Министерство образования и науки Российской Федерации

МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ

(государственный университет)

ФАКУЛЬТЕТ УПРАВЛЕНИЯ И ПРИКЛАДНОЙ МАТЕМАТИКИ

КАФЕДРА ТЕОРЕТИЧЕСКОЙ И ПРИКЛАДНОЙ ИНФОРМАТИКИ

"03.04.01 Прикладные математика и физика"

Поддержка системы команд THUMB в двоичной трансляции

Магистерская диссертация

студента 176 группы

Занегина Александра Геннадьевича

Научный руководитель

Тормасов А.Г., д.ф.-м.н.

г. Долгопрудный

2017

Содержание

Введение

Данная работа посвящена увеличению производительности программной виртуальной машины Parallels ARM Emulator, разрабатываемой на кафедре. Основной задачей этого программного комплекса является предоставление возможности запуска и отладки программ, предназначенных для выполнения на процессорах семейства архитектур ARM, на машинах с поддержкой архитектур семейства x86. В рамках работы рассматривается поиск самых актуальных направлений оптимизации, имплементация двоичной трансляции, а также ряд небольших улучшений, связанных с выбором структур данных и алгоритмов.

Семейство процессорных архитектур ARM прочно занимает огромную долю рынка микропроцессоров для пользовательских устройств в течении уже нескольких лет, в то время как x86 не теряет своих позиций. [13] Производители операционных систем и конечных устройств поддерживают оба семейства архитектур, в том числе и в одних продуктах сразу. [14] Для того, чтобы успешно поддерживать свои продукты в условиях раздробленности парка устройств у конечных пользователей, разработчикам приложений приходится тратить кратно больше ресурсов и времени на отладку. Программная виртуальная машина позволяет сократить траты ресурсов в обмен на уменьшение скорости отладки, объем которого обратно зависит от качества виртуальной машины. В совокупности, это показывает актуальность данной работы.

Целью работы является улучшение скоростных характеристик вышеупомянутого программного комплекса виртуализации. Основной ориентир идет на скорость исполнения программ, собранных для работы под управлением операционной системы Android и среды выполнения Android NDK, однако это не является ограничением для рассматриваемой виртуальной машины или применяемых методик.

Современные варианты ARM включают в себя поддержку двух архитектур системы команд: одноименной arm, с фиксированным размером инструкции 32

бита, и Thumb, инструкции которой могут занимать либо 16, либо 32 бита. Первая имеет большую регулярность и выразительную способность инструкций, вторая жертвует этим в пользу высокой плотности кода. На аппаратном обеспечении, arm позволяет достичь большей производительности, однако высокие требования по памяти вынуждают разработчиков использовать Thumb вне критических участков. Так, Android NDK приложения и библиотеки по-умолчанию компилируются в Thumb. Все это привело к выбору имплементации двоичной трансляции данной системы команд как очевидного пути оптимизации.

Под двоичной (бинарной) трансляцией машинного кода обычно понимается техника виртуализации, включающая в себя модификацию машинного кода, исполняемого эмулируемой (гостевой) системой, заранее (Ahead-of-time, AOT) или же непосредственно во время исполнения (Just-in-time, JIT, “на лету”). Изначально данная техника виртуализации была рассчитана на виртуальные машины, у которых целевая (хост, реальная) архитектура системы команд совпадает с эмулируемой (гость, виртуальной), и была направлена на компенсацию некоторых недостатков классического trap-and-emulate метода. В таком варианте, модификации подвергались в основном операнды инструкций, такие как адреса в памяти или регистры. Главный недостаток классической виртуализации, который и привел к созданию метода бинарной трансляции, это невозможность (либо абсурдные накладные расходы времени) аппаратного перехвата некоторых классов инструкций, например, требующих привилегированных режимов выполнения. Это либо делает невозможным работу самого программного комплекса виртуальной машины вне привилегированного режима, либо вынуждает использовать медленные и не всегда безопасные обходные пути. Для виртуальных машин с различающимися целевыми и эмулируемыми архитектурами систем команд эти проблемы тоже актуальны, так как различные процессорные архитектуры могут иметь неэквивалентные режимы привилегий. [2]

Очевидно, что помимо трансляции адресов, сами инструкции тоже должны быть модифицированы. Алгоритмическая возможность отражения набора инструкций одной микропроцессорной архитектуры на другую в случае ARM и x86

естественно вытекает из тезиса Чёрча и Тьюринг-полноты этих систем. Говоря о технической возможности и эффективности данного подхода, необходимо помнить о различающемся процессорном контексте. Отличия регистровых файлов, размеров слов, состояний контекста и механизмов адресации и исключений, создают основные сложности для трансляции.

Главные накладные расходы по времени исполнения для механизма бинарной трансляции, это переключение между исполнением гостевого кода в трансляционном режиме и в режимом обычной эмуляции. Необходимость таких переключений возникает в нескольких случаях. Самый частый вариант, это возникновение инструкции, для которой трансляция не определена, например, инструкции переключения режимов процессора. Однако важнейшим источником переключений для транслятора являются переходы между блоками транслируемого кода. [3]

1. Обзор существующих решений

На момент выполнения работы, автору известно о двух возможных альтернативах разработанному программному комплексу. Первый и самый важный, это проект с открытым исходным кодом Quick Emulation (QEMU). [1]

1.1 QEMU

QEMU предлагает пользователям полную независимость от архитектуры, за счет системы промежуточного представления кода. Проект разделен на две части, фронт-энд - занимается обработкой эмулируемого кода и представлением его в виде промежуточного кода, и бэк-энд, который компилирует промежуточный код, применяет оптимизации и отдает объектные куски на исполнение реальной машины в её командах.

Все это работает во время исполнения. В качестве бэк-энда QEMU использует не общедоступные наборы компиляторов, а специализированный: tiny code generator, что позволяет контролировать выходной поток инструкций в обмен на архитектурно-специфические оптимизации.

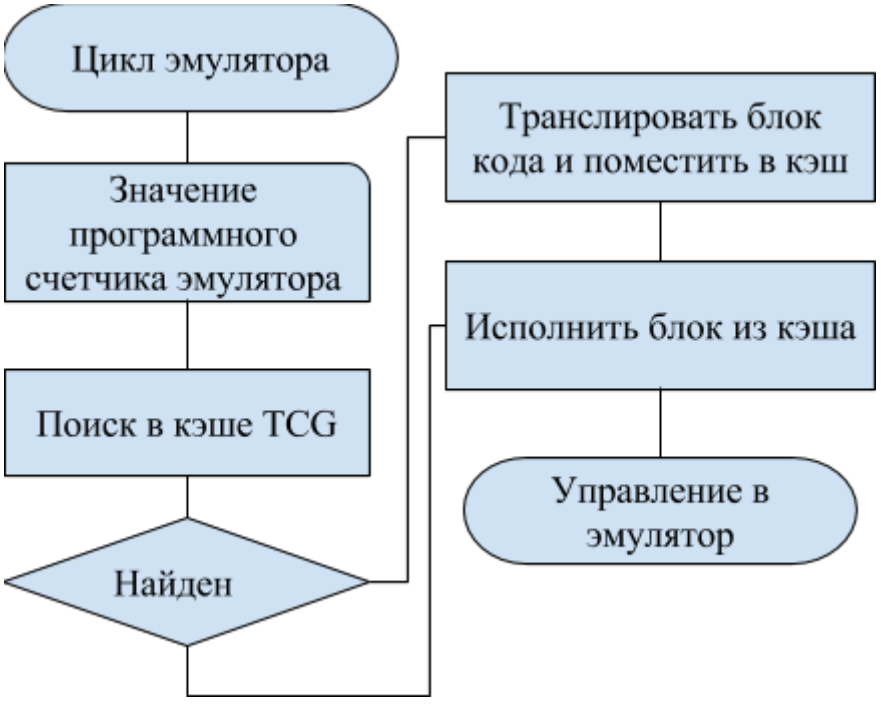


Иллюстрация 1. Схема работы транслятора QEMU

Фронт-энд QEMU для трансляции инструкций гостевой архитектуры использует представление в виде микроопераций. Микрооперации представляют собой специфический вариант низкоуровневого языка с простой трехоперандной адресацией и огромным набором временных регистров.

Помимо микроопераций, разработчики QEMU используют специальные вспомогательные прекомпилированные функции для обработки “сложных” инструкций.

Проект предлагает как полную эмуляцию всей системы, так и эмулирование исключительно пространства пользователя. Для обоих вариантов эмуляции, QEMU предлагает отладочные инструменты в виде интерфейса для отладчика GNU Debugger (GDB).

Как будет видно позднее, несмотря на большие возможности и поддержку открытого сообщества, для заявленных целей Quick Emulation обладает слишком низкой производительностью.

1.2 Houdini

Второй альтернативой является проприетарная библиотека Houdini, предоставляемая Intel производителям устройств на операционной системе Android и микропроцессорах архитектуры x86 для установки в качестве слоя совместимости между Android NDK и пользовательскими приложениями, которые не имеют собранных под архитектуру x86 исполняемых файлов и библиотек в пакете приложения.

Документация для libhoudini практически отсутствует в открытом доступе.

Известно, что libhoudini используется нативной Java виртуальной машиной Android в качестве бэкэнда при невозможности разрешить пути до библиотек x86 в файле приложения.

Модули libhoudini для сравнения производительности были получены из проекта Android x86.

Различные эксперименты, в том числе проведенные внутри компании ARM, так и независимой группой специалистов из Китая, показывают потерю производительности от 40% до 90% между использованием libhoudini для трансляции и использованием библиотек специально собранных под архитектуру x86. [14]

2. Архитектура системы команд Thumb

ARM (v7) является архитектурой с сокращенным набором команд (Reduced Instruction Set Computer, RISC). Размер машинного слова составляет 32 бита, размер команды фиксирован и вмещается в слово или половину слова. Обычные инструкции принимают в качестве операндов только регистры и непосредственно числа. Работа с памятью осуществляется при помощи специализированных инструкций. [12]

Важным аспектом данного набора команд является возможность условного выполнения практически любой инструкции. Первые четыре бита кодируют условие относительно флагов состояния процессора, которое декодер сверяет для каждой загруженной команды.

Все это позволяет набору команд arm достигать высокой производительности и относительной простоты реализации в аппаратном виде. В качестве стороннего эффекта это значительно облегчает эмуляцию по сравнению с x86 архитектурой.

Однако, огромные затраты вторичной памяти на приложения вынудили инженеров создать альтернативу в виде архитектуры системы команд Thumb, и в дальнейшем, расширение Thumb-2.

Все инструкции первой версии этого набора команд имели размер в половину машинного слова (за исключением инструкции условного перехода, которая выполняется в два такта). По сути они являлись сокращениями наиболее часто используемых инструкций оригинального набора.

Возможность условного исполнения каждой инструкции также отключена для Thumb. Начиная со второй версии расширений, она заменена специальной инструкцией IT (If-Then), которая использует контекст процессора для условного выполнения до четырех последующих за ней инструкций.

```
if ConditionPassed() then
d = UInt(Rdn);
n = UInt(Rdn);
m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
setflags = !InITBlock();
shifted = Shift(R[m], shift_t, shift_n, APSR.C);
(result, carry, overflow) = AddWithCarry(R[n], shifted, APSR.C);
if d == 15 then
// Can only occur for ARM encoding
ALUWritePC(result); // setflags is always FALSE here
else
R[d] = result;
if setflags then
APSR.N = result<31>;
APSR.Z = IsZeroBit(result);
APSR.C = carry;
APSR.V = overflow;
```

Листинг 1. Сравнение псевдокода для аналогов инструкции Add with Carry (ADC) между arm и Thumb

```
if ConditionPassed() then
if Rd == '1111' && S == '1' then SUBS PC, LR;
d = UInt(Rd);
 n = UInt(Rn);
 m = UInt(Rm);
 setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
shifted = Shift(R[m], shift_t, shift_n, APSR.C);
(result, carry, overflow) = AddWithCarry(R[n], shifted, APSR.C);
if d == 15 then
// Can only occur for ARM encoding
ALUWritePC(result); // setflags is always FALSE here
else
R[d] = result;
if setflags then
APSR.N = result<31>;
APSR.Z = IsZeroBit(result);
APSR.C = carry;
APSR.V = overflow;
```

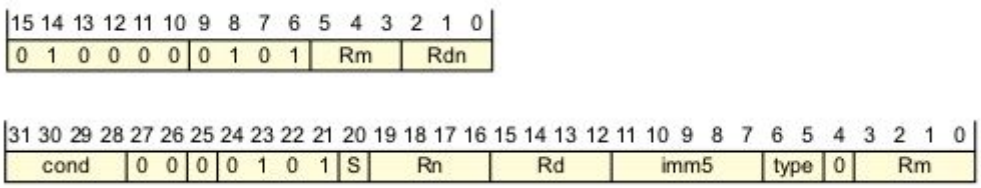


Иллюстрация 2. Сравнение кодировки операции между Thumb (сверху) и arm(снизу).[12]

В отличии от arm, где для арифметико-логических операций обновление флагов состояния кодируется при помощи специального бита в инструкции, для арифметических операций Thumb выставление флагов является поведением по-умолчанию.

Исключением является их исполнение в блоке условного выполнения If Then, и инструкции прямого сравнения CMP.

Еще одним важным отличием от основного набора инструкций является отсутствие сдвигов, встроенных в опкод инструкции.

В оригинальной системе команд arm, второй операнд может быть подвергнут арифметическому либо логическому сдвигу без использования дополнительных инструкций. Это позволяет расширить количество представлений для непосредственных значений, которое ограничено. Очевидно, что команда занимающая целое машинное слово не может включать в себя непосредственное значение объемом тоже в целое машинное и слово, и при этом иметь смысл.

Для обхода этого ограничения в Thumb введены специальные инструкции LSL, LSR, и т.д.

```
Disassembly of section .text:

00000240 <foo>:
 240:   b580            push    {r7, lr}
 242:   b082            sub     sp, #8
 244:   af00            add     r7, sp, #0
 246:   2300            movs    r3, #0
 248:   607b            str     r3, [r7, #4]
 24a:   687b            ldr     r3, [r7, #4]
 24c:   3301            adds    r3, #1
 24e:   607b            str     r3, [r7, #4]
 250:   46c0            nop                     ; (mov r8, r8)
 252:   46bd            mov     sp, r7
 254:   b002            add     sp, #8
 256:   bd80            pop     {r7, pc}

00000258 <__real_main>:
 258:   b580            push    {r7, lr}
 25a:   b082            sub     sp, #8
 25c:   af00            add     r7, sp, #0
 25e:   f7ff ffef       bl      240 <foo>
 262:   2300            movs    r3, #0
 264:   607b            str     r3, [r7, #4]
 266:   687b            ldr     r3, [r7, #4]
 268:   3301            adds    r3, #1
 26a:   607b            str     r3, [r7, #4]
 26c:   2300            movs    r3, #0
 26e:   0018            movs    r0, r3
 270:   46bd            mov     sp, r7
 272:   b002            add     sp, #8
 274:   bd80            pop     {r7, pc}
       ...

00000278 <main>:
 278:   e59fc004        ldr     ip, [pc, #4]    ; 284 <main+0xc>
 27c:   e08cc00f        add     ip, ip, pc
 280:   e12fff1c        bx      ip
 284:   fffffd5         .word   0xfffffd5
```

Листинг 2. Пример дизассемблированного машинного кода, собранного gcc для машины, поддерживающей первую версию Thumb.

```
Disassembly of section .text:

00000240 <foo>:
 240:   e52db004        push    {fp}            ; (str fp, [sp, #-4]!)
 244:   e28db000        add     fp, sp, #0
 248:   e24dd00c        sub     sp, sp, #12
 24c:   e3a03000        mov     r3, #0
 250:   e50b3008        str     r3, [fp, #-8]
 254:   e51b3008        ldr     r3, [fp, #-8]
 258:   e2833001        add     r3, r3, #1
 25c:   e50b3008        str     r3, [fp, #-8]
 260:   e1a00000        nop                     ; (mov r0, r0)
 264:   e28bd000        add     sp, fp, #0
 268:   e49db004        pop     {fp}            ; (ldr fp, [sp], #4)
 26c:   e12fff1e        bx      lr

00000270 <main>:
 270:   e92d4800        push    {fp, lr}
 274:   e28db004        add     fp, sp, #4
 278:   e24dd008        sub     sp, sp, #8
 27c:   ebffffef        bl      240 <foo>
 280:   e3a03000        mov     r3, #0
 284:   e50b3008        str     r3, [fp, #-8]
 288:   e51b3008        ldr     r3, [fp, #-8]
 28c:   e2833001        add     r3, r3, #1
 290:   e50b3008        str     r3, [fp, #-8]
 294:   e3a03000        mov     r3, #0
 298:   e1a00003        mov     r0, r3
 29c:   e24bd004        sub     sp, fp, #4
 2a0:   e8bd8800        pop     {fp, pc}
```

Листинг 3. Тот же код, выраженный в arm

Сокращенный размер инструкции обеспечивает меньше места на кодирование операндов. Даже отказ от трех-регистровой схемы “регистр назначения”:”регистр первого операнда”:”регистр второго операнда”, применяемой в оригинальной системе команд, не позволяет всем командам Thumb адресовать все регистры процессора.

Потому для этой системы команд, регистровый файл разбили на две части, верхнюю: регистры с восьмого по шестнадцатый (Hi), и нижнюю: с нулевого по седьмой (Lo).

В отличии от x86, где компилятору доступны всего восемь целочисленных регистров общего назначения, с большим списком оговорок и конвенциальных использований для половины из них[9], в ARM регистровый файл состоит из пятнадцати регистров общего назначения, плюс двух специализированных регистров. Банк регистров так же различен для разных режимов.

	User	System	Hyp †	Supervisor	Abort	Undefined	Monitor ‡	IRQ	FIQ
R0	R0_usr								
R1	R1_usr								
R2	R2_usr								
R3	R3_usr								
R4	R4_usr								
R5	R5_usr								
R6	R6_usr								
R7	R7_usr								
R8	R8_usr								R8_fiq
R9	R9_usr								R9_fiq
R10	R10_usr								R10_fiq
R11	R11_usr								R11_fiq
R12	R12_usr								R12_fiq
SP	SP_usr		SP_hyp	SP_svc	SP_abt	SP_und	SP_mon	SP_irq	SP_fiq
LR	LR_usr			LR_svc	LR_abt	LR_und	LR_mon	LR_irq	LR_fiq
PC	PC								
APSR	CPSR								
			SPSR_hyp	SPSR_svc	SPSR_abt	SPSR_und	SPSR_mon	SPSR_irq	SPSR_fiq
			ELR_hyp						

Иллюстрация 3, банк регистров архитектуры ARM [12]

Программный счетчик и регистр флагов являются единственными специальными регистрами; все остальные подчиняются общим правилам. Исторически сложилось использование тринадцатого регистра для хранения указателя стека (Stack Pointer, SP), а четырнадцатого для хранения адреса возврата при вызове процедур (Link Register, LR).

Для инструкций arm это не более чем конвенция. Компилятор или программист могут использовать их по своему усмотрению.

Однако для первой версии набора команд Thumb это не так. В большинстве своем, этим инструкциям доступны для адресации только нижние регистры, а тринадцатый адресуется особым образом и всегда используется только для хранения указателя стека.

Вторая версия расширений Thumb, помимо инструкции условного исполнения IT, добавила множество инструкций размером в машинное слово.

Эти дополнения сделали из набора команд Thumb полноценную архитектуру команд, позволив закодировать ей весь исполняемый файл целиком, без процедур-оболочек на arm. (см. Листинг 2 и Листинг 3)

Они по прежнему не допускают условного исполнения, зашифрованного прямо в инструкции. Для этого используется IT.

Помимо множества сервисных инструкций, вторая версия Thumb включает в себя расширенные до слова арифметическо-логические инструкции. Они, как и arm-версии, позволяют использовать сдвиги закодированные прямо в инструкции, а также, для непосредственных значений, могут шифровать их используя паттерны часто используемых значений, расширяя один байт до целого слова.

```
Disassembly of section .text:

00000240 <foo>:
 240:   b480            push    {r7}
 242:   b083            sub     sp, #12
 244:   af00            add     r7, sp, #0
 246:   2300            movs    r3, #0
 248:   607b            str     r3, [r7, #4]
 24a:   687b            ldr     r3, [r7, #4]
 24c:   f103 13ff       add.w   r3, r3, #16711935       ; 0xff00ff
 250:   607b            str     r3, [r7, #4]
 252:   bf00            nop
 254:   370c            adds    r7, #12
 256:   46bd            mov     sp, r7
 258:   bc80            pop     {r7}
 25a:   4770            bx      lr

0000025c <main>:
 25c:   b580            push    {r7, lr}
 25e:   b082            sub     sp, #8
 260:   af00            add     r7, sp, #0
 262:   f7ff ffed       bl      240 <foo>
 266:   2300            movs    r3, #0
 268:   607b            str     r3, [r7, #4]
 26a:   687b            ldr     r3, [r7, #4]
 26c:   3301            adds    r3, #1
 26e:   607b            str     r3, [r7, #4]
 270:   2300            movs    r3, #0
 272:   4618            mov     r0, r3
 274:   3708            adds    r7, #8
 276:   46bd            mov     sp, r7
 278:   bd80            pop     {r7, pc}
```

Листинг 4. Дизассемблер примитивной программы, закодированной gcc под thumb-2.

Главным же дополнением второй версии расширения Thumb стала возможность одновременной работы инструкций размером в слово и полуслово внутри одной процедуры, с одновременным расширением списка инструкций, которые поддерживают переход, переключающий режим процессора между двумя архитектурами системы команд.

3. Транслятор

3.1 Теоретическое основание

За основание, на которое опирается данная работа, является теорема о виртуализируемости и последующие работы. Архитектура Parallels ARM Emulator представляет собой эмулирующий монитор виртуальной машины с поддержкой статической (на текущий момент) двоичной трансляции для прикладного уровня.

Иллюстрация 4. Монитор виртуальной машины

Данная схема была выбрана из-за невозможности классической виртуализации архитектуры ARM на целевой архитектуре x86.

Классическая виртуализация (trap & emulate схема) предусматривает выполнение эмулируемого и реального кода в в одном и том же пространстве памяти, на одном и том же реальном устройстве, с сохранением контекста эмулируемой системы в заранее определенном участке памяти.

При этом обращения эмулируемого кода к участкам памяти реальной системы, смены привилегий и обращения к устройствам обрабатываются монитором отдельно (trap).

Для такой схемы была разработана модель, используя которую возможно доказательство возможности построения эффективного монитора виртуальной машины для заданной архитектуры. [11]

Такой монитор обладает тремя важнейшими характеристиками. Первое, он предоставляет среду выполнения, не отличимую от настоящего аппаратного обеспечения.

Второе, замедление исполнения программ в контролируемой монитором среде исполнения должно быть минимальным и идеально различия должны быть не детектируемы изнутри среды.

Третье, монитор должен обладать абсолютным контролем над ресурсами предоставляемым виртуальной среде.

Существуют дополнения данной схемы, которые расширяют теоретический базис на более современные модели архитектур и также рассматривают возможность кроссплатформенной виртуализации (с отличающимися эмулируемыми и реальными архитектурами). [16]

Однако в данном случае классическая виртуализация попросту невозможна, что доказывается в работе Нилса Пеннмана и прочих. [15] Основной причиной для этого является наличие в системах команд таких инструкций, как SVC, SEV, WFE и других, которые являются чувствительными инструкциями, доступным пользователю, что нарушает условия теоремы о виртуализируемости.

Виртуальной машиной называется среда исполнения, создаваемая монитором виртуальных машин. В нашем случае пользовательские приложения видят среду исполнения как машину с процессором generic ARM архитектуры и конфигурируемым набором остальной аппаратуры, чье пользовательское

окружение (набор библиотек и операционная система) полностью повторяет данное у реальной системы, на которой запущен эмулятор. [11]

Состояние виртуальной машины в любой момент времени описывается четверкой

$$S = \{E, M, P, R\}, S \in C$$

Где E это линейная область исполняемой памяти, M это контекст процессора (состоящий как минимум из двух режимов: привилегированного и пользовательского), P программный счетчик, и R - схема адресации в E. Множество C - конечное множество состояний.

Инструкцией классическая теория виртуальных машин называет отображение $i : C \to C$. Инструкции могут быть перехвачены процессором в случае недопустимости с точки зрения уровня привилегий, или доступа к неправильному или запрещенному участку памяти. При перехвате, управление передается в заранее определенный обработчик в памяти, а состояние системы сохраняется для восстановления после обработки.

Все инструкции в заданной система набора команд делятся на три класса.

Первый - привилегированные инструкции, которые выполняются корректно в привилегированном (заданном множестве таковых) режиме, и перехватываются во всех остальных.

Второй - чувствительные инструкции, класс состоящий из контроль-чувствительных и поведение-чувственных инструкций. Контроль-чувственные инструкции пытаются изменить режим исполнения, в котором находится система, либо количество доступной памяти. Поведение-чувственные могут иметь различный результат выполнения, в зависимости от их положения в памяти, либо от режима исполнения, в котором находится система.

Третий класс состоит из всех остальных инструкций.

Как было сказано выше, обе системы команд в архитектуре ARM содержат инструкции, которые являются чувствительными, но не являются привилегированными. Примерами таких инструкций являются SEV, SVC, WFI, MCR, LDC, и т.д.

В своей работе Пеннман и соавторы отмечают два возможных пути для эффективной виртуализации ARM, а именно паравиртуализацию и динамическую двоичную трансляцию.

Паравиртуализацией называют методику системной виртуализации, которая включает в себя использование модифицированной версии операционной системы для запуска внутри виртуальной машины. Обычный набор модификаций включает в себя адаптацию кода системы к работе на более низком машинном уровне привилегий.

Для упрощения паравиртуализации используются аппаратные расширения. Данная методика рассматривается в их работе с точки зрения ARM-на-ARM полносистемной виртуализации, и не представляет интереса в рамках выполнения данной работы.

Полная виртуализация также достижима и без поддержки со стороны оборудования, используя двоичную трансляцию. Это продолжение методики гибридного монитора из классической теории, для реализуемости которого также доказан критерий.

Гибридный монитор виртуальной машины это такой монитор, который позволяет обычное исполнение вне привилегированного режима, но эмулирует выполнение инструкций в привилегированном.

Так, например, классический гибридный монитор исполнял бы инструкции операционной системы как эмулятор, а пользовательские приложения за редким исключением запускал бы как обычный монитор.

Критерием для построения гибридного монитора является привилегированность еще одного введенного класса инструкций, чувствительных к пользователю.

Обновленная модель виртуальной машины, используемая в данной работе, выглядит следующим образом.

Состояние в любой момент времени описывается в виде кортежа восьмерки:

$$S = \{E, M, P, G, C, A, D_m, D_p\}$$

, где первые три параметра сохраняют свой первоначальный смысл. Однако, множество режимов процессора теперь делится на m_u - непривилегированный пользовательский режим, и множество привилегированных режимов M_p:

$$M = \left(M_p \bigcup m_u\right),\ m_u \notin M_p$$

Множество G определяет состояния всех регистров общего назначения, кроме, соответственно, программного счетчика.

Множество C определяет состояние всех конфигурационных регистров.

Доступ к памяти E теперь осуществляется через карту трансляции виртуальной памяти A.

Последние два множества определяют состояние устройств ввода-вывода.

Карта трансляции адресов A представляет собой множество троек

$$(v, p, x),\ v \in V,\ p \in P,\ x \in X$$

Где v это адрес виртуальной памяти, p это адрес в физической памяти, x определяет разрешения доступа. Для любого v, существует хотя-бы одна такая тройка в A, что содержит v. Функция $T_A(v)$ определяет отображение адреса из согласно этим тройкам. Если v нету в A, или x запрещает доступ, происходит перехват по доступу к памяти.

В обновленной модели мы можем различать дополнительные подклассы чувствительных инструкций, такие как чувствительность по конфигурации и по пользователю.

Чувствительность к конфигурации означает зависимость исхода инструкции от состояния конфигурационных регистров. В формальном виде, инструкция называется чувствительной к конфигурации в том случае, когда для некоторых двух состояний

$$S_1 = (e,\ m,\ p,\ g,\ c_1,\ a,\ dm,\ dp)$$

и

$$S_2 = (e,\ m,\ p,\ g,\ c_2,\ a,\ dm,\ dp)$$

в которых не происходит перехватывания по доступу к памяти, верно, что

$$i(S_1) \neq i(S_2)$$

Введем определение инструкции, чувствительной к пользователю. Инструкция i(S) называется чувствительной к пользователю, если существует состояние S такое, что i(S) чувствительно по контролю или/и чувствительно к конфигурации.

Дополнив классы чувствительных и привилегированных функций, можно расширить на эту модель результаты, доказанные для старой модели. [15]

Во-первых, анализ наборов команд Thumb показывает, что в нём существует четыре инструкции, чувствительные по пользователю:

1. SVC - переводит процессор в привилегированный режим supervisor
2. SEV - необязательная для имплементации инструкция-подсказка, часть механизма энергосбережения в мультипроцессорных системах.
3. WFE - то же самое
4. WFI - необязательная для имплементации инструкция-подсказка, часть механизма энергосбережения.

Первая из списка является привилегированной, однако остальные три - нет. Это означает, в свою очередь, невозможность построения гибридного монитора виртуальной машины для системы команд Thumb.

Однако, руководство разработчика ARM упоминает, что оставшиеся три инструкции являются необязательными к исполнению на чипе, и в случае, когда они не поддерживаются, процессору предписывается заменить их на инструкцию пропуска такта NOP. Она не является чувствительной, а значит, построение гибридного монитора виртуальной машины возможно с соблюдением условия игнорирования данных трех инструкций. Потери каких-то свойств монитора при этом не произойдет, так как функциональность энергосбережения в модель не входит.

3.2 Общая схема работы

Монитор виртуальной машины и сама виртуальная машина исполняется в разных потоках.

На вход монитор виртуальной машины принимает исполняемый файл собранный для машины под управлением ARM микропроцессора и путь до системных библиотек.

Исполняемый файл анализируется на наличие зависимостей от библиотечных функций, составляется карта релокаций, зависимости обрабатываются и разрешаются относительно библиотек предоставленных монитору.

После этого образ файла загружается в память и начинается его предварительная обработка.

Первым этапом обработки является анализ потока исполнения. Каждая инструкция, начиная с входной инструкции исполняемого файла, декодируется, и по специальной таблице категорируется. На данном этапе происходит разбиение исполняемого файла на непрерывные блоки инструкций. В первую очередь отбирается множество непривилегированных инструкций. Т.е. Доступных для исполнения в пользовательском режиме.

Далее анализатор интересуют различные виды инструкций, изменяющих поток управления. Условные и безусловные переходы, точки возврата и т.д. Такие инструкции маркируют начало и конец непрерывного блока исполнения.

Третья интересующая анализатор категория это “сложные” функции: такие, которые могут изменять контекст процессора, а также актуальные для ARM инструкции работы с памятью. В дальнейшем они либо будут сигнализировать прекращение трансляции и переход в режим эмуляции, либо изменяют свою категорию в зависимости от контекста анализатора.

Анализатор также вычисляет адреса переходов в зависимости от инструкций.

После построения и линковки дерева блоков, начинается этап анализа использования регистров. Каждая инструкция в дереве обрабатывается согласно предустановленному для нее правилу, и для нее сохраняется карта использования регистров ARM <-> x86.

Далее идет прогон транслятора. Транслятор выделяет память в кэше трансляции, для каждого блока согласует карты использования регистров и записывает в кэш код каждой инструкции вместе с необходимыми системными заглушками и ретранслированными адресами вызовов. Если что-то на данном этапе не прошло, например, карты использования регистров не могут быть согласованы, транслятор помечает весь блок как не транслированный и в дальнейшем он будет исполняться в режиме эмуляции.

```
Dump of assembler code from 0xf7fbd000 to 0xf7fbd08a:
   0xf7fbd000:          mov    0x0(%ebp),%ebx
   0xf7fbd003:          mov    0x18(%ebp),%edx
   0xf7fbd006:          mov    0x58(%ebp),%eax
   0xf7fbd009:          mov    0x68(%ebp),%ecx
   0xf7fbd00c:          mov    0x70(%ebp),%esi
   0xf7fbd00f:sub     $0x4,%ecx
   0xf7fbd015:          mov    %eax,(%ecx)
   0xf7fbd017:          mov    %ecx,%eax
   0xf7fbd019:          add    $0x0,%eax
   0xf7fbd01f:sub     $0xc,%ecx
   0xf7fbd025:          mov    $0x0,%edx
   0xf7fbd02b:          sub    $0x8,%eax
   0xf7fbd031:          mov    %edx,(%eax)
   0xf7fbd033:          add    $0x8,%eax
   0xf7fbd039:          sub    $0x8,%eax
   0xf7fbd03f:mov     (%eax),%edx
   0xf7fbd041:          add    $0x8,%eax
   0xf7fbd047:          add    $0xff0000,%edx
   0xf7fbd04d:          add    $0xff,%edx
   0xf7fbd053:          sub    $0x8,%eax
   0xf7fbd059:          mov    %edx,(%eax)
   0xf7fbd05b:          add    $0x8,%eax
   0xf7fbd061:          mov    %eax,%ecx
   0xf7fbd063:          add    $0x0,%ecx
   0xf7fbd069:          mov    (%ecx),%eax
   0xf7fbd06b:          add    $0x4,%ecx
   0xf7fbd071:          sub    $0x4,%ecx
   0xf7fbd077:          mov    %esi,0x78(%ebp)
   0xf7fbd07a:          mov    %ebx,0x0(%ebp)
   0xf7fbd07d:          mov    %edx,0x18(%ebp)
   0xf7fbd080:          mov    %eax,0x58(%ebp)
   0xf7fbd083:          mov    %ecx,0x68(%ebp)
   0xf7fbd086:          mov    %esi,0x70(%ebp)
   0xf7fbd089:          ret
End of assembler dump.
```

Листинг 5, Транслированная в x86 с Thumb функция, хранимая в памяти эмулятора.

Очень важным для эффективной работы этой стадии является процесс декодирования инструкций. Он осуществляется с помощью специальных, составленных вручную таблиц операционных кодов и процедур обратного вызова, по которым в момент загрузки монитора виртуальной машины строятся поисковые хеш-таблицы.

Каждая инструкция всех систем команд была проанализирована вручную и для неё были написаны соответствующие процедуры обратного вызова.

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

  struct {
    user_cfa_insn_flags_t user_cfa_flags;
    void *user_cfa_callback;

    regs_usage_handle_t regs_usage_callback;
    translation_handle_t translator_callback;  };

} instr_desc_t;
// ...
instr_desc_t thumb_instr_table[] = {
{ 0x0000ffc0, 0x00004140, 0, 0, "adc",    1,      { EMU_CB(thumb_adc_reg,          NULL)
USER_TRANSLATE_CB(trans_thumb_adc, reg_usage_thumb_data)}},       // register,
ARMv4T, >=ARMv5T*
{ 0x0000fe00, 0x00001c00, 0, 0, "add",    1,      { EMU_CB(thumb_add_imm1,
NULL) USER_CFA_FLAGS(.is_complex = 1)
USER_TRANSLATE_CB(trans_thumb_add_imm, reg_usage_thumb_arithm) }},       //
immediate, ARMv4T, >=ARMv5T*
{ 0x0000f800, 0x00003000, 0, 0, "add",    1,      { EMU_CB(thumb_add_imm2,
NULL) USER_CFA_FLAGS(.is_complex = 1)
USER_TRANSLATE_CB(trans_thumb_add_imm, reg_usage_thumb_arithm) }},       //
immediate, ARMv4T, >=ARMv5T*
//...
}
```

Листинг 6. Структура описания инструкции и часть таблицы декодирования

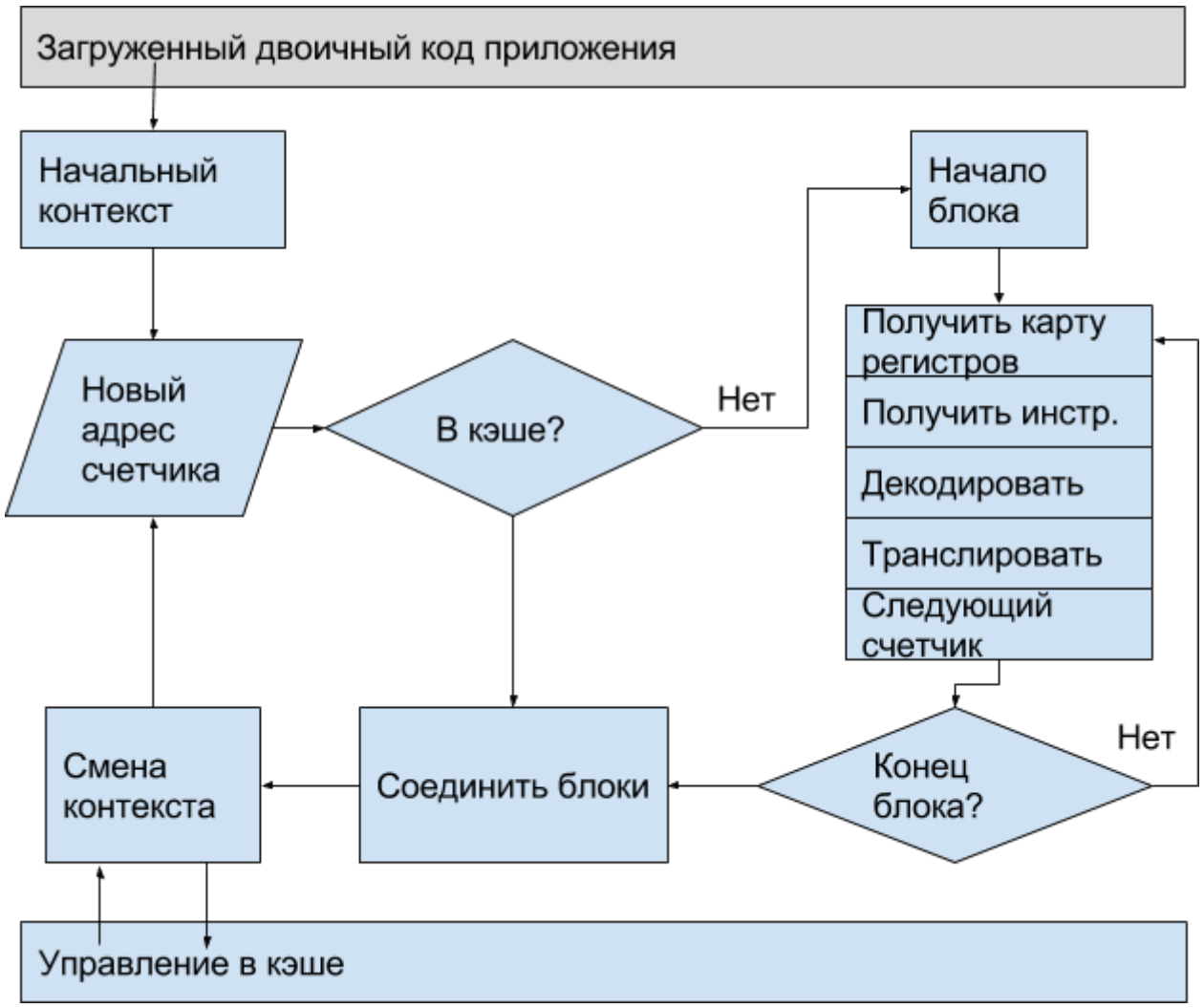


Иллюстрация 5. Схема потока данных в бинарной трансляции

После обработки данные таблицы приобретают несколько уровней, в зависимости от коллизий. Вместо классической хеш-функции используется наложение специализированных под каждую систему команд масок.

Когда весь исполняемый файл был обработан, в отдельном потоке запускается виртуальная машина, в которой начинается исполнение с входного блока. Блоки слинкованы прямо в кэше монитора виртуальной машины - выход в режим эмуляции происходит только в случае неподдерживаемой инструкции, либо исключения, либо ошибки.

Одним из возможных исключений является вызов библиотечной функции. Важнейшим преимуществом Parallels ARM Emulator является возможность

напрямую использовать системные библиотеки, собранные под архитектуру x86, из эмулятора.

```
translated_block_t *trans_block = block->translated_data;
addr_t addr = dynamic_mapping (trans_block);
asm volatile(
  "pusha \n"
  "mov %0, %%eax \n"
  "mov %1, %%ebp \n"
  "call *%%eax \n"
  "popa \n"
  ::"r"((uint32_t)addr), "r"((uint32_t)(&(vcpu->ctx)))
  :"eax"
);
processor_set_cond_flags(vcpu, vcpu->ctx.CPSR);
SET_PC(vcpu, GET_REG(vcpu, REGPC));
```

Листинг 7. Обработчик смены контекста между монитором и транслированным блоком из кэша.

```
static void *main_vcpu_thread(void *arg)
{
  vcpu_t *vcpu = (vcpu_t *)arg;
  uint32_t break_condition;

  set_vcpu_id(vcpu->num);

  while (!(break_condition = handle_async_events(vcpu))) {
     if (!native_execute || native_execute(vcpu))
        cpu_execute_op(vcpu);
  };

  sync_lock(&main_lock);
  vcpu->break_condition = break_condition;
  sync_wake(&main_sync);
  sync_unlock(&main_lock);

  return NULL;
}
```

Листинг 8. Основной цикл виртуальной машины

```
trans_status_t trans_thumb_add_imm(trans_insn_t *insn,
x86_reg_t reg_map[ARM_REG_COUNT], uint8_t **write_ptr)
{
  arm_reg_t rd = REG_NONE, rn = REG_NONE;
  uint32_t imm;

  if (GET_BIT(insn->op, 1))
     THUMB_DECODE_IMM3(insn->op, rd, rn, imm);
  else
     THUMB_DECODE_IMM8(insn->op, rd, imm);

  if (REGPC == rn)
     MOVD_REG_IMM32(write_ptr, reg_map[rd], insn->addr + 8);
  else
     MOV_REG_REG(write_ptr, reg_map[rd], reg_map[rn]);

  ADD_REG_IMM32(write_ptr, reg_map[rd], imm);

  if (REGPC == rd)
     MOVD_MEM_REG_DISP8(write_ptr, REG_EBP, reg_map[rd],
ARM_REG_ADDR_DISP8(REGPC));

  return TRANS_SUCCESS;
```

Листинг 9. Пример процедуры обратного вызова для трансляции кода

```
/*
* As per A6.2.5
* Miscellaneous 16-bit instructions
* Thumb 16-bit
*/
void reg_usage_thumb_misc(trans_insn_t *insn)
{
  const uint32_t opcode = (insn->op >> 8) & 0xf;
  switch (opcode)
  {
     case 0b0000: // add/sub sp imm
        REG_TMP_NEEDED(insn);
        break;
     case 0b0001: // cbz/cbnz
     case 0b0011:
     case 0b1001:
     case 0b1011:
        REG_TMP_NEEDED(insn);
        UPDATE_REG_USAGE_MASK(insn, (insn->op) & 0x7);
        break;
     case 0b0010: // stx(hb)/utx(hb)
     case 0b1010: // rev(16/sh)
        UPDATE_REG_USAGE_MASK(insn, (insn->op) & 0x7);
        UPDATE_REG_USAGE_MASK(insn, (insn->op >> 3) & 0x7);
        break;
     case 0b1100: // pop mult
     case 0b1101:
     case 0b0100: // push mult
     case 0b0101:
        REG_TMP_NEEDED(insn);
        break;
  }
}
```

Листинг 10. Пример процедуры обратного вызова для анализа использования регистров.

```
trans_status_t trans_thumb_cmp_imm(trans_insn_t *insn,
                                   x86_reg_t
reg_map[ARM_REG_COUNT],
                                   uint8_t **write_ptr)
{
  uint32_t _imm;
  arm_reg_t _rn;
  trans_insn_t *dummy_insn;
  INSTRUCTION_COPY(insn, dummy_insn);

  THUMB_ENCODE_PROLOGUE(write_ptr, insn);

  thumb_decode_args_to_arm_imm(insn->op, dummy_insn->op);

  trans_status_t ret = trans_arm_cmp_imm(dummy_insn,
                                         reg_map,
                                         write_ptr);

  If (ret == TRANS_SUCCESS)
  {
    THUMB_ENCODE_EPILOGUE(write_ptr, insn);
  }

  return ret;
}
```

Листинг 11а. Вызов для трансляции кода набора команд arm и соответствующий ему обработчик thumb.

```
trans_status_t trans_arm_cmp_imm(trans_insn_t *insn, x86_reg_t
reg_map[ARM_REG_COUNT], uint8_t **write_ptr)
{
  uint32_t imm;
  arm_reg_t rn;
  ARM_DECODE_ARGS_IMM_NI(insn->op, rn, imm);

  if (rn == REGPC)
     MOVD_REG_IMM32(write_ptr, REG_TMP, insn->addr + 8);
  else
     MOV_REG_REG(write_ptr, REG_TMP, reg_map[rn]);

  SUB_REG_IMM32(write_ptr, REG_TMP, imm);

  UPDATE_COND_FLAGS(write_ptr, REG_TMP, REG_EBP);

  return TRANS_SUCCESS;
}
```

Листинг 11б.Вызов для трансляции кода набора команд arm и соответствующий ему обработчик thumb.

Как видно из листинга 5, монитор виртуальной машины предоставляет текущий контекст процессора в виде структуры, для хранения адреса которой используется регистр EBP. Это дополнительно ограничивает количество доступных для транслятора регистров общего назначения.

Однако проблема составления карт регистров выходит за область данной работы.

3.3 Особенности реализации поддержки Thumb

Разработка поддержки архитектуры семейства команд Thumb была значительно облегчена наличием уже существующей и рабочей инфраструктуры транслятора системы команд arm.

Основная работа заключалась в анализе инструкций системы команд Thumb и составлении процедур обратного вызова для подсистем анализа потока исполнения, составления карты регистров и трансляции.

Внушительная часть инструкций первой версии Thumb имеет прямые аналоги в наборе команд arm и значит, что для их реализации можно воспользоваться уже существующими процедурами обратного вызова с правильной пре- и пост-обработкой карты регистров и входных данных инструкции.

Основную сложность вызывали инструкции второй версии системы команд Thumb, для которых такого отображения подчас не существовало из-за особенностей условного исполнения инструкцией If-Then и кодирования непосредственных значений для команд Thumb.

Анализ использования регистров был облегчен ограничениями, описанными в главе 2. Основной прирост производительности транслятора для Thumb системы команд ожидается благодаря более эффективному маппингу регистров. [10]

В то же время, “специальные” и “сервисные” инструкции Thumb потребовали модификации представления контекста процессора, в виде добавления в него соответствующих полей. В том числе флагов условного исполнения для инструкции IT.

Во время работы над процедурами обратного вызова для подсистемы анализа потока исполнения в ней были выявлены ряд существенных недостатков.

Все они были связаны с алгоритмической не оптимальностью используемых структур данных.

Поэтому вторая часть данной работы посвящена короткому описанию модификаций данной системы.

4. Анализатор потока исполнения

4.1 Описание анализатора

Данный модуль занимается представлением исполняемого файла в виде ориентированного графа блоков.

Вершины этого дерева содержат блоки: последовательности инструкций, которые гарантированно должны быть исполнены от начала до конца блока одна за одной.

Ребра, соединяющие вершины, обозначают условный переход, вызов функции, либо возврат из вызова, плюс вырожденный случай выхода из режима трансляции.

На вход модуль принимает загруженный в память образ исполняемого файла и адрес точки входа.

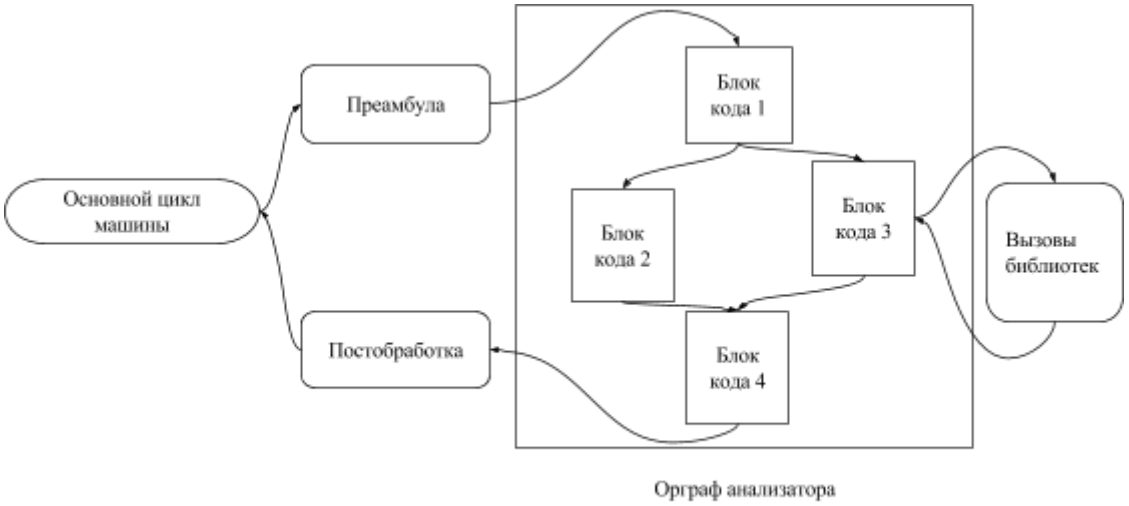
```
typedef struct source_block {
  /* translated blocks are organized in a list */
  SLIST_ENTRY(source_block) list;
  SLIST_ENTRY(source_block) block_in_func_list;

  /* the [begin; end) of this block */
  addr_t begin, end;

  struct source_block *next_block;
  struct source_block *branch_block;
  void *translated_data;
  int is_entry_point : 1;
  int next_insn_is_invalid : 1;
  int is_referenced : 1;
  int is_external_reference : 1;
  int is_final_block : 1;
} source_block_t;
```

Листинг 11. Структура, описывающие представление блока

Не смотря на это, как видно из листинга 6, в коде монитора виртуальной машины использовалось представление блоков в виде обыкновенных односвязных списков.



Орграф анализатора

Иллюстрация 6. Схема представления блоков в графе анализатора

Экспериментальным путем, после запуска анализатора на множестве использующихся в условиях производства исполняемых файлов, таких как различные сторонние графические библиотеки и движки, было выяснено, что в среднем, анализатор составляет один блок кода на ~73 инструкций.

Это означает, что при запуска 400kB исходника (встраиваемая поисковая библиотека Algolia) может быть создано до тысячи блоков. Для графических приложений, например, движка трехмерной графики Unity, объем библиотек может составлять пару мегабайт.

Конечно, данные числа приблизительны, и очевидно, что на практике такие прикидки сказываются слабо за счет того, что не все из этого места является исполняемым кодом.

Однако даже такие грубые прикидки показывают, что использование других структур данных позволит выиграть время на этапе работы анализатора.

4.2 Выбор структуры данных

Непосредственная работа с данным ориентированным графом, в основном осуществляется ради поиска в нем блоков по заданному логическому адресу

```
/*
* Find an untranslated block which contains given "addr".
* This is used to split blocks during code flow analysis.
*/
static source_block_t
*user_cfa_find_block_for_addr(translated_image_t *image,
addr_t addr);
```

Листинг 12. Дефиниция процедуры, являющейся “бутылочным горлышком”

Прямой перебор по односвязному списку очевидно является неэффективным подходом к решению данной задачи. [7]

Как можно увидеть на листинге 11, блок представляет собой суть последовательность прогрессирующих целочисленных адресов. Известно, что адреса непрерывны и не пересекаются, каждый блок определяется своим начальным и конечным адресом в последовательности.

Задача поиска среди множества блоков такого, в который входит заданный адрес, сводится к абстрактной задаче поиска интервала. Для решения этой задачи

обычно используется та или иная модификация структуры данных, называемой интервальным деревом. [4]

Выбор интервального дерева, основанного на самобалансирующемся красно-черном дереве, позволил улучшить общую производительность анализатора на Thumb-коде и больших исполняемых файлах.

5. Измерение производительности

Для сравнения полученного прироста производительности, необходимо было подобрать качественные сценарии ее оценки, которые всесторонне отразили бы результаты данной работы.

Ожидается, что имплементация двоичной трансляции для одной системы набора команд должна в результате дать значительный прирост производительности на исполняемых файлах, закодированных исключительно данным набором команд и не дать регрессий (т.е. потери свойств монитора виртуальной машины) либо сторонних программных ошибок.

Также ожидается, что на исполняемых файлах, закодированных смешанно, различными системами машинных команд, прирост скорости будет получен за счет:

1. Увеличения производительности Thumb-кодированных участков
2. Уменьшения количества переключений в режим эмуляции при переходе между Thumb и ARM базовыми блоками.

Не должно наблюдаться уменьшения производительности на исполняемых файлах, закодированных исключительно системой команд arm.

Для проверки данных гипотез был подобран специализированный набор тестовых исполняемых файлов. Он состоит из трех исполняемых файлов; один закодирован исключительно в Thumb, второй смешанный, третий исключительно arm.

```c
#include <stdio.h>

int next_prime(int i, int x[]) {
  while (x[i] == 0) i++;
  return i;
}

int main(int argc, char* argv[]) {
  int* primes;
  int i = 0;
  int j = 0;
  int k = 0;
  const int _end = 100000;

  for(i=2; i < _end; i++) {
    primes[i] = i;
//    printf("%d\n", i);
  }

  for(i=2; i < _end; i = next_prime(i+1, primes)) {
    for(j = (i*2); j < _end; j += i) {
//      printf("i: %d, j: %d\n", i, j);
      primes[j] = 0;
    }
  }

  for(i=2; i < _end; i++) {
    if ( primes[i] != 0 ) {
      printf("Prime number: %d\n", primes[i]);
    }
  }

  return 0;
}
```

Листинг 13. Исходный код примитивной программы для замера производительности целочисленных расчетов.

Код, из которого собраны данные исполняемые файлы, представляет собой примитивную реализацию решета Эратосфена. Выходной файл покрывает около двух третей арифметико-логических инструкций, имеет простую с точки зрения анализа потока исполнения структуру и не использует условное исполнение.

Для проверки первой гипотезы, Parallels ARM Emulator был собран в двух версиях, одна из которых поддерживала только эмуляцию Thumb, а другая поддерживала и эмуляцию, и двоичную трансляцию.

Для всех остальных измерений был использован свободный набор бенчмарков MiBench. [17] Его выбор был обусловлен простотой использования, специализацией на встраиваемые RISC системы, а также лицензированием.

Для сборки кода был использован кросс-компилятор, полученный от проекта GCC, версии 7.1.0.

```
GNU C99 (GCC) version 7.1.0 (arm-linux-gnueabi)
        compiled by GNU C version 7.0.1 20170421 (Red Hat 7.0.1-0.15), GMP version 6.1.2, MPFR version 3.1.5, MPC version 1.0.2, isl version isl-0.16.1-GMP

GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072

COLLECT_GCC_OPTIONS='-std=c99' '-fno-exceptions' '-fno-unwind-tables' '-nostartfiles' '-masm-syntax-unified' '-fno-pie' '-fno-gnu-tm' '-v' '-march=armv7ve' '-mtls-dialect=gnu'
```

Листинг 14. Версия и конфигурация используемого компилятора.

Сравнение производительности было также осуществлено относительно основных конкурентов (см. главу 2). В сравнении участвовали версии Quick Emulator версии 2.9

В качестве Houdini была использована поставка Android 5, сборка Android on x86, на устройстве Asus ZenFone 4 (A450CG), результаты были нормированы по тактовой частоте устройств.

Характеристики тестового стенда для остальных измерений указаны в листинге 15. Тестовый стенд находится под управлением операционной системы GNU\Linux.

```
$ cat /proc/version
Linux version 4.11.5-300.fc26.x86_64 (mockbuild@bkernel02.phx2.fedoraproject.org)
(gcc version 7.1.1 20170526 (Red Hat 7.1.1-2) (GCC) ) #1 SMP Wed Jun 14 19:16:35
UTC 2017
$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
Vendor ID:             GenuineIntel
Model:                 42
Model name:            Intel(R) Core(TM) i7-2760QM CPU @ 2.40GHz
Stepping:              7
CPU MHz:               799.951
CPU max MHz:           3500,0000
CPU min MHz:           800,0000
$ cat /proc/meminfo
MemTotal:       12182760 kB
```

Листинг 15. Характеристики тестового стенда для измерений производительности.

Заключение

Проведена профилировка производительности монитора виртуальных машин Parallels ARM Emulator. Анализ результатов профилировки и основных ожидаемых и целевых сценариев работы указал на нехватку поддержки архитектуры системы команд Thumb, а также на неоптимальное использование структур данных в модуле анализа потока исполнения.

Был предложен и имплементирован способ реализации поддержки системы команд Thumb основываясь на уже существующих наработках в архитектуре монитора виртуальных машин.

Существующий код модуля анализа потока исполнения был оптимизирован.

Были проведены эксперименты по сравнению производительности на синтетических тестах монитора виртуальных машин Parallels ARM Emulator и основных аналогов.

Синтетические тесты содержали арифметические операции, и не содержали операций ввода-вывода. Для тестирования прироста производительности был также собран тест со смешанным кодом систем команд arm и thumb.

Данные тесты показали прирост производительности после выполненных в данной работе оптимизаций приблизительно на 600% в случае исключительно Thumb кода и на ~27% в случае приближенного к реальному смешанного (interweaving) кода.

По сравнению с Quick Emulation (QEMU), Parallels ARM Emulator показал приблизительно до двух порядков лучшую скорость в обоих случаях.

По сравнению с libhoudini существенного отличия в производительности обнаружить не удалось. Возможно это свидетельствует об ошибке во время настройки тестового окружения.

Все это позволяет сказать о том, что поставленные в работе цели были успешно достигнуты. Предложенные в ходе работы и затем реализованные методики улучшают скоростные характеристики программного комплекса виртуализации Parallels ARM Emulator, что было успешно доказано в ходе измерений в финальной части работы.

Дальнейшее продолжение работ в данном направлении видиться возможным и актуальным. Касательно исключительно вопросов оптимизации бинарного транслятора, особенно перспективными видятся попытки дополнительных оптимизаций транслируемого машинного кода (вопрос изложен в [18], [19]).

Список литературы

[1] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator // Proceedings of the USENIX Annual Technical Conference 2005

[2] Adams K., Agesen O. A comparison of software and hardware techniques for x86 virtualization // ACM SIGPLAN Notices. 2006. № 11 (41). С. 2.

[3] Altman E., Kaeli D., Sheffer Y. Welcome to the opportunities of binary translation // Computer. 2000. № 3 (33). С. 40–45.

[4] De Berg, Mark, Marc Van Kreveld, Mark Overmars, and Otfried Cheong Schwarzkopf. "Computational geometry." In *Computational geometry*, pp. 1-17. Springer Berlin Heidelberg, 2000.

[5] Choi M. et al. Faster Translated Binary Execution on Mobile System through Virtualization // 2014 IEEE 28th International Conference on Advanced Information Networking and Applications. 2014.

[6] Choi M., Lim S.-H. x86-Android performance improvement for x86 smart mobile devices // Concurrency and Computation: Practice and Experience. 2014. № 10 (28). С. 2770–2780.

[7] Cormen T.H. et al. Introduction to algorithms / T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Cambridge (Massachusetts): The MIT Press, 2009.

[8] Hess K., Newman A. Practical virtualization solutions: virtualization from the trenches / K. Hess, A. Newman, Upper Saddle River, N.J.: Prentice Hall, Pearson Education, 2010.

[9] Intel, Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2: Instruction Set Reference & Volume 3: System Programming Guide, A-Z, Intel Corporation, 2011.

[10] Menken I. A complete guide on virtualization / I. Menken, Australia: Emereo Pty Ltd., 2008.

[11] Popek G.J., Goldberg R.P. Formal requirements for virtualizable third generation architectures // Communications of the ACM. 1974. № 7 (17). С. 412–421.

[12] ARM Architecture Group, ARM® Architecture Reference Manual: ARM® v7-A and ARM® v7-R edition – errata markup, ARM Limited,
ARM 0406C.c errata 2014 (ID051414) edition, 2014.

[13] Gartner Report.
URL: http://www.psfk.com/2012/03/cloud-replace-pc-headlines.htm ,
(01.06. 2017).

[14] Rik Myslewski, Rod Watt ARM tests: Intel flops on Android compatibility
URL:https://www.theregister.co.uk/2014/05/02/arm_test_results_attack_intel/
(01.06.2017).

[15] Niels Penneman, Danielius Kudinskas, Alasdair Rawsthorne, Bjorn De Sutter, Koen De Bosschere,
Formal virtualization requirements for the ARM architecture, 2000
MSC: 68M01, 68N25, 68N30

[16] H. Dong, Q. Hao, Extension to the model of a virtualizable computer and analysis on the efficiency of a virtual machine, in: Second International Conference on

Computer Modeling and Simulation, volume 2 of ICCMS 2010, IEEEComputerSociety, Los Alamitos, California, USA, 2010, pp. 503–507.

[17] Guthaus M. R. et al. MiBench: A free, commercially representative embedded benchmark suite //Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on. – IEEE, 2001. – С. 3-14.

[18] Spink, Tom, Harry Wagstaff, Björn Franke, and Nigel Topham. "Efficient code generation in a region-based dynamic binary translator." In *ACM SIGPLAN Notices*, vol. 49, no. 5, pp. 3-12. ACM, 2014.

[19] D'antras A. et al. Optimizing indirect branches in dynamic binary translators //ACM Transactions on Architecture and Code Optimization (TACO). – 2016. – Т. 13. – №. 1. – С. 7.