Программное моделирование вычислительных систем

Учебное пособие

Версия 2.6
7 октября 2013 г.

Оглавление

Об этой книге

Главы данной книги соответствуют основным лекциям курса «Основы программного моделирования ЭВМ», читаемого в Московском физико-техническом институте.

Нам очень важно мнение читателя. Если вы обнаружили опечатку, стилистическую, фактическую ошибку, которые, более чем вероятно, встречаются в тексте, или имеете замечания по содержанию и предложения по тому, как можно улучшить данный материал, то просим сообщить об этом по e-mail

grigory.rechistov@phystech.edu

Авторы

Данную книгу подготовил следующий коллектив лаборатории суперкомпьютерных технологий для биомедицины, фармакологии и малоразмерных структур им. В. М. Пентковского МФТИ: Г. С. Речистов, Е. А. Юлюгин, А. А. Иванов, П. Л. Шишпор, Н. Н. Щелкунов.

Актуальная версия данного текста доступна в Интернет по адресу `http://iscalare.mipt.ru/materials/course_materials/`.

Благодарности

Авторы выражают также благодарность всем слушателям курса и следующим людям, сообщившим свои замечания и исправления к тексту книги: Илье Куприку, Денису Шиляеву, Денису Лытову, Анатолию Костину, Виталию Антонову, Даниилу Альфонсо, Дмитрию Бородий, Ивану Андрееву, Наталье Иванчиковой, Марине Шимченко, Максиму Кузнецову, Святославу Кузьмичу.

Предисловие

> Мы легко принимаем
> действительность, может быть,
> потому, что интуитивно чувствуем:
> ничто реально не существует.
>
> *Хорхе Луис Борхес*

Симулятор, эмулятор, модель ЭВМ — под этими понятиями подразумевается компьютерная программа, способная имитировать работу некоторой реальной вычислительной системы (рис. 1.0). Процесс работы такой программы именуется *симуляцией*, и подразумевает изучение эволюции состояния модели во времени, отражающей изменения в поведении и состоянии изучаемого аппаратно-программного комплекса.

Существует несколько различных трактовок терминов *симуляция* и *эмуляция*. Наиболее общеупотребительное понимание различия между ними таково: симуляция — некий процесс, имитирующий внешние проявления системы, внутреннее его устройство при этом не повторяет детально оригинал; эмуляция же, кроме внешних эффектов, представляет внутреннюю структуру, максимально приближенную к оригинальной системе. Размышляя над этими определениями, можно заметить, что смысловая грань между ними очень тонка и в основном зависит от того, насколько глубоко мы готовы «заглянуть» в модель и в объект моделирования при исследовании; при этом эмуляция может легко оказаться симуляцией. По этой причине далее всюду в тексте понятия *симулятор*, *эмулятор*, *модель* будут использоваться взаимозаменяемо, и их значение будет больше зависеть от контекста обсуждения, чем от строгих определений.

Существующие модели ЭВМ характеризуются и различаются множеством параметров, таких как фокусировка на различных аспектах работы изучаемой системы, точность соответствия поведения реальной и моделируемой системы, скорость работы, внутренний дизайн и внешние интерфейсы к нему, задействование различных оптимизационных техник и др. Изучение этих вопросов и составляет цель данной

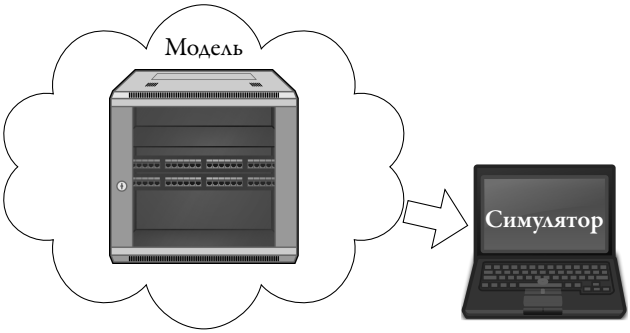


Рис. 1.0. Основная идея симуляции. Модель некоторой вычислительной системы в виде программы исполняется на компьютере другой конфигурации и/или архитектуры

работы.

Для максимально эффективного усвоения материала книги читателю рекомендуется иметь начальные знания по архитектуре ЭВМ; рекомендуется обратиться к великолепной книге [2]. Желательно понимание читателем общих принципов работы операционных систем, а также знакомство как минимум с одним языком программирования.

Нельзя не упомянуть книгу о виртуальных машинах, которая вдохновила авторов на написание этой работы [1]. Знакомство с ней также настоятельно рекомендуется всем читателям, желающим глубже разобраться в вопросе виртуализации.

Необходимо понимать, что методика моделирования применима не только к изучению вычислительных или цифровых, но и практически к любых технических, социальных или каких-либо иных систем. Читателю, желающему расширить своё понимание метода, рекомендуется книга [3].

В главе 1 дано описание областей применения симуляторов, введены ключевые понятия и приведены примеры использования технологии симуляции в прошлом и в настоящее время.

В главе 2 приводится пример построения простого симулятора процессора, основанного на интерпретации инструкций.

В главе 3 представлены дальнейшие пути увеличения скорости мо-

делей процессоров, такие как двоичная трансляция и аппаратная виртуализация.

В главе 4 рассматривается исследование вычислительных систем с помощью трасс исполнения — записи истории внешних событий.

Глава 5 описывает подходы к симуляции системы с множеством устройств, включая многопроцессорные системы.

В главе 6 рассматриваются подходы к параллельной симуляции; показывается, что эффективная и корректная реализация таких систем возможна, но в общем случае довольно сложна.

В главе 7 описан подход к потактовой симуляции, обусловленный иным характером обработки событий в системе и потому отличный от ранее рассмотренных.

В главе 8 показываются примеры организации хранилищ внутреннего состояния устройств, а также возможные для них оптимизации.

В главе 9 кратко рассматриваются назначение и устройство сверхоперативной памяти (кэш-память), а также подходы к её моделированию.

В главе 10 определяются существующие языки, используемые для написания моделей устройств, а также показывается их связь с языками, задействованными на поздних фазах, при проектировании устройств.

Глава 11 знакомит с особенностями обеспечения взаимодействия симуляции с внешним физическим миром.

В главе 12 даётся теоретический критерий возможности эффективной виртуализации вычислительных систем; затем проверяется соответствие ряда современных архитектур процессоров этому условию.

Обозначения

Всюду в тексте данной книги применяются следующие шрифтовые выделения и обозначения.

- Обычный текст используется для основного материала.
- `Моноширинный текст` вводится для исходных текстов программ на различных (псевдо)языках программирования, выделения ключевых слов, имён регистров устройств, листингов машинного кода.
- *Наклонный текст* служит для выделения новых понятий.
- Числа в шестнадцатеричной системе счисления имеют префикс **0x** (например, 0x12345abcd), в двоичной системе счисления — суффикс **b** (например, 10010011b).

При введении терминов, заимствованных из английского языка и не имеющих известных авторам общепринятых переводов на русский, в скобках после них будут указываться оригинальные иностранные выражения.

Замечание. *Параграфы, оформленные таким образом, являются необязательными для понимания курса и введены для того, чтобы показать порой неочевидные связи между приёмами моделирования и различными научными идеями и теориями.*

Литература

1. *Smith James E.*, *Nair Ravi* Virtual machines – Versatile Platforms for Systems and Processes. — Elsevier, 2005. — ISBN: 978-1-55860-910-5.
2. *Паттерсон Дэвид*, *Хэннесси Джон* Архитектура компьютера и проектирование компьютерных систем. — 4-е изд. — Питер, 2012. — ISBN: 978-5-459-00291-1.
3. *Тюкин В.Н.* Моделирование систем. — 2009. — URL: `http://gendocs.ru/v14098/?cc=1` (дата обр. 11.02.2013).

1. Применение программных моделей

...цифровая машина, получившая непосильное для неё задание, вместо того, чтобы заниматься решением проблемы самой, строит, если перейдён определённый порог, называемый барьером мудрости, следующую машину...

Станислав Лем. Кибериада. Сказки роботов

1.1. Введение

Разработка новых устройств для ЭВМ, таких как центральные процессоры, графические карты и сопроцессоры, наборы системной логики, сетевые карты, а также проектирование вычислительных комплексов и средств их взаимодействия, являются сложными и длительными процессами. Сложность эта обусловлена многими факторами, среди которых можно выделить следующие.

- Большое число составляющих систему устройств со сложными взаимосвязями, как явными, так и неявными, скрытыми или даже паразитными.
- Необходимость сохранения обратной совместимости с уже существующим оборудованием, дополнительно усложняющая дизайн.
- Необходимость обеспечения поддержки аппаратуры программными продуктами, помогающими раскрыть их полный потенциал: драйверами для операционных систем, компиляторами, профилировщиками и др. средствами разработки.

– Необходимость знать характеристики новой технологии (например, производительность, мощность энерговыделения, размеры) как можно раньше, чтобы принимать обоснованные в рамках рыночной конкуренции решения о целесообразности создания продуктов на её основе.
– Важность выявления ошибок проектирования на ранних стадиях. Стоимость исправления недостатков резко возрастает вместе с этапом проектирования, на котором они были обнаружены. Исправить ошибку в описании на бумаге в начале проекта несоизмеримо дешевле, чем отзывать партии бракованных изделий, уже изготовленных по спецификациям, содержащим изъяны (рис. 1.1).

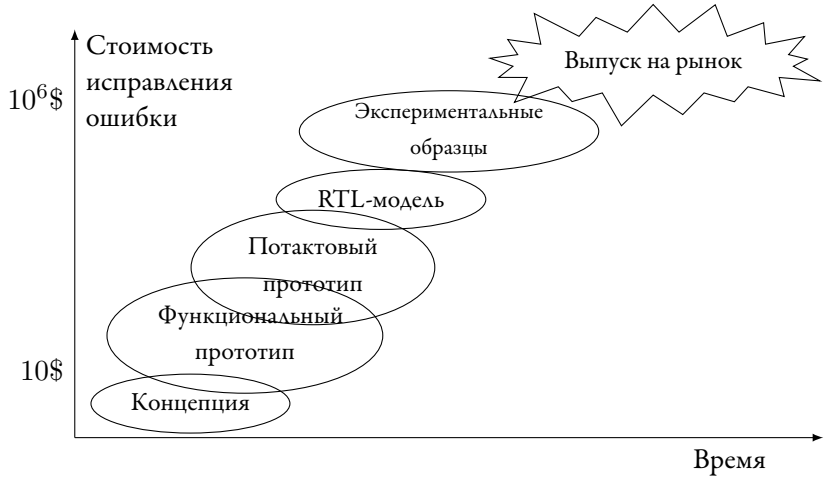


Рис. 1.1. Рост стоимости исправления ошибки с фазой проектирования устройства. Различные этапы проектирования могут перекрываться во времени, однако каждый из них подразумевает всё большие вложения ресурсов. Ошибки в уже выпущенном продукте могут повлечь за собой существенные финансовые и имиджевые потери для допустившей их компании

По этим причинам широко используется подход, когда разработка нового устройства предваряется созданием и сопровождается использованием его компьютерных моделей, способных с различной

точностью проявлять себя так, как работает реальное устройство. Построенные согласно этим принципам инструменты различаются назначением, точностью соответствия реальной аппаратуре, количеством моделируемых и конфигурируемых параметров, скорости выполнения, внутренней организацией и принципами работы. Даже не полностью соответсвующая реальности модель может быть практически полезной.

Замечание. *Моделирование возможно благодаря применению общесистемной методики борьбы со сложностью — модульностью подсистем и абстракцией их функций. Устройства, входящие в состав компьютера, соединяются между собой через строго определённые интерфейсы, диктующие лишь то, что будет передаваться через них и какой тип результата будет возвращён, но не определяющие, что будет располагаться на другом конце. Это позволяет оборудованию различных производителей быть совместимым друг с другом без необходимости раскрытия внутренней документации конкурентам. Также это позволяет подставлять вместо реальных устройств их модели. Более того, по разные стороны интерфейсов мы можем размещать модели различных подсистем, таким образом создавать полную модель всего компьютера, компоненты которой «не знают», что за интерфейсом скрывается не реальное устройство, а его модель (рис. 1.2).*

Рис. 1.2. Соединение компонент сложной системы через интерфейсы, определяющие форматы запроса и отклика, но не специфицирующие детали реализации необходимой функциональности. Это позволяет заменять некоторые или даже все части такой системы на программные модели

1.2. Применения моделей ЭВМ

Перечислим лишь некоторые практические способы применения программных моделей.

Раннее обнаружение ошибок проектирования.
Программирование — процесс значительно менее затратный, чем испытания реального железа, и что куда важнее, исправление ошибки в программе занимает минуты, тогда как повторный выпуск опытного образца аппаратуры может занять месяцы.

Написание сопутствующего аппаратуре ПО. Ранняя доступность модели устройства позволяет использовать её для разработки драйверов, прошивок (таких как BIOS и UEFI [5]) и даже операционных систем и компиляторов параллельно с разработкой самого устройства. В наше время нередка ситуация, что драйвера для нового оборудования готовы и отлажены ещё до официальной доступности предназначенного для них оборудования.

Построение и исследование экспериментальных решений.
Моделирование позволяет быстрее и дешевле изучать пространство проектирования (*англ.* design space) для определения параметров, при которых устройство или система будет иметь наилучшие характеристики. Для исследователей интерес часто представляют количественные характеристики новых систем, такие как скорость работы, степень загруженности подсистем, потребление энергии и т.п. Иногда подобный анализ можно провести и без симуляции, используя аналитические методики, теорию массового обслуживания, экстраполяцию измерений на существующей аппаратуре и т.д. Однако моделирование даёт наибольшую гибкость.

Качественно-функциональные свойства. Под этим термином понимается изучение, работает или нет новая технология в принципе, безотносительно её скоростных характеристик. В этом случае альтернатив симуляции практически не остаётся, по-

скольку необходимо изучить функционирование системы.

Выполнение программ на «неродной» архитектуре. В этом случае модель обеспечивает прослойку, позволяющую выполнять приложения без перекомпиляции на машинах, изначально не предназначенных для исполнения этих программ.

1.3. Терминология

Существует много терминов, относящихся к изучаемой области моделирования. Определим основные из них, которые будут использоваться в дальнейшем.

Эмулятор (*англ.* emulator) — программа, моделирующая некоторую физическую систему путём имитации внутренней структуры и процессов, происходящих внутри подсистем аппаратуры.

Симулятор (*англ.* simulator) — программа, моделирующая некоторую физическую систему через предоставление корректных интерфейсов входящих в неё подсистем и обеспечивающая правильное их функционирование, но не гарантирующая того, что их внутреннее устройство будет похоже на устройство аналогичных подсистем реальной ЭВМ (т.е. работающая как «чёрный ящик»).

Следует отметить, что разница в определениях симулятора и эмулятора размыта, поэтому мы будем считать оба термина эквивалентными.

Хозяин (*англ.* host) — физическая вычислительная система, на которой исполняются программы, в том числе моделирующие другие ЭВМ. При этом потребляются хозяйские ресурсы (процессорное время, память, электроэнергия и т.п.). Также в литературе встречается синонимичный термин *инструментальная система*.

Гость (*англ.* guest) — система, поведение которой призван отражать симулятор и внутри которой исполняются гостевые приложения. Синонимичным является понятие *целевая система* (*англ.* target system).

Виртуализация (*англ.* virtualization) — выполнение одной или более гостевых программ, в т.ч. операционных систем, внутри изолированных друг от друга окружений. При этом управляющая программа, в данном случае называемая *гипервизором* (*англ.* hypervisor) или монитором виртуальных машин (*англ.* virtual machine monitor, VMM), контролирует доступ виртуализованных приложений к физическим ресурсам системы. В главе 12 рассматриваются теоретические и практические аспекты виртуализации. Сейчас же определим два основных типа гипервизоров.

Гипервизоры первого типа (автономные гипервизоры) работают прямо на хозяйской аппаратуре, т.е. не требуют для своей работы операционной системы, они берут её функции на себя и являются привилегированными приложениями. Данное обстоятельство позволяет минимизировать накладные расходы виртуализации. Вместе с тем при разработке автономного монитора приходится тратить много усилий на поддержку в нём функций операционной системы. На рис. 1.3 приведён пример взаимного расположения программных компонент при использовании гипервизора первого типа.

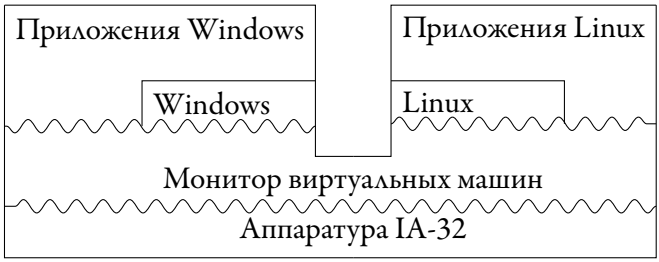


Рис. 1.3. Пример использования гипервизора первого типа для одновременного запуска приложений двух различных операционных систем

Примеры существующих мониторов виртуальных машин первого типа: VMware ESX(i) Server [25], Xen [16].

Гипервизоры второго типа не заменяют операционную систему, но работают поверх неё как обычное пользовательское приложе-

ние (рис. 1.4), иногда требуя установки драйверов или модулей ядра, работающих с повышенным приоритетом. Примеры таких программных продуктов: Oracle VirtualBox [24], KVM [10] (*англ.* kernel-based virtual machine). Накладные расходы на виртуализацию при их работе выше, чем при использовании мониторов первого типа.

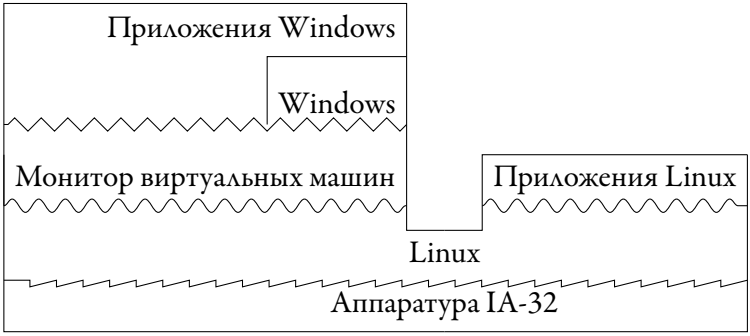


Рис. 1.4. Пример использования гипервизора второго типа для запуска приложений второй операционной системы при уже загруженной основной

Полноплатформенный симулятор (*англ.* full platform simulator) — модель, включающая в себя компоненты, достаточные для получения поведения некоторой ЭВМ в целом, т.е. состоящая как минимум из следующих основных устройств: процессора, памяти, дискового устройства, сетевого устройства, клавиатуры, мыши, монитора и др. Внутри такого симулятора возможно запустить немодифицированную операционную систему, и она будет работать так же, как работала бы на реальной аппаратуре.

Симулятор режима приложения (*англ.* application mode simulator) — программа, предназначенная для запуска «обычных» прикладных приложений (т.е. не операционных систем, BIOS или другого системного ПО). Целевые программы при этом ожидают активное присутствие определённой операционной системы, и потому симулятор обязан в том числе эмулировать необходимые системные вызовы для того, чтобы создать окружение, неотличимое от предоставляемого операционной систе-

мой. При этом модель получается жёстко привязанной к конкретному варианту системного ПО, так как список и формат системных вызовов и прочих интерфейсов приложений может заметно меняться между ОС (например, Windows, Linux и Mac OS имеют разные механизмы вызова операций в контексте ОС) и даже внутри одной ОС между её версиями (например, Linux 2.4 и Linux 2.6). Как правило, количество моделируемого при этом аппаратного обеспечения минимально.

Функциональная модель (*англ.* functional model) — симулятор, точность которого ограничена корректной функциональностью целевых приложений без обеспечения правильных значений длительностей операций, наблюдаемых в реальности. Например, доступ к памяти возвращает правильное значение, но за один такт моделируемого времени, тогда как в реальности он занял бы от 20 до 100 тактов в зависимости от состояния системы кэшей. Подобные модели недостаточно точны для предсказания производительности, но, как правило, достаточны для корректной работы большинства гостевого ПО, включая операционные системы, так как алгоритмы отдельных инструкций соответствуют реальности.

Потактовая модель (*англ.* cycle precise model, performance model) — симулятор, корректно высчитывающий ход времени внутри моделируемой системы. Он моделирует её внутреннее устройство более детально, чем это делается в функциональных моделях. Потактовые модели обычно во много раз медленнее функциональных.

Гибридная модель (*англ.* hybrid model) — система, частично реализованная в программе для обычной ЭВМ (например, для персонального компьютера), а частично — на специализированном оборудовании (например, на ПЛИС[1]). Применяется в тех случаях, когда чисто программное моделирующее решение недостаточно быстро.

[1]Программируемая логическая интегральная схема.

1.4. Симуляция и виртуализация на различных уровнях абстракций исследуемых систем

Для того чтобы яснее понять сходства различных типов симуляторов, а также более чётко определить специфику термина *виртуализация*, рассмотрим иерархию абстракций, наблюдаемую при работе программы на вычислительной машине, и их место в ней (рис. 1.5).

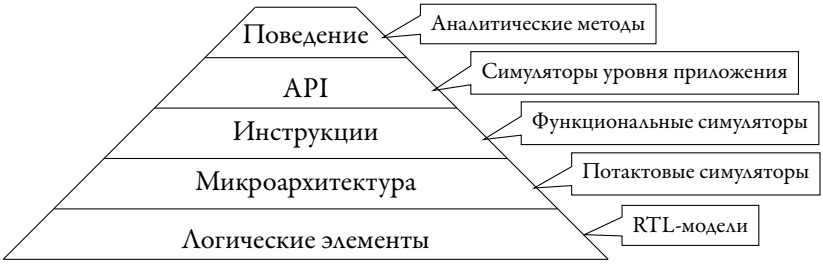


Рис. 1.5. Место различных типов симуляции в стеке существующих интерфейсов вычислительных систем

На самом верхнем уровне находятся её алгоритмы, в общем случае не привязанные к аппаратуре. Поскольку любую последовательность вычислений человек теоретически может провести с использованием ручки и бумаги (оставим в стороне вопрос, сколько времени у него это займёт), то «имитация» работы компьютера на этом уровне представляет собой анализ алгоритма самого вычисления. При этом такая аналитическая модель может быть неполной, упрощённой для того, чтобы передать лишь существенные для конкретного исследования аспекты работы, например, только её энергопотребление или только детали взаимодействия с внешними агентами. В этом случае «хозяйской» системой является сам человек.

Реальные программы редко бывают написаны полностью с нуля, чаще всего они используют в своей работе сторонние библиотеки, подпрограммы, процедуры, функции и т.п., в том числе сюда относятся сервисы операционной системы — системные вызовы. Для возможности эффективного взаимодействия кода библиотек и программ вводятся соглашения, такие как интерфейс пользовательских прило-

жений (*англ.* application program interface, API) и интерфейс двоичных приложений (*англ.* application binary interface, ABI), определяющие форматы передачи входных данных и результатов, а также ожидаемую от подпрограмм функциональность. Симулятор заменяет алгоритм каждого вызова API/ABI другим, с достаточной точностью передающим работу оригинальной подпрограммы и имеющим совместимый формат данных. При этом хозяйские и гостевые системы не обязаны использовать одни и те же соглашения. Таким образом работают модели уровня приложений — они заменяют операционную систему, ожидаемую пользовательским кодом, набором собственных реализаций API. Примеры описаний программных интерфейсов приложений — стандарт MPI [13] и стандарт OpenMP [15]. Пример документа, описывающего двоичный интерфейс — AMD64 ABI [22].

Если посмотреть в работу приложений ещё глубже, то любой вычислительный процесс состоит из последовательного и параллельного исполнения инструкций одного или более процессоров. Формат и ожидаемая функция каждой из них описаны в специальных документах — руководствах для разработки программ (*англ.* software development manual, SDM). Примеры таких документов для ISA (*англ.* instruction set architecture, архитектура набора инструкций): [1, 2, 9, 20, 26]. Функциональный симулятор заменяет алгоритм каждой гостевой инструкции на эквивалентный, но представленный в терминах хозяйской системы.

Исполнение каждой машинной инструкции может быть разделено на несколько стадий, имеющих различную длительность, величина которой может зависеть от ряда факторов. Кроме того, в работе вычислительной системы могут присутствовать процессы, не отражённые на уровне ISA, но тем не менее влияющие на её функционирование, например, работа кэшей или изменение частоты процессора. Для их учёта симулятор должен абстрагировать процессы на ещё более низком, микроархитектурном уровне. При этом становится возможным более точно моделировать времена работы приложений как сумму длительностей микроопераций. Отметим, что документация на данный уровень представления процессоров чаще всего является внутренним секретом компаний, недоступна для независимых разра-

ботчиков приложений и может быть получена только при подписании ряда соглашений о неразглашении (*англ.* non disclosure agreement, NDA).

Последний рассматриваемый нами уровень абстракции вычислительной системы — это уровень логических узлов, таких как триггеры, счётчики, отдельные логические элементы И, ИЛИ, НЕ, шины передачи данных и т.п. Их симуляция наиболее точно передаёт внутренние процессы, которые будут происходить при функционировании микросхемы. Работа каждого её узла в модели будет реализована с помощью отдельной процедуры симулятора. Однако отметим, что при этом уже достаточно сложно становится отслеживать, что приложение выполняет на макроскопическом, алгоритмическом уровне, с которого мы начали наше рассмотрение.

Как и симуляция, виртуализация основывается на принципе замены реализаций алгоритмов используемой вычислительной системы начиная с некоторого слоя абстракции. Однако при этом акцент делается на обеспечение следующих двух свойств гостевых систем: их изолированности от хозяина и друг от друга и разделения (совместного использования) хозяйских ресурсов, таких как оперативная память, процессорное время, дисковое пространство и т.д. (рис. 1.6).

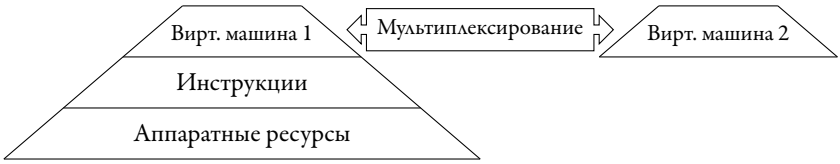


Рис. 1.6. Виртуализация как сочетание изоляции виртуальных машин и разделения хозяйских ресурсов между ними. Под мультиплексированием понимается временно́е разделение выполнения нескольких виртуальных процессоров на одном хозяйском

Таким образом, почти любой симулятор является виртуальной машиной, потому что он обеспечивает изоляцию (при условии, что реализован корректно), а несколько его копий, запущенные одновременно, обеспечат разделение ресурсов. Монитор виртуальных машин не обязательно является симулятором в том смысле, что архитекту-

ра систем гостя и хозяина могут совпадать (т.е. симуляция при этом «тривиальна», однако в главе 3 показывается, что этот случай не так прост); разделение ресурсов при виртуализации, как правило, более эффективно, и она обеспечивает меньшие накладные расходы.

В заключение отметим, что предложенная выше классификация допускает возможность присутствия симулятора более чем на одном уровне абстракции. Например, модели уровня приложения могут эмулировать не только системные вызовы, но также некоторые инструкции, а функциональный симулятор может включать модель кэша.

1.5. История использования симуляции

В различных формах компьютерные симуляторы используются с зари возникновения вычислительной техники. Так, IBM System/360 Model 67 выпуска 1967 года поддерживала виртуальные машины на аппаратном уровне [4], а саму System/360 эмулировали многие последующие ЭВМ, такие как RCA Spectra/70.

Приведём лишь несколько примеров использования симуляторов на ранних стадиях разработки новых архитектур и для обеспечения совместимости с существующим кодом.

- Интересным примером использования симуляции для обеспечения обратной совместимости является продукция компании Apple. Первые компьютеры Machintosh (1980-е гг.) были построены на процессорах Motorola 68x0 (общее название для серии чипов). В 1994 году новые компьютеры Apple стали использовать процессоры PowerPC. Для обеспечения работы приложений, написанных для старого оборудования, с ними поставлялся эмулятор [23], работа которого была максимально прозрачна для пользователя и приложений. В 2006 году произошёл ещё один переход — на архитектуру Intel®IA-32. И снова для совместимости новые Макинтоши имели встроенный эмулятор с именем Rosetta [17, 19].
- В 2001 году для новой архитектуры Intel® Itanium™ был использован симулятор Gambit [6].

– В 2001 году для портирования операционной системы NetBSD на тогда ещё официально не выпущенную архитектуру AMD64 был использован симулятор Virtutech Simics [14].
– В современных компьютерных системах часто используются подсистемы, предназначенные для обеспечения совместимости с устаревшим ПО и фактически являющиеся своеобразными симуляторами.
– Во всех 32-битных ОС Microsoft Windows серии NT существует система NTVDM [18] — эмулятор 16-битного режима MS-DOS. Отметим, что в 64-битных редакциях Windows по ряду причин технического характера подобного слоя совместимости нет. В свою очередь, запуск 32-битных приложений в 64-битных вариантах также требует создания специального окружения, отличного от того, в котором исполняются «родные» приложения [11, глава 3].
– В некоторых версиях Microsoft Windows 7 (Professional, Ultimate и Enterprise) доступен режим совместимости с Microsoft Windows XP [27], выполненный в виде предустановленной в Virtual PC операционной системы, взаимодействие с которой производится по протоколу RDP.
– Для архитектуры Intel® Itanium™ существует система совместимости для запуска кода архитектуры IA-32 [7], активно задействующая технологии статической и динамической двоичной трансляции (см. главу 3).
– В 2012 году компания ARM объявила о введении нового 64-битного расширения своей архитектуры ARMv8. Первые образцы реальных процессоров ожидаются в 2013 году, до этого момента разработка и адаптация существующего кода может проводиться на симуляторе [21].

1.6. Обзор существующих симуляторов и виртуальных машин

VMware ESX(i) Server [25]. Коммерческий продукт, являющийся гипервизором первого типа. Предназначен для виртуализации

крупных систем уровня предприятия. VMware ESXi Server доступен бесплатно, тогда как VMware ESX Server требует коммерческой лицензии и предоставляет расширенные возможности.

VMware Workstation Проприетарный продукт, являющийся монитором виртуальных машин второго типа. Работает на операционных системах Windows и Linux. Бесплатный вариант для некоммерческого использования называется VMware Player.

Xen Открытый монитор виртуальных машин первого типа, развиваемый компанией Citrix [16]. Работает на большом числе хозяйских архитектур, включая ARM и IA-32. Применяется для крупномасштабной виртуализации (используется, например, компанией Amazon в облачном сервисе Amazon Elastic Compute Cloud).

Qemu Открытый симулятор различных систем [3]. Портирован для большого числа операционных систем. В качестве гостевых архитектур поддерживает системы IA-32, IA-32 EMT64, IA-64, PowerPC, Alpha, SPARC 32/64, ARM...; в качестве хозяйских систем могут использоваться IA-32, IA-32 EMT64, ARM, CRIS, LM32, MicroBlaze, MIPS, SPARC 32/64, PowerPC.

KVM (*англ.* Kernel-based Virtual Machine). Открытый монитор виртуальных машин второго типа, основанный на технологиях Qemu и встроенный в ядро операционной системы Linux [10]. Популярен для задач виртуализации Linux и развивается фирмой Red Hat.

Oracle VirtualBox Открытый монитор виртуальных машин второго типа [24] для гостевых и хозяйских архитектур IA-32 и портирован для работы внутри Windows, Linux, Mac OS X и других операционных системах. Является весьма популярным решением для «домашней» пользовательской виртуализации. Разрабатывается компанией Oracle.

Bochs Открытый монитор виртуальных машин второго типа [12]. Работает на Windows, Linux, Mac OS X и других операционных системах. Является популярным решением для поддержки выполнения программ, скомпилированных для IA-32, на архитектурах, отличных от IA-32.

1.7. Производительность симуляции

Любая модель должна описывать изучаемый объект с точностью, достаточной для нужд исследования. В случае компьютерной симуляции не менее важным фактором становится темп протекания процесса исследования — **скорость симуляции**. С одной стороны, она напрямую зависит от быстродействия хозяйской системы, на которой запущена программа. С другой стороны, важно то, насколько оптимизирован сам симулятор для того, чтобы использовать весь предлагаемый аппаратурой потенциал. Наконец, скорость критическим образом зависит от самого сценария симуляции, т.е. какие гостевые приложения запущены и насколько требовательны они к ресурсам гостевой системы, особенно к наиболее сложно виртуализуемым их классам, например, высокоскоростным периферийным устройствам. Другими словами, практически для любого симулятора можно найти «плохой» гостевой код, при симуляции которого скорость будет очень низкой. Отметим, что запуск такого приложения на реальной системе в большинстве случаев также покажет невысокую производительность.

1.7.1. Способы определения скорости

Скорость симуляции, понимаемая как темп изменения значения виртуального времени, является первичной метрикой. Однако молниеносно пролетающие виртуальные секунды ещё не означают, что гостевые приложения эффективно его используют.

Второй метрикой является демонстрируемая гостевым программным обеспечением скорость работы. Единицей измерения при этом выступает IPS (*англ.* instructions per second). Однако важно при этом понимать, что для учёта влияния симуляции на скорость эта величина равна среднему числу *гостевых* инструкций, исполняемых за одну *хозяйскую* секунду. Чаще всего используют более крупные единицы, например, миллионы инструкций в секунду — MIPS.

Следующая используемая на практике величина для характеризации производительности — отношение времени работы интересующей исследователя программы внутри модели к длительности её ис-

полнения «снаружи», на идентичной хозяйской системе. В случае тождественности архитектур гостя и хозяина эта величина почти всегда больше единицы (внутри симулятора программа работает дольше), поэтому она получила название **накладные расходы, вызванные виртуализацией** (*англ.* simulation overhead).

Для приложений, производящих большое количество вычислений (например, задачи математического моделирования, решение задач уравнений математической физики и т.п.), применяется также другая метрика — **FLOPS** (*англ.* floating point operations per second), определяющая количество операций над числами с плавающей запятой (*англ.* floating point number), совершаемых за одну секунду. Допустимые форматы таких чисел (т.н. одинарная, двойная точность и т.п.) определяются стандартом IEEE 754-2008 [8].

1.7.2. Соотношение скоростей симулируемого и реального времени

Рассмотрим три варианта, как могут соотноситься скорости течения времени внутри (гостевое, симулируемое время) и снаружи (реальное, абсолютное время) симулятора.

1. *Симулируемое время течёт медленнее реального.* Этот случай очень часто встречается на практике из-за необходимости программной реализации всех алгоритмов и механизмов, в реальной аппаратуре воплощённых «в железе». Так, существующие модели требуют исполнения от десятков до тысяч или более инструкций для симуляции одной. Другой пример — моделирование двухпроцессорной системы на однопроцессорной требует как минимум в два раза больше реального времени.
2. *Симуляция быстрее реальности.* Такая ситуация также встречается на практике. Например, на процессоре с частотой 1 ГГц моделируется похожий процессор с частотой 10 МГц. При достаточно эффективной схеме работы может получиться, что модель будет работать в 10—100 раз быстрее, чем она работала бы в реальности. Другая ситуация — использование гиперсимуляции, при которой модель быстро продвигает время вперёд, не изме-

няя состояния, тогда как реальная аппаратура «честно» выполнила бы все циклы. Столь быстрое исполнение не всегда желаемо, например, при взаимодействии с пользователем вводимые клавиши будут нажиматься очень быстро, и человек не успеет прореагировать. В таких случаях достаточно легко снизить скорость симуляции с помощью пауз абсолютного времени, искусственно вставляемых между исполнениями устройств.

3. *Темп симуляции равен (или почти равен) темпу течения физического времени* Как правило, это необходимо в интерактивных системах, зависящих от ввода пользователя, например учебная или игровая симуляция управления автомобилем, самолётом. Для обеспечения такого режима необходимо специально следить за тем, чтобы скорость исполнения модели выдерживалась в определённых рамках, тогда как искусственно замедлить её относительно легко, ускорить исполнение зачастую непросто; возможное решение — использования более простых моделей, дающих меньшую, но приемлемую точность состояния системы по сравнению с тем, что мы имели бы в реальности. Для описанного выше примера это может быть связано с уменьшением числа кадров в секунду, точности прорисовки деталей, «затуманиванием» удалённого пространства и т.д. В задачах симуляции ЭВМ понижение точности исполнения инструкций или поведения устройств недопустимо, поэтому для них требование исполнения в реальном времени ставится очень редко.

1.8. Вопросы к главе 1

Вариант 1

1. Определите понятие «функциональный симулятор».
2. Определите понятие «полноплатформенный симулятор».
3. Перечислите все правильные виды сложностей, возникающих при разработке цифровых систем, успешно решаемых с помощью моделирования.

    а) Необходимость знать характеристики новой технологии

как можно раньше.
   b) Необходимость выявления ошибок проектирования на ранних стадиях.
   c) Большое энергопотребление реальных образцов.

4. Критерий изоляции исполнения гостевого приложения.
5. Как расшифровывается обозначение «RTL-модель» в контексте разработки аппаратуры?

   a) Run-time library.
   b) Register transfer level.
   c) Register-transistor logic.

6. Определение гипервизора первого типа.
7. Определение величины MIPS, используемой для измерения скорости программ.
8. Какой из указанных ниже бенчмарков используется для оценки и сравнения эффективности работы систем виртуализации:

   a) SPECfp,
   b) SPECpower,
   c) SPECint,
   d) SPECvirt,
   e) SPECjbb?

Вариант 2

1. Определение потактового симулятора.
2. Определение симулятора уровня приложений.
3. Перечислите все правильные виды сложностей, возникающих при разработке цифровых систем, успешно решаемых с помощью моделирования.

   a) Большое число составляющих систему устройств со сложными взаимосвязями.
   b) Сложность получения лицензий на новое оборудование.
   c) Обеспечение поддержки аппаратуры программными средствами разработки.

4. Перечислите стадии создания нового устройства с задействованием моделирования в правильном порядке.
    a) Функциональная модель.
    b) Разработка концепции устройства.
    c) RTL-модель.
    d) Потактовая модель.
    e) Выпуск продукции на рынок.
    f) Экспериментальные образцы.
5. Определение гибридного симулятора.
6. Определение гипервизора второго типа.
7. Определение понятия FLOPS.
8. Определение понятия *floating point number*.

Литература

1. Alpha Architecture Book. — 1998. — URL: http://openwatcom.mirror.fr/devel/docs/alphaarchitecturehandbook.pdf (дата обр. 20.10.2012) ; http://lib.mipt.ru/book/282937/.
2. AMD64 Architecture Programmer's Manual Volume 1: Application Programming. — Advanced Micro Devices. 2012. — URL: http://support.amd.com/us/Processor_TechDocs/24592_APM_v1.pdf (дата обр. 29.12.2012).
3. *Bellard Fabrice* QEMU, a Fast and Portable Dynamic Translator // FREENIX Track: 2005 USENIX Annual Technical Conference. — 2005. — URL: http://www.usenix.org/publications/library/proceedings/usenix05/tech/freenix/full_papers/bellard/bellard.pdf (дата обр. 15.02.2012).
4. *Blaauw G.A.* The structure of SYSTEM/360, Part V: Multisystem organization // IBM Systems Journal. — 1964. — Т. 3. — С. 181. — URL: http://www.research.ibm.com/journal/sj/032/blaauw.pdf.
5. *Doran Mark*, *Zimmer Vincent*, *Rothman Michael* Beyond BIOS: Exploring the Many Dimensions of the Unified Extensible Firmware Interface // Intel Technology Journal. — 2011. — Окт. — Т. 15, вып. 1. — С. 8—21. — ISSN: 1535-864X. — URL: http://www.intel.com/technology/itj/2011/v15i1/index.htm (дата обр. 02.07.2012).
6. *Dulong Carole*, *Shrivastav Priti*, *Refah Azita* The Making of a Compiler for the Intel® Itanium™ Processor // Intel Technology Journal. — 2001. — Авг. — URL: http://download.intel.com/technology/itj/q32001/pdf/art_4.pdf.
7. IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems / Leonid Baraz [и др.] // In 36th International Symposium on Microarchitecture. — 2003. — 191–201.

8. 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.
9. Intel® 64 and IA-32 Architectures Software Developer's Manual. Volumes 1–3. — Intel Corporation. 2012. — URL: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html (дата обр. 25.06.2012).
10. KVM wiki. — URL: http://www.linux-kvm.org/page/Main_Page.
11. *M. Russinovich*, *D. Solomon*, *A. Ionescu* Windows Internals, 6th Edition, Part 1. — Microsoft Press, 2012. — ISBN: 978-0-7356-4873-9.
12. *Mihoka Darek*, *Shwartsman Stanislav* Virtualization Without Direct Execution or Jitting: Designing a Portable Virtual Machine Infrastructure // ISCA-35 Proceedings of the 1st Workshop on Architectural and Microarchitectural Support for Binary Translation. — URL: http://bochs.sourceforge.net/Virtualization_Without_Hardware_Final.pdf (дата обр. 05.05.2012).
13. MPI: A Message-Passing Interface Standard. Version 2.2. — Message Passing Interface Forum. Сент. 2009. — URL: http://www.mpi-forum.org/docs/docs.html.
14. NetBSD/amd64. — 2007. — URL: http://www.netbsd.org/ports/amd64/ (дата обр. 09.09.2012).
15. OpenMP Application Program Interface version 3.0. — 2008. — URL: http://www.openmp.org/mp-documents/spec30.pdf.
16. *Pratt Ian* Xen and the art of virtualization. — 2006. — URL: http://www.cl.cam.ac.uk/netos/papers/2006-xen-ols.pdf.
17. Rosetta. — Apple Computer Inc. — URL: http://www.apple.com/asia/rosetta/ (дата обр. 09.09.2012).

18. Running Nonnative Applications in Windows 2000 Professional. — Microsoft Corporation. — URL: http://technet.microsoft.com/ru-ru/library/cc939094(en-us).aspx (дата обр. 09.09.2012).

19. *Singh Amit* Mac OS X Internals: A Systems Approach. — Addison Wesley. — ISBN: 978-0-321-27854-8. — URL: http://osxbook.com/ (дата обр. 09.09.2012).

20. *Sloss Andrew N.*, *Symes Dominic*, *Wright Chris* ARM System Developer's Guide. Designing and Optimizing System Software. — Morgan Kaufmann, 2004. — ISBN: 1-55860-874-5.

21. *Smith Tony* ARMv8 Tools – Everything You Need To Develop for AArch64. — Окт. 2012. — URL: http://blogs.arm.com/software-enablement/827-armv8-tools-everything-you-need-to-develop-for-aarch64/ (дата обр. 28.12.2012).

22. System V Application Binary Interface. AMD64 Architecture Processor Supplement. — AMD Corporation. — URL: http://www.x86-64.org/documentation/abi.pdf (дата обр. 14.02.2012).

23. The 68LC040 Emulator. — Apple Computer Inc., 1996. — URL: http://developer.apple.com/legacy/mac/library/documentation/mac/PPCSoftware/PPCSoftware-13.html (дата обр. 09.09.2012).

24. VirtualBox architecture. — Oracle Corporation. — URL: http://www.virtualbox.org/wiki/VirtualBox_architecture (дата обр. 25.09.2010).

25. VMware ESXi info page. — VMWare. — URL: http://www.vmware.com/products/vsphere/esxi-and-esx/index.html.

26. *Weaver D.L.*, *Germond T.*, *International SPARC* The SPARC architecture manual: version 9. — PTR Prentice Hall, 1994. — ISBN: 9780130992277. — URL: http://books.google.ru/books?id=JNVQAAAAMAAJ.

27. Режим Windows XP. — Microsoft Corporation. — URL: http://windows.microsoft.com/ru-RU/windows7/products/features/windows-xp-mode (дата обр. 09.09.2012).

2. Модели процессора на основе интерпретации

Понять — значит построить модель.

Уильям Томсон (лорд Кельвин)

«Интерпретатор» в общем значении слова — тот, кто занимается переводом текста с одного языка на другой. В контексте вычислительной техники этот термин противопоставляется трансляторам и компиляторам; последние два понятия описывают программы, преобразующие тексты на входном языке (машинном или высокого уровня) в новое представление, оперируя при этом достаточно большими его блоками — файлами, модулями, функциями и т.п. Интерпретатор же ограничивается работой над одной «строкой» (например, машинной инструкцией) входного языка. Следующая строка будет преобразована (*проинтерпретирована*) тогда, когда в этом возникнет необходимость.

2.1. Архитектурное состояние

Прежде чем перейти к описанию общей картины алгоритма, рассмотрим, какие структуры данных используются для представления состояния процессора в функциональной модели[1].

В любом классическом процессорном устройстве всегда присутствует регистр, хранящий адрес текущей исполняемой инструкции. Например, в архитектуре IA-32 [1] для этого используется семейство xIP: `IP`, `EIP`, `RIP`, в архитектуре ARM [5] он имеет название `pc`, в других системах он может называться по-другому, например, `IC` (*англ.* instruction counter). В дальнейшем для единообразия мы будем использовать обозначение `PC` (*англ.* program counter).

Кроме указателя инструкций, процессоры содержат множество других регистров, типы, назначение и параметры которых зависят от

[1]Более детально об архитектурном состоянии рассказывается в главе 8.

модели. В большинстве случаев присутствуют регистры общего назначения (*англ.* general purpose registers, GPR), используемые в арифметических операциях и при адресации памяти.

В языке Си (и C++) описание состояния может быть представлено структурой `state_t`, содержащей поля для всех регистров, а также ссылки на внешние устройства:

```
typedef uint32_t register_t; // ширина гостевых регистров
const int n_regs = 16; // число регистров
typedef struct {
    register_t pc; // счётчик инструкций
    register_t gpr[n_regs]; // регистры общего назначения
    uint8_t *memory; // указатель на ОЗУ
} state_t;
```

Заметим, что данное описание очень далеко от того, чтобы быть полным, однако оно даёт базовое представление того, с чем приходится иметь дело в начале разработки новой модели.

2.2. Стадии работы

Алгоритм работы в общих чертах напоминает стадии конвейера исполнения команд в настоящем процессоре[1] (рис. 2.1).

1. Извлечение (*англ.* fetch) кода инструкции из памяти по адресу, вычисляемому из значения `PC`. Конкретная формула зависит от деталей архитектуры и текущего режима процессора.
   В модели это действие идентично операции чтения из памяти и может вызывать соответствующие побочные эффекты.
2. Задача декодирования (*англ.* decode) состоит в том, чтобы по числу, полученному в предыдущей фазе, определить, какую операцию следует выполнить и какие аргументы в ней будут участвовать. Например, число `0x706a` в архитектуре IA-32 обозначает команду `PUSH 0x70` — поместить в стек число 0x70.
   Алгоритм и сложность декодирования сильно зависят от сложности самого языка инструкций целевой машины. Как правило,

[1] Отметим, что число стадий может быть различно у разных моделей и варьируется от трёх до двадцати и более.

Рис. 2.1. Рабочий цикл интерпретатора

процесс состоит из поиска и сопоставления битовых полей считанного машинного слова со значениями из заранее созданных таблиц. В силу многих факторов (например, переменной длины инструкций, различного смысла значений в различных режимах процессора, использования префиксов и т.п.) оно может занимать существенную часть времени работы интерпретатора. Мы рассмотрим декодирование подробнее в секции 2.4.

3. Исполнение (*англ.* execute) состоит из непосредственной симуляции функции только что декодированной инструкции. Как правило, это вычисление результата арифметической или логической операции, изменение режима модели процессора или передача контроля управления в другую секцию алгоритма.
   В модели каждому коду машинной операции (*опкоду*) должна соответствовать моделирующая процедура. Выбор нужной процедуры производится по опкоду:

```
switch (opcode) {
    case OPERATION1: ...
    case OPERATION2: ...
```

```
    ...
    default: ... // unknown command
}
```

4. Запись результата (*англ.* write back) операции в архитектурные регистры. Часть результатов также может быть расположена в оперативной памяти. Как и при её чтении (на этапе извлечения кода инструкции или получения входных операндов), при записи модель должна симулировать все побочные эффекты.
5. Продвижение указателя команд (*англ.* advance PC) на значение, соответствующее следующей инструкции. Т.к. большая часть существующих алгоритмов состоит из линейных участков, изредка прерываемых операциями ветвления, к нему прибавляется длина только что завершённой машинной команды.
   При моделировании необходимо учитывать ограниченность ширины регистра PC и возможность его переполнения.

```
const int instr_size = 2; // 16 bit CPU
const int addr_mask = 0xffff; // mask overflowed bits
state_t processor; // our CPU
...
processor.pc = (processor.pc + instr_size) & addr_mask;
```

2.3. Исключения и прерывания

Часто при обработке текущей инструкции возникает ситуация, когда нормальное её выполнение не может быть завершено, потому что были обнаружены недопустимые условия на входные операнды (например, целочисленное деление на ноль или недоступность памяти), или возникло какое-то внешнее условие, требующее немедленной обработки. При этом архитектурное состояние процессора изменяется определённым способом (как правило, управление передаётся на обработчик возникшей ситуации), в том числе регистр PC начинает указывать на новый участок кода.

На рис. 2.2 изображён цикл интерпретации, учитывающий тот факт, что практически в любой момент может произойти переход в состояние обработки исключительной ситуации, вносящее изменения

в архитектурное состояние, после чего цикл интерпретации начинается заново уже с новым PC.

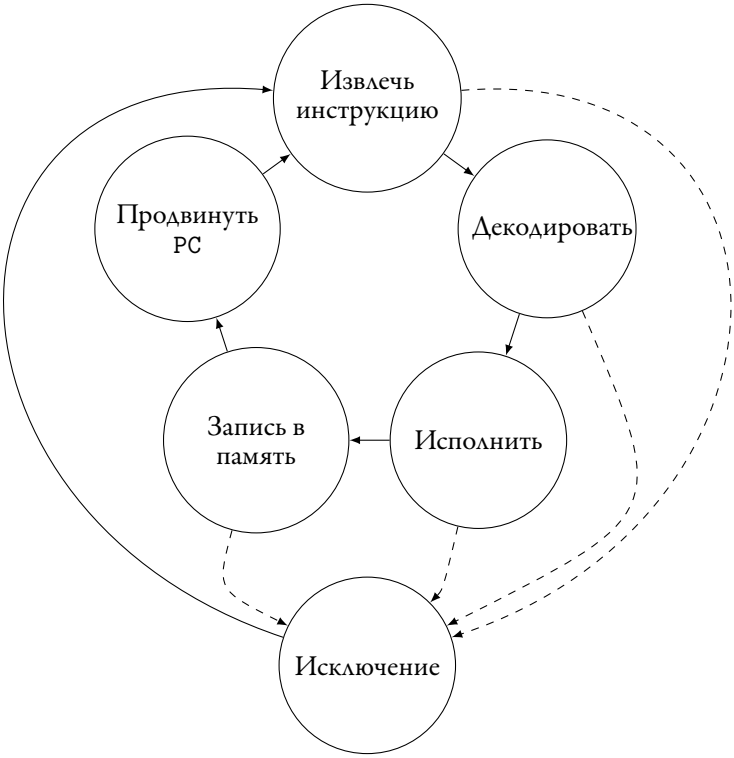


Рис. 2.2. Рабочий цикл интерпретатора. Показана возможность возникновения исключительной ситуации на любой стадии симуляции

2.3.1. Классификация

В документациях к различным процессорам [2, 5, 7] даны различные, зачастую внутренне противоречивые определения терминов, связанных с исключительными ситуациями. Тем не менее важно различать природу событий, их связь с текущим контекстом выполнения для того, чтобы корректно симулировать их эффекты.

Выделим основные группы исключительных событий по признакам наличия причинной связи между событиями, влиянием среды исполнения на возможность их возникновения, а также адресом возврата после их обработки. В скобках к терминам будут даны те имена, под какими они чаще всего встречаются в литературе; однако не следует полагаться на строгость данных соответствий. См. также замечания ниже.

– *Синхронные с повторением текущей инструкции* (промах, *англ.* fault) — событие, связанное причинно-следственно с выполнением текущей инструкции и обусловленное «неготовностью» среды исполнения к её успешному завершению. Примеры таких ситуаций: отсутствие физической страницы памяти с необходимыми данными; неготовность сопроцессора выполнять работу, т.к. он требует дополнительной инициализации. В этих случаях обработчик ситуации, находящийся в операционной системе, может модифицировать среду исполнения так, что завершение инструкции станет возможным, например, загрузить нужную страницу или включить сопроцессор. Дальнейшее возвращение на *тот же* PC с перезапуском инструкции позволит устранить проблему прозрачно для пользовательского приложения.
– *Синхронные без повторения текущей инструкции* (исключения, *англ.* exception). Как и в предыдущем случае, событие порождено текущей инструкцией. Однако его обработка не подразумевает её повтора, так как причина события неустранима и не связана со средой исполнения, но связана только с самой операцией и операндами. Примеры: инструкция целочисленного деления на регистр, содержащий ноль, всегда будет давать ошибку; инструкция, запрещённая к выполнению в текущем уровне привилегий, не может быть на нём исполнена никогда. Чаще всего (но не всегда!) исключения обозначают ошибку в программе. Управление после возврата из обработчика будет передано в место, не связанное с PC того кода, где произошло событие.
– *Ловушки* (*англ.* trap) также синхронны. При этом они обозначают явное «желание» программы быть прерванной и передать управление в определённую область кода — обработчик вызова.

Примером является инструкция SYSCALL, вызывающая системные функции операционной системы, такие как работа с файлами, создание новых процессов и т.п. Другой пример — команда, предназначенная устройству-сопроцессору, физически отсутствующему в системе, однако ОС умеет её эмулировать и таким образом способна вернуть правильный результат прозрачно для пользовательской задачи. С точки зрения прикладного ПО ловушка — это инструкция, семантика которой определяется не спецификацией ЦПУ, а используемой операционной системой и средой исполнения.

После обработки ловушки и возврата счётчик инструкций будет указывать на следующую команду в потоке исполнения, т.е. будет соответствовать нормальному потоку. Отметим, что разница между ловушками и промахами минимальна и их классификация зависит от верхнеуровневого смысла, который вкладывают в соответствующую подпрограмму-обработчик.

– *Асинхронные прерывания* (*англ.* interrupt). В отличие от всех ранее рассмотренных событий, они вызваны причинами, внешними по отношению к текущему контексту исполнения, и означают некоторое состояние внешней среды, требующее внеочередной обработки. Примеры: жёсткий диск готов передать новую порцию данных, ранее запрошенную независимым процессом; температурный датчик сигнализирует о превышении измеряемой температуры порогового значения; таймер сообщил о прошествии запрограммированного в него интервала. Прерывание никак не связано с опкодом, адресом или аргументами инструкций — оно могло произойти чуть раньше или позже, а могло и вовсе не произойти. Однако игнорировать его в общем случае нельзя. Часто недопустимо откладывать вызов обработчика во времени дольше, чем на некоторый краткий период времени.

Будет ли после возвращения из обработчика перезапущена инструкция, на которой возникло прерывание, зависит от конкретной архитектуры процессора, однако в любом случае её исполнение должно пройти таким образом, чтобы сам факт обработки был скрыт от прерванного приложения.

Следует также отметить следующие особенности существующих центральных процессоров.

1. Программное прерывание (*англ.* software interrupt) — событие, вызываемое специальной инструкцией (например, в IA-32 это `INT`), обработка которого напоминает вызов процедуры. Т.е., несмотря на название, оно соответствует ловушке, а не прерыванию.
2. В некоторых архитектурах, например SPARC [7], подпрограмма-обработчик синхронного события может сама выбрать, следует ли перезапускать текущую инструкцию. Для возвращения из подпрограммы-обработчика могут использоваться две различные инструкции — `RETRY` для перезапуска (в случае обработки промаха) и `DONE` для исполнения следующей команды за текущей (для выхода из обработки ловушек). Для поддержки такой возможности в архитектуру введён регистр `nPC`, в любой момент указывающий на следующую за текущей инструкцию.

2.3.2. Обработка исключительных ситуаций при симуляции

Существование исключений и прерываний (а они отсутствуют разве что только в узкоспециализированных микроконтроллерах) существенно усложняет логику как аппаратной системы, так и моделирующей среды. Непредсказуемость их возникновения создаёт множество веток исполнения в структурированном коде, усложняя его структуру, а также негативно влияя на скорость исполнения.

Обычный структурированный код процедурных и объектно-ориентированных языков высокого уровня состоит из вложенных вызовов процедур (методов), каждая из которых по окончании работы возвращает управление в вызвавшую процедуру по адресу, сохранённому на стеке. Однако моделирование исключительных ситуаций подразумевает возможность их возникновения в множестве мест — индивидуальных блоках эмуляции инструкций. При этом после изменения архитектурного состояния управление должно быть передано на начало следующего цикла интерпретации.

Забегая вперёд, заметим, что особенно остро эта проблема передачи управления встаёт не в интерпретаторах, а в двоичных трансляторах, часть кода которых создаётся динамически. При этом часто при исполнении этого кода заранее нельзя сказать, какова будет структура стека в момент обнаружения исключительной ситуации, и его развёртывание до необходимого уровня вложенности с помощью серии обычных возвратов из процедур будет достаточно дорогостоящей операцией, нивелирующей преимущества быстрого исполнения.

Естественным способом передачи управления в такой ситуации является нелокальный «прыжок» — переход, использующий пару функций `setjmp()` и `longjump()`, описанных в стандарте библиотеки Си.

Функция `setjmp` сохраняет контекст в переменной `env` и возвращает 0, если выход из неё был после её прямого вызова. Если произошёл возврат из `longjmp`, то функция возвращает ненулевое значение.

Функция `longjmp` возвращает выполнение в точку вызова `setjmp` со значением `val`. При этом все объекты с неавтоматическим выделением памяти сохраняют своё значение.

Пример использования `setjmp()` и `longjmp()`[1]:

```
#include <stdio.h>
#include <setjmp.h>

static jmp_buf buf;
void second(void) {
  printf(''second\n'');/* печать на экран */
  longjmp(buf,1);/* переходит по метке buf и возвращает код 1*/
}

void first(void) {
  second();
  printf(''first\n'');/* этой печати не произойдёт*/
}

int main() {
  if ( ! setjmp(buf) ) {
    first(); /* при исполнении вернёт код 0*/
  } else { /* по возвращении из longjmp вернёт 1*/
```

[1]Пример взят из Википедии: `http://en.wikipedia.org/wiki/Setjmp.h`.

```
    printf(''main\n'');/* печать на экран*/
  }
  return 0;
}
```

Нельзя отрицать, что использование как нелокальных[1] переходов с помощью `longjump`, так и локальных[2] переходов по метке с помощью оператора `goto` языка Си нарушает модульность кода и лёгкость его чтения, а также может быть источником алгоритмических ошибок. Однако на это приходится идти ради увеличения скорости работы приложения.

2.4. Реализация декодера

Теория вопросов разбора и лексического анализа выражений хорошо разработана для языков высокого уровня и описывается во всех книгах, посвящённых задаче построения компиляторов [6]. Считаемая классической «Книга дракона» [8] также подробно рассматривает вопрос разбора выражений.

2.4.1. Особенности разбора машинных языков

Машинное представление инструкций некоторой системы — всего лишь один из языков, и вся теория разбора выражений к нему применима. Однако, имеется ряд особенностей, позволяющих в ряде практически важных случаев строить более простые декодеры для машинного кода.

Переменная или постоянная длина инструкций. Многие RISC-процессоры имеют фиксированную длину инструкций, например, 16 или 32 бита. При этом адрес всех инструкций всегда выровнен. Для однозначного декодирования достаточно считать из памяти одно машинное слово.

[1] Т.е. пересекающих границу отдельной процедуры.
[2] Внутри одной процедуры.

С другой стороны, более древние CISC-системы чаще всего используют переменную длину инструкций. Так, в архитектуре IA-32 длина инструкций может составлять от 1 до 15 байт.

Префиксный код. В подавляющем числе случаев набор инструкций определяется *префиксным кодом* — никакая последовательность бит, определяющая разрешённую инструкцию, не является точным префиксом для другой инструкции. Это свойство означает отсутствие неоднозначности при декодировании инструкций с переменной длиной.

Влияние режима процессора на смысл. В значительной части архитектур процессор может находится в нескольких режимах работы, определяющих его функциональность. Например, процессоры IA-32 могут иметь следующие режимы работы[1]: 16-битный реальный, 16-битный «нереальный», 16-битный защищённый, 32-битный защищённый, 64-битный защищённый. Процессор ARM может быть в режиме 32-битных команд, 16-битных Thumb-команд, а также в недавно появившемся 64-битном режиме, доступным для некоторых моделей. При этом кодировка команд разных режимов может быть несовместима. Так, в архитектуре IA-32 в 64-битном режиме однобайтные последовательности из диапазона 0x40–0x4f не являются полными инструкциями, а представляют собой части более длинных команд. Во всех остальных режимах им соответствуют варианты инструкции `DEC`. Поэтому при декодировании необходимо учитывать режим.

Странности Иногда ISA может иметь совершенно неожиданные особенности. Например, в Intel Itanium ширина группы из трёх инструкций, составляющих связку(*англ.* bundle), почти всегда равна 128 битам. При этом ширина каждой отдельной инструкции равна 41 биту, а пять оставшихся бит несут общую информацию о группе в целом (т.н. шаблон). *Для некоторых* шаблонов ширина одной из инструкций удваивается до 82 бит, таким образом, в связке остаётся лишь две инструкции!

[1]Для краткости описания здесь опущены такие режимы, как System Management Mode и VMX root/non-root.

2.4.2. Ввод и вывод процедуры декодера

В реальном процессоре за задачу декодирования отвечает отдельный блок логических элементов микросхемы. В симуляторе ему соответствует некторая процедура, написанная на языке программирования. Рассмотрим, что подаётся на её вход и какие результаты она должна выдавать.

Как должно быть понятно из описанного выше, на вход декодера подаётся массив байт известной длины, полученный на фазе Fetch. Кроме того, ему может быть известен текущий режим процессора и адрес начала массива в памяти гостя.

В результате работы декодер должен вернуть код ошибки и результаты анализа последовательности в виде списка полей результата (мы вернёмся к ним чуть позже). При этом возможны следующие значения для кода ошибки.

1. Декодирование успешно (код 0). Массив байт был распознан как допустимая инструкция, и список полей содержит информацию о коде операции и её аргументах.
2. Декодирование неуспешно (код 1). Ни одна инструкция, определённая в архитектуре, не соответствует входному массиву байт. При этом содержимое полей результата не несёт смысла. Что происходит в этой ситуации дальше на этапе исполнения? Это зависит от архитектуры. Чаще всего невозможность декодировать ведёт к генерации исключения[1]. В некоторых случаях некорректная инструкция может быть воспринята как `NOP` — отсутствие операции.
3. Для ISA с переменной длиной инструкций возможна третья ситуация — входных данных недостаточно для принятия однозначного решения (код -1). Другими словами, на вход декодера передали только часть инструкции, и он, не имея информации о том, какие байты идут в памяти дальше, сообщает об этом.

[1]Подчеркнём, что эта ситуация не является внутренней ошибкой самого симулятора — поведение процессора на неизвестных инструкциях должно быть описано в документации и является штатной ситуацией в его работе.

На рис. 2.3 приведён пример алгоритма, сочетающего в себе итерации фаз Fetch и Decode и позволяющего провести декодирование для инструкций с переменной длиной.

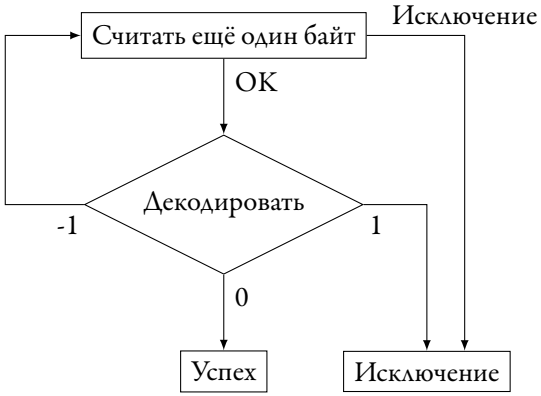


Рис. 2.3. Блок-схема декодирования, учитывающая переменную длину инструкции

У наблюдательного читателя может появиться вопрос: зачем использовать этот достаточно сложный и наверняка неэффективный алгоритм? Поскольку размер самой длинной инструкции всегда известен, а используемый код префиксный, то можно сделать Fetch последовательности, достаточной для вмещения как минимум одной инструкции. Затем декодировать её первый префикс, а оставшиеся «лишние» байты проигнорировать.

К сожалению, этот метод может генерировать исключения, отсутствующие в реальной системе, при попытке декодирования инструкций, находящихся близко к концу страницы или сегмента симулируемой памяти. Это связано с тем, что в системах, использующих механизмы страничной адресации или сегментации, разные диапазоны памяти имеют разные свойства. Если при чтении массива для декодирования «с запасом» пересекается граница между двумя такими диапазонами, и второй из них при чтении вызывает исключение, то и всё декодирование будет давать ложное исключение, тогда как на самом деле текущая инструкция корректна, не пересекает границ и должна

быть успешно распознана (рис. 2.4). Для того, чтобы избежать подобной ситуации, в алгоритме рис. 2.3 на каждой итерации читается и добавляется только один байт.

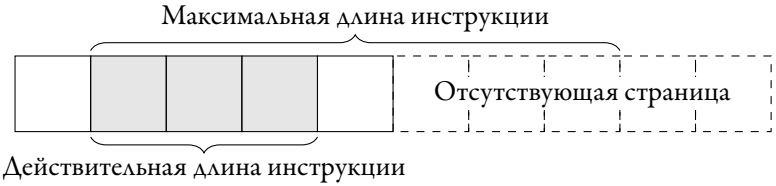


Рис. 2.4. Пересечение границы страниц при декодировании, вызывающее ложное исключение

2.4.3. Поля результата

Какую информацию должны содержать поля результата при успешном декодировании?

- Код операции (опкод), определяющий функцию, выполняемую инструкцией.
- Длину инструкции для ISA, в которых она является переменной.
- Информацию о каждом операнде. Она может включать в себя: порядковый номер регистра, его ширину, тип; для операций обращения к памяти — адрес и его ширина.
- Дополнительная информация, влияющая на исполнение. К ней может относиться наличие префиксов, модифицирующих операцию или размеры операндов и т.п.

Если сохранять результат в виде структуры языка Си, то она будет иметь следующий тип:

```
typedef struct decode_result {
    int length; // длина инструкции
    opcode_t opcode; // код операции
    int num_operands; // число операндов инструкции
    struct {
```

```
        operand_type_t type; // тип операнда
        union {
            int32    i32;
            int16    i16;
            int8     i8;
            float    f32;
            offset_t off;
        } value; // варианты хранимого значения
    } operands[MAX_OPERANDS]; // массив с операндами
} decode_result_t;
```

Для каждой конкретной архитектуры поля данной структуры будут свои собственные, отражающие особенности её формата инструкций.

2.4.4. Декодирование как распознавание шаблонов

В документации на центральные процессоры формат инструкций чаще всего описывается в виде таблиц, определяющих, какие биты машинного представления инструкции определяют логические поля, такие как опкод и операнды (см. рис. 2.5). Группы инструкций могут описываться одним и тем же форматом. Верно и обратное — несколько форматов могут описывать варианты одной и той же инструкции. При этом полное число различных форматов зависит от самой ISA и может быть достаточно велико.

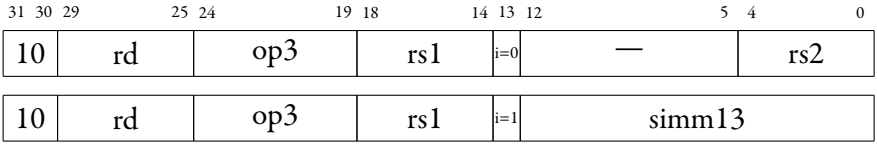


Рис. 2.5. Описание битовых полей инструкции. Пример взят из описания архитектуры SPARC [7], инструкция ADD и её варианты

Задача декодера состоит в сопоставлении входной строки данному набору шаблонов, нахождении совпадения, вычислении значений отдельных битовых полей и формировании значений полей логических. Для корректного декодирования любой входной последовательности должен соответствовать максимум один шаблон. Если же совпадений больше, то либо в кодировке инструкций, либо в реализации

декодера присутствует ошибка.

Связь битовых и логических полей инструкции

Каждое логическое поле инструкции, такое как номер регистра, может зависеть от нескольких битовых полей, а также режима процессора. Например, в последних поколениях IA-32 номер одного из векторных регистров ZMM (шириной 5 бит) в 64-битном режиме процессора определяется как комбинация следующих битовых полей : 3 бита `ModRM.Reg`, 1 бит `REX.R` и 1 бит $\texttt{EVEX.}\bar{\texttt{R}}$, причём последний из них следует инвертировать.

2.5. Увеличение скорости работы интерпретатора

Главным преимуществом рассмотренной схемы является её простота в реализации, модификации и отладке. Практически всегда в проектах по созданию программной модели нового процессора первым этапом является разработка интерпретационной модели, которая затем используется как эталон для тестирования последующих улучшений модели, оптимизирующих эффективность исполнения.

Основным недостатком интерпретатора является низкая скорость. Были разработаны многочисленные приёмы увеличения скорости интерпретации. Рассмотрим базовые идеи, используемые в них.

2.5.1. Сцепленная интерпретация

Одной из причин низкой скорости работы является неэффективное использование различных аппаратных ресурсов хозяйской системы, призванных уменьшить влияние явлений, разрушительных для конвейерной обработки. Так, из-за использования единого `switch` в теле цикла, из которого передача управления может быть осуществлена во множество мест, предсказатель переходов процессора не может каждый раз правильно предугадать адрес инструкции перехода, что вызывает сброс конвейера и задержку в несколько тактов. Этот негативный эффект проявляется в начале обработки каждой новой гостевой инструкции. Вместо концентрации условного перехода в одном

месте желательно «размазать» его по многим местам в коде, уменьшив в каждом из них число вариантов адреса (в идеале — до одного). Этого можно достичь, если вызывать обработчик следующей инструкции сразу после конца работы текущей инструкции, без возвращения в общий цикл. Такой алгоритм интерпретации называется *сцепленным* (*англ.* threaded), см. рис. 2.6.

TODO Две блок-схемы

Рис. 2.6. Сравнение методов переключаемой и сцепленной интерпретаций

Пример реализации сцепленной интерпретации в псевдокоде дан ниже. Предполагается, что этап декодирования уже проведён, и в памяти содержится информация о том, какой будет следующая инструкция.

```
// Массив labels содержит адреса переходов для всех обработчиков
labels = [INSTR_A, INSTRb, ... INSTR_X, ... INSTR_Y ...];

INSTR_X: // Текущая инструкция X
    X_handler(operands, PC); // обработчик инструкции
    PC++;
    goto label[PC]; // Сразу к обработчику новой инструкции
```

Для реализации сцепленной схемы используемый для написания модели язык должен поддерживать указатели на метки в коде. Стандарт ANSI Си не позволяет этого делать. Но, например, в GNU C доступно соответсвующее расширение языка.

2.5.2. Интерпретация с кэшированием

Промежуточным звеном между интерпретатором и транслятором является кэширующий интерпретатор. В нём вместо достаточно медленной отдельной фазы генерации кода используется только кэш (промежуточное хранилище с быстрым доступом) декодированных инструкций (рис. 2.7). Если он реализован эффективно, то решение будет сбалансировано: при исполнении зацикленного кода модель ЦПУ будет достаточно быстрой, а при исполнении линейного ко-

да будет незначительно проигрывать простому интерпретатору, рассмотренному ранее.

2.6. Модификация интерпретатора — добавление новых инструкций

Часто возникает задача расширения функциональности некоторой модели для представления функциональности нового процессора, отличающегося от старого наличием новых инструкций и дополнительных регистров процессора. Например, начиная с Intel Pentium IV в 2001 году были введены команды семейства SSE2, работающие с регистрами `XMM0–XMM7`.

Для того чтобы минимально модифицировать старый, хорошо отлаженный код модели, но при этом и поддержать новые системы, можно воспользоваться тем обстоятельством, что оригинальная модель не распознаёт новые инструкции как допустимые и должна вызвать обработку исключения `#UD` (*англ.* undefined opcode). Однако, мы даём модели «второй шанс», вызывая второй декодер новых инструкций. Если он подтверждает, что может декодировать переданный ему машинный код, вызывается новая часть интерпретатора, ответственная за новый набор инструкций (рис. 2.8).

Очевидно, что данную схему можно расширить для каскадного включения большего числа новых наборов инструкций. Её достоинство — гибкость подключения новой функциональности к уже существующей модели; дополнительные декодеры и симуляторы инструкций могут быть взяты из независимых источников и сравнительно легко адаптированы для использования. Недостаток тоже очевиден: последовательный вызов декодеров менее быстр, чем реализация, объединяющая их все в единую сущность.

2.7. Простой пример

Приведем код (на языке Си) интерпретационной модели некоторого упрощённого для целей данного примера процессора со следу-

ющей архитектурой.

2.7.1. Регистры

В рассматриваемом примере три регистра для арифметических операций и один указатель текущей инструкции.

- `R0` — регистр общего назначения
- `R1` — регистр общего назначения
- `R2` — регистр общего назначения
- `IP` — указатель команд

2.7.2. Команды

Набор инструкций включает в себя только две арифметические операции, а также работу с памятью.

- `ADD` — сложение, можно прибавлять к регистру регистр или число
- `SUB` — вычитание, можно вычитать из регистра число
- `LOAD` — загрузка ячейки памяти в регистр
- `STORE` — сохранение регистра в памяти

2.7.3. Код модели

```
struct DecodedInstr {
    enum Operation Op;
    enum Argument Arg1;
    enum Argument Arg2;
};



int R0, R1, R2, IP;  // Модель регистров
class Memory Mem;    // Модель внешней памяти
```

```
for (;;) { // бесконечный цикл
    int Instr = FetchInstr();
    struct DecodedInstr DecInstr = Decode(Instr);
    Execute(DecInstr);
}

int FetchInstr() {
    return Mem.Load32Bits(IP); // Загружаем 4 байта из памяти
    по адресу PC
}

struct DecodedInstr Decode(int Instr) {
    switch (Instr) { // Перебираем все реализованные инструкции
    case 0: // ADD R0, R0
        return {.Op = OP_ADD, .Arg1 = ARG_R0, .Arg2 = ARG_R0
    };
    case 1: // ADD R0,R1
        return {.Op = OP_ADD, .Arg1 = ARG_R0, .Arg2 = ARG_R1
    };
    // ...
    }
}

void Execute(struct DecodedInstr DecInstr) {
    int *Arg1, *Arg2;  // Указатели на аргументы операции
    // Какой первый аргумент операции?
    switch (DecInstr.Arg1) {
        case ARG_R0: Arg1 = &R0; break;
        case ARG_R1: Arg1 = &R1; break;
        case ARG_R2: Arg1 = &R2; break;
    }
    // Какой второй аргумент операции?
    switch (DecInstr.Arg2) {
        case ARG_R0: Arg2 = &R0; break;
        case ARG_R1: Arg2 = &R1; break;
        case ARG_R2: Arg2 = &R2; break;
    }
    // Выполнить операцию
    switch (DecInstr.Op) {
    case OP_ADD:
        *Arg1 += *Arg2;
        IP += 4; // Продвинуть указатель команд на следующую
    инструкцию
```

```
            break;
        case OP_SUB:
            *Arg1 -= *Arg2;
            IP += 4;
            break;
        case OP_LOAD:
            *Arg1= Mem.Load32Bits(*Arg2);
            IP += 4;
            break;
        // ...
        }
}
```

2.8. Заключительные замечания

Проект Bochs [3] является хорошим примером зрелого интерпретатора, содержащего сложную модель процессора для существующей архитектуры IA-32. В технических заметках к программе [4] её авторы описывают множество полезных приёмов, применимых как к организации модели-интерпретатора для процессора любой архитектуры, так и специфичных для архитектуры IA-32, являющейся одной из сложнейших в реализации.

2.9. Вопросы к главе 2

Вариант 1

1. Какие из указанных ниже компонентов обязательны для реализации интерпретатора:
    a) декодер,
    b) дизассемблер,

c) кодировщик (енкодер),
d) блоки реализации семантики отдельных инструкций,
e) кэш декодированных инструкций.

2. Опишите, что происходит на стадии Fetch работы процессора.
3. Опишите, что происходит на стадии **Writeback** работы процессора. Для каких инструкций эта стадия будет опущена?
4. Какой вид программ обычно исполняется в привилегированном режиме процессора?
5. Какие эффекты могут наблюдаться при невыровненном (unaligned) чтении из памяти в существующих архитектурах:

   a) возникновение исключения,
   b) замедление операции по сравнению с аналогичной выровненной,
   c) данные будут считаны лишь частично,
   d) возможны все перечисленные выше ситуации?

6. Какая из следующих типов ситуаций при исполнении процессора является асинхронной по отношению к работе текущей инструкции?

   a) прерывание (interrupt),
   b) ловушка (trap),
   c) исключение (exception),
   d) промах (fault)?

7. Выберите правильный вариант окончания фразы: Сцепленный интерпретатор работает быстрее переключаемого (switched), так как

   a) удачно использует предсказатель переходов хозяйского процессора,
   b) кэширует недавно исполненные инструкции,
   c) транслирует код в промежуточное представление,
   d) не требует обработки исключений.

Вариант 2

1. Какой из типов регистров всегда присутствует во всех классических архитектурах:
    a) указатель стека,
    b) аккумулятор,
    c) указатель текущей инструкции,
    d) регистр флагов,
    e) индексный регистр.
2. Опишите, что происходит на стадии **Decode** работы процессора.
3. Опишите, что происходит на стадии **Advance PC** работы процессора. Для каких инструкций эта стадия будет опущена?
4. Какой вид программ обычно исполняется в непривилегированном режиме процессора?
5. Почему самый простой вид декодера машинных инструкций — однотабличный — не пользуется большой популярностью?
6. Выберите правильные варианты окончания фразы: Наличие единственного `switch` для всех гостевых инструкций в коде интерпретатора
    a) увеличивает его скорость по сравнению со схемой сцепленной интерпретации,
    b) упрощает его алгоритмическую структуру по сравнению со схемой сцепленной интерпретации,
    c) уменьшает его скорость по сравнению со схемой сцепленной интерпретации,
    d) не влияет на скорость работы интерпретатора.
7. Почему редко представляется возможным при симуляции процессора разместить все гостевые регистры на физических регистрах?

Рис. 2.7. Схема работы кэширующего интерпретатора

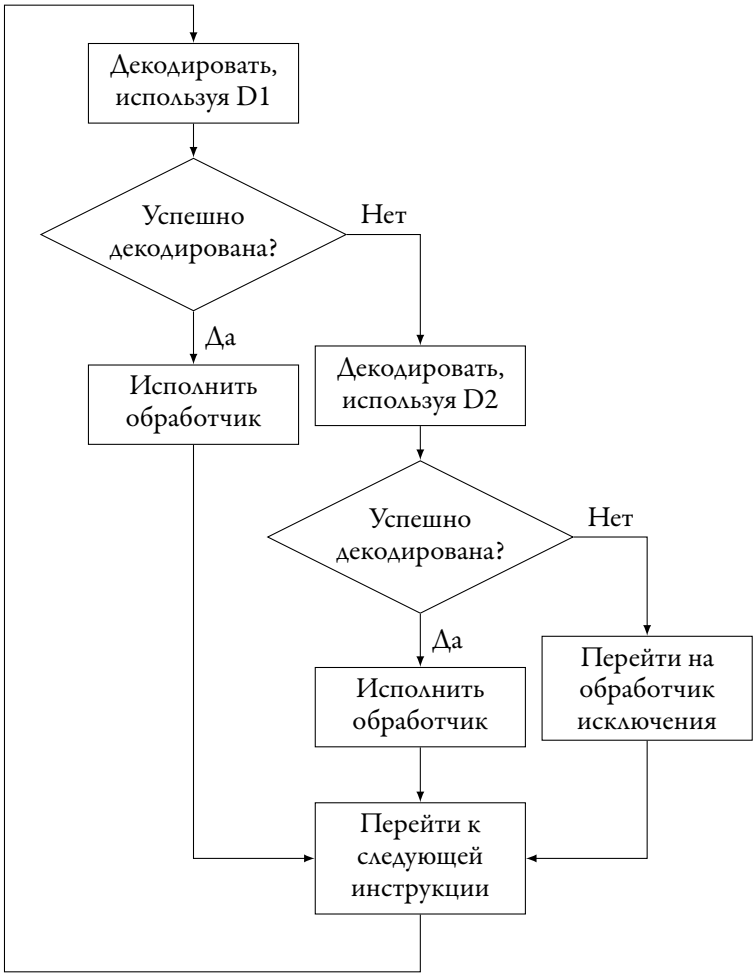


Рис. 2.8. Ступенчатая схема вызова декодеров при обнаружении инструкции, не поддерживаемой оригинальной моделью. При обнаружении в потоке инструкций машинного кода, не распознаваемого D1, управление передаётся на D2

Литература

1. Intel® 64 and IA-32 Architectures Software Developer's Manual. Volume 2A. — Intel Corporation.
2. Intel® 64 and IA-32 Architectures Software Developer's Manual. Volumes 1–3. — Intel Corporation. 2012. — URL: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html (дата обр. 25.06.2012).
3. *Mihoka Darek*, *Shwartsman Stanislav* Virtualization Without Direct Execution or Jitting: Designing a Portable Virtual Machine Infrastructure // ISCA-35 Proceedings of the 1st Workshop on Architectural and Microarchitectural Support for Binary Translation. — URL: http://bochs.sourceforge.net/Virtualization_Without_Hardware_Final.pdf (дата обр. 05.05.2012).
4. *Shwartsman Stanislav*, *Mihoka Darek* How Bochs Works Under the Hood. 2nd edition. — Тех. отч. — 2012. — URL: http://bochs.sourceforge.net/HowtheBochsworksunderthehood2ndedition.pdf (дата обр. 12.07.2012).
5. *Sloss Andrew N.*, *Symes Dominic*, *Wright Chris* ARM System Developer's Guide. Designing and Optimizing System Software. — Morgan Kaufmann, 2004. — ISBN: 1-55860-874-5.
6. *Torczon Linda*, *Cooper Keith* Engineering A Compiler. — 2nd. — San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. — ISBN: 012088478X.
7. *Weaver D.L.*, *Germond T.*, *International SPARC* The SPARC architecture manual: version 9. — PTR Prentice Hall, 1994. — ISBN: 9780130992277. — URL: http://books.google.ru/books?id=JNVQAAAAMAAJ.

8. 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.

3. 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.

3.1. Двоичная трансляция

Как было показано в предыдущей главе, симулирование исполнения процессора через интерпретацию обладает как положительными качествами, такими как простота разработки и модификации, так и 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.

3.1.1. 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. 3.1 показано, как связаны части исходного кода гостевой программы и результат трансляции, состоящий из хозяйских инструкций.

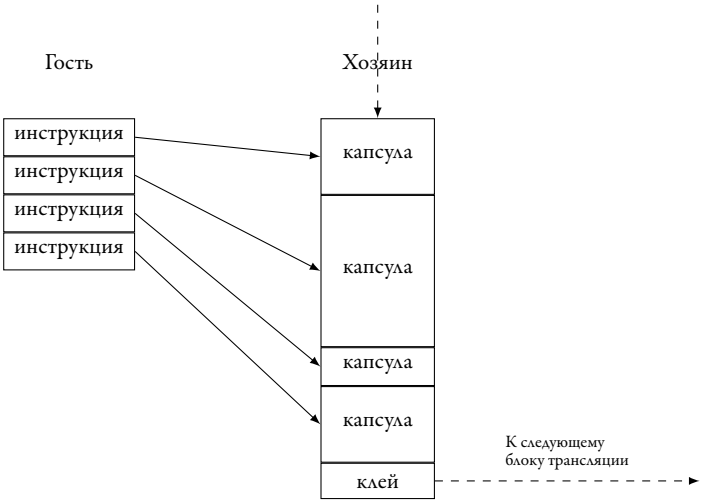


Рис. 3.1. Исходный код базового блока приложения и его связь с результатом двоичной трансляции. Штриховыми линиями показаны точки входа и выхода

3.1.2. Пример преобразования одной инструкции

На рис. 3.2 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)).

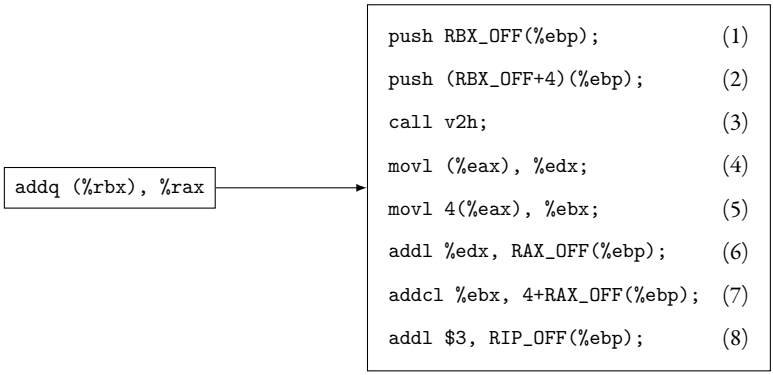


Рис. 3.2. Пример соответствия гостевой инструкции и хозяйской капсулы, эмулирующей её семантику и написанной на языке ассемблера. В этом примере хозяйский регистр EBP хранит указатель на структуру гостевого состояния, макросы вида RxX_OFF — смещение внутри гостевого состояния для регистра RxX, а v2h — 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 8.

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.

3.1.3. 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 3.5 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 [7], 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 [5]. Библиотека SoftFloat [6] 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 [10]. 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.

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 12.

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 максимально возможное число целевых регистров на хозяйских, чтобы при об-

ращении 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 8, 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 прочитать максимум 32 бита, тогда как в гостевой архитектуре требуе-

мый 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».

3.1.4. 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 [2], 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 3.3).

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. 3.3).

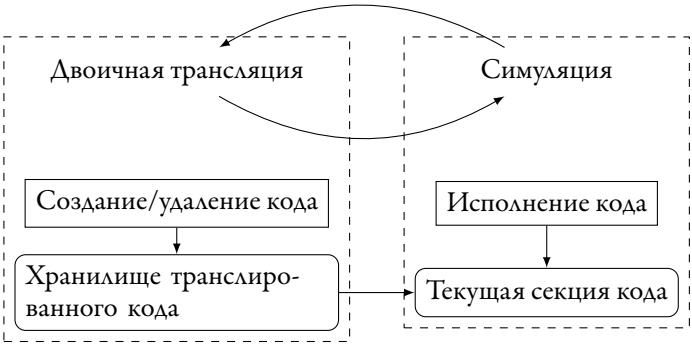


Рис. 3.3. Динамическая ДТ. Фаза симуляции, использующая сгенерированный код, периодически сменяется фазой трансляции, хранящей уже существующие и создающей новые секции хозяйского кода из гостевого

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 3.4.

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. 3.4). 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.

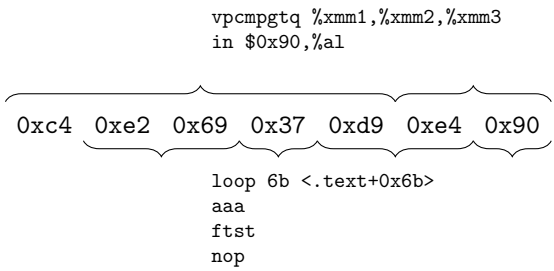


Рис. 3.4. Обнаружение кода. Смысл содержимого памяти меняется при изменении стартового адреса. Пример двух интерпретаций для фрагмента кода архитектуры IA-32

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 3.2.

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 [11].

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. 3.5). 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. 3.6). 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 [12].

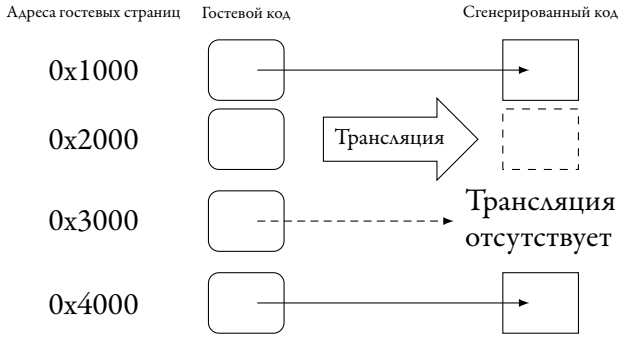


Рис. 3.5. Двоичная трансляция целых страниц. Для ранее исполненных блоков переиспользуются оттранслированные секции хозяйского кода. Процесс симуляции прерывается для трансляции новой страницы

3.2. 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.

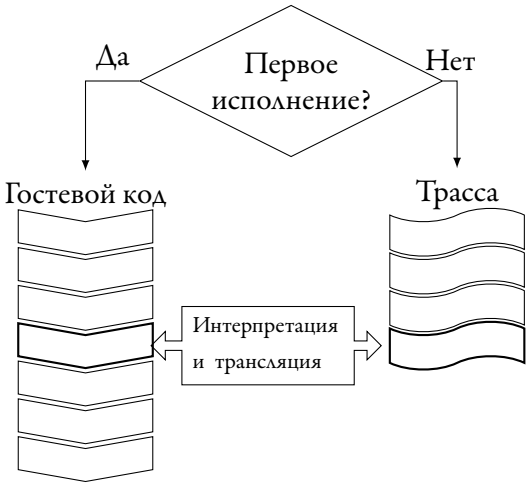


Рис. 3.6. ДТ с трассами исполнения. Первое исполнение каждой гостевой инструкции производится с помощью интерпретатора, при этом также осуществляется её трансляция и сохранение результатата в трассе

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 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.

3.3. 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. 3.7 приведён пример часто используемой оптимизации некоторого блока трансляции с одной точкой входа и выхода [4] 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 [13], применимы и при двоичной трансляции.

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

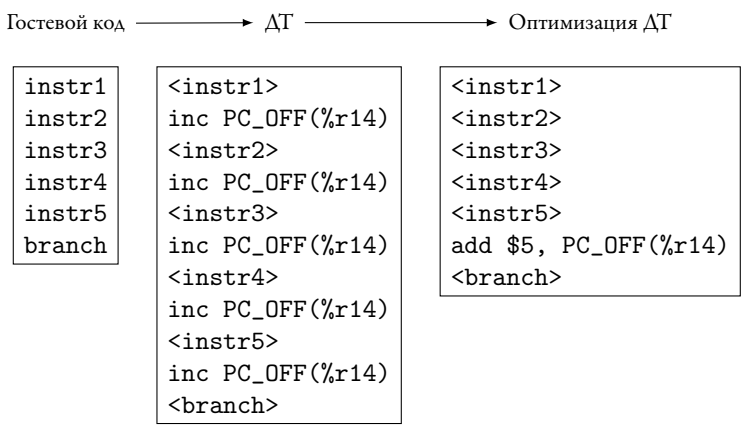


Рис. 3.7. Пример простой оптимизации кода блока трансляции. Инструкции `inc` продвижения регистра `PC` после каждой капсулы заменены одним сложением `add` в конце блока

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 3.7.

3.4. 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. 3.8 и 3.9 приведено соотношение этапов исполнения и ожидания для этих двух активностей в случае последовательной ДТ и ДТ, вынесенный в отдельный поток.

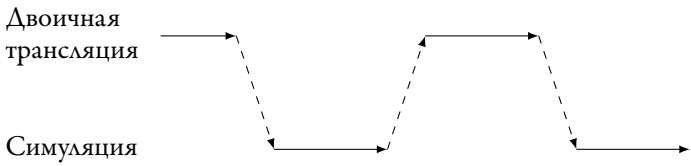


Рис. 3.8. ДТ и симуляция выполняются последовательно

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-

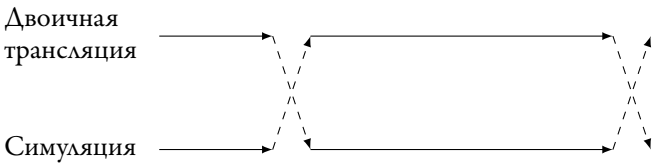


Рис. 3.9. ДТ вынесена в отдельный поток

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.

3.5. 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.

3.6. 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. 3.10). 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-

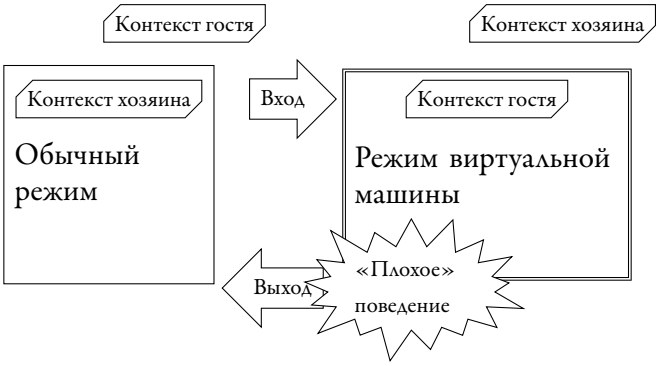


Рис. 3.10. Аппаратная поддержка переключения контекста между монитором виртуальных машин и гостевой системой. При входе в режим виртуальной машины с помощью системной инструкции контекст хозяина сохраняется в выделенной области памяти и заменяется контекстом гостя. При любой причине обратного перехода 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 [3]. Такие операции могут производиться аппаратурой, а не программой хозяина, что также упрощает (и делает более надёжной) его реализацию.

3.7. 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 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. 3.11). Данный «режим» имеет название *гиперсимуляция* (*англ.* hypersimulation).

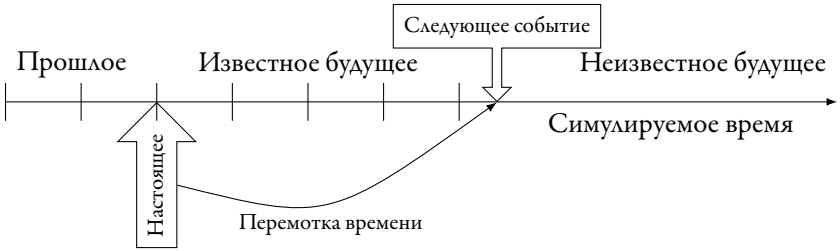


Рис. 3.11. Гиперсимуляция. При обнаружении возможности симулируемое время продвигается вперёд до следующего события, состояние процессора при этом неизменно

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 атомарно 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 5), 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 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 скачком переместить симулируемое время первого процессора до конца его квоты, не изменив при этом его архитектурное состояние. Затем передать управление второму процессору, который выполнит необходимую разблокировку.

Нахождение шаблонов гиперсимуляции. Для того, чтобы использовать гиперсимуляцию, симулятор должен уметь детектировать ситуации, в которых она применима. Такая функциональность может быть реализована двумя способами.

Задавать шаблоны вручную. В этом случае машинный код цикла — *шаблон гиперсимуляции* — формируется пользователем и передаётся симулятору, который при работе проверяет, соответствует ли ему гостевой год. Если найдено совпадение, то применяется гиперсимуляция. Для этого подхода характерны следующие особенности.

– Пользователь должен выполнить работу по анализу своего сценария симуляции, идентифицировать в нём места, которые выиграют в производительности от гиперсимуляции, и сформулировать для них шаблоны. Как правило, эта работа выполняется, если обнаружено, что для конкретного сценария наблюдаются проблемы с производительностью.

– Для каждого сценария шаблоны могут быть свои. Например, каждая операционная система может реализовывать атомарные примитивы собственным образом.
– Симулятор не может проверить, действительно ли переданные ему шаблоны корректны, т.е. что симуляция с их использованием не приведёт к некорректному изменению состояния гостевой системы. Вся ответственность при этом ложится на человека.

Автоматическое детектирование кода, допускающего гиперсимуляцию. Чтобы избавить человека от кропотливой работы по выявлению шаблонов, разумно попытаться переложить её на сам симулятор. Для этого он должен уметь анализировать циклы в гостевом коде и обнаруживать среди них те, для которых допустимо пропустить часть итераций. Для двоичного транслятора это может делаться в одной из стадий оптимизаций результатов трансляции, когда известны связи между отдельными блоками. Рассмотрим плюсы и минусы этого подхода.

– Самое очевидное преимущество — это избавление от необходимости делать ручную работу по выявлению шаблонов для каждого сценария симуляции.
– Программа зачастую способна выявить связи между блоками лучше, чем это смог бы человек, и таким образом обнаружить больше шаблонов для оптимизации, обладающих сложными условиями применения.
– С другой стороны, автоматическое детектирование, не обладающее полным представлением о специфике сценария симуляции, должно действовать консервативно. Это приводит к тому, что шаблоны для некоторых классов алгоритмов не будут построены. Например, если внутри цикла происходит чтение или запись в устройство, то он не может быть автоматически гиперсимулирован, т.к. неизвестны побочные эффекты от таких обращений.

3.8. 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 системы 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. 3.12).

- 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.

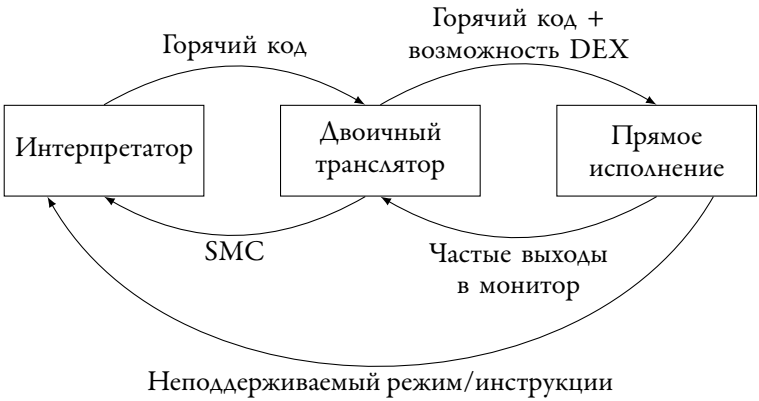


Рис. 3.12. Динамическое переключение режимов симуляции

3.9. 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.

3.9.1. 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
```

3.9.2. 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,0xfffffffe4
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%
```

3.10. Вопросы к главе 3

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. *Chernoff Anton*, *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.
3. *Drepper Ulrich* The Cost of Virtualization // ACM Queue. — 2008. — Янв. — 30–35. — URL: http://queue.acm.org/detail.cfm?id=1348591.
4. 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.
5. 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.
6. *Hauser John* SoftFloat. — 1 июн. 2010. — URL: http://www.jhauser.us/arithmetic/SoftFloat.html (дата обр. 08.02.2013).
7. 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.
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. — C. 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. *Seebach Peter* Unrolling AltiVec, Part 1: Introducing the PowerPC SIMD unit. — 2005. — URL: http://www.ibm.com/developerworks/power/library/pa-unrollav1.

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

12. *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.

13. 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.

4. Моделирование с использованием трасс

Там на неведомых дорожках следы невиданных зверей.

А.С. Пушкин

Симуляция на основе трасс (*англ.* trace driven simulation) базируется на возможности переиспользования истории дискретных событий для независимых экспериментов по изучению некоторой системы. Под трассами (*англ.* trace — след) понимаются истории событий, произошедших в системе за определённый период времени, записанные в файл в порядке их возникновения [2]. В отдельном событии может быть отмечена информация о том, как изменилось состояние системы из-за внешних или внутренних факторов. При этом повторная симуляция системы состоит в «проигрывании» трассы[1] и соответствующем ей изменении состояния модели.

Рассмотренные ранее функциональные модели были предназначены для демонстрации эффектов в изучаемых системах в режиме реального хозяйского времени, также соответствующего (как правило, замедленному) течению симулируемого времени. В частности, это позволяло обеспечивать взаимодействие с внешним пользователем, таким образом приводя систему к различным промежуточным и конечным состояниям. Такие модели принято характеризовывать как онлайн (*англ.* online). Трассы же отображают эволюцию, произошедшую когда-то в прошлом и в общем случае не подлежащую модификации — оффлайн (*англ.* offline).

Формат и содержимое трассы зависят от их назначения и изучаемой системы. В общем случае она должна содержать упорядоченную запись всех внешних событий. Например, в случае изучения некоторого ЦПУ для него таковыми являются доступы в память, порты ввода-вывода и прерывания, и текстовое представление трассы может иметь

[1] По аналогии с магнитными лентами, когда-то использовавшимися для хранения данных.

следующий вид:

```
time=10 read  addr=0x45df4 result=0x0455
time=14 write addr=0x35df4 data=0xffff
time=20 interrupt 10
time=25 port write addr=0x10 data=0xabcd
```

В данном примере отражены характерные составляющие трасс:

- Моменты времени возникновения событий.
- Описание типа события.
- Параметры события.
- Результаты выполнения (если есть).

4.1. Изучение пространства конфигураций с помощью трассировки

Нет большой пользы в том, чтобы раз за разом моделировать одно и то же явление, всегда, в конце концов, получая один и тот же результат. Однако можно в некоторых пределах менять характеристики частей модели, при этом оставляя трассу неизменной. Ключевая идея состоит в том, что порядок и структура событий будут одинаковыми для параметризованных запусков систем, и поэтому нет необходимости многократного прогона имитационной модели, достаточно сохранить порядок событий один раз. Например, можно модифицировать модель потребления электроэнергии узлами системы. При этом история доступов в память не изменится, значит, её можно сохранить в трассу, которую затем использовать для прогона на изменённой модели для получения новых значений искомых величин.

Рассмотрим другой пример использования симуляции с помощью трасс: изучение скорости работы некоторой программы на новой микроархитектуре; при этом производится сравнение со старой уже существующей. Обе системы совместимы на уровне макрокоманд, однако их внутреннее устройство различно.

1. Трасса записывается на реальной (старой) аппаратуре, в неё попадают все доступы в память (с результатами отдельных чтений

и записей), а также другие внешние и внутренние события (прерывания, исключения).
2. Сохранённые результаты подаются на модель, которая использует их как историю взаимодействия с внешним миром, при этом изменяя своё внутреннее состояние соответствующим образом и сообщая задержки, при этом возникающие.
3. На этапе анализа не приходится писать точную имитационную модель новой аппаратуры, достаточно иметь лишь упрощённую схему задержек (рис. 4.1).

Рис. 4.1. Процесс использования трассы в оффлайн-симуляторе

Заметим, что начиная со второго шага нет необходимости иметь доступ к изучаемому приложению ни в виде исходного кода, ни даже в виде кода скомпилированного — после генерации трассы они не нужны [1]. Это может оказаться важным в случае, если изучаемое приложение является закрытым или каким-либо образом ограниченным в распространении — мы можем его исследовать и получить важные характеристики его работы по «безликой» трассе.

Важно понимать, что в трассе должны быть отражены только внешние события: изменения во внутреннем состоянии должны отслеживаться самой моделью. В качестве примера рассмотрим задачу изучения производительности системы памяти и кэшей (о моделировании кэшей см. главу 9). Трасса содержит только информацию о помледовательности, типах и адресах доступов. Количество линий, их ёмкость, топология соединений и содержимое отдельных ячеек, а также временные характеристики кэшей определяется и отслеживаются самой моделью, которая также отвечает за полезные результаты — вычисление среднего времени доступа в память, составление профиля времён

доступов в зависимости от адресов и т.п.

4.2. Ограничения метода

Очевидно, что ценность методики трассировки зависит от степени устройчивости поведения приложения при работе. Это не всегда возможно, так как поведение моделей может сильно отличаться даже на одних и тех же программах. Особенно серьёзно проблема встаёт для параллельных систем с большим числом агентов, регулярно синхронизирующих своё исполнение с помощью передачи сообщений. В этом случае затруднительно получить трассу, отражающую следующий важный аспект их работы. Порядок событий зависит от относительных задержек между потоками, а они в свою очередь сильно зависят от параметров симуляции. Трасса одних только архитектурных событий, не учитывающая явным образом алгоритмические аспекты используемых методов синхронизации, может оказаться не отражающей функционирование той же самой системы в немного модифицированных условиях.

4.3. Области применения

Кроме рассмотренного ранее в этой главе сценария изучения производительности новых систем, трассы могут использоваться для проверки корректности модели путём сравнения трассы, полученной на ней, с эталонной, взятой или с реальной аппаратуры, или с точного (но медленного) потактового симулятора.

Другая важная область использования трасс — валидация имитационных моделей, т.е. доказательство их корректности. При этом на каждом шаге сравнивается состояние модели и состояние, записанное в «эталонной» трассе, полученной с помощью некоторого доверенного источника. При их расхождении мы можем точно сказать, какая операция привела к ошибке, что сокращает время её исправления.

Тем не менее в ряде работ озвучивается мнение, что время использования трасс для непосредственного изучения архитектур ушло, поскольку они не могут отразить многие особенности современных па-

раллельных систем со сложными спекулятивными микроархитектурами, допускающими откат состояния, не отражаемый в архитектурной трассе.

4.4. Сэмплирование трассы

Полная трасса некоторого процесса может содержать миллиарды событий и занимать гигабайты на устройстве хранения. Полное её проигрывание при этом отнимает много времени. Для сокращения длины эксперимента измерения проводятся только для серии коротких отрезков. Сами отрезки (*сэмплы*) в исходной трассе выбираются или через регулярные интервалы, или случайным образом. Такой подход называется **сэмплированием** (*англ.* sample) и позволяет получить компромисс между длительностью анализа и его точностью.

На рис. 4.2 показана последовательность трёх используемых фаз при сэмплировании.

- Функциональная симуляция обладает высокой скоростью, поскольку опускает большинство внутренних деталей реализации. Она используется для быстрого **перематывания** участков между отрезками измерения. При этом потактовая модель отключена, её внутреннее состояние неопределено.
- **Разогрев** потактовой модели, которая получает на вход данные из трассы и симулирует изменения в состоянии модели устройства и связанные с ними задержки, однако выдаваемые ей результаты игнорируются, так как они не соответствуют корректному исходному состоянию устройства.
- **Измерение** на сэмпле производится со включенной потактовой моделью, состояние которой при достаточном разогреве соответствует реальной системе.
- По окончании обработки всех сэмплов полученные на них результаты суммируются и нормируются для того, чтобы быть приведёнными к длине полной трассы.

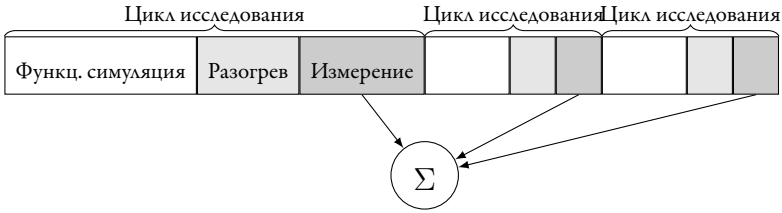


Рис. 4.2. Сэмплирование трассы. Потактовая модель включена только на этапах разогрева и измерения, результаты собираются только при измерениях

4.5. Вопросы к главе 4

Вариант 1

1. Что из нижеперечисленного может входить в трассу, используемую для симуляции:

   a) доступы во внешнюю память,
   b) внешние прерывания,
   c) состояние регистров,
   d) временные метки,
   e) дизассемблер текущих инструкций?

2. Какие сценарии представляют наибольшую сложность для метода симуляции с помощью трасс:

   a) многопоточная гостевая система,
   b) гостевое приложение с закрытым исходным кодом,
   c) изучение производительности приложений?

3. Как называется методика, призванная уменьшить объём данных трассы, требуемых для анализа работы приложения:

   a) манипулирование,
   b) фильтрация,
   c) интегрирование,
   d) сэмплирование?

Вариант 2

1. Какой вид активности невозможен при симуляции трасс:
    a) интерактивное взаимодействие с пользователем,
    b) загрузка операционной системы,
    c) работа с периферийными устройствами?
2. Какие типы событий должны быть отражены в трассе работы приложения для того, чтобы она была полезна:
    a) только внешние события: доступы в память, к устройствам,
    b) только внутренние события: изменения регистров,
    c) и внутренние, и внешние события?
3. Выберите правильный порядок операций при обработке трассы:
    a) перематывание – измерение – разогрев,
    b) разогрев – перематывание – измерение,
    c) перематывание – разогрев – измерение.

Литература

1. *Cain Harold W.* Precise and Accurate Processor Simulation. — 2002. — URL: http://pages.cs.wisc.edu/~cain/pubs/caecw2002_final.pdf.
2. Trace-driven simulation of multithreaded applications / Alejandro Rico [и др.] // ISPASS. — IEEE Computer Society, 2011. — 87–96. — ISBN: 978-1-61284-367-4. — URL: http://ispass.org/ispass2011/slides/3_1.pdf (дата обр. 01.05.2012).

5. Моделирование полной платформы. Дискретная симуляция событий

Караван идёт со скоростью самого медленного верблюда

Восточная пословица

В предыдущих главах подробно разбираются различные методики эффективного моделирования центральных процессорных устройств (точнее, пары «процессор плюс рудиментарная модель внешней памяти»). Однако современные вычислительные комплексы сложнее в своём устройстве: во-первых, они состоят из множества независимых устройств, большинство которых сильно отличается от ЦПУ по принципам своей работы (и, как мы дальше увидим, их симуляция также производится другим образом); во-вторых, в одной системе может быть больше одного процессора. В реальности все устройства должны работать одновременно, тогда как модели исполняются последовательно в одном потоке (вопросы параллельных моделей составляют отдельную объёмную тему и будут рассмотрены в главе 6). Рассмотрим существующие подходы к решению указанных задач.

5.1. Дискретная модель событий

Наиболее общая методология описания произвольных дискретных систем опирается на следующие упрощающие предположения о поведении их компонент.

5.1.1. Дискретность событий и времени

Во-первых, напомним, что необходимое условие для моделирования системы — возможность полного описания её состояния (получаемого как сумма состояний входящих в неё подсистем, т.е.

устройств) в конечном количестве ячеек памяти. Любое событие заключается в изменении этого состояния.

В реальных электронных системах практически все процессы передачи информации происходят в течение некоторого промежутка времени. Например, при передаче 8 бит по кабелю последовательного порта существует момент начала передачи, соответствующий первому биту, и момент, соответствующий передаче последнего.

В дискретной модели событий мы предполагаем, что каждое событие и связанные с ним изменения в состоянии модели происходят «мгновенно» в момент, соответствующий моменту завершения процесса. Таким образом, мы избавляемся от необходимости изучать течение события целиком. Если окажется, что точности такого представления недостаточно, единое событие заменяется на несколько более мелких, каждое из которых дискретно.

Окружающее нас время непрерывно, и не существует общепризнанных физических доказательств его дискретности, т.е. существования мельчайшего неделимого промежутка времени. Поэтому и дискретные события могут быть привязаны к непрерывным моментам времени. Однако в цифровых синхронных системах время дискретно, при этом мельчайшим интервалом является один цикл генератора тактовых импульсов, причем моменты дискретных событий в такой системе привязаны к границам тактов.

Уточним дополнительно понятие «событие»: изменение состояния системы или порождение одного или более новых событий, которые запланированы произойти в будущем. Например, некоторый таймер, будучи включённым, генерирует прерывание каждые 10 тактов. Тогда событие этого устройства заключается в изменении состояния сигнальной шины и создании следующего события, отдалённого на 10 тактов в будущее.

5.1.2. Симуляция с фиксированным шагом

Симуляция с фиксированным шагом (*англ.* time stepped) [3] — наверное, самая первая идея, которая приходит на ум. Интуитивно понятно, что для цифровых систем, управляемых тактовым генератором с фиксированной частотой, существует минимальный интервал вре-

мени Δt, разделяющий события. В таком случае алгоритм состоит в том, чтобы продвигать симулируемое время скачками Δt, на каждом шаге исполняя все события, случившиеся на этом шаге, если их число больше нуля.

Это последнее обстоятельство играет существенную роль. В реальности далеко не на каждом такте происходят какие-либо события, требующие симуляции. Большую часть времени симулятор будет останавливаться только для того, чтобы обнаружить, что на текущем такте делать нечего. Конечно, это обстоятельство ограничивает скорость симуляции.

Отметим, что моделирование процессоров с помощью интерпретации является примером симуляции с фиксированным шагом, равным длительности одной инструкции.

5.1.3. Симуляция, управляемая событиями

Рассмотрим более эффективную схему симуляции, иногда характеризуемую как управляемая событиями (*англ.* event driven). Продвижение симулируемого времени при этом делается «скачками» от одного события до следующего.

Мы интерпретируем события как изменение состояния одного или нескольких моделируемых устройств. Это позволяет нам упорядочить события всей системы по значениям меток времени, когда они должны произойти. Соответственно моделироваться они будут в указанном порядке.

В практических реализациях для хранения событий используется структура данных очередь с приоритетами, в которую добавляются новые события, создаваемые при обработке уже существующих. Процесс моделирования сводится к выборке запланированных событий в правильном порядке и их «выполнение» (т.е. изменение состояний согласно тому, что представляют события). На рис. 5.1 приведён пример состояния очереди событий для некоторой системы.

Добавление новых событий. Приоритетом при добавлении новых событий в описанную выше очередь является метка времени. Таким образом, из очереди первыми извлекаются события с наивысшим

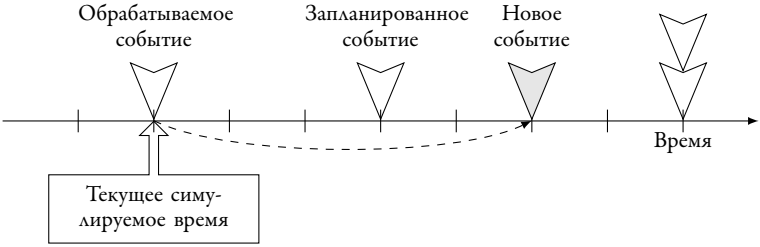


Рис. 5.1. Дискретная симуляция событий. События привязаны к границам тактов. Показано состояние системы в некоторый момент: несколько запланированы в будущем, одно исполняется, при этом оно порождает новое и добавляет его в очередь

приоритетом, т.е. с наименьшим значением метки, а при их равенстве у двух событий первым будет добавленный в очередь раньше, т.е. согласно принципу работы очереди.

Введённая методика моделирования получила название *симуляции дискретных событий* (*англ.* discrete event simulation, DES) [1, 2, 5], которая является достаточно общей: с её помощью можно описать и исследовать поведение очень широкого класса вычислительных устройств, а также любых других систем, никак не связанных с компьютерами. Важным её преимуществом является разделение архитектурного состояния симуляции, хранимого в составе модельных объектов, и порядка вызова обработчиков этого состояния, определяемого единой очередью.

Замечание. *По своей сути подход DES во многом схож с тем, что мы наблюдаем при отделении класса цифровых электронных схем от аналоговых, когда изучаемым сигналом становится не напряжение на выходах схем, а логический уровень. Несмотря на то, что схемы не перестают быть подчинены законам физики и напряжение на узлах остаётся аналоговой величиной, принимающей непрерывный диапазон значений, цифровые системы проектируются и должны функционировать таким образом, чтобы для их описания было достаточно лишь двух уровней сигнала. При этом мы пренебрегаем всеми переходными процессами, когда сигнал на входах узлов меняется непрерывным способом, а также тем*

обстоятельством, что скорость их распространения по проводам ограничена. Всё это позволяет значительно упростить анализ поведения системы, так как мы изолируем важные для нас качества и пренебрегаем частью несущественных свойств. Соответственно границы применимости наших суждений о её последующем поведении определяются справедливостью исходных предположений.

За значение текущего времени в симулируемой системе принимается значение временной метки последнего обработанного события. Ниже перечислены замечания к описанной выше базовой идее.

- Порождаемые события не могут попасть в прошлое, т.е. иметь метку времени меньше, чем текущее время.
- Обработка событий может не только порождать события в будущем, но и отменять некоторые из них (ещё не обработанные). Пример: описанный ранее таймер с периодом работы 10 тактов получил сигнал о полном выключении на 5-м такте своей работы. Обработка этого события заключается в изменении внутреннего состояния устройства, а также отмене ранее запланированного события, так как оно уже не произойдёт.
- Несколько событий могут иметь одинаковую метку времени. Чаще всего придерживаются правила, что они будут обработаны в порядке их добавления в очередь.
- В модели может существовать больше одной очереди. Например, можно иметь одну очередь, хранящую события, привязанные к инструкциям процессора, а другую — к фронтам тактового генератора. Другой вариант — многопроцессорные системы, в которых с каждым ЦПУ связана своя очередь. Обработка событий из всех очередей происходит в порядке, определяемом принципами работы и требованиями на очерёдность исполнения событий модели.

5.2. Два класса моделей

К настоящему времени мы рассмотрели несколько алгоритмов симуляции устройств. Их можно разделить на два класса по тому при-

знаку, какие силы приводят к изменению состояния участвующих моделей устройств.

1. Внутренние факторы, заложенные в саму модель устройства. Интерпретатор, двоичный транслятор и метод прямого исполнения выполняют шаг за шагом до тех пор, пока не будут остановлены или внешним воздействием (прерыванием, установкой флага исключения и т.п.), или окончанием входных данных (достижение лимита числа исполненных иструкций, времени симуляции и т.п.). Такие модели мы будем называть *исполняющими*, или управляемыми исполнением (*англ.* execution driven)
2. Внешние факторы, стимулирующие модель изменить однократно своё состояние и затем вернуть управление. Любое устройство, входящее в схему DES, является примером такого подхода. Соответствующие им модели — *неисполняющие*, или управляемые событиями (*англ.* event driven).

Можно заметить, что DES не является самым удобным представлением для моделирования центрального процессора с точки зрения обеспечения высокой производительности симуляции. ЦПУ исполняет инструкции на каждом такте (шаге) своей работы. Необходимость часто проверять состояние очереди событий сводит эффективность «улучшенных» техник (двоичная трансляция, прямое исполнение и т.п.) симуляции на нет.

В реальных системах обычно присутствуют устройства, некоторые из которых удобно моделировать как исполняющие, тогда как остальные должны быть неисполняющими. Необходимо как-то сочетать оба класса, и для этого существует следующее решение.

1. Определяется длительность интервала, в течение которого в моделируемой системе не произойдёт никаких событий. Эта величина равна расстоянию от текущего момента до самого раннего ещё не обработанного события в очереди.
2. Управление передаётся в модель процессора, которая исполняется некоторое время, не превышающее найденное в первом пункте значение. Затем она останавливается и возвращает управление симулятору.

3. Симулируемое время продвигается на число тактов, потраченных процессором. События обрабатываются по модели DES. Затем мы переходим к первому шагу.

Схема чередования симуляции исполняющего устройства и обработки событий из очереди событий изображена на рис. 5.2.

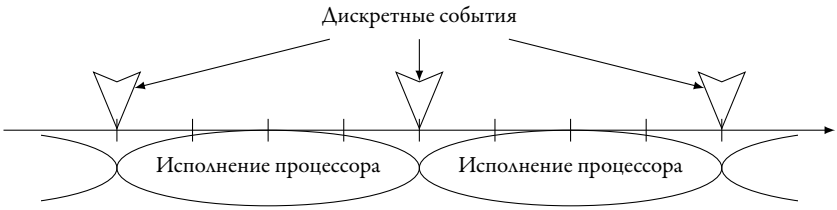


Рис. 5.2. Симуляция исполняющего устройства, перемежающаяся с обработкой дискретных событий

В каких случаях следует использовать модели каждого из описанных типов? Устройство выгодно представлять исполняющей моделью, если для него верно следующее:

1. Оно меняет своё состояние каждый или почти каждый такт.
2. События к нему от сторонних устройств приходят в среднем редко (раз в 100–1000 тактов).
3. Интерес для исследователя представляют внутренние процессы устройства [4].

Устройство следует симулировать как неисполняющее, если ему присущи следующие свойства.

1. Изменение его состояния происходит в среднем редко и асинхронно по отношению к остальным устройствам.
2. Характер взаимодействия с другими агентами представлятся в виде «запрос–отклик».

3. Оно может быть представлено как «чёрный ящик» без внутренней структуры.

Совместную работу системы моделей, состоящих как из исполняющих, так и неисполняющих устройств, можно представить как совместную работу двух или более симуляторов (*косимуляцию*), чередующих своё исполнение таким образом, чтобы выдерживались инварианты, определяющие согласованное течение симулируемого времени (рис. 5.3).

Рис. 5.3. Косимуляция моделей исполняющих устройств и DES. При каждой передаче управления соответствующая модель сообщает другой о том, насколько было продвинуто симулируемое время

Отметим, что то, какую модель устройства — исполняющую или неисполняющую — создавать, определяется сценарием работы симулятора в целом и этой модели в его составе. Иногда микропроцессор необходимо моделировать очередью событий, но чаще всего он представляется как исполняющая модель. И наоборот, модель некоторого периферийного устройства может быть исполняющей.

5.3. Моделирование многопроцессорных систем

Рассмотрим случай, когда в моделируемой системе присутствует более одного исполняющего устройства, например несколько процессоров. При этом в реальности они работают одновременно (параллельно), и данный факт необходимо отразить при их моделировании. Отметим, что устройства взаимодействуют всегда с помощью сообщений, время доставки которых конечно и составляет как минимум один такт.

Самое очевидное решение — чередовать исполнение всех процессоров на каждом шаге. В таком случае их состояние и «локальное» симулируемое время всегда будут отличаться не более чем на один такт.

Недостаток подхода тоже легко понять — такая система будет иметь низкую скорость работы из-за частого переключения моделей и связанного с ними моделируемого состояния. Режимы двоичной трансляции и прямого исполнения невозможно будет задействовать.

Облегчающим обстоятельством является тот факт, что в большинстве случаев нет необходимости выдерживать относительный сдвиг времени процессоров очень малым — ведь в реальности синхронная работа процессоров не наблюдается, и она не гарантируется исполняющимися на них программами. Поэтому мы можем исполнять отдельные процессоры достаточно большими «кусками», перемежая исполнение всех моделей, которые получают возможность задействовать оптимизирующие техники, например ДТ.

Отрезок времени, выделяемый устройству на исполнение, именуется *квотой* (другие названия — квант времени, quota, quantum, time slice). Устройство, находящееся в процессе исполнения в рамках своей квоты, считается текущим. Процесс исполнения многопроцессорной системы проиллюстрирован на рис. 5.4 и 5.5.

5.3.1. Замечания к предложенной схеме

– Процессор может исполнить меньше инструкций, чем содержится в выданной ему квоте. Пример причины для ранней остановки — событие во внешнем устройстве, которое необходимо обработать согласно его метке времени. После обработки всех

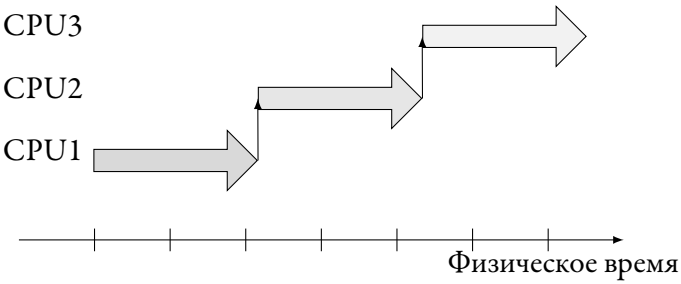


Рис. 5.4. Совместная симуляция нескольких процессоров. Отдельные интервалы симуляции чередуются на хозяйском процессоре

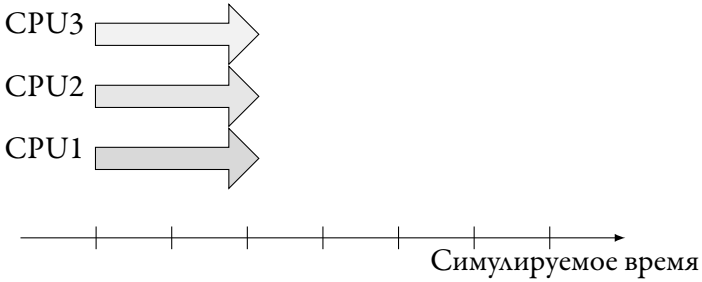


Рис. 5.5. Совместная симуляция нескольких процессоров. В контексте моделируемой системы процесс выглядит как параллельная непрерывная работа

событий, «мешавших» продвижению текущей модели, она либо может продолжить исполнение остатка своей квоты, либо передать управление другому устройству — это зависит от деталей алгоритма планировщика.

– Не следует увлекаться излишне большими квотами, пытаясь ускорить исполнение — это может негативно повлиять на точность модели, потому что моменты прерывания работы исполняющих моделей являются точками синхронизации состояния всей системы. Между ними каждое устройство работает в полной изоляции, в то время как остальные заморожены и не могут посылать ему никакие сигналы. Если программа, исполняемая

в модели, ожидает сообщений от других систем в течение ограниченного времени, то квота не должна превышать это время тайм-аута.
– В духе модели DES операция переключения текущего исполняющего устройства может быть реализована как псевдособытие, периодически вставляемое в очередь и при своей обработке вызывающее деактивацию текущего и выбор следующего активного процессора.

5.4. Вопросы к главе 5

Вариант 1

1. Определение понятия «квота», используемого в симуляции многопроцессорных систем.
2. Определение понятия «неисполняющее устройство».
3. Выберите правильный вариант продолжения фразы: Симулируемое время в моделях DES
   a) изменяется непрерывно,
   b) изменяется скачками фиксированной длительности,
   c) изменяется скачками, длительность которых различна.
4. Выберите правильный вариант окончания фразы: в моделях DES события могут обрабатываться, если они находятся
   a) только в голове очереди событий (самые поздние),
   b) только в хвосте очереди событий (самые ранние),
   c) в любой позиции в очереди событий.
5. Выберите правильный вариант окончания фразы: в моделях DES новые события могут быть добавлены
   a) только к голове очереди событий,
   b) только к хвосту очереди событий,
   c) в любую позицию в очереди событий.
6. Выберите сценарии, когда скорость симуляции, превышающая скорость работы реальной системы, нежелательна:

a) программа вычисляет значение некоторой функции в узлах сетки и выводит результаты на экран,
b) система ожидает ввода пользователя в течение ограниченного времени,
c) программа взаимодействует по моделируемой сети с другой моделируемой системой.

Вариант 2

1. Как могут проявиться недостатки слишком большой квоты?
2. Определение понятия «исполняющее устройство».
3. Выберите правильные возможности из перечисленных.

    a) Скорость течения симулируемого времени может быть меньше скорости течения реального времени.
    b) Скорость течения симулируемого времени может быть больше скорости течения реального времени.
    c) Скорость течения симулируемого времени приблизительно равна скорости течения реального времени.
    d) Все вышеперечисленные варианты верны.

4. Выберите правильный вариант окончания фразы: в моделях DES одно значение метки времени

    a) может соответствовать максимум одному событию,
    b) может соответствовать нескольким событиям, порядок их обработки при этом неопределён,
    c) может соответствовать нескольким событиям, порядок их обработки при этом определён,
    d) всегда соответствует нескольким событиям, некоторые из них могут быть псевдособытиями.

5. Выберите правильный вариант окончания фразы: в моделях DES события из очереди могут быть удалены

    a) только из головы очереди событий,
    b) только из хвоста очереди событий,
    c) из любой позиции в очереди событий.

6. Выберите правильное выражение для отношения скоростей моделирования систем с N гостевыми процессорами и с одним хозяйским процессором при однопоточной симуляции:
   a) $\frac{S(N)}{S(1)} = O(1/N)$,
   b) $\frac{S(N)}{S(1)} = O(N)$,
   c) $\frac{S(N)}{S(1)} = O(1/N^2)$,
   d) $\frac{S(N)}{S(1)} = O(N^2)$,
   e) $\frac{S(N)}{S(1)} = O(\ln N)$.

Литература

1. *Albrecht M. C. (Mike)* Introduction to Discrete Event Simulation. — 2010. — URL: http://www.albrechts.com/mike/DES/Introduction\%20to\%20DES.pdf.
2. *Cain Harold W.* Precise and Accurate Processor Simulation. — 2002. — URL: http://pages.cs.wisc.edu/~cain/pubs/caecw2002_final.pdf.
3. *Ferscha Alois* Parallel and Distributed Simulation of Discrete Event Systems. — 1995. — URL: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=0D07CB3110B890F247E12489F3264E62?doi=10.1.1.19.6226&rep=rep1&type=pdf (дата обр. 21.09.2013).
4. *Fritzson P.A.* Principles of object-oriented modeling and simulation with Modelica 2.1. — IEEE Press, 2004. — ISBN: 9780471471639. — URL: http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=5264347.
5. *Fujimoto Richard M.* Parallel and Distributed Simulation Systems. — 1st. — New York, NY, USA: John Wiley & Sons, Inc., 2000. — ISBN: 0471183830.

6. Параллельные симуляторы

> Эту игру смогла бы пройти и моя бабка, если бы она сохранялась столько же, сколько и ты!
>
> *Postal 2*

Некоторое время назад неограниченный рост частоты процессоров прекратился по различным причинам, в основном связанным с невозможностью удерживать их тепловыделение в допустимых пределах без специальных ухищрений, увеличивающих стоимость продуктов и уменьшающих их применимость для встраиваемых и мобильных решений. Силы проектировщиков микросхем обратились к другому способу, потенциально позволяющему повысить производительность — увеличивать количество независимых вычислительных ядер в составе вычислительного комплекса. Оставляя за рамками обсуждения вопросы целесообразности, эффективности и программируемости таких систем, рассмотрим, как распространение многопроцессорных систем влияет на постановку и решения задачи компьютерной симуляции.

1. *Необходимость моделирования многопроцессорных систем.* Как было рассмотренно в предыдущих главах, возможно моделировать многие устройства в одном потоке исполнения, перемежая их исполнение и следя за тем, чтобы разница в виртуальном времени не превышала допустимых для симуляции пределов. Однако при этом мы сталкиваемся с тем, что время исполнения модели растёт линейно с числом последовательно моделируемых, и скорость симуляции быстро становится неприемлемо малой.
2. *Возможность эффективного задействования многопроцессорных ресурсов хозяйской системы.* Если есть возможность задействовать все доступные ресурсы аппаратуры для работы программы и при этом получить ускорение, то это необходимо сделать.

Замечания о терминологии. Всюду в данной главе мы будем отвлекаться от деталей иерархической организации многопроцессорных систем и реализаций систем памяти. При этом термин «процессор» будет использоваться для обозначения устройства, исполняющего ровно один поток последовательных инструкций и имеющий одну копию архитектурного состояния. Также, если это явно не будет оговорено, понятия «процесс» и «поток» исполнения будут использоваться взаимозаменяемо.

6.1. Последовательные модели

Напомним кратко принцип работы двух последовательных однопоточных схем, рассмотренных ранее: симуляция многопроцессорных систем и модель дискретных событий. Затем попытаемся сформулировать способы их превращения в параллельные схемы, способные задействовать более одного хозяйского потока для симуляции, и рассмотрим возникающие при этом сложности.

6.1.1. Симуляция нескольких гостевых процессоров

В главе 5 был описан способ моделирования многопроцессорных систем в однопоточной симуляции: отдельные процессоры выполняются последовательно друг за другом, выдерживая определённый максимально допустимый разброс в значениях симулируемого времени. Остальные процессоры при этом «заморожены». Очевидно, что для такого алгоритма работы для модели системы N процессоров относительное замедление относительно модели с одним процессором составит в лучшем случае N; на практике оно будет больше из-за необходимости периодического переключения контекста — состояния модели от одного процессора к другому.

6.1.2. Дискретные события

При использовании модели DES имеется одна очередь событий, которую используют все моделируемые устройства. Скорость симуляции определяется количеством и сложностью обработки отдель-

ных элементов этой очереди. С ростом числа устройств она неизбежно падает. Растут также требования на другие ресурсы системы, такие как память, требуемая для хранения элементов очереди и архитектурного состояния.

6.2. Параллельные модели

Первый шаг к созданию параллельной симуляции — это распределение работы на несколько независимых потоков. Каждый из них выполняет свою часть работы и некоторым образом способен взаимодействовать с остальными потоками той же симуляции.

Обрисуем достаточно общий и одновременно важнейший с практической точки зрения сценарий симуляции, требующий параллельного исполнения. На рис. 6.1 приведена общая схема распределения модели SMP-системы на несколько потоков. Каждый из них содержит группу моделируемых устройств, включая процессоры и периферию, имеет собственную очередь сообщений для хранения информации о запланированных событиях и может взаимодействовать с другими потоками (пока что не будем специфицировать точный механизм коммуникаций). Особое место в этой картине занимает моделируемая память — она общая для всей симуляции, чтение и запись её может происходить из любого моделируемого процессора или устройства.

Симуляция многопроцессорной системы. Отдельные потоки в этом случае содержат один или более гостевой процессор и исполняют его инструкции, модифицирующие память и вызывающие методы периферийных устройств, разделяемых с остальными гостевыми процессорами, заключёнными в своих потоках. Каждый поток исполнения при этом самостоятельно ведёт счёт исполненных шагов симуляции, т.е. количества завершённых инструкций.

PDES. При построении параллельной модели дискретных событий мы получаем т.н. parallel DES (PDES) [8—10, 19]. При этом с каждым потоком исполнения ассоциируется собственная очередь дис-

Рис. 6.1. Общая схема распределения симуляции на несколько потоков. Устройства и процессоры содержатся целиком внутри одного потока, тогда как модель памяти необходимо сделать доступной для всех

кретных событий, текущее значение симулируемого времени и архитектурное состояние одного или нескольких моделей устройств. Поскольку два взаимодействующих устройства могут оказаться в разных потоках, необходимо предусмотреть возможность создавать события в очередях, отличных от той, в которой находится порождающее событие.

Подчеркнём ещё раз важное по сравнению с последовательным вариантом изменение — каждый поток параллельной системы имеет своё значение симулируемого времени, которое используют заключённые в нём устройства. Разброс значений этих времён между потоками в общем случае не ограничен сверху заранее известной величиной. По этой причине обе описанные выше схемы не будут корректно работать по ряду причин, которые мы рассмотрим ниже. И, даже если предприняты меры по устранению таких причин, не следует ожидать линейного роста производительности с увеличением числа задействованных потоков из-за потерь на синхронизации различного рода, которые также будут рассмотрены.

6.3. Препятствия на пути к созданию корректной параллельной модели

Параллельное программирование в целом является на удивление сложным занятием по сравнению с написанием классических последовательных программ. Этому есть много причин, вызванных техническими трудностями проектирования и психологическими особенностями человеческого восприятия, которые достаточно подробно рассмотрены в обширной литературе [3, 13, 30]. Здесь лишь отметим четыре обстоятельства, важных для дальнейшего обсуждения и напрямую влияющих на задачу проектирования симуляторов.

1. Возможность гонок данных (*англ.* data race) при одновременном доступе к общим данным и необходимость использования различных механизмов синхронизации, чтобы избежать их.
2. Возможность взаимоблокировок двух или более процессов, пытающих получить доступ к общим ресурсам.
3. Неэффективная работа параллельного приложения из-за интенсивной или неправильной синхронизации его потоков, из-за чего значительная часть их простаивает, не выполняя полезной работы.
4. Результат исполнения параллельного приложения при идентичных входных данных может отличаться при различных его запусках из-за неопределённости порядка исполнения отдельных потоков. Недетерминистичность значительно усложняет отладку приложений.

Далее в этой главе рассматривается, как эти проблемы и особенности могут проявиться при создании и работе параллельных симуляторов.

6.3.1. Атомарные инструкции в моделях многопроцессорных систем

Для обеспечения работы примитивов синхронизации параллельных программ современные процессоры предоставляют так называ-

емые атомарные инструкции, при исполнении которых гарантируется, что чтение и/или модификация региона общей памяти, указанной в них, не будет пересекаться с доступом к той же памяти другими потоками. Существует множество вариантов этих инструкций в разных архитектурах [31].

В последовательном симуляторе атомарность гостевых инструкций[1] обеспечивается автоматически: в любой момент времени максимум один процессор активен, никто не может повлиять на результат исполнения. Корректная поддержка этих инструкций в случае параллельного симулятора требует явных усилий со стороны разработчика. Рассмотрим известные для этого способы.

Использование атомарных хозяйских инструкций. Идея использовать существующие хозяйские атомарные инструкции для симуляции атомарных гостевых достаточно проста. Она позволяет переложить задачу обеспечения корректной синхронизации с программы на аппаратуру. К сожалению, применимость этого способа ограничена случаями, когда набор атомарных инструкций хозяина достаточен для реализации всех инструкций гостя. Например, IA-32 содержит более десятка инструкций, которые можно использовать с префиксом `LOCK`, т.е. атомарных, тогда как ARM имеет только две инструкции данного типа — `LDREX` и `STREX`[2]. Несомненно, этот подход применим для случаев совпадения архитектур хозяина и гостя [17].

Использование критических секций. Следующим достаточно простым возможным решением проблемы обеспечения эксклюзивного доступа к памяти при симуляции гостевых атомарных операций является использование критических секций (или семафоров, замков, мьютексов и т.п.) в симуляторе. Прежде чем исполнять часть гостевой инструкции, модифицирующую память, соответствующий поток симулятора обязан получить эксклюзивное право на модификацию ре-

[1] Вообще всех инструкций, даже тех, которые в реальности не являются атомарными и могут участвовать в гонках данных.

[2] Атомарные инструкции `SWP` и `SWPB`, присутствующие в ARMv5, объявлены устаревшими в последующих расширениях [4].

гиона памяти, содержащего операнд [24]. Этим регионом может выступать вся физическая память (в таком случае есть только один замок на все потоки) или её блок, например, страница.

Тем не менее, использование критических секций для симуляции *только* атомарных инструкций оказывается недостаточно для корректной работы гостевых приложений. В [6] приводится пример практически важного гостевого сценария, приводящего к некорректной блокировке при исполнении на симуляторе. Это происходит из-за того, что обычные, не атомарные, гостевые доступы в память не требуют вхождения в критическую секцию и тем самым способны создать гонку данных при одновременной симуляции с атомарными. Решение — обязать *все* обращения в симулируемую память использовать вход в критическую секцию. Однако такое решение практически сведёт на нет весь выигрыш от параллельного исполнения, т.к. обращения к памяти из разных потоков приложения очень часты на практике.

Использование операций compare-and-swap и load-linked/ store-conditional. Проанализируем оба описанных выше приёма ещё раз и попытаемся наметить путь к общему решению.

Если удастся выразить все существующие атомарные инструкции несколькими универсальными операциями, то для хозяйских архитектур, реализующих этот базовый набор, удастся симулировать все инструкции гостевых систем. В работе по теоретическим основаниям параллельных алгоритмов [12] показано, что существующие синхронизационные примитивы могут быть реализованы с помощью атомарной операции «сравнить и обменять значения» (*англ.* compare-and-swap, CAS). Однако также показано, что алгоритмы для некоторых структур данных, использующие CAS, могут работать некорректно[1]. Решением является использование расширенной операции, известной как «сравнение и обмен двух ячеек» (*англ.* double compare-and-swap, DCAS). Однако ни одна современная архитектура не имеет машинных инструкций, соответствующих DCAS.

[1]Так называемая «проблема ABA» [7].

Следующий вариант — использование пары инструкций, называемых load-link и store-conditional (LL/SC) [15], позволяющих проверить, что цикл «загрузить-изменить-сохранить» для некоторого адреса в памяти прошёл без внешнего вмешательства, и сообщить об успехе или неудаче, что позволит повторить попытку провести операцию. Использование этой пары инструкций позволяет реализовать алгоритмы DCAS, CAS и другие примитивы.

Указанная возможность проверить успех завершения гостевой атомарной операции и в случае неудачи повторять блок симулирующего её кода является ответом также и на вторую проблему, связанную с небезопасностью использования критических секций только для атомарных операций.

Использование LL/SC можно считать частным случаем так называемой транзакционной памяти для оптимистичной синхронизации.

6.3.2. Модели консистентности памяти

Кроме обеспечения атомарности исполнения для подмножества инструкций, современные архитектуры накладывают определённые условия на то, в каком порядке могут быть видны все обращения в память из разных процессоров, определяя т.н. *модель консистентности памяти*. Она описывает, насколько свободно аппаратура может переставлять чтения и записи как одного, так и нескольких потоков для того, чтобы увеличить скорость работы системы в целом[1]. Модели консистентности характеризуются т.н. «силой», т.е. тем, насколько строго должен соответствовать наблюдаемый порядок доступов описанному в программе. Чем слабее модель, тем более «вольно» они могут быть переставлены, тем больше может быть выигрыш в производительности. Однако работа такой системы происходит менее интуитивно понятно с точки зрения программиста.

Подробное рассмотрение существующих моделей консистентности памяти выходит за рамки данной главы. Интересующийся чита-

[1]Отметим, что даже для однопроцессорной системы порядок доступов в память может не совпадать с программным как из-за влияния компилятора, переставившего их на этапе компиляции, так и аппаратуры, динамически определяющей порядок исполнения команд.

тель может найти подробную информацию об этой теме в обширной литературе [2], [22], [28, глава 9 и приложение A.7]. Тем не менее, различия в используемых моделях могут повлиять на корректность симуляции в случае, если гарантии хозяйской системы слабее тех, которые требуются для работы гостевого окружения.

Для целей контроля над порядком исполнения в точках приложения, для которых такой порядок критически важен, все современные архитектуры предоставляют так называемые инструкции-*барьеры* (*англ.* fence), которые гарантируют, что на момент их завершения все доступы в память определённого типа (чтения и/или записи) или завершились (для инструкций, предшествующих барьеру), или ещё не начались (для находящихся после неё в потоке исполнения). Для IA-32 это инструкции `LFENCE`, `SFENCE`, `MFENCE`, для IA-64 — `mf` [1], для ARM — `DMB`, `DSB`, `ISB`, для PowerPC — `sync`, `eieio`.

Отметим, что без достаточно точной модели протокола передачи сообщений между процессорами и элементами подсистемы памяти невозможно обеспечить строгое выполнение модели консистентности гостевой системы. Однако функциональные модели могут быть написаны и корректно работать, если они обеспечивают более сильную модель, что всегда может быть достигнуто с помощью барьеров.

6.3.3. Нарушения каузальности

Рассмотрим ещё одну проблему, подстерегающую на пути к построению параллельной модели, на примере модели дискретных событий с несколькими очередями, одновременно изменяющими своё состояние. Нарушение требования соответствия порядка событий в симулируемой системе относительно их порядка в системе реальной можно проиллюстрируем следующим образом.

На рис. 6.2 два потока взаимодействуют, посылая друг другу сообщения в виде событий, помещаемых в очередь и запланированных на исполнение в некоторые моменты в будущем. При некотором соотношении текущих симулируемых времён T_1 и T_2 может случится так, что добавляемое из другой очереди событие окажется в прошлом для получателя.

В реальной системе новое событие в обоих случаях получило бы

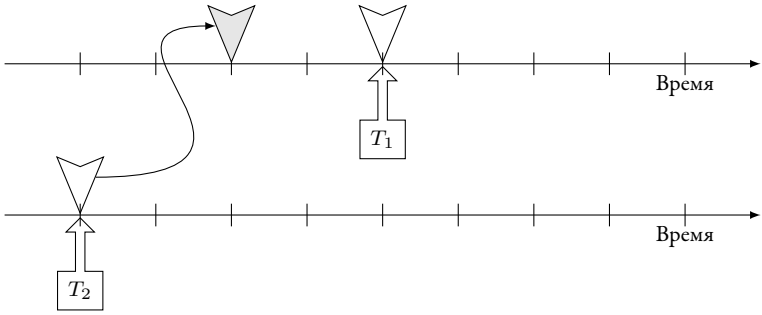


Рис. 6.2. Нарушение корректности относительного положения событий в случае $T_2 < T_1$. Новое событие оказалось в прошлом и не может быть обработано

строго одно и то же положение и относительный порядок во времени; при наивной параллельной симуляции возникают каузальные ошибки (нарушение причинно-следственной связи событий).

Отметим, что соотношение между T_1 и T_2 может быть произвольным по многим причинам, чаще всего не контролируемым пользователем. Например, хозяйская операционная система может решить вытеснить (т.е. временно заморозить) один из потоков, может случиться промах страницы в виртуальной памяти, промах в системе кэшей и т.п. Поэтому такая ситуация, когда один поток исполняется быстрее остальных и имеет значение симулируемого времени, значительно отличающееся от остальных, более чем вероятна.

Что же делать? Прежде всего, достаточно легко сформулировать, как распознавать ситуацию нарушения каузальности и корректировать её для первого случая $T_2 > T_1$. Достаточно к сообщениям об обращении к разделяемому состоянию добавлять метку времени того события, которое производит доступ. Поток, получающий сообщение (или использующий модифицированные данные), сравнивает значение этой метки со своим локальным временем, корректирует момент обработки события, а при обнаружении логического несоответствия (например, если новое событие оказывается в прошлом) сигнализирует о проблеме (рис. 6.3). Конечно, этот метод не позволяет разрешать возникшее затруднение, но только обнаруживать его.

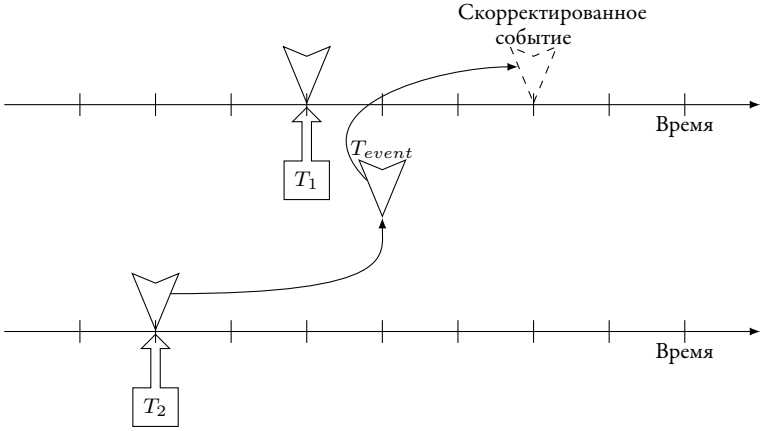


Рис. 6.3. Пересылка меток времени создания событий вместе с самим событием. Поток-приёмник корректирует положение события и принимает решение о том, соответствует ли ситуация условиям корректной симуляции

Пути решения проблемы нарушения каузальности Существует два принципиально различных подхода проектирования, способных помочь нам.

1. Расширить схему PDES таким образом, чтобы в принципе *не допускать* каузальных ошибок. Такие системы назовём *консервативными*.
2. *Детектировать* нарушения уже после их возникновения, а затем *исправлять* ситуацию, возможно, с помощью частичного перезапуска симуляции. Назовём этот класс *оптимистические* системы.

6.4. Консервативные модели

Мы хотим исключить возможность возникновения ситуаций, когда сообщение от потока, отставшего в симулируемом времени, приходит другому потоку «в прошлое». Предлагаемое решение — придержать (блокировать) исполнение отправителя сообщения до тех пор,

пока приёмник сам не продвинется в симулируемом времени до точки приёма [21]. Пример такой ситуации изображён на рис. 6.4. Очевидно, что блокировать процесс следует, если он посылает новое событие в чужую очередь, но не в собственную — в этом случае порядок нарушиться не может. Описанная схема позволяет добиться корректности с помощью того, что потоки, вырывающиеся вперёд, вынуждены впоследствии ожидать более медленных.

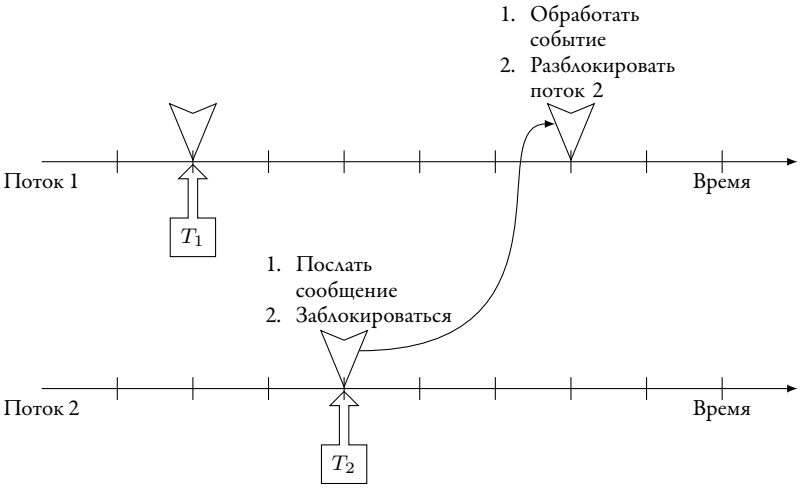


Рис. 6.4. Консервативный сценарий параллельной симуляции. Поток, желающий добавить событие в очередь другого потока, блокируется до момента обработки этого события

6.4.1. Необходимость предпросмотра

Предпросмотр (*англ.* lookahead) — определение того, на какое значение продвижение виртуального времени потока безопасно, т.е. не может вызвать нарушения каузальности. Чем больше это значение, тем большую «независимость» имеют отдельные потоки симуляции в своей работе, и тем выше её производительность в целом. Чрезмерно большое значение этой величины будет вызывать нарушения в

логике работы гостевых приложений. Наиболее естественный выбор значения для предпросмотра равен характерной задержке передачи сообщений между устройствами в реальной системе. Необходимость использования предпросмотра для консервативных схем [8] обусловлена невозможностью узнать состояние других потоков без получения от них сообщений.

6.4.2. Проблема взаимоблокировок

К сожалению, такая схема совсем не гарантирует, что сообщения в системе будут передаваться и симуляция будет прогрессировать [21] — возможна ситуация взаимоблокировки (*англ.* deadlock). На рис. 6.5 приведён пример такой ситуации.

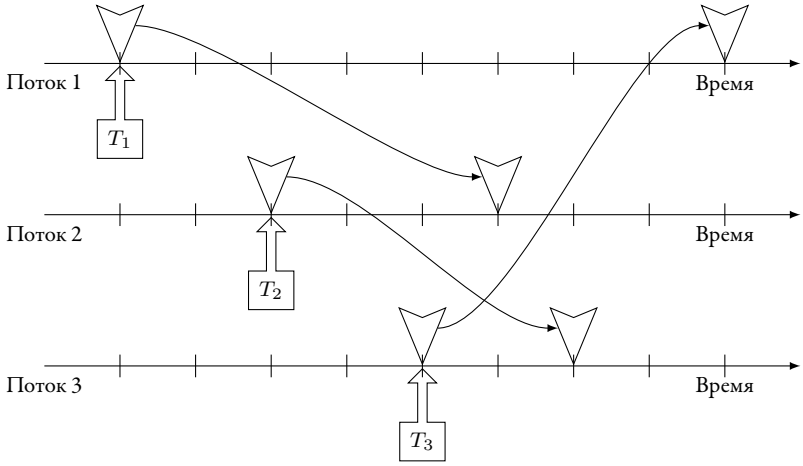


Рис. 6.5. Ситуация взаимной блокировки в системе с тремя потоками. Каждый процесс ожидает сигнала окончания блокировки от какого-либо другого потока

В случае большого числа потоков в симуляции взаимоблокировка может затронуть только часть из них, при этом остальные будут продолжать исполняться до тех пор, пока не посылают сообщения к заблокированной группе. Как разрешать сложившуюся ситуацию? Если уметь обнаруживать взаимоблокировку, то возможно принудительно

освободить один из участвующих в ней поток, разрешив ему исполняться. Это не нарушит условий корректности симуляции. Для определённости освобождать будем тот процесс, который имеет наименьшее значение текущего симулируемого времени. На рис. 6.5 им будет поток номер 1. Его разблокировка также оправдана с точки зрения выдерживания наименьшего разброса значений времён в разных потоках. Продвижение освобождённого потока в конечном счёте активирует и другие процессы, избавив их от блокировок, и симуляция продолжится в полном объёме.

Как можно детектировать ситуацию взаимоблокировки? В литературе описывается ряд алгоритмов, некоторые подразумевают присутствие центрального наблюдателя, некоторые являются децентрализованными. Кратко рассмотрим алгоритм, предложенный в [21]. Он основана на передаче специального сообщения (маркера) между процессами. Каждый процесс обязан передать его в течение ограниченного времени. В состоянии маркера хранится количество и состояния (блокирован или обрабатывает сообщения) посещённых процессов. Если в некоторый момент число обнаруженных маркером блокированных процессов достигает критического значения, то объявляется обнаружение ситуации взаимоблокировки.

Как корректно разрешить обнаруженную взаимоблокировку? Для этого достаточно освободить один из процессов, участвующих в ней. Какой из многих выбрать? Следует выбирать поток, значение симулируемого времени которого минимально — это позволит ему обработать сообщения от остальных заблокированных потоков, таким образом освободив их и позволив всей глобальной симуляции продвигаться.

Можно ли полностью исключить ситуацию блокировок? Поскольку сам факт их возникновения связан с тем обстоятельством, что отдельные потоки не имеют знания о симулируемом времени в остальных потоках, необходимого, чтобы самостоятельно принять решение, безопасно ли им продвигать вперёд своё время и посылать сообщения другим процессам, не опасаясь нарушения причинной связи. Как было описано ранее, метки времени приходят вместе с сообщениями. Но что делать, если архитектурных событий, вызывающих сооб-

щения, не предвидится? В таком случае можно отправлять т.н. *пустые сообщения* (*англ.* null messages), несущие только метку времени. Они должны периодически (в терминах физического времени) рассылаться/получаться всем и всеми агентами внутри одной симуляции; при приёме очередной метки поток может оценить, до какого момента следует продвигать своё время без опасности при этом послать некорректное сообщение. Блокировка всех потоков исключена, т.к. самый медленный поток не блокируется.

Примечательное следствие из этого обстоятельства состоит в следующем: даже если по архитектурным причинам все потоки заблокировались (например, все процессоры перешли в выключенное состояние и больше не создают новые события в своих очередях), периодическая отсылка пустых сообщений будет вынуждать их продвигать локальное симулируемое время.

Как часто необходимо слать пустые сообщения? При частой отправке значительная доля времени работы потока-отправителя тратится не на симуляцию, а на синхронизацию. При редких синхронизациях потоки-получатели будут простаивать, ожидая сигналов о безопасности собственного продвижения вперёд.

Следующий аспект функционирования такой схемы — в каком порядке и каким адресатам должны слаться пустые сообщения. Это можно делать несколькими способами, например следующими.

Всем агентам в системе. При таком сценарии мы обеспечиваем наиболее актуальную информацию всем потокам о глобальном симулируемом времени. Однако с ростом их числа эта схема плохо масштабируется из-за квадратичного роста числа передаваемых сообщений.

Случайным адресатам [11]. При таком подходе отправитель каждый раз выбирает случайным образом небольшую долю от полного числа потоков для сообщения им своего состояния. Благодаря постоянно изменяющемуся списку адресатов можно ожидать, что информация о каждом потоке будет постепенно распространяться по всей системе за конечное время.

Запрос получателем. Поток, участвующий в симуляции, при необходимости узнать значение виртуального времени другого по-

тока может сам явным образом запросить его, не ожидая ближайшей рассылки. Преимущество данного способа — пустые сообщения передаются только тогда, когда они действительно запрошены. Недостаток связан в необходимости передачи двух сообщений «запрос — ответ», что занимает в два раза больше времени.

Отметим, что схема, в которой каждый поток периодически шлёт пустые события только фиксированному набору адресатов, не гарантирует невозможность возникновения в ней неразрешимых взаимоблокировок.

6.5. Оптимистичные модели

На практике события нарушения каузальности могут происходить достаточно редко. Тем не менее, консервативный алгоритм будет блокировать потоки каждый раз, когда потенциально возможно получить их рассинхронизацию, тем самым снижая эффективность параллельной симуляции. Вычислить заранее, действительно ли блокировка будет необходима, затруднительно (см. секцию 6.11).

Поступим иначе. Во-первых, некоторым образом будем сохранять информацию о моментах в симулируемом прошлом. Необходимо иметь возможность восстановить состояние модели к каждому из выбранных моментов, не начиная симуляцию с самого начала. Назовём такой набор данных *точкой сохранения* (*англ.* checkpoint). В простейшем случае точка сохранения просто содержит значения всех архитектурных состояний всех устройств, входящих в симуляцию, а также значение симулируемого времени.

Во-вторых, позволим параллельной симуляции исполняться без блокировок потоков. При этом возможны нарушения каузальности, поэтому необходимо при передаче сообщений проверять метки их времени. Что же делать, если такое нарушение было обнаружено? В этому случае состояние модели восстанавливатеся из одной из точек сохранения, про которую известно, что на момент её создания нарушений обнаружено не было. Затем симуляция «опасного» участка

производится повторно оптимистичным образом или с использованием других описанных ранее схем, см. рис. 6.6.

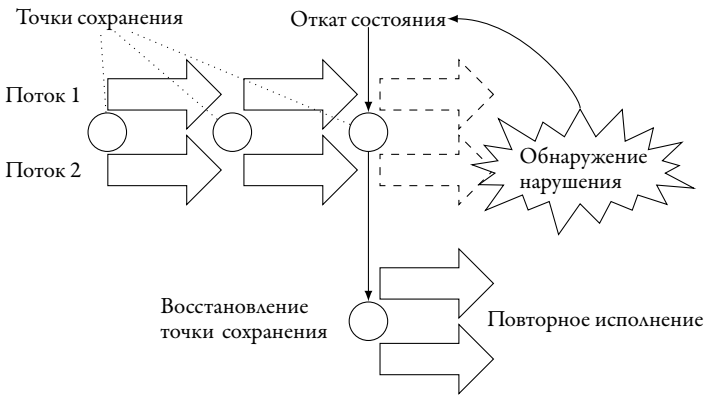


Рис. 6.6. Симуляция с периодическими точками сохранения. Штрихами показана часть симуляции, приведшая к нарушению корректности и откату к ближайшему сохранённому состоянию

Очевидно, что выигрыш в производительности от использования оптимистичной схемы возникает в предположении, что нарушения каузальности и связанные с ними откаты состояния будут нечастыми. На производительность влияют и другие аспекты схемы.

Цена создания точек сохранения. Для сохранения состояния симуляции требуется как место в памяти хозяина, так и некоторый интервал хозяйского времени, в течение которого вся симуляция остановлена. Способ минимизировать обе величины — сохранять в новой точке лишь изменения в состояниях устройств относительно предыдущей, т.е. использовать *инкрементальные* точки сохранения.

Частота создания точек сохранения. Чем ближе к потенциальному месту нарушения имеется записанное состояние симуляции, тем меньше шагов необходимо переисполнять в случае отката.

Стоимость отката состояния. Как и создание, откаты можно сделать инкрементальными, то есть затрагивающими лишь те части

симуляции, которые изменились с момента последнего сохранения.

Разработаны многочисленные варианты оптимистичных алгоритмов параллельной дискретной симуляции. Рассмотрим наиболее известный протокол, получивший название Time Warp[1] [14].

6.5.1. Time Warp

Определим основные понятия, используемые при описании алгоритма Time Warp.

- Сообщение — набор данных, описывающих событие, которое должно быть добавлено в одну из очередей событий. Оно характеризуется, кроме своего непосредственного содержимого, виртуальными временами отправки t^{send} и обработки $t^{receive}$.
- LVT (*англ.* local virtual time) — значение симулируемого времени отдельного потока, участвующего в симуляции. Для создаваемых событий их время отправки t^{send} равно значению LVT отправителя. В отличие от консервативных схем, эта величина может как расти в процессе симуляции, так и убывать в случае отката процесса.
- GVT (*англ.* global virtual time) — глобальное время для всей симуляции, определяющее, до какой степени возможно её откатывать. Глобальное время всегда монотонно растёт, всегда оставаясь позади локального времени самого медленного потока, а также оно меньше времени отправки самого раннего ещё не доставленного события:

$$GVT \leq \min\left(\min_i LVT_i, \min_k t_k^{receive}\right).$$

- Отставшее сообщение (*англ.* straggler) — событие, пришедшее в очередь с меткой времени $t_{straggler}^{receive}$, меньшей, чем LVT получателя. Его обнаружение вызывает откат текущего состояния, при

[1] Его авторы описывают свой протокол как «виртуальное время» по аналогии с существующим понятием «виртуальная память».

этом LVT уменьшается, пока не станет меньше, чем $t^{receive}_{straggler}$, после чего оно может быть обработано. После этого возобновляется прямая симуляция.

– Антисообщение (*англ.* antimessage) — механизм обеспечения откатов в Time Warp. Каждое антисообщение соответствует одному ранее созданному сообщению, порождённому в интервале симулируемого времени $[t_{straggler}, LVT_i]$ и вызывает эффект, обратный его обработке (т.е. возвращает состояние в исходное). Поток, обнаруживший прибытие в свою очередь отставшего сообщения, при своём откате рассылает антисообщения всем потокам, с которыми он успел провзаимодействовать, таким образом сообщая о том, что необходимо отменить часть их прямой симуляции, на которую он успел повлиять. Каждый получатель запроса на отмену исполняет его, тем самым уничтожая эффекты от предыдущего события. Если же получатель обнаружит, что антисообщение имеет метку времени меньше его LVT (т.е. оно для него является отставшим), то он также производит откат, рассылая новые антисообщения. Благодаря этому последствия некорректной симуляции постепенно отменяются глобально.
– Сбор окаменелостей (*англ.* fossil collection) — своеобразное название механизма сбора мусора (*англ.* garbage collection). Как будет показано дальше, из всех обработанных всеми потоками событий безопасно удаляемы только те, что имеют метку времени, меньшую чем GVT. Для того, чтобы объём потребляемой при симуляции памяти не рос безгранично, необходимо регулярно её освобождать для последующего переиспользования. Непосредственно сбором окаменелостей может заниматься или специально выделенный для этого поток, или же сами потоки симуляции, для чего их придётся периодически освобождать от задачи обработки событий.

На рис. 6.7 приведены все компоненты схемы Time Warp. Рассмотрим подробнее процессы, происходящие при работе такой модели.

– Каждый поток обрабатывает свою очередь событий, состоящую из сообщений, порождённых как им самим, так и присланными

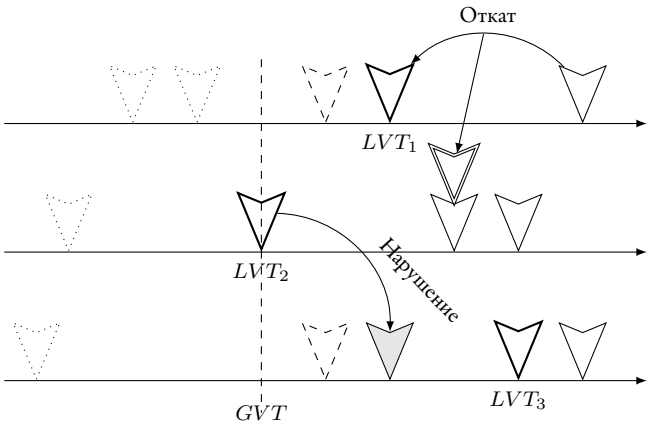


Рис. 6.7. Оптимистичная симуляция Time Warp. Точками обозначены события, подлежащие процессу сборки окаменелостей, штриховыми линиями — уже обработанные, к которым симуляция потенциально может откатиться, жирным — текущее событие, тонкой линией — запланированные в будущем. Серым фоном выделено отставшее сообщение, а двойной линией — антисообщение

остальными потоками, в порядке возрастания меток времени. При этом продвигается значение его LVT, рассылаются новые сообщения, каждое из которых несёт метку времени, когда оно должно быть обработано приёмником.

– Если при обработке очередного сообщения оно оказывается отставшим, получивший его поток начинает процесс отката. При этом он продвигает своё локальное время в обратном направлении, обрабатывая встречающиеся события «наоборот», т.е. вызывая изменения, обратные записанным в них. При этом вместо обычных рассылаются антисообщения. Этот процесс происходит до тех пор, пока исходное отставшее сообщение не перестанет быть таковым, т.е. пока $t_{straggler} < LVT$. После этого события начинают вновь обрабатываться в порядке возрастания LVT.
– Ни один поток по определению не имеет значение времени, меньшее GVT, и ни одно необработанное сообщение (в том числе те, что впоследствии окажутся отставшими) во всей симуляции не может иметь $t^{receive} < GVT$. Из этого следует, что при дальнейшей симуляции ни один поток не будет вынужден откатывать значение своего LVT в прошлое дальше, чем до значения GVT. Таким образом, эта величина имеет принципиальный смысл и обозначает границу между свершившимся и потенциально отменяемым прошлым симуляции. В отличие от последовательной DES и консервативных схем, обработка события в оптимистичной PDES ещё не означает, что занимаемые им ресурсы можно освободить. Это происходит потому, что в дальнейшем в случае отката это же событие могут исполнить ещё раз. Лишь когда оно окажется слева от границы глобального времени, у нас есть гарантия того, что оно «на самом деле свершилось».

6.6. Распределённая общая память

Отдельно стоит упомянуть вопрос, каким способом необходимо организовывать модель памяти, общей для всех моделируемых пото-

ков. В случае, когда симуляция исполняется на одном многопроцессорном узле с общей памятью, все хозяйские потоки могут читать и писать её общие регионы, и аппаратура следит за тем, чтобы у всех хозяйских потоков имелось единое представление о её содержимом. Симулятору остаётся следить за корректным исполнением атомарных операций (см. секцию 6.3.1).

Ситуация усложняется в случае распределения симуляции на несколько узлов, каждый из которых имеет собственную память, и которые объединены с помощью сети передачи данных. В этом случае необходимо использовать доступные отправки и получения сообщений между узлами для того, чтобы все симулируемые потоки имели видимость общей памяти, т.е. необходимо создание представления *распределённой общей памяти* (*англ.* distributed shared memory).

В литературе описаны различные решения данной задачи [5, 11]. Рассмотрим основные приёмы, описанные в них.

– Главная задача системы DSM — поддержка иллюзии единого пространства. Это означает, что все обращения узлов к памяти должны перехватываться и обрабатываться таким образом, чтобы исполняющиеся процессы имели консистентное представление о значениях, хранящихся в ней. С другой стороны, ненулевая вероятность того, что требуемые приложению данные находятся на другом узле, и что для их получения приходится использовать относительно медленный (по сравнению с обращениями к локальной памяти) канал передачи данных, может серьёзно ограничить производительность и масштабируемость всей системы. Таким образом, приходится использовать ухищрения, минимизирующие число таких «дальних» запросов.
– Во-первых, если некоторый регион адресов памяти используется только для чтения (например, это программный код, константы, строки), то каждый из потоков может держать у себя его локальную копию. Таким образом избегается передача сообщений по сети и связанная с ней задержка.
– Во-вторых, если обнаружено, что большую часть времени блок данных читается одним и тем же узлом, то разумным представ-

ляется переместить такие данные непосредственно на этот узел, т.е. провести *миграцию* данных. Возможен и обратный подход — переместить само приложение ближе к данным, т.е. провести миграцию кода [16].

– Наконец, если несколько узлов одновременно пишут и читают один и тот же блок, то неизбежно большинству из них придётся обращаться к нему по сети, тогда как для одного он будет размещён локально.
– Используемая в системе DSM гранулярность хранения регионов памяти влияет на требуемый объём служебной информации и на издержки при передаче изменённых блоков по сети. Как правило, она выбирается равной размеру страницы физической памяти и составляет от 4 кбайт до нескольких мегабайт.

Существуют коммерческие продукты, предназначенные для виртуализации группы вычислительных узлов, соединённых сетью, в представление единой системы с общей памятью и объединённым числом вычислительных ядер [26, 29]. Они позволяют исполнять программы, написанные для систем с единой памятью, без изменений в их коде и логике работы.

Отметим, что, кроме общей оперативной памяти, аналогичные подходы могут использоваться для обеспечения прозрачного распределённого доступа к другим ресурсам, например, энергонезависимому хранилищу, файловой системе [20, 25] и т.д.

6.7. Балансировка скорости отдельных потоков

Параллельная симуляция демонстрирует прирост в скорости по сравнению с последовательной, только если участвующие в ней потоки действительно выполняют полезную работу. Как мы видели раньше, препятствовать этому могут многие факторы. Для консервативных схем это, в первую очередь, блокировка процессов, ожидающих, пока остальные компоненты симуляции продвинутся в будущее. Для оптимистичных моделей — необходимость отката и повторного исполнения для частей симуляции, слишком далеко «забежавших» в

будущее. В обоих случаях соответствующий поток не принимает участия в продвижении глобального состояния модели в будущее. Причиной этому является несбалансированность скоростей работы его самого и взаимодействующих с ним агентов. Можно заметить, что производительность параллельных алгоритмов определяется самым медленным участвующим в них процессом.

Для повышения степени равномерности скоростей потоков вводится механизм *балансировки* нагрузки на них. Способ его осуществления зависит от структуры модели. Наиболее общий принцип — миграция части данных и обрабатывающих их сущностей с сильно нагруженного потока на менее занятый. Например, если в очереди событий первого потока значительно больше событий, чем у второго, разумным представляется передать часть из них так, чтобы после завершения процесса балансировки оба они имели приблизительно равные длины очередей.

Отметим, что на практике практически никогда нет возможности понять, как работа должна быть распределена, заранее, до запуска симуляции. Поэтому разработаны методы динамической балансировки, когда периодически производится оценка эффективности эксплуатации потоков, и на основании результатов принимается решение о переносе части задачи с более нагруженных на менее нагруженные потоки [23].

6.8. Барьерная синхронизация

Как было показано ранее, консервативная симуляция может «самопроизвольно» приходить в состояние глобальной взаимоблокировки, в котором ни один из потоков не может безопасным образом обрабатывать события своей очереди без риска нарушить каузальность системы. Таким образом, симуляция при этом состоит из двух чередующихся фаз: обработка событий одним или более потоками и взаимоблокировка в ожидании сигнала к продолжению.

Некоторые протоколы симуляции подразумевают явное приведение симуляции к полной остановке в известные моменты времени — достижение *барьера* синхронизации. Каждый из участвующих в

работе потоков, дойдя до определённой точки в своей работе, останавливается, ожидая сигнала от остальных, что они также достигли этой точки. После того, как все сообщения получены, он возобновляет свою работу до следующего барьера.

При этом периодически возникает «стабильное» состояние всей системы, в котором известны состояния всех участников, в том числе их локальные значения виртуальных времён[1]. Это позволяет определить, какие из событий во всей симуляции безопасно обрабатывать. При этом значение величины предпросмотра непосредственно определяет расстояние между соседними барьерами.

К сожалению, несмотря на более понятную структуру алгоритмов, барьерные схемы могут страдать от двух проблем, негативно влияющих на их производительность.

1. Низкая степень утилизации вычислительных ресурсов. На рис. 6.8 показано, что длительность каждой фазы исполнения определяется самым медленным из потоков. Остальные участники, выполнив свою долю работы, вынуждены при этом простаивать до конца фазы. Проблема усиливается при несбалансированности вычислительной нагрузки на потоки. Она менее заметна, если темп их работы приблизительно одинаков.
2. Необходимость передачи большого числа сообщений для определения момента достижения барьера может занимать значительную долю времени работы алгоритма. С ростом числа потоков это обстоятельство может уничтожить весь выигрыш от параллелизма, т.к. почти всё время будет тратиться на обслуживание барьера.

6.9. Детерминизм параллельных моделей

Детерминистичный (*англ.* deterministic) симулятор — модель, которая в независимых своих запусках вычисляет идентичное конечное

[1]Более точно, известно, что ни один поток не отстал в прошлое на неопределённое число шагов, т.к. все они уже достигли барьера.

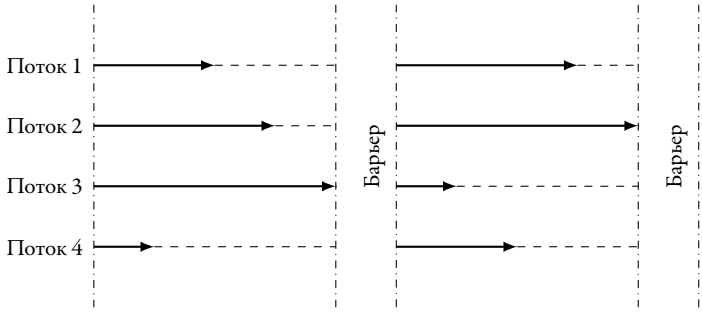


Рис. 6.8. Симуляция с барьерной синхронизацией. Жирными сплошными линиями показаны активные участки симуляции, штриховыми — ожидание остальных потоков без выполнения полезной работы

состояние моделируемой системы для любого начального состояния при условии, что исходное состояние при каждом запуске было одинаковое. Такая программа будет повторять своё поведение при последовательных запусках на каждом своём шаге, таким образом проходя через одну и ту же последовательность состояний.

Проявление детерминизма в поведении при исследовании целевой системы является чрезвычайно полезным свойством. Так, значительно упрощается отладка гостевых приложений, потому что становится возможным установить момент, в который в них возникает ошибочное поведение, и изучить его причины. Если записана история эволюции состояния моделируемой системы, то, изменяя состояние в обратном порядке, можно создать видимость «обращения» течения времени (см. главу 8, секцию 8.8.2).

6.9.1. Условия детерминизма

Для последовательного симулятора DES для повторяемости исполнений достаточно соблюдения следующих условий.

1. Он написан без ошибок типа «обращение к неинициализированной памяти». При нарушении этого условия сложно утверждать даже о корректности симуляции, не говоря уже об её детерминистичности.

2. Все обработчики событий ведут себя повторяемым образом. Результат их исполнения не должен зависеть от работы генератора случайных чисел, состояния хозяйских файлов, изменяемых сторонними программами, сетевых пакетов реальной сети, интерактивного вмешательства пользователя и т.п. факторов.
3. Обработка событий из очереди производится в одном и том же порядке в каждом запуске симулятора. Для событий к несовпадающими метками времени он диктуется корректностью самого алгоритма. В случае совпадения меток у двух или более событий необходимо упорядочить их повторяемым между отдельными запусками способом. Поскольку последовательная программа создаёт такие события одно за другим, естественный способ состоит в том, чтобы использовать детерминированный порядок их создания для расстановки приоритетов при обработке в случае совпадения меток времени.

Последовательная симуляция общего назначения, не использующая в свой работе вероятностные механизмы (метод Монте Карло и т.п.), должна обеспечивать детерминизм, иначе возникают законные вопросы об её корректности.

Замечание. *Необходимо отметить, что в окружающем нас мире детерминизм не такое частое явление. Механистическая картина мира, в которой частицы движутся по законам Ньютона, не в силах сойти в предопределённой им начальным состоянием траектории, оказалась неприменимой в масштабах микро-, а значит и макромира. На смену ей пришла квантовая механика [32]. По этой причине следует понимать, что модели, описываемые в данной книге, имеют далеко не универсальную применимость.*

Кроме того, как будет показано дальше, выполнение условий для повторяемости параллельной симуляции негативным образом влияет на её производительность, так как требует дополнительной синхронизации потоков. Наконец, ограничивая модель многоагентной системы в её поведении только одним сценарием из множества реально допустимых, мы лишаем себя возможности наблюдать их проявления и как-то повлиять на них. Например, последовательная модель многопроцессорной

системы часто неспособна воспроизвести проблемы гонок данных в программах, наблюдаемых на реальной аппаратуре.

По этим причинам нельзя с определённостью заявлять, что детерминизм для параллельной симуляции является обязательным или приоритетным свойством во всех случаях.

6.9.2. События с одинаковой меткой времени

Для параллельной симуляции ситуация несколько сложнее. Три сформулированные выше условия должны выполняться для каждого участвующего в ней потока. Возникает новой источник неопределённости — события с одинаковой меткой времени могут приходить с сообщениями от других потоков, при этом из-за различий в скоростях их исполнения в разных запусках порядок создания может быть различным (рис. 6.9).

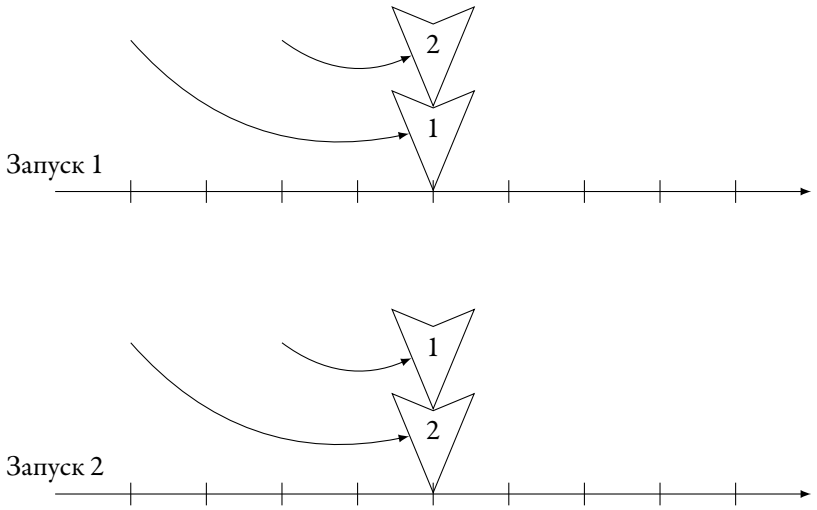


Рис. 6.9. Различный порядок получения сообщений от внешних потоков

Каким образом можно детерминистичным образом упорядочить обработку событий в данных условиях?

Определённый порядок получения событий. Можно попытаться

получать события от внешних потоков в строго фиксированном порядке. Например, в модели, содержащей три процесса, первый всегда ожидает сообщения от второго потока, прежде чем перейти к получению данных от третьего. Это устраняет указанный выше источник неопределённости, однако накладывает серьёзные ограничения на характер коммуникаций в системе, т.к. неосторожное использование операции блокирующего чтения может привести к взаимоблокировке.

Расширение метки времени дополнительными битами точности. В данном подходе принимаются меры, чтобы времена событий никогда полностью не совпадали. Для этого метка времени должна содержать дополнительные «младшие» биты, уникальные для всех событий во всей симуляции. При этом для любых двух симулируемых событий расширенные таким образом метки времени никогда не могут совпасть. Уникальные младшие биты могут формироваться на основе двух чисел: порядкового номера создаваемого события, монотонно возрастающего для каждого потока-отправителя, и порядкового номера создающего его хозяйского потока. Альтернативный подход заключается в использовании генератора псевдослучайных чисел, инициируемого одним и тем же начальным значением так, что производимая им последовательность одинакова при всех запусках симуляции[1].

6.9.3. Домены синхронизации

Рассмотрим пример использования консервативного алгоритма с барьерной синхронизацией, применимого в ситуациях, когда в симуляции можно выделить области, характерная частота коммуникаций между которыми превышает частоту коммуникаций внутри каждой из них, а доставка сообщений может быть задержана до барьера. Такие области обозначаются как *домены синхронизации* [27]. Необходи-

[1]Отметим, что построение генератора, пригодного для детерминистичной многопоточной генерации псевдослучайных чисел, нетривиально [18].

мый квант синхронизации для сообщений между доменами как пустых, так и информационных между устройствами выбирается равным этой задержке. Для подобной схемы организации должны выполняться следующие условия.

1. Все домены могут работать параллельно и независимо до тех пор, пока не достигнут границы текущей квоты симулируемого времени, где они останавливаются на синхронизационном барьере, ожидая, пока все остальные домены также достигнут его.
2. Устройства, требующие частых коммуникаций или зависящие от точного моделирования вариаций длительностей задержек, не могут быть разнесены в разные домены без дополнительных ухищрений. Например, информация об изменениях в общей памяти всегда будет задерживаться от момента записи до момента междоменной синхронизации, когда может оказаться, что в симуляции произошло каузальное нарушение. В таком случае остаётся использовать откат некоторых или всех конфликтующих потоков с их повторным исполнением до тех пор, пока все потоки не достигнут барьера непротиворечивым способом. Т.е. приходится применять оптимистичный подход.

На практике при симуляции многомашинных конфигураций в составе одного домена размещается один гостевой компьютер, а моделирование передачи сетевых пакетов (Ethernet, Infiniband и т.п.) производится через описанную схему. При этом достигается наилучший баланс между скоростью симуляции (т.к. характерные задержки сетевых устройств больше, чем наблюдаемые между, скажем, памятью и процессором) и её точностью (протоколы сетевого взаимодействия толерантны к большим задержкам и высокой неопределённости времени доставки пакетов). Однако с ростом скоростей (для современных серверных и HPC сетевых карт скорость доставки пакетов приближается к скорости работы локальных жёстких дисков) квант синхронизации уменьшается, что приводит к повышению частоты организации барьеров с соответствующим падением скорости симуляции.

Описанная система доменов может быть вложенной, т.е. внутри одного домена определены несколько меньших, синхронизация кото-

рых также происходит с фиксированной задержкой. При этом интервал синхронизации поддоменов должен быть короче используемого для их внешнего, «родительского» домена.

Поскольку порядок передачи сообщений между потоками симулятора при достижении барьера контролируется центральным агентом, он достаточно просто может проводить его повторяемым между симуляциями образом, что позволяет сделать доменную схему детерминистичной.

6.10. Параллельная симуляция одного процессора

У наблюдательного читателя может возникнуть вопрос об «обратной» ситуации: можно ли получить выигрыш от многих ядер хозяйской системы, если требумое число симулируемых ядер при этом значительно меньше? То есть — возможно ли каким-то образом ускорить симуляцию одного гостевого процессора, используя для этого несколько параллельных хозяйских потоков?

Ответ зависит от того, на каком уровне абстракции лежит модель. Для уровня архитектуры (набора инструкций) ответ почти всегда отрицательный. Эффективное решение этой задачи — параллельное исполнение нескольких машинных инструкций последовательной программы — равносильно построению *быстрой* модели внеочередного исполнения (*англ.* out of order execution), способной «на лету» находить независимые по данным инструкции и исполнять их. Весь эффект от параллельного исполнения при этом будет нивелирован длительным анализом. Если даже упростить задачу, позволив проводить статический, а не динамический анализ, мы сводим её к проблеме построения автопараллелизующего компилятора. Как известно, далеко не всякий код подвержен такому преобразованию.

Значительный потенциал к параллелизации модели процессора возникает при переходе к потактовым моделям. На уровне симуляции представления синхронной цифровой схемы, состоящей из множества узлов, каждый шаг выполняющих свои фиксированные функции на данных, полученных с предыдущего шага работы, становится

возможным запускать большое число компонент схемы параллельно. При этом меняются принципы организации симуляторов для таких моделей. Мы рассмотрим их подробнее в главе 7.

6.11. Заключение

В данной главе были обрисованы лишь основные принципы организации симуляции, использующей более одного потока хозяйской системы. Конечно, за десятки лет исследований в этой области были созданы многочисленные варианты многопроцессорных, параллельных и распределённых решений. За границами данного описания остались многие важные вопросы, в том числе следующие.

- Оптимизации, используемые для минимизации числа необходимых синхронизаций в консервативных моделях.
- Решения, позволяющие уменьшить частоту и стоимость откатов в схемах оптимистичных.
- Потребление памяти параллельным симулятором по сравнению с последовательной версией.
- Построение консервативных и оптимистичных схем с нулевым предпросмотром, и проблематика обеспечения детерминизма для этого случая.
- Эффективное взаимодействие планировщика потоков хозяйской системы и симуляции.
- Проблема учёта событий, уже отправленных, но ещё не полученных, т.е. находящихся «в полёте».
- Способы эффективного вычисления GVT, организации барьеров, создания точек сохранения и другие распределённые алгоритмы, необходимые для нормальной работы параллельного симулятора.

Интересующийся читатель может найти подробное описание этих и других вопросов распределённой симуляции в [9]. Тем не менее, хотелось бы ответить на два важных вопроса, связанных с основной темой данной главы: почему именно параллельный симулятор так слож-

но построить? и если он таки построен, то возможно ли получить выигрыш в скорости?

Почему параллельная симуляция настолько сложна? В работе [10] в секции «Why PDES is hard?» приводится достаточно краткий и одновременно подробный ответ на этот вопрос. Приведём свободный перевод: «*...необходимо определить, может или нет сообщение E_1 быть обработано одновременно с E_2. Но каким образом узнать, влияет или нет E_1 на E_2, без его симуляции?*»

Переформулируем это высказывание следующим образом: в отличие от других параллельных приложений, в которых зависимости между потоками статически определены и известны ещё до стадии исполнения, алгоритмы симуляции не определяют своё поведение полностью. Значительная часть его содержится в подлежащей симуляции программе, про которую в общем случае ничего не известно.

Потолок ускорения от параллелизма. Верхнюю границу значения ускорения, получаемого в случае использования параллельного симулятора вместо последовательного, устанавливает степень параллелизма, проявляемого гостевым приложением. Никакой параллельный симулятор не будет в состоянии ускорить код, каждый этап исполнения которого зависит от предыдущего. Поэтому, прежде чем вкладывать усилия в трудоёмкий процесс написания параллельных моделей, необходимо определить, способен ли целевой класс гостевых приложений использовать весь создаваемый потенциал.

6.12. Вопросы к главе 6

TODO Расширить и переработать

Вариант 1

1. Какие из типов схем PDES позволяют добиться детерминизма симуляции?
    a) Барьерная (с доменами синхронизации).

b) Консервативная.
c) Оптимистичная.
d) Наивная.

2. Чем чревата излишне частая отправка пустых (null) сообщений в консервативной схеме PDES с детектированием взаимоблокировок?
3. Выберите правильные продолжения фразы: Частая отправка пустых (null) сообщений нежелательна, так как

   a) это может ограничивать скорость симуляции,
   b) это может вызвать нарушение каузальности симуляции,
   c) это может привести к взаимоблокировке потоков,
   d) это может привести к переполнению очередей сообщений.

4. Выберите правильные ответы.

   a) Симуляция, реализованная с помощью схемы PDES, всегда детерминистична.
   b) Симуляция, реализованная с помощью схемы PDES, недетерминистична из-за возможности потери сообщений между потоками.
   c) Симуляция, реализованная с помощью схемы PDES, недетерминистична из-за возможности блокировки отдельных потоков.
   d) Симуляция, реализованная с помощью схемы PDES, недетерминистична из-за варьирующейся скорости работы отдельных потоков.

Вариант 2

1. Почему не будет работать **наивная** схема параллельного DES? Выберите верные ответы.

   a) Недетерминизм модели.
   b) Невозможно организовать передачу сообщений между потоками.
   c) Возможно нарушение каузальности.
   d) Невозможно подобрать точно квоту выполнения.

2. Чем чревата недостаточно частая отправка пустых (null) сообщений в консервативной схеме PDES с детектированием взаимоблокировок?
3. Выберите правильные ответы.
    a) Консервативные схемы PDES не допускают нарушения каузальности.
    b) Консервативные схемы PDES допускают нарушения каузальности.
    c) Консервативные схемы PDES допускают нарушения каузальности, но впоследствии их корректируют.
    d) Оптимистичные схемы PDES не допускают нарушения каузальности.
    e) Оптимистичные схемы PDES допускают нарушения каузальности.
    f) Оптимистичные схемы PDES допускают нарушения каузальности, но впоследствии их корректируют.
4. Выберите правильные свойства домена синхронизации в модели PDES.
    a) Количество моделируемых устройств внутри одного домена фиксировано.
    b) Не происходит взаимодействия устройств, находящихся в различных доменах.
    c) Количество моделируемых устройств внутри одного домена ограничено.
    d) Характерная частота коммуникаций между доменами превышает частоту коммуникаций внутри каждого.
    e) Характерная частота коммуникаций между доменами равна частоте коммуникаций внутри отдельного домена.

Литература

1. A Formal Specification of Intel® Itanium® Processor Family Memory Ordering. — Intel Corporation, окт. 2002. — URL: http://download.intel.com/design/itanium/downloads/25142901.pdf (дата обр. 03.09.2013).
2. *Adve Sarita V.*, *Gharachorloo Kourosh* Shared memory consistency models: A tutorial // IEEE Computer. — 1996. — Т. 29. — С. 66—76. — DOI: 10.1.1.106.5742. — URL: http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-95-7.pdf (дата обр. 02.09.2013).
3. *Andrews Greg R* Foundations of Parallel and Distributed Programming. — 1st. — Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. — ISBN: 0201357526.
4. ARM Synchronization Primitives Development Article. Appendix A. SWP and SWPB. — ARM. — URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dht0008a/CJHIGEFG.html (дата обр. 01.09.2013).
5. *Bugnion Edouard*, *Devine Scott*, *Rosenblum Mendel* Disco: Running commodity operating systems on scalable multiprocessors // ACM Transactions on Computer Systems. — 1997. — 143–156. — URL: http://www.cis.upenn.edu/~cis700-6/04f/papers/bugnion-disco.pdf.
6. COREMU: a scalable and portable parallel full-system emulator / Zhaoguo Wang [и др.] // Proceedings of the 16th ACM symposium on Principles and practice of parallel programming. — San Antonio, TX, USA: ACM, 2011. — 213–222. — (PPoPP '11). — ISBN: 978-1-4503-0119-0. — DOI: 10.1145/1941553.1941583. — URL: http://doi.acm.org/10.1145/1941553.1941583.

7. *Dechev Damian*, *Pirkelbauer Peter*, *Stroustrup Bjarne* Understanding and Effectively Preventing the ABA Problem in Descriptor-Based Lock-Free Designs // 2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC). — 2010. — 185–192. — ISSN: 1555-0885. — DOI: 10.1109/ISORC.2010.10. — URL: http://www.stroustrup.com/isorc2010.pdf (дата обр. 01.09.2013).

8. *Ferscha Alois* Parallel and Distributed Simulation of Discrete Event Systems. — 1995. — URL: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=0D07CB3110B890F247E12489F3264E62?doi=10.1.1.19.6226&rep=rep1&type=pdf (дата обр. 21.09.2013).

9. *Fujimoto Richard M.* Parallel and Distributed Simulation Systems. — 1st. — New York, NY, USA: John Wiley & Sons, Inc., 2000. — ISBN: 0471183830.

10. *Fujimoto Richard M.* Parallel discrete event simulation // Commun. ACM. — 1990. — Окт. — Т. 33, № 10. — С. 30—53. — ISSN: 0001-0782. — DOI: 10.1145/84537.84545. — URL: http://doi.acm.org/10.1145/84537.84545.

11. Graphite: A Distributed Parallel Simulator for Multicores / Jason E. Miller [и др.] // The 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA). — 2010. — Янв.

12. *Herlihy Maurice* Wait-free synchronization // ACM Trans. Program. Lang. Syst. — 1991. — Янв. — Т. 13, № 1. — С. 124—149. — ISSN: 0164-0925. — DOI: 10.1145/114005.102808. — URL: http://doi.acm.org/10.1145/114005.102808.

13. *Herlihy Maurice*, *Shavit Nir* The Art of Multiprocessor Programming. — San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. — ISBN: 0123705916, 9780123705914.

14. *Jefferson David R.* Virtual Time // ACM Transactions on Programming Languages and Systems. — 1985. — Т. 7, № 3. — С. 404—425.

15. *Jensen Eric H.*, *Hagensen Gary W.*, *Broughton Jeffrey M.* A New Approach to Exclusive Data Access in Shared Memory Multiprocessors //. — 1987. — Ноя. — Technical Report UCRL-97663. — URL: https://e-reports-ext.llnl.gov/pdf/212157.pdf (дата обр. 01.09.2013).

16. *Khan Omer*, *Lis Mieszko*, *Devadas Srinivas* Instruction-Level Execution Migration. — Тех. отч. MIT-CSAIL-TR-2010-019. — 2010. — URL: http://dspace.mit.edu/bitstream/handle/1721.1/53748/MIT-CSAIL-TR-2010-019.pdf (дата обр. 08.09.2013).

17. *Lantz Robert E.* Parallel SimOS: Performance and Scalability for Large System Simulation. — Докт. дисс. Computer Systems Laboratory, Stanford University, 2007. — URL: http://cs.stanford.edu/~rlantz/papers/lantz-thesis.pdf (дата обр. 26.03.2012) ; Ph.D. Dissertation.

18. *Leiserson Charles E.*, *Schardl Tao B.*, *Sukha Jim* Deterministic parallel random-number generation for dynamic-multithreading platforms // SIGPLAN Not. — 2012. — Фев. — Т. 47, № 8. — С. 193—204. — ISSN: 0362-1340. — DOI: 10.1145/2370036.2145841. — URL: http://doi.acm.org/10.1145/2370036.2145841.

19. *Liu Jason* Parallel Discrete-Event Simulation. — 2009. — URL: http://www.cis.fiu.edu/~liux/research/papers/pdes-eorms09.pdf (дата обр. 26.03.2012).

20. LustreFS. — Xyratex, 2013. — URL: http://lustre.org/ (дата обр. 08.09.2013).

21. *Misra Jayadev* Distributed discrete-event simulation // ACM Computing Surveys. — 1986. — Т. 18. — 39–65. — URL: http://www.cis.udel.edu/~cshen/861/papers/p39-misra.pdf.

22. *Mosberger David* Memory Consistency Models. — 1993. — URL: http://citeseerx.ist.psu.edu/viewdoc/download;?doi=10.1.1.44.5376.

23. *Peschlow Patrick*, *Honecker Tobias*, *Martini Peter* A Flexible Dynamic Partitioning Algorithm for Optimistic Distributed Simulation // PADS. — IEEE Computer Society, 2007. — С. 219—228. — ISBN: 0-7695-2898-8. — (Дата обр. 01.09.2013).

24. 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.

25. *Schmuck Frank*, *Haskin Roger* GPFS: A Shared-Disk File System for Large Computing Clusters // In Proceedings of the 2002 Conference on File and Storage Technologies (FAST. — 2002. — С. 231—244. — URL: https://www.cct.lsu.edu/~kosar/csc7700-fall06/papers/Schmuck02.pdf (дата обр. 08.09.2013).

26. SGI UV: Big Brain for No-Limit Computing. — SGI, 2013. — URL: http://www.sgi.com/products/servers/uv/ (дата обр. 03.09.2013).

27. Simics Accelerator User's Guide 4.6. — Wind River, 2011.

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

29. vSMP Foundation Free. — ScaleMP, 2013. — URL: http://www.scalemp.com/products/vsmp-foundation-free/ (дата обр. 01.09.2013).

30. *В.В. Топорков* Модели распределенных вычислений. — Физматлит, 2004. — ISBN: 5-9221-0495-0.

31. *Зайцев Роман* Atomic operations. — 2012. — URL: http://habrahabr.ru/post/157163/ (дата обр. 01.09.2013).

32. *М.Г. Иванов* Как понимать квантовую механику. — РХД, 2012. — ISBN: 978-5-93972-944-4. — URL: http://mezhpr.fizteh.ru/biblio/q-ivanov.html (дата обр. 18.09.2013).

7. Потактовая симуляция

Вот эту руку — сюда, эту — сюда,
Ногу вот так.
Вот эту голову так, смотри на меня,
Двигайся в такт.

Враги — Танго

7.1. Мотивация к созданию ещё одного подхода к моделированию систем

В предыдущих главах мы рассмотрели сценарии моделирования, которые условно можно классифицировать по соотношению количества независимых агентов в системе и частоте возникновения событий.

1. Одно устройство; события на каждом шаге симуляции. Характерная ситуация для моделирования процессора.
2. Много устройств; события возникают редко (значительно реже, чем на каждом такте). Ситуация возникает при симуляции дискретных событий (DES).
3. Мало устройств; мало событий. При использовании аналитических методов (QSP) мы имеем максимально простую модель системы, не требующую компьютерной симуляции.

В данной классификации не хватает последней, четвёртой, возможности.

4. Много устройств; события на каждом такте.

В действительности такая ситуация возникает в системах, созданных для потактового моделирования цифровых схем. Они наиболее точно отражают процессы, происходящие в аппаратуре, и пото-

му каждый внутренний блок некоторого кристалла имеет своё отображение на часть модели. Принципиально ничто не мешает использовать схему DES и здесь — она будет корректно работать, однако не будет давать наилучшую визуализацию процессов, происходящих в системе, — симулятор вынужден постоянно обрабатывать длинные очереди сообщений, привязанных к одному симулируемому такту (рис. 7.1), тогда как чаще всего от такой модели требуется выдавать информацию о потоках данных между узлами, детали их коммуникаций. Кроме того, в этом случае DES не позволяет построить эффективный симулятор с точки зрения скорости исполнения.

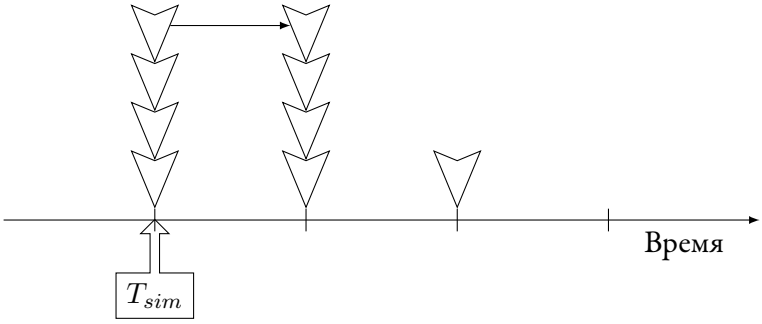


Рис. 7.1. Потактовая симуляция с DES. Длинные цепочки событий, ожидающих обработки на каждом такте, замедляют симуляцию. Кроме того, это усложняет понимание связей между происходящими в системе процессами

Современные цифровые схемы являются синхронными — темп вычислений задаётся единым тактовым генератором. Состояния всех субъединиц изменяются одновременно по его сигналу. Кроме того, результаты вычислений с текущего такта работы некоторого узла не будут переданы/доступны получателю до наступления следующего такта. Необходимо отразить эти факты при моделировании.

7.2. Сложности моделирования

При проектировании схемы работы симулятора необходимо учесть ряд дополнительных усложняющих факторов (рис. 7.2).

– Узлы реальной схемы работают параллельно, тогда как симулятор должен допускать последовательную реализацию, в которой одновременные события будут обрабатываться в некотором порядке.
– Субмодули могут иметь много входов и выходов, соединённых сложным образом.
– Выдаваемые узлом данные часто меняют состояние не только последующих за ним, но и предыдущих узлов (и даже самого себя, как будет показано далее).
– Вычисление некоторых операций может составлять несколько тактов; результат их вычислений выдаётся с задержкой по отношению к моменту прибытия входных данных.
– Узлы часто состоят из более мелких единиц, которые потребуется симулировать на поздних стадиях исследования при уточнении модели.

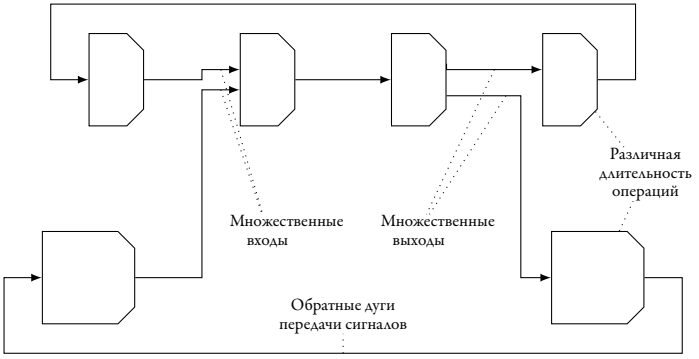


Рис. 7.2. Пример соединения узлов в некоторой синхронной цифровой схеме

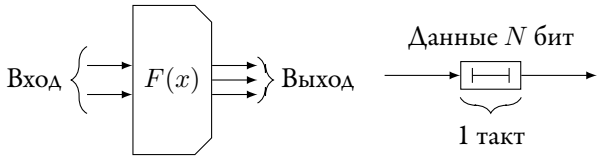
Основная проблема здесь — как обеспечить изоляцию данных между тактами? При непосредственном соединении модулей то, какие данные будут на входе устройства, зависит от того, отработал ли предыдущий в цепочке соединения узел свой текущий такт, и на его выходе уже новое значение, или нет, и тогда значения относятся к предыдущему такту. Корректность можно обеспечить установкой по-

рядка вызова, но при больших масштабах и частых изменениях модели за этим становится сложно следить, что приводит к существенным сложностям поддержки модели и массе трудноуловимых ошибок.

7.3. Схема симуляции

Необходим масштабируемый и достаточно простой подход. Решение заключается в изначальном отделении временных аспектов от функциональных, а также вынесения состояния системы контролируемым и иерархическим образом.

- Функции узлов являются функциями в математическом смысле — они дают результат мгновенно, без побочных эффектов, и результат зависит только от входных данных (рис. 7.3a). При наличии у узла множественных входов или выходов они объединяются в один логический с соответствующей суммарной шириной в битах.
- Время, затрачиваемое на выполнение операции, представлено в виде устройства линии задержки с фиксированной длиной и пропускной способностью (рис. 7.3b).
- Состояние узлов не хранится внутри блоков (подробнее — в секции 7.4.4).



a) Функциональный блок

b) Порт — линия задержки

Рис. 7.3. Два класса элементов, используемых при тактовой симуляции

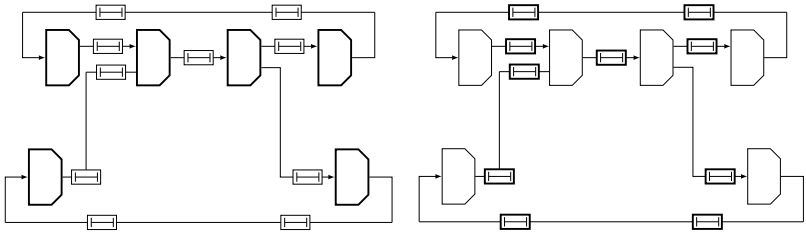
Моделирование в потактовых моделях ненулевой длительности передачи сигналов линиями задержки широко принято, эта абстракция имеет название *порты* [2].

7.3.1. Алгоритм работы

Главное правило соединения двух типов элементов состоит в том, что нельзя напрямую соединять два функциональных элемента — между ними должен находиться как минимум один порт. Каждый такт симуляции состоит из двух стадий, в течение которых изменяется состояние только одного класса объектов. Таким образом, они изолированы друг от друга.

Вычисление функций. Вычисляются все функциональные блоки (рис. 7.4a). Результаты вычислений помещаются на их выходы. Порядок обхода блоков и вызова их функций неважен, т.к. на этой стадии каждый из них отделён от результатов работы остальных с помощью портов.

Передача данных. Активны объекты портов. При этом каждый переносит значение со своего входа на свой выход (рис. 7.4b). Как и для первой стадии, порядок обхода неважен из-за взаимной изоляции портов.



a) Вычисление функциональных блоков

b) Передача данных портами

Рис. 7.4. Две фазы работы симуляции с использованием портов для одного симулируемого такта

7.4. Замечания к реализации схемы

Обозначенный в этой главе подход является достаточно универсальным, чтобы быть использованным для построения потактовых моделей. Рассмотрим теперь некоторые дополнительные детали, полезные для практической реализации.

7.4.1. Готовность данных

Как уже было отмечено, длительность операций может превышать один такт; в таком случае на выходе соответствующего порта не должно быть выходного результата в течение некоторого времени. Тем не менее каждый порт читает значение на своём входе и передаёт его на выход на каждом такте, не анализируя, корректно ли значение. Для унификации указанных двух ситуаций к входным и выходным данным добавляется ещё один бит «данные валидны» (рис. 7.5). Функциональные блоки имеют доступ к этому биту и могут помечать выходное значение как правильное или как пустое в зависимости от реализуемой в них логики.

Рис. 7.5. Бит валидности. Если он равен нулю, значение, передаваемое в поле данных, не имеет смысла

Отметим, что такой подход напоминает реализацию, используемую в реальной аппаратуре, например, при обращении к медленной памяти: после подачи запроса на шину адреса считывание результата с

шины данных не будет производиться до тех пор, пока на отдельном контакте готовности не появится соответствующий сигнал.

7.4.2. Латентность и пропускная способность портов

На рис. 7.6 показано, как выражаются понятия латентности λ (такт) и пропускной способности X (бит/такт) некоторого узла в модели, использующей порты. Обработка одного блока информации целиком требует количество тактов, равное числу последовательных узлов. Однако одновременно в обработке на разных стадиях может находиться более одного такого блока, и за каждый такт количество выдаваемых данных определяется шириной портов.

L = 5 тактов

Рис. 7.6. Различие между латентностью и пропускной способностью порта. Тёмными блоками обозначены валидные данные. В одной линии задержки могут одновременно находиться несколько транзакций на различных стадиях своего пути от отправителя к получателю

Используя порты, конструкторы новых систем могут изучать изменения в производительности моделируемого устройства в зависимости от длин задержек, для этого им достаточно варьировать параметры соответствующих линий.

Однако возникает проблема: корректно ли соединять порты непосредственно друг с другом? Ведь это нарушает принцип изолированности стадий симуляции. Можно предложить три способа решения данной проблемы.

1. Разделять однотактовые порты простыми функциональными устройствами — повторителями сигнала. В данном случае мы запрещаем непосредственное соединение портов так же, как это сделано для функциональных элементов.

2. В отличие от функциональных элементов, порты всегда имеют ровно один вход и выход и потому могут быть относительно легко упорядочены внутри группы при последовательном соединении и соответсвенно симулироваться в порядке, приводящем к правильному потоку данных.
3. Можно обеспечивать длинные задержки не однотактными портами, а реализовать многотактный вариант, имеющий внутреннее состояние и самостоятельно следящий за всеми транзакциями, находящимися внутри него.

Замечание. *В физике для описания волновых процессов [5] существуют понятия «фазовой скорости» для движения фазового фронта (поверхности постоянной фазы) и «групповой скорости» для максимума амплитудной огибающей квазимонохроматического волнового пакета. Оба они характеризуют один периодический процесс, однако относятся к разным типам возмущения. Проводя аналогию между этими понятиями и пропускной способностью с латентностью*[1] *некоторой системы, можно сказать, что первая величина характеризует темп обработки в установившемся, непрерывном режиме подачи данных на вход, тогда как вторая характеризует реакцию системы на внезапное изменение внешних условий.*

7.4.3. Композиция узлов

На рис. 7.7 показано, каким образом можно скрывать части симулируемой системы внутри одного блока на примере последовательного соединения. Если для нас не представляют интереса внутренние процессы, мы можем заменить несколько мелких блоков одним, выполняющим их функцию. При этом его задержка будет равна суммарной длине цепочки портов исходной системы.

7.4.4. Хранение состояния узлов

В предложенном ранее дизайне функционального элемента не предусмотрена возможность хранения им внутреннего состояния.

[1] *Точнее, с величиной, обратной латентности* — $\frac{1}{\lambda}$, *имеющей размерность частоты.*

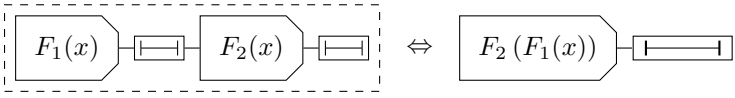


Рис. 7.7. Использование композиции устройств в потактовой модели, построенной с помощью портов. Два последовательно соединённых узла можно заменить одним, совмещающим функции обоих. Порты заменяются одним с задержкой, равной сумме исходных

Однако оно необходимо для моделирования многих устройств, например триггеров, регистров, кэшей. Изящное решение заключается в вынесении внутреннего состояния на текущем такте как части выходного результата и передачи его на вход того же устройства через порт на следующий такт (рис. 7.8).

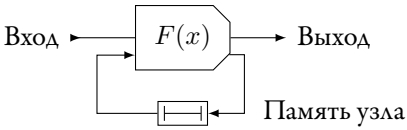


Рис. 7.8. Хранение состояния узла с помощью задержки на один такт

7.5. Реализация потактовых моделей на микросхемах FPGA

Из-за необходимости симулирования в потактовых моделях большого числа деталей поведения реальной системы скорость работы программной реализации на классических ЭВМ крайне низка — замедление относительно прямого исполнения может достигать сотен тысяч раз. Поэтому довольно популярным является решение, при котором модель переносится на специальный ускоритель — вычислительное устройство на основе FPGA[1]. Их особенностью является высокая частота внутреннего тактового генератора и возможность реа-

[1] *англ.* Field Programmable Gate Array, что приблизительно соответствует понятию ПЛИС — программируемые логические интегральные схемы.

лизации модели с высокой степенью параллельности — как уже было описано ранее, внутри своей фазы симуляции отдельные функциональные узлы и порты могут исполняться независимо друг от друга. Недостатком метода является относительная сложность программирования этих устройств, а также их высокая цена. Кроме того, масштабы моделей некоторых устройств могут оказаться таковы, что их затруднительно разместить целиком на одной плате с FPGA-чипом; при этом приходится идти на всевозможные ухищрения.

Примеры использования FPGA для потактовых моделей можно найти в работах [1, 3].

7.6. Взаимодействие функциональной и потактовой моделей

Создание потактового симулятора некоторого устройства часто начинается после того, как в наличии имеется его рабочий функциональный вариант модели. При этом желательно переиспользовать часть функциональности для того, чтобы уменьшить объём работы, а также избавиться от необходимости его отлаживать. Поэтому разумным решением является написание потактовой части как *модели задержек* (*англ.* timing model), отвечающей лишь за определение того, сколько времени займёт на аппаратуре тот или иной процесс, например исполнение очередной инструкции. Тем, какая это будет инструкция и какие связанные с ней архитектурные эффекты будут наблюдаться, т.е. декодирование и исполнение, занимается функциональная модель.

7.7. Заключительные замечания

Пример открытой реализации симулятора, использующего концепцию портов, можно найти на странице учебно-исследовательского проекта MDSP [4].

7.8. Вопросы к главе 7

Вариант 1

1. Выберите правильные варианты ответов:
    a) функциональные модули не имеют внутреннего состояния,
    b) функциональные модули могут иметь внутреннее состояние,
    c) функциональные модули всегда имеют внутреннее состояние.
2. Выберите правильные варианты ответов:
    a) ширина входа и выхода порта должны быть равны,
    b) ширина входа и выхода порта могут различаться,
    c) количество выходов функционального элемента должно быть равно единице,
    d) количество входов и выходов функционального элемента должно совпадать.
3. Выберите правильные варианты продолжения фразы: процесс исполнения потактовой модели на основе портов
    a) всегда содержит две фазы, которые обязаны чередоваться,
    b) всегда содержит две фазы, порядок которых не фиксированный,
    c) всегда содержит одну фазу, в течение которой работают все субъединицы,
    d) может содержать более двух чередующихся фаз.

Вариант 2

1. Выберите правильные варианты ответов:
    a) при передаче данных порты не сохраняют бит валидности данных,
    b) при передаче данных порты не сохраняют бит валидности данных, только если он снят,

c) при передаче данных порты не сохраняют бит валидности данных, только если он поднят,
d) при передаче данных порты сохраняют бит валидности данных.

2. Выберите правильные варианты продолжения фразы: внутри исполнения фазы функциональных элементов потактовой модели на основе портов

a) первыми должны исполняться функции, расположенные в графе правее,
b) порядок исполнения функций устройств неважен,
c) первыми должны исполняться функции, расположенные в графе левее.

3. Выберите правильные варианты продолжения фразы: в модели, описанной на основе портов,

a) функциональные модули имеют различные задержки выполнения,
b) функциональные модули не имеют определённой задержки выполнения,
c) функция портов является функцией тождественности, а задержка нулевая,
d) функция портов является функцией тождественности, а задержка ненулевая,
e) функция портов не является функцией тождественности, а задержка нулевая.

Литература

1. A-Ports: an efficient abstraction for cycle-accurate performance models on FPGAs / Michael Pellauer [и др.] // Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays. — Monterey, California, USA: ACM, 2008. — С. 87—96. — (FPGA '08). — ISBN: 978-1-59593-934-0. — DOI: 10.1145/1344671.1344685. — URL: http://doi.acm.org/10.1145/1344671.1344685.
2. Asim: A Performance Model Framework / Joel Emer [и др.] // Computer. — 2002. — Т. 35. — 68–76. — ISSN: 0018-9162. — DOI: 10.1109/2.982918.
3. HAsim: FPGA-based high-detail multicore simulation using time-division multiplexing / Michael Pellauer [и др.] // High-Performance Computer Architecture, International Symposium on. — 2011. — 406–417. — DOI: 10.1109/HPCA.2011.5749747.
4. *Titov Alexander* Communication Between Modules Through Ports. — URL: http://code.google.com/p/mdsp/wiki/CommunicationBetweenModulesThroughPorts.
5. *Д.В. Сивухин* Общий курс физики в 5 т. 4. Оптика. — Физматлит, 2005.

8. Архитектурное состояние

> **bytesexual**, *adj.* [rare] Said of hardware, denotes willingness to compute or pass data in either big-endian or little-endian format (depending, presumably, on a mode bit somewhere).
>
> *The Jargon File (version 4.4.7)*[1]

Практически все устройства, составляющие современные вычислительные системы, являются синхронными электрическими цепями, изменяющими уровни сигналов на своих выходах каждый такт. Правила изменения этих сигналов зависят от значений на входных контактах, а также от некоторых значений внутри самого устройства — его внутреннего состояния. Хранение таких значений, как правило, реализовано аппаратно в форме различных регистров, банков памяти, а также линий прерываний. Состояние может быть сугубо внутренним для устройства, когда никакая другая реальная система не может его непосредственно считывать или модифицировать, или же быть представлено на архитектурном уровне. В симуляции мы обычно имеем доступ ко всему состоянию (однако степень детализации модели не всегда требует подробного соответствия внутренних частей).

8.1. О единицах данных, манипулируемых вычислительными системами

8.1.1. Байт

В большинстве вычислительных архитектур `байт` — это минимальный независимо адресуемый набор данных. В современных вычислительных системах байт считается равным восьми битам, однако в истории компьютеров известны решения с другим размером байта, например 6 бит для мейнфреймов IBM, 36 бит для ЭВМ PDP-10 и др. В

[1] `http://www.catb.org/jargon/`

компьютерных стандартах и официальных документах для однозначного обозначения 8-битной единицы информации используется термин «**октет**» (*лат.* octet).

8.1.2. Слово

Машинное слово (*англ.* machine word) — машинно-зависимая и платформозависимая величина, измеряемая в битах и равная разрядности регистров процессора и/или разрядности его шины данных. Оно определяет максимальную разрядность данных, которым данный процессор может оперировать за одну инструкцию; при необходимости обработки чисел шире, чем слово, требуются более одной инструкции, векторные команды или какие-либо другие приёмы. Из-за этого разрядность машинного слова в том числе определяет максимальный объём ОЗУ, напрямую доступный процессору. Как правило, данные, загружаемые процессором из/в оперативную память, должны иметь адреса, выровненные по ширине машинного слова.

Используются производные величины для обозначения относительного размера данных: половина слова (*англ.* halfword), двойное слово (*англ.* double word), четырёхкратное слово (*англ.* quad word) и т.д.

Для архитектур Intel существует отдельная традиция использования термина «word» для обозначения величин шириной в 16 бит на всех выпущенных за последние 40 лет процессорах, даже на современных 32-битных и 64-битных вариантах архитектуры IA-32. Это вызвано желанием выдержать одинаковую нотацию в документации ко всем ним.

8.2. Взаимодействие устройств

В вычислительных системах устройства должны иметь возможность запрашивать и изменять состояние других устройств. В зависимости от дизайна системы, ширины[1] передаваемых данных, необходимости адресации внутри элемента состояния, частоты обращения

[1]Количество бит в двоичном представлении числа.

к каналу данных и других факторов, можно выделить несколько классов доступов.

- Синхронное чтение регистра одного устройства другим или самим собой (рис. 8.1).
- Чтение/запись блока данных фиксированной ширины из памяти по известному адресу (рис. 8.2).
- Сигнализирование прерывания, сообщающее о внешних событиях, требующих внимания процессора (рис. 8.3).

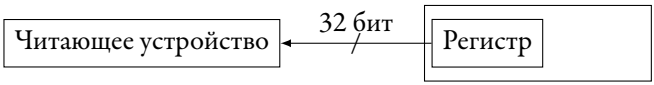


Рис. 8.1. Чтение регистра фиксированной ширины может быть произведено связанным с ним устройством на каждом такте работы

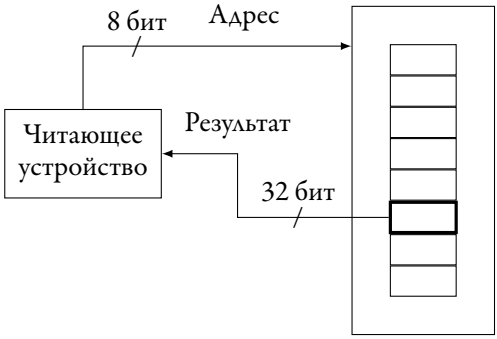


Рис. 8.2. Пример чтения из блока памяти. Для указания адреса блока используется шина адреса. Чтение результата, как правило, происходит по другой группе проводов, составляющих шину данных. Не показаны, но часто присутствуют дополнительные линии шины адреса для передачи типа операции (чтение/запись), а также линия шины результата, используемая для индикации его готовности

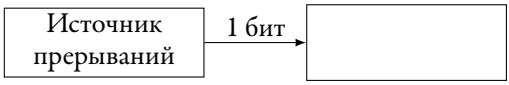


Рис. 8.3. Линия для сигнализации ситуации прерывания состоит из одного провода, уровень или фронт сигнала на котором обозначает событие, требующее внимания ЦПУ

8.3. Банки регистров и блоки памяти

Универсальная абстракция для отдельного регистра при функциональной симуляции — переменная достаточной ширины, чтобы уместить все биты искомого устройства[1].

```
uint32_t register_a;
```

Однако, как правило, необходимо иметь и симулировать множество (*банк*) регистров. Естественным является представление их как массива переменных.

```
uint32_t register_bank[bank_size];
```

Положение каждого отдельного регистра в таком случае определяется его индексом. Однако зачастую архитектура устройства подразумевает наличие регистров различной ширины и иногда доступ к отдельным их частям (например, к младшей половине) как к независимым устройствам. Наилучшим представлением банка регистров является массив байт. Адрес отдельного элемента при этом определяется смещением первого байта внутри массива, а его ширина задаётся явно в самих операциях чтения и записи.

```
uint8_t reg_space[bank_width]; // определение банка регистров
uint32_t rd_a = *(uint32_t*)(reg_space + offset_a); //
    чтение регистра
*(uint32_t*)(reg_space + offset_a); // запись регистра
```

[1]Используемые целые типы фиксированной ширины `uint16_t`, `uint32_t`, `uint64_t` и т.п. определены в стандартном заголовочном файле библиотеки libc `<stdint.h>`. Обычные целые типы `char`, `short`, `int`, `long` и их беззнаковые варианты не рекомендуется использовать, так как их ширина не задана стандартом языка Си.

Принцип моделирования блоков памяти не отличается от рассмотренной выше схемы банка регистров; однако ячейки памяти не имеют ассоциированных с ними имён и характеризуются только смещением относительно начала банка; ширина читаемых из памяти данных тоже может варьироваться и определяется архитектурными особенностями моделируемой системы.

8.3.1. Endianness

Дополнительным фактором, который всегда необходимо учитывать при проектировании и написании моделей, является, возможно, различный порядок адресации байт внутри слов, двойных слов и прочих многобайтных чисел (*англ.* Endianness), используемых системами хозяина и гостя.

- Big endian[1] — соглашение, по которому байты машинного слова хранятся в памяти в том же порядке, в котором они присутствуют в позиционной записи, начиная с младшего разряда. Например, 0x11223344 будет сохранено как последовательность 0x11, 0x22, 0x33, 0x44. Для получения младшего байта впоследствии нам необходимо знать, что записанное число имело ширину в 32 бита и что он, таким образом, расположен по смещению 3.
- Little endian[2] — младшие байты расположены по младшим же адресам. Число 0x11223344 в памяти будет храниться как последовательность 0x44, 0x33, 0x22, 0x11. Младший байт при этом всегда будет расположен по смещению 0 независимо от ширины слова.

Существование двух несовместимых соглашений на хранение данных в памяти осложняет написание переносимого кода и требует особого внимания программиста и использования средств статического анализа кода для проверки его корректности для обоих соглашений. Чтобы окончательно запутать читателя, отметим ещё два обстоятельства, связанных с порядком байт.

[1]Также называемый сетевым порядком байт (*англ.* network byte order) из-за использования в стеке TCP/IP.

[2]Соглашение, используемое в том числе в архитектуре Intel IA-32.

– Некоторые архитектуры, например PDP-11, имеют т.н. middle-endian форматы для чисел, превышающих размер архитектурного слова; при этом слова данных имеют порядок, отличный от порядка байт в слове.
– В архитектуре IA-64 endianness может быть изменена при инициализации системы, а для ARM [5] существует инструкция, изменяющая порядок интерпретации байт динамически во время работы приложения.

Endianness приходится учитывать при взаимодействии систем, хранящих данные с несовпадающими соглашениями на порядок байт. В нашем случае это хозяйская и гостевая системы — если порядок байт в словах различен, то приведённые в примере выше операции приведения типов будут некорректно загружать значения симулируемых регистров. Приходится аккуратно отслеживать и конвертировать многобайтные значения при операции с ними. Лучше всего использовать для этого функции, определённые стандартом POSIX: `htonl()`, `htons()`, `ntohl()`, `ntohs()` или аналогичные им.

– `uint32_t htonl(uint32_t hostlong)` — конвертирует 32-битную беззнаковую величину из локального порядка байтов в сетевой;
– `uint16_t htons(uint16_t hostshort)` — конвертирует 16-битную беззнаковую величину из локального порядка байтов в сетевой;
– `uint32_t ntohl(uint32_t netlong)` — конвертирует 32-битную беззнаковую величину из сетевого порядка байтов в локальный;
– `uint16_t ntohs(uint16_t netshort)` — конвертирует 16-битную беззнаковую величину из сетевого порядка байтов в локальный.

Разработчики и пользователи любых протоколов, задействованных в гетерогенных системах (PCI, USB, LAN), вынуждены учитывать все изложенные выше обстоятельства при разработке ПО.

8.4. Побочные эффекты

С помощью показанной выше абстракции можно моделировать произвольные запросы к устройствам. Кроме непосредственного хранения значений в регистрах, запись или чтение из них может вызывать эффекты, непосредственно не связанные с сохраняемым значением, например: дисковая операция, зажигание точек на дисплейном устройстве, отсылка пакетов в сетевой карте и т.п. Моделирование такого регистра усложняется и отличается от простого обращения с ячейкой памяти:

```
read_val = f_read(addr);
f_write(addr, value);
```

Если возможен невыровненный доступ к регистру, то необходимо учитывать и смещение данных относительно границ слов.

Самые простые побочные эффекты

1. Устройство только для чтения. Попытка записи в него вызывает архитектурное исключение.
2. Регистр, игнорирующий запись. Все записи в него не имеют эффекта, и при чтении возвращается заранее определённое значение.

8.5. Пространства памяти

Набор инструкций большинства микропроцессоров подразумевает доступ к достаточно ограниченному набору непрерывных изолированных друг от друга пространств адресов (чаще всего это пары инструкций `LOAD/STORE` и `IN/OUT` для обращения к двум независимым областям адресации). При этом устройств и их адресуемых регистров в ЭВМ обычно значительно больше. Потому банки регистров отдельных устройств сопоставляются с регионами адресов пространств памяти(рис. 8.4). Так ОС и программы получают возможность обращаться к ним как к обычной памяти с помощью инструкций; однако

вместо чтения/записи ячейки ОЗУ будут происходить побочные эффекты, связанные с обработкой выбранным устройством запроса.

Исторически в большинстве архитектур для доступа к устройствам предназначалось исключительно пространство портов ввода-вывода (*англ.* I/O space), адресуемое с помощью `IN/OUT` и непересекающееся с обычной памятью [3]. Однако такой подход является неоправданным с точки зрения скорости работы и гибкости конфигурирования при увеличении скорости устройств и масштаба вычислительных комплексов. В современных системах устройства могут быть отображены на любое из доступных пространств адресов.

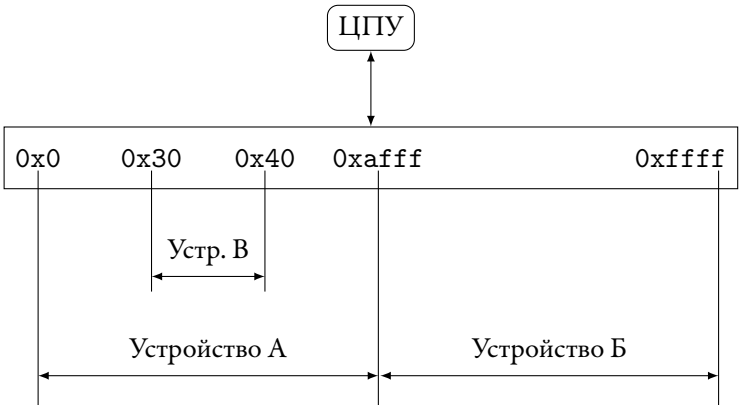


Рис. 8.4. Отображение регистров устройств на пространство памяти. Диапазоны адресов памяти, принадлежащие различным устройствам, в общем случае могут перекрываться (устройства А и В в данном примере), при этом необходимо определять приоритет их расположения

В операционной системе Windows в стандартном диспетчере устройств можно наблюдать, какие диапазоны адресов физической памяти выделены для различных периферийных устройств, если выбрать представление «устройства по подключению». На рис. 8.5 приведена часть вывода для 64-битного варианта этой операционной системы.

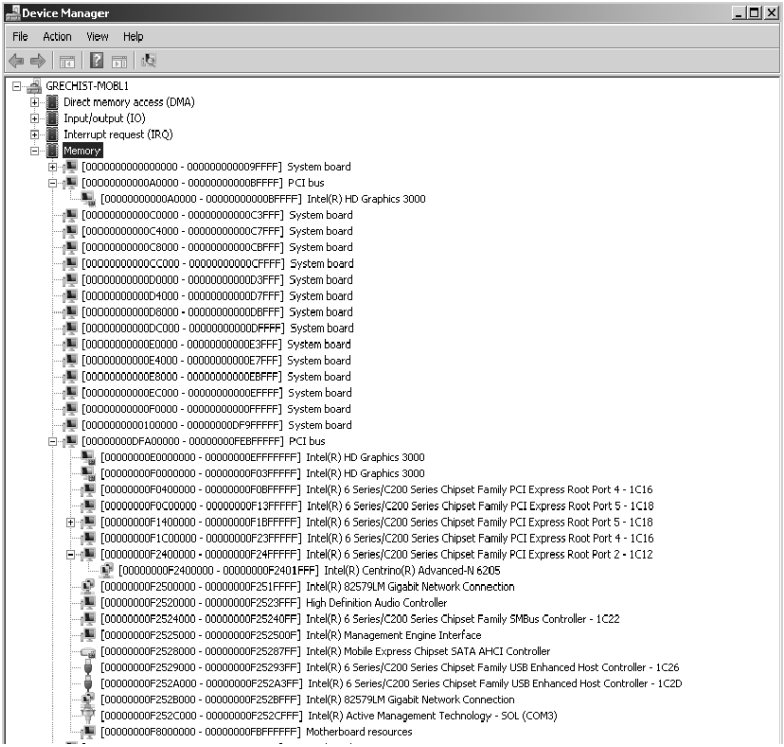


Рис. 8.5. Выделение диапазонов памяти под нужды периферийных устройств в Windows, наблюдаемое через диспетчер устройств

8.5.1. Карты пространств памяти (memory space mappings)

В реальных сценариях работы функционирование пространств памяти может усложняться следующими обстоятельствами.

- Состояние устройств в пространствах памяти может меняться со временем согласно внутренним принципам процесса загрузки системы; количество их регистров также подвержены изменению, их адреса в памяти — тоже.
- Несколько устройств могут быть расположены в пересекающихся диапазонах адресов, требуя динамического разрешения конфликтов.

– Одно и то же устройство может одновременно быть расположено по нескольким диапазонам адресов и при этом обеспечивать в них различные побочные эффекты.

Для учёта всех этих особенностей в симуляции используется такая структура, как карта памяти. Для любого доступа по адресу в памяти алгоритм определения устройства, отвечающего за его обработку, следующий [4] (рис. 8.6).

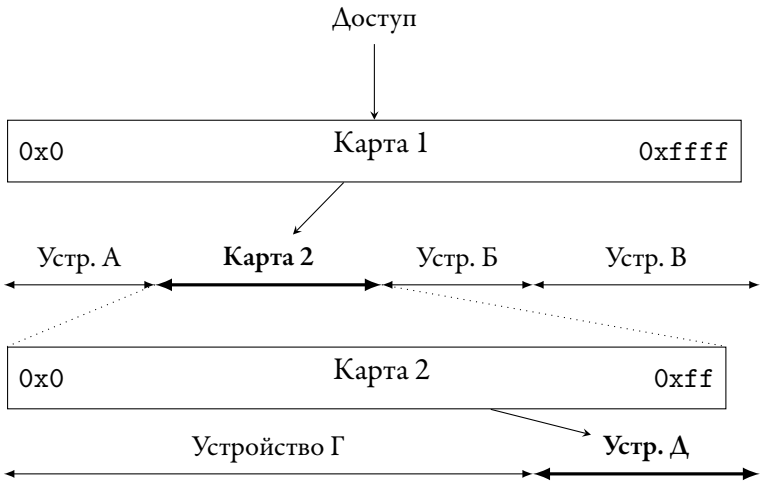


Рис. 8.6. Алгоритм поиска устройства в иерархии карт памяти. Перед передачей транзакции доступа конечному устройству её параметры, например адрес, могут быть изменены

1. Для доступа определяется его тип (чтение, запись, предвыборка, чтение инструкции и т.п.), адрес начала, длина доступа.
2. В карте памяти по адресу начала определяется нижележащее устройство, которое, в свою очередь, может быть или конечным устройством, или ещё одной картой памяти.
3. Если ни одно устройство не имеет записи в карте, используется устройство по умолчанию (*англ.* default target), если оно определено для текущей карты.

4. При необходимости параметры доступа модифицируются так, чтобы соответствовать устройству, например адрес смещается на фиксированную величину.
5. Если полученное устройство — карта памяти, то поиск продолжается в ней, иначе доступ в память передаётся найденному устройству на обработку.
6. Ситуация, когда для данного адреса не существует трансляции, является ошибкой в конфигурации модели. Также, скорее всего (но необязательно), по конвенциям работы аппаратуры и исполняющейся программы некорректным будет являться многобайтный доступ, части которого попадают в различные устройства.

Для разрешения неоднозначности при ситуации, когда одно и то же устройство размещено в нескольких картах, в доступе в память из карты добавляется дополнительное число — номер функции. При обработке транзакции устройство может проверить данный номер и скорректировать своё исполнение. Ещё одна задача, которую выполняют с помощью карт памяти — динамическое изменение порядка байт (endianness) доступа, если конечное устройство это требует.

8.6. Линии прерываний

Прерывания являются ещё одним методом коммуникации периферийных устройств, чаще всего с ЦПУ. Однако, в отличие от рассмотренных ранее способов, оно характеризуется следующими отличиями.

– Инициатором взаимодействия является периферийное устройство. Процессор в зависимости от ряда условий может отложить обработку полученных прерываний до момента, когда это станет возможным или алгоритмически корректным, или же немедленно перейти в процедуру обработки.
– Прерывание несёт один бит информации, означающий, что на каком-то из внешних устройств имеется событие, требующее

внимания. На каждую линию может быть подключено несколько устройств; в таком случае получатель сигнала прерывания не способен сразу различить, которое из них было инициатором — требуется проинспектировать все из них.
- Коммуникации однонаправленны в отличие от, например, чтения из регистра, при котором как читающее, так и пишушее устройство имеет возможность получить некоторую информацию о взаимодействии.

Моделирование работы линии прерываний может быть выполнено следующим способом: в модели архитектурного состояния устройства-получателя заводится флаг `bool interrupt_raised`, обозначающий, что произошло прерывание. Процессор после каждой исполненной инструкции проверяет состояние этого флага и в случае, когда он выставлен, меняет своё состояние соответствующим образом, при этом сбрасывая флаг в начальное положение. Недостатки данного подхода: 1) обнаружение события прерывания сдвигается на границу между исполнением инструкций, *после* текущей, что не всегда корректно — некоторые прерывания могут отменять текущую инструкцию, не допуская завершения её симуляции; 2) частый опрос флага снижает скорость работы модели. Альтернативный подход: использовать обратный вызов (*англ.* callback) — указатель на функцию (метод), непосредственно оперирующий состянием процессора. Инициатор прерывания имеет возможность вызвать указанную функцию для сигнализации прерывания, и затем уже логика работы самого процессора должна определять, в какой момент оно будет им обработано.

8.7. Оптимизации при моделировании

Как было описано выше, достаточно просто обеспечить моделирование состояния с помощью переменных и массивов переменных, хранящихся в памяти. Такое решение универсально и используется во всех случаях. Тем не менее для ускорения работы моделей иногда применимы описанные далее оптимизации.

8.7.1. Прямое использование состояния хозяина

В ряде ситуаций хозяйское оборудование предоставляет аппаратные ресурсы, в том числе и регистры, которые можно переиспользовать для нужд модели, одновременно ускоряя её. Состояние гостевой системы (частично) хранится не в медленной оперативной памяти, а в более быстрых регистрах.

Такой режим работы модели приобретает признаки прямого исполнения. Цикл симуляции при этом имеет следующие стадии.

- Гостевое состояние регистров/памяти загружается на соответствующие регистры хозяина.
- Симуляция проводится в течение некоторого времени.
- При восстановлении контекста хозяина гостевое состояние сохраняется во внешнюю память.

Как и во всех случаях использования прямого исполнения, выигрыш от него нивелируется необходимостью переключения контекста хозяина и гостя, и поэтому перед его реализацией следует оценить, будет ли польза от достаточно кропотливой работы по включению подобной функциональности в модель. Кроме того, применимость метода сильно зависит от степени соответствия архитектур хозяина и гостя.

8.7.2. Кэширование доступов к картам памяти

Как было показано в п. 8.5.1, обращение по адресу к физической памяти подразумевает инспектирование одной или более карт для определения адресата. Эта процедура может быть длительной и замедлять моделирование исполняющих устройств. Возможность оптимизации здесь проистекает из наблюдения, что подавляющая часть архитектурных запросов в память оканчиваются в ОЗУ-подобных устройствах, не предоставляющих побочных эффектов. Поэтому сам факт обращения к карте памяти можно отложить или скомбинировать с последующими запросами с помощью кэширования на стороне запрашивающего исполняющего устройства, что позволит ему проводить агрессивные оптимизации своей работы.

Подобная техника должна аккуратно отслеживать изменения в используемых картах памяти. Так, при смене типа устройства, обслуживающего некоторый диапазон пространства адресов, все агенты, имевшие возможность кэшировать его, должны быть уведомлены об этом для обеспечения корректной работы модели.

8.7.3. Ленивое вычисление флагов

Один из типов регистров, встречающихся в современных процессорах — это регистр флагов, каждый бит которого означает, что результат предыдущей операции (как правило, арифметической или логической) имеет определённые свойства. Так, флаг ZF (*англ.* zero flag) выставляется в значение 1, если результат предыдущей команды равен нулю, OV (*англ.* overflow flag) — если он не может быть сохранён в ограниченном числе двоичных разрядов, отведённых для него, и т.д [1]. Корректное моделирование очень большого числа инструкций подразумевает вычисление и сохранение отдельных битов регистра флагов. Даже для простой операции сложения, результат которой получается с помощью единственной машинной инструкции хозяина, приходится включать длинный блок в десятки инструкций для проверки свойств данного результата.

Общим наблюдением, ведущим к идее следующей оптимизации, является тот факт, что результат, хранящийся в регистре флагов, используется далеко не каждой последующей инструкцией, и потому нет нужды обновлять его так часто. Достаточно производить такую работу только когда это действительно будет необходимо для корректной обработки операции, зависящей от регистра флагов [2], или при инспектировании состояния пользователем.

Замечание. *В программировании подход (стратегия) вычислений, при котором анализ и исполнение выражения происходит лишь при необходимости использовать его результат, называется* ***ленивым*** *(англ. lazy), в отличие от* ***энергичного*** *(англ. eager) подхода, при котором выражение вычисляется сразу после доступности значений всех его входных слагаемых.*

8.8. Точки сохранения

Возможность сохранения состояния программы в файл с возможностью последующего восстановления и возобновлением работы из него является необходимым для широкого класса приложений: параллельные программы, отказоустойчивые системы, энергосберегающие программы. Для симуляторов наличие такой функциональности означает экономию времени при изучении поведения гостевых систем. Рассмотрим, что должно входить в такую точку сохранения (*англ.* checkpoint или savepoint).

- Архитектурное состояние всех моделируемых устройств. Если использовались некоторые из описанных выше техник увеличения скорости симуляции, например, размещение состояния на хозяйских регистрах, то необходимо предварительно привести модель в «стабильный» режим, когда все подлежащие сохранению регистры имеют гарантированное расположение в памяти.
- Если часть элементов архитектурного состояния функционально связана с другими элементами, то возможно исключить их из содержимого точки сохранения и пересоздавать их при восстановлении. Это поможет избежать возможности рассинхронизации таких элементов. Состояние программы, не определяющее архитектурное состояние моделей, как правило, не стоит сохранять — это сэкономит время на отладку. К примеру, не стоит дорожить кэшем двоичной трансляции.
- Информация о соединении моделей отдельных устройств друг с другом. Для различения объектов необходимо использовать некоторые идентификаторы, которые можно будет использовать в качестве ссылок на них в других устройствах при хранении. Такие идентификаторы должны «переживать» уничтожение самих объектов с их последующим пересозданием. Поэтому, например, машинные адреса не подходят, так как при перезапуске они каждый раз будут различаться.

8.8.1. Переносимость точек сохранения

Если симулятор планируется использовать на различных хозяйских системах, то необходимо предусмотреть сценарий сохранения гостевого состояния на одной, а восстановление — на другой, и что edianness, разрядность и форматы хранения данных при этом могут различаться. Например, нельзя хранить указатели, так как на другой системе они будут бесполезны.

8.8.2. Обращённое во времени исполнение

Периодическое сохранение состояния симулируемой системы в сочетании с детерминизмом симулятора позволяет реализовать такой невозможный в реальном мире сценарий исполнения, как обращённое во времени исполнение (*англ.* reverse execution), см. рис. 8.7.

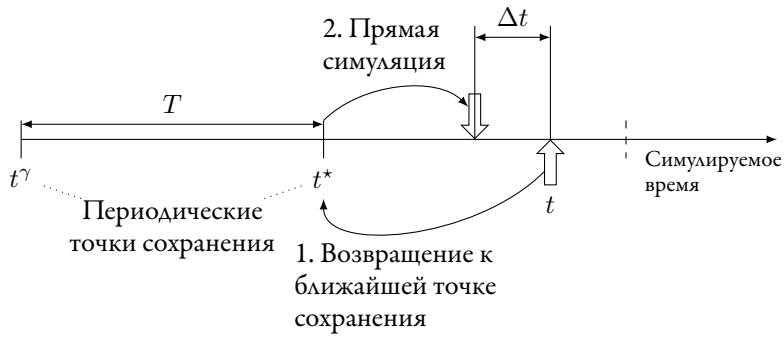


Рис. 8.7. Обращённое во времени исполнение

При периоде снятия точек сохранений T, текущем времени симуляции t и необходимости откатиться на Δt секунд выполняются следующие шаги.

1. Восстановление к точке $t^\star$, ближайшей слева к моменту $t - \Delta t$:
$$t^\star = \lfloor (t - \Delta t)/T \rfloor \cdot T.$$
2. Прямая симуляция до необходимой точки в течение t_{direct}:
$$t_{direct} = t - \Delta t - t^\star.$$

Поскольку любая симуляция из фиксированного состояния всегда приводит к одинаковому финальному состоянию, то указанная последовательность действий эквивалентна «скачку» из момента t в момент $t - \Delta t$.

Постепенно увеличивая значения Δt, можно создать видимость того, что состояние модели эволюционирует в обратном направлении в симулируемом времени.

Скорость обращённой симуляции будет обратно линейно зависеть от значения T — чем чаще делаются точки сохранения, тем в среднем короче длина необходимой прямой симуляции. Однако объём памяти, требуемый для хранения всех точек сохранения, также растёт с уменьшением T.

8.8.3. Миграция

Процесс возобновления симуляции гостя на хозяйской системе, отличной от той, на которой была создана точка сохранения, получил название «миграция». Возможность миграции важна для распределённых систем виртуализации, призванных обеспечить устойчивость исполнения и минимальный простой виртуальных машин при угрозах отказа аппаратуры физических компьютеров. Если некоторая хозяйская система сигнализирует о сбое из-за отказа жёстких дисков, памяти, питания и т.п., то виртуальные машины, исполнявшиеся до этого на ней, перезапускаются из точек сохранения, регулярно создаваемых и сохраняемых на внешнем хранилище, на оставшихся исправных хозяевах. Наблюдаемое клиентами время простоя сервиса, обеспечиваемого мигрированными виртуальными машинами, минимально.

8.8.4. Формат файлов точек сохранения

Не существуют общепринятого соглашения о формате файлов, используемых для сохранения состояния гостевых систем, так как они очень сильно зависят от структуры моделей и особенностей симулятора. Сформулируем некоторые общие принципы их дизайна.

– Файлы могут быть как двоичные, так и текстовые. Хотя первый формат может обеспечить в среднем меньший размер файлов, содержимое его будет нечитаемым для человека, что усложняет отладку симулятора, разработку инструментов для обработки точек. С другой стороны, некоторые данные изначально неудобны для анализа человеком, например, содержимое памяти, дисков и других массивов данных, тогда как представление их в текстовом виде сильно раздувает их объём. Такие данные следует хранить в двоичном виде, а оставшиеся текстовые данные могут быть сжаты архиватором.
– Следует предусмотреть возможность использования общепринятых стандартов хранения данных при проектировании нового формата точек сохранения, т.е. не следует «изобретать велосипед». Например, для представления иерархических данных можно использовать XML (*англ.* extended markup languange) или JSON. Для хранения однородных массивов данных, используемых для представления состояния жёстких дисков, оперативной памяти и т.п. сущностей существуют форматы QCOW2 или VMDK. Такой шаг повысит удобство конвертации в/из сторонних смуляторов.
– Необходимо явно прописывать endianness сохраняемых данных для выбранного формата. Это поможет в дальнейшем при адаптации симулятора для новых гостевых и хозяйских систем.

8.8.5. Инкрементальные точки сохранения

В ситуациях, когда несколько точек сохранения создаются последовательно с некоторым интервалом для одной симуляции, первая из них должна полностью описывать состяние системы, а последующие точки могут опускать часть данных, не изменившихся с предыдущего раза, и содержать только «дельту», а также ссылку на предыдущую точку. Это позволяет экономить хозяйское дисковое пространство. Особенно сильный выигрыш происходит при хранении образов дисков и памяти.

8.9. Вопросы к главе 8

Вариант 1

1. Какой байт будет расположен первым в памяти на Little Endian системе при записи числа `0xaabbccdd` в память?
2. Выберите правильное отношение для фразы «машинное слово длиной w байт выровнено в памяти по адресу $addr$»:
   a) $addr \neq 0 \pmod{w}$ (адрес не делится нацело на длину слова),
   b) $w = 2^k$ и $addr = 2^m$, $k, m \in \mathbb{N}$ (адрес и длина являются степенями двойки),
   c) $w = 2^k$ и $addr = 2^m$, $k \leq m$,$k, m \in \mathbb{N}$ (адрес и длина являются степенями двойки, степень длины меньше степени адреса),
   d) $addr = 0 \pmod{w}$ (адрес делится нацело на длину слова).
3. Почему симулятор не имеет права кэшировать регионы гостевой памяти, помеченные как отображённые на устройства?
4. Определение понятия «машинное слово».
5. Какой интегральный тип языка Си наиболее удачно использовать для хранения состояния моделируемого регистра шириной 32 бита?
   a) `int`,
   b) `unsigned int`,
   c) `uint32_t`,
   d) зависит от хозяйской системы.

Вариант 2

1. Какой байт будет расположен первым в памяти на Big Endian системе при записи числа `0xbaadc0de` в память?
2. Какую стратегию подразумевает концепция ленивого вычисления?
   a) Замена точного значения выражения приближённым, но получаемым за меньшее время.

b) Запуск вычисления выражения происходит лишь при необходимости использовать его результат.
c) Выражение вычисляется сразу после доступности значений всех его входных слагаемых.
d) Значение подвыражения, используемого в нескольких других выражениях, сохраняется при первом вычислении и затем переиспользуется.

3. Определение понятия «байт».
4. Сколько бит информации получает процессор при первоначальном возникновении сигнала на линии прерывания?

a) 1 бит.
b) 8 бит.
c) Зависит от архитектуры.

5. Выберите правильные окончания фразы: карта памяти

a) использует цель по умолчанию, если обрабатываемый запрос не попадает ни в одно из устройств,
b) может указывать на устройство не более одного раза,
c) должна указывать на все присутствующие в гостевой системе устройства,
d) может указывать не только на устройства, но и на другие карты памяти.

Литература

1. Intel® 64 and IA-32 Architectures Software Developer's Manual. Volume 1. — Intel Corporation.
2. *Mihoka Darek*, *Shwartsman Stanislav* Virtualization Without Direct Execution or Jitting: Designing a Portable Virtual Machine Infrastructure // ISCA-35 Proceedings of the 1st Workshop on Architectural and Microarchitectural Support for Binary Translation. — URL: http://bochs.sourceforge.net/Virtualization_Without_Hardware_Final.pdf (дата обр. 05.05.2012).
3. *Pfeiffer J.* CS 473. Input and output lecture notes. — New Mexico State University, 2006. — URL: http://www.cs.nmsu.edu/~pfeiffer/classes/473/notes/io.html.
4. Simics Model Builder Guide 4.6. — Wind River, 2011.
5. *Sloss Andrew N.*, *Symes Dominic*, *Wright Chris* ARM System Developer's Guide. Designing and Optimizing System Software. — Morgan Kaufmann, 2004. — ISBN: 1-55860-874-5.

9. Сверхоперативная память — кэши

Эта глава могла бы носить более претенциозное название — «Хранение и получение информации»; с другой стороны, её можно было бы назвать кратко и просто — «Просмотр таблиц».

Дональд Кнут. Искусство программирования. Сортировка и поиск

9.1. Стена памяти

С развитием технологии микропроцессоры становились всё быстрее за счёт повышения тактовой частоты, увеличения ширины машинного слова и других факторов. В определённый момент было замечено, что все эти улучшения не дают существенного прироста скорости исполнения программ по той причине, что рост скорости оперативной памяти, используемой в ЭВМ, не столь стремителен. В результате процессор вынужден простаивать, ожидая, пока данные из памяти будут доставлены. Данная проблема получила название *стены памяти* (*англ.* memory wall, рис. 9.1) и означает, что в случае неизменности основных принципов организации вычислений рост производительности систем будущего ограничен именно скоростью оперативной памяти.

9.2. Назначение, принцип работы

Существует как минимум две области применения сверхоперативной памяти. Первая из них связана с описанной выше проблемой медленности доступа к ОЗУ. Вторая — с обеспечением корректной и быстрой синхронизации работы многопроцессорных систем.

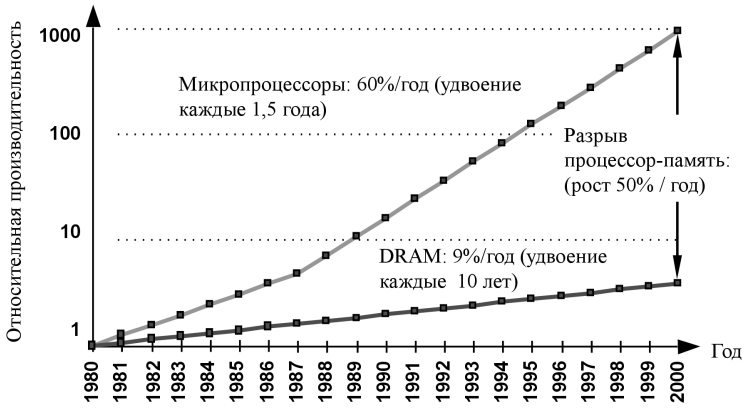


Рис. 9.1. Сравнительный рост скорости оперативной памяти и микропроцессоров

9.2.1. Ускорение обращений в память

Частично компенсировать данную проблему призваны устройства сверхоперативной памяти, в настоящее время чаще всего именуемые *кэшами* (*англ.* cache). Для большинства алгоритмов наблюдается выполнение принципов как временной, так и пространственной локальности: данные, к которым обращались недавно, скорее всего, будут запрошены снова в ближайшем будущем; кроме того, вероятно, что соседние с ними данные тоже будут запрошены (рис. 9.2.). Представляется разумным иметь небольшое по сравнению с основной оперативной памятью хранилище, расположенное ближе к процессору и потому работающее быстрее, чтобы хранить в нём наиболее часто запрашиваемые данные. В кэше хранятся копии блоков информации, соответствующих подмножеству адресов оперативной памяти [1, 2].

Данные в кэш попадают двумя способами. Во-первых, новый регион памяти может быть запрошен впервые за время работы, причем его придётся извлекать из основной памяти с задержкой, однако копия попадёт в кэш, и последующие обращения будут обработаны быстрее. Во-вторых, программист может предусмотреть, что некоторый диапазон памяти вскоре будет использован, и использовать явную *предва-*

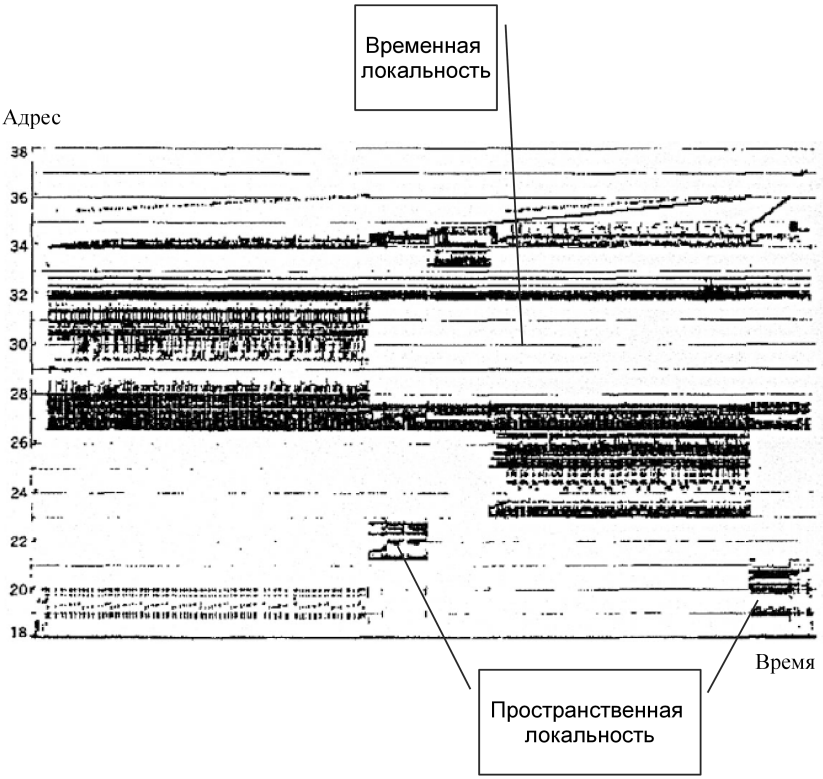


Рис. 9.2. Пример явлений временной и пространственной локальности доступов памяти при работе приложения. Иллюстрация взята из [3]

рительную загрузку (*англ.* prefetching) ещё до первого к нему обращения. Многие архитектуры имеют для этого специальные инструкции.

Поскольку ёмкость кэша невелика, а рабочее множество программы со временем меняется, неизбежно возникает ситуация, когда часть кэша придётся использовать для новых данных, а хранившийся там блок или отбросить, или, если он был изменён, записать обратно в память. Такой процесс называется *вытеснение* (*англ.* eviction).

Случаются ситуации, когда содержимое всего кэша перестаёт быть релевантным исполняемому коду. В таком случае происходит *сброс* (*англ.* flush) кэша, когда из него принудительно вытесняются все дан-

ные.

9.2.2. Поддержка транзакций

Использование дополнительного буфера в виде кэша позволяет откладывать момент записи в память, накопляя в нём совершённые последовательные изменения, применённые к различным адресам в памяти. Затем содержимое может быть записано за один раз, создавая видимость неделимости (атомарности) акта модификации — для внешнего наблюдателя памяти это будет выглядеть так, как будто сразу, без промежуточного состояния, изменилось большое число байт. Мы получаем транзакционную семантику записей в память. С другой стороны, если по каким-то причинам было решено, что все доступы в память, содержащиеся в кэше некоторого процессора, уже неактуальны, можно просто очистить его — откатить транзакцию, и для внешнего наблюдателя это будет выглядеть так, как будто никаких операций над памятью не было выполнено.

Описанный механизм является основным для реализации т.н. систем с аппаратной **транзакционной памятью** [8].

Замечание. *Акцент данной главы смещён от собственно сценариев/алгоритмов эмуляции к описанию архитектурных принципов работы самой сверхоперативной памяти.*

Первый сценарий применения систем кэшей подразумевает улучшение временных характеристик основного оперативного запоминающего устройства прозрачно для исполняющихся программ. Во многих функциональных моделях, пренебрегающих задержками, модели кэша может и не быть вовсе, несмотря на то, что в реальной аппаратуре он присутствует. Необходимость в его аккуратном моделировании возникает при исследовании производительности подсистемы памяти.

Во втором случае влияние кэшей видимо на функциональном уровне и уже нельзя пренебрегать их функциональным моделированием. В наборе инструкций могут присутствовать команды управления транзакциями. Примером является расширение Intel®TSX [5, глава 8], ожидаемое в процессорах микроархитектуры Haswell.

9.3. Устройство кэша — линии, тэги, ассоциативность

Рассмотрим общий принцип организации кэша[1]. См. рис. 9.3.

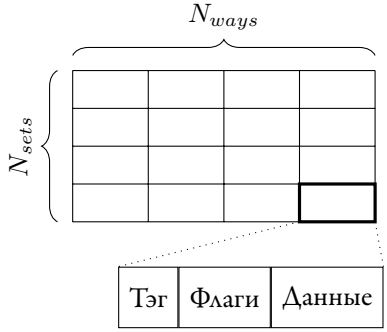


Рис. 9.3. Организация кэша в линии, сеты N_{sets}, пути N_{ways}. Ячейка содержит информацию: тэг, флаги и собственно данные линии

Единица хранимых данных именуется *линией кэша*. Как правило, она имеет ёмкость в несколько машинных слов и хранит копию последовательной области памяти, начиная с адреса, выровненного по размеру линии. В современных процессорах размер линии может быть 32 или 64 байта.

Смысл кэша состоит в возможности быстрого нахождения соответствия «адрес–данные» (ситуация *попадания в кэш*, *англ.* cache hit) или констатации отсутствия такого соответствия (ситуация *промаха*, *англ.* cache miss). Алгоритм такого поиска состоит из следующих шагов.

1. Каждая ячейка кэша кроме собственно копии данных содержит вспомогательную информацию, состоящую из тэга и группы флагов.
2. Из адреса в памяти выделяются смещение байта внутри линии *offset*, номер множества (сета) n_{set} и тэг tag. Пример такого раз-

[1]Однако отметим, что описанная далее схема не является единственно возможной. Интересующийся читатель найдёт более полную картину в [4].

биения приведён на рис. 9.4. Алгоритм выделения n_{set} и tag может быть произвольным, однако он должен позволять однозначно идентифицировать адрес линии.

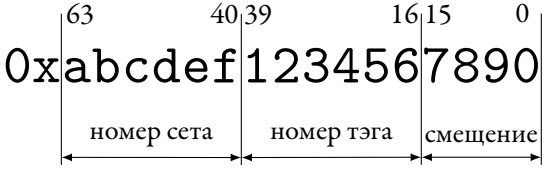


Рис. 9.4. Пример выделения значений номера сета, тэга и смещения по диапазонам бит из адреса в шестнадцатеричном представлении

3. Выбирается сет с указанным n_{set}. Искомая линия может находиться в любом из N_{ways} позиций в нём; ищется ячейка, содержащая тэг, равный tag. Аппаратная реализация большинства кэшей подразумевает, что поиск этот происходит параллельно во всех линиях сета и потому занимает фиксированное время.
4. Если удовлетворяющая условию ячейка найдена, проверяется, активна ли она или является «мусором», оставшимся от предыдущей работы.
5. В случае нахождения тэга и валидности мы имеем попадание в кэш, иначе же — промах.

Таким образом, следующие параметры определяют описанный выше простой кэш.

– Ёмкость линии данных S_{line}, измеряемая в байтах.
– Ассоциативность N_{ways} определяет, сколькими способами одна и та же линия может быть размещена в ячейках кэша, а также адреса линий, конкурирующих с ней за место (очевидно, что линии с разным значением n_{set} всегда будут в разных сетах и не могут вытеснять друг друга). Если $N_{ways} = 1$, то мы имеем кэш *прямого отображения* (*англ.* directly mapped cache), в котором линия с определённым адресом, если она присутствует, всегда занимает одну и ту же ячейку. Соответственно, если $N_{sets} = 1$,

мы имеем полностью *ассоциативный кэш* (*англ.* fully associative cache), в котором линия может занимать любую ячейку.
- Количество сетов N_{sets} должно быть достаточно большим, чтобы вместе с ёмкостью тэга быть способным полностью адресовать все возможные адреса линий в памяти системы. В противном случае некоторые диапазоны памяти просто не смогут быть отображены на ячейки.
- Ёмкость кэша — C. Очевидно, что она равна произведению рассмотренных ранее величин:

$$C = S_{line} N_{ways} N_{sets}.$$

9.4. Промахи. Алгоритмы вытеснения линий

Что должно происходить, если был детектирован промах кэша? Отсутствующие данные запрашиваются из памяти и, как правило, после получения помещаются в кэш. Для размещения новой линии может быть использована любая пустая, т.е. не содержащая актуальных данных, ячейка.

Как поступать, если все ячейки сета содержат актуальные данные? В таком случае выбирается одна из них, и её содержимое вытесняется и заменяется на новое. Перед этим проверяется ещё один флаг — *модификации* (*англ.* dirty), поднимаемый при первой записи в ячейку. Если хранящиеся в линии данные отличаются от версии, содержащейся в оперативной памяти, то необходимо передать их из кэша обратно в ОЗУ. Для ячеек, использовавшихся только для чтения, в этом нет необходимости, и они могут быть сразу перезаписаны.

Как выбрать, какую ячейку текущего сета следует вытеснить? «Идеальный» алгоритм — выбирать линию с адресом, к которому впоследствии программа не будет обращаться дольше всего. Однако его реализация подразумевает знание поведения алгоритма в будущем, а такой анализ в общем случае как минимум затруднителен, а с практической стороны и вовсе невозможен. Мы можем отталкиваться от истории предыдущих доступов к линиям при формулировке *политики вытеснения* (*англ.* replacement policy) линий.

Перечислим лишь некоторые из существующих алгоритмов [6].

- Вытеснять всегда первую ячейку. Очевидно, что это — единственная доступная схема работы для кэша с прямым отображением. Для ассоциативных кэшей она нецелесообразна.
- Вытеснять случайную ячейку. Политика довольно проста в реализации, не требует хранения дополнительных данных для каждой ячейки, но может оказаться субоптимальной на практике.
- FIFO (*англ.* first in first out) — *классическая очередь*. Для каждого сэта хранится порядок, в котором занимались его ячейки. Для вытеснения выбирается самая «старая» ячейка. Достоинство данной политики — в простоте и небольших накладных расходах «в железе». Однако для наибольшей эффективности алгоритма необходимо, чтобы приложение, использующее память, имело потоковый характер чтения памяти, что далеко не всегда наблюдается на практике — к некоторым ранее затребованным адресам оно может обращаться снова и снова с нерегулярными интервалами.
- LRU (*англ.* least recently used). Для каждой ячейки хранится её «возраст» — величина, пропорциональная времени, прошедшему с момента последнего к ней обращения. При вытеснении выбирается самая «старая» ячейка, т.к. к ней не обращались дольше всего и потому, возможно, не обратятся в ближайшем будущем. Данная политика (и её различные оптимизации) является самой популярной из-за наилучшего сочетания точности работы и сложности реализации.

9.5. Трансляция адресов и кэш

В большинстве современных архитектур процессоры имеют поддержку виртуальной (в русской традиции называемой *математической*) памяти для многозадачных режимов работы. При этом каждая программа, исполняющаяся на машине, видит собственное упрощённое адресное пространство, содержащее код и данные только её самой, и использует его вне зависимости от местоположения в физической, «настоящей» памяти.

Поиск в кэше может происходить по физическому адресу, по виртуальному или даже по их комбинации. Схема разбиения адресов на тэг и номер сета может быть различная, но подразумевающая однозначное соответствие линий в памяти и ячеек кэша[1].

Наличие виртуальной памяти требует от процессора проведения трансляции виртуальных адресов, используемых программой, в физические адреса, соответствующие реальному местоположению в ОЗУ. При этом возникают следующие обстоятельства.

– *Задержка.* Значение физического адреса будет готово только спустя некоторое время (несколько тактов) после запроса преобразования виртуального. Потому при обращении к кэшу только по физическим адресам мы будем вынуждены ожидать завершения процесса преобразования, если не используется специальное устройство для кэширования отображений виртуальных адресов — буфер ассоциативной трансляции (*англ.* TLB, translation look-aside buffer).
– *Эффект наложения.* Несколько виртуальных адресов могут соответствовать одному физическому (*англ.* aliasing). Поэтому требуется проверка, что только одна линия с данным физическим адресом находится в кэше в любой момент времени.

По использованию виртуальной адресации кэши могут быть классифицированы следующим образом.

1. Physically indexed, physically tagged (PIPT) — *физически индексируемые и физически тегируемые*. Они просты и избегают проблем с наложением, но медленны, так как перед обращением в кэш требуется запрос физического адреса в TLB. Этот запрос может вызвать промах в TLB и дополнительное обращение в основную память перед тем как наличие данных будет проверено в кэше.

[1]В этой и последующих секциях данной главы часть текста по архитектуре кэшей адаптирована из русского и английского разделов Википедии: `http://ru.wikipedia.org/wiki/Кэш_процессора`.

2. Virtually indexed, virtually tagged (VIVT) — *виртуально индексируемые и виртуально тегируемые.* И для тегирования, и для индекса используется виртуальный адрес, благодаря чему проверки наличия данных в кэше происходят быстрее, не требуя использования трансляции. Однако возникает проблема наложения, когда несколько виртуальных адресов соответствуют одному и тому же физическому. В этом случае данные будут дважды помещены в разные ячейки, что усложняет поддержку когерентности. Другой проблемой являются *гомонимы* (*англ.* homonyms)— ситуации, когда в один и тот же виртуальный адрес (из разных пользовательских процессов) отображаются различные физические адреса. Невозможно различить их исключительно по виртуальному индексу. Возможные решения: сброс кэша при переключении между задачами (context switch), требование непересечения адресных пространств отдельных процессов, тегирование виртуальных адресов идентификатором адресного пространства, использование физических тегов.
3. Virtually indexed, physically tagged (VIPT) — *виртуально индексируемые и физически тегируемые.* Для индекса используется виртуальный адрес, а для тега — физический. Преимуществом над первым типом является меньшая задержка, поскольку можно искать кэш-линию одновременно с трансляцией адресов в TLB, однако сравнение тега всё равно задерживается до момента получения физического адреса. Преимуществом над вторым типом является безопасное обнаружение гомонимов, так как тег содержит физический адрес. Для данного типа требуется выделять больше бит для хранения тега, поскольку индексные биты используют иной тип адресации.
4. Physically indexed, virtually tagged — *физически индексируемые и виртуально тегируемые*. Такие кэши не дают существенных преимуществ и в настоящее время представляют исключительно академический интерес.

9.6. Иерархии кэшей

9.6.1. Многоуровневые системы

Может показаться очевидным, что чем больше ёмкость установленного кэша, тем эффективнее будет работать подсистема памяти из-за меньшей частоты промахов, необходимости вытеснения линий в ассоциативных сетах. Однако на практике эта величина ограничена многими факторами, в первую очередь доступной площадью на кристалле и энерговыделением. Кроме того, увеличение количества линий ведёт к росту геометрических размеров кристалла, время, затрачиваемое на передачу сигналов, растёт, нивелируя эффект от большего количества данных.

Невозможность улучшать уже присутствующие в системе устройства приводит к необходимости введения ещё одного промежуточного хранилища между ядром процессора и ОЗУ — кэш второго уровня (L2). Он располагается сразу после кэша первого уровня (L1) и характеризуется большими задержками доступа, но при этом может иметь большую ёмкость, см. рис. 9.5.

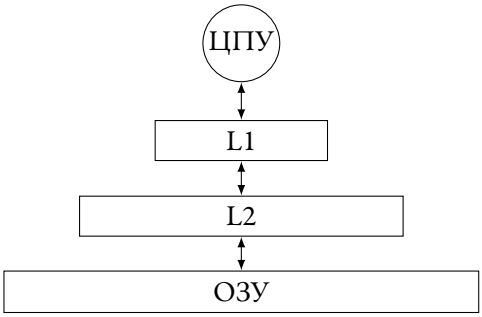


Рис. 9.5. Иерархия памяти, состоящая из двух кэшей и ОЗУ

Теперь при ситуации промаха в кэше первого уровня данные сначала ищутся во втором уровне, и лишь при втором промахе обращение идёт в память.

Естественно, что иерархию памяти можно продолжать наращивать, добавляя новые, менее быстрые и более ёмкие уровни сверхоператив-

ной памяти.

Для таких систем возникают следующие вопросы проектирования: на каких уровнях системы должна содержаться линия с определённым адресом при различных историях обращений к ней? Может ли одна линия находиться одновременно в нескольких уровнях кэша?

В одном случае могут потребовать, чтобы все данные, хранящиеся в кэше L1, находились также и в кэше L2. Такие пары кэшей называют строго *инклюзивными* (*англ.* inclusive). Другие процессоры могут не иметь подобного требования, тогда кэши называются *эксклюзивными* (*исключительными*) — данные могут быть либо в L1, либо в L2 кэше, но никогда не могут быть одновременно в обоих.

До сих пор другим процессорам не требовалось, чтобы данные в кэше первого уровня также размещались в кэше второго уровня, тем не менее они продолжают так делать. Нет общепринятого имени для этой промежуточной политики, хотя часто используется термин *инклюзивно* (*англ.* mainly inclusive).

9.6.2. Кэши инструкций и данных

Инструкции, как и данные, хранятся в общем пространстве памяти[1]; как и в случае данных, быстрый доступ к ним является необходимым условием скорости исполнения приложений. Однако множества используемых адресов и характерные шаблоны доступов для кода и данных чаще всего различны, что при их совместном кэшировании создало бы чрезмерную нагрузку и неоптимальное использование ресурсов сверхоперативной памяти. Поэтому отдельно выделяют кэш инструкций (*англ.* instruction cache, IC) и кэш данных (*англ.* data cache, DC). Отметим, что в IC приходят только запросы на чтение. При работе такой системы нужно следить за тем, чтобы запись в DC по адресам линий, находящихся в IC, приводила к их сбросу[2].

[1]Для ЭВМ архитектуры фон Неймана; тем не менее дальнейшие рассуждения в основном применимы и для систем с гарвардской архитектурой.

[2]В некоторых архитектурах для этого следует явно вызывать инструкцию, в остальных происходит автоматически.

9.7. Кэши в многопроцессорных системах

В современных ЭВМ доступ к памяти могут одновременно иметь как несколько независимых процессоров (в многоядерных системах), так и ряд периферийных устройств (в системах с DMA — direct memory access). Каждый из них может иметь свои приватные кэши, в которых хранятся копии линий, в том числе и локально модифицированных. Необходимо, чтобы все агенты имели единое представление о содержимом ОЗУ. Кроме того, на производительность иерархии памяти, связанной с одним ядром, влияет то, как линии переходят между разными его уровнями.

9.7.1. Классификация моделей согласованности доступов в память

Модель согласованности (консистентности) представляет собой некоторый договор между программами и памятью, в котором указывается, что работа модуля памяти будет корректной при соблюдении программами определённых правил. Существует их общепринятая алгоритмическая классификация [7], основанная на различии тех моментов, когда транзакция становится видимой для сторонних наблюдателей, а также разрешённого порядка записи. Ниже перечислены лишь некоторые существующие модели.

1. *Строгая согласованность.* Операция «чтение ячейки памяти с адресом x» должна возвращать значение, записанное самой последней операцией «запись» с адресом x. В системе со строгой консистентностью должно присутствовать единое абсолютное время.
2. *Последовательная согласованность* — модель, в которой результат выполнения должен быть тот же, как если бы инструкции операторов всех процессов выполнялись в некоторой последовательности, определяемой программой для этого процессора. При параллельном выполнении все процессы должны видеть одну и ту же последовательность записей в память, то есть разрешаются запаздывания для чтения.

3. *Причинная согласованность* — модель, не требующая, чтобы все процессы видели одну и ту же последовательность записей в память. Таким образом, проводится различие между потенциально-зависимыми (запись одной может зависеть от результата чтения другой ячейки) и потенциально-независимыми (параллельными) операциями записи.

На практике модели согласованности выражаются политиками записи в память, реализуемыми аппаратурой сверх- и оперативной памяти.

9.7.2. Политики записи: WT, WB, WC, UC

При записи данных в кэш должен существовать определенный момент времени, когда они будут записаны в основную память, что контролируется *политикой записи* (*англ.* write policy). Для кэшей с политикой *сквозной записи* (WT, *англ.* write through) любая запись приводит к немедленной записи в память, происходящей параллельно с записью в ячейку кэша.

Другая политика, именуемая *обратной записью* (WB, *англ.* write back), откладывает запись на более позднее время, которая производится при вытеснении подобной линейки из кэша. Таким образом, промах в кэше, использующем политику обратной записи, может потребовать двух операций доступа в память — один для сброса состояния старой линейки и другой для чтения новых данных.

Режим *совмещения записи* (WC, *англ.* write combining) позволяет сохранять данные перед записью из кэша в память в специальном буфере, сбрасываемом за одну операцию, — *всплеск* (*англ.* burst), вместо того, чтобы писать эти линии немедленно по их прибытии, что приводит к повышению средней скорости записи.

Такой режим не следует использовать для доступа к «обычной» памяти кода и данных в силу её модели *слабой согласованности* (*англ.* weak ordering), которая не гарантирует, что последовательность записей и чтений будет выполнена в ожидаемом порядке. Например, комбинация «запись—чтение—запись» некоторого адреса при совмещённой записи выльется в последовательность «чтение—запись—

запись», т.к. при чтении будет получено значение, хранящееся в памяти, а не в буфере. Для решения этой проблемы буфер может быть дополнен функциональностью полностью ассоциативного кэша — дополнительного уровня в иерархии, что приведёт к усложнению его реализации.

Однако для видеопамяти слабая согласованность не является препятствием, и поэтому политика WC может быть использована для реализации быстрых драйверов видеокарт [9].

В реальных системах различные диапазоны памяти могут иметь различные типы политик записи в память, настраиваемых с помощью атрибутов страниц ОЗУ или специальных системных регистров. В качестве одного из возможных типов может быть выбран полный *запрет кэширования* (UC, *англ.* uncacheable) — такой режим необходим для регионов, соответствующих периферийным устройствам, доступ к которым вызывает немедленные побочные эффекты.

9.7.3. Алгоритмы поддержания когерентности

Для того чтобы информация о состояниях линий в независимых кэшах соответствовала выбранной модели согласованности, используются специальные протоколы когерентности. Каждая ячейка кэша получает расширенный набор флагов, описывающий то, как её состояние соотносится с состояниями ячеек с тем же адресом, но находящихся в кэшах других агентов.

При изменении состояния некоторой линии необходимо каким-то образом сообщать о таком факте остальным кэшам. Генерируются сообщения, доставляемые по какой-либо среде внутри многопроцессорной системы (это может быть общая шина, полностью связанная сеть или сеть общего вида; сообщения могут быть адресными или широковещательными), и связанная с указанным фактом задержка влияет на скорость работы подсистемы памяти.

Было придумано много вариантов протоколов когерентности, отличающихся алгоритмами и количеством состояний, которые различаются по своей скорости работы и масштабируемости. Большинство современных протоколов представляют вариации т.н. MESI-

протокола[1].

MESI

В этой схеме каждая линия имеет одно из взаимоисключающих состояний.

- *Модифицированная* (*M*) *англ.* modified: данные в кэш-линии, помеченной как модифицированная, имеются только в одном кэше во всей системе. Линия может читаться и быть записана без опроса остальных агентов в системе.
- *Исключительная* (*эксклюзивная*) (*E*) *англ.* exclusive: кэш-линия, как и *M*-линия, хранится только в одном кэше системы, однако она ещё не подверглась изменениям; данные в ней идентичны хранящимся в ОЗУ. Поскольку эксклюзивная кэш-строка хранится только в одной кэш-памяти, она может быть считана или записана без внешних запросов. После записи в линию она отмечается как модифицированная.
- *Разделяемая* (*S*) *англ.* shared: линия может одновременно находиться в нескольких кэшах и использоваться совместно двумя или более агентами. Запросы на запись к такой линии всегда идут на внешнюю шину данных независимо от политики записи (WT или WB) , что приводит линии в других кэшах в состояние «недействительно». При этом содержимое основной памяти также обновляется.
- *Недействительная* (*I*) *англ.* invalid: линия, отмеченная как недействительная, становится логически недоступной. Это происходит в случаях, если она пуста или содержит устаревшую информацию.

Схема переходов представлена на рис. 9.6.

[1]Хороший обзор существующих протоколов с диаграммами переходов можно найти по ссылке `http://pg-server.csc.ncsu.edu/mediawiki/index.php/CSC/ECE_506_Spring_2011/ch8_mc`.

MOESI

Данный протокол является оптимизацией обычного протокола MESI. При этом флаги состояния расширяются состоянием *O* (*англ.* owned), означающим, что данные в линии одновременно и модифицированы, и разделяются (modified и shared). Указанное состояние позволяет избежать необходимости записи модифицированной линии обратно в основную память, прежде чем другие процессоры системы смогут ее прочесть. Используется микропроцессорами AMD Opteron.

– *Владелец* (Owned). Кэш-линия в этом состоянии содержит наиболее свежие, корректные данные. Описанное состояние похоже на Shared в том, что оно обозначает, что другие процессоры могут иметь копию наиболее свежих и корректных данных. Однако, в отличие от него, оно также обозначает, что в основной памяти данные могут быть устаревшими. Только один из агентов может иметь данную линию в состоянии Owned, вместе с тем он отвечает на запросы о чтении вместо памяти, с помощью этого ускоряя работу остальных агентов.

MESIF

Данный протокол актуален для систем с неоднородной когерентной памятью (*англ.* cache coherent non-uniform memory access, ccNUMA), когда каждый процессорный сокет имеет свою, ближайшую к нему, память, а обращение к адресам других сокетов происходит через промежуточные узлы и занимает в 1,5—2 раза больше времени. Введение к протоколу MESI нового состояния *F* (*англ.* forward) сделано для уменьшения объёма синхронизационного траффика в таких системах; фактически *F* — это вариант *S*, присутствующий ровно в одном кэше в любой момент времени.

– *Передовой* (*F*): означает, что данный кэш является единственным выбранным ответчиком (*англ.* designated responder) для любых запросов к линии с данным адресом. Остальные линии

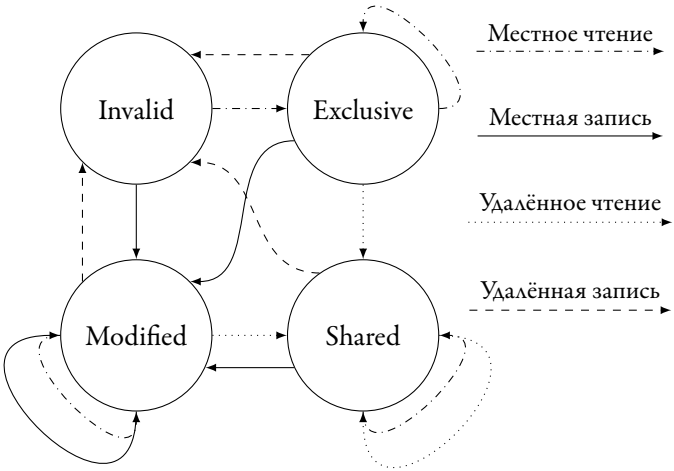


Рис. 9.6. Диаграмма переходов протокола MESI. Доступ является местным, если он был инициирован процессором данного кэша, и удалённым, если запрос возник в любом другом

(в состоянии *S*) при этом не отвечают на запросы когерентности.

9.8. Моделирование

При симуляции систем кэшей количество необходимых усилий напрямую зависит от реализуемой архитектуры и сценария использования этой модели.

9.8.1. «Честное» моделирование

Зачастую модель кэша повторяет устройство реальной аппаратуры очень близко для того, чтобы учесть все возможные причины возник-

новения задержек. При этом функционально моделируется хранение в ячейке данных, метаданных (тэги, флаги, информация для механизма вытеснения). Для иерархий кэшей учитываются передачи данных между уровнями. Для когерентных систем приходится добавлять модель среды для передачи данных и симулировать полную маршрутизацию сообщений от одних кэшей до других.

Отметим, что механизм ассоциативного поиска тэга внутри сета нереализуем программно (по крайне мере в одном потоке исполнения), и потому он заменяется более традиционными алгоритмами — последовательный перебор, деревья, хэш-таблицы; величина задержки этой процедуры, конечно, учитывается вне зависимости от деталей модельной реализации.

9.8.2. Модель задержек

Этот подход применим в том случае, когда кэши функционально «прозрачны» и их наличие выражается только в изменении времён доступов к памяти. В таком случае мы можем упростить их моделирование, избавившись от хранения линий — они всегда будут лежать в устройстве памяти, кэш не будет содержать их копии. Для вычисления же времён задержек все метаданные (тэги, сеты, флаги) ячеек моделируются так же, как и в общем случае.

Для моделирования аспектов задержки доступов, связанной с необходимостью передачи данных для поддержания когерентности, может быть использована более простая схема, отражающая функциональность, но не детали её реализации; при этом время высчитывается по заранее измеренным «расстояниям» между агентами (рис. 9.7). Такое разделение упрощает разработку модели кэшей и ускоряет исследование различных её вариантов. Однако оно неприменимо для случая моделирования транзакций, так как фактически данные никуда не перемещаются и их копии не создаются.

9.8.3. Влияние моделей кэшей на скорость симуляции

Подключение системы кэшей к функциональной модели делает её (хотя бы частично) потактовой, и естественно ожидать связанное с

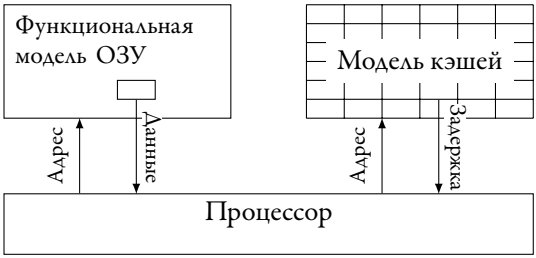


Рис. 9.7. Раздельное моделирование функциональности и длительности доступов в память. По указанному адресу данные возвращаются моделью памяти (кэши при этом не оказывают никакого влияния), задержки независимо вычисляются моделью кэшей

этим замедление симуляции. На практике скорость может упасть в тысячи раз. Поэтому часто модели кэши делают динамически отсоединяемыми от иерархии памяти, и они подключены лишь на период симуляции, когда исполняется изучаемое приложение, чтобы не тормозить предваряющие этапы загрузки операционной системы.

Однако очевидно, что эффективность кэша будет зависеть от того, заполнен ли он актуальными данными или же пуст; во втором случае многие ранние доступы будут промахами. Поэтому после подключения некоторое время тратится на *разогрев* (*англ.* warm up) системы, причем статистика и задержки игнорируются (рис. 9.8). Длительность этого процесса зависит от ёмкости кэшей и интенсивности обращений приложения к памяти; как минимум длина его не должна быть меньше длины периода измерения. Лишь после разогрева появляются основания полагать, что измеряемые на модели результаты производительности будут адекватны реальности.

9.9. Вопросы к главе 9

Вариант 1

1. Выберите правильные варианты продолжения фразы: использование кэшей при работе приложения целесообразно, если

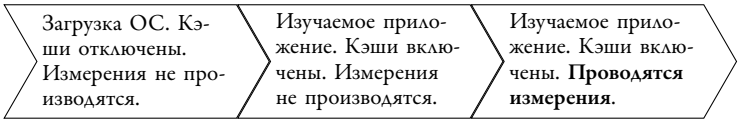


Рис. 9.8. «Разогрев» кэшей. Этап симуляции загрузки ОС проводится с помощью быстрой функциональной модели. Затем некоторое время система кэшей подключена и обрабатывает запросы, но статистика и сообщаемые ей задержки не принимаются во внимание. На последнем отрезке собираются результаты моделирования

a) программа показывает временную локальность доступов,
b) программа не обращается в оперативную память,
c) программа работает с очень большим объёмом данных,
d) программа показывает пространственную локальность доступов,
e) программа работает с объёмом данных, меньшим ёмкости кэша.

2. Выберите все правильные окончания фразы: функциональные симуляторы часто не содержат в себе модель кэша, потому что

a) они влияют только на задержки, но не на семантику инструкций,
b) всегда имеется возможность переиспользовать хозяйский кэш для нужд симуляции,
c) такие модели сильно замедляют симуляцию.

3. Данные могут попадать в кэш при следующих операциях:

a) чтение памяти (load),
b) запись в память (store),
c) арифметические операции,
d) операции с числами с плавающей запятой,
e) предвыборка данных (prefetch),
f) загрузка инструкции (fetch),
g) инвалидация линии (invalidate).

Вариант 2

1. Выберите правильные варианты окончания: линия данных с фикированным адресом
    a) всегда попадает в одну и ту же ячейку кэша,
    b) всегда попадает в один и тот же сет,
    c) может быть сохранён в любой ячейке кэша.
2. Выберите правильные варианты.
    a) Темпы роста скорости оперативной памяти и процессоров одинаковы с 80-х годов XX века.
    b) Темп роста скорости оперативной памяти опережает темпы роста скорости работы процессоров.
    c) Темп роста скорости процессоров опережает темпы роста скорости оперативной памяти.
3. Кэши необходимо симулировать даже в функциональной модели, если они используются для
    a) создания транзакционной памяти,
    b) моделирования работы ЭВМ гарвардской архитектуры,
    c) поддержания когерентности в SMP системах.

Литература

1. *Drepper Ulrich* What every programmer should know about memory //. — 2007. — Ноя. — URL: http://www.akkadia.org/drepper/cpumemory.pdf (дата обр. 20.06.2012).
2. *Drepper Ulrich* Что каждый программист должен знать о памяти / пер. С.В. Капустин, М.Ульянов, Н.Ромоданов. — Май 2012. — URL: http://rus-linux.net/lib.php?name=/MyLDP/hard/memory/memory.html (дата обр. 20.05.2012).
3. *Hatfield Donald J.*, *Gerald Jeanette* Program Restructuring for Virtual Memory // IBM Systems Journal. — 1971. — Т. 10, № 3. — 168–192.
4. *Hennessy John L.*, *Patterson David A.* Computer Architecture – A Quantitative Approach (5. ed.) — Morgan Kaufmann, 2012. — ISBN: 978-0-12-383872-8.
5. Intel® Architecture Instruction Set Extensions Programming Reference. — Intel Corporation. Июл. 2012. — С. 596. — URL: http://software.intel.com/file/41417 (дата обр. 29.11.2012).
6. *Megiddo Nimrod*, *Modha Dharmendra S.* Outperforming LRU with an Adaptive Replacement Cache Algorithm // IEEE Computer. — 2004. — Т. 37, № 4. — 58–65.
7. *Mosberger David* Memory Consistency Models. — 1993. — URL: http://citeseerx.ist.psu.edu/viewdoc/download;?doi=10.1.1.44.5376.
8. *Rajwar Ravi* Speculation-based techniques for transactional lock-free execution of lock-based programs. — Докт. дисс. University of Wisconsin – Madison, 2002.
9. Write Combining Memory Implementation Guidelines. — Intel Corporation, дек. 1998. — URL: http://download.intel.com/design/pentiumii/applnots/24442201.pdf.

10. Языки разработки моделей и аппаратуры

Talk is cheap. Show me the code.

Linus Torvalds

Поскольку симуляторы и отдельные модели устройств сами по себе являются программами, они пишутся на некотором выбранном языке (или нескольких языках) программирования, как правило, высокого уровня.

В законченном симуляторном решении, претендующем на право называться удобным и расширяемым, можно выделить функциональные блоки, различные по своему назначению (примерный состав приведён на рис. 10.1).

Рис. 10.1. Классификация компонент симулятора. Графический интерфейс и командная строка отвечают за взаимодействие с пользователем, поддержка языков сценариев (скриптов) обеспечивает средства автоматизации и расширения функциональности, модели устройств являются основным вопросом данной главы

При этом каждая подсистема может быть реализована на своём язы-

ке, наиболее адекватном требуемой функциональности. Например, графический интерфейс может быть сделан с помощью библиотек (языков) Qt (C++), AWT (Java), Tk (Tcl) и т.д. Интерпретатор сценариев, скорее всего, будет выполнен на том же динамическом языке, который предполагается использовать для написания скриптов.

Перейдём к основной части любого симулятора — моделям устройств. Для их создания можно использовать языки общего назначения, такие как Си, Си++, и писать весь симулятор и каждую модель «с нуля». Однако специфика создаваемого кода такова, что является возможным выделение некоторых абстракций в законченные модули, удобные для повторного использования, и интеграции моделей, написанных разными людьми, с помощью набора документированных интерфейсов.

По этой причине в индустрии программного моделирования можно выделить два подхода.

1. *Создание библиотек для* Си/Си++, *реализующих полезные для моделирования примитивы.* Желаемые понятия формулируются на языках общего назначения в терминах типов данных и функций или объектов, полей и методов для объектно-ориентированных языков. У разработчика, использующего такую систему, остаётся полная свобода выражения, ограниченная лишь синтаксисом применяемого языка.
2. *Использование специализированных языков описания моделей.* Концепции моделирования вносятся в ядро языка, что уменьшает сложность создаваемых кодов и повышает темп разработки. Однако требуется некоторое время и усилия для того, чтобы разработчики освоили новый язык.

Схожая задача возникает перед разработчиками аппаратуры — необходимость иметь способ описания алгоритма работы устройства. При этом для данной задачи языки общего назначения применимы ещё меньше, так как оперируют не теми базовыми абстракциями; поэтому практически всегда используются языки специализированные. Однако здесь преследуется иная цель: получить структуру, из которой возможно генерировать описание физического расположения

элементов на кристалле разрабатываемой микросхемы, пригодного для изготовления настоящих экземпляров устройства на фабрике. Генерация симулятора системы также возможна и широко используется на финальных этапах разработки, однако степень детализации симуляции часто превышает необходимую — модель получается чрезвычайно медленная. Это, в том числе, ограничивает масштаб модели одной или несколькими микросхемами, тогда как часто требуется симулятор, содержащий в себе весь комплекс целиком.

10.1. Разработка моделей

10.1.1. Требования на языки

Всякий язык высокого уровня призван сделать процесс разработки программ определённого типа менее сложным, чем это представляется при использовании языков данной машины — ассемблера или даже двоичного машинного кода. Достигается такое упрощение введением некоторых понятий, часто применяемых в программе, и переформулированием их в виде, удобном для человека. Например, в спецификации языка Си можно увидеть такие понятия, как переменная, имеющая имя (в машинном коде есть только безымянные ячейки памяти), типы, логически выстраивающие структуру данных, функции, объединяющие в себе действия, часто совершаемые совместно в фиксированном порядке и т.п.

Указанный перечень абстракций применим для очень широкого класса создаваемых программ, в том числе и программ-симуляторов аппаратуры. Однако обычно его недостаточно или он неточно ложится в рамки задачи (например, в счётчике импульсов нет переменных, зато есть регистры). Перечислим некоторые дополнительные абстракции, часто встречаемые в аппаратуре, но не в программах общего назначения.

1. *Сигналы.* Простейший сигнал — это логический уровень (единица или ноль), наблюдаемый на одном проводнике. Также сигналом может быть факт изменения (т.н. фронт) с высокого уровня на низкий или наоборот.

2. *Шины.* За один такт часто требуется передать не один бит информации, а несколько, при этом набор проводников объединяется в логическую группу, например, из 8, 16, 32 однобитных сигналов.
3. *Операции над отдельными битами чисел.* Базовая функциональность присутствует во всех языках общего назначения, но описание функций аппаратуры требует более гибких методов, как то: операции над диапазонами бит, применение масок, сдвиги, числа с длиной, не кратной восьми, и т.п.
4. *Транзакции* отражают мгновенность акта передачи нескольких сигналов по шине, а также направление сигнала (т.е. отправителя и получателя).
5. *Расширенные значения для уровней сигналов.* Кроме «высокого» и «низкого» значения в реальной аппаратуре, выходы могут быть в т.н. непроводящем (hi-Z) или в неопределённом (X) состояниях.
6. *Абстракции хранения данных* включают в себя одиночные и группы регистров разной ширины, банки памяти различной ёмкости.
7. *Карты памяти.* Обращение некоторого устройства по адресу в памяти может быть обработано различными устройствами в зависимости от значения адреса; при этом правило определения обработчика может быть сложным или динамически зависеть от сторонних условий.
8. *Побочные эффекты* от обращения к регистрам при чтении и записи.
9. *Задержки событий.* Симулируемые действия могут иметь различные задержки в симулируемом времени.

10.1.2. SystemC и TLM

SystemC — язык проектирования и верификации моделей системного уровня, реализованный в виде библиотеки C++, которая включает компоненты дискретного моделирования событий [7].

SystemC курируется организацией «Open SystemC Initiative», созданной в 1999 году, был принят ассоциацией IEEE как стандарт IEEE

1666-2005, обновленный в 2011 году как IEEE 1666-2011. Существует «эталонная» реализация данной библиотеки, однако большое количество компаний-разработчиков аппаратуры выпустили свои реализации и инструменты, основанные на спецификации и совместимые между собой.

Язык использует возможности нижележащего C++ для объектной декомпозиции разрабатываемых моделей, а также возможностей шаблонного описания используемых данных.

Несмотря на то, что первая версия SystemC решала поставленную перед ней задачу адекватного отражения необходимых понятий моделирования, было принято решение об улучшении степени абстрагирования отдельных подкомпонент и транзакций от деталей их реализации. Это расширение было создано согласно принципам **TLM** (*англ.* Transaction Level Modeling) [3] и вошло во вторую версию стандарта SystemC.

Пример кода, использующего SystemC[1].

```
#include ''systemc.h''
SC_MODULE(adder) {        // module (class) declaration
  sc_in<int> a, b;        // ports
  sc_out<int> sum;
  void do_add() {         // process
    sum.write(a.read() + b.read()); //or just sum = a + b
  }
  SC_CTOR(adder) {        // constructor
    SC_METHOD(do_add);    // register do_add to kernel
    sensitive << a << b;  // sensitivity list of do_add
  }
};
```

10.1.3. Специализированные языки

Как уже было сказано во введении к главе, зачастую создаются **предметно-ориентированные** языки (*англ.* domain specific languages), специально предназначенные для разработки аппаратуры. Это раци-

[1]Приведённые ниже примеры кода на языках SystemC, Verilog и VHDL взяты из Wikipedia.

онально при условии, что они используются как часть большого пакета для моделирования систем.

Пример: DML Другим примером языка, специально созданного для описания функциональных моделей устройств, является DML (Device Modeling Language), используемый в симуляторе Simics [4], в котором упор сделан на максимально быстрое модельное прототипирование устройств, т.е. создание заготовки устройства, не имеющей полной функциональности, но предоставляющей все внешние интерфейсы реального устройства. Этот подход позволяет реализовывать сложные системы постепенно, притом концентрироваться на самых важных функциональных аспектах в первую очередь и дописывать недостающие компоненты после. Основной тип устройств, описываемых с помощью DML, — это неисполняющие устройства, он не используется для создания моделей процессоров.

Синтаксис DML предоставляет программисту конструкции для описания банков регистров, интерфейсов и функционального поведения устройств. Использование DML для написания модели автоматически гарантирует многие декларируемые Simics свойства у получаемых моделей.

- Явное представление архитектурного состояния в *атрибутах*.
- Корректное сохранение и загрузка состояния симуляции из точек сохранения.
- Безопасная многопоточность (*англ.* thread safety).
- Поддержка правильного порядка байт данных (т.н. Endianness) при взаимодействии моделей между собой.
- Генерация текста документации из комментариев и строк описания деталей модели.

Существующий компилятор `DMLC` является т.н. source-to-source компилятором, т.е. результатом его работы являются не машинные инструкции, а промежуточный текст на языке Си, который затем обрабатывается компилятором GCC. Однако при этом двухстадийном процессе сохраняется отладочная информация о строках исходного

DML-кода, что позволяет использовать специально модифицированный вариант отладчика GDB, «понимающего» синтаксис DML для работы с исходным, а не с промежуточным кодом при отладке. Дополнительно этот подход позволяет при необходимости с помощью специальных команд языка включать код на Си в программу на DML, при этом такие куски передаются в промежуточный код без изменений.

Пример кода на DML

```
register lcr {
  parameter soft_reset_value = 0x00;
  parameter hard_reset_value = 0x00;
  field wls          [1:0] ''Word length select '';
  field stb          [2] ''Number of stop bits (0 = 1, 1 = 2)
    '';
  field pen          [3] ''Parity enable (0 = disable, 1 =
    enable)'';
  field eps          [4] ''Even parity select (0=Odd, 1=Even)
    '';
  field stick_parity [5] ''Stick parity'';
  field set_break    [6] ''Set break'';
  field dlab         [7] ''Divisor latch access bit'';
  // After a write to this register, check the contents of
    WLS and
  // set the character length mask appropriately
  method after_write (memop) {
    if      ($wls == 0) $mask = 0x1F;
    else if ($wls == 1) $mask = 0x3F;
    else if ($wls == 2) $mask = 0x7F;
    else                $mask = 0xFF;
  }
}
```

10.1.4. Языки описания набора инструкций

Отдельное внимание следует уделить разработке моделей процессоров и языкам, предназначенным для их создания.

Сложность данной задачи заключается в том, что при разработке совершенно нового устройства его авторам требуется иметь в рабочем состоянии одновременно несколько инструментов и документов из

следующего списка:

- функциональный симулятор набора инструкций;
- точная потактовая модель;
- дизассемблер машинного кода;
- компилятор с языка высокого уровня в машинный код;
- документация к аппаратуре;
- иногда необходимо также уметь генерировать синтезируемое описание новой архитектуры.

Если каждая компонента разрабатывается отдельно, то при изменении спецификации процессора (что происходит часто на ранних этапах исследования) приходится вносить изменения во всех программах и документах, что чревато ошибками и десинхронизацией инструментов, каждый из которых фактически имеет собственный «взгляд» на одно и то же устройство.

Существует несколько проектов языков, призванных решить всю описанную задачу. В качестве примера см. LISA [2, 9, 10], ISDL [5, 6] и [1]. Решение состоит в автоматической генерации необходимых инструментов из одного описания (рис. 10.2).

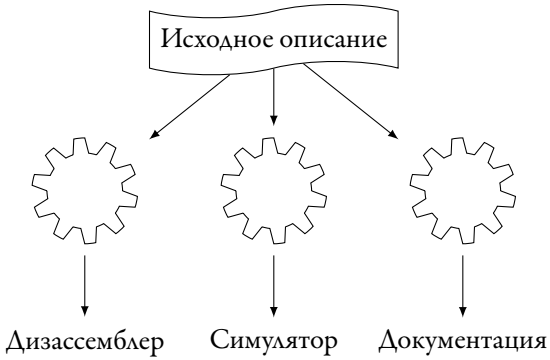


Рис. 10.2. Генерация инструментов разработки из общего описания архитектуры

Одним из недостатков этого подхода является зачастую субоптимальная скорость или качество работы получаемых инструментов; на-

пример, компилятор может генерировать не самый быстрый или компактный код, функциональная модель работает медленнее, чем могла бы будучи написанной человеком, синтезируемое описание занимает излишне много места на кристалле и т.д. Тем не менее скорость создания, модификации и степень согласованности всех инструментов часто перевешивают эти огрехи на ранних этапах, а затем, после фиксации спецификаций, все инструменты могут быть переписаны «вручную», с необходимыми оптимизациями.

Другой аспект, возникающий при создании моделей процессоров — желание иметь её с более чем одним механизмом симуляции, например, создать интерпретатор и двоичный транслятор и затем иметь возможность переключаться между ними. И снова для того, чтобы избежать рассинхронизации суб-моделей при правках спецификаций, чаще всего избирается подход, при котором они генерируются из одного описания семантики инструкций [8].

Для построения двоичных трансляторов также может использоваться преобразование описаний семантики гостевых инструкций в заготовки машинного кода, при этом исходное описание является неким метаассемблером, преобразуемым в настоящий ассемблер хозяйской архитектуры (рис. 10.3). Такой подход позволяет разработчику иметь наибольший контроль над создаваемым двоичным транслятором [11]. Существенным недостатком является полная непереносимость модели на другую хозяйскую архитектуру, т.к. при этом приходится переписывать весь код эмуляции инструкций.

10.2. Языки разработки аппаратуры

В заключение кратко познакомимся с двумя самыми популярными языками описания аппаратуры (*англ.* Hardware Definition Language, HDL), используемыми в настоящее время — Verilog и VHDL.

10.2.1. Verilog

Verilog был создан Филом Мурби и Прахбу Гоелом в 1984 году в фирме Automated Integrated Design Systems. Он был принят как стан-

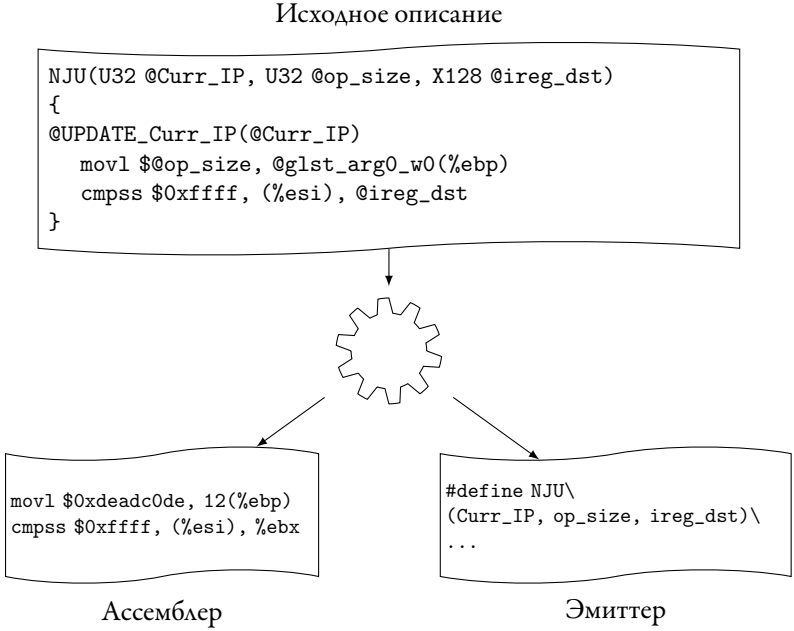


Рис. 10.3. Создание двоичного транслятора из метаассемблера. Исходное описание содержит в себе аргументы создаваемой функции и ассемблерные инструкции хозяйской архитектуры. После процесса обработки мы имеем два блока исходного кода — ассемблерный код, используемый при симуляции, а также код на Си (эмиттер), передающий первому аргументы на этапе трансляции

дарт IEEE 1364-1995. Позже дополнения к языку Verilog-95 были приняты как IEEE 1364-2001 (или Verilog-2001). Следующий вариант, Verilog 2005 (стандарт IEEE 1364-2005), добавил небольшие исправления, уточнения спецификаций и несколько новых синтаксических конструкций.

Разработчики Verilog сделали его синтаксис очень похожим на синтаксис языка C, что упрощает освоение. Язык имеет препроцессор, очень похожий на препроцессор языка C, и основные управляющие конструкции «`if`», «`while`» также подобны одноимённым конструкциям языка C.

Существует подмножество инструкций языка Verilog, называемое синтезируемым. Модули, которые написаны на этом подмножестве, называют **RTL** (*англ.* register transfer level — уровень регистровых передач). Они могут быть физически реализованы с использованием САПР-синтеза. Данные САПР по определённым алгоритмам преобразуют абстрактный исходный код на Verilog в netlist — логически эквивалентное описание, состоящее из элементарных логических примитивов (например, AND, OR, NOT, триггеры), которые доступны в выбранной технологии производства СБИС или программирования БМК и ПЛИС. Дальнейшая обработка netlist в конечном итоге порождает фотошаблоны для литографии или прошивку для FPGA.

Что отличает этот язык от обычных языков общего назначения? Во-первых, разделение всех команд на **синтезируемые**, т.е. непосредственно представляемые в аппаратуре, и на **несинтезируемые**, используемые только для отладки и симуляции.

Оператор <= в Verilog является ещё одной особенностью языка описания аппаратных средств, отличающей его от процедурных языков общего назначения. Сама операция известна как *неблокирующее присваивание*. Применение оператора не имеет внешне видимого эффекта до наступления следующего такта. Это означает, что порядок таких присваиваний в коде не может влиять на суммарный эффект функции, т.к. все они произойдут одновременно и произведут тот же результат: значения `flop1` и `flop2` будут обмениваться значениями на каждом такте.

Другой оператор присваивания, «=», является блокирующим. Когда он используется, переменная с его левой стороны обновляется немедленно. В приведённом выше примере, если бы использовался «=» вместо «<=», `flop1` и `flop2` не обменялись бы значениями. Вместо того, как и в традиционном процедурном программировании, компилятор воспринял бы это как указание сделать их содержимое одинаковым.

Пример кода на Verilog: триггер

```
module toplevel(clock,reset);
 input clock;
 input reset;
 reg flop1;
```

```
 reg flop2;
 always @ (posedge reset or posedge clock)
 if (reset)
   begin
     flop1 <= 0;
     flop2 <= 1;
   end
 else
   begin
     flop1 <= flop2;
     flop2 <= flop1;
   end
endmodule
```

10.2.2. VHDL

VHDL был разработан в 1983 г. по заказу Министерства обороны США с целью формального описания логических схем для всех этапов разработки электронных систем, начиная с модулей микросхем и заканчивая крупными вычислительными системами.

Первоначально язык предназначался для моделирования, но позднее из него было выделено синтезируемое подмножество. Средствами языка VHDL возможно проектирование на различных уровнях абстракции (поведенческом или алгоритмическом, регистровых передач, структурном) в соответствии с техническим заданием и предпочтениями разработчика. Представляется возможным выделить следующие три составные части языка: алгоритмическую, основанную на языках Ada и Pascal и придающую языку VHDL свойства процедурных языков, и проблемно ориентированную, обращающую VHDL в язык описания аппаратуры, а также объектно-ориентированную, интенсивно развиваемую в последнее время.

Пример кода на VHDL

```
-- latch template 1:
Q <= D when Enable = '1' else Q;
-- latch template 2:
process(D,Enable)
begin
  if Enable = '1' then
    Q <= D;
```

```
  end if;
end process;
```

10.3. Вопросы к главе 10

Вариант 1

1. Какое утверждение наилучшим образом характеризует термин SystemC?
    a) Компилятор языка Си с дополнениями для моделирования систем.
    b) Язык программирования, похожий на Си.
    c) Язык программирования, похожий на C++.
    d) Набор библиотек для C++.
2. Язык DML используется для разработки
    a) функциональных моделей,
    b) потактовых моделей,
    c) гибридных моделей.
3. Текущая реавлизация комилятора DMLC является
    a) компилятором типа source-to-source с промежуточным языком C++,
    b) компилятором, преобразующим исходный текст в байткод Java,
    c) компилятором типа source-to-source с промежуточным языком Си,
    d) классическим компилятором,
    e) частичным интерпретатором.
4. Закончите фразу: Языки разработки аппаратуры
    a) не используются для начального моделирования устройств, так как могут быть преобразованы только в netlist,
    b) не используются для начального моделирования устройств, так как получаемые модели очень медленны,

c) не используются для начального моделирования устройств, так как могут содержать в себе синтезируемую часть,
d) используются для начального моделирования устройств.

Вариант 2

1. Какое утверждение наилучшим образом характеризует термин TLM?
   a) Язык программирования, похожий на Си.
   b) Язык программирования, похожий на C++.
   c) Среда исполнения моделей DES.
   d) Расширение стандарта SystemC.
2. Язык DML используется для разработки
   a) неисполняющих моделей,
   b) исполняющих моделей,
   c) как исполняющих, так и неисполняющих моделей.
3. Какой способ наиболее удобен и надёжен для поддержания набора инструментов моделирования в синхронизированном состоянии при постоянном изменении входной спецификации процессора?
   a) Генерация всех инструментов из единого описания.
   b) Тщательное сравнение всех инструментов после каждого изменения одного из них.
   c) Создание одного инструмента, поддерживающего максимальное количество функций.
4. Закончите фразу: Синтезируемое подмножество языков разработки аппаратуры
   a) не может быть использовано для создания netlist и RTL-описаний,
   b) используется только для отладки моделей,
   c) используется для создания netlist и RTL-описаний.

Литература

1. A Novel Methodology for the Design of ASIPs Using a Machine Description Language / Andreas Hoffmann [и др.] // Computer-Aided Design. — 2001. — Т. 20, № 11. — 1338–1354. — URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=959863.
2. Architecture implementation using the machine description language LISA / Oliver Schliebusch [и др.] // ASPDAC 02 Proceedings of the 2002 conference on Asia South Pacific design automationVLSI Design. — 2002. — 239–244. — URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=994928.
3. *Cai L*, *Gajski D* Transaction level modeling: an overview // First IEEE ACM IFIP International Conference on Hardware Software Codesign and Systems Synthesis IEEE Cat No03TH8721. Т. 57. — ACM, 2003. — 19–24. — URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1275250.
4. DML Tutorial. — Virtutech, 2007. — URL: http://www.cs.utah.edu/~manua/sim_doc/dml-tutorial.pdf.
5. *Hadjiyiannis G*, *Hanono S*, *Devadas S* ISDL: An Instruction Set Description Language For Retargetability // Proceedings of the 34th Design Automation Conference. — 1997. — 299–302. — URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=597161.
6. *Hadjiyiannis George*, *Hanono Silvina*, *Devadas Srinivas* ISDL: An Instruction Set Description Language for Retargetability. — 1997. — URL: http://www.caa.lcs.mit.edu/~devadas/pubs/isdl.ps.
7. IEEE Standard SystemC Language Reference Manual // IEEE Computer Society IEEE Computer Society. — 2006. — Map.

8. *Larsson Fredrik* SimGen User Manual (v0.10.18). — Тех. отч. — Дек. 2001.
9. *Wahlen Oliver* C compiler aided design of application specific instruction set processors using the machine description language LISA. — Aachen: Shaker Verlag, 2004. — ISBN: 3832230351.
10. *Zivojnovic V*, *Pees S*, *Meyr H* LISA - Machine Description Language and Generic Machine Model for HW/SW Co-Design // VLSI Signal Processing IX. — 1996. — 127–136. — URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=558311.
11. *Речистов Г.С.* Методика бинарной трансляции с динамической модификацией кода в Intel Platform Simulator // Труды 52-й научной конференции МФТИ. т. 1. Вып. 1. — 2009. — 101–103.

11. Взаимодействие симуляции с внешним миром

> Мимо люди разные ходят, но никто внимания не обращает: эка невидаль, в самом деле, два олдовых человека реальность проткнуть пытаются.
>
> *Дмитрий Гайдук. Про олдовых людей*

11.1. Необходимость взаимодействия симуляции и реальности

Как и большинство прикладных программ, симуляторы не создаются по мотивам философской концепции «вещи в себе», а имеют средства взаимодействия с человеком. Пользователю необходимо как вводить некоторые данные в систему, так и получать её отклик, ради которого она и создавалась. Выделим два класса для таких активностей.

1. *Взаимодействие с моделью, повторяющее действия, осуществляемые с её физическим прототипом.* Если у реального компьютера есть клавиатура и оператор может нажимать её клавиши, нечто аналогичное должно быть у симулируемой модели. Если в конфигурации присутствует монитор или иное средство для вывода информации, оно в той или иной форме должно быть и в модели.
   Симулируемое окружение вступает во взаимодействие с настоящим миром, находящимся вне модели, со всеми вытекающими из этого обстоятельствами, например невозможностью гарантировать повторяемость симуляций: даже если человек нажимает одни и те же клавиши в различных запусках модели, длительность и паузы между ними всегда будут различными.

Рис. 11.1. Изоляция симулируемой системы от внешней среды

Отметим, что полная аналогичность пользовательского интерфейса необязательна — так, ввод с клавиатуры для модели на самом деле может идти из заранее записанного файла, расположенного на хозяйской файловой системе, а вывод видеокарты — сохраняться в сетевой поток, транслируемый на удалённый сервер.

2. *Инспектирование состояния модели и вмешательство в её работу, неосуществимые на реальной аппаратуре.* Сюда входит чтение содержимого регистров, памяти, их изменение «вручную», остановка симулируемого времени и т.п. Такие действия чаще всего не имеют соответствия в сценариях работы аппаратуры в реальном мире.

11.2. Паравиртуализационные расширения

Согласно принципу полной изоляции виртуализированного окружения находящаяся в нём программа не должна иметь возможностей отличить ситуацию, когда она выполняется на реальной аппаратуре, от работы внутри виртуального окружения. Это условие очень важно, так как позволяет использовать одно и то же программное обеспечение без необходимости модификаций и применять результаты,

полученные на модели (доказательства корректности, предсказания производительности, энергопотребления и т.п.), к реальности.

Однако часто оказывается выгодным «пробить» изоляцию для увеличения производительности симуляции, повышения удобства пользования или получения новых сценариев взаимодействия окружений. При этом гостевое приложение или ОС модифицируются таким образом, чтобы задействовать некоторую функциональность аппаратуры, присутствующую только внутри модели, но не на реальных системах. Этот приём имеет общее название **паравиртуализация**.

11.2.1. Волшебные инструкции

Для того чтобы иметь возможность совершать некоторые действия по достижении приложением некоторого этапа своей работы, можно использовать *точки остановки симуляции* (*англ.* breakpoints) по значению регистра-счётчика текущей инструкции. Иногда удобнее реагировать на исполнение некоторой специально выбранной инструкции вне зависимости от её адреса. В таком случае инструкция получает название «волшебной», т.к. с ней связаны необычные эффекты. С помощью такой инструкции мы можем помечать в приложении интересующие места, при этом на неё налагаются следующие ограничения.

- Инструкция не должна использоваться самим приложением, иначе мы будем получать ложные срабатывания и будем вынуждены как-то фильтровать их.
- Желательно, чтобы она несла минимальную семантическую нагрузку, что позволило бы использовать её во многих сценариях. Так, желательно, чтобы инструкция не была неопределённой на реальной аппаратуре, иначе приложение с ней не будет корректным вне симулятора.
- В идеале она должна сохранять неизменным содержимое памяти, работать во всех режимах процессора, предсказуемо менять счётчик инструкций, т.е. не быть инструкцией перехода.

Второму условию наилучшим образом отвечают варианты инструкции NOP — no operation. В самом деле, подобная инструкция изме-

няет архитектурное состояние минимальным образом (т.к. на её исполнение всё-таки тратится время). Однако компиляторы часто используют NOP для своих целей выравнивания кода и т.п., что не соответствует первому условию. Исключением является архитектура IA-32, где имеются две инструкции NOP — классическая однобайтовая (`0x90`) и расширенная (`0x0F 0x1F /0`) [1]; последняя может иметь длину от трёх до девяти байт и один операнд. Она является хорошим кандидатом для того, чтобы стать «волшебной» внутри симуляции.

Другой альтернативой являются недокументированные инструкции, которые не встречаются в приложениях. Однако их неопределённая функциональность на реальной аппаратуре ограничивает удобство использования.

Третьим вариантом можно считать инструкции с необычными аргументами, префиксами и другими особенностями, не встречающиеся в обычном коде. Например, для архитектуры IA-32 это может быть `CPUID` с операндами вне допустимого диапазона, инструкции с префиксами, не влияющими на её исполнение и т.п.

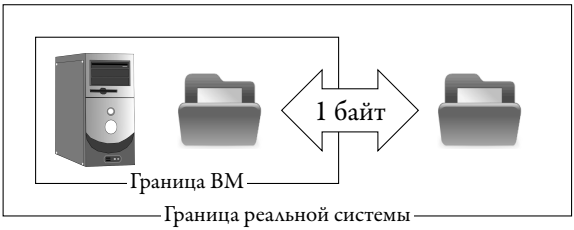


Рис. 11.2. Передача файлов с помощью магических инструкций

11.2.2. Паравиртуальные устройства

Сценарий использования «волшебной» инструкции, описанный ранее, передаёт лишь один бит информации между гостем и хозяином — это сам факт присутствия. Часто возникает необходимость передать большие объёмы информации. Несколько байт ещё могут вместиться как операнды инструкции или в регистрах процессора. Для передачи большего объёма данных за один раз необходимо иметь

больший буфер для данных. Самым очевидным представляется выделить для этого часть физического адресного пространства гостя. Чтобы чтения и записи этого диапазона обрабатывались симулятором, с ним ассоциируется псевдоустройство; запись в ассоциированную с ним память вызывает приостановку симуляции и обработку новых данных. О способе использования этого механизма рассказывается в следующей секции.

11.2.3. Ускорение ввода-вывода через периферийные устройства

Виртуализация выступает как дополнительная «обёртка» между виртуальными и реальными устройствами ввода-вывода (установленными в PCI, PCI-Express слоты расширений, подключенными к IDE и SATA шинам дисками и т.п.), что приводит к необходимости двойной (иногда тройной) передаче данных — один раз внутри модели, второй — в реальной системе. Это приводит к копированию больших объёмов данных из одной области памяти в другую без какой-либо обработки и сильно ограничивает скорость работы высокоскоростных периферийных устройств.

Для ускорения необходимо избавиться от прослойки симуляции для определённых устройств. Реализуется это модификацией гостевой операционной системы — в неё включаются драйвера паравиртуальных устройств, способных напрямую инструктировать виртуальную машину о необходимости передачи данных.

Изменённое ядро является недостатком этого подхода и существенно ограничивает его применимость для систем, коды/интерфейсы ядра которых закрыты.

11.2.4. Проброс устройства

Иногда представляется возможным перенаправлять все запросы на доступ к некоторому устройству прямо из гостя, без обработки команд симулятором. Это требует вмешательства уже в хозяйскую ОС, так как обычно она ограничивает прямой доступ к аппаратуре со стороны пользовательских приложений.

В случае паравиртуализации периферийное устройство может быть разделено между хозяином и несколькими гостями. При **пробросе** чаще всего оно полностью отдано одному из них. Пример эксклюзивного использования хозяйских устройств — проброс USB-устройств в Oracle Virtualbox; при этом оно пропадает в хозяйской системе до тех пор, пока подключено к виртуальной машине.

В современных материнских платах появилась аппаратная поддержка виртуализации периферийных устройств, что избавляет виртуальные машины от ещё одного уровня косвенности — он переносится в «железо». Что интересно, уровень позволяет также эффективно использовать одно устройство в нескольких независимых изолированных окружениях. Например, сетевая карта при этом будет иметь несколько независимых MAC-адресов, что зачастую избавляет от необходимости использования паравиртуализации. Примеры таких технологий — Intel VTd и AMD IOMMU.

11.2.5. Дополнительные каналы передачи данных

Описанные выше техники в общем случае позволяют внести в симулируемое окружение дополнительные способы обмена данными между хозяином и гостем. При этом паравиртуализация может проявляться на разных уровнях программного стека гостевой ОС, а не только на уровне драйверов устройств. Например, «волшебные инструкции» могут быть использованы для реализации утилиты копирования отдельных файлов (в общем случае — потока байт) между гостем и хозяином. Если требуется более тесная интеграция, то может быть использован драйвер файловой системы для монтирования директории машины хозяина внутри гостя, при этом все изменения, сделанные внутри гостя, становятся видны снаружи в хозяине. Такая функциональность существует во многих популярных виртуальных машинах, например в виде файловых систем `hostfs` в составе Simics или `vboxsf` из состава гостевых дополнений Virtualbox.

11.3. Интерактивные устройства для взаимодействия с пользователем

Простейшее средство для общения пользователя с системой может быть представлено двумя односторонними потоками символов. Базовое устройство, предоставляющее такой функционал, — это последовательный порт RS-232. Несмотря на приличный возраст стандарта и невысокую скорость передачи данных (до 115,2 кбит/с), он до сих пор является популярным интерфейсом для многих приложений, поддерживается практически всеми существующими операционными системами и доступен на самых ранних этапах загрузки ЭВМ. Часто именно этот вид периферийного устройства реализуется в новом симуляторе в первую очередь.

Ввиду простоты используемой абстракции передачи данных модель последовательного порта со стороны реального мира может быть использована множеством способов. Например, возможно подключение к эмулятору терминала и использование модели для интерактивного взаимодействия с пользователем. Выходящий поток будет записываться в файл, а ввод-вывод — перенаправлен в сетевой сокет, именованный канал Unix/Windows, символьное устройство или виртуальный параллельный порт или даже подключен к реальному порту хозяйской системы.

Более сложными с точки зрения моделирования устройствами ввода являются клавиатура и мышь. Существуют варианты их подключения к материнской плате через различные интерфейсы: последовательный порт, PS/2, USB... Клавиатура должна моделировать события нажатия и отпускания, а также допускать возможность одновременного нажатия нескольких клавиш. Способы подключения к реальности, как и в случае последовательного интерфейса, могут быть разнообразными.

Наиболее популярным устройством вывода графической информации является монитор. Его моделирование подразумевает создание сложного комплекса моделей: от PCI (Express) или AGP слотов на материнской плате до видеокарты, далее через цифровую начинку монитора к его дисплею. Серьёзная задача состоит в обеспечении

достаточной скорости прорисовки изображения, формируемого гостем, особенно если это трёхмерные сцены или интенсивная двухмерная графика (видео). В таких сценариях использования «программная» эмуляция только средствами центрального процессора хозяина, как правило, неспособна обеспечить комфортную для пользователя частоту обновления экрана. Решение состоит в задействовании аппаратных ресурсов хозяйской машины, осуществляемое либо через паравиртуализацию, либо пробросом PCI устройства в гостевую систему.

11.4. Диски

Под дисками мы будем подразумевать устройства хранения данных на жёстких магнитных дисках (стандарты SATA, IDE, FireWire, SCSI), твердотельных накопителях (USB-флешки, SSD), а также оптические диски (CD, DVD, Blu-ray) и теряющие актуальность гибкие диски (*англ.* floppy disks).

С моделированием дисковой подсистемы связано несколько специфичных вопросов.

- Обеспечение высокой скорости симуляции. Объём передаваемых данных для ряда приложений может быть большим, как и связанное с этим замедление модели.
- Обеспечение непосредственного хранения массивов данных. Ёмкость моделируемых дисков может достигать десятков терабайт.
- Постоянство хранилища модели. В отличие от ОЗУ и регистров устройств, жёсткие диски не теряют данные при выключении или перезагрузке компьютера. Однако сохранение состояния между запусками симуляции нарушает принцип её воспроизводимости и повторяемости.

11.4.1. Скорость

В общем случае замедление связано с необходимостью многократного копирования данных между симуляцией и реальной системой

хранения. Как было описано раньше, сценарии решения этой проблемы заключаются в паравиртуализации или в пробросе устройства внутрь симуляции.

11.4.2. Форматы хранения

Поскольку диск представляет собой устройство с произвольным доступом, естественная форма хранения его данных — это файл в хозяйской системе, блоки содержимого которого соответствуют данным гостевого диска.

В простейшем случае файл должен содержать просто копию байт-в-байт всего содержимого реального диска, это т.н. «сырой» (*англ.* raw) образ диска (рис. 11.3). При этом все сектора гостевого диска расположены последовательно в том же порядке, который они имели бы в реальности. Преимущество такого способа хранения — его простота и универсальность. Создание образа диска из существующего физического элементарно[1]. Практически все существующие симуляторы поддерживают образы в «сыром» формате. Основной его недостаток — нерациональность использования ресурсов хозяина. Например, симуляция установки ОС может занять 1 Гбайт места на образе диска в 100 Гбайт; результирующий образ диска будет занимать 100 Гбайт, при этом 99% его будут потеряны для хозяйской системы впустую.

Многие симуляторы поддерживают более компактные способы хранения, в которых в файл записываются только изменённые сектора диска; заголовочная часть файла содержит список этих секторов и их местоположение (рис. 11.4). Как правило, каждая программа имеет свой формат и иногда поддерживает другие или позволяет конвертировать их друг в друга. Примеры: Qcow2 [3] (Qemu), VDI (Oracle Virtualbox), VMDK [5] (VMware ESX), VHD (Microsoft VirtualPC), HDD (Parallels Desktop), CRAFF (Wind River Simics).

Некоторые форматы поддерживают прозрачное сжатие записанных данных, когда вместо обычной копии каждого гостевого сектора хранится его представление, полученное применением одного из

[1]Например, с помощью Unix-утилиты dd.

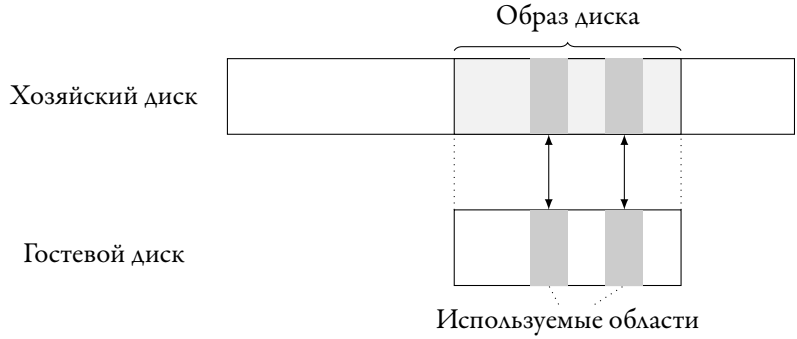


Рис. 11.3. «Сырой» образ диска. Хранится каждый сектор, даже если он не задействоован внутри гостевой системы, что приводит к нерациональному расходованию места

известных алгоритмов сжатия без потерь, например Gzip, Bzip2 или LZMA [4].

Для образов оптических дисков, которые в большинстве случаев являются носителями с данными только для чтения, используются сырые образы. Чаще всего они именуются ISO-образами по имени стандарта используемой на них файловой системы ISO 9660[1].

Из-за своего небольшого размера (меньше 3 Мбайт) образы гибких дисков хранятся в raw-формате.

11.4.3. Сохранение состояния дисков

Зачастую нежелательно модифицировать исходный файл образа диска: экспериментальное ПО/вирусы/ошибки пользователя внутри симулятора могут сделать его неработоспособным, или же желательно впоследствии запускать симуляцию из первоначального состояния.

Для таких целей в большинстве симуляторов существует опция: все изменённые секторы сохранять не в оригинальном, а в дополнительном **разностном** файле (также называемом **дельтой**). Для модели-

[1]Хотя это не единственный стандарт для оптических дисков; альтернативой является UDF (universal disk format), призванный обойти многие ограничения ISO 9660.

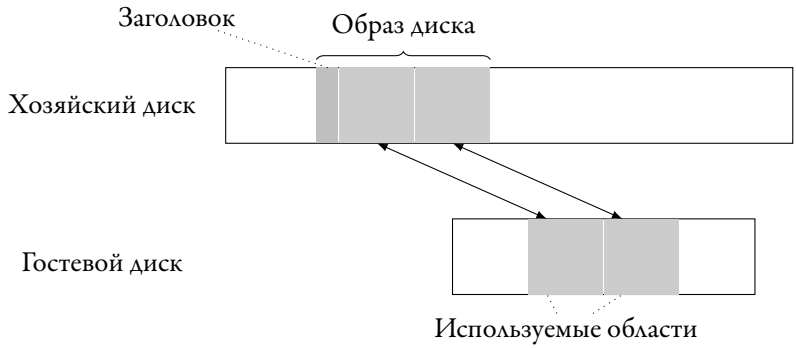


Рис. 11.4. Компактный формат образа диска. Хранятся только данные используемых секторов гостевого диска, а также служебная информация, описывающая их местоположение

руемого приложения указанная схема абсолютно прозрачна — внутри симуляции изменения видны там, где и должны быть. Однако после выключения симуляции допустимо отбросить дельту и использовать оригинальный образ. В случае появления желания зафиксировать внесённые в результате последней симуляционной сессии правки — следует воспользоваться утилитами слияния оригинального образа и дельты.

Развивая эту идею, можно вообразить себе схему с несколькими дельтами, полученными на разных этапах симуляции (и даже «дельты к дельтам»), одновременно наложенными на диск. Таким образом, можно иметь множество снимков состояний дискового хранилища, суммарно занимающие места меньше, чем занимали бы отдельные полные копии.

11.5. Сеть

Современный Интернет был спроектирован так, чтобы обеспечивать обмен информацией между системами совершенно различной структуры, масштабов, используемых технологий. Взаимодействие симуляции и реальности через сеть является наиболее «естествен-

ным» подходом, поскольку у агентов имеются лишь самые общие предположения о свойствах друг друга.

Существует семь уровней абстракции данных OSI ISO [6]. Ниже описана возможная классификация способов взаимодействия хозяина и гостя по их соответствию некоторым из этих уровней. Отметим, что такая классификация не является строгой и приводится для подчёркивания различий между ними.

- *Физический уровень*. Для прозрачного использования гостевыми системами хозяйской сети необходимо иметь работающую модель сетевой карты (*англ.* network interface card, NIC). Она может быть соединена с другими чисто программными моделями сетевых устройств (NIC, маршрутизаторами и т.д.) внутри симуляции или представлять собой проброшенную в симуляцию физическую NIC хозяйской системы.
- *Канальный уровень*. Драйвер, именуемый TAP [2], обеспечивает модель сетевой карты и работает с кадрами Ethernet. Он устанавливается в хозяине. После этого хозяйская система может через новый интерфейс псевдо NIC взаимодействовать с гостевой системой (при условии правильной настройки адресов и маршрутизации). Для того чтобы гость был представлен в том сегменте реальной сети, в котором находится хозяин, необходимо объединить в мост (*англ.* bridge) одну из реальных и виртуальную карты. Очевидный недостаток — хозяин теряет одно устройство NIC на каждой симуляции.
- *Сетевой уровень*. TUN-драйвер обеспечивает более высокоуровневую модель и оперирует IP-пакетами, а не кадрами Ethernet. Такое соединение может использоваться для создания сетевого туннеля, например VPN. С точки зрения целей симуляции TUN не предоставляет существенных преимуществ над TAP.
- *Транспортный уровень*. Для одностороннего доступа гостей к внешней сети, как правило, используется решение, в котором он размещается в отдельной подсети, а хозяин выступает как NAT-сервер. Все исходящие TCP-соединения выглядят как исходящие от хозяина. Недостаток данного подхода состоит в невоз-

можности открыть соединение извне с симуляцией. Для обхождения этой проблемы используют переадресацию фиксированных номеров портов системы хозяина. Соединения на такие порты автоматически перенаправляются внутрь симуляции.

– *Уровень приложений*. Для обеспечения работы некоторых гостевых приложений достаточно имитировать наличие в сети сервера, способного отвечать на их запросы. Например, виртуальный DHCP-сервер может раздавать IP-адреса симулируемым картам, виртуальный роутер обеспечивать NAT, виртуальный NFS- или Samba-серверы — предоставлять доступ к части файловой системы хозяина.

11.6. Вопросы к главе 11

Вариант 1

1. Выберите свойства, которые должны выполняться для идеальной «волшебной» инструкции:
   a) должна быть допустимой во всех режимах работы процессора,
   b) должна быть привилегированной,
   c) не должна иметь явных аргументов,
   d) не должна генерироваться обычными компиляторами,
   e) не должна вызывать эффектов (т.е. быть NOP),
   f) не должна иметь неявных аргументов.
2. Какая инструкция для архитектуры IA-32 не может быть использована как волшебная?
   a) `CPUID` — идентификация процессора,
   b) `INT` — программное прерывание,
   c) `NOP` — пустая операция.
3. Для какой из перечисленных ниже операционных систем паравиртуализационные расширения сложно писать из-за закрытости исходного кода?
   a) Microsoft Windows,

b) GNU/Linux,
c) FreeBSD.

4. В чём состоят недостатки сырого формата дисков?

   a) невозможность случайного доступа к секторам диска,
   b) нерациональное расходование дискового пространства гостя,
   c) нерациональное расходование дискового пространства хозяина,
   d) отсутствие публичной документации на формат.

Вариант 2

1. Почему передача большого объёма данных между гостём и хозяином с помощью волшебной инструкции неэффективна?

   a) за один раз можно передать только несколько байт,
   b) побочные эффекты множества волшебных инструкций подряд могут нарушить работу гостя,
   c) побочные эффекты множества волшебных инструкций подряд могут нарушить работу хозяина,
   d) направление передачи данных ограничено только из гостя в хозяина.

2. Назовите приём виртуализации, в котором гостевое приложение модифицируются таким образом, чтобы задействовать некоторую функциональность аппаратуры, присутствующую только внутри модели, но не на реальных системах?

   a) гиперсимуляция,
   b) метавиртуализация,
   c) паравиртуализация,
   d) изоляция.

3. Дайте определение термину «проброс устройства»:

   a) передача устройства в эксклюзивное пользование нескольким гостям,
   b) передача устройства в эксклюзивное пользование хозяину,

c) передача устройства в эксклюзивное пользование единственному гостю.

4. Для чего используются разностные файлы?

a) хранение изменений гостевого диска за время работы симуляции,
b) сжатие оригинального образа гостевого диска для того, чтобы он занимал меньше места,
c) прозрачное шифрование оригинального образа гостевого диска,
d) расширения размера гостевого диска в случае, когда старый полностью заполнен.

Литература

1. Intel® 64 and IA-32 Architectures Software Developer's Manual. Volume 2A. — Intel Corporation.
2. *Krasnyansky Maxim* Universal TUN/TAP device driver. — 2000. — URL: http://www.kernel.org/pub/linux/kernel/people/marcelo/linux-2.4/Documentation/networking/tuntap.txt.
3. *McLoughlin Mark* The QCOW2 Image Format. — 2008. — URL: http://people.gnome.org/~markmc/qcow-image-format.html (дата обр. 22.10.2012).
4. *Sayood K.* Lossless Compression Handbook. — Elsevier Science, 2002. — (Communications, Networking and Multimedia). — ISBN: 9780080510491. — URL: http://books.google.co.uk/books?id=LjQiGwyabVwC.
5. Virtual Machine Disk Format (VMDK). — VMware, 2012. — URL: http://www.vmware.com/technical-resources/interfaces/vmdk.html (дата обр. 22.10.2012).
6. ГОСТ Р ИСО/МЭК 7498-1-99 «Взаимосвязь открытых систем. Базовая эталонная модель». — 1999.

12. Современная виртуализация

> When this sort of deliberate disconnection from reality happens with people, it generally goes by names like deceit, fraud, misrepresentation, or simply lying. When it happens with computers, it's called virtualization.
>
> *Harlan McGhan*

12.1. Введение

Для понимания того, каким образом современные вычислительные системы, их новые свойства, инструкции и режимы призваны поддерживать виртуализацию, в этой главе мы рассмотрим теоретические основания возможности её **эффективной** реализации.

Виртуализация представляла интерес ещё до изобретения микропроцессора, во времена преобладания больших систем — мейэнфреймов, ресурсы которых были очень дорогими, и их простой был экономически недопустим. Виртуализация позволяла повысить степень утилизации таких систем, при этом избавив пользователей и прикладных программистов от необходимости переписывать своё ПО, так как с их точки зрения виртуальная машина была идентична физической. Пионером в этой области являлась фирма IBM с мэйнфреймами System/360, System/370, созданными в 1960–1970-х гг.

12.2. Классический критерий виртуализуемости

Неудивительно, что критерии возможности создания эффективного монитора виртуальных машин были получены примерно в то же время. Они сформулированы в классической работе 1974 г. Жеральда Попека и Роберта Голдберга «Formal requirements for virtualizable third generation architectures» [9]. Рассмотрим её основные предпосылки и сформулируем её основной вывод.

12.2.1. Модель системы

В дальнейшем используется упрощённое представление «стандартной» ЭВМ, состоящей из (одного) центрального процессора и линейной однородной оперативной памяти. Периферийные устройства, а также средства взаимодействия с ними опускаются. Процессор поддерживает два режима работы: режим супервизора, используемый операционной системой, и режим пользователя, в котором исполняются прикладные приложения. Память поддерживает режим сегментации, используемый для организации виртуальной памяти.

Выдвигаемые требования на монитор виртуальных машин (ВМ):

Изоляция — каждая виртуальная машина должна иметь доступ только к тем ресурсам, которые были ей назначены. Она не должна иметь возможности повлиять на работы как монитора, так и других ВМ.

Эквивалентность — любая программа, исполняемая под управлением ВМ, должна демонстрировать поведение, полностью идентичное её исполнению на реальной системе, *за исключением* эффектов, вызванных двумя обстоятельствами: различием в количестве доступных ресурсов (например, ВМ может иметь меньший объём памяти) и длительностями операций (из-за возможности разделения времени исполнения с другими ВМ).

Отметим, что для симуляторов в общем смысле эквивалентность не является требованием, т.к. в случаях, когда хозяйская и гостевая архитектуры не совпадают, поведение гостя и хозяина различаются.

Эффективность — в оригинальной работе условие сформулировано следующим образом: «статистически преобладающее подмножество инструкций виртуального процессора должно исполняться напрямую хозяйским процессором, без вмешательства монитора ВМ». Другими словами, значительная часть инструкций должна симулироваться в режиме прямого исполнения. Требование эффективности является самым неоднозначным из трёх перечисленных требований, и мы вернёмся к нему в секции 12.3.

В случае симуляторов, основанных на интерпретации инструкций, условие эффективности не выполняется, т.к. каждая инструкция гостя требует обработки симулятором.

12.2.2. Классы инструкций

Состояние процессора содержит минимум три регистра: M, определяющий, находится ли он в режиме супервизора s или пользователя u, P — указатель текущей инструкции и R — состояние, определяющее границы текущего сегмента памяти[1]. При исполнении каждая инструкция i в общем случае может изменить как (M, P, R), так и память E, т.е. она является функцией преобразования

$$(M_1, P_1, R_1, E_1) \overset{i}{\mapsto} (M_2, P_2, R_2, E_2).$$

Память E состоит из фиксированного числа ячеек, к которым можно обращаться по их номеру t, например, $E[t]$. Размер памяти и ячеек для данного рассмотрения несущественен.

Считается, что для некоторых входных условий инструкция вызывает исключение **ловушки** (*англ.* trap), если в результате её исполнения содержимое памяти не изменяется, кроме единственной ячейки $E[0]$, в которую помещается предыдущее состояние процессора (M_1, P_1, R_1). Новое состояние процессора (M_2, P_2, R_2) при этом копируется из $E[1]$. Другими словами, ловушка позволяет сохранить полное состояние программы на момент до начала исполнения её последней инструкции и передать управление обработчику, в случае обычных систем обычно работающему в режиме супервизора и призванного обеспечить дополнительные действия над состоянием системы, а затем вернуть управление в программу, восстановив состояние из $E[0]$.

Далее, ловушки могут иметь два признака.

1. Вызванные попыткой изменить состояние процессора (ловушка **потока управления**).

[1] В простейшем случае $R = (l, b)$, где l — адрес начала диапазона, b — его длина.

2. Обращения к содержимому памяти, выходящему за пределы диапазона, определённого в R (ловушка **защиты памяти**).

Отметим, что эти признаки не взаимоисключающие. То есть результатом исполнения могут быть одновременно ловушка потока управления и защиты памяти.

Машинные инструкции рассматриваемого процессора можно классифицировать следующим образом:

Привилегированные (*англ.* privileged). Инструкции, исполнение которых с $M = u$ всегда вызывает ловушку потока управления. Другими словами, такая инструкция может исполняться только в режиме супервизора, иначе она обязательно вызывает исключение.

Служебные (*англ.* sensitive[1]). Класс состоит из двух подклассов. 1. Инструкции, исполнение которых закончилось без ловушки защиты памяти и вызвало изменение M и/или R. Они могут менять режим процессора из супервизора в пользовательский или обратно или изменять положение и размер доступного сегмента памяти. 2. Инструкции, поведение которых в случаях, когда они не вызывают ловушку защиты памяти, зависят или от режима M, или от значения R.

Безвредные (*англ.* innocuous). Не являющиеся служебными. Самый широкий класс инструкций, не манипулирующие ничем, кроме указателя инструкций P и памяти E, поведение которых не зависит от того, в каком режиме или с каким адресом в памяти они расположены.

12.2.3. Достаточное условие построения монитора ВМ

Соблюдение трёх сформулированных выше условий возможности построения монитора виртуальных машин даётся в следующем предложении: **множество служебных инструкций является подмножеством привилегированных инструкций** (рис. 12.1). Опуская фор-

[1]Установившего русского термина для этого понятия нет. Иногда в литературе встречается перевод «чувствительные» инструкции.

мальное доказательство теоремы 1 из статьи, отметим следующие обстоятельства.

- Изоляция обеспечивается размещением монитора в режиме супервизора, а ВМ — только в пользовательском. При этом последние не могут самовольно изменить системные ресурсы (M, R) — попытка вызовет ловушку потока управления на служебной инструкции и переход в монитор, а также память $E[0, 1]$ из-за того, что конфигурация R не допускает этого, и процессор выполнит ловушку защиты памяти.
- Эквивалентность доказывается тем, что безвредные инструкции выполняются одинаково вне зависимости от того, присутствует ли в системе монитор или нет, а служебные всегда вызывают исключение и интерпретируются. Отметим, что даже в описанной выше простой схеме проявляется первое ослабляющее условие: даже без учёта памяти, необходимой для хранения кода и данных гипервизора, объём доступной для ВМ памяти будет как минимум на две ячейки меньше, чем имеется у хозяйской системы.
- Эффективность гарантируется тем, что все безвредные инструкции внутри ВМ исполняются напрямую, без замедления. При этом подразумевается, что их множество включает в себя «статистически преобладающее подмножество инструкций виртуального процессора».

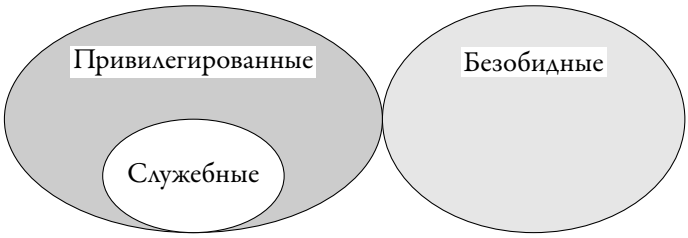


Рис. 12.1. Выполнение условия виртуализуемости. Множество служебных инструкций является подмножеством привилегированных

12.3. Ограничения применимости критерия виртуализуемости

Несмотря на простоту использованной модели и полученных из неё выводов, работа Голдберга и Попека является актуальной до сих пор. Следует отметить, что несоблюдение описанных в ней условий вовсе не делает создание или использование виртуальных машин на некоторой архитектуре принципиально невозможным, и есть практические примеры реализаций, подтверждающие это. Однако соблюсти оптимальный баланс между тремя свойствами: изоляцией, эквивалентностью и эффективностью, — становится невозможным. Чаще всего расплачиваться приходится скоростью работы виртуальных машин из-за необходимости тщательного поиска и программного контроля за исполнением ими служебных, но не привилегированных инструкций, так как сама аппаратура не обеспечивает этого (рис. 12.2). Даже единственная такая инструкция, исполненная напрямую ВМ, угрожает стабильной работе монитора, и поэтому он вынужден сканировать весь поток гостевых инструкций.

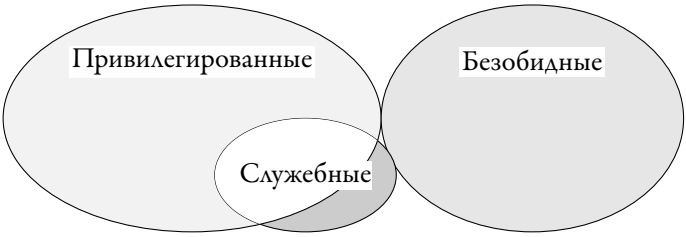


Рис. 12.2. Невыполнение условия виртуализуемости. Служебные, но не привилегированные инструкции требуют реализации сложной логики в мониторе

В самой работе [9] присутствуют как явно указанные упрощения исследуемой структуры реальных систем (отсутствие периферии и системы ввода-вывода), так и неявные предположения о структуре исполняемых гостевых программ (почти полностью состоящих из безвредных инструкций) и хозяйских систем (однопроцессорность).

Рассмотрим теперь данные ограничения более детально, а также

предложим, каким образом можно расширить степень применимости критерия к дополнительным ресурсам, требующим виртуализации, и таким образом повысить его практическую ценность для архитекторов новых вычислительных систем.

12.3.1. Структура гостевых программ

Для эффективной работы программ внутри ВМ необходимо, чтобы большая часть их инструкций являлись безвредными. Как правило, это верно для прикладных приложений. Операционные системы, в свою очередь, предназначены для управления ресурсами системы, что подразумевает использование ими привилегированных и служебных инструкций, и монитору приходится их перехватывать и интерпретировать с соответствующим падением производительности. Поэтому в идеале в наборе инструкций должно быть как можно меньше привилегированных для того, чтобы частота возникновения ловушек была минимальной.

12.3.2. Периферия

Поскольку периферийные устройства являются служебным ресурсом ЭВМ, очевидно, что для обеспечения условий изоляции и эквивалентности необходимо, чтобы все попытки доступа к ним были контролируемы монитором ВМ так же, как они контролируются в многозадачной операционной системе её ядром. В настоящее время доступ к устройствам чаще всего производится через механизм отражения их в физической памяти системы (*англ.* memory mapped I/O), что означает, что внутри монитора это чтение/запись некоторых регионов должно или вызывать ловушку защиты памяти, или быть не служебным, т.е. не вызывать ловушку и не влиять на состояние неконтролируемым образом.

Интенсивность взаимодействия приложений с периферией может быть различна и определяется их функциональностью, что сказывается на их замедлении при виртуализации. Кроме того, монитор ВМ может делать различные классы периферии, присутствующей на хозяине, доступными внутри нескольких ВМ различными способами.

Выделенное устройство — устройство, доступное исключительно внутри одной гостевой системы. Примеры: клавиатура, монитор.

Разделяемое — общее для нескольких гостей. Такое устройство или имеет несколько частей, каждая из которых выделена для нужд одного из них (*англ.* partitioned mode), например, жёсткий диск с несколькими разделами, или подключается к каждому из них поочерёдно (*англ.* shared mode). Пример: сетевая карта.

Полностью виртуальное — устройство, отсутствующее в реальной системе (или присутствующее, но в ограниченном количестве) и моделируемое программно внутри монитора. Примеры: таймеры прерываний — каждый гость имеет собственный таймер, несмотря на то, что в хозяйской системе есть только один, и он используется для собственных нужд монитора.

12.3.3. Прерывания

Прерывания являются механизмом оповещения процессора о событиях внешних устройств, требующих внимания операционной системы. В случае использования виртуальных машин монитор должен иметь возможность контролировать доставку прерываний, так как часть или все из них необходимо обрабатывать именно внутри монитора. Например, прерывание таймера может быть использовано им для отслеживания/ограничения использования гостями процессорного времени и для возможности переключения между несколькими одновременно запущенными ВМ. Кроме того, в случае нескольких гостей заранее неясно, какому из них следует доставить прерывание, и принять решение должен монитор.

Простейшее решение, обеспечивающее изоляцию, — это направлять **все** прерывания в монитор ВМ. Эквивалентность при этом будет обеспечиваться им самим: прерывание при необходимости будет доставлено внутрь гостя через симуляцию изменения его состояния. Монитор может дополнительно создавать виртуальные прерывания, обусловленные только логикой его работы, а не внешними событиями. Однако эффективность такого решения не будет оптимальной. Как правило, реакция системы на прерывание должна произойти в

течение ограниченного времени, иначе она потеряет смысл для внешнего устройства или будет иметь катастрофические последствия для системы в целом. Введение слоя виртуализации увеличивает задержку между моментом возникновения события и моментом его обработки в госте по сравнению с системой без виртуализации. Более эффективным является аппаратный контроль за доставкой прерываний, позволяющий часть из них сделать безвредными для состояния системы и не требовать каждый раз вмешательства программы монитора.

12.3.4. Многопроцессорные системы

Синхронизация и виртуализация

Введение в рассмотрение нескольких хозяйских и гостевых процессоров оставляет условие эффективной виртуализуемости в силе. Однако необходимо обратить внимание на выполнение условий эффективности работы многопоточных приложений внутри ВМ. В отличие от однопоточных, для них характерны процессы синхронизации частей программы, исполняющихся на различных виртуальных процессорах. При этом все участвующие потоки ожидают, когда все они достигнут заранее определённой точки алгоритма, т.н. барьера. В случае виртуализации системы один или несколько гостевых потоков могуть оказаться неактивными, вытесненными монитором, из-за чего остальные будут попусту тратить время.

Примером такого неэффективного поведения гостевых систем является синхронизация с задействованием циклических блокировок (*англ.* spin lock) внутри ВМ [11]. Будучи неэффективной и поэтому неиспользуемой для однопроцессорных систем, в случае нескольких процессоров она является легковесной альтернативой классическим замкам (*англ.* lock), используемым для входа в критические секции параллельных алгоритмов. Чаще всего они используются внутри операционной системы, но не пользовательских программ, так как только ОС может точно определить, какие из системных ресурсов могут быть эффективно защищены с помощью циклических блокировок. Однако в случае виртуальной машины планированием ресурсов занимается не ОС, а монитор ВМ, который в общем случае не осведомлён

о них и может вытеснить поток, способный освободить ресурс, тогда как второй поток будет выполнять циклическую блокировку, бесполезно тратя процессорное время. Оптимальным решением при этом является деактивация заблокированного потока до тех пор, пока нужный ему ресурс не освободится.

Существующие решения для данной проблемы описанны ниже.

1. Монитор ВМ может пытаться детектировать использование циклических блокировок гостевой ОС. Это требует анализа кода перед исполнением, установки точек останова по адресам замка. Способ не отличается универсальностью и надёжностью детектирования.
2. Гостевая система может сигнализировать монитору о намерении использовать циклическую блокировку с помощью специальной инструкции. Способ более надёжный, однако требующий модификации кода гостевой ОС.

Прерывания в многопроцессорных системах

Наконец, отметим, что схемы доставки и обработки прерываний в системах с несколькими процессорами также более сложны, и это приходится учитывать при создании монитора ВМ для таких систем, при этом его эффективность может оказаться ниже, чем у однопроцессорного эквивалента.

12.3.5. Преобразование адресов

Модель машинных инструкций, использованная ранее для формулировки основного утверждения данной главы, использовала простую линейную схему трансляции адресов, основанную на сегментации, популярную в 70-х годах прошлого века. Она является вычислительно простой, не изменяется при введении монитора ВМ, и поэтому анализа влияния механизма преобразования адресов на эффективность не производилось.

В настоящее время механизмы страничной виртуальной памяти и применяют нелинейное преобразование виртуальных адресов поль-

зовательских приложений в физические адреса, используемые аппаратурой. Участвующий при этом системный ресурс — регистр-указатель адреса таблицы преобразований[1]. В случае использования ВМ этот указатель необходимо виртуализовать, так как у каждой гостевой системы содержимое регистра своё, как и положение/содержимое таблицы. Стоимость программной реализации этого механизма внутри монитора высока, поэтому приложения, активно использующие память, могут терять в эффективности при виртуализации.

Для решения этой проблемы используется двухуровневая аппаратная трансляция адресов (рис. 12.3). Гостевые ОС видят только первый уровень, тогда как генерируемый для них физический адрес в дальнейшем транслируется вторым уровнем в настоящий адрес.

TLB

Другой ресурс ЭВМ, отвечающий за преобразование адресов, — это буфер ассоциативной трансляции (*англ.* translation lookaside buffer, TLB), состоящий из нескольких записей. Каждая гостевая система имеет своё содержимое TLB, поэтому при смене активной ВМ или переходе в монитор он должен быть сброшен. Это негативно сказывается на производительности систем, так как восстановление его содержимого требует времени, в течение которого приходится использовать менее эффективное обращение к таблице трансляций, расположенной в памяти.

Решение состоит в разделении ресурсов TLB между всеми системами [12]. Каждая строка буфера ассоциируется с идентификатором — тэгом, уникальным для каждой ВМ. При поиске в нём аппаратурой учитываются только строки, тэг которых соответствует текущей ВМ.

Преобразование адресов для периферийных устройств

Кроме процессоров к оперативной памяти напрямую могут обращаться и периферийные устройства — с помощью технологии DMA

[1]Чаще всего на практике используется несколько таблиц, образующих иерархию, имеющую общий корень.

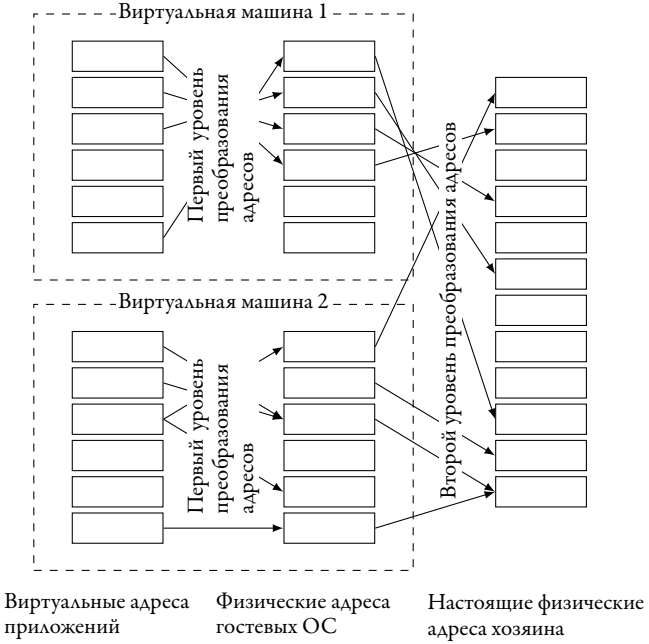


Рис. 12.3. Двухуровневая трансляция адресов. Первый уровень контролируется гостевыми ОС, второй — монитором виртуальных машин

(*англ.* direct memory access). При этом обращения в классических системах без виртуализации идёт по физическим адресам. Очевидно, что внутри виртуальной машины необходимо транслировать такие адреса, что превращается в накладные расходы и понижение эффективности монитора.

Решение состоит в использовании устройства IOMMU (*англ.* Input output memory management unit), позволяющего контролировать обращения хозяйских устройств к физической памяти.

12.3.6. Расширение принципа

Расширим условие виртуализуемости, заменив в нём слово «инструкция» на «операция»: **множество служебных *операций* явля-**

ется подмножеством привилегированных. При этом под операцией будем подразумевать любую архитектурно определённую активность по чтению или изменению состояния системы, в том числе инструкции, прерывания, доступы к устройствам, преобразования адресов и т.п.

При этом условие повышения эффективности виртуализации будет звучать следующим образом: **в архитектуре системы должно присутствовать минимальное число служебных операций**. Достигать его можно двумя способами: переводя служебные инструкции в разряд безвредных или уменьшая число привилегированных. Для этого большинство архитектур пошло по пути добавления в регистр состояния M нового режима r — режима монитора ВМ (*англ.* root mode). Он соотносится с режимом s так, как s — с u; другими словами, *обновлённый* класс привилегированных инструкций теперь вызывает ловушку потока управления, переводящую процессор из s в r.

12.4. Статус поддержки в современных архитектурах

Рассмотрим основные современные архитектуры вычислительных систем, используемых на серверах, рабочих станциях, а также во встраиваемых системах, с точки зрения практической реализации описанных выше теоретических принципов. См. также серию статей [5—7].

12.4.1. IBM POWER

Компания IBM была одной из первых, выведших архитектуру с аппаратной поддержкой виртуализации на рынок серверных микропроцессоров в серии POWER4 в 2001 году. Она предназначалась для создания изолированных логических разделов (*англ.* logical partitions, LPAR), с каждым из которых ассоциированы один или несколько процессоров и ресурсы ввода-вывода. Для этого в процессор был добавлен новый режим гипервизора к уже присутсвовавшим режимам супервизора и пользователя. Для защиты памяти каждый LPAR огра-

ничен в режиме с отключенной трансляцией адресов и имеет доступ лишь к небольшому приватному региону памяти; для использования остальной памяти гостевая ОС обязана включить трансляцию, контролируемую монитором ВМ.

В 2004 году развитие этой архитектуры, названное POWER5, принесло серьёзные усовершенствования механизмов виртуализации. Так, было добавлено новое устройство таймера, доступное только для монитора ВМ, что позволило ему контролировать гостевые системы более точно и выделять им процессорные ресурсы с точностью до сотой доли от процессора. Также монитор ВМ получил возможность контролировать адрес доставки прерываний — в LPAR или в гипервизор. Самым важным же нововведением являлся тот факт, что присутствие гипервизора являлось обязательным — он загружался и управлял системными ресурсами, даже если в системе присутствовал единственный LPAR-раздел. Поддерживаемые ОС (AIX, Linux, IBM i) были модифицированы с учётом этого, чтобы поддерживать своеобразную паравиртуализационную схему. Для управления устройствами ввода-вывода один (или два, для балансировки нагрузки) из LPAR загружает специальную операционную систему — virtual I/O server (VIOS), предоставляющую эти ресурсы для остальных разделов.

12.4.2. SPARC

Компания Sun, развивавшая системы UltraSPARC и ОС Solaris, предлагала виртуализацию уровня ОС (т.н. контейнеры или зоны) начиная с 2004 г. В 2005 году в многопоточных процессорах Niagara 1 была представлена аппаратная виртуализация. При этом гранулярность виртуализации была равна одному потоку (всего чип имел восемь ядер, четыре потока на каждом).

Для взаимодействия ОС и гипервизора был представлен публичный и стабильный интерфейс для привилегированных приложений [3], скрывающий от ОС большинство архитектурных регистров.

Для трансляции адресов используется описанная ранее двухуровневая схема с виртуальными, реальными и физическими адресами. При этом TLB не хранит промежуточный адрес трансляции.

12.4.3. Intel IA-32 и AMD AMD64

В отличие от POWER и SPARC, архитектура IA-32 (и её расширение AMD64) никогда не была подконтрольна одной компании, которая могла бы добавлять функциональность (пара)виртуализации между аппаратурой и ОС, нарушающую обратную совместимость с существующими операционными системами. Кроме того, в ней явно нарушены условия эффективной виртуализации — около 17 служебных инструкций не являются привилегированными, что мешало создать аппаратно поддерживаемые мониторы ВМ. Однако программные мониторы существовали и до 2006 года, когда Intel представила технологию VT-x, а AMD — похожую, но несовместимую с ней AMD-V.

Были представлены новые режимы процессора — VMX root и non root, и уже существовавшие режимы привилегий 0–3 могут быть использованы в обоих из них. Переход между режимами может быть осуществлён с помощью новых инструкций `vmxon` и `vmxoff`.

Для хранения состояния гостевых систем и монитора используется новая структура VMCS (*англ.* virtual machine control structure), копии которой размещены в физической памяти и доступны для монитора ВМ.

Интересным решением является конфигурируемость того, какие события в госте будут вызывать событие ловушки и переход в гипервизор, а какие оставлены на обработку ОС. Например, для каждого гостя можно выбрать, будут ли внешние прерывания обрабатываться им или монитором; запись в какие биты контрольных регистров `CR0` и `CR4` будет перехватываться; какие исключения должны обрабатываться гостём, а какие — монитором и т.п. Данное решение позволяет добиваться компромисса между степенью контроля над каждой ВМ и эффективностью виртуализации. Таким образом, для доверенных гостей контроль монитора может быть ослаблен, тогда как одновременно исполняющиеся с ними сторонние ОС будут всё так же под его строгим наблюдением. Для оптимизации работы TLB используется описанная выше техника тэгирования его записей с помощью ASID (*англ.* address space identifier). Для ускорения процесса трансляции адресов двухуровневая схема трансляции получила имя Intel EPT (*англ.*

extended page walk).

12.4.4. Intel IA-64 (Itanium)

Intel добавила аппаратную виртуализацию в Itanium (технология VT-i [4]) одновременно с IA-32 — в 2006 году. Специальный режим включался с помощью нового бита в статусном регистре `PRS.vm`. С включенным битом ранее служебные, но не привилегированные инструкции начинают вызывать ловушку и выход в монитор. Для возвращения в режим гостевой ОС используется инструкция `vmsw`. Часть инструкций, являющаяся служебными, при включенном режиме виртуализации генерируют новый вид синхронного исключения, для которого выделен собственный обработчик.

Поскольку операционная система обращается к аппаратуре посредством специального интерфейса PAL (*англ.* processor abstraction level), последний был расширен, чтобы поддерживать такие операции, как создание и уничтожение окружений для гостевых систем, сохранение и загрузка их состояния, конфигурирование виртуальных ресурсов и т.д. Можно отметить, что добавление аппаратной виртуализации в IA-64 потребовало меньшего количества усилий по сравнению с IA-32.

12.4.5. ARM

Архитектура ARM изначально была предназначена для встраиваемых и мобильных систем, эффективная виртуализация которых, по сравнению с серверными системами, долгое время не являлась ключевым фактором коммерческого и технологического успеха. Однако в последние годы наметилась тенденция к использованию ВМ на мобильных устройствах для обеспечения защиты критически важных частей системного кода, например, криптографических ключей, используемых при обработке коммерческих транзакций. Кроме того, процессоры ARM стали продвигаться на рынок серверных систем, и это потребовало расширить архитектуру и добавить в неё такие возможности, как поддержка адресации больших объёмов памяти и виртуализация.

Оба аспекта были отражены в избранном компанией ARM подходе к развитию своей архитектуры. На рис. 12.4 представлена схема, подразумевающая вложенность двух уровней виртуализации, представленная в 2010 году в обновлении архитектуры Cortex A15 [1].

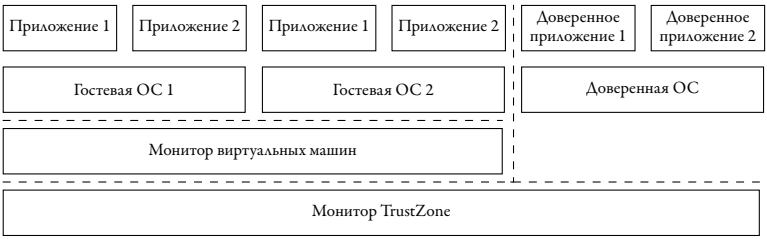


Рис. 12.4. Виртуализация ARM. Монитор TrustZone обеспечивает изоляцию и криптографическую аутентификацию доверенного «мира». В обычном «мире» используется собственный монитор ВМ

Для обеспечения изоляции критических компонент используется первый слой виртуализации, называемый TrustZone. С его помощью все запущенные программные компоненты делятся на два «мира» — доверенный и обычный. В первой среде исполняются те части системы, работа которых не должна быть подвластна внешним влияниям обычного кода. Во второй среде исполняются пользовательские приложения и операционная система, которые теоретически могут быть скомпрометированы. Однако обычный «мир» не имеет доступа к доверенному. Монитор TrustZone обеспечивает доступ в обратном направлении, что позволяет доверенному коду контролировать состояние аппаратуры.

Второй слой виртуализации исполняется под управлением недоверенного монитора и предоставляет возможности мультиплексирования работы нескольких пользовательских ОС. В нём добавлены новые инструкции `HVC` и `ERET` для входа и выхода в/из режим(а) гипервизора. Для событий ловушки использован ранее зарезервированный вектор прерываний 0x14, добавлены новые регистры: указатель стэка `SPSR`, состояние виртуальных ресурсов `HCR` и регистр «синдрома» `HSR`, в котором хранится причина выхода из гостя в монитор, что позволяет последнему быстро проанализировать ситуацию и проэму-

лировать необходимую функциональность без избыточного чтения состояния гостя.

Так же, как это сделано в рассмотренных ранее архитектурах, для ускорения механизмов трансляции адресов используется двухуровневая схема, в которой физические адреса гостевых ОС являются промежуточными. Внешние прерывания могут быть настроены как на доставку монитору, который потом перенаправляет их в гость с помощью механизма виртуальных прерываний, так и на прямую отправку в гостевую систему.

12.4.6. MIPS

Процессоры MIPS развивались в направлении, обратном наблюдаемому для ARM: от высокопроизводительных систем к встраиваемым и мобильным. Тем не менее, аппаратная виртуализация для неё появилась относительно недавно, в 2012 г. Архитектура MIPS R5 принесла режим виртуализации MIPS VZ [2]. Он доступен как для 32-битного, так и для 64-битного варианта архитектуры.

Добавленное архитектурное состояние позволяет хранить контекст ВМ и монитора отдельно. Например, для нужд гипервизора введена копия системного регистра `COP0`, независимая от копии гостя. Это позволяет оптимизировать время переключения между ними, в то время как переключение между несколькими гостевыми ОС требует обновления `COP0` содержимым из памяти и является менее эффективным. Кроме того, часть бит гостевого регистра, описывающие набор возможностей текущего варианта архитектуры и потому ранее используемые только для чтения, из режима монитора доступны для записи, что позволяет ему декларировать возможности, отличные от действительно присутствующих на хозяине.

Привилегии гипервизора, операционной системы и пользователя образуют т.н. луковую (*англ.* onion) модель. В ней обработка прерываний идёт снаружи внутрь, т.е. сначала каждое из них проверяется на соответствие правилам монитора, затем ОС. Синхронные исключения (ловушки), наоборот, обрабатываются сперва ОС, а затем монитором.

Так же, как это сделано в рассмотренных ранее архитектурах,

для ускорения механизмов трансляции адресов используют тэги в TLB и двухуровневую трансляцию в MMU. Для поддержки разработки паравиртуализационных гостей добавлена новая инструкция `hypercall`, вызывающая ловушку и выход в режим монитора.

12.5. Дополнительные темы

В заключение данной главы рассмотрим дополнительные вопросы обеспечения эффективной виртуализации, связанные с переключением между режимами монитора и ВМ.

12.5.1. Уменьшение частоты и выходов в режим монитора с помощью предпросмотра инструкций

Частые прерывания работы виртуальной машины из-за необходимости выхода в монитор негативно влияют на скорость симуляции. Несмотря на то, что производители процессоров работают над уменьшением связанных с этими переходами задержек (для примера см. таблицу 12.1), они всё же достаточно существенны, чтобы пытаться минимизировать их частоту возникновения.

Микроархитектура	Дата запуска	Задержка, тактов
Prescott	3 кв. 2005	3963
Merom	2 кв. 2006	1579
Penryn	1 кв. 2008	1266
Nehalem	3 кв. 2009	1009
Westmere	1 кв. 2010	761
Sandy Bridge	1 кв. 2011	784

Таблица 12.1. Длительность перехода между режимами аппаратной виртуализации для различных поколений микроархитектур процессоров Intel IA-32 (данные взяты из [10])

Как уже было обозначено в главе 3, если одна из техник симуляции оказывается неэффективной, имеет смысл переключиться на некото-

рую другую, например, на интерпретацию или двоичную трансляцию.

На практике исполнения ОС характерна ситуация, что инструкции, вызывающие ловушки потока управления, образуют *кластера*, в которых две или более из них находятся недалеко друг от друга, тогда как расстояние между кластерами значительно. В следующем блоке кода для IA-32 приведён пример такого кластера. Звёздочкой обозначены все инструкции, вызывающие выход в монитор.

```
* in %al,%dx
* out $0x80,%al
  mov %al,%cl
  mov %dl,$0xc0
* out %al,%dx
* out $0x80,%al
* out %al,%dx
* out $0x80,%al
```

Для того, чтобы избежать повторения сценария: выход из ВМ в монитор, интерпретация инструкции, обратный вход в ВМ только для того, чтобы на следующей инструкции вновь выйти в монитор, — используется *предпросмотр* инструкций [10]. После обработки ловушки, прежде чем монитор передаст управление обратно в ВМ, поток инструкций просматривается на несколько инструкций вперёд в поисках привилегированных инструкций. Если они обнаружены, симуляция на некоторое время переключается в режим двоичной трансляции. Тем самым избегается негативное влияние эффекта кластеризации привилегированных инструкций.

12.5.2. Рекурсивная виртуализация

Ситуация, когда монитор виртуальных машин запускается под управлением другого монитора, непосредственно исполняющегося на аппаратуре, называется *рекурсивной виртуализацией*. Теоретически она может быть не ограничена только двумя уровнями — внутри каждого монитора ВМ может исполняться следующий, тем самым образуя иерархию гипервизоров.

Возможность запуска одного гипервизора под управлением монитора ВМ (или, что тоже самое, симулятора) имеет практическую цен-

ность. Любой монитор ВМ — достаточно сложная программа, к которой обычные методы отладки приложений и даже ОС неприменимы, т.к. он загружается очень рано в процессе работы системы, когда отладчик подключить затруднительно. Исполнение под управлением симулятора позволяет инспектировать и контролировать его работу с самой первой инструкции.

Голдберг и Попек в своей упомянутой ранее работе рассмотрели вопросы эффективной поддержки в том числе и рекурсивной виртуализации. Однако их выводы, к сожалению, не учитывают многие из упомянутых выше особенностей современных систем.

Рассмотрим одно из затруднений, связанных со спецификой вложенного запуска мониторов ВМ — обработку ловушек и прерываний. В простейшем случае за обработку всех типов исключительных ситуаций всегда отвечает самый внешний монитор, задача которого — или обработать событие самостоятельно, тем самым «спрятав» его от остальных уровней, или передать его следующему.

Как для прерываний, так и для ловушек это часто оказывается неоптимальным — событие должно пройти несколько уровней иерархии, каждый из которых внесёт задержку на его обработку. На рис. 12.5 показана обработка двух типов сообщений — прерывания, возникшего во внешней аппаратуре, и ловушки потока управления, случившейся внутри приложения.

Для оптимальной обработки различных типов ловушек и прерываний для каждого из них должен быть выбран уровень иерархии мониторов ВМ, и при возникновении события управление должно передаваться напрямую этому уровню, минуя дополнительную обработку вышележащими уровнями и без связанных с этим накладных расходов.

Существуют предложения об интерфейсах между вложенными уровнями виртуализации [8], которые позволили бы эффективно поддерживать вложенность нескольких мониторов ВМ. Однако на практике не было анонсировано реализации подобной или аналогичной технологии в продукции. Современные процессоры аппаратно поддерживают максимум один монитор ВМ.

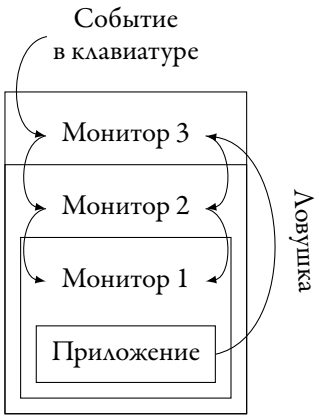


Рис. 12.5. Рекурсивная виртуализация. Все события должны обрабатываться внешним монитором, который спускает их вниз по иерархии, при этом формируется задержка

12.6. Вопросы к главе 12

Вариант 1

1. Может ли привилегированная инструкция когда-либо вызывать событие ловушки?
2. Сколько режимов процессора используется в модели, описанной в работе Голдберга и Попека?
   a) 1,
   b) 2,
   c) 3,
   d) 4.
3. Какие из указанных ниже ситуаций не нарушают принципа эквивалентности виртуального и реального окружений?
   a) Инструкция `FOOBAR` имеет различающуюся семантику.
   b) Инструкция `FOOBAR` выполняется в два раза медленнее.
   c) Инструкция `FOOBAR` не существует в хозяине.
   d) Инструкция `FOOBAR` не может обратиться к физической памяти, потому что внутри ВМ объём ОЗУ меньше.

4. Дайте определение понятия «привилегированная инструкция».
5. Каким термином был обозначен новый режим процессора в системах, поддерживающих аппаратную виртуализацию?

    a) kernel mode,
    b) protected mode,
    c) trusted mode,
    d) root mode.

Вариант 2

1. Выберите правильные варианты продолжения фразы: инструкция может одновременно быть привилегированной и

    a) безвредной,
    b) служебной,
    c) безвредной и служебной.

2. Какие переходы между режимами возможны при возникновении события ловушки?

    a) Из привилегированного в пользовательский.
    b) Из пользовательского в привилегированный.
    c) Из привилегированного в привилегированный.
    d) Из пользовательского в пользовательский.

3. Дайте определение понятия «безвредная инструкция».
4. Какие из нижеперечисленных особенностей реальных ЭВМ опущены в модели Голдберга и Попека?

    a) Существование внешних прерываний.
    b) Присуствие оперативной памяти.
    c) Наличие внешних постоянных хранилищ.
    d) Механизмы виртуальной памяти.

5. Каким образом можно избежать излишне частого сброса содержимого TLB при работе нескольких ВМ?

Литература

1. *Goodacre John* Hardware accelerated Virtualization in the ARM Cortex™ Processors. — ARM Ltd., ноя. 2011. — URL: http://xen.org/files/xensummit_seoul11/nov2/2_XSAsia11_JGoodacre_HW_accelerated_virtualization_in_the_ARM_Cortex_processors.pdf (дата обр. 30.01.2013).
2. Hardware-assisted Virtualization with the MIPS® Virtualization Module. — MIPS Technologies, 2012. — URL: https://www.mips.com/application/login/login.dot?product_name=/auth/MD00994-2B-VZMIPS-WHT-01.00.pdf (дата обр. 29.01.2013).
3. Hypervisor/Sun4v Reference Materials. — Oracle Corporation. — URL: http://kenai.com/projects/hypervisor/pages/ReferenceMaterials (дата обр. 31.01.2013).
4. Intel® Virtualization Technology / F. Leung [и др.] // Intel Technology Journal. — 2006. — Авг. — Т. 10, вып. 03. — ISSN: 1535-864X. — DOI: 10.1535/itj.1003.01. — URL: http://www.intel.com/technology/itj/2006/v10i3/ (дата обр. 20.06.2012).
5. *McGhan Harlan* The gHost in the Machine: Part 1 // Microprocessor Report. — 2007. — Map. — URL: http://mpronline.com (дата обр. 26.01.2013).
6. *McGhan Harlan* The gHost in the Machine: Part 2 // Microprocessor Report. — 2007. — Map. — URL: http://mpronline.com (дата обр. 26.01.2013).
7. *McGhan Harlan* The gHost in the Machine: Part 3 // Microprocessor Report. — 2007. — Map. — URL: http://mpronline.com (дата обр. 26.01.2013).

8. *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.

9. *Popek Gerald J.*, *Goldberg Robert P.* Formal requirements for virtualizable third generation architectures // Communications of the ACM. Т. 17. Вып. 7. — Июл. 1974.

10. 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: https://www.usenix.org/system/files/conference/atc12/atc12-final158.pdf (дата обр. 04.10.2013).

11. *Southern Gabriel* Analysis of SMP VM CPU Scheduling //. — 2008. — URL: http://cs.gmu.edu/~hfoxwell/cs671projects/southern_v12n.pdf (дата обр. 29.02.2013).

12. *Yang Rongzhen* Virtual Translation Lookaside Buffer. — Patent Application US 2008/0282055 A1 (US). — 13 ноя. 2008. — URL: http://www.patentlens.net/patentlens/patent/US_2008_0282055_A1/en/.

13. Заключение

> A computer, it turns out, is just a particular kind of machine that works by pretending to be another machine. This is precisely what today's computers do – they pretend to be calculators, ledgers, typewriters, film splicers, ...[1]
>
> *Ian Bogost*

Один из основателей той области пересечения математики и технических наук, которая в настоящее время именуется «computer science», Алан Тьюринг сделал достаточно большой вклад в развитие компьютерной симуляции. Это и так называемая машина Тьюринга вкупе с тезисом Тьюринга—Чёрча, определяющие теоретическую возможность представления функциональности одной вычислительной машины через имитацию её действий на другой. Это и тест Тьюринга — мысленный эксперимент, отделяющий понятия «человек» и «разумность» и являющийся предтечей к организации взаимодействия реальной и виртуальной систем через «ширму» интерфейсов таким образом, что ни одна из них не может определить, насколько реальна вторая. За более чем полвека эти идеи развились до текущего их состояния, и теперь компьютерную симуляцию мы можем наблюдать вокруг себя ежеминутно: электронные деньги и письма вместо бумажных, текстовый процессор вместо ручки и бумаги, беспроводной телефон вместо его привязанного к розетке предшественника, онлайн-игры вместо догонялок во дворе...

Виртуализация настолько прочно вошла в обиход при использовании ЭВМ, что мы почти никогда не отдаём себе отчёт, что пользуемся ей. Как минимум с одной её формой мы сталкиваемся каждый раз, когда взаимодействуем с компьютером, на котором работает многозадачная ОС, — ведь каждый процесс изолирован в своём «контейнере», обеспеченный виртуальными ресурсами, скрывающими за

[1] `http://www.theatlantic.com/technology/archive/2012/07/the-great-pretender-turing-as-a-philosopher-of-imitation/259824/`

собой реальные физические. Пользовательской программе не приходится учитывать, что одновременно с ней на процессорное время претендуют многие задачи и что значительную часть времени она может находиться в замороженном состоянии.

К сожалению, в этой книге остались незатронутыми некоторые важные темы, связанные с моделированием вычислительных систем. Среди них стоит упомянуть такие вопросы, как эффективное представление графических ускорителей внутри виртуальных окружений, разработка гибридных симуляторов, использование моделей для верификации корректности... При этом существующие главы не настолько полны, иллюстрированы и ясно изложены, как этого хотелось бы авторам, которые, однако, не оставляют надежды приблизиться к желаемому идеалу полноты в последующих редакциях этого учебника. При этом симуляция, моделирование и виртуализация как области технического знания не стоят на месте и активно развиваются, всё больше проникая в повседневную жизнь, создавая новые возможности в работе с уже привычными вещами.

Остаётся надеяться, что в процессе чтения этой книги читатель осознал универсальность и несомненную практическую ценность идеи представления вычислительной машины одного типа на компьютерах иногда совсем иной структуры, открыл для себя новые возможности привычных для него вещей, получил ответы на имевшиеся у него вопросы или же просто хорошо провёл время.

Приложения

A. Ответы на вопросы к главам книги

Правильные ответы в вопросах с вариантами и в открытых вопросах помечены словом **Решение**.

A.1. Ответы к главе 1

Вариант 1

1. Определите понятие «функциональный симулятор». **Решение.** Модель, обеспечивающая корректное выполнение алгоритмов отдельных инструкций, но при этом не гарантирующая корректность симулируемых длительностей операций.
2. Определите понятие «полноплатформенный симулятор». **Решение.** Симулятор, способный запускать операционные системы и потому содержащий модели периферийных устройств.
3. Перечислите все правильные виды сложностей, возникающих при разработке цифровых систем, успешно решаемых с помощью моделирования.

    a) **Решение.** Необходимость знать характеристики новой технологии как можно раньше.
    b) **Решение.** Необходимость выявления ошибок проектирования на ранних стадиях.
    c) Большое энергопотребление реальных образцов.

4. Критерий изоляции исполнения гостевого приложения. **Решение.** Приложение не должно иметь возможности обнаружить следующие факты: 1) исполняется оно внутри виртуальной машины или на реальной аппаратуре; 2) исполняются ли помимо него другие гости. Приложение не должно иметь возможность безконтрольно изменять состояние монитора виртуальных машин.

5. Как расшифровывается обозначение «RTL-модель» в контексте разработки аппаратуры?
    a) Run-time library.
    b) **Решение.** Register transfer level.
    c) Register-transistor logic.
6. Определение гипервизора первого типа. **Решение.** Гипервизоры первого типа (автономные гипервизоры) работают прямо на хозяйской аппаратуре, т.е. не требуют для своей работы операционной системы, беря её функции на себя и являясь привилегированными приложениями.
7. Определение величины MIPS, используемой для измерения скорости программ. **Решение.** Количество миллионов инструкций, исполняющихся за одну секунду.
8. Какой из указанных ниже бенчмарков используется для оценки и сравнения эффективности работы систем виртуализации:
    a) SPECfp,
    b) SPECpower,
    c) SPECint,
    d) **Решение.** SPECvirt,
    e) SPECjbb?

Вариант 2

1. Определение потактового симулятора. **Решение.** Модель, обеспечивающая корректное выполнение алгоритмов отдельных инструкций и при этом высчитывающая задержки, возникающие при их исполнении.
2. Определение симулятора уровня приложений. **Решение.** Модель, обеспечивающая корректную работу гостевых пользовательских приложений, состоящих из непривилегированных инструкций, а также эмулирующая базовый набор системных вызовов некоторой ОС.
3. Перечислите все правильные виды сложностей, возникающих при разработке цифровых систем, успешно решаемых с помощью моделирования.

a) **Решение.** Большое число составляющих систему устройств со сложными взаимосвязями.
b) Сложность получения лицензий на новое оборудование.
c) **Решение.** Обеспечение поддержки аппаратуры программными средствами разработки.

4. Перечислите стадии создания нового устройства с задействованием моделирования в правильном порядке.

   a) Функциональная модель.
   b) Разработка концепции устройства.
   c) RTL-модель.
   d) Потактовая модель.
   e) Выпуск продукции на рынок.
   f) Экспериментальные образцы.

   Решение. Правильная последовательность: 2 – 1 – 4 – 3 – 6 – 5.
5. Определение гибридного симулятора. **Решение.** Модель, частично реализованная в программе для обычного компьютера, а частично — на специализированном оборудовании, например, на ПЛИС.
6. Определение гипервизора второго типа. **Решение.** Гипервизоры второго типа не заменяют операционную систему, но работают поверх неё как обычное пользовательское приложение.
7. Определение понятия FLOPS. **Решение.** Количество арифметических операций над числами с плавающей запятой, выполняемых за одну секунду.
8. Определение понятия *floating point number*. **Решение.** Число с плавающей запятой, записываемое в формате

$$mantissa \cdot 2^{exponent}$$

$$1 \leq mantissa < 2.$$

A.2. Ответы к главе 2

Вариант 1

1. Какие из указанных ниже компонентов обязательны для реализации интерпретатора:
   a) **Решение.** декодер,
   b) дизассемблер,
   c) кодировщик (енкодер),
   d) **Решение.** блоки реализации семантики отдельных инструкций,
   e) кэш декодированных инструкций.
2. Опишите, что происходит на стадии Fetch работы процессора. **Решение.** Четние из памяти машинного кода, соответствующего текущей инструкции.
3. Опишите, что происходит на стадии **Writeback** работы процессора. Для каких инструкций эта стадия будет опущена? **Решение.** Запись результатов исполнения инструкции в память. Если результат должен быть сохранён в регистре, то фаза опускается.
4. Какой вид программ обычно исполняется в привилегированном режиме процессора? **Решение.** Операционные системы, мониторы виртуальных машин первого типа.
5. Какие эффекты могут наблюдаться при невыровненном (unaligned) чтении из памяти в существующих архитектурах:
   a) **Решение.** возникновение исключения,
   b) **Решение.** замедление операции по сравнению с аналогичной выровненной,
   c) данные будут считаны лишь частично,
   d) возможны все перечисленные выше ситуации?
6. Какая из следующих типов ситуаций при исполнении процессора является асинхронной по отношению к работе текущей инструкции?
   a) **Решение.** прерывание (interrupt),
   b) ловушка (trap),

c) исключение (exception),
d) промах (fault)?

7. Выберите правильный вариант окончания фразы: Сцепленный интерпретатор работает быстрее переключаемого (switched), так как
   a) **Решение.** удачно использует предсказатель переходов хозяйского процессора,
   b) кэширует недавно исполненные инструкции,
   c) транслирует код в промежуточное представление,
   d) не требует обработки исключений.

Вариант 2

1. Какой из типов регистров всегда присутствует во всех классических архитектурах:
   a) указатель стека,
   b) аккумулятор,
   c) **Решение.** указатель текущей инструкции,
   d) регистр флагов,
   e) индексный регистр.
2. Опишите, что происходит на стадии **Decode** работы процессора. **Решение.** Анализ машинного слова считанной инструкции для определения кода операции и операндов.
3. Опишите, что происходит на стадии **Advance PC** работы процессора. Для каких инструкций эта стадия будет опущена? **Решение.** Изменение указателя текущей инструкции таким образом, чтобы он указывал на инструкцию, следующую за только что исполненной. Если же произошла передача управления, то счётчик инструкций уже был изменён, и фаза опускается.
4. Какой вид программ обычно исполняется в непривилегированном режиме процессора? **Решение.** Пользовательские приложения.
5. Почему самый простой вид декодера машинных инструкций — однотабличный — не пользуется большой популярностью? **Решение.** Размер таблицы растёт экспоненциально от длины

опкода. Для архитектур, использующих больше 16 бит для кодирования инструкций, такая таблица становится несообразно огромной.

6. Выберите правильные варианты окончания фразы: Наличие единственного `switch` для всех гостевых инструкций в коде интерпретатора

   a) увеличивает его скорость по сравнению со схемой сцепленной интерпретации,
   b) упрощает его алгоритмическую структуру по сравнению со схемой сцепленной интерпретации,
   c) **Решение.** уменьшает его скорость по сравнению со схемой сцепленной интерпретации,
   d) не влияет на скорость работы интерпретатора.

7. Почему редко представляется возможным при симуляции процессора разместить все гостевые регистры на физических регистрах? **Решение.** Число регистров недостаточно. Некоторые регистры могут иметь особенный смысл в хозяйской архитектуре и не допускают произвольных манипуляций.

A.3. Ответы к главе 3

TODO ДОПОЛНИТЬ

Вариант 1

1. Какой вид программ обычно исполняется в непривилегированном режиме процессора? **Решение.** Пользовательские приложения.
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) **Решение.** программа изменяет байт в секции кода?

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

    a) v2p,
    b) **Решение.** v2h,
    c) p2h?

4. Перечислите отличия ДТ от компиляции с ЯВО, мешающие применению классических оптимизаций последнего. **Решение.** В входном языке ДТ отсутствуют имена переменных, границы блоков алгоритмов, нет разделения между кодом и данными.
5. Выберите правильные составляющие задачи «code discovery» (обнаружение кода) в ДТ:

    a) поиск кода внутри исполняемого файла,
    b) **Решение.** поиск границ инструкций при работе двоичного транслятора,
    c) поиск границ инструкций при работе интерпретатора,
    d) **Решение.** 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) **Решение.** привилегированные,
    c) с плавающей запятой,
    d) условные и безусловные переходы?

2. Какой вид программ обычно исполняется в привилегированном режиме процессора? **Решение.** Операционные системы, мониторы виртуальных машин первого типа.
3. Определение понятия *капсула*, используемого в двоичной трансляции. **Решение.** Блок хозяйского машинного кода, моделирующий одну конкретную гостевую инструкцию.

4. Какие порядки размеров капсул в системе двоичной трансляции наиболее вероятны:

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

5. Выберите все необходимые условия корректности применения гиперсимуляции процессора:

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

A.4. Ответы к главе 4

Вариант 1

1. Что из нижеперечисленного может входить в трассу, используемую для симуляции:

    a) **Решение.** доступы во внешнюю память,
    b) **Решение.** внешние прерывания,
    c) состояние регистров,
    d) **Решение.** временные метки,
    e) дизассемблер текущих инструкций?

2. Какие сценарии представляют наибольшую сложность для метода симуляции с помощью трасс:

    a) **Решение.** многопоточная гостевая система,
    b) гостевое приложение с закрытым исходным кодом,
    c) изучение производительности приложений?

3. Как называется методика, призванная уменьшить объём данных трассы, требуемых для анализа работы приложения:

a) манипулирование,
b) фильтрация,
c) интегрирование,
d) **Решение.** сэмплирование?

Вариант 2

1. Какой вид активности невозможен при симуляции трасс:
   a) **Решение.** интерактивное взаимодействие с пользователем,
   b) загрузка операционной системы,
   c) работа с периферийными устройствами?
2. Какие типы событий должны быть отражены в трассе работы приложения для того, чтобы она была полезна:
   a) **Решение.** только внешние события: доступы в память, к устройствам,
   b) только внутренние события: изменения регистров,
   c) и внутренние, и внешние события?
3. Выберите правильный порядок операций при обработке трассы:
   a) перематывание – измерение – разогрев,
   b) разогрев – перематывание – измерение,
   c) **Решение.** перематывание – разогрев – измерение.

A.5. Ответы к главе 5

Вариант 1

1. Определение понятия «квота», используемого в симуляции многопроцессорных систем. **Решение.** Максимальное количество шагов, симулируемых одним процессором без переключения на симуляцию других.
2. Определение понятия «неисполняющее устройство». **Решение.** Модель, изменения в состоянии которой происходят через интервалы, равные нескольким шагам симулируемого времени.

3. Выберите правильный вариант продолжения фразы: Симулируемое время в моделях DES
    a) изменяется непрерывно,
    b) изменяется скачками фиксированной длительности,
    c) **Решение.** изменяется скачками, длительность которых различна.
4. Выберите правильный вариант окончания фразы: в моделях DES события могут обрабатываться, если они находятся
    a) только в голове очереди событий (самые поздние),
    b) **Решение.** только в хвосте очереди событий (самые ранние),
    c) в любой позиции в очереди событий.
5. Выберите правильный вариант окончания фразы: в моделях DES новые события могут быть добавлены
    a) только к голове очереди событий,
    b) только к хвосту очереди событий,
    c) **Решение.** в любую позицию в очереди событий.
6. Выберите сценарии, когда скорость симуляции, превышающая скорость работы реальной системы, нежелательна:
    a) программа вычисляет значение некоторой функции в узлах сетки и выводит результаты на экран,
    b) **Решение.** система ожидает ввода пользователя в течение ограниченного времени,
    c) программа взаимодействует по моделируемой сети с другой моделируемой системой.

Вариант 2

1. Как могут проявиться недостатки слишком большой квоты? **Решение.** Симуляция будет некорректной — будут происходить тайм-ауты взаимодействия моделируемых процессоров.
2. Определение понятия «исполняющее устройство». **Решение.** Модель, изменения состояния которой происходят на каждом каждом шаге симулируемого времени.

3. Выберите правильные возможности из перечисленных.

    a) Скорость течения симулируемого времени может быть меньше скорости течения реального времени.
    b) Скорость течения симулируемого времени может быть больше скорости течения реального времени.
    c) Скорость течения симулируемого времени приблизительно равна скорости течения реального времени.
    d) **Решение.** Все вышеперечисленные варианты верны.

4. Выберите правильный вариант окончания фразы: в моделях DES одно значение метки времени

    a) может соответствовать максимум одному событию,
    b) может соответствовать нескольким событиям, порядок их обработки при этом неопределён,
    c) **Решение.** может соответствовать нескольким событиям, порядок их обработки при этом определён,
    d) всегда соответствует нескольким событиям, некоторые из них могут быть псевдособытиями.

5. Выберите правильный вариант окончания фразы: в моделях DES события из очереди могут быть удалены

    a) только из головы очереди событий,
    b) только из хвоста очереди событий,
    c) **Решение.** из любой позиции в очереди событий.

6. Выберите правильное выражение для отношения скоростей моделирования систем с N гостевыми процессорами и с одним хозяйским процессором при однопоточной симуляции:

    a) **Решение.** $\frac{S(N)}{S(1)} = O(1/N)$,
    b) $\frac{S(N)}{S(1)} = O(N)$,
    c) $\frac{S(N)}{S(1)} = O(1/N^2)$,
    d) $\frac{S(N)}{S(1)} = O(N^2)$,
    e) $\frac{S(N)}{S(1)} = O(\ln N)$.

A.6. Ответы к главе 6

TODO Расширить и переработать

Вариант 1

1. Какие из типов схем PDES позволяют добиться детерминизма симуляции?

   a) **Решение.** Барьерная (с доменами синхронизации).
   b) Консервативная.
   c) Оптимистичная.
   d) Наивная.

2. Чем чревата излишне частая отправка пустых (null) сообщений в консервативной схеме PDES с детектированием взаимоблокировок? **Решение.** Низкой скоростью работы симулятора из-за перегруженности хозяйской сети.
3. Выберите правильные продолжения фразы: Частая отправка пустых (null) сообщений нежелательна, так как

   a) **Решение.** это может ограничивать скорость симуляции,
   b) это может вызвать нарушение каузальности симуляции,
   c) это может привести к взаимоблокировке потоков,
   d) это может привести к переполнению очередей сообщений.

4. Выберите правильные ответы.

   a) Симуляция, реализованная с помощью схемы PDES, всегда детерминистична.
   b) Симуляция, реализованная с помощью схемы PDES, недетерминистична из-за возможности потери сообщений между потоками.
   c) Симуляция, реализованная с помощью схемы PDES, недетерминистична из-за возможности блокировки отдельных потоков.
   d) Симуляция, реализованная с помощью схемы PDES, недетерминистична из-за варьирующейся скорости работы отдельных потоков.

Вариант 2

1. Почему не будет работать **наивная** схема параллельного DES? Выберите верные ответы.

    a) **Решение.** Недетерминизм модели.
    b) Невозможно организовать передачу сообщений между потоками.
    c) **Решение.** Возможно нарушение каузальности.
    d) Невозможно подобрать точно квоту выполнения.

2. Чем чревата недостаточно частая отправка пустых (null) сообщений в консервативной схеме PDES с детектированием взаимоблокировок? **Решение.** Низкой скоростью работы симулятора из-за частой блокировки потоков, слишком далеко убежавших в симулируемое будущее.
3. Выберите правильные ответы.

    a) **Решение.** Консервативные схемы PDES не допускают нарушения каузальности.
    b) Консервативные схемы PDES допускают нарушения каузальности.
    c) Консервативные схемы PDES допускают нарушения каузальности, но впоследствии их корректируют.
    d) Оптимистичные схемы PDES не допускают нарушения каузальности.
    e) Оптимистичные схемы PDES допускают нарушения каузальности.
    f) **Решение.** Оптимистичные схемы PDES допускают нарушения каузальности, но впоследствии их корректируют.

4. Выберите правильные свойства домена синхронизации в модели PDES.

    a) Количество моделируемых устройств внутри одного домена фиксировано.
    b) Не происходит взаимодействия устройств, находящихся в различных доменах.
    c) Количество моделируемых устройств внутри одного доме-

на ограничено.

d) **Решение.** Характерная частота коммуникаций между доменами превышает частоту коммуникаций внутри каждого.

e) Характерная частота коммуникаций между доменами равна частоте коммуникаций внутри отдельного домена.

A.7. Ответы к главе 7

Вариант 1

1. Выберите правильные варианты ответов:

   a) **Решение.** функциональные модули не имеют внутреннего состояния,

   b) функциональные модули могут иметь внутреннее состояние,

   c) функциональные модули всегда имеют внутреннее состояние.

2. Выберите правильные варианты ответов:

   a) **Решение.** ширина входа и выхода порта должны быть равны,

   b) ширина входа и выхода порта могут различаться,

   c) количество выходов функционального элемента должно быть равно единице,

   d) количество входов и выходов функционального элемента должно совпадать.

3. Выберите правильные варианты продолжения фразы: процесс исполнения потактовой модели на основе портов

   a) **Решение.** всегда содержит две фазы, которые обязаны чередоваться,

   b) всегда содержит две фазы, порядок которых не фиксированный,

   c) всегда содержит одну фазу, в течение которой работают все субъединицы,

d) может содержать более двух чередующихся фаз.

Вариант 2

1. Выберите правильные варианты ответов:
   a) при передаче данных порты не сохраняют бит валидности данных,
   b) при передаче данных порты не сохраняют бит валидности данных, только если он снят,
   c) при передаче данных порты не сохраняют бит валидности данных, только если он поднят,
   d) **Решение.** при передаче данных порты сохраняют бит валидности данных.
2. Выберите правильные варианты продолжения фразы: внутри исполнения фазы функциональных элементов потактовой модели на основе портов
   a) первыми должны исполняться функции, расположенные в графе правее,
   b) **Решение.** порядок исполнения функций устройств неважен,
   c) первыми должны исполняться функции, расположенные в графе левее.
3. Выберите правильные варианты продолжения фразы: в модели, описанной на основе портов,
   a) функциональные модули имеют различные задержки выполнения,
   b) **Решение.** функциональные модули не имеют определённой задержки выполнения,
   c) функция портов является функцией тождественности, а задержка нулевая,
   d) **Решение.** функция портов является функцией тождественности, а задержка ненулевая,
   e) функция портов не является функцией тождественности, а задержка нулевая.

A.8. Ответы к главе 8

Вариант 1

1. Какой байт будет расположен первым в памяти на Little Endian системе при записи числа `0xaabbccdd` в память? **Решение.** 0xdd
2. Выберите правильное отношение для фразы «машинное слово длиной w байт выровнено в памяти по адресу $addr$»:

   a) $addr \neq 0 \pmod w$ (адрес не делится нацело на длину слова),
   b) $w = 2^k$ и $addr = 2^m$, $k, m \in \mathbb{N}$ (адрес и длина являются степенями двойки),
   c) $w = 2^k$ и $addr = 2^m$, $k \leq m$,$k, m \in \mathbb{N}$ (адрес и длина являются степенями двойки, степень длины меньше степени адреса),
   d) **Решение.** $addr = 0 \pmod w$ (адрес делится нацело на длину слова).

3. Почему симулятор не имеет права кэшировать регионы гостевой памяти, помеченные как отображённые на устройства? **Решение.** В отличие от простой памяти, хранящей данные без изменений, устройство может менять своё состояние и соответственно содержимое регионов памяти на каждом доступе к нему.
4. Определение понятия «машинное слово». **Решение.** Максимальный объём данных, который процессор данной архитектуры способен обработать за одну инструкцию.
5. Какой интегральный тип языка Си наиболее удачно использовать для хранения состояния моделируемого регистра шириной 32 бита?

   a) `int`,
   b) `unsigned int`,
   c) **Решение.** `uint32_t`,
   d) зависит от хозяйской системы.

Вариант 2

1. Какой байт будет расположен первым в памяти на Big Endian системе при записи числа `0xbaadc0de` в память? **Решение.** 0xba
2. Какую стратегию подразумевает концепция ленивого вычисления?

    a) Замена точного значения выражения приближённым, но получаемым за меньшее время.
    b) **Решение.** Запуск вычисления выражения происходит лишь при необходимости использовать его результат.
    c) Выражение вычисляется сразу после доступности значений всех его входных слагаемых.
    d) Значение подвыражения, используемого в нескольких других выражениях, сохраняется при первом вычислении и затем переиспользуется.

3. Определение понятия «байт». **Решение.** Минимальная адресуемая в данной архитектуре единица информации.
4. Сколько бит информации получает процессор при первоначальном возникновении сигнала на линии прерывания?

    a) **Решение.** 1 бит.
    b) 8 бит.
    c) Зависит от архитектуры.

5. Выберите правильные окончания фразы: карта памяти

    a) **Решение.** использует цель по умолчанию, если обрабатываемый запрос не попадает ни в одно из устройств,
    b) может указывать на устройство не более одного раза,
    c) должна указывать на все присутствующие в гостевой системе устройства,
    d) **Решение.** может указывать не только на устройства, но и на другие карты памяти.

A.9. Ответы к главе 9

Вариант 1

1. Выберите правильные варианты продолжения фразы: использование кэшей при работе приложения целесообразно, если
    a) **Решение.** программа показывает временную локальность доступов,
    b) программа не обращается в оперативную память,
    c) программа работает с очень большим объёмом данных,
    d) **Решение.** программа показывает пространственную локальность доступов,
    e) программа работает с объёмом данных, меньшим ёмкости кэша.
2. Выберите все правильные окончания фразы: функциональные симуляторы часто не содержат в себе модель кэша, потому что
    a) **Решение.** они влияют только на задержки, но не на семантику инструкций,
    b) всегда имеется возможность переиспользовать хозяйский кэш для нужд симуляции,
    c) **Решение.** такие модели сильно замедляют симуляцию.
3. Данные могут попадать в кэш при следующих операциях:
    a) **Решение.** чтение памяти (load),
    b) **Решение.** запись в память (store),
    c) арифметические операции,
    d) операции с числами с плавающей запятой,
    e) **Решение.** предвыборка данных (prefetch),
    f) **Решение.** загрузка инструкции (fetch),
    g) инвалидация линии (invalidate).

Вариант 2

1. Выберите правильные варианты окончания: линия данных с фикированным адресом
    a) всегда попадает в одну и ту же ячейку кэша,

b) **Решение.** всегда попадает в один и тот же сет,
c) может быть сохранён в любой ячейке кэша.

2. Выберите правильные варианты.
   a) Темпы роста скорости оперативной памяти и процессоров одинаковы с 80-х годов XX века.
   b) Темп роста скорости оперативной памяти опережает темпы роста скорости работы процессоров.
   c) **Решение.** Темп роста скорости процессоров опережает темпы роста скорости оперативной памяти.
3. Кэши необходимо симулировать даже в функциональной модели, если они используются для
   a) **Решение.** создания транзакционной памяти,
   b) моделирования работы ЭВМ гарвардской архитектуры,
   c) поддержания когерентности в SMP системах.

A.10. Ответы к главе 10

Вариант 1

1. Какое утверждение наилучшим образом характеризует термин SystemC?
   a) Компилятор языка Си с дополнениями для моделирования систем.
   b) Язык программирования, похожий на Си.
   c) Язык программирования, похожий на C++.
   d) **Решение.** Набор библиотек для C++.
2. Язык DML используется для разработки
   a) **Решение.** функциональных моделей,
   b) потактовых моделей,
   c) гибридных моделей.
3. Текущая реавлизация комилятора DMLC является
   a) компилятором типа source-to-source с промежуточным языком C++,

b) компилятором, преобразующим исходный текст в байткод Java,
c) **Решение.** компилятором типа source-to-source с промежуточным языком Си,
d) классическим компилятором,
e) частичным интерпретатором.

4. Закончите фразу: Языки разработки аппаратуры

a) не используются для начального моделирования устройств, так как могут быть преобразованы только в netlist,
b) **Решение.** не используются для начального моделирования устройств, так как получаемые модели очень медленны,
c) не используются для начального моделирования устройств, так как могут содержать в себе синтезируемую часть,
d) используются для начального моделирования устройств.

Вариант 2

1. Какое утверждение наилучшим образом характеризует термин TLM?

a) Язык программирования, похожий на Си.
b) Язык программирования, похожий на C++.
c) Среда исполнения моделей DES.
d) **Решение.** Расширение стандарта SystemC.

2. Язык DML используется для разработки

a) **Решение.** неисполняющих моделей,
b) исполняющих моделей,
c) как исполняющих, так и неисполняющих моделей.

3. Какой способ наиболее удобен и надёжен для поддержания набора инструментов моделирования в синхронизированном состоянии при постоянном изменении входной спецификации процессора?

a) **Решение.** Генерация всех инструментов из единого описания.
b) Тщательное сравнение всех инструментов после каждого изменения одного из них.
c) Создание одного инструмента, поддерживающего максимальное количество функций.

4. Закончите фразу: Синтезируемое подмножество языков разработки аппаратуры

a) не может быть использовано для создания netlist и RTL-описаний,
b) используется только для отладки моделей,
c) **Решение.** используется для создания netlist и RTL-описаний.

A.11. Ответы к главе 11

Вариант 1

1. Выберите свойства, которые должны выполняться для идеальной «волшебной» инструкции:

a) **Решение.** должна быть допустимой во всех режимах работы процессора,
b) должна быть привилегированной,
c) не должна иметь явных аргументов,
d) **Решение.** не должна генерироваться обычными компиляторами,
e) **Решение.** не должна вызывать эффектов (т.е. быть NOP),
f) не должна иметь неявных аргументов.

2. Какая инструкция для архитектуры IA-32 не может быть использована как волшебная?

a) `CPUID` — идентификация процессора,
b) **Решение.** `INT` — программное прерывание,
c) `NOP` — пустая операция.

3. Для какой из перечисленных ниже операционных систем паравиртуализационные расширения сложно писать из-за закрытости исходного кода?
    a) **Решение.** Microsoft Windows,
    b) GNU/Linux,
    c) FreeBSD.
4. В чём состоят недостатки сырого формата дисков?
    a) невозможность случайного доступа к секторам диска,
    b) нерациональное расходование дискового пространства гостя,
    c) **Решение.** нерациональное расходование дискового пространства хозяина,
    d) отсутствие публичной документации на формат.

Вариант 2

1. Почему передача большого объёма данных между гостём и хозяином с помощью волшебной инструкции неэффективна?
    a) **Решение.** за один раз можно передать только несколько байт,
    b) побочные эффекты множества волшебных инструкций подряд могут нарушить работу гостя,
    c) побочные эффекты множества волшебных инструкций подряд могут нарушить работу хозяина,
    d) направление передачи данных ограничено только из гостя в хозяина.
2. Назовите приём виртуализации, в котором гостевое приложение модифицируются таким образом, чтобы задействовать некоторую функциональность аппаратуры, присутствующую только внутри модели, но не на реальных системах?
    a) гиперсимуляция,
    b) метавиртуализация,
    c) **Решение.** паравиртуализация,
    d) изоляция.

3. Дайте определение термину «проброс устройства»:

    a) передача устройства в эксклюзивное пользование нескольким гостям,
    b) передача устройства в эксклюзивное пользование хозяину,
    c) **Решение.** передача устройства в эксклюзивное пользование единственному гостю.

4. Для чего используются разностные файлы?

    a) **Решение.** хранение изменений гостевого диска за время работы симуляции,
    b) сжатие оригинального образа гостевого диска для того, чтобы он занимал меньше места,
    c) прозрачное шифрование оригинального образа гостевого диска,
    d) расширения размера гостевого диска в случае, когда старый полностью заполнен.

A.12. Ответы к главе 12

Вариант 1

1. Может ли привилегированная инструкция когда-либо вызывать событие ловушки? **Решение.** Да, если это ловушка защиты памяти, т.е. инструкция обратилась к памяти, не входящей в текущий разрешённый сегмент.
2. Сколько режимов процессора используется в модели, описанной в работе Голдберга и Попека?

    a) 1,
    b) **Решение.** 2,
    c) 3,
    d) 4.

3. Какие из указанных ниже ситуаций не нарушают принципа эквивалентности виртуального и реального окружений?

    a) Инструкция `FOOBAR` имеет различающуюся семантику.

b) **Решение.** Инструкция `FOOBAR` выполняется в два раза медленнее.
c) Инструкция `FOOBAR` не существует в хозяине.
d) **Решение.** Инструкция `FOOBAR` не может обратиться к физической памяти, потому что внутри ВМ объём ОЗУ меньше.

4. Дайте определение понятия «привилегированная инструкция». **Решение.** Инструкции, исполнение которых в режиме пользователя всегда вызывает ловушку потока управления.
5. Каким термином был обозначен новый режим процессора в системах, поддерживающих аппаратную виртуализацию?
   a) kernel mode,
   b) protected mode,
   c) trusted mode,
   d) **Решение.** root mode.

Вариант 2

1. Выберите правильные варианты продолжения фразы: инструкция может одновременно быть привилегированной и
   a) безвредной,
   b) **Решение.** служебной,
   c) безвредной и служебной.
2. Какие переходы между режимами возможны при возникновении события ловушки?
   a) Из привилегированного в пользовательский.
   b) **Решение.** Из пользовательского в привилегированный.
   c) **Решение.** Из привилегированного в привилегированный.
   d) Из пользовательского в пользовательский.
3. Дайте определение понятия «безвредная инструкция». **Решение.** Инструкция, не являющаяся служебной.
4. Какие из нижеперечисленных особенностей реальных ЭВМ опущены в модели Голдберга и Попека?
   a) **Решение.** Существование внешних прерываний.

b) Присуствие оперативной памяти.
c) **Решение.** Наличие внешних постоянных хранилищ.
d) Механизмы виртуальной памяти.

5. Каким образом можно избежать излишне частого сброса содержимого TLB при работе нескольких ВМ? **Решение.** Использовать TLB с поддержкой тэгов для того, чтобы помечать принадлежащие независимым ВМ записи.

B. Альтернативные подходы к изучению цифровых систем

> Contrary to common belief,
> performance evaluation is an art.
>
> *Raj Jain*

Было бы неверным полагать, что симуляция является единственным, оптимальным и незаменимым инструментом анализа поведения вычислительных систем. Существуют другие методики анализа поведения иерархических систем, к которым относятся в том числе вычислительные машины.

В данной главе мы рассмотрим два подхода, призванных отвечать на вопрос: какие значения выходных характеристик функционирования системы мы получим, если в качестве входных данных подадим некоторые другие, экспериментально полученные числа?

B.1. Сети массового обслуживания

Эта методика (*англ.* queuing networks) [4] названа по своему основному инструменту — направленному графу (сети), в узлах которого находятся автономные части изучаемой аппаратной системы. Анализ заключается в выполнении следующих шагов.

1. Обрисовка схемы функционирования системы с помощью диаграммы обслуживания, отражающей наличие в ней клиентов, (опционально) прибывающих снаружи, перемещающихся по центрам обслуживания и затем (опционально) убывающих из неё или возвращающихся к началу (рис. B.1). Пример явления, описываемого такой схемой, — очередь на регистрацию на авиарейс; при этом люди в очереди — это клиенты, а стойка регистрации — центр обслуживания. Если при этом багаж пассажирам приходится сдавать в отдельном месте, то багажная

стойка также является центром обслуживания (заметьте, что у неё тоже может образоваться очередь).

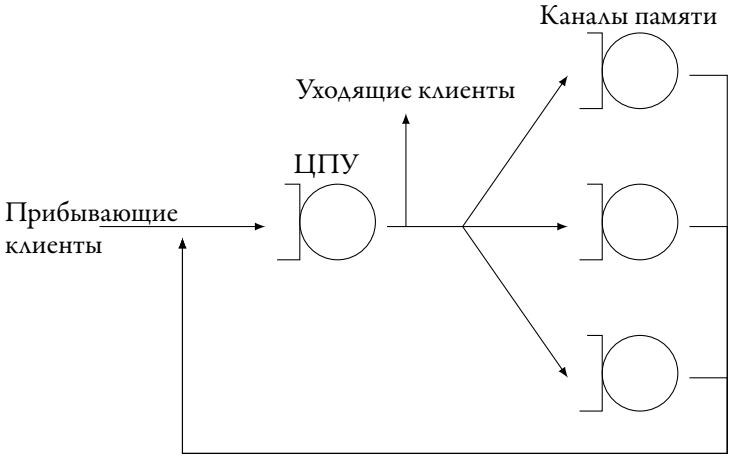


Рис. В.1. Пример представления системы в виде сети обрабатывающих центров. Стрелками показаны направления прибытия, отправления и перемещения клиентов; кружками — обслуживающие центры, прямыми скобками — очереди, в которых могут находиться клиенты, ожидающие сервиса

2. Нахождение входных величин, описывающих известные параметры функционирования системы или её частей. Они берутся из внешних источников, например, из эксперимента или документации.
3. Использование законов поведения системы обслуживания для нахождения неизвестных характеристик поведения.

Понятия, которыми оперирует методика анализа сетей обслуживания, в основном характеризуют или скорость совершения каких-либо действий в центре обслуживания, или время неких процессов, происходящих с одним потребителем. При этом она описывает установившиеся состояния систем, при которых величины не изменяются во времени, и призвана описывать средние величины для статистических популяций клиентов.

Основные понятия, вводимые для описания состояния системы и её частей, и их обозначения приведены далее.

- Скорость прибытия (*англ.* arrival rate) λ — как часто возникают новые потребители.
- Пропускная способность (*англ.* throughput) X — как быстро некоторый центр способен их обслуживать.
- Степень утилизации (*англ.* utilization) U — доля времени, в течение которой центр занят обслуживанием.
- Среднее время обслуживания запроса (*англ.* service requirement per request) S — сколько один потребитель находится в некотором центре.

Следующим компонентом подхода являются законы, связывающие величины. Зачастую они легко выводятся из общих соображений, которые имеют все люди, когда либо пытавшиеся получить какую-либо услугу в любой бюрократической организации этого мира и пытавшиеся облегчить свою участь.

- Закон использования (The utilization law): $U = X \cdot S$.
- Закон Литтла (*англ.* Little's law): среднее число N клиентов за достаточно долгосрочный период в устойчиво функционирующей системе равно средней норме или скорости прибытия, умноженной на определённое за тот же период среднее время T, которое один клиент проводит в системе: $N = \lambda T$.
  Одно из следствий этого закона позволяет понять, как можно «выменивать» пропускную способность на задержку при передаче сообщений: в одной и той же системе мы можем или послать одно большое сообщение, состоящее из нескольких маленьких, за один раз и достаточно быстро, однако после этого вынуждены ожидать, пока это комбинированное сообщение придёт к отправителю, или же отправлять их сразу по готовности, при этом каждое будет вызывать достаточно длинную задержку.
- Соотношение для времени отклика системы: $R = N/X - Z$. Данное соотношение связывает ощущаемое клиентом время

ожидания R, количество клиентов N, пропускную способность узла X и время «размышления» клиента Z, в течение которого он сам по себе не требует немедленного сервиса.
- Закон вынужденного потока (*англ.* the forced flow law) выражает пропорциональность загруженности подсистем полной пропускной способности всей системы при условии ненасыщенного состояния отдельных центров: $X_k = V_k X$.
- Баланс потока: $\lambda = X$ — по сути переформулировка понятия устоявшегося режима, при котором число клиентов в системе постоянно и лишь слабо флуктуирует в моменты переходов клиентов между центрами.

Несмотря на простоту используемых формул, при умелом построении схемы и правильном использовании методика позволяет достаточно точно, а главное быстро, без использования большого количества вычислений, предсказывать характер поведения комплекса при изменении его характеристик и, в частности, давать ответы на следующие вопросы:

1. Как изменится производительность при увеличении или изменении характера нагрузки?
2. Как отразится модификация подсистемы (например, апгрейд) на работу целого [7]?
3. Будет ли получен эффект от удвоения (утроения, уменьшения...) числа обслуживающих центров при неизменности прочих компонент?

Одна из мощных методик, построенных на теории центров обслуживания, имеет название анализ *средних значений* (*англ.* mean value analysis, MVA). Характерной её особенностью является решение системы уравнений, связывающих искомые и известные величины численными итеративными методами.

B.2. Симуляция методами Монте-Карло

Все описанные ранее алгоритмы симуляции (интерпретация, двоичная трансляция и т.п.) обладали одним общим свойством — они

были детерминированными, как и описываемые ими цифровые системы. При этом строгая повторяемость результатов, как правило, является предметом гордости её авторов. Несмотря на это, в науке существует широкий класс методик, основанных на принципиальной случайности ряда входных или промежуточных величин. Класс таких методов моделирования получил название *Монте-Карло* в честь города с большим количеством казино, где случай решает всё.

Анализ с помощью очередей и центров обслуживания имеет свои основания в теории стохастических процессов, применённой к процессу вычислений [1]. При этом используются *частные* выводы из общих результатов, верные для стационарных процессов и установившихся состояний. Исследование, явным образом учитывающее случайность «мгновенных» значений входных данных, даёт возможность изучить поведение более детально, в частности, «увидеть» переходные процессы, измерить реакцию системы на задачах, различающихся не только своими макрохарактеристиками (средняя интенсивность), но и статистическими параметрами, учесть нелинейный характер отдельных элементов системы и т.п.

Общая схема моделирования с использованием методов Монте-Карло выглядит следующим образом:

1. Построить функцию, описывающую поведение интересующей нас системы и зависящую от ряда входных параметров. Функция также может зависеть от результатов предыдущих её запусков (например, быть марковским процессом).
2. Провести акт симуляции системы, заключающийся в генерации набора случайных входных данных и вычисления на них функции системы.
3. Повторить симуляции большое количество раз для достижения статистически значимых результатов.
4. Усреднить отдельные результаты актов симуляции для получения конечных чисел, характеризующих систему.

Использование случайных последовательностей отражает тот факт, что не всегда мы имеем информацию о точной последовательности внешних воздействий на систему, а лишь некие усреднённые ве-

личины; кроме того, иногда точность принципиально недостижима или даже вредна: например, мы хотим знать поведение на обширном классе задач, при этом предварительно составлям его статистический «портрет» и затем исполняем свою модель с его учётом.

Методы Монте-Карло относительно слабо используются в исследованиях ЭВМ в настоящее время, т.к. имитационная симуляция в большинстве случаев может быть выполнена быстро и дать более точные данные о работе приложения, не обезличенные усреднением. Однако она перестаёт быть лучшим решением в случаях, когда исследователя интересуют системы гигантских размеров (миллионы и миллиарды агентов: процессоров, ядер, узлов и т.п.). При этом нет возможности проследить за всеми переходами в её глобальном состоянии и одновременно нет удобных способов применить декомпозицию для изучения подсистем. Построение детальной имитационной модели не представляется возможным, поэтому ограничиваются более-менее высокоуровневыми описаниями. Один из примеров практических реализаций — пакет для моделирования сетей NS-3 [3].

Существенное достоинство методик, использующих случайные числа, — эффективность параллельной симуляции отдельных экспериментов, т.к. они совершенно независимы друг от друга: их входные данные не связаны между собой, а состояние системы может зависеть только от предыдущего её состояния [6]. Это обстоятельство открывает путь к практически линейному ускорению симуляции при увеличении числа используемых вычислительных узлов.

Однако при этом строгие требования выдвигаются на используемый генератор случайных чисел. Он должен обладать следующими свойствами:

1. Случайность и взаимная независимость генерируемых величин на всех потоках симуляции. Нарушение этого условия может привести к серьёзным искажениям результатов.
2. Высокая скорость создания случайной последовательности и доставки её членов до потребителей. Невыполнение этого требования делает генератор случайных чисел узким местом, ограничивающим выигрыш от параллелизма.

Таким условиям могут удовлетворять только аппаратно реализу-

емые генераторы, псевдослучайные последовательности не годятся. Подобные генераторы могут быть выполнены в качестве плат расширения (PCI, PCI-Express и т.п.) или входить в состав центрального процессора. Например, в процессоры микроархитектуры Ivy Bridge компании Intel была включена инструкция `RDRAND` [2, 5], возвращающая 64-битные случайные числа, пригодные в том числе для задач криптографии.

Литература

1. *Allen Arnold O.* Probability, Statistics, and Queueing Theory with Computer Science Applications (Second Edition). — Academic Press Inc., 1990. — С. 768. — ISBN: 0-12-051051-0.
2. Bull Mountain Software Implementation Guide. — Intel Corporation. Июн. 2011. — URL: http://software.intel.com/file/37157.
3. Network simulator NS-3. — URL: http://www.nsnam.org.
4. Quantitative system performance: computer system analysis using queueing network models / Edward D. Lazowska, John Zahorjan, G. Scott Graham, Kenneth C. Sevcik. — Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1984. — ISBN: 0-13-746975-6.
5. *Taylor Greg, Cox George* Behind Intel's New Random-Number Generator. — 2011. — URL: http://spectrum.ieee.org/computing/hardware/behind-intels-new-randomnumber-generator/0 (дата обр. 08.07.2012).
6. *Глинский Б.М., Родионов А.С., Марченко М.А.* Об агентно-ориентированном подходе к имитационному моделированию суперэвм экзафлопсной производительности // Труды международной суперкомпьютерной конференции и конференции молодых учёных «Научный сервис в сети Интернет. Экзафлопсное будущее». — 2011. — 159–165. — ISBN: 978-5-211-06229-0.
7. *Тульчинский В. Г.* Когда суперкомпьютер работает хуже персоналки // Суперкомпьютеры. — 2012. — 1(9). — 47–53. — URL: http://www.supercomputers.ru.

C. История изменений документа

Наше время нельзя просто издать выпустить книгу и надеяться, что её содержимое будет актуально хотя бы десять лет. Особенно это касается столь динамичных тем, как симуляция и виртуализация. Поэтому мы стараемся выкладывать обновления для этой книги достаточно часто. Побочным эффектом этого, конечно, могут являться опечатки и просто не до конца дописанные секции. Для каждого бумажного переиздания проводится тщательная работа по выявлению и исправлению всех огрехов. Веб-версия позволяет знакомить читателя с наиболее актуальными тенденциями и технологиями.

Следующая таблица даёт представление о том, какие изменения были внесены на различных этапах работы над этой книгой.

Дата	**Версия**	**Описание**
07 декабря 2011	0.1	Первый черновик
21 декабря 2011	0.2	Сборка глав в единый документ
10 января 2012	0.3	Добавлена глава 3
12 января 2012	0.4	Предисловие расширено
20 января 2012	0.5	Проверена орфография
08 февраля 2012	0.6	Добавлена глава «Взаимодействие симуляции с внешним миром». Расширена библиография главы «Альтернативы симуляции»
25 февраля 2012	0.7.2	Добавлена глава «Сверхоперативная память — кэши»
06 марта 2012	0.8	Расширено «Предисловие». Исправлены ошибки
19 марта 2012	0.9	Добавлена глава «Потактовая симуляция». Библиография приведена к стандарту ГОСТ 7.0.5-2008
03 апреля 2012	0.10	Обновлена глава «Модели процессора на основе интерпретации»
30 апреля 2012	0.11	Исправлены обнаруженные ошибки
08 июля 2012	0.12.1	Расширена библиография некоторых глав

31 июля 2012	0.13.1	Расширена глава «Модели процессора на основе интерпретации»
09 сентября 2012	0.14	Трассировка выделена в отдельную главу «Моделирование с использованием трасс»
10 сентября 2012	0.9.1	Подготовка к изданию
11 сентября 2012	0.9.2	Изменено форматирование. Библиография объединена
13 сентября 2012	1.0.0	Рукопись отдана на рецензирование
22 сентября 2012	1.0.2	Исправлены недочёты
08 октября 2012	1.2	Пройден второй круг рецензирования
18 октября 2012	1.3	Исправлены недочёты оформления
18 октября 2012	1.3.1	Расширены главы 10, 11
15 ноября 2012	1.4	Исправлена глава «Взаимодействие симуляции с внешним миром»
28 ноября 2012	1.5	Исправлена глава «Архитектурное состояние»
17 декабря 2012	1.6	Добавлены контрольные вопросы к главам
29 декабря 2012	1.7	Добавлена глава «Заключение». Различия между виртуализацией и симуляцией иллюстрированы в главе 1
05 января 2013	1.8	Главы 1 и 2 объединены
11 февраля 2013	1.9	Добавлена глава «Современная виртуализация»
14 мая 2013	1.10	Исправлены опечатки
24 мая 2013	1.90	Подготовка к изданию
6 июля 2013	2.1	Начата переработка иллюстраций
20 июля 2013	2.2	Учтены редакторские правки
4 августа 2013	2.3	Иллюстрации приведены к общему стилю
4 октября	2.4	Переработка структуры книги, содержания отдельных глав
7 октября 2013 г.	2.5	Изменение названия. Переработка содержания

Список TODO

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

- Интерпретация — описать аборт
- 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.
- Описать split transactions
- Заключение — написать перспективы развития.