УДК 004.41 DOI: 10.17587/it.25.157-164

Е. А. Юлюгин, инженер по разработке программного обеспечения, e-mail: evgeny.yulyugin@intel.com, Intel Sweden AB

Корректное и быстрое исполнение отдельных инструкций архитектуры Intel® 64 в виртуальном окружении

Архитектура Intel®64 постоянно развивается. Новые процессоры обычно сохраняют обратную совместимость и включают всю программно доступную функциональность предыдущих поколений. Было обнаружено, что поведение некоторых инструкций может нарушить совместимость при исполнении внутри виртуальной машины. В работе описываются разработанные методы и инструменты, позволяющие создавать быстрое и корректное виртуальное окружение, использующее аппаратные средства виртуализации.

Ключевые слова: *Wind River Simics®, виртуализация, моделирование, симуляция, x86, Intel VT-x, гипервизор, lzcnt*

Введение

В настоящее время программное моделирование является одним из важнейших элементов процесса создания вычислительных систем. Моделирование позволяет начать разработку и отладку программного стека задолго до появления аппаратного решения, тем самым минимизируя временной интервал между выходом аппаратной платформы и программного обеспечения.

Использование программных симуляторов остается актуальным даже после выпуска новой аппаратной платформы, так как предоставляет возможности для отладки, отсутствующие в физических системах [1]. Для того чтобы быть применимой для загрузки и отладки полного программного стека, модель должна демонстрировать высокую скорость работы, сохраняя при этом функциональную корректность. Увеличение производительности возможно с помощью прямого исполнения кода целевой системы на хозяйском оборудовании, если моделируемая и хозяйская архитектуры частично совпадают.

В работе описываются современные подходы к функциональному моделированию вычислительных систем. Акцент делается на изучении технологии прямого исполнения, позволяющей достичь скорость моделирования, близкую к скорости работы аппаратуры. В рамках этой работы был обнаружен и описан новый класс инструкций, расширяющий классификацию Попека и Голдберга [2] и делающий существующий способ организации прямого исполнения, основанный на технологии "trap and emulate", неприменимым для ряда важных сценариев. Цель исследования заключается в разработке нового метода, позволяющего организовать эффективное и корректное прямое исполнение кода, содержащего инструкции обнаруженного класса. Работа была проведена на основе функционального симулятора Wind River* Simics* [1].

1. Обзор литературы

Существующие подходы к моделированию процессоров можно разделить на три типа: программное моделирование, аппаратная виртуализация и их комбинация.

1.1. Программное моделирование

Программное моделирование является гибким решением, основанным на технологиях интерпретации и двоичной трансляции [3]:

- *интерпретация* — наиболее универсальный и аппаратно независимый, однако самый медленный подход к моделированию процессоров. Для увеличения производительности Bochs [4] и другие симуляторы, основанные исключительно на технологии интерпретации, обычно используют только переносимые и архитектурно независимые оптимизации, такие как трассы инструкций и кэширование результатов трансляции адресов;
- *двоичная трансляция* (англ. binary translation) позволяет наиболее эффективно моделировать простые инструкции, сокращая накладные расходы, связанные с декодированием и выборкой следующей инструкции для исполнения.

Многие симуляторы, такие как Zsim [5], совмещают использование технологий интерпре-

тации и двоичной трансляции. Данные подходы являются наиболее гибкими, так как не полагаются на аппаратные возможности конкретной платформы. Однако часто они не обеспечивают необходимую производительность.

1.2. Аппаратная виртуализация

Возможность создания эффективного монитора виртуальных машин была теоретически обоснована в 1974 г. Ж. Попеком и Р. Голдбергом [2]. Для доказательства они использовали упрощенную модель вычислительной системы, состоящую из центрального процессора и линейной сегментированной оперативной памяти. Периферийные устройства в рамках предложенной модели опускаются; центральный процессор поддерживает два режима работы: режим пользователя (англ. user mode), исполняющий прикладные программы, и режим супервизора (англ. supervisor mode), в котором исполняется код операционной системы. Память поддерживает механизм сегментации, который используется для организации виртуальной памяти.

Считается, что для некоторых входных условий инструкция рассматриваемого процессора вызывает ловушку (англ. trap), если результатом ее исполнения является сохранение состояния процессора в одну ячейку памяти и загрузка нового состояния из другой. Иначе говоря, ловушка позволяет сохранить состояние программы перед исполнением инструкции и передать управление обработчику, обычно работающему в режиме супервизора. Управление может быть возвращено обратно с помощью восстановления сохраненного состояния. Ловушки бывают двух типов: 1) ловушки потока управления — вызванные попыткой изменить состояние процессора; 2) ловушки защиты памяти — вызванные обращением к содержимому памяти, выходящему за пределы заданного сегмента. Данные типы не взаимоисключающие, т. е. результатом исполнения инструкции могут быть одновременно ловушки защиты памяти и потока управления.

Инструкции рассматриваемого процессора можно разделить на три класса [3]:

- привилегированные (англ. privileged) — инструкции, которые безусловно вызывают ловушку при исполнении в режиме пользователя;
- служебные (англ. sensitive). Данный класс состоит из двух подклассов: 1) инструкции, которые могут менять режим работы процессора или размер и положение доступного сегмента памяти; 2) инструкции, поведение которых зависит от режима процессора или конфигурации используемого сегмента памяти;
- безвредные (англ. innocuous) — самый широкий класс, включающий в себя все инструкции, не являющиеся служебными. Поведение инструкций этого класса не зависит ни от режима работы процессора, ни от настроек сегмента памяти.

Достаточное условие построения монитора виртуальных машин (ВМ): множество служебных инструкций является подмножеством привилегированных инструкций.

Изначально архитектура Intel® 64 не выполняла достаточные условия построения монитора ВМ, что было исправлено с появлением расширения Intel VT-x [6]. Монитор виртуальных машин исполняется в режиме VMX root, в котором доступны команды, управляющие поведением виртуальных машин, работающих в режиме VMX non-root. Исполнение привилегированных или служебных инструкций в режиме VMX non-root вызывает VM-exit — передачу управления монитору виртуальных машин с последующим моделированием данной инструкции и возвращением управления гостевому программному обеспечению.

Аппаратные виртуальные машины программно моделируют только привилегированные и служебные инструкции, исполняя остальные операции на хозяйском процессоре (рис. 1). Данная технология называется "trap and emulate" [7] и используется во многих гипервизорах, таких как Oracle VirtualBox [8], Microsoft* Hyper-V [9], KVM [10], Vmware* ESXi [11].

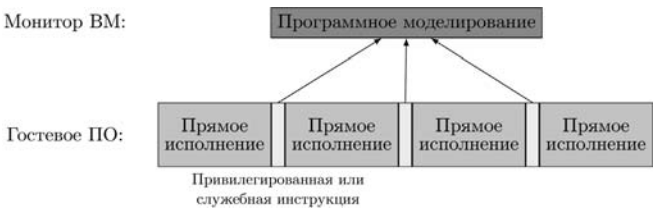

Рис. 1. Технология "trap and emulate", используемая при создании аппаратных систем виртуализации

1.3. Комбинированный подход

Программное моделирование позволяет создавать гибкие модели, предоставляя возможности для изучения недоступных в аппаратуре технологий. Однако производительность таких виртуальных машин остается достаточно низкой. Аппаратная виртуализация, напротив, позволяет достичь высокой скорости моделирования, не давая возможности использовать отсутствующие в аппаратуре инструкции.

Simics комбинирует технологии программного и аппаратного моделирования. Интерпретация и двоичная трансляция используется для

моделирования привилегированных, служебных и отсутствующих в аппаратуре инструкций. Режим прямого исполнения VMP [12], основанный на технологии Intel VT-x, позволяет обеспечить максимальную производительность при моделировании процессоров, основанных на архитектуре Intel 64. Технологии программного моделирования также используются в случаях, когда прямое исполнение оказывается неэффективным [13].

Simics — функционально точный полноплатформенный симулятор, позволяющий создавать высокопроизводительное виртуальное окружение, способное моделировать поведение любых вычислительных схем, начиная с одной платы и заканчивая многопроцессорными и многомашинными системами. Разработанные с помощью Simics модели позволяют исследовать, тестировать, а также заниматься отладкой немодифицированного программного обеспечения. Созданное виртуальное окружение способно работать как с низкоуровневым ПО, таким как BIOS и операционные системы, так и с любыми приложениями, оперирующими на уровне пользователя.

2. Одна кодировка — разные инструкции

Теория, разработанная Попеком и Голдбергом в 1974 г., все еще актуальна для задачи создания виртуального процессора, имеющего в точности такую же архитектуру, что и хозяйский. Однако существует ряд задач, для которых архитектура виртуализуемого процессора не полностью совпадает с хозяйской:

- динамическая миграция (англ. live migration) — перенос виртуальной машины с одной физической системы на другую без прекращения ее работы. При этом архитектура хозяйских систем, между которыми происходит миграция, может отличаться. Например, гипервизоры, такие как Microsoft* Hyper-V и VMware* ESXi, реализуют динамическую миграцию между системами с процессорами Intel и AMD* разных поколений, т. е. поддерживающих разные наборы инструкций;
- программное моделирование, основанное на комбинированном подходе, подразумевает, что архитектуры гостевого и хозяйского процессоров могут отличаться. При этом нередко гостевая система содержит функциональность, недоступную на данный момент в реальной аппаратуре.

В рамках выполненного исследования был обнаружен новый класс команд процессора, расширяющий теорию Попека и Голдберга. На первый взгляд, исследуемые инструкции могут быть отнесены к классу безвредных, но результат их исполнения может отличаться на разных поколениях процессоров одной и той же архитектуры. Аналогично привилегированным данные инструкции угрожают корректному исполнению в виртуальном окружении, однако в отличие от первых они не могут быть перехвачены с помощью аппаратуры. Существующие механизмы виртуализации не могут быть использованы для обнаружения и предотвращения некорректного исполнения инструкций, входящих в обнаруженный класс, так как они не вызывают исключения ловушки.

2.1. Переиспользование кодировок

В 2007 г. компания AMD* выпустила процессор с кодовым именем Barcelona [14], в котором комбинация REP префикса (байт 0xF3) и кодировки инструкции BSR была переиспользована в качестве кодировки для новой инструкции — LZCNT. Такое переиспользование привело к тому, что одна и та же последовательность байтов стала соответствовать разным инструкциям в зависимости от поколения процессора. Данные инструкции имеют разную семантику, так что попытка моделирования новой инструкции с помощью прямого исполнения на старой аппаратуре приводит к неправильному результату, поскольку заданная последовательность байтов интерпретируется аппаратурой как другая инструкция. Инструкция LZCNT была добавлена в процессоры Intel начиная с Intel® Core™ четвертого поколения (кодовое имя Haswell) в 2013 г.

Единственный публично известный способ для уменьшения эффектов обнаруженной проблемы — ограничение возможностей виртуальной машины до безопасного набора команд [9], выпущенного в 2010 г. Ограничение возможностей виртуального процессора реализуется с помощью инструкции идентификации процессора CPUID. Например, Microsoft Hyper-V всегда сообщает отсутствие поддержки инструкции LZCNT, если включен режим динамической миграции. Такое ограничение устанавливает излишне жесткие границы на производительность системы, так как только более старые инструкции могут быть использованы. Более того, данный подход неприменим для задачи программного моделирования, поскольку возможности, которые должна реализовывать виртуальная машина, строго зафиксированы в этом случае.

Инструкция CPUID может быть использована для определения наличия поддержки новой инструкции заданным процессором.

Тестовая программа, написанная на языке C (листинг 1), проверяет согласованность CPUID информации и результата исполнения последовательности байтов, соответствующей инструкции LZCNT. Программа выполняет инструкцию CPUID для того, чтобы определить наличие поддержки LZCNT, а затем исполняет последовательность байтов 0xF30FBDD8, которая соответствует либо инструкции "lzcnt ebx, eax", либо "bsr ebx, eax" в зависимости от поколения процессора, на котором эта программа будет выполняться. Результат исполнения, записанный в регистр EBX, затем используется для того, чтобы определить, какая инструкция была исполнена. Ожидаемым результатом исполнения программы считается печать сообщения "LZCNT is supported. 0xF30FBD corresponds to LZCNT." или "LZCNT is not supported. 0xF30FBD corresponds to BSR.".

Ограничение возможностей виртуального процессора на самом деле не позволяет достичь корректного поведения виртуальной машины, и запуск тестовой программы покажет различный результат в зависимости от хозяйского оборудования, на котором будет исполняться виртуальная машина. Исполнение данной тестовой программы внутри виртуальной машины, работающей под управлением Microsoft* Hyper-V в режиме динамической миграции, при использовании хозяйского процессора Intel Core i7-2600, не поддерживающего инструкцию LZCNT, показало правильный результат: "LZCNT is not supported. 0xF30FBD corresponds to BSR.". Результат исполнения той же конфигурации с использованием хозяйского процессора Intel Core i5-4300U, поддерживающего LZCNT, оказался неверным: "LZCNT is not supported. 0xF30FBD corresponds to LZCNT.".

Обратная ситуация происходит, когда программные симуляторы, такие как Simics, используют аппаратную поддержку виртуализации для моделирования гостевого кода через прямое исполнение. Модель более нового процессора сообщает о поддержке LZCNT, тогда как более старый хозяйский процессор, на котором будет происходить прямое исполнение, эту инструкцию не реализует. Кодировка, соответствующая новой инструкции, будет исполнена как старая, что приведет к неверному результату исполнения тестовой программы: "LZCNT is supported. 0xF30FBD corresponds to BSR.".

Листинг 1. Проверка корректности отображения функциональности LZCNT

```
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>

int main () {
    uint32_t ecx, cpuid_leaf = 0x80000001;
    asm volatile("cpuid" :"=c"(ecx) :"a"(cpuid_leaf) :"ebx", "edx");
    bool has_lzcnt = ecx & (1 << 5);
    printf("LZCNT is%s supported.", has_lzcnt ? "" : " not");

    uint32_t out, in = 0x11aa00bb;
    // "lzcnt ebx, eax" or "bsr ebx, eax" depending on CPU generation
    asm volatile(".byte 0xf3, 0x0f, 0xbd, 0xd8" :"=b"(out) :"a"(in));
    bool lzcnt = out == 0x3;
    printf(" 0xF30FBD corresponds to %s.\n", lzcnt ? "LZCNT" : "BSR");

    return 0;
}
```

Одновременно с инструкцией LZCNT в процессорах Intel Core четвертого поколения была добавлена инструкция TZCNT, входящая в состав расширения архитектуры команд BMI. Кодировка TZCNT является комбинацией REP префикса и кодировки инструкции BSF, которая не вызывает исключение ловушки при исполнении на процессорах предыдущих поколений. Операции, выполняемые инструкциями BSF и TZCNT, не взаимозаменяемы и поэтому также угрожают корректности моделирования при использовании прямого исполнения.

2.2. Инструкции сохранения и восстановления контекста XSAVE

Семейство инструкций XSAVE было впервые добавлено в процессорах Intel Core второго поколения одновременно с набором векторных инструкций Intel Advanced Vector Extension (Intel AVX). Расширение XSAVE реализует сохранение и восстановление компонент состояния (регистров или частей регистров) процессора в память. Данное расширение состоит из восьми инструкций:

- XSAVE, XSAVEC, XSAVEOPT, XSAVES — инструкции, сохраняющие состояние регистров процессора в память;
- XRSTOR, XRSTORS — инструкции, восстанавливающие состояние процессора из памяти;
- XGETBV, XSETBV — инструкции, оперирующие над расширенными регистрами управления (англ. extended control register, XCR).

Инструкции сохранения и восстановления состояния процессора принимают значение контрольного регистра XCR0 неявным аргументом. Каждый бит регистра XCR0 соответствует определенному компоненту состояния процессора. Различные поколения процессоров могут поддерживать разное число компонент состояния. Наборы команд, требующие использования XSAVE, называются XSAVE-enabled расширениями. Intel AVX, Intel AVX2, Intel AVX-512 и Intel Memory Protection Extension (Intel MPX) являются XSAVE-enabled-расширениями набора команд. Данный список

может быть расширен в дальнейшем при введении новых расширений для архитектуры Intel 64.

Моделирование инструкций расширения XSAVE через прямое исполнение не вызывает никаких проблем, если гостевое программное обеспечение использует компоненты состояния, которые поддерживаются хозяйским оборудованием. Если же гостевое ПО настроило регистр XCR0 так, что активен хотя бы один компонент, не поддерживаемый хозяйской аппаратурой, то прямое исполнение инструкций расширения XSAVE должно быть запрещено, так как хозяйские инструкции не могут корректно обработать отсутствующие компоненты состояния. Исполнение инструкций XSAVES и XRSTORS в режиме VMX non-root зависит от настроек контрольной структуры VMCS и может быть перехвачено. Инструкция XSETBV в режиме VMX non-root безусловно вызывает передачу управления монитору виртуальных машин. Поведение остальных инструкций расширения XSAVE в режиме VMX non-root никак не изменено, однако исполнение всех инструкций расширения может быть запрещено через контрольный регистр CR4.

В отличие от инструкций LZCNT и TZCNT исполнение инструкций, входящих в расширение XSAVE, может быть отключено при прямом исполнении и тем самым перехвачено. Однако вместе с инструкциями XSAVE также будет запрещено исполнение всех XSAVE-enabled-инструкций. В момент исследования XSAVE-enabled-расширения включали в себя несколько сотен инструкций, которые были недоступны для прямого исполнения из-за расширения XSAVE, если гостевое ПО включило хотя бы одну компоненту состояния, не поддерживаемую хозяйским оборудованием.

2.3. Известные проявления проблемы

Было обнаружено неправильное поведение некоторых операционных систем при исполнении в виртуальной среде. Например, ОС Google* Android* версии 4.4, основанная на ядре Linux* 3.10, использует инструкцию LZCNT независимо от идентификационной информации, доступной через CPUID. Ограничение возможностей виртуального процессора, реализованное с помощью инструкции CPUID, не даст ожидаемого эффекта в данном случае. Загрузка операционной системы Android в виртуальной среде, использующей технологию прямого исполнения, на хозяйском оборудовании, не поддерживающем инструкцию LZCNT, завершится ошибкой, так как вместо инструкции LZCNT будет исполнена инструкция BSR.

Операционные системы Linux и Android используют инструкцию TZCNT. На старом оборудовании, не поддерживающем расширение BMI, кодировка этой инструкции будет исполнена как инструкция BSF. Семантика этих инструкций очень близка, но не взаимозаменяема ввиду нескольких ключевых особенностей — результатом исполнения TZCNT при нулевом входном значении является размер операнда, тогда как результат исполнения инструкции BSF не определен в этом случае. Инструкция TZCNT также обновляет флаг переноса (англ. carry flag), тогда как значение этого флага после исполнения BSF не определено. Компилятор GCC может генерировать инструкцию TZCNT даже если целевая архитектура не поддерживает ее [15], таким образом, делая эту инструкцию достаточно популярной в современном программном обеспечении. Использование BSF вместо TZCNT в виртуальном окружении обычно не является опасным вследствие незначительной разницы в их поведении, однако подобная замена все же может привести к некорректному исполнению.

Инструкции семейства XSAVE используются всеми известными современными операционными системами, работающими на архитектуре Intel 64, для переключения контекста (англ. context switch) и инициализации процессов. Также большинство гипервизоров, включая Simics, используют XSAVE-инструкции для управления состоянием виртуальных машин.

3. Обеспечение корректного моделирования

Решение задачи обеспечения корректного моделирования исследуемых инструкций заключается в том, чтобы предотвратить прямое исполнение этих инструкций в виртуальном окружении — они должны быть промоделированы программно. Выявление подобных инструкций должно быть сделано заранее, так как исправить ошибку после исполнения невозможно.

В аппаратуре код виртуальных машин организован в страницы — выровненные блоки памяти фиксированного размера (типичный размер 4 Кбайт). С каждой страницей связан набор флагов, определяющих права доступа к соответствующей странице: страница может быть доступна для записи, чтения, исполнения или для комбинации этих прав.

Перед тем как разрешить исполнение кода, находящегося на какой-либо гостевой странице, она должна быть просканирована на наличие шаблонов, описывающих проблемные

инструкции. Если совпадения не найдены, то данная страница помечается безопасной. В дальнейшем исполнение с этой страницы происходит напрямую с помощью аппаратуры. Страница должна быть защищена от записи, так как ее содержимое должно быть пересканировано после каждого изменения.

Задача определения наличия заданных инструкций в памяти не является тривиальной ввиду особенностей архитектуры Intel 64 — переменной длины команды с неопределенными границами и эффектов, возникающих на границах страниц памяти. Исполнение может происходить по любому смещению внутри страницы, кодировка инструкции может начинаться на одной странице, а заканчиваться на следующей. Полное декодирование является медленным и требует знания всех точек входа на заданной странице, что требует полного анализа потока управления. В данной работе предлагается более простое и быстрое решение — частичное декодирование с поиском предопределенного набора битовых последовательностей. Например, для того чтобы убедиться, что на заданной странице памяти нет инструкции LZCNT, сканер шаблонов должен убедиться, что на этой странице не встречаются последовательности байтов [0xF3, 0x0F, 0xBD] и [0xF3, REX.W, 0x0F, 0xBD], где REX.W имеет значение в интервале 0x40-0x4F.

В результате сканирования страницы гостевого кода делятся на три типа (рис. 2):

- безопасные — просканированные страницы, не содержащие проблемных инструкций;
- опасные — просканированные страницы, вероятно содержащие проблемные инструкции, хотя это не может быть доказано без изучения истории исполнения кода, находящегося на этой странице;
- новые страницы, еще не прошедшие сканирование.

Рис. 2. Классификация страниц памяти в зависимости от наличия проблемных инструкций

Рис. 3. Цикл жизни страницы

Листинг 2. Последовательность инструкций, интерпретируемая как LZCNT

```
0fbaf30f        btr ebx, 0xf
bdddccbbaa      mov ebp, 0xaabbccdd
```

Следует отметить, что описанный механизм сканирования позволяет гарантировать отсутствие опасных инструкций на заданной странице, но не позволяет с точностью определить наличие данных инструкций. Точное определение наличия таких инструкций на заданной странице требует анализа потока управления или изучение истории исполнения. Листинг 2 показывает пример последовательности инструкций, которая будет определена сканером шаблонов как LZCNT, так как кодировка, соответствующая последовательности этих инструкций, содержит непрерывный набор байтов 0xf30fbd.

Содержимое страницы должно быть пересканировано, если была проведена запись в эту страницу, так как инструкции, содержащиеся на этой странице, могли быть изменены. Цикл жизни страницы с кодом изображен на рис. 3. Начальное состояние — невыделенная страница.

4. Измерения

Рис. 4 (см. вторую сторону обложки) показывает время загрузки основных операционных систем, измеренное при использовании трех различных режимов моделирования:

- оригинальный VMP — немодифицированный алгоритм прямого исполнения, основанный на технологии аппаратной виртуализации Intel VT-x;
- адаптированный VMP — алгоритм прямого исполнения, реализующий изменения, описанные в разделе 3;
- выключенный VMP — программное моделирование, основанное на технологиях интерпретации и двоичной трансляции.

Эксперименты проводили на рабочей станции с одним центральным процессором Intel® Xeon® E3-1270 v5 3.60 ГГц (микроархитектура Skylake) и 64 Гбайт ОЗУ. Использовалась 64-битная операционная система SUSE Linux Enterprise Server 11. Измерения проводили на симуляторе Wind River Simics версии 5. Для оценки влияния нового алгоритма на производительность симулятор сканировал и исключал из прямого исполнения страницы гостевой памяти, содержащие шаблоны инструкций LZCNT и TZCNT, несмотря на то что в этом не было необходимости, так как и моделируе-

мая, и хозяйская системы поддерживали эти инструкции.

Измерения показали, что новый алгоритм в среднем в 1,18 раз медленнее, чем традиционный (но небезопасный) подход к организации прямого исполнения. Разработанная технология также оказалась в 3,32 раза быстрее, чем моделирование, основанное только на интерпретации и двоичной трансляции. Сканирование гостевой памяти не оказывает видимого влияния на производительность симулятора, что подтверждается на примере сценария загрузки операционной системы FreeBSD 10.3, время работы которого не изменилось. При этом все страницы памяти, с которых происходило исполнение, были изучены на предмет наличия шаблонов, соответствующих небезопасным инструкциям. Исследование показало, что все исполняемые страницы были безопасными.

4.1. Загрузка операционной системы Fedora 23

Загрузка операционной системы Fedora 23 (ядро 4.2.3) продемонстрировала наиболее существенное замедление при использовании модифицированного VMP-алгоритма — в 1,7 раз медленнее по сравнению с оригинальным механизмом прямого исполнения VMP. Моделируемая система состояла из двух процессоров Intel Xeon (микроархитектура Skylake) и имела 4 Гбайт оперативной памяти. 66 из 8706 просканированных страниц были помечены как небезопасные по причине наличия шаблона, соответствующего инструкции TZCNT. Около 9,7 млн инструкций TZCNT были промоделированы во время исполнения сценария, состоящего из 40,1 млрд инструкций. Все исполненные TZCNT инструкции находились на 64 различных страницах памяти. Шаблон, соответствующий инструкции LZCNT, был обнаружен только на одной странице. Ни одной инструкции LZCNT не было исполнено во время загрузки сценария.

Эти измерения показывают, что небезопасные инструкции нечасто используются современным программным обеспечением — 0,02 % от общего числа исполненных инструкций во время загрузки Fedora 23. Только 0,77 % страниц памяти были помечены как небезопасные и исключены из прямого исполнения. 99,5 % этих страниц на самом деле содержали инструкцию TZCNT. На первый взгляд, число небезопасных для прямого исполнения страниц может показаться незначительным, но блокирование этих страниц привело к существенному увеличению доли гостевого кода, исполненного с использованием технологий интерпретации и двоичной трансляции, — с 1,3 до 9,7 %. Также блокировка прямого исполнения с небезопасных страниц привела к значительному увеличению числа дорогостоящих переходов между режимами программного моделирования и прямого исполнения — 3,7 млн переключений для оригинального VMP алгоритма и 4,3 млн переключений для модифицированного VMP. Вместе существенное увеличение числа переключений между режимами моделирования и рост доли программного моделирования привели к увеличению времени работы сценария.

Заключение

Исследование показало, что существующая технология аппаратной виртуализации не может быть использована для моделирования будущих поколений процессоров той же архитектуры. Для обеспечения корректного и быстрого исполнения в виртуальной машине необходима комбинация аппаратной виртуализации и описанной программной технологии.

Данное исследование было начато во время работы над моделью процессора Intel Core четвертого поколения (кодовое имя Haswell), который расширил набор команд Intel 64 инструкциями LZCNT и TZCNT. Описанная технология позволила разработать быструю и корректную модель, которая использовалась для разработки программного обеспечения до появления процессора на рынке. Новая технология моделирования позволила загрузить и отладить основные операционные системы и гипервизоры, используя модель процессора Intel Core четвертого поколения, исполняющуюся на процессоре второго поколения, что было бы невозможно при использовании традиционной технологии аппаратной виртуализации.

Проведенное исследование показывает, что проблемные инструкции встречаются достаточно редко, так что большинство инструкций может быть исполнено напрямую, если известно, что они являются безвредными или привилегированными. Измерения показывают, что небольшое число гостевых страниц памяти требует принудительного программного моделирования. В результате работы было обнаружено, что современные процессоры Intel 64 содержат восемь инструкций, прямое исполнение которых может угрожать корректности моделирования.

Список литературы

1. **Aarno D., Engblom J.** Software and System Development using Virtual Platforms. 1st Edition. Elsevier, 2014. 366 p.

2. **Popek G. J., Goldberg R. P.** Formal Requirements for Virtualizable Third Generation Architectures // Communications of the ACM. 1974. P. 412—421.

3. **Речистов Г. С., Юлюгин Е. А., Иванов А. А. и др.** Основы программного моделирования ЭВМ: Учеб. пособие. М.: Издательство МФТИ, 2013. 222 с.

4. **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. 2008.

5. **Yair Lifshitz, Robert Cohn, Inbal Livni et al.** Zsim: A Fast Architectural Simulator for ISA Design-Space Exploration // 3rd Workshop on Infrastructures for Software/Hardware Co-Design. WISH-3. 2011.

6. **Intel Corporation.** Intel® 64 and IA-32 Architectures Software Developer's Manual. 2018. Vol. 1—4.

7. **Adams Keith, Agesen Ole.** A Comparison of Software and Hardware Techniques for x86 Virtualization // ASPLOS. 2006.

8. **Oracle Corporation.** Oracle VM VirtualBox® User Manual.

9. **Tulloch Mitch.** Understanding Microsoft Virtualization Solutions / Microsoft. 2 edition. Microsoft Press, 2010. 452 p.

10. **KVM** wiki. URL: http://www.linux-kvm.org/page/Main_Page (дата обращения: 16.08.2018).

11. **VMware** ESXi: The Purpose-Built Bare Metal Hypervisor. URL: https://www.vmware.com/products/esxi-and-esx.html (дата обращения: 16.08.2018).

12. **Wind River.** Simics Hindsight Guide 5.0. 2018.

13. **Agesen Ole, Mattson Jim, Rugina Radu, Sheldon Jeffrey.** Software Techniques for Avoiding Hardware Virtualization Exits // USENIX. 2012. P. 373—385.

14. **AMD.** Software Optimization Guide for AMD Family 10h and 12h Processors, 2011. 348 p.

15. **x86:** emit tzcnt unconditionally. URL: https://gcc.gnu.org/ml/gcc-patches/2012-04/msg01765.html (дата обращения: 16.08.2018).

E. A. Yulyugin, Software Engineer, e-mail: evgeny.yulyugin@intel.com,
Intel Corporation

Fast and Accurate Direct Execution of Particular Intel® 64 Instructions in Virtual Environment

Intel® 64 architecture processors are constantly evolving, with new generations regularly introduced on the market. A new processor is usually backwards compatible and includes all the software-visible functionality of previous generations. This allows existing software to run on the new hardware without changes or recompilation. However, the guarantee of backwards compatibility can break in certain scenarios when software is running in a virtual machine. We discovered that certain machine instructions behave differently on past and present generations of AMD Intel processors. The identified instructions extend traditional understanding of hardware-assisted virtualization developed by Popek and Goldberg in 1974. To solve this problem new methods and tools have been developed to establish correct and fast hardware-assisted virtual platform solution. The key to solving this problem is to make sure that no such instruction is allowed to be directly executed in a virtualized environment — they all have to be emulated using software techniques. Detection of the instructions has to be done in advance since the problem cannot be corrected after the execution. The detection is achieved by a pattern matching decoder that looks through the guest code in prior to its execution in hardware-assisted virtual environment. The developed solution for the problem has been implemented, tested and proven to work using Wind River Simics® virtual platform framework.

Keywords: *Wind River Simics, virtualization, software simulation, x86, Intel VT-x, hypervisor, lzcnt, virtual machine monitor, virtual platform, Simics, live migration*

DOI: 10.17587/it.25.157-164

References

1. **Aarno Daniel, Engblom Jakob.** Software and System Development using Virtual Platforms, 1st Edition. Elsevier, 2014. 366 p.

2. **Popek G. J., Goldberg R. P.** Formal Requirements for Virtualizable Third Generation Architectures // Communications of the ACM, 1974. P. 412—421.

3. **Rechistov G., Yulyugin E., Ivanov A. et al.** Fundamentals of computer simulation software. 2nd edition revised and enlarged. Textbook. MIPT, 2013. 222 p. (in Russian).

4. **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, 2008.

5. **Yair Lifshitz, Robert Cohn, Inbal Livni et al.** Zsim: A Fast Architectural Simulator for ISA Design-Space Exploration // 3rd Workshop on Infrastructures for Software/Hardware Co-Design. WISH-3, 2011.

6. **Intel Corporation.** Intel® 64 and IA-32 Architectures Software Developer's Manual. Vol. 1—4, 2018.

7. **Adams Keith, Agesen Ole.** A Comparison of Software and Hardware Techniques for x86 Virtualization // ASPLOS, 2006.

8. **Oracle Corporation.** Oracle VM VirtualBox® User Manual.

9. **Tulloch Mitch.** Understanding Microsoft Virtualization Solutions / Microsoft. 2 edition. Microsoft Press, 2010. 452 p.

10. **KVM** wiki. Available at http://www.linux-kvm.org/page/Main_Page (date of access: 16.08.2018).

11. **VMware** ESXi: The Purpose-Built Bare Metal Hypervisor. Available at https://www.vmware.com/products/esxi-and-esx.html (date of access: 16.08.2018).

12. **Wind River.** Simics Hindsight Guide 5.0. — 2018.

13. **Agesen Ole, Mattson Jim, Rugina Radu, Sheldon Jeffrey.** Software Techniques for Avoiding Hardware Virtualization Exits // USENIX, 2012. P. 373—385.

14. **AMD.** Software Optimization Guide for AMD Family 10h and 12h Processors, 2011. 348 p.

15. **x86:** emit tzcnt unconditionally. Available at https://gcc.gnu.org/ml/gcc-patches/2012-04/msg01765.html (date of accesss: 16.08.2018).