Основы программного моделирования ЭВМ

Учебное пособие

Версия 2.4
29 сентября 2013 г.

Оглавление

1 Улучшенные техники моделирования процессора 4
1.1 Двоичная трансляция 4
1.1.1 Преобразование гостевого кода 5
1.1.2 Пример преобразования одной инструкции . 6
1.1.3 Особенности реализации ДТ 8
1.1.4 Статическая и динамическая двоичная трансляция 12
1.2 Проблема самомодифицирующегося кода 17
1.3 Оптимизирующая трансляция 20
1.4 Вынесение фазы трансляции в отдельный поток 22
1.5 Прямое исполнение 23
1.6 Виртуализационные расширения 25
1.7 Гиперсимуляция 26
1.8 Динамическое переключение режимов симуляции . . 28
1.9 Пример практической двоичной трансляции 30
1.9.1 Исходный блок инструкций 31
1.9.2 Результат трансляции 31
1.10 Вопросы к главе 1 35

```
\include{contrib}

\include{overview}

\include{chapter01}

\include{chapter03}
```

1. Улучшенные техники моделирования процессора

> Лично я вижу в этом перст судьбы — шли по лесу и встретили программиста.
>
> *Аркадий и Борис Стругацкие. Понедельник начинается в субботу*

Рассмотрим принципы и алгоритмы, лежащие в основе таких методов симуляции, как двоичная трансляция и прямое исполнение, в том числе с аппаратной поддержкой. В конце главы описывается, как можно сочетать лучшие стороны всех рассмотреных подходов в составе одного симулятора.

1.1. Двоичная трансляция

Как было показано в предыдущей главе, симулирование исполнения процессора через интерпретацию обладает наряду с рядом положительных черт, таких как простота разработки и модификации, существенным недостатком — очень низкой скоростью работы получаемой модели, зачастую недостаточной для практического применения. Так, загрузка операционной системы на интерпретирующем симуляторе может занять дни.

Как и в случае с исполнением программ, написанных на языках высокого уровня, имеется следующее решение: вместо того, чтобы на каждом шаге анализировать текст, мы единожды компилируем его в машинный код и затем запускаем полностью подготовленную программу. При этом нет необходимости в перекомпиляции перед каждым запуском.

Если взять набор инструкций целевой машины за входной язык, а инструкции хозяйской машины — за выходной, то можно попытаться «скомпилировать» блоки целевого кода один раз и затем многократно переиспользовать результаты этой работы. При этом исчезает

необходимость обращаться к интерпретации инструкций на каждом шаге исполнения.

Подобный процесс получил собственное название *двоичная трансляция* (*ДТ*, также *бинарная трансляция*, *БТ*, *англ.* binary translation, BT) [1]. Несмотря на концептуальную схожесть с компиляцией языков высокого уровня, двоичная трансляция имеет существенные особенности, во многом связанные с тем фактом, что исходный для неё язык — машинный код целевой архитектуры — в отличие от языков высокого уровня содержит гораздо меньше информации об алгоритме программы и при этом может быть нагружен различными индивидуальными ограничениями гостевой ЭВМ, затрудняющими эффективную трансляцию и повышающими трудоёмкость написания транслятора.

1.1.1. Преобразование гостевого кода

Общий принцип ДТ состоит в том, что на некотором этапе работы транслятора для блока инструкций, взятых из гостевого приложения и принадлежащих гостевому ISA, в процессе трансляции создаётся новый блок, использующий хозяйские инструкции. Результаты исполнения гостевого кода на гостевой системе и транслированного на хозяйской должны совпадать, т.е. быть семантически эквивалентны. Одновременно могут существовать несколько блоков трансляции, соответствующих разным секциям исходного кода. Каждый их них имеет минимум одну точку входа — адрес, с которого содержащийся в нём код должен начинать исполняться, — и несколько (по крайней мере одну) точек выхода, соответсвующих различным ситуациям, при которых симуляция его покидает.

Отдельные блоки трансляции могут быть связаны вместе с помощью т.н. «клея» (*англ.* glue code), т.е. кода, не соответствующего никакому гостевому, но необходимого для передачи управления между блоками. На рис. 1.1 показано, как связаны части исходного кода гостевой программы и результат трансляции, состоящий из хозяйских инструкций.

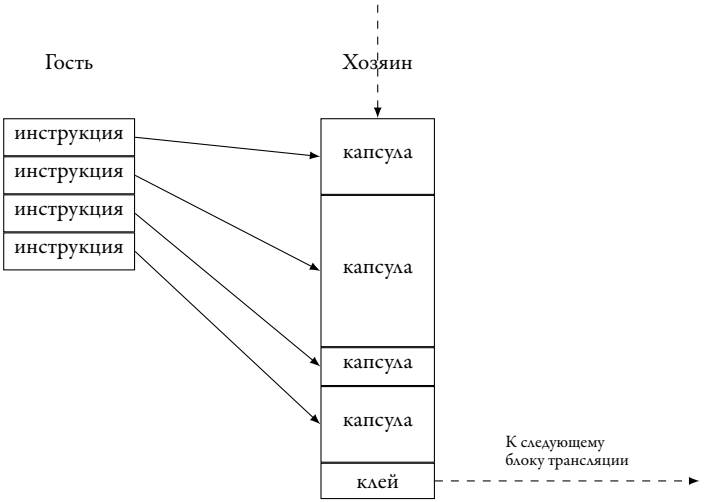

Рис. 1.1. Исходный код базового блока приложения и его связь с результатом двоичной трансляции. Штриховыми линиями показаны точки входа и выхода

1.1.2. Пример преобразования одной инструкции

На рис. 1.2 приведён пример соответствия гостевой 64-битной инструкции процессора архитектуры Intel®EM64T и блока хозяйского кода, называемого *капсулой* или сервисной процедурой (*англ.* service routine), хозяйского процессора, поддерживающего только 32-битные инструкции Intel®IA-32.

Доступ к гостевым регистрам. В рассматриваемом примере используется массив в памяти, различные ячейки которого хранят гостевые регистры. Хозяйский регистр `EBP` указывает на начало этого массива. По некоторому смещению от его начала, обозначенному `RAX_OFF`, хранится значение гостевого регистра `RAX` (строки (6) и (7)), `RBX_OFF` — смещение для регистра `RBX` и .т.д. Для того, чтобы выполнить операцию сложения, содержимое памяти загружается в пару 32-битных регистров EDX, EBX (строки (4) и (5)).

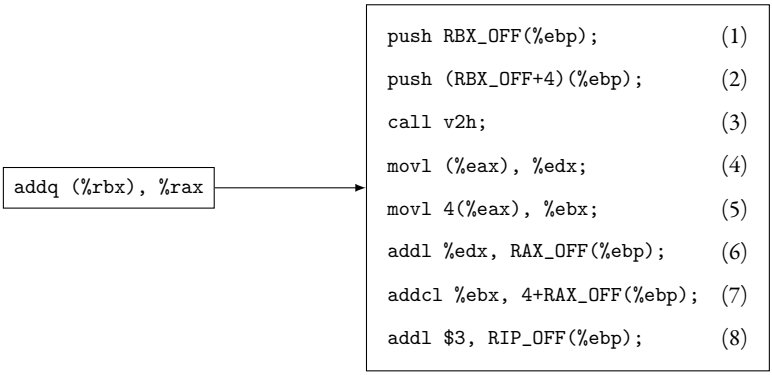

Рис. 1.2. Пример соответствия гостевой инструкции и хозяйской капсулы, эмулирующей её семантику и написанной на языке ассемблера. В этом примере хозяйский регистр EBP хранит указатель на структуру гостевого состояния, макросы вида RxX_OFF — смещение внутри гостевого состояния для регистра RxX, а v2h — функция преобразования виртуальных гостевых адресов в хозяйские

Выполнение операции. Поскольку в наборе инструкций IA-32 нет инструкций для операции с 64-битными числами, сложение проводится в два этапа. Сначала складываются младшие 32 бита операндов с помощью инструкции ADDL строка (6). Затем — старшие 32 бита с учётом возможного флага переноса разряда от предыдущего сложения с помощью ADDCL, строка (7).

Чтение гостевой памяти. Ситуация с обращениями к гостевой памяти несколько сложнее. Для её моделирования уже недостаточно просто завести массив в памяти. В общем случае связь гостевых данных и их положения в хозяйском пространстве памяти нелинейна и сложна. В нашем примере это отражено тем, что, перед тем как загрузить первый операнд, вызывается функция v2h, строка (3), единственный аргумент которой сохранён в стеке, строки (1) и (2). Подробнее о том, что эта функция выполняет, рассказывается в главе **??**.

Последняя хозяйская инструкция продвигает симулируемый регистр RIP на длину только что обработанной (3 байта) так, чтобы он

указывал на начало следующей инструкции (строка (8)).

Размер капсулы

Для «идеального» ДТ для некоторой пары архитектур желательно выдерживать соответствие «одна хозяйская инструкция эмулирует одну гостевую» для каждой капсулы. Из-за неполного соответствия окружений гостя и симулятора это почти никогда не выполняется, возможны следующие ситуации.

1. На одну гостевую приходится несколько хозяйских инструкций, в сумме компенсирующих различия между архитектурами.
2. На одну гостевую приходится ноль хозяйских инструкций. Такая ситуация возникает, если исходная команда не изменяет архитектурного состояния и может быть опущена в функциональной модели. Примеры: операции предвыборки в кэш, подсказки для предсказателя переходов.
3. Соединяющий блоки трансляции клей не соответствует ни одной гостевой инструкции и необходим только для работы симулятора.

1.1.3. Особенности реализации ДТ

Разумно ожидать, что чем больше похожи целевая и хозяйская архитектура, тем проще создавать ДТ и тем быстрее он должен работать. Для особого случая, когда эти архитектуры совпадают, может оказаться, что никакого преобразования производить и не требуется — целевой код уже «готов» для исполнения (см. секцию 1.5 о преградах на пути к такому бесхитростному подходу). Верно и обратное — чем сильнее различаются архитектуры гостя и хозяина, тем больше усилий приходится вкладывать в реализацию ДТ и симулятора в целом.

Семантика инструкций

Всё множество команд современных процессоров можно разделить на несколько классов согласно выполняемой ими функции. Рассмот-

рим особенности, характерные для симуляции инструкций каждого из них.

Арифметические целочисленные. Практически все существующие ISA имеют команды для арифметических, логических и сдвиговых операций над целыми числами, и их эффективное моделирование в составе ДТ, как правило, вызывает минимальные проблемы.

Инструкции с числами с плавающей запятой. Поддержка разными процессорами существенно различается, несмотря на наличие стандарта IEEE 754 [6], призванного внести унификацию. Некоторые архитектуры могут оперировать числами только одинарной (32 бита) или двойной (64 бита) точности. Другие используют нестандартные форматы, например, сопроцессор x87 IA-32 использует внутреннее представление чисел шириной 80 бит, а в IA-64 машинный формат имеет 82 бита. Машинная поддержка половинной (16 бит) и четырёхкратной (128 бит) точности, а также форматов с основанием десять присутствует в ограниченном числе систем. Кроме представления чисел, сами арифметические операции могут быть реализованы по-разному. Они различаются доступными режимами округления результатов, способами индикации ошибочных ситуаций, поведением для т.н. *денормализованных* (*англ.* denormalized) чисел и т.д. Интересующийся читатель найдёт подробное описание в [4]. Библиотека SoftFloat [5] реализует стандарт IEEE 754 с помощью только целочисленной арифметики, тем самым предоставляя переносимую реализацию.

Векторные инструкции. Используются для параллельного выполнения операции над векторами значений, хранящихся в специальных регистрах шириной до 512 бит. Примеры: Intel® SSE, AVX, AVX2, IBM AltiVec [11]. При симуляции в случае, если хозяин не имеет аналогичной инструкции, она может быть представлена с помощью последовательного выполнения операции над всеми элементами вектора. Таким образом, векторные операции сводятся к своим последовательным вариантам.

Контроль управления. В этот класс включаются инструкции, изме-

няющие значение указателя текущей команды PC, т.е. условные и безусловные переходы, вызовы процедур и возвращения из них, программного прерывания и т.д. В разных архитектурах они отличаются очень сильно. Поэтому чаще всего их капсулы получаются достаточно длинными. Общая задача симулятора при их обработке — вычисление точки входа в новый блок трансляции, соответствующий гостевому адресу перехода. При этом приходится учитывать возможность ситуации, в которой она отсутствует или некорректна. Более подробно вопрос обработки переходов в гостевом коде рассматривается в секции **??**

Привилегированные инструкции. В большинстве архитектур некоторая часть команд может исполняться, только если процессор находится в специальном режиме, иначе они вызывают исключение. В этом режиме работает операционная система, имеющая неограниченный доступ ко всем ресурсам системы. Привилегированные инструкции специфичны для каждой системы и обычно семантически нагружены, поэтому их симуляция требует длинных капсул.

Ситуация не улучшается даже при полном совпадение архитектур гостя и хоязина. Так как исполнение привилегированных команд в непривилегированном режиме, в котором обычно работает сама программа-симулятор, невозможно, их приходится заменять последовательностью разрешённых инструкций. Более подробно этот вопрос разбирается в главе **??**.

Прочие. Существует достаточно много инструкций различных ISA, не подпадающих под данную выше классификацию или имеющих специфику, требующую особого внимания при симуляции. Это могут быть строковые, предикатные, длинные инструкций, слоты задержки у переходов и т.п.

Сходства и различия в архитектурных состояниях

Хранение состояния целевой системы в выделенном буфере памяти обладает недостатком — необходимостью часто обращаться к медленному ОЗУ и испытывать большие задержки при промахах кэша. Поэтому создатели систем ДТ стараются разместить максималь-

но возможное число целевых регистров на хозяйских, чтобы при обращении к ним требовалось минимальное время. Это легко осуществить, если в архитектуре хозяина предусмотрено большее число регистров, чем необходимо гостю. Например, это верно для комбинации гостевой системы IA-32 с 8 регистрами общего назначения и хозяина архитектуры MIPS с 31 регистром. При этом максимальная ширина доступных регистров также может различаться. Например, для модели 64-битной архитектуры IA-64 не получится уместить гостевой регистр целиком в хозяйском, если хозяйская система — 32-битная, например, ARMv7.

Если эти условия на регистровый файл не выполняются, то приходится прибегать к различным ухищрениям. Так, только часть регистров может быть отдана под нужды симуляции, а один гостевой регистр приходится «разбивать» на несколько частей, по отдельности умещающихся в хозяйских. См. также главу **??**, в которой подробнее разбираются вопросы моделирования состояния.

Особенности обработки доступов к памяти и устройствам

Несмотря на то, что операции чтения и записи памяти присутствуют почти во всех архитектурах процессоров, за историю развития вычислительной техники было придумано неисчислимое количество способов адресации и обращения к ней. Не пытаясь объять необъятное, приведём лишь несколько примеров.

- В архитектуре IA-32 адрес операнда в памяти может определяться несколькими регистрами и константами, закодированными в инструкции. В самом общем случае в ней определяется сегмент, база, индекс и масштабный коэффициент, а также одна константа, определяющая смещение и поле, изменяющее ширину. Для контраста: в системах с процессорами MIPS в адресация используется один регистр и одна константа.
- В ряде случаев ячейка памяти может адресоваться нулём операндов, т.е. неявно, например, располагаться на вершине стека.
- Поддерживаемые размеры доступов в память могут быть различными. Например, хозяин за одну операцию может прочи-

тать максимум 32 бита, тогда как в гостевой архитектуре требуемый размер считываемых данных равен 64 битам. Это усложняет моделирование атомарных операций, т.к. приходится разбивать гостевой доступ на несколько транзакций, нарушая исходные предположения о неделимости последнего. Обратная ситуация, когда, например, требуется прочитать 1 байт гостевой памяти, но хозяин может адресовать только 4 байта, тоже может привести к ошибкам вы симуляции.

– Отдельно следует отметить различия в требованиях разных систем к выравниванию (*англ.* alignment) доступов в память[1]. Некоторые архитектуры запрещают **невыровненные доступы** — при попытке прочитать или записать данные по такому адресу возникает исключение, тогда как другие процессоры это позволяют, зачастую облагая такой доступ повышенным временным «пенальти».

1.1.4. Статическая и динамическая двоичная трансляция

Попытаемся ответить на два следующих вопроса.

1. Какой должна быть единица ДТ? Другими словами, чем определяется количество и расположение целевых инструкций, обрабатываемых за один проход транслятора?
2. Как должны быть связаны во времени фазы трансляции и симуляции? Должна ли одна из них предшествовать второй, или они должны чередоваться?

Для обычных языков высокого уровня ответ на первый вопрос почти очевиден — исходный файл с текстом программы (или модуля) компилируется в приложение (объектный файл), самодостаточное в плане дальнейшего исполнения или использования. Более мелкие единицы компиляции, такие как процедуры, также имеет смысл транслировать целиком, так как при их использовании понадобится весь их код.

[1]Блок памяти длиной w является выровненным по адресу A, если $A = 0 \mod w$, т.е. A нацело делится на w. При этом чаще всего рассматривается выравнивание по степеням двойки.

В случае ДТ возникают сложности из-за того, что входной текст таких систем — «монолитный» машинный код, не имеющий меток начала отдельных субъединиц, зачастую с перемешанными секциями кода и данными, неопределёнными адресами переходов и т.п.

Статическая ДТ. Хотя аналогичная компиляции техника трансляции гостевого приложения целиком в образ хозяйского кода (статическая ДТ) иногда применялась [10], она не получила широкого распространения по ряду причин.

Будучи применимым для трансляции отдельных пользовательских приложений, статическая ДТ становится невозможной в случае полноплатформенной симуляции, при которой пришлось бы транслировать всю память гостевой ЭВМ. Во-первых, объём входного текста может быть огромен, и время трансляции, и размер результирующего файла окажутся непозволительно большими. Во-вторых, содержимое памяти, в том числе секций с кодом, изменяется в ходе работы (см. также дальше секцию «Проблема самомодифицирующегося кода»), что делает статическую ДТ бессмысленной — результирующий код в силу своей неизменности не будет отражать правильное состояние изменяемой памяти.

С другой стороны, будучи однажды полученным и сохранённым в файле на диске, результат статического преобразования приложения может запускаться неограниченное число раз, что компенсирует время, потраченное на его получение. Поэтому на этапе ДТ могут быть применены разнообразные оптимизации, нацеленные на создание максимально эффективного кода (см. секцию 1.3).

Динамическая ДТ. Для задач симуляции более адекватным является иной подход, в котором моделирование гостевой системы (то есть исполнение оттранслированного кода) перемежается с запусками механизма двоичной трансляции для новых блоков кода, которые будут вскоре исполненны, а также с обновлениями трансляций для блоков, изменивших своё содержимое. При этом в памяти симулятора хранятся ранее оттранслированные секции для их переиспользования в случае, если управление вновь перейдёт на них (рис. 1.3).

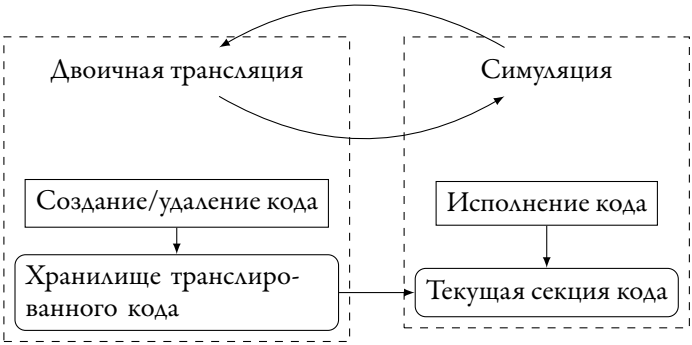

Рис. 1.3. Динамическая ДТ. Фаза симуляции, использующая сгенерированный код, периодически сменяется фазой трансляции, хранящей уже существующие и создающей новые секции хозяйского кода из гостевого

Отметим, что, в отличие от статической, при динамической ДТ время, потраченное на фазу преобразования, фактически отнимается у фазы симуляции, т.е. негативно сказывается на производительности модели. Поэтому спектр возможных оптимизаций более ограничен, использованы могут быть только достаточно быстрые из них. См. также секцию 1.4.

Обнаружение кода. Следующие обстоятельства необходимо учитывать в процессе трансляции блоков инструкций.

- В оперативной памяти данные программ (переменные, массивы) и код (инструкции), их обрабатывающий, хранятся вместе. В общем случае никаких границ между ними не обозначено. Трансляция секций данных бесполезна: управление никогда не будет передано на них, — и даже вредна: затрачиваемое время уходит впустую. Необходим критерий, определяющий целесообразность выполнения ДТ для некоторого региона памяти.
- В архитектурах, допускающих переменную длину инструкций, очень важен адрес, с которого начинается их декодирование, интерпретация или трансляция. Сдвиг даже на один байт приводит к изменению смысла до неузнаваемости (рис. 1.4). Кроме

того, результат декодирования может зависить от режима процессора, поэтому, если в ходе симуляции он изменился (например, процессор перешёл из 32-битного в 64-битный режим), то предыдущие блоки трансляции, скорее всего, перестали соответствовать исходному коду.

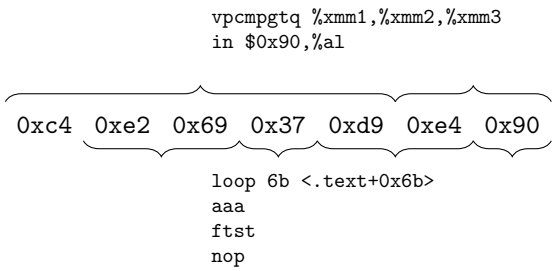

Рис. 1.4. Обнаружение кода. Смысл содержимого памяти меняется при изменении стартового адреса. Пример двух интерпретаций для фрагмента кода архитектуры IA-32

Указанные проблемы определяют задачу *обнаружения кода* (*англ.* code discovery). Точное её решение зависит от особеностей архитектур гостя и хозяина. Отметим лишь два ключевых момента.

– Некоторый регион в памяти разумно подвергать ДТ, если вероятность того, что в некоторый он будет исполнен, хотя бы ненулевая. Это верно в случае, когда он достижим из других, уже оттранслированных частей программ, т.е. известно, что некоторые инструкции передачи управления указывают на него. Если код исполнялся раньше, также велика вероятность того, что он исполнится в будущем.
– Очень важно кроме собственно содержимого блока трансляции хранить и все допустимые точки входа в него, т.е. адреса, попадающие на границы инструкций, а также ассоциировать режим процессора, для которого блок был создан.

Отметим, что задача обнаружения кода при ДТ во многом связана с поддержкой самомодифицирующегося кода, описываемой в сек-

ции 1.2.

Единицы трансляции. Память хозяйской системы ограничена, что возвращает нас к первому вопросу — как выделять и организовывать блоки трансляций, чтобы получить приемлемую скорость симуляции, при этом не исчерпав ёмкость хозяйского ОЗУ? Кроме того, необходимо определиться, какие блоки хранить, а какие выбрасывать, какую длину в байтах они должны иметь. Рассмотрим два возможных решения этих задач, которые основываются на принципе локальности исполнения и ограниченности рабочего набора [12].

1. Трасса исполнения — это запись истории того, в каком порядке инструкции когда-то были исполнены. Как правило, трасса имеет ровно одну точку входа, соответствующую первой её инструкции. Из общих свойств алгоритмов следует высокая вероятность того, что впоследствии эти инструкции будут исполнены снова в том же порядке. При этом если они формируют базовый блок (т.е. среди них не встречается команд условного или непрямого перехода), то порядок их исполнения будет в точности такой же, как и в первый раз. Следует отметить, что первоначальное создание трасс, когда никакой истории исполнения ещё нет, приходится организовывать с помощью альтернативного механизма симуляции, например, интерпретацией (рис. 1.5). Прерывать создание трассы нужно по ряду условий в гостевом коде, после которых направление исполнения неизвестно или существенно отличается, например, на исключениях, прерываниях, командах смены режима процессора и т.п.
2. Инструкции, располагающиеся в памяти по соседним адресам, скорее всего, относятся к связанным частям алгоритма программы, будут выполняться вместе и поэтому могут быть оттранслированы в один блок (рис. 1.6). В этом случае единицей трансляции является гостевая страница фиксированного размера. В отличие от трасс, страница трансляции может иметь множество точек входа — каждый адрес, соответствующий началу гостевой инструкции на ней, может быть использован таким образом. Однако необходимо следить, чтобы управление не переда-

валось «в середину» инструкции — в таком случае трансляция некорректна.

Кроме того, трансляция кода на текущей странице может быть прервана по достижении блока хозяйского кода определённого объёма. Как и в случае с трассами, разумно прерывать процесс ДТ при обнаружении инструкции условного или непрямого перехода.

Хорошее описание приёмов ДТ, в ряде источников называемой *JIT-компиляцией* (*англ.* just in time), дано в [14].

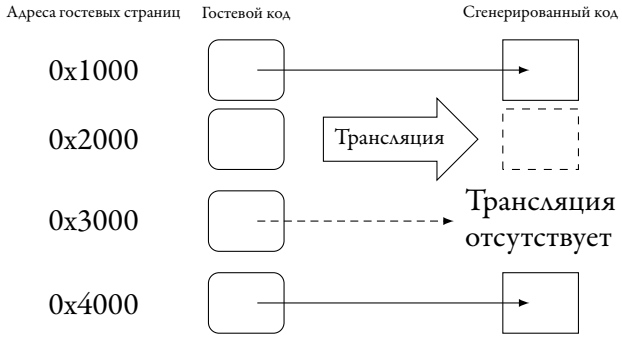

Рис. 1.5. Двоичная трансляция целых страниц. Для ранее исполненных блоков переиспользуются оттранслированные секции хозяйского кода. Процесс симуляции прерывается для трансляции новой страницы

1.2. Проблема самомодифицирующегося кода

Большая доля современных архитектур процессоров для ЭВМ построена согласно принципам фон Неймана. Один из них состоит в том, что исполняемый код и обрабатываемые им данные располагаются в одной физической памяти. Следствие этого — возможность создания программ, которые в процессе работы изменяют код других программ и, в частности, свой собственный. Затем этот новый код может быть исполнен. Мы будем обобщённо обозначать такое явление,

как **самомодифицирующийся код** (*англ.* self-modifying code, SMC). Для программ с SMC не все инструкции приложения известны до момента их генерации во время работы уже запущенного приложения.

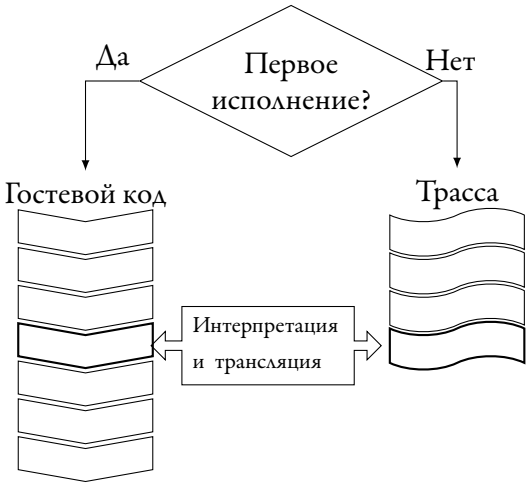

Рис. 1.6. ДТ с трассами исполнения. Первое исполнение каждой гостевой инструкции производится с помощью интерпретатора, при этом также осуществляется её трансляция и сохранение результатата в трассе

Это обстоятельство фактически делает системы статической ДТ, не имеющие слой симуляции времени выполнения, функционально несостоятельными — они не могут корректно транслировать такой код.

Замечание. *Симулятор, задействующий динамическую двоичную трансляцию, сам по себе является программой с самомодифицирующимся кодом, так как на фазе симуляции управление передаётся на код, отсутствующий в исходном файле приложения, — он был создан «на лету» на фазе трансляции.*

При исполнении самомодифицирующейся программы гостевой код изменяется, и есть вероятность, что уже существующие блоки транслированного кода, соответствовавшие первоначальному состоянию памяти, перестанут подходить новому содержимому, и при пе-

редаче на них исполнения результат вычислений будет некорректен. Для предупреждения этого необходимо отслеживать все записи в память и сбрасывать или ретранслировать затронутые при этом блоки.

Поскольку процесс ДТ одного блока занимает существенное время, скорость работы симулятора для участков программ с самомодифицирующимся кодом может резко падать — блоки живут недолго, часто отбрасываются как устаревшие, исполнение часто прерывается на ретрансляцию. В таких случаях простой интерпретатор может показывать более высокую скорость симуляции.

Следует иметь в виду, что детали поведения процессора при SMC могут отличаться на разных архитектурах. Обусловлено это тем, что в реальности инструкции также берутся не непосредственно из памяти, а из более быстрых буферов, куда они были помещены специальными механизмами предварительной загрузки, и состояние памяти может не соответствовать их содержимому. Так, для систем с раздельными кэшами инструкций и данных (ARM, MIPS) результат модификации кода проявится только после выполнения специальных инструкций сброса кэшей. В архитектуре Intel®IA-32 гарантируется, что результат SMC будет виден для исполняющего устройства немедленно. Исключением является изменение инструкции непосредственно под указателем инструкций — оно не будет видно программе, пока текущая инструкция не закончится.

В любом случае обеспечение работы SMC требует сброса части состояния и повторного считывания его из памяти, что вносит некоторую задержку в исполнение, и при неправильной организации кода его производительность может сильно пострадать.

Ситуация усложняется, когда в моделируемой системе есть несколько агентов, способных модифицировать память, например, в многопроцессорных системах или в платформах, где устройства могут писать в память напрямую (*англ.* direct memory access, DMA). В таких случаях модель должна отслеживать все такие доступы и отбрасывать устаревшие блоки.

1.3. Оптимизирующая трансляция

После того, как некоторый блок трансляции создан, может оказаться, что возможно преобразовать его так, чтобы он исполнялся быстрее, при этом сохранив его семантику; другими словами, провести *оптимизирующую* трансляцию. Этот процесс по смыслу аналогичен фазе оптимизаций обычного компилятора, позволяющей уменьшить время выполнения программ. Подчеркнём критически важное условие неизменности алгоритма фрагмента до и после преобразования. Если есть ненулевая вероятность того, что в каких-то случаях результат исполнения после применения определённой оптимизации будет отличаться от исходного, то её применять нельзя.

На рис. 1.7 приведён пример часто используемой оптимизации некоторого блока трансляции с одной точкой входа и выхода [3] для некоторой архитектуры. Гостевой код состоит из пяти арифметических инструкций `instr1` ...`instr5` и инструкции `branch`. При трансляции капсулы отдельных инструкций[1] берутся последовательно друг за другом, формируя блок. При этом последняя машинная инструкция `inc` каждой из них предназначена для продвижения симулируемого регистра `PC`.

Оптимизация в данном случае основана на том факте, что значение этого регистра `PC` на протяжении почти всего блока никто не читает, и неважно, изменилось оно или нет. Поэтому можно отложить все изменения до того момента, когда понадобится новое значение, т.е. до капсулы инструкции перехода `<branch>`. Следующий напрашивающийся шаг — использовать очевидное равенство $x+1+1+1+1+1 = x + 5$ и заменить пять инструкций сложения одной.

Как видно даже из столь простого примера, после оптимизациии границы между исходными капсулами размываются, т.к. составляющие инструкции могут быть переставлены местами, заменены другими или вообще убраны.

Следующие типы оптимизаций, используемых в обычных компиляторах [15], применимы и при двоичной трансляции.

[1] Несущественное для данного объяснения содержимое капсул объединено и обозначено угловыми скобками.

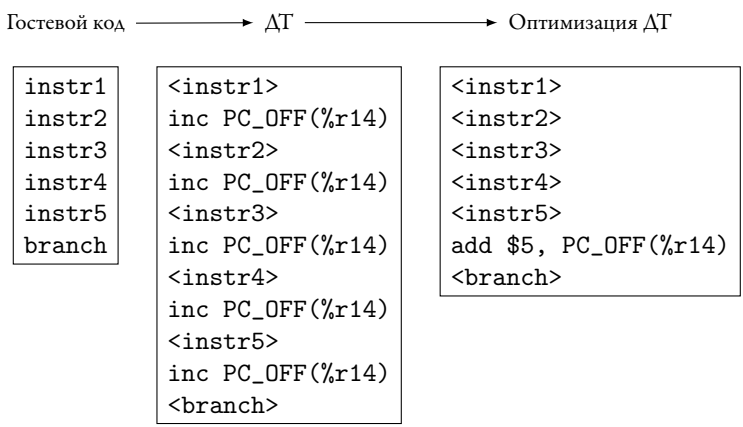

Рис. 1.7. Пример простой оптимизации кода блока трансляции. Инструкции `inc` продвижения регистра `PC` после каждой капсулы заменены одним сложением `add` в конце блока

Удаление мёртвого кода (*англ.* dead code elimination) — нахождение команд, не влияющих на исполнение последующего кода. Вычисляемые ими значения не используются, поэтому и сами инструкции без вреда могут быть удалены.

Удаление общих подвыражений (*англ.* common subexpression elimination) — для вычислений, выполняемых более одного раза на рассматриваемом участке, второе и последующие их вхождения могут быть убраны и заменены уже вычисленным значением.

Свёртка констант (*англ.* constant folding) и **дублирование констант** (*англ.* constant propagation) — оптимизации для замены константных выражений и переменных на их значения, вычисленные при трансляции.

Анализ соседних инструкций (*англ.* peephole optimization) — класс оптимизаций, основанных на знании особенностей хозяйской архитектуры и стоимости выполнения инструкций. Например, две подряд идущие команды могут быть заменены на одну более быструю.

Как правило, блоки трансляции не включают в себя циклы. По этой причине такие оптимизации, как раскрытие, слияние, инверсия циклов (*англ.* loop unrolling, loop fusion, loop inversion) и т.п., связанные с анализом потока управления, ограниченно доступны для задач симуляции. Примером такой оптимизации может считаться гиперсимуляция, описываемая в секции 1.7.

1.4. Вынесение фазы трансляции в отдельный поток

В описанном выше алгоритме динамической двоичной трансляции её фазы: ДТ и собственно симуляция — чередуются, взаимно исключая исполнение друг друга. Однако осмысленным с точки зрения повышения производительности является вынесение процесса ДТ в отдельный хозяйский поток, исполняющийся параллельно с основным потоком, используемым для симуляции, и поставляющий для его нужд блоки трансляций [9].

Для сравнения на рис. 1.8 и 1.9 приведено соотношение этапов исполнения и ожидания для этих двух активностей в случае последовательной ДТ и ДТ, вынесенный в отдельный поток.

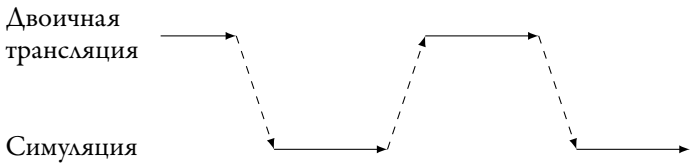

Рис. 1.8. ДТ и симуляция выполняются последовательно

Очевидно, что выигрыш в производительности у такого решения будет наблюдаться, только если поток трансляции будет успевать генерировать новые блоки раньше, чем они понадобятся потоку симуляции. В противном случае последний всё равно будет вынужден простаивать. При этом структура симулятора значительно усложняется, так как необходимо производить координацию и синхрониза-

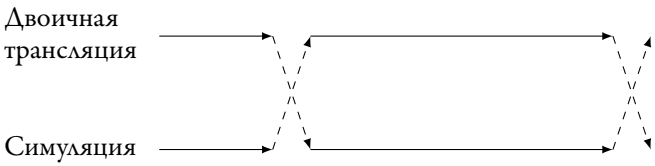

Рис. 1.9. ДТ вынесена в отдельный поток

цию двух потоков, не допуская использования блоков до того, как они будут полностью готовы, и следя за тем, чтобы поток ДТ работал с самой актуальной версией гостевого кода.

1.5. Прямое исполнение

Интересным и важным на практике случаем ДТ является ситуация, когда архитектуры гостя и хозяина совпадают (или почти совпадают). При этом возникает возможность значительно упростить трансляцию — в некоторых случаях она сводится к копированию гостевого кода как хозяйского или даже исполнению его «на месте», без дублирования. Подобные режимы симулирования имеют общее название *прямое исполнение* (*англ.* direct execution, DEX).

Несмотря на кажущуюся простоту реализации, необходимо отметить особенности и требования, усложняющие схемы DEX. Изолирование выполнения кода гостевых приложений и кода самого симулятора является главным. Гостевое приложение не должно иметь возможность определить факт того, что оно исполняется внутри модели, и тем более влиять на её работу через модификацию памяти.

– *Доступы к памяти и периферии.* Адресное пространство гостя занимает часть памяти симулятора и не обязательно размещено по тем же самым абсолютным адресам, где гостевое приложение ожидает его увидеть. ДТ поэтому должен перехватывать все доступы в память и «переписывать» их так, чтобы они всегда указывали на корректные данные и не могли повредить память самой модели. Аналогично, гостевое приложение не долж-

но иметь прямого доступа к периферийным устройствам хозяина.

– *Архитектурное состояние.* Регистровый файл гостя и хозяина совпадает, поэтому невозможно полностью разместить гостевые регистры на одноимённых хозяйских — некоторое их количество зарезервировано для нужд самого симулятора. Опять же, гость не должен получать доступ к состоянию регистров, задействованных моделью, — необходимо перехватывать обращения к ним и подставлять правильные значения. Как правило, регистровый файл используется в двух переключаемых режимах: во время исполнения транслированного кода большая его часть заполнена состоянием гостя, а при выходе в симулятор оно сбрасывается в память, и регистры используются для нужд симулятора. По возвращении в транслированный код состояние восстанавливается.
– *Привилегированные инструкции [8].* Будучи пользовательским приложением, симулятор работает в непривилегированном режиме, тогда как гостевое приложение может исполнять инструкции системных режимов. Без явного контроля со стороны ДТ это, скорее всего, приведёт к аварийному завершению процессов. Поэтому симулятор должен заранее находить в потоке команд гостя «опасные» инструкции и заменять собственными обработчиками. Альтернативно, иногда имеется аппаратно поддерживаемая возможность перехватить исключение от попытки исполнения привилегированной инструкции, промоделировать её в обработчике исключения и вернуться к исполнению. Интересные особенности при этом существуют в архитектуре IA-32 — семантика некоторых инструкций меняется в зависимости от того, в каком режиме процессора они исполняются. Пример — `POPF` (*англ.* pop flag register), которая модифицирует флаг Interrupt Enable, будучи исполнена в привилегированном режиме; в пользовательском режиме она может изменить все флаги, кроме вышеуказанного.

1.6. Виртуализационные расширения архитектуры и их использование для симуляции

Поскольку сценарий симуляции с совпадающей архитектурой гостя и хозяина является практически важным, в ряде ЭВМ существует аппаратная поддержка типичных операций, встречаемых в симуляторах такого типа. Например, дорогая с точки зрения числа циклов операция перехвата и трансляции адресов гостя в реальные адреса хозяина может быть поддержана аппаратно с помощью дополнительного уровня косвенности в механизме обращения к страницам, позволяющего быстро переключать контекст памяти симулятора и симулируемого приложения.

ЭВМ IBM System/370 была спроектирована таким образом, что позволяла исполнять напрямую приложения в изолированных контейнерах. В случае, когда встречалась привилегированная инструкция, она обрабатывалась симулятором (в терминологии System/370 — Control Program, CP) прозрачно для приложения.

Архитектура IA-32 довольно долгое время не имела эффективных механизмов поддержки изолированного исполнения приложений. Оно было реализовано (расширение Intel®VTx добавлено в 2005 г.; в настоящее время существует несколько версий этого расширения) в виде дополнительных режимов процессора и нескольких команд, позволяющих переходить между ними и стандартными, не виртуализационными режимами.

В режим монитора виртуальных машин процессор попадает, когда работа одного из исполняемых гостей требует его вмешательства (рис. 1.10). При этом ему доступно всё архитектурное состояние гостя, которое можно инспектировать, модифицировать соответственно причине события внутри гостя. Затем процессор может вернуться обратно в «непривилегированный» режим исполняемой виртуальной машины; переключение состояния будет произведено автоматически.

Введение аппаратно поддерживаемого прямого исполнения позволило ускорить обработку таких дорогих с точки зрения потребляемо-

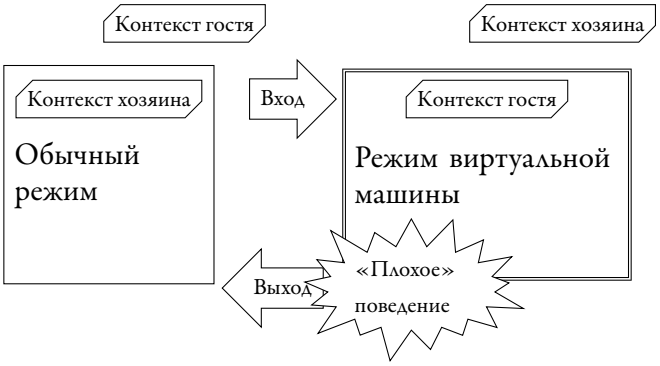

Рис. 1.10. Аппаратная поддержка переключения контекста между монитором виртуальных машин и гостевой системой. При входе в режим виртуальной машины с помощью системной инструкции контекст хозяина сохраняется в выделенной области памяти и заменяется контекстом гостя. При любой причине обратного перехода в режим монитора (например, при попытке выполнить зарезервированную операцию) этот контекст обменивается с гостевым

го на их симуляцию времени операций гостевых систем, как преобразование виртуальных адресов в физические преобразования TLB [2]. Такие операции могут производиться аппаратурой, а не программой хозяина, что также упрощает (и делает более надёжной) его реализацию.

1.7. Гиперсимуляция

Как было показано выше, выигрыш в скорости от использования технологий ДТ обусловлен переиспользованием однажды сгенерированного хозяйского кода для многих последующих циклов симуляции; этот выигрыш теряется при несоблюдении условий постоянства машинного кода, как было показано в секции про SMC.

Заметим, что, несмотря на то, что при благоприятных для ДТ условиях блок оттранслированного кода неизменен, при различных вхо-

дах в симуляцию с одного и того же указателя гостевых инструкций архитектурное состояние и содержимое памяти могут различаться.

Рассмотрим такие случаи, когда архитектурное состояние при всех входах в некоторый блок кода остаётся неизменным. Исходя из свойства детерминистичности вычислительных систем, можно утверждать, что по выходу из такого блока состояние системы каждый раз будет одно и то же. Очевидно, что для таких участков кода нет нужды каждый раз их симулировать — достаточно один раз запомнить результат вычисления и просто изменять состояние системы после входа в блок на конечное, таким образом полностью избегая вычислений и достигая «бесконечно высокой» скорости симуляции (рис. 1.11). Данный «режим» имеет название *гиперсимуляция* (*англ.* hypersimulation).

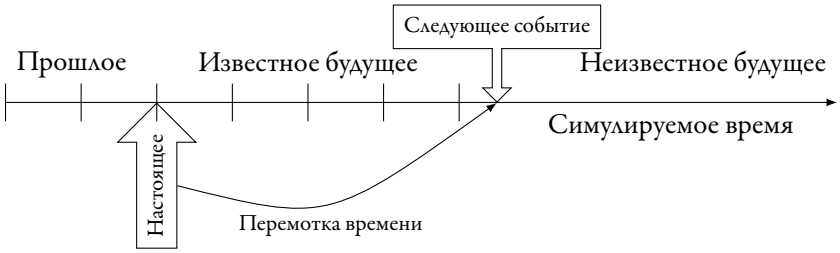

Рис. 1.11. Гиперсимуляция. При обнаружении возможности симулируемое время продвигается вперёд до следующего события, состояние процессора при этом неизменно

Упомянутые выше условия, позволяющие применить данную оптимизацию и налагаемые на код, очень жестки. На практике они выполняются только для очень небольших блоков кода, например, в реализациях примитива синхронизации «циклическая блокировка» (*англ.* spin lock). Типичная реализация для IA-32, написанная на ассемблере, выглядит следующим образом:

```
spin_lock:
movl $1, %eax
lock xchgl (locked), %eax
testl %eax, %eax
jnz spin_lock
```

В примере процессор непрерывно записывает в переменную `locked` до тех пор, пока она не станет равной нулю. Изменить её может другой процессор, разделяющий память с первым (о симуляции многопроцессорных систем см. главу **??**), например, следующим образом:

```
spin_unlock:
movl $0, %eax
xchgl (locked), %eax
```

При последовательной симуляции этих двух гостевых процессоров на одном хозяйском (т.е. на симуляцию каждого отведена фиксированная квота времени, в течение которой состояние неактивного процессора не изменяется) значение `locked` не меняется в процессе симуляции первого. Поэтому вместо того, чтобы тратить время на моделирование этого «бесконечного» цикла, следует просто считать, что за выделенную квоту времени состояние процессора не изменилось.

В гостевом приложении присутствуют различные участки кода, выполнение которых возможно в режиме гиперсимуляции. Автоматическое их выявление затруднено, однако чаще всего они характеризуются с помощью нескольких шаблонов, которые необходимо задавать вручную.

1.8. Динамическое переключение режимов симуляции на различных участках работы системы

Все продемонстрированные выше техники симуляции — интерпретация, двоичная трансляция, прямое исполнение, аппаратно ускоренная симуляция, гиперсимуляция — характеризуются условиями, в которых их применение оправдано, т.е. они дают выигрыш в скорости, и ситуациями, когда использование невыгодно. Поэтому на практике часто применяется комбинированное использование двух или более техник с переключением между ними на различных этапах симуляции, при этом выбирается наиболее быстрая из доступных.

Рассмотрим пример: моделирование загрузки операционной си-

стемы на архитектуре IA-32 с последующим запуском пользовательского приложения. На нём разберём, какие из изученных подходов оптимальны.

– В первые секунды работы, когда активна программа BIOS, процессор IA-32 находится в т.н. реальном режиме, исторически реализованным первым. При этом используется двоичная трансляция.
– Затем начинает загружаться операционная система. Режим процессора меняется на защищённый, и аппаратная поддержка прямого исполнения, недоступная для реального режима, может быть задействована. Используется прямое исполнение. На участках синхронизации с внешними устройствами может быть задействована гиперсимуляция.
– Запускается пользовательское приложение, активно использующее SMC. В таких условиях все «оптимизирующие» режимы теряют свои преимущества, и потому исполнение происходит с помощью интерпретации.

Для эффективного переключения между режимами необходимо иметь алгоритм, согласно которому выбирается режим работы. Однозначного решения тут нет, как правило, решение принимается в каждом случае и включает некоторые эвристики, выработанные для конкретного применения симулятора. Перечислим наиболее частые подходы к динамическому определению оптимального режима (рис. 1.12).

– Собирается статистика частоты нахождения в различных блоках гостевого кода. Если обнаруживается, что в каком-то блоке программа проводит много времени, то для него включается ДТ. Для остальных блоков продолжается использоваться интерпретация.
– Программой измеряется скорость собственной работы в оптимизированных режимах. Если обнаруживается, что она ниже некоторого порога, то происходит возвращение к интерпретации; при этом экономится время, ранее тратившееся на неэффективную ДТ.

– Анализируется статистика частот событий, мешающих оптимизированным режимам эффективно работать. Для ДТ это случаи необходимости ретрансляции блоков, для аппаратно поддерживаемого прямого исполнения это события возвращения в программу-монитор. Если такие события происходят чаще некоторого порога, то соответствующий режим отключается.

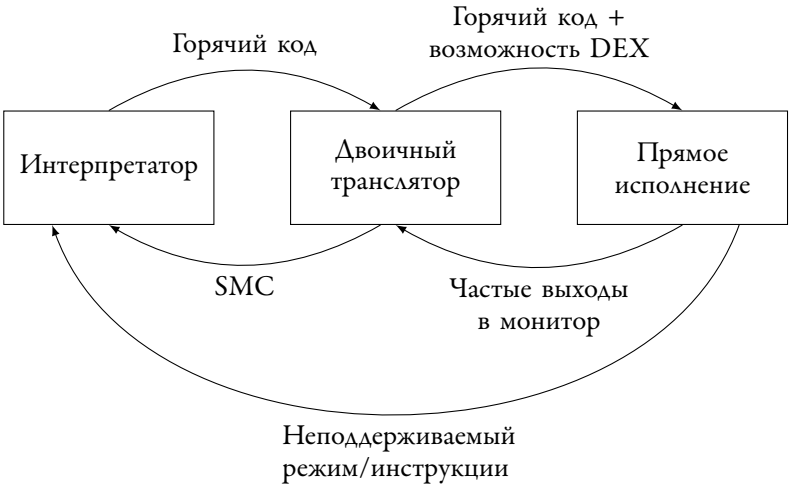

Рис. 1.12. Динамическое переключение режимов симуляции

1.9. Пример практической двоичной трансляции

Для иллюстрации того, насколько видоизменяется машинный код при трансляции, рассмотрим реальный пример работы Simics. Компилятор JIT в этом симуляторе производит несколько стадий преобразования промежуточного представления в конечный код, включая стадии распределения регистров и оптимизации полученного кода. Для ускорения симуляции эти шаги производятся статически, на этапе компиляции. Ниже приведены 16 исходных гостевых инструкций

процессора IA-32, а также результат их преобразования — 532 хозяйские инструкции для той же самой архитектуры.

1.9.1. Исходный блок инструкций

```
simics> viper.mb.cpu0.core[0][0].disassemble-block %rip
Block 0x111cb .. 0x111fd matched. Compiled at 5176663075.
    Use count 1.
532 host instructions / 16 target instructions (= 33.3).

v:0x00000000000111cb p:0x00000000000111cb mov eax,ebx
v:0x00000000000111cd p:0x00000000000111cd mov ecx,ebx
v:0x00000000000111cf p:0x00000000000111cf cdq
v:0x00000000000111d0 p:0x00000000000111d0 idiv dword ptr
    -28[ebp]
v:0x00000000000111d3 p:0x00000000000111d3 mov eax,dword ptr
    -28[ebp]
v:0x00000000000111d6 p:0x00000000000111d6 sub ecx,edx
v:0x00000000000111d8 p:0x00000000000111d8 sub eax,edx
v:0x00000000000111da p:0x00000000000111da mov dword ptr -40[
    ebp],ecx
v:0x00000000000111dd p:0x00000000000111dd mov cl,byte ptr
    -32[ebp]
v:0x00000000000111e0 p:0x00000000000111e0 mov dword ptr -36[
    ebp],eax
v:0x00000000000111e3 p:0x00000000000111e3 mov eax,dword ptr
    -40[ebp]
v:0x00000000000111e6 p:0x00000000000111e6 shl edx,cl
v:0x00000000000111e8 p:0x00000000000111e8 cmp eax,dword ptr
    [0x30e68]
v:0x00000000000111ee p:0x00000000000111ee lea edx,0x70000[
    esi][edx]
v:0x00000000000111f5 p:0x00000000000111f5 mov dword ptr -44[
    ebp],edx
v:0x00000000000111f8 p:0x00000000000111f8 je 0x11332
```

1.9.2. Результат трансляции

```
0x155d69a0  sub dword ptr [cpu + turbo_event_counter],0x10
0x155d69a7  jle 0x155d7157
0x155d69ad  mov eax,dword ptr [cpu + RBX]
0x155d69b3  mov ebp,eax
0x155d69b5  mov dword ptr [cpu + RAX],ebp
```

```
0x155d69bb  mov dword ptr [cpu + hi32(RAX)],0x0
0x155d69c5  mov dword ptr 4[esp],eax
0x155d69c9  mov dword ptr [cpu + RCX],eax
0x155d69cf  mov dword ptr [cpu + hi32(RCX)],0x0
0x155d69d9  mov dword ptr 8[esp],ebp
0x155d69dd  mov edi,ebp
0x155d69df  shr edi,31
0x155d69e2  test edi,edi
0x155d69e4  je 0x155d7134
0x155d69ea  mov edi,0xffffffff
0x155d69ef  mov dword ptr 12[esp],edi
0x155d69f3  mov dword ptr [cpu + RDX],0xffffffff
0x155d69fd  mov dword ptr [cpu + hi32(RDX)],0x0
0x155d6a07  mov ecx,dword ptr [cpu + RBP]
0x155d6a0d  mov dword ptr 16[esp],ecx
0x155d6a11  mov edx,ecx
0x155d6a13  add edx,0xfffffe4
0x155d6a16  mov ebx,dword ptr 20[esp]
0x155d6a1a  xor ebx,ebx
0x155d6a1c  mov eax,dword ptr [cpu + ss_base]
0x155d6a22  mov ecx,dword ptr [cpu + hi32(ss_base)]
0x155d6a28  mov edi,edx
0x155d6a2a  mov ebp,ebx
0x155d6a2c  add edi,eax
0x155d6a2e  adc ebp,ecx
0x155d6a30  mov eax,edi
0x155d6a32  shr eax,8
0x155d6a35  and eax,0x3ff0
0x155d6a3b  add eax,dword ptr [cpu + stc_load_current_mode]
0x155d6a41  cmp ebp,dword ptr 4[eax]
0x155d6a44  jne 0x155d711c
0x155d6a4a  mov ebp,dword ptr [eax]
0x155d6a4c  xor ebp,edi
0x155d6a4e  and ebp,0xfffff003
0x155d6a54  jne 0x155d711c
0x155d6a5a  mov ebp,dword ptr 8[eax]
0x155d6a5d  mov ebp,dword ptr 0[ebp][edi]
0x155d6a61  mov edi,dword ptr 24[esp]
0x155d6a65  xor edi,edi
0x155d6a67  mov ebx,ebp
0x155d6a69  test ebp,ebp
0x155d6a6b  jne 0x155d6a83
0x155d6a6d  test edi,edi
```

```
0x155d6a6f  jne 0x155d6a7d
0x155d6a71  add dword ptr [cpu + turbo_event_counter],0xc
0x155d6a78  call turbo_raise_exception
0x155d6a7d  mov ebp,dword ptr 28[esp]
0x155d6a81  jmp 0x155d6a87(unknown call target)
0x155d6a83  mov ebp,dword ptr 28[esp]
0x155d6a87  xor ebp,ebp
0x155d6a89  mov edi,dword ptr 32[esp]
0x155d6a8d  xor edi,edi
0x155d6a8f  mov dword ptr 32[esp],edi
0x155d6a93  mov edi,dword ptr 8[esp]
0x155d6a97  or ebp,edi
0x155d6a99  mov dword ptr 28[esp],ebp
0x155d6a9d  mov ebp,dword ptr 32[esp]
0x155d6aa1  mov edi,dword ptr 12[esp]
0x155d6aa5  or edi,ebp
0x155d6aa7  mov eax,ebx
0x155d6aa9  cdq
0x155d6aaa  push edx
0x155d6aab  push eax
0x155d6aac  push edi
0x155d6aad  mov ebp,dword ptr 40[esp]
0x155d6ab1  push ebp
0x155d6ab2  call turbo_sdiv64
0x155d6ab7  add esp,0x10
0x155d6aba  mov ecx,edx
0x155d6abc  mov ebp,eax
0x155d6abe  test edx,edx
0x155d6ac0  jl 0x155d6ad8
0x155d6ac2  jg 0x155d6acc
0x155d6ac4  cmp eax,0x7fffffff
0x155d6aca  jbe 0x155d6ad8

<... Пропущено ...>

0x155d7045  mov al,byte ptr [cpu + pc_flags.zf]
0x155d704b  movzx eax, al
0x155d704e  jmp 0x155d7012(unknown call target)
0x155d7050  push 2
0x155d7055  push ebp
0x155d7056  push ecx
0x155d7057  mov ebp,dword ptr 124[esp]
0x155d705b  push ebp
```

```
0x155d705c  mov ebp,dword ptr 124[esp]
0x155d7060  push ebp
0x155d7061  call turbo_stc_miss_store_uint32_le
0x155d7066  add esp,0x14
0x155d7069  mov ebp,dword ptr 104[esp]
0x155d706d  jmp 0x155d6fe4(unknown call target)
0x155d7072  push 4
0x155d7077  push 0
0x155d707c  push 200296
0x155d7081  call turbo_stc_miss_load_uint32_le
0x155d7086  add esp,0xc
0x155d7089  mov edi,eax
0x155d708b  mov edx,dword ptr 12[esp]
0x155d708f  jmp 0x155d6f1c(unknown call target)
0x155d7094  push 6
0x155d7099  push eax
0x155d709a  push ebp
0x155d709b  call turbo_stc_miss_load_uint32_le
0x155d70a0  add esp,0xc
0x155d70a3  mov ebp,eax
0x155d70a5  jmp 0x155d6d94(unknown call target)
0x155d70aa  push 7
0x155d70af  push edx
0x155d70b0  push edi
0x155d70b1  mov ebp,dword ptr 72[esp]
0x155d70b5  push ebp
0x155d70b6  mov ebp,dword ptr 24[esp]
0x155d70ba  push ebp
0x155d70bb  call turbo_stc_miss_store_uint32_le
0x155d70c0  add esp,0x14
0x155d70c3  mov edi,dword ptr 16[esp]
0x155d70c7  jmp 0x155d6d44(unknown call target)
0x155d70cc  push 8
0x155d70d1  push ecx
0x155d70d2  push ebp
0x155d70d3  call turbo_stc_miss_load_uint8
0x155d70d8  add esp,0xc
0x155d70db  mov ecx,dword ptr 4[esp]
0x155d70df  jmp 0x155d6cc8(unknown call target)
0x155d70e4  push 9
0x155d70e9  push ecx
0x155d70ea  push edi
0x155d70eb  mov ebp,dword ptr 56[esp]
```

```
0x155d70ef  push ebp
0x155d70f0  mov ebp,dword ptr 64[esp]
0x155d70f4  push ebp
0x155d70f5  call turbo_stc_miss_store_uint32_le
0x155d70fa  add esp,0x14
0x155d70fd  mov edi,dword ptr 16[esp]
0x155d7101  jmp 0x155d6c73(unknown call target)
0x155d7106  push 12
0x155d710b  push ecx
0x155d710c  push edi
0x155d710d  call turbo_stc_miss_load_uint32_le
0x155d7112  add esp,0xc
0x155d7115  mov ebp,eax
0x155d7117  jmp 0x155d6b9f(unknown call target)
0x155d711c  push 13
0x155d7121  push ebx
0x155d7122  push edx
0x155d7123  call turbo_stc_miss_load_uint32_le
0x155d7128  add esp,0xc
0x155d712b  mov ebp,eax
0x155d712d  mov edi,edx
0x155d712f  jmp 0x155d6a67(unknown call target)
0x155d7134  mov edi,dword ptr 12[esp]
0x155d7138  xor edi,edi
0x155d713a  mov dword ptr 12[esp],edi
0x155d713e  mov dword ptr [cpu + RDX],0x0
0x155d7148  mov dword ptr [cpu + hi32(RDX)],0x0
0x155d7152  jmp 0x155d6a07(unknown call target)
0x155d7157  mov dword ptr [cpu +
    turbo_exit_reason_and_offset],0x1001cb01
0x155d7161  ret
532 instructions, 1986 bytes, 0 spill instructions 0.00%, 65
     copy instructions 12.22%
```

1.10. Вопросы к главе 1

TODO ДОПОЛНИТЬ

Вариант 1

1. Какой вид программ обычно исполняется в непривилегированном режиме процессора?

2. Какие из нижеперечисленных сценариев подпадают под определение *самомодифицирующийся код*:

   a) программа читает один байт секции кода,
   b) программа изменяет один байт в секции данных,
   c) программа читает один байт из секции данных,
   d) программа изменяет байт в секции кода?

3. Какой вид преобразования адресов специфичен только для систем виртуализации:

   a) v2p,
   b) v2h,
   c) p2h?

4. Перечислите отличия ДТ от компиляции с ЯВО, мешающие применению классических оптимизаций последнего.
5. Выберите правильные составляющие задачи «code discovery» (обнаружение кода) в ДТ:

   a) поиск кода внутри исполняемого файла,
   b) поиск границ инструкций при работе двоичного транслятора,
   c) поиск границ инструкций при работе интерпретатора,
   d) различение гостевого кода от гостевых данных,
   e) декодирование гостевых инструкций,
   f) поиск некорректных гостевых инструкций.

Вариант 2

1. Какой тип инструкций наиболее сложен с точки зрения симуляции в режиме прямого исполнения:

   a) арифметические,
   b) привилегированные,
   c) с плавающей запятой,
   d) условные и безусловные переходы?

2. Какой вид программ обычно исполняется в привилегированном режиме процессора?

3. Определение понятия *капсула*, используемого в двоичной трансляции.
4. Какие порядки размеров капсул в системе двоичной трансляции наиболее вероятны:

    a) 1 инструкция,
    b) 10 инструкций,
    c) 100 инструкций,
    d) 1000 инструкций,
    e) 10000 инструкций?

5. Выберите все необходимые условия корректности применения гиперсимуляции процессора:

    a) нет обращений к внешней памяти,
    b) нет обращений к внешним устройствам,
    c) только один процессор в системе,
    d) состояние внешних устройств не меняется,
    e) состояние процессора не меняется.

Литература

1. Binary translation / Richard L. Sites [и др.] // Communications of the ACM. — 1993. — Фев. — Т. 36, № 2. — 69–81.
2. *Drepper Ulrich* The Cost of Virtualization // ACM Queue. — 2008. — Янв. — 30–35. — URL: http://queue.acm.org/detail.cfm?id=1348591.
3. Fast Instruction Set Simulation Using LLVM-based Dynamic Translation / Claude Helmstetter, Vania Joloboff, Zhou Xinlei, Gao Xiaopeng // International MultiConference of Engineers and Computer Scientists 2011. Т. 2188. — Springer, 2011. — С. 212–216. — URL: http://hal.inria.fr/hal-00646947.
4. Handbook of Floating-Point Arithmetic / Jean-Michel Muller [и др.]. — Birkhäuser Boston, 2010. — URL: http://perso.ens-lyon.fr/jean-michel.muller/Handbook.html ; ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-0-8176-4704-9.
5. *Hauser John* SoftFloat. — 1 июн. 2010. — URL: http://www.jhauser.us/arithmetic/SoftFloat.html (дата обр. 08.02.2013).
6. IEEE Standard for Floating-Point Arithmetic. — IEEE Computer Society, авг. 2008. — DOI: 10.1109/IEEESTD.2008.4610935. — URL: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4610933 ; IEEE Std 754-2008.
7. *Poon Wing-Chi*, *Mok A.K.* Improving the Latency of VMExit Forwarding in Recursive Virtualization for the x86 Architecture // System Science (HICSS), 2012 45th Hawaii International Conference on. — 2012. — С. 5604–5612. — DOI: 10.1109/HICSS.2012.320.
8. *Popek Gerald J.*, *Goldberg Robert P.* Formal requirements for virtualizable third generation architectures // Communications of the ACM. Т. 17. Вып. 7. — Июл. 1974.

9. PQEMU: A Parallel System Emulator Based on QEMU / Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, Yeh-Ching Chung // Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems. — Washington, DC, USA: IEEE Computer Society, 2011. — С. 276–283. — (ICPADS '11). — ISBN: 978-0-7695-4576-9. — DOI: 10.1109/ICPADS.2011.102. — URL: http://dx.doi.org/10.1109/ICPADS.2011.102.

10. *Ray Anton Chernoff, Hookway Ray* DIGITAL FX!32 Running 32-Bit x86 Applications on Alpha NT // in Proceedings of the USENIX Windows NT Workshop, USENIX Association. — 1997. — 37–42.

11. *Seebach Peter* Unrolling AltiVec, Part 1: Introducing the PowerPC SIMD unit. — 2005. — URL: http://www.ibm.com/developerworks/power/library/pa-unrollav1.

12. *Smith James E., Nair Ravi* Virtual machines – Versatile Platforms for Systems and Processes. — Elsevier, 2005. — ISBN: 978-1-55860-910-5.

13. Software techniques for avoiding hardware virtualization exits / Ole Agesen, Jim Mattson, Radu Rugina, Jeffrey Sheldon // Proceedings of the 2012 USENIX conference on Annual Technical Conference. — Boston, MA: USENIX Association, 2012. — С. 35–35. — (USENIX ATC'12). — URL: http://dl.acm.org/citation.cfm?id=2342821.2342856.

14. *Topham Nigel, Jones Daniel* High speed CPU simulation using JIT binary translation // mobs. — 2007. — URL: http://homepages.inf.ed.ac.uk/npt/pubs/mobs-07.pdf.

15. Компиляторы: принципы, технологии и инструментарий, 2 издание / Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман ; пер. И. В. Красиков. — Вильямс, 2008. — ISBN: 978-5-8459-1349-4.

Список TODO

Даннаяя секция предназначена для напоминания авторам, какие темы необходимо раскрыть в последующих редакциях книги. Всем остальным просьба не обращать внимания.

- Интерпретация — более подробно разобрать декодирование.
- Интерпретация — описать проблему работы с cross-page инструкциями.
- Гиперсимуляция — описать механизм автоматической гиперсимуляции.
- Доступ к гостевым образам дисков — libguestfs `http://libguestfs.org/`.
- Обратное исполнение — выделить в главу, описать сценарии откатов, стратегии хранения точек отката, требования на память и т.д.
- Потактовая симуляция — рассказать про 0-cycle связи внутри узла.
- Современная виртуализация — описать стоимость VMEXIT [13] и проблемы с рекурсивной виртуализацией [7].
- Заключение — написать перспективы развития.