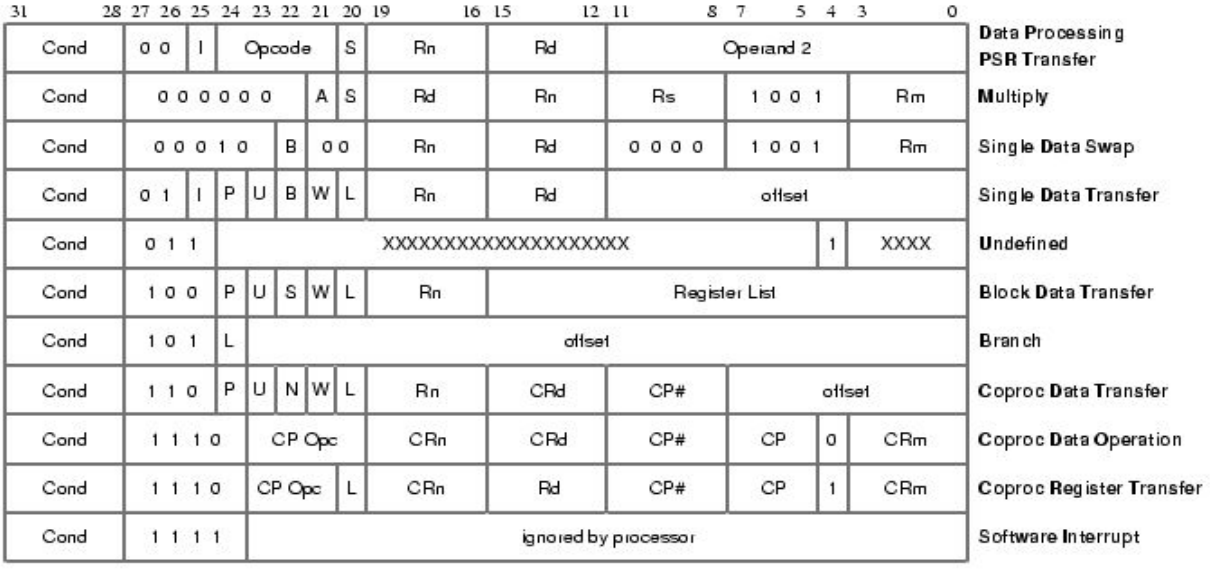

Рис. 2. Опкоды машинных инструкций ARM.

В текущей реализации декодинг реализуется за счёт так называемых таблиц декодинга – многоуровневых таблиц, позволяющий находить нужный обработчик за константное время. Это возможно благодаря тому, что в архитектуре ARM команды группируются по битовым маскам, что облегчает поиск (Рис. 2).

Далее декодированная машинная инструкция попадает в соответствующую ей функцию-обработчик. В общем случае эта функция:

1. Разбирает аргументы инструкции.

2. Эмулирует действие этой инструкции над vcpu.

3. Опционально – устанавливает необходимые флаги, меняет режим процессора.

4.2 User-mode эмуляция

Так как целью данной работы является предоставить возможность исполнения Android-приложений, содержащих нативный код, скомпилированный под архитектуру ARM, на системах, базирующихся на архитектуре x86 под управлением Android, то целесообразно было не использовать описанный выше полный цикл эмуляции, а оставить из него лишь часть, непосредственно необходимую для выполнения задачи.

В этой связи было решено отказаться от полной эмуляции Android-системы и использовать так называемую user-mode эмуляцию. Она подразумевает применение технологий виртуализации на уровне отдельного приложения, и без ненужной эмуляции устройств, памяти, сети, так как всё это уже и так есть в хостовой системе.

Таким образом задача существенно упрощается, так как можно ограничиться узким кругом необходимых инструментов. Например вместо полного виртуального представления процессора осталось необходимым поддерживать лишь режим пользователя (user mode), так как именно в этом режиме выполняются все прикладные программы. Это сократило количество

хранимых регистров, упростило операции с ними, так как больше не требовалось контролировать работу перекрывающихся банков с регистрами.

Также это позволило упростить операцию декодирования, сократив необходимый набор инструкций. Некоторые инструкции ARM могут быть выполнены только в привилегированном системном, супервизорном режиме или в режиме прерывания. Очевидно, что все эти режимы не требуются, когда речь идёт о выполнения программного кода в режиме пользователя.

Значительные изменение постигла также и подсистему памяти. Вместо прослойки, предоставляющей эмулятору честную систему виртуальной памяти было решено использовать виртуальную память хостовой ОС. Таким образом для гостевого кода ничего не изменилось – он всё так же работает с виртуальной памятью. Однако теперь нет необходимости заботится о выделении/утилизации страниц, а также об обеспечении корректного доступа к ним – вся эта работа выполняется хостовой ОС.

4.3 Загрузчик

Ещё одной крупной частью данной работы является реализация бинарного загрузчика, необходимого для корректного запуска гостевого машинного кода, представленного в виде объектных файлов. Остановимся на формате объектных файлов подробнее.

В настоящее время в среди Linux (а значит и Android) единственным используемым форматом бинарного кода является ELF (Executable and Linkable Format — формат исполнимых и компонуемых файлов) [11]. Файл такого формата состоит из нескольких частей (описания даны в виде

структур языка C, так как именно в таком виде с ними велась работа):

1. Заголовок (ELF header). Содержит в себе необходимую информацию об исполняемом файле

```
typedef struct
{
    unsigned char   e_ident[EI_NIDENT];     /* Сигнатура и прочая информация */
    Elf32_Half      e_type;                 /* Тип объектного файла */
    Elf32_Half      e_machine;  /* Аппаратная платформа (архитектура) */
    Elf32_Word      e_version;  /* Номер версии */
    Elf32_Addr      e_entry;    /* Адрес точки входа (стартовыйадрес программы) */
    Elf32_Off       e_phoff;    /* Смещение таблицы программных сегментов */
    Elf32_Off       e_shoff;    /* Смещение таблицы заголовков секций */
    Elf32_Word      e_flags;    /* Специфичные флаги процессора */
    Elf32_Half      e_ehsize;   /* Размер ELF-заголовка в байтах */
    Elf32_Half      e_phentsize;/* Размер записи в таблице программных сегментов */
    Elf32_Half      e_phnum;    /* Количество записей в таблице программных сегментов */
    Elf32_Half      e_shentsize;/* Размер записи в таблице заголовков секций */
    Elf32_Half      e_shnum;    /* Количество записей в таблице заголовков секций */
    Elf32_Half      e_shstrndx; /* Расположение сегмента, содержащего таблицу строк */
} Elf32_Ehdr;
```

2. Таблица программных заголовков (сегментов). Каждая запись этой таблицы содержит в себе информацию о соответствующем сегменте бинарного файла – его содержании, типе, а также том, каким образом он должен быть загружен в память.

```
typedef struct
{
```

```
  Elf32_Word p_type;            /* Тип программного сегмента */
  Elf32_Off    p_offset;     /* Смещение сегмента в файле */
  Elf32_Addr  p_vaddr;     /* Виртуальный адрес сегмента для загрузки */
  Elf32_Addr  p_paddr;     /* Физический адрес сегмента для загрузки */
  Elf32_Word p_filesz;     /* Размер сегмента в файле */
  Elf32_Word p_memsz;    /* Размер сегмента в памяти при загрузке */
  Elf32_Word p_flags;      /* Флаги и параметры сегмента */
  Elf32_Word p_align;      /* Выравнивание в памяти при загрузке */
} Elf32_Phdr;
```

3. Таблица секций. Каждый сегмент содержит в себе несколько секций исполняемого файла. Каждая секция бинарного файла хранит определенный вид данных.

```
typedef struct
{
  Elf32_Word sh_name;     /* Имя секции */
  Elf32_Word sh_type;     /* Тип секции */
  Elf32_Word sh_flags;    /* Флаги и параметры */
  Elf32_Addr  sh_addr;    /* Адрес в памяти */
  Elf32_Off    sh_offset;   /* Смещение в файле */
  Elf32_Word sh_size;     /* Размер в байтах */
  Elf32_Word sh_link;     /* Смещение связанной секции */
  Elf32_Word sh_info;     /* Информация */
  Elf32_Word sh_addralign;       /* Выравнивание адреса */
  Elf32_Word sh_entsize;  /* Размер одной записи */
} Elf32_Phdr;
```

В настоящее время нет чёткой стандартизации именования и содержания секций исполняемых файлов. Однако вот самые распространенные секции, встречающиеся в любой программе:

.text – исполняемый код программы. В архитектуре ARM данная секция может содержать исключительно инструкции процессора

.data – инициализированные данные программы (константы, строки и т.д.)

.bss – не инициализированные данные, заполняются нулями при загрузке

.plt – Procedure Linkage Table. Содержит код стабов вызовов внешних функций, адреса которых лежат в .got, и могут быть не известны на этапе компиляции (например если исполняемый файл использует динамические библиотеки)

.rel – таблица релокаций. список мест в секции .text, которые нужно изменить при связывании этого объектного файла с другими объектными файлами.

.got – Global Offset Table – глобальная таблица смещений. Содержит смещения всех гобальных переменных, находящихся за пределами данного объектного файла.

.symtab – таблица символов файла, содержит все внутренние и вснешние символы (глобальные переменные, функции).

.strtab – таблица строк файла, содержащая имена символов, функций, отладочную информацию и т.д.

Этот список далеко не полный, и всего лишь отражает наиболее распространенные секции, а также секции, с которыми пришлось непосредственно столкнуться при написании загрузчика.

Таким образом видно, что сам по себе формат исполняемых файлов ELF обладает сложной структурой, что влечёт за собой необходимость в определенной внимательности к деталям. Далее будет дано полное описание алгоритма загрузчика, который был реализован в ходе выполнения данной работы.

После инициализации vcpu, а также смежных с ним систем за дело принимается загрузчик, получивший путь к исполняемому файлу архитектуры ARM. В первую очередь читает ELF header переданного файла,

и проверяет его на валидность, а также удостоверяется, что загрузчику был передан файл, собранный под нужную архитектуру. Далее считывается таблица программных сегментов и подсчитывается количество страниц памяти, для размещения исполняемого файла в памяти. Так как это новый процесс, то имеется возможность удовлетворить требованиям к размещению программных сегментов по тем виртуальным адресам, что были указаны в ELF файле.

Затем после выделения памяти под все сегменты, подлежащих загрузке в память, начинается процесс их разбора. В первую очередь определяется тип программного сегмента. В случае, если это PT_LOAD, то есть сегмент, предназначенный для загрузки, производится так называемый маппинг (отображение) части файла, который его содержит на соответствующую ему область виртуальной памяти. Отдельно в данном случае обрабатывается сегмент данных, который как правило содержит секцию .bss – не инициализированные данные. Для экономии места в исполняемых файлах они хранятся лишь в виде обозначений размера, в виртуальной же памяти под эту секцию выделяется память, по умолчанию заполняемая нулями. Также при маппировании происходит учёт сдвигов виртуальных адресов, который происходит при выравнивании сегментов. В данном случае это критично, так как архитектура ARM требует, чтобы адреса машинных инструкций в памяти были выровнены на 2 или 4 байта.

В случае, когда типом сегмента является PT_DYNAMIC – это значит, что сегмент содержит информацию, необходимую для динамического связывания объектных файлов (например при динамической загрузке библиотек). Информация представлена в виде таблицы ссылок на

соответствующие секции исполняемого файла. Для дальнейшей работы сохраняется динамическая таблица символов и таблица строк.

После разбора всех программных сегментов начинается этап поиска и разрешения зависимостей. Для каждой записи в сегменте, отвечающем за динамическое связывание и имеющем тип DT_NEEDED сохраняем имя необходимой внешней библиотеки. Далее для каждой из библиотек, от которых зависит исходный исполняемый файл, необходимо рекурсивно вызвать процедуру линковки. Таким образом в первую очередь удовлетворяются самые глубокие зависимости, а при обратном ходе рекурсии все необходимые библиотеки будут подгружены и слинкованы.

На момент, когда в виртуальной памяти находятся все необходимые сегменты исходного исполняемого файла, а также библиотек, от которых он зависит, начинается процедура разрешения зависимостей. Начиная с самого глубокого уровня, загрузчик просматривает таблицу релокаций и удовлетворяет каждую из них.

Механизм релокаций в современных системах используется для удобного манипулирования библиотеками, а также возможностью связывать исполняемые файлы и библиотеки друг с другом не на этапе сборки, а во время запуска, что значительным образом сокращает объёмы запускаемых программ, так как более нет нужды заранее собирать исполняемый файл из всех использующихся в нём библиотек. Однако при таком подходе появляется существенная проблема – если в коде используется, к примеру, внешняя функция, то становится невозможным на этапе компиляции подставить в машинные инструкции корректный адрес, так как этот адрес не известен заранее. Более того, в настоящее время ради экономии ресурсов

программы, как правило, не используют копии общесистемных библиотек (таких, как libc), потому что это так же влечёт за собой дополнительные расходы памяти. Вместо этого операционная система предоставляет общий доступ к тем сегментам библиотеки, которые не могут быть модифицированы (сегмент кода), и предоставляет копию тех участков, что модифицированы могут быть (сегмент данных). Однако это достигается не на уровне загрузчика, а на уровне системы виртуальной памяти, и такой подход называется Copy-on-Write [12].

Таким образом при старте выполнения загрузчик должен найти все адреса внешних вызовов и переменных в программе, и заменить их на корректные адреса из внешних библиотек. Возможность этого достигается за счёт применения релокаций – каждая релокация содержит информацию о месте, куда необходимо подставить адрес неизвестного символа, его имя, а также формат, в котором следует подставить этот адрес. В настоящее время самым распространенным подходом является так называемый PIC (position-independent code), который подразумевает подстановку адресов не в абсолютных значениях, а в виде разницы между адресом и тем местом, откуда его будут вызывать. Это сделано для возможности безболезненного перемещения сегментов кода по виртуальной памяти при необходимости. В этой связи в данный момент вызов внешних функций в коде начинается с перехода в так называемый PLT-stub, находящийся в таблице связывания процедур, откуда уже происходит переход по адресу, определенному релокацией. Это было сделано для того, чтобы собрать все вызовы каждой внешней функции в одном месте, что позволило сократить время разрешения зависимостей на старте программы. Также для экономии времени применяется приём под названием lazy linking, когда релокация разрешается

непосредственно перед первым вызовом функции, однако эта оптимизация в текущей работе не была выполнена.

После разрешения всех зависимостей необходимо передать управление эмулятору, сообщив также адрес функции main, являющейся стандартной входной точкой. Для этого вместо таблицы сегментов подгружается таблица секций, а в ней ищется символьная таблица (SHT_SYMTAB). Далее в ней просматриваются все записи типа STT_FUNC и ищется необходимый символ. Получив её смещение и зная, по какому виртуальному адресу бы загружен соответствующий сегмент можно определить адрес необходимой функции. Закончив работу, загрузчик передаёт управление эмулятору, а тот в свою очередь начинает исполнение гостевого кода начиная с точки входа в программу

5. Оптимизация эмуляции

После реализации загрузчика встаёт вопрос о возможных оптимизациях полученной системы. Учитывая, что несмотря на различные архитектуры гостевой и хостовой платформ, обе они управляются одной и той же ОС можно сделать вывод, что вместо эмуляции окружения (в данном случае библиотек, динамически связываемых с исполняемым файлом), можно использовать уже существующие в системе библиотеки.

Это решение обладает следующими преимуществами:

1. Можно использовать стандартные средства связывания динамических библиотек, а также поиска необходимых символов в них. Это уменьшает количество ошибок, а также увеличивает скорость запуска исполняемого файла.

2. Благодаря использованию родных для системы библиотек стало возможным исполнять все вызовы к ним нативно. Таким образом эмулируется только выполнение кода исполняемого файла, все внешние вызовы исполняются непосредственно процессором.

Все вышеобозначенные цели были достигнуты путём модификации реализованного загрузчика. Основные изменение коснулись этапа поиска и разрешения зависимостей. Вместо рекурсивной линковки необходимых библиотек загрузчик ищет в таблице соответствия файл, соответствующий необходимой гостевой библиотеке и подгружет его с помощью стандартной функции dlopen. Далее обработчик загруженного файла помещается в кэш для предотвращения лишней повторной загрузки.

Далее на этапе разрешения зависимостей при поиске виртуального адреса очередного необходимого символа загрузчик обращается в кэш динамических библиотек, слинкованных с исходным исполняемым файлом, и ищет необходимый символ (функция dlsym). Найденный адрес заносится в специальную таблицу, а на место предполагаемой релокации помещается токен, соответствующий индексу в таблице символов. Так было сделано для того, чтобы эмулятор исполняя очередную команду мог понять, когда необходимо сделать нативный вызов.

Во время обработки каждой машинной инструкции гостевого кода эмулятор производит проверку на соответствие вида очередной инструкции паттерну токена. В случае, если это так, производится переход в код хостовый код соответствующей функции.

Рассмотрим подробнее нативный вызов из эмулируемого кода и

передачу аргументов в него. Проблема в данном случае состоит в том, что существуют разные соглашения о вызове (так называемые calling conventions) для различных архитектур. Так в случае x86 при 32-х битной адресации все параметры передаются через стек, справа-налево (то есть первый аргумент всегда оказывается на вершине стека) [13,14,15]. Так было сделано для удобства реализации функций с переменным числом аргументов (например printf из стандартной библиотеки C). В случае же архитектуры ARM первые 4 аргумента передаются через регистрые r0-r1, а остальные – таким же образом через стек. Так как в обоих случаях конвенция вызовов предполагает, что стек очищает вызывающая функция, то нет нужды беспокоиться о количестве передаваемых аргументов. На данном этапе реализации эмулятора во внешнюю библиотеку всегда передаётся 8 аргументов.

Таким образом, каждый раз, когда эмулятор встречает токен на месте очередной машинной инструкции, происходит выход в функцию нативного вызова, которая, в свою очередь по токену определяет адрес библиотечной функции, которую необходимо вызвать. Далее происходит процедура передачи пареметров через реальный стек. Для этого в буфер копируются значения первых 4-х виртуальных регистров, содержащихся в vcpu, а затем через указатель на виртуальный стек, находящийся там же происходит копирование в буфер ещё 4-х аргументов. После этого Весь буфер помещается в стек и происходит вызов внешней библиотечной функции. Затем значения регистров EAX и EBX, в которых, согласно соглашению о вызовах, лежат возвращаемые значения вызванной функции копируются в виртуальные регистры r0 и r1 соответственно. Происходит очищение реального стека. Таким образом на выходе из функции нативного вызова в соответствующих регистрах архитектуры ARM находится корректное возвращаемое значение, полученное после проброса библиотечной функции

в хостовую систему.

Стоит заметить, что предлагаемая оптимизация использует тот факт, что Android-приложения используют JNI как правило именно для вызовов нативных библиотек. Таким образом потери производительности, связанные с эмуляцией в реальных приложениях будут несущественными, поскольку сам код библиотеки, как было показано, будет исполняться непосредственно на процессоре.

6. Оценка быстродействия эмуляции

Важной частью представленной работы является оценка производительности предлагаемого подхода. Эта характеристика особенно важна для проверки жизнеспособности предлагаемого подхода в реальных условиях.

Для оценки скорости эмуляции машинных инструкций был выбран синтетический тест – вычисление n-го числа Фибоначчи рекурсивным методом. Такой алгоритм активно использует стек, вызовы функций, а также целочисленную арифметику. На Рис. 3 представлен график зависимости логарифма времени вычисления n первых чисел Фибоначчи в зависимости от n.

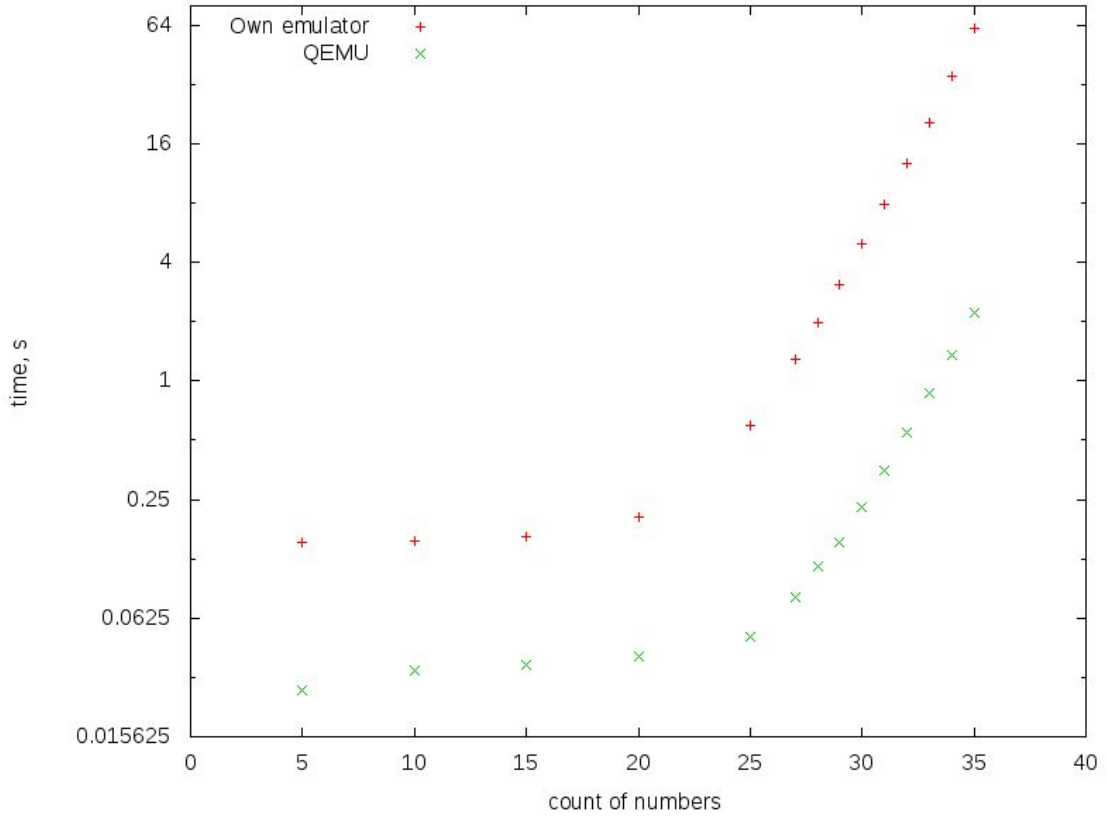

Рис. 3. График времени исполнения программы эмулятором и QEMU.

Кол-во чисел Фибоначчи	Время работы эмулятора, с	Время работы QEMU, с
5	0.152	0.027
10	0.155	0.034
15	0.161	0.036
20	0.202	0.042
25	0.594	0.051
27	1.295	0.082
28	1.975	0.114
29	3.091	0.151
30	4.937	0.229
31	7.868	0.351
32	12.6	0.548
33	20.437	0.863

34	34.875	1.341
35	61.263	2.208

Табл. 1. Время исполнения программы эмулятором и QEMU.

Из Табл. 1 видно, что в среднем эмулятор проигрывает QEMU примерно 1-2 порядка при исполнении машинного кода без библиотечных вызовов. Здесь надо принимать во внимание, что как уже было сказано выше, в основе своей работы QEMU использует динамическую трансляцию для отдельных участков кода, что позволяет ему большую часть времени выполнять машинные инструкции гостевого кода с помощью небольшого количества инструкций процессора хостовой системы, что влечёт за собой существенно меньшие накладные расходы по сравнению с полной эмуляцией каждой из этих инструкций.

Можно также заметить, что вплоть до 20-го числа Фибоначчи время исполнения программы остается практически константным, как для представленного в данной работе эмулятора, так и для QEMU. Это время, необходимое системе эмуляции для того, чтобы начать выполнение машинных инструкций гостевого кода. В случае представленного эмулятора можно считать, что это время, необходимое загрузчику для линковки входной программы и разрешения зависимостей. В данном случае это время порядка 0.15 секунд.

Однако, как уже было сказано выше, в реальных приложениях гостевой код в большинстве случаев будет представлять собой лишь прослойку между Java-кодом и кодом библиотек, код которых, с применением предлагаемой оптимизации, будет исполняться нативно. Для оценки производительности в этом случае используется программа, содержащая большое количество

вызовов функций для обработки строк из стандартной библиотеки языка C. Также в данном случае быстродействие сравнивается с версией программы, изначально скомпилированной под хостовую систему и запущенной на ней.

На Рис. 4 видно, что в случае, когда программа состоит в основном из библиотечных вызовов, быстродействие эмулятора отстаёт менее чем на порядок. По факту в приведенном тесте гостевая программа в эмуляторе выполняется в 2-3 раза медленнее, чем нативно. При таком уровне быстродействия можно говорить, что при проведении дальнейших оптимизаций данное решение пригодно для использования в десктопных эмуляторах Android, используемых разработчиками для отладки. Исходя из всего вышесказанного можно сделать вывод, что предлагаемый метод эмуляции можно использовать, однако не в качестве самостоятельного решения, а именно как тонкую прослойку, предоставляющую нативный интерфейс для Android-приложений.

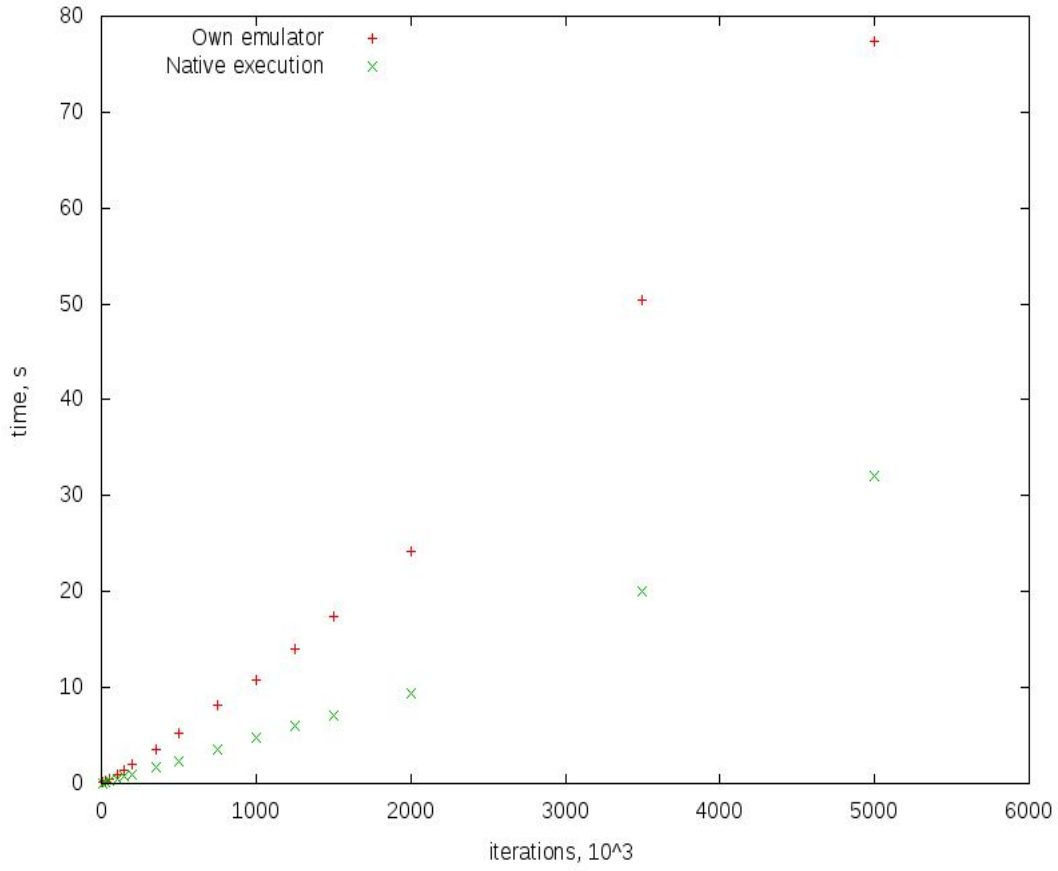

Рис. 4. График времени исполнения тестовой программы №2.

7. Бинарная трансляция

Как уже было показано ранее, первоначальный экспериментальных подход с эмуляцией промежуточного машинного кода мог работать с приемлемым быстродействием только в случае, если прослойка кода невелика и основную часть времени нативный код используется для вызова библиотечных функций, для которых существуют версии, скомпилированные под хостовую архитектуру, то есть под x86. В этому случае быстродействие кода приближается к нативному, ограничения наступают по большей части со стороны Java-машины.

Совсем иначе ведет себя производительность системы в случае, когда

целевое приложение совершает существенную работу в нативном контексте, но при этом без частого вызова внешних библиотек. В этом случае происходит пошаговая эмуляция выполнения инструкций машинного кода процессорной архитектуры ARM, что влечёт за собой существенную потерю производительности. Разберемся, почему так происходит.

В текущей реализации эмулятора после получения опкода происходит процедура декодирования инструкции, которая сопоставляет опкоду, либо классу опкодов инструкций структуру данных, необходимых для виртуализации исполнения данной инструкции. Проблема заключается в том, всего валидных инструкций в процессорной архитектуре ARMv7 несколько сотен, что влечёт за собой необходимость использовать некоторый контейнер, не нуждающийся в модификации (так как его содержимое фиксировано на всём протяжении работы программы), однако обеспечивающим быстрый (асимптотически константный) поиск среди опкодов на предмет нужного.

В текущей реализации в качестве такого контейнера была использована многоуровневая хеш-таблица, которая статически располагается в памяти. Однако из-за объёма набора инструкций возникла проблема с возникновением коллизий в текущей реализации этой хеш-таблицы. Вместо стандартного подхода для разрешения коллизий с помощью цепочек значений было решено использовать подход с иерархическими хеш-таблицами. Это решение с одной стороны обеспечивало схожую производительность, а с другой позволило семантически разделить пространство инструкций на классы. И только в таблицах декодирования нижнего уровня применялись цепочки значений для разрешения редких возникающих коллизий. Таким образом с программной точки зрения

структура контейнера, содержащего необходимые для виртуализации данные выглядит так:

```
typedef struct item {
   item_type_t utype;
   union {
      const instr_desc_t    *op;
      struct search_table   *tab;
      list_t                *list;
   } u;
} item_t;
```

Отсюда видно, что на каждом уровне таблицы декодирования может содержаться либо “лист” дерева, то есть непосредственно описатель инструкции, либо список таких описателей, среди которых ищется нужный в случае возникновения коллизии, либо хеш-таблицы более низкого уровня, в которой следует продолжить поиск.

Проблема при таком подходе заключается в том, что несмотря на то, что данные в таблицы заносятся статически на этапе компиляции, нет возможности повлиять на их расположение в памяти. Это ведёт к большому количеству переходов по указателям в память, расположенную далеко друг от друга. Самым значимым последствием такого поведения является большое количество промахов кэш-памяти, что влечёт за собой сильную просадку производительности, так как в данном случае от производительности процессора практически ничего не зависит - в хеш-таблице не применяются хеш-функции как таковые, вместо этого в качестве ключа используются семантически-значимые части опкодов машинных инструкций.

Таким образом хеширование в данном случае сводится к простому вычислению значения опкода по маске, что производится за считанные такты процессора, в то время как переход в память, которой нет на момент исполнения соответствующего участка кода может повлечь за собой простой процессора в течении сотен тактов, если произошёл так называемый промах кэша. Это ситуация, когда данные по запрашиваемому адресу отсутствуют в памяти и шине данных требуется существенное время, чтобы получить их из медленной (по сравнению с кэшем) оперативной памяти.

Вторая проблема состоит в несоответствии разделения опкодов машинных инструкций и частоты их встречаемости в реальных приложениях. Существующие компиляторы как правило используют характерные паттерны и наборы инструкций, что позволяет выделить подмножество опкодов, которые будут в среднем выполняться на порядки чаще остальных инструкций. Также существуют довольно крупные классы инструкций, которые могут вовсе не встретиться в приложении, и которые логично декодировать отдельно. К таким в первую очередь относятся векторные инструкции, а также инструкции для работы с числами с плавающей точкой. За счёт того, что как правило в целевом приложении они используются блоками из нескольких команд без переходов и ветвлений внутри них можно выделить декодирование инструкций такого класса в отдельную процедуру и использовать одноуровневую реализацию хеш-таблицы, минимизировав количество промахов кеша.

К сожалению невозможно доподлинно определить статистику по частоте встречаемости тех или иных инструкций в машинном коде реальных приложений, особенно учитывая, что NDK в общем случае позволяет

применять в коде вставки ассемблерного кода, что резко расширяет разнообразие применяемых машинных инструкций. Однако в целях оптимизации существующей таблицы декодирования было предложено собрать некоторую количественную статистику по частоте встречаемости инструкций. В качестве целевого приложения в этом эксперименте выступало ядро Linux от этапа загрузки и вплоть до приглашения командной строки пользователя (для этих целей применялся system-mode режим эмулятора).

С помощью полученной статистики была произведена следующая оптимизация. Вместо семантически обоснованной иерархической структуры хеш-таблицы было решено перейти к автоматически генерируемой структуре. В такой хеш-таблица наиболее часто встречающиеся в реальных приложениях опкоды инструкций должны располагаться на самом верхнем уровне, если представить хеш-таблицу как сильно ветвистое дерево. На основании собранной ранее информации о наиболее часто встречающихся инструкциях были сгенерированы маски опкодов инструкций, которые максимизировали количество часто встречающихся инструкций, для которых не требуется разрешения коллизий и при этом минимизировало количество коллизий в таблице верхнего уровня, а также количество инструкций с высоким рейтингом встречаемости, но попавшим в таблицы более низких уровней. Важно, что таблицы в данном случае генерируются автоматически и единственная необходимая информация — это набор инструкций, подлежащих декодированию, а также весовые коэффициенты их встречаемости. В конечном итоге такой подход привёл к тому, что в абсолютном большинстве случаев декодирование сводится к поиску по одногарговой хеш-таблице за константное время.

Рассмотрим подробнее структуру данных, которая хранится в таблице декодинга в качестве значений, соответствущим ключам-опкодам машинных инструкций.

```
typedef struct instr_description {
   uint32_t mask;
   uint32_t value;
   uint32_t nmask;
   uint32_t nvalue;
   const char *name;
   uint32_t weight;

   struct {
      emulation_handle_t execute;
      execute2_t execute2;

#if defined(USER_CFA)
      user_cfa_insn_flags_t user_cfa_flags;
      void *user_cfa_callback; // meaning depends on flags
#endif /* defined(USER_CFA) */

#if defined(USER_MODE)
      regs_usage_handle_t regs_usage_callback;
      translation_handle_t translator_callback;
#endif /* defined(USER_MODE) */
   };
```

Эта структура содержит все необходимые данные для виртуализации одной отдельно взятой инструкции, рассмотрим их последовательно. В первую очередь это так называемый “обработчик эмуляции” или emulation_handle_t execute, который по сути является обычной функцией со следующей сигнатурой:

```
exec_status_t execute(vcpu_t *vcpu, const struct instr_description *instr, op_t op);
```

Вобщем случае функция принимает на вход некоторый виртуальный контекст процессора, отражающий реальную процессорную архитектуру ARMv7, описание инструкции, для который в данный момент производится эмуляция, а также непосредственно опкод инструкции, который необходим для получения некоторых особых параметров инструкции, таких как, к примеру флаги условного исполнения, которые далеко широко применяются при автоматической генерации кода компиляторами как средство уменьшения количества инструкций ветвления, которые приводят могут приводить к сбросу предсказателей ветвлений в современных процессорах.

Именно здесь кроется проблема, связанная с производительностью виртуализационного решения, заключающегося в эмуляции процессорных инструкций. К сожалению даже использование даже такого сравнительно низкоуровневого языка программирования как C не обеспечивает достаточно компактный с точки зрения количества машинных инструкций код. Характерная величина функции эмуляции — 150-200 ассемблерных инструкций x86, что уже гарантирует просадку в производительности на уровне двух порядков. Если добавить к этому то что одни из наиболее часто встречающихся инструкций — инструкции для работы с памятью, то становится понятно, что суммарный оверхед на эмуляцию настолько существенен, что не позволяет использовать эмуляцию в качестве полноценного способа виртуализации исполнения ARM машинного кода на Android-x86.

Исходя из этого было решено найти более производительный способ решения поставленной задачи, по возможности минимизировав при этом необходимую работу и предоставив возможности по валидации

предлагаемого решения. Логичным решением в данном случае стала бинарная трансляция, как наиболее производительный и компактный способ произвести вышеобозначенные манипуляции. Сильные и слабые стороны данного подхода были описаны ранее, поэтому в первую очередь будет описан путь и конкретные подходы бинарной трансляции, с помощью которых были решены поставленные задачи по виртуализации.

7. 1 Декомпозиция задачи бинарной трансляции

В первую очередь стоит оговориться, что задача бинарной трансляции к сожалению не является тривиальной, и реализация любого трансляционного подхода (как статического, так и динамического) сопряжено с трудностями. Заключаются он в первую очередь в различиях процессорных платформ. В нашем случае задача осложнялась тем, что с точки зрения интерфейса целевой процессор принадлежал так называемому семейству RISC — архитектуре процессора, в котором быстродействие увеличивается за счёт упрощения инструкций, чтобы их декодирование было более простым, а время выполнения — меньшим.

В это же время хостовой процессор, то есть x86 или, иначе говоря IA-32 принадлежит семейству CISC, являющуюся в некотором роде противоположностью RISC. Наиболее значимыми особенностями, которые встают на пути при трансляции являются:

1) Переменные значения длины команды. В нашем случае эта длина может колебаться от 1-го до 15-ти байт. Это создаёт трудности в первую очередь при выделении памяти под исполняемый код, сгенерированной в процессе трансляции — становится невозможным заранее узнать сколько страниц памяти понадобится.
2) Небольшое число регистров, каждый из которых выполняет строго

определённую функцию. В случае x86 принято говорить, что в распоряжении программиста находятся 8 регистров общего назначения, а именно {EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP}, что во-первых меньше 16-ти регистров общего назначения в архитектуре ARMv7, а во-вторых — регистры x86 в отличие от их RISC-собратьев не являются симметричными, некоторые инструкции, такие как к примеру длинное целочисленное умножение (IMUL) можно произвести только над определенным и четко фиксированным набором регистров (EAX, EBX, EDX).

Таким образом уже на уровне постановки задачи наметились проблемы, которые было необходимо решать на архитектурном уровне, еще до детальной имплементации всех необходимых для решения задачи модулей. Заметим отдельно, что решено было выбрать путь динамической бинарной трансляции, так как такой подход:

1. Позволял реализовать задуманное в более короткие сроки. В отличие от динамического подхода статическая бинарная трансляция подразумевает, что весь исполняемый машинный код целевого бинарника будет транслирован еще до непосредственного запуска исполняемого кода, а возможно и до этапа загрузки. Таким образом возникает проблема, связанная с тем, что внешние зависимости для любого ELF-файла, очевидно разрешаются на этапе загрузки его в память, мы же не могли полноценно реализовать такой подход. При динамическом же подходе можно было воспользоваться уже существующим загрузчиком, который способен корректно разрешать все внешние зависимости целевых исполняемых файлов, и уже полученные разрешённые релокации использовать при генерации

кода для внешних библиотечных вызовов из транслируемого кода.

2. Обеспечивал возможность применения итерационного подхода, то есть реализация бинарной трансляции для примеру самого распространенного подмножества ARM-инструкций, а оставшуюся их часть всё так же виртуализировать за счет эмуляции. Также, учитывая что эмуляция машинного кода уже была достаточно хорошо проверена — появилась возможность использовать такой подход для валидации оттранслированного кода за счёт параллельного исполнения его с эмуляцией и сравнивая результаты.

Закрепив динамическую трансляцию как основной способ реализации виртуализированного выполнения машинного кода ARM необходимо было разделить процесс на определенные стадии, которые позволили бы упростить задачу. В конечном счете архитектурно были выявлены следующие стадии:

1. Модуль анализа так называемого code-flow машинного кода. Иными словами модуль анализировал целевой машинный код ARM, уже загруженный в память и с полностью разрешёнными релокациями. В текущей реализации на выходе из данного модуля мы получаем набор блоков машинного кода, которые выполняются строго последовательно и с соответствующей информацией о связях между ними. Данная задача осложняется тем, что каждый опкод в наборе машинных инструкций ARM содержит блок бит, отвечающих за так называемое условное исполнение (conditional execution). Эта битовая маска позволяет выполнять или же не выполнять инструкции в зависимости от содержащихся данных в регистре флагов ARM процессора. Условное исполнение дополняет возможности ветвления машинного кода наравне с условиями условных и безусловных прыжков (или

branch, ветвей в ARM-терминологии).

2. Модуль анализа использования регистров. Так как набор и свойства регистров целевой и хостовой платформы существенно отличаются, то данная информация является необходимой для следующей стадии. На данном этапе является возможным реализовать следующую оптимизацию — для каждого опкода целевого машинного кода не просто хранить информацию об используемых данной командой регистров, а также разделять это множество на множество тех, которые используются только для чтения и тех, которые используются также и для записи. Это позволит в дальнейшем при необходимости переупорядочить инструкции для исполнения для того, чтобы сгруппировать их по принципу локальности использования регистров целевой платформы, что приводит к возможности уменьшения перераспределения регистров хостового процессора внутри непрерывного блока машинного кода [18].
3. Регистровый аллокатор. Данный модуль отвечает за установление корректного соответствия между регистрами x86 и ARM перед генерацией машинного кода хостовой платформы. В данном случае задача сходна с решением распространенной задачи, которая решается при построении компиляторов — как оптимальным образом расположить переменные некоторого языка программирования по нативным регистрам целевой платформы таким образом, чтобы минимизировать с одной стороны обращение к памяти и стеку, а с другой не допустить неконсистентности и некорректного поведения.
4. Непосредственно бинарный транслятор. Зная информацию о используемых регистрах, а также о том, каким регистрам целевой x86 платформы они соответствуют данный модуль генерирует для каждой ARM инструкции некоторый по возможности оптимальный и

минимальный набор инструкций, который реализует её полностью нативно без необходимости прибегать к какой бы то ни было эмуляции. Минимальной единицей трансляции в данном случае является блок машинного кода ARM (соответствующий блоку, полученному после этапа CFA или code-flow analysis).

5. Модуль, отвечающий за связывание блоков между собой. Как уже было сказано ранее блоки кода являются непрерывными с точки зрения исполнения, то есть точка исполнения программы гарантированно выполнит блок до конца, попав в некоторую его точку. Заметим, что в настоящей имплементации модуль CFA разбивает целевой машинный код на блоки, в которых невозможны прыжки в середину блока (если такой блок возникает, то его разбивают на несколько). Такой подход может гарантировать, что после этапа регистровой аллокации схема соответствия регистров для блока машинного кода будет валидной. Это позволяет отказаться от загрузки/сохранения регистров до и после каждой оттранслированной инструкции и ограничиться такими участками шаблонного кода до и после непрерывных блоков, а также в случае, если необходимо некоторое фиксированное соответствие регистров (к примеру как в случае с операцией длинного умножения).

Рассмотрим каждый из представленных этапов бинарной трансляции отдельно, а затем опишем каким образом была реализована композиция предлагаемых подходов.

7.2 Модуль code-flow анализа архитектуры ARM

Как уже было сказано ранее модуль code-flow (или CFA - code-flow analysis) был необходим для корректного разделения исполняемого кода

ARM на непрерывные блоки машинных инструкций, которые подчинялись бы простым правилам.

1. Для каждого блока возможна лишь единственная точка входа в данный блок, то есть гарантируется, что в исполняемом файле не будет существовать инструкций условных или безусловных переходов, которые отправляли бы процессор исполнять инструкции из середины блока.
2. Блок сам по себе может закончиться как инструкцией перехода, так и нет. В данном случае подразумевается, что в данном месте возможен прыжок на данную позицию в коде извне. В случае же, если этого не происходит, то блок заканчивается инструкцией перехода. При этом нужно оговорить отдельно, что так как архитектура ARM является симметричной с точки зрения процессорных регистров (к примеру REGLR REGPC могут использоваться в арифметических операциях наравне с остальными регистрами) возникает сложность с тем, как понимать такие команды как переход по адресу в REGLR или загрузка некоторой константы в REGPC.

Отдельно заметим, что в данном подходе модуль CFA работает сразу после стадии загрузки исполняемого кода в память, то есть в тот момент, когда все внешние релокации уже разрешены и в машинном коде и GOT таблице находятся валидные значения адресов. Рассмотрим часть структуры данных, которую использует модуль для выполнения своих функций.

```
typedef struct source_block {
    SLIST_ENTRY(source_block) list;

    addr_t begin, end;
    struct source_block *next_block;
    struct source_block *branch_block;
```

```
    int is_referenced : 1;
} source_block_t;
```

Непрерывны блок машинного кода может закончиться по разным причинам. Например если при анализе будет обнаружена инструкция условного или безусловного перехода в середину существующего блока, который уже был анализирован ранее. В этом случае блок кода будет разделён на два блока (A и B), причём блок B начнётся с целевого адреса инструкции перехода. В этом случае указатели в приведённой структуре данных будут сформированы следующим образом:

A->next_block == B; A->branch_block == NULL

Ещё один случай это условные инструкции. Отдельно укажем, что в данном случае имеются в виду не только инструкции условных прыжков, но и обычные ARM инструкции, которые в своём машинном опкоде в поле условного исполнения содержат бинарную маску, отличную от 0x0111 (То есть выполнять всегда). Единственная сложность в данном случае состоит в ветвлении, статистический результат которого на эта CFA предсказать невозможно. Таким образом в данном место сложно добавить какую-то оптимизацию, связанну/ с бинарной трансляцией. В конечном итоге получаем:

A->next_block == B; A->branch_block == NULL

Последним случаем является безусловная инструкция ветвления. Для блоков, которые заканчиваются такой инструкцией необходимо сформировать ещё один CFA-блок, начинающийся с адреса, соответствующего целевому адресу инструкции безусловного перехода. Тогда для изначального блока получаем:

A->next_block == NULL; A->branch_block == B

Заметим, что в текущей реализации модуля необходимо задать точку входа в бинарный файл, то есть адрес, с которого необходимо начинать анализ потока исполнения машинного кода. Однако разрешено указывать несколько точек входа и расширять объем проанализированного кода, потому что в случае разделяемых библиотек как правило имеется несколько точек входа в машинный код ELF-файла. В этом случае необходимо после этапа загрузки взять адреса внешних символов разделяемой библиотеки, которые участвовали в релокациях и запустить анализ кода начиная с этих адресов. В конечном итоге на выходе из описываемого модуля мы получаем древовидную структуру данных, которая хранит в себе полную информацию о ходе исполнения программы.

Данный подход не покрывает случаи, когда к примеру REGPC грузится некоторый адрес машинной инструкции, который вычисляется динамически в ходе работы нативного кода. В этом случае возможны два подхода:

1. Выходить в эмулятор и эмулировать исполнение нативного кода вплоть до ближайшего блока оттранслированного кода, далее возвращаться к исполнению нативного кода.
2. Динамически в момент наступления ситуации, описанной выше вызывать последовательно всю процедуру бинарной трансляции начиная с целевого адреса перехода. такой подход к сожалению чреват непредвиденными задержками в работе нативного кода, так как процедура бинарной трансляции с одной стороны весьма требовательна к производительности, а с другой стороны активно обращается к различным разрозненным структурам в памяти, необходимым для корректного функционирования. Всё это не позволяет эффективным образом использовать кеш процессора, что

делает весь участок кода, связанный с бинарной трансляцией весьма ресурсоемким.

В конечном счёте в текущей реализации был выбран первый подход, так как он позволяет довольно гибко поступать в случае встречи неоттранслированного кода с одной стороны, и при этом ничто не мешает в дальнейшем расширить систему на второй случай, который сулит гораздо большую производительность, хотя и может вносить непредвиденные задержки.

7.3 Анализ использования регистров и регистровый аллокатор

В целях оптимизации производительности в текущей реализации бинарной трансляции было принято решение использовать нативные x86 регистры в качестве основного инструмента манипуляции данными виртуального контекста процессора. Первоначально предлагалось использовать для этих целей уже существующую и описанную выше структуру VCPU, которая является одной из основных частей эмулятора. Таким образом в любой момент при исполнении оттранслированного кода предлагалось поддерживать консистентное состояние виртуального контекста, что позволяло в любой момент переключаться на исполнение кода с помощью эмулятора. Однако при подобном подходе будет происходить постоянное обращение к оперативной памяти. Даже при условии нахождения данных в кэше время обращения к ним существенно выше, чем к нативным регистрам. Также, учитывая что оттранслированный код активно обращается к памяти довольна вероятна ситуация, когда виртуальный контекст будет находиться не в кэше верхнего уровня, что драматическим образом вносит задержки на каждом обращении к нему [20,21,22].

После проведения замеров производительности, был сделан вывод, что такой первоначальный подход вносит до двух порядков задержки по времени. На каждый такт исполнения нативной инструкции процессор простаивал бы несколько десятков тактов при обращении к кэшу второго уровня (или к оперативной памяти в худшем случае). В конечном счете было принято решение использовать нативные регистры и попытаться оттранслировать ARM инструкции как можно ближе семантически в инструкции x86.

Самой большой проблемой в данном случае помимо сложности и архитектурного несоответствия между двумя разнородными наборами инструкций является нехватка x86 регистров по сравнению с ARM собратьями. Заметим, что в данном случае среди хостовых регистров мы можем свободно использовать лишь 6 (EAX, EBX, ECX, EDX, ESI, EDI). Стековый регистр ESP необходимо сохранить в нетронутом состоянии, так как стек необходим для выполнения как оттранслированного кода, так и кода эмулятора, а ещё один регистр используется для хранения указателя на виртуальный контекст процессора, на случай особых случаев, когда всё же требуется выйти в эмулятор.

В конечном итоге для построения соответствия между ARM регистрами и x86 регистрами был реализован модуль, действующий подобно регистровому аллокатору в компиляторе [16,17]. Для каждого блока кода опкоды машинных инструкций анализируются с точки зрения использования ими регистров. Далее строится граф использования регистров, при этом разделяются случаи, когда регистры используются для чтения или записи. Далее данный граф максимально глубоко “раскрашивается” в доступные

шесть цветов (по количеству доступных регистров). Таким образом для блока кода строится максимальная последовательность инструкций, которые могут быть исполнены с использованием нативных регистров. Далее, в случае если был покрашен не весь граф, пройденные вершины отсекаются, а процедура покраски начинается заново. Заметим, что в данном случае мы применяем по сути вид жадного алгоритма, когда на каждом этапе покраски мы пытаемся покрасить максимальный подграф. Учитывая, что машинный код ARM изначально попадает к нам будучи скомпилированным с помощью NDK, этот подход будет работать эффективно, так как компилятор gcc изначально старается разложить код при компиляции по нативным регистрам наиболее эффективным образом [17,18]. В результате использование жадного алгоритма позволило решить эту задачу эффективно как с точки зрения конечного результата, так и с точки зрения асимптотической оценки по времени (линейно).

Остановимся чуть подробнее на том, каким образом анализируется использование регистров для каждого опкода. В данном случае механизм работы схож с декодированием. Для каждой инструкции или схожего набора инструкций в таблице декодирования хранится указатель на функцию, которая заполняет соответствующую структуру данных, используемую для хранения информации об использовании регистров. Учитывая, что в архитектуре ARM есть не так уж и много различного рода видов инструкций можно эффективным образом разделить весь набор опкодов на классы, которые будут анализироваться конкретным образом. Это позволяет с одной стороны упростить код, а с другой стороны сократить количество вызовов, необходимое для анализа использования регистров. Заметим, что в целях сохранения модульности и взаимозаменяемости данный код существует от дальнейшего этапа, которым и является регистровый аллокатор.

На выходе из данного модуля для каждого блока кода мы получаем некоторую последовательность соответствий регистров ARM-x86, которую можно применять в дальнейшем при трансляции.

7.4 Модуль бинарной трансляции

В данной главе будет механизм динамической бинарной трансляции, который позволил существенным образом ускорить выполнение машинного кода ARM целевого исполняемого файла. На момент написания данной работы в текущей версии динамического транслятора реализован весь набор инструкций набора команд ARM уровня пользователя, а также ведется работа по имплементации набора инструкций THUMB. Оценочно можно сказать что трансляцию этого укороченного набора инструкций можно будет реализовать существенно проще, так как он включает в себя меньшее количество опкодов, а также из за сокращенного опкода обладает в среднем более простым поведением в расчете на инструкцию, что упрощает трансляцию.

Напомним в общих чертах какая задача стояла перед началом реализации динамической бинарной трансляции машинного кода ARM в набор инструкций архитектуры IA-32. В первую очередь стоит упомянуть, что на момент начала разработки уже были проведены эксперименты с использованием нативных x86 регистров вместо постоянных обращений к виртуальному контексту процессора, поэтому окончательный вариант алгоритма трансляции осуществлялся сразу с прицелом на данную оптимизацию и в расчете на максимальную производительность. В конечном счете за счет модуля анализа потока исполнения кода (CFA) задачу удалось

удачным образом декомпозировать на подзадачи гораздо меньшего размера (непрерывные блоки кода), которые гораздо проще было формализовать и оптимизировать. На входе транслятору подаётся не просто описатель блока кода, а специализированная структура данных, детально описывающая поведение машинных инструкций. Рассмотри представление данных при трансляции более подробно. После стадии CFA и регистрового аллокатора на вход алгоритму трансляции последовательно подаются на вход описатели блоков кода в удобной форме, и далее происходит работа с ними.

```
typedef struct {
        stub_chunk_t* head_chunk;
        uint8_t *stub_addr;
        uint8_t *stub_tail_addr;

        x86_reg_t in_reg_map[ARM_REG_COUNT];
        x86_reg_t out_reg_map[ARM_REG_COUNT];

        trans_insn_t *branch_insn;
        trans_insn_t *insn_list_head;
        trans_insn_t *insn_list_tail;
} translated_block_t;
```

В данном случае стоит оговориться, что так как трансляция происходит в среде процессорной архитектуры IA-32, это налагает определенные ограничения. Первым из них является невозможность исполнение кода из произвольного места в памяти. Поэтому было принято решение использовать отдельный модуль выделения страниц памяти, для которых возможно размещение и исполнение кода. Заметим, что в данном случае выделение

памяти происходило не с помощью платформо-независимых вызовов malloc (или схожих с ним), а посредством системного вызова mmap с надлежащим образом выставленными флагами. Из-за того, что подобный механизм может выделять память исключительно страницами — изредка в транслируемый код требуется добавлять инструкции перехода между подобными страницами исполняемого кода. В целях минимизации подобного поведения память под исполняемый код выделяется так называемыми “большими страницами”, объемом 2 мегабайта вместо стандартных 4-х килобайт. За счет этого инструкция перехода между страницами исполняемого кода встречается настолько редко, что фактически не влияет на производительность.

В процессе своей работы транслятор машинного кода заполняет процессорными инструкциями описанную выше память. Для этого в структуре, соответствующей блоку кода содержатся указатель на страницу с кодом (stub_chunk_t* head_chunk), а также адреса начала и конца области памяти с кодом, который соответствует данному блоку ARM кода (uint8_t *stub_addr; uint8_t *stub_tail_addr).

Также данная структура данных включает в себя полную информацию о инструкциях ARM кода, а также о использовании ими регистров (данная информация заполняется на этапе анализа использования регистров). Регистровый аллокатор в свою очередь для данной последовательности инструкций распределяет нативные x86 регистры и сохраняет данную информацию, передавая в транслятор. Таким образом мы получаем список описаний инструкций данного блока кода (trans_insn_t *insn_list_head; trans_insn_t *insn_list_tail), а также в случае, если блок кода заканчивается инструкцией ветвления — ее описатель отдельно (trans_insn_t *branch_insn).

Два массива номеров регистров представляют собой карты соответствия регистрами ARM — x86. Данная информация уже известна

после стадии регистрового аллокатора, и далее используется в стадии оптимизации за счет связывания блоков, которая будет описана далее.

Рассмотрим в каком виде хранится информация о использовании регистров, а также о соответствии между ARM регистрами и нативными x86 регистрами. Для этих целей используется отдельная структура, содержащая следующие поля:

```
typedef struct trans_insn {
        addr_t addr;
        const struct instr_description *desc;
        size_t len;
        op_t op;
        uint16_t reg_usage_mask;
        union {
                arm_reg_t strict_reg_map[4];
                uint32_t strict_reg_map_uint;
        };
        struct trans_insn *prev;
        struct trans_insn *next;
} trans_insn_t;
```

Данная структура в первую очередь поддерживает двусвязный список за счет двух указателей (struct trans_insn *prev; struct trans_insn *next). Кроме информации о самой инструкции (ее адрес, длина, опкод) в данной структуре также хранится маска использования ARM регистров (uint16_t reg_usage_mask).

Однако наиболее проблематичной для трансляции в данном случае является использование фиксированного отображения (arm_reg_t strict_reg_map[4]). Это необходимо из-за того, что определенная функциональность в архитектуре IA-32 ограничена определенными

регистрами. К примеру результат умножения двух 32-битных чисел может быть полностью сохранен только в пару регистров EAX:EDX.

Другой пример — это сдвиг регистра на нефиксированное количество бит, которое хранится в другом регистре в архитектуре x86 может производиться исключительно с помощью регистра ECX. В это же время в симметричной архитектуре ARM данные операции могут быть произведены над любыми наборами регистров, что заставляет в некоторых случаях дополнительно и заранее фиксировать отображение регистров.

Получив вышеописанную информацию алгоритм трансляции последовательно для инструкций вызывает функции трансляции для определённых классов инструкций или их отдельных представителей, которые в свою очередь после анализа опкода оригинальной ARM инструкции генерируют исполняемый x86 код и записывают его в соответствующие страницы исполняемой памяти. В данном случае подобный механизм схож с процедурой эмуляции, с той разницей, что вместо непосредственного исполнения инструкции генерируется машинный код, который выполняет эквивалентную процедуру, используя при этому описанный выше механизм отображения регистров.

В качестве примера рассмотрим одну из функций трансляции — ARM инструкции складывания регистров, при этом регистр источника может быть побитово сдвинут.

```
trans_status_t trans_arm_add_reg(trans_insn_t *insn, x86_reg_t reg_map[ARM_REG_COUNT], uint8_t **write_ptr)
{
    arm_reg_t rd, rn, rm;
    ARM_DECODE_ARGS_DNMS(insn->op, rd, rn, rm, bit_s);
    if (rd == REGPC && bit_s)
```

```
        return TRANS_UNPREDICTABLE;
    uint8_t shift_type = (insn->op >> 5) & 0b11;
    uint8_t shift = (insn->op >> 7) & 0b11111;
    MOV_REG_REG(write_ptr, REG_TMP, reg_map[rm]);
    BARREL_SHIFTER_IMM_SHIFT(write_ptr, shift_type, REG_TMP, shift);
    if (rn == REGPC)
        ADD_REG_IMM32(write_ptr, REG_TMP, insn->addr + 8);
    else
        ADD_REG_REG(write_ptr, REG_TMP, reg_map[rn]);
    if (rd == REGPC)
        MOVD_MEM_REG_DISP8(write_ptr, REG_EBP, REG_TMP,
ARM_REG_ADDR_DISP8(REGPC));
    else
        MOV_REG_REG(write_ptr, reg_map[rd], REG_TMP);
    return TRANS_SUCCESS;
}
```

На выходе после этапа трансляции получаем набор полностью транслированных блоков кода, которые полностью повторяют функциональность соответствующих блоков ARM кода. Заметим, что за счет полного покрытия набора инструкций отпадает необходимость выхода в эмулятор при встрече некоторых инструкций. Первоначально это требовалось для исполнения отдельного класса инструкций с нетривиальной имплементацией, а также атомарных инструкций, но сейчас, как уже было сказано, — набор инструкций ARM полностью покрыт.

7.5 Модуль связывания блоков

Как уже было сказано ранее, в первоначальной версии реализации динамической трансляции применялся подход, когда точка исполнения нативного кода возвращалась в эмулятор как только она достигала конца непрерывного блока ARM кода.

Очевидно, что в таком случае много времени теряется во первых во

время собственно переключения контекста (с реального на виртуальный), что включает в себя активную работу со стеком и обращения к памяти, в котором находится VCPU — виртуальный контекст процессора.

Также, так как инструкции переходов между блоков не исполнялись нативно, а эмулировались — были дополнительные потери производительности, которые не позволяли достичь максимальной производительности, которую может предоставить подход с полной динамической трансляцией. Однако, с другой стороны, подход с разбиением машинного кода ARM на непрерывные блоки давал такие неоспоримые преимущества, как возможность декомпозировать задачу трансляции на гораздо более мелкие и, что самое главное, независимые задачи с одной стороны, а с другой — позволял всецело положиться на подход анализа использования регистров, описанный выше.

Заметим, что в случае трансляции всего бинарного файла с машинным кодом нам потребовалось бы применить гораздо более сложный алгоритм, так как code-flow в произвольном месте ARM кода пришлось бы вычислять не заранее, а динамически.

Таким образом, обосновав необходимость каким-то образом избавиться от выходов в эмулятор на границе блоков было принято решение использовать процедуру связывания блоков посредством инструкций ветвления из набора машинных инструкций процессорной архитектуры x86. Причем в каждом конкретном случае применялись инструкции, максимально близкие (а порой и эквивалентные) тем, которые встречались в конце блоков кода. Однако в данном случае нас подстерегала проблема, которая заключалась в архитектурном несоответствии флагов, которые влияли на поведение инструкций условного перехода. Рассмотрим подробнее флаги условного исполнения для обеих архитектур.

В случае ARM за данную функциональность отвечают четыре бита в регистре CPSR (N, Z, C и V соответственно):

1. Negative – результат операции получился отрицательным,
2. Zero – результат равен нулю,
3. Carry – при выполнении операции с беззнаковыми числами произошел перенос,
4. oVerflow – при выполнении операции со знаковыми числами произошло переполнение, результат не помещается в регистр.

Как уже было сказано выше — каждый опкод в процессорной архитектуре ARM имеет так называемую маску условного исполнения. На практике это означает, что от первых четырёх бит опкода зависит, будет ли процессор исполнять очередную инструкцию, и при какой комбинации значений, находящихся в битах условного исполнения. Ниже представлена таблица корректных значений битов, находящихся в маске условного исполнения опкода, а также описания условий, при которых рассматриваемый опкод выполнится.

Code	**Suffix**	**Description**	**Flags**
0000	EQ	Equal / equals zero	Z
0001	NE	Not equal	!Z
0010	CS / HS	Carry set / unsigned higher or same	C
0011	CC / LO	Carry clear / unsigned lower	!C
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	!N

0110	VS	Overflow	V
0111	VC	No overflow	!V
1000	HI	Unsigned higher	C and !Z
1001	LS	Unsigned lower or same	!C or Z
1010	GE	Signed greater than or equal	N == V
1011	LT	Signed less than	N != V
1100	GT	Signed greater than	!Z and (N == V)
1101	LE	Signed less than or equal	Z or (N != V)
1110	AL	Always (default)	any

В то же самое время в случае процессорной архитектуры x86 дела обстоят несколько иначе. За поведение условных инструкций отвечает специальный регистр EFLAGS, который помимо значений, которые в данном случае не имеют значения содержит также четые бита флагов.

Флаги состояния отражают результат выполнения арифметических инструкций, таких как ADD, SUB, MUL, DIV.

- CF — устанавливается при переносе из/заёме в (при вычитании) старший значащий бит результата и показывает наличие переполнения в беззнаковой целочисленной арифметике. Также используется в длинной арифметике.
- ZF — устанавливается, если результат машинной операции по модулю 2 в степени k (где k - разрядность ячейки) равен нулю.

- SF — равен значению старшего значащего бита результата, который является знаковым битом в знаковой арифметике.
- OF — устанавливается, если целочисленный результат слишком длинный для размещения в целевом операнде (регистре или ячейке памяти). Этот флаг показывает наличие переполнения в знаковой целочисленной арифметике (в дополнительном коде).

В длинной целочисленной арифметике флаг CF используется совместно с инструкциями сложения с переносом (ADC) и вычитания с заёмом (SBB) для распространения переноса или заёма из одного вычисляемого разряда длинного числа в другой.

Инструкции условного перехода J*cc* (переход по условию *cc* — например, JNZ для перехода, если результат не ноль), SET*cc* (установить значение байта-результата в зависимости от условия *cc*), LOOP*cc* (организация цикла) и CMOV*cc* (условное копирование) используют один или несколько флагов состояния для проверки условия. Например, инструкция перехода JLE (jump if less or equal — переход, если «меньше или равен», ≤) проверяет условие «ZF=1 или SF ≠ OF».

В целом архитектура x86 содержит большое разнообразие инструкций перехода с условным выполнением, однако было решено ограничиться использованием узкого подмножества в целях упрощения кода.

Таким образом в текущей реализации каждый блок непрерывного кода пропущенный через стадию трансляции также дополнительно дополняется инструкцией перехода, поведение которой полностью совпадает с поведением целевой инструкции машинного кода ARM. В случае, если в процессе трансляции встретилась инструкция, которая не является инструкцией ветвления, но которая требует условного исполнения — она

также оборачивается в блок нативного кода, который повторяет ее поведение. Отдельно стоит отметить, что несмотря на схожесть флагов в обеих архитектурах, различия в них заставляют отдельно поддерживать во флагах архитектуры x86 значения, которые можно безболезненно интерпретировать при переходе в эмулятор.

В конечном счете данная оптимизация позволила полностью отказаться от выхода в эмулятор и за счет этого был достигнут существенный прирост в производительности.

8. Оценки производительности

В данном разделе будет подведены заключительные итоги о производительности предлагаемого виртуализационного решения. Напомним, что изначальный подход, который основывался на реализации эмуляции исполнения машинного кода ARM и поддержании консистентного (эквивалентного реальному) виртуального контекста процессора не позволил добиться необходимой производительности в связи с чем было принято решение применить один из видов бинарной трансляции машинного кода ARM в процессорные инструкции IA-32.

Рассмотрим отдельно подход, с помощью которого производилась оценка производительности бинарной трансляции. В целях адекватного отслеживания изменения быстродействия предлагаемого решения были введены две метрики: медианное количество машинных инструкций x86 на одну инструкцию ARM и отношение скорости выполнения синтетических тестов в нативном случае на процессоре с архитектурой ARMv7 к скорости выполнения его с помощью бинарной трансляции. Заметим, что вторая

метрика бралась с нормировкой на тактовую частоту процессора, на котором выполнялись тесты.

В случае первой метрики окончательное медианное значение количества x86 инструкций на одну ARM инструкцию было снижено до 4-5. Заметим, что это значение варьируется в зависимости от содержания синтетического теста и имеет смысл рассматривать именно диапазон значений, за которые никогда не выходит данное значение.

Полученное значение с одной стороны говорит о том, что в рамках текущего подхода удалось реализовать достаточно эффективный алгоритм преобразования машинного кода, который позволил минимизировать количество генерируемого машинного кода, что снижает как затраты на трансляцию кода, так и оптимизирует использование кэша непосредственно в процессе исполнения транслируемого кода. К сожалению, как уже было сказано выше, не всегда удаётся напрямую провести преобразование между опкодами разнородных машинных архитектур, и становится необходимым использовать дополнительную логику в генерируемом коде для того, чтобы полностью реализовать логику поведения целевого опкода включая все граничные случаи. Инструкции, которые требуют подобного поведения, к счастью, встречаются не слишком часто: это в первую очередь загрузка/выгрузка серии регистров по маске в/из памяти, работа со стеком, длинное умножение, а также операции битового сдвига регистров операндов на переменную величину. Отметим также, что большинство ARM инструкций позволяет выполнять несколько арифметических действий над операндами, что невозможно в случае архитектуры IA-32.

Еще одним источником “лишних” генерируемых инструкций является необходимость перераспределения ARM регистров по нативным x86

регистрам при трансляции. И из-за недостатка последних зачастую в длинных блоках кода, а также при переходе между блоками приходится добавлять дополнительные инструкции обмена регистрами, что семантически не несет полезной нагрузки, но необходимо для достижения консистентности отображения регистров, а также для корректного функционирования оттранслированного кода. Количество таких инструкций было уменьшено за счет использования регистрового аллокатора и связывания блоков (это позволило избавиться от полной загрузки/выгрузки регистров на входе и выходе из блоков).

Однако куда более интересной с практической точки зрения является вторая метрика, которая описывает реальную производительность текущей реализации. Отметим, что все замеры производительности были нормированы на тактовую частоту процессора, на котором производились замеры, чтобы нивелировать воздействие различного быстродействия железа. Также намеренно в качестве тестовых устройств использовались процессоры похожих классов с похожими объемами кэша различных уровней.

В качестве синтетических тестов производительности использовались два вида тестовых программ: хеширующая большой случайный объем данных, и программа, содержащая большое количество инструкций ветвления. Первый тест по своей сути позволяет проанализировать быстродействие и эффективность бинарной трансляции на длинных непрерывных участках кода, но с простой логикой с точки зрения процессора (то есть с минимальным количеством ветвлений). Второй же напротив, является стресс-тестом модуля CFA, а также модуля связывания блоков, показывающий, насколько эффективно транслируется и выполняется код, состоящий из большого количества небольших участков непрерывного кода.

Таким образом измеряются потери производительности на трансляцию кода переходов между блоками и перераспределения ARM регистров по нативным x86 регистрам.

После проведенных замеров все данные были усреднены, и избавлены от выбросов. Также отдельно были проведены замеры под нагрузкой, как по процессору, так и по памяти, что также дополнило картину. Анализ всех полученных данных позволил сформировать следующую картину производительности предлагаемого решения (значения даны в доле от нативного исполнения теста на процессоре с архитектурой ARMv7 в таких же условиях):

	Тест хеширования	Тест ветвлений	Тест хеширования (CPU stress)	Тест ветвлений (memory stress)
Лучший результат	0,82	0,52	0,75	0,41
Худший результат	0,68	0,39	0,67	0,35

В приведенной таблице результаты приведены с отброшенными выбросами (около 5% результатов, большинство из них связано со стресс-тестами). В итоге сделаем главные выводы из приведенных цифр:

1. Была получена производительная реализация бинарной трансляции, которая обеспечивает производительность, близкую к нативной.
2. Текущий подход стабильно работает как при ненагруженной системе, так и под нагрузкой, не показывая значительного уменьшения скорости выполнения оттранслированного кода.
3. Наиболее проблематичной ситуацией является наличие в программе кода с малым количеством непрерывных и протяженных блоков кода и с большим количеством ветвлений. Однако даже в этом случае производительность остается на приемлемом уровне.

В качестве итога можно сказать, что полученная реализация технологии виртуализации исполнения машинного кода ARM является достаточно производительной и эффективной, что позволяет использовать ее в качестве основной системы в Parallels ARM Emulator.

9. Заключение

В заключение хотелось бы еще раз отметить, что в настоящее время технологии виртуализации являются бурно развивающейся областью информационных технологий, привлекающие внимание ведущих компаний, работающих в этой сфере. В данной работе была освещена проблематика виртуализации архитектуры ARM на процессорах x86 в приложении к проблеме запуска Android-приложений, содержащих нативный код.

В обзорной части данной работы был проведён обзор существующих технологий виртуализации, которые могли быть применены в данной сфере. Также был подробно рассмотрен способ, используемый в QEMU, дано подробное описание алгоритма и принципа работы динамической трансляции, используемый этой программой. В практической части была поставлена задача реализации эмулятора, позволяющего осуществлять запуск отдельных программ, скомпилированных под ARM на процессоре с архитектурой x86, на базе существующей системы полной эмуляции аппаратного окружения. В ходе осуществления данного этапа была показана принципиальная возможность используемого подхода. Также был написан загрузчик, необходимый для корректного запуска исполняемого файла, а также для его связывания (линковки) с внешними библиотеками. Выполнение этого этапа позволило рассмотреть работу эмулятора на реальных программах.

Была осуществлена оптимизация полученной системы путем реализации возможности перенаправления библиотечных вызовов в хостовую систему. Это позволило избавиться от необходимости сопровождения гостевого исполняемого кода копиями необходимых библиотек. Данная оптимизация также дала значительный прирост производительности. Как было показано в разделе, посвященном оценке производительности, быстродействие программ, содержащих большое количество библиотечных вызовов, выполняемых данным эмулятором отличается от их нативного исполнения менее, чем на порядок. Таким образом в ходе выполнения данной работы была получена программная система, выполняющая поставленные задачи, а также показана жизнеспособность предлагаемого решения. При дальнейших оптимизациях и внедрении динамической трансляции, полученное решение стало возможным использовать в существующей системе виртуализации Parallels ARM Emulator.

Даже первоначальная реализация, которая не обладала всем набором необходимых оптимизаций уже позволила достичь производительности, которая превосходила QEMU. В дальнейшем велась работа по улучшению предложенного решения, а также его существенная оптимизация. Наиболее существенный прирост быстродействия в данном случае был осуществлен за счет использования нативных x86 регистров, а также связывания блоков исполняемого кода в целях непрерывного исполнения. Все вместе, это позволило создать систему виртуализации исполнения машинного кода ARM, которая по своей производительности приближается к скорости нативного исполнения кода и полностью решает поставленные перед ней задачи.

Хотелось бы поблагодарить своего научного руководителя Тормасова Александра и руководителя проекта Корякина Алексея за неоценимую помощь в работе.

10. Список используемой литературы

[1] Surhone, L.M. and Tennoe, M.T. and Henssonow, S.F. «Java Native Interface» – VDM Publishing, 2010 – ISBN:9786134548700

[2] Krajci, I. and Cummings, D. «Android on X86: An Introduction to Optimizing for Intel Architecture» – Apress, 2013 – ISBN:9781430261308

[3] Hess, K. and Newman, A. «Practical Virtualization Solutions: Virtualization from the Trenches» – Pearson Education, 2009 – ISBN:9780137055005

[4] Menken, I. «Virtualization - The Complete Cornerstone Guide to Virtualization Best Practices Concepts, Terms, and Techniques for Successfully Planning, Implementing and Managing Enterprise IT Virtualization Technology» – Emereo Pty Limited, 2008 – ISBN:9781921523915

[5] Wolf, C. and Halter, E.M. «Virtualization: From the Desktop to the Enterprise» – Apress, 2005 – ISBN:9781430200277

[6] Wikipedia, S. and Books Llc «Free Virtualization Software: Qemu, Openjdk, Xen, Openvz, Free Java Implementations, Freebsd Jail, Chroot, Marionnet, Cooperative Linux» – General Books LLC, 2010 – ISBN:9781157247128

[7] Surhone, L.M. and Timpledon, M.T. and Marseken, S.F. «Qemu» – VDM Publishing, 2010 – ISBN:9786130948931

[8] Seal, D. «ARM Architecture Reference Manual» – Addison-Wesley, 2001 – ISBN:9780201737196

[9] Gibson, J.R. «ARM Assembly Language: An Introduction» – Lulu.com, 2007 – ISBN:9781847536969

[10] Sloss, A. and Symes, D. and Wright, C. «ARM System Developer's Guide: Designing and Optimizing System Software» – Elsevier Science, 2004 – ISBN:9780080490496

[11] Love, R. «Linux System Programming: Talking Directly to the Kernel and C Library» – O'Reilly Media, 2013 – ISBN:9781449341541

[12] Kerrisk, M. «Linux System and Network Programming: A Complete Guide» – Apress L. P, 2009 – ISBN:9781430224716

[13] Intel Corporation «IA-32 Intel Architecture Software Reference Manual: Basic architecture, vol 1» – Intel Corporation, 2005

[14] Intel Corporation «IA-32 Intel Architecture Software Reference Manual: Instruction set reference, vol 2A» – Intel Corporation, 2005

[15] Intel Corporation «IA-32 Intel Architecture Software Reference Manual: Instruction set reference, vol 2B» – Intel Corporation, 2005

[16] Теория и практика языков программирования. Учебник для вузов. Стандарт 3-го поколения – Орлов Сергей Александрович – Издательский дом "Питер", 2012 – ISBN:9785496000321

[17] Языки программирования и методы трансляции. Учебное пособие. – Опалева Э. – БХВ-Петербург, 2005 – ISBN:9785941573271

[18] Современные операционные системы. 4-е изд. – Таненбаум Эндрю С, Бос Херберт – Издательский дом "Питер", 2015 – ISBN:9785496013956

[19] Cache and Memory Hierarchy Design: A Performance-directed Approach – Steven A. Przybylski – Morgan Kaufmann –ISBN:9781558601369

[20] The Cache Memory Book – Jim Handy – Morgan Kaufmann – ISBN:9780123229809

[21] Processor Microarchitecture: An Implementation Perspective – Antonio Gonzalez, Fernando Latorre, Grigorios Magklis – Morgan & Claypool Publishers, 2010 – ISBN:9781608454525

[22] Analysis of multi-megabyte secondary CPU cache memories – Richard Eugene Kessler – University of Wisconsin–Madison, 1991 – ISBN:9780818670947