Методы повышения производительности обратной отладки

М.А. Климушенкова <maria.klimushenkova@ispras.ru>
П.М. Довгалюк <pavel.dovgaluk@ispras.ru>
Новгородский государственный университет имени Ярослава Мудрого
173003, Россия, г. Великий Новгород, ул. Большая Санкт-Петербургская, д. 41

Аннотация. Обратная отладка — это инструмент разработки ПО, позволяющий более эффективно справляться с ошибками, возникающими при недетерминированном поведении программы. Она позволяет изучать прошедшие состояния программы без ее повторного запуска. В работе описана реализация обратной отладки на основе детерминированного воспроизведения в симуляторе QEMU 2.0. Предлагаются несколько способов повышения производительности отладки за счет сокращения дополнительно записываемых данных, оптимального сохранения снимков системы, индексации и сжатия журнала событий. Симулятор может работать совместно с интерактивным отладчиком GDB, что позволяет использовать команды reverse-continue, reverse-nexti, reverse-stepi и reverse-finish в процессе отладки. Скорость работы этих команд зависит от периода сохранения состояний системы в процессе записи ее работы. В статье представлена оценка наилучшего периода для оптимальной скорости работы команды reverse-continue.

Ключевые слова: обратная отладка; детерминированное воспроизведение; QEMU; симулятор

DOI: 10.15514/ISPRAS-2015-27(2)-8

Для цитирования: Климушенкова М.А., Довгалюк П.М. Методы повышения производительности обратной отладки. Труды ИСП РАН, том 27, вып. 2, 2015 г., стр. 127-144. DOI: 10.15514/ISPRAS-2015-27(2)-8.

1. Введение

Неотъемлемым этапом разработки каждой программы является отладка. Процесс отладки может оказаться очень трудоемким и занять непредсказуемое количество времени. Есть много инструментов, которые

помогают и облегчают программисту этот процесс, но сама процедура отладки, как правило, одина и та же.

Каждый раз поиск ошибки начинается с выполнения программы до момента ее первого проявления, после этого программист идет в обратном направлении по пути выполнения до причины ее возникновения. Классическим способом является циклическая отладка. Программист перезапускает программу, останавливает ее в более раннем моменте времени и изучает ее состояние с помощью отладчика и трассировки, так происходит несколько раз. Этот подход к отладке сложно использовать, если программа недетерминирована.

Основными трудностями являются:

- Недетерминированное поведение программы. Возникновение таких недетерминированных событий, как прерывания, переключения потоков, ввод от пользователя или получение данных из сети изменяет ход выполнения программы
- Программа работает длительное время до возникновения ошибки (часы, дни, месяцы)
- Сам факт отладки может повлиять на работу программы. При остановке или выполнении программы по шагам обмен сетевыми пакетами может быть нарушен из-за истечения таймаута соединения
- Непосредственное взаимодействие с аппаратной частью. Аппаратные устройства являются источниками недетерминизма, и алгоритм их работы может быть нарушен из-за задержек при отладке

Решение проблем, вызванных описанными трудностями, может быть достигнуто посредством использования механизма обратной отладки. Она позволяет программисту изучать уже прошедшие состояния программы без повторного ее выполнения с самого начала. Обратная отладка может быть реализована с помощью трассировки или детерминированного воспроизведения. В случае трассировки доступна только та информация о работе системы, которая была записана в трассу, для получения дополнительных данных необходимо повторно выполнить программу и записать новую трассу. В случае же детерминированного воспроизведения можно восстановить состояние всей системы и получить любую нужную информацию. По этой причине детерминированное воспроизведение является более предпочтительным способом реализации обратной отладки [1].

Для создания интерактивного обратного отладчика необходимо реализовать детерминированное воспроизведение работы отлаживаемой системы, механизм перехода на произвольное количество инструкций вперед и назад и возможность исследования состояния системы в заданный момент времени.

2. Детерминированное воспроизведение

2.1 Средства детерминированного воспроизведения

Путь выполнения любой программы зависит не только от передаваемых пользователем входных данных и окружения, в котором она запускается, но и от большого количества прочих факторов, которые сложно контролировать. Например, прерывания аппаратных устройств, моменты прихода сетевых пакетов и их содержимое, работа менеджера памяти системы и прочее. Во время отладки необходимо повторить тот же путь выполнения, при котором возникла ошибка. Сделать это вручную для сложных приложений реального времени крайне трудно. С помощью симулятора с поддержкой детерминированного воспроизведения можно повторить тот же самый путь [2]. Симулятор позволяет запустить внутри себя исследуемое приложение, которое будет функционировать так же, как если бы оно работало в реальной системе.

На рынке существует ряд программных симуляторов с функцией детерминированного воспроизведения. В своей основе все симуляторы используют механизмы виртуализации. Она может быть реализована как на аппаратном, так и на программном уровне.

Аппаратная виртуализация требует от хостовой платформы поддержки специальной процессорной архитектуры. При этом не требуется совпадения архитектур хостовой и гостевой систем, но требуется их совместимость. Например, возможна реализация 32-битных гостевых систем на 64-битных хостовых системах. Использование аппаратной виртуализации позволяет достичь высокой производительности, близкой к производительности невиртуализованной системы. Она используется в виртуальных машинах ReVirt [3], XenLR [4] и VMWare версии 7.0 [5], которые поддерживают детерминированное воспроизведение.

В части программной виртуализации можно выделить несколько основных подходов: паравиртуализация, динамическая трансляция и интерпретация. Техника паравиртуализации подразумевает внесение изменений в гостевую ОС, это возможно только, если ОС имеет открытый исходный код. При этом если необходимо эмулировать много аппаратных устройств, то количество правок будет велико. Паравиртуализация используется таким инструментом обратной отладки как TTVM[6], созданным на основе симулятора User-Mode Linux [7].

Динамическая трансляция представляет собой трансляцию машинного кода гостевой архитектуры в код архитектуры хостовой машины, при этом обе архитектуры никак между собой не связаны. Этот механизм реализован в симуляторе QEMU[8], который поддерживает широкий спектр аппаратных платформ. На основе QEMU был создан проект FREE [9] – еще один вариант детерминированного воспроизведения работы системы. Эта реализация поддерживает только гостевую архитектуру x86.

В основе полносистемного симулятора Bochs[10] лежит интерпретация гостевого кода, она медленнее динамической трансляции, однако более точно эмулирует поведение гостевой системы. На базе Bochs создан инструмент воспроизведения ExecRecoder [11], но сам симулятор эмулирует только архитектуру x86.

Чем меньше ограничений накладывает симулятор на возможную конфигурацию системы, тем шире спектр приложений, для которых можно применить обратную отладку, основанную на нем. Аппаратная виртуализация и паравиртуализация ограничивают архитектуру и ОС гостевой системы, что делает невозможным использование их, к примеру, при отладке мобильных приложений на настольных компьютерах. Динамическая трансляция и интерпретация позволяют выполнять код одной архитектуры на машине с другой архитектурой, то есть можно реализовать поддержку любой аппаратной платформы.

2.2 Корректность и производительность воспроизведения

Скорость работы системы внутри симулятора сравнима со скоростью ее работы на физической машине, поэтому симулятор, как правило, никак не влияет на ход выполнения приложения внутри него. Однако сохранение дополнительных данных, необходимых для воспроизведения, может существенно замедлить работу симулятора. Это будет критичным при отладке приложений, работающих в реальном времени, также может быть нарушено взаимодействие с сетью. Сохранение дополнительных данных включает в себя запись журнала недетерминированных событий и периодическое создание снимков системы, которые позднее используются при отладке. Журнал представляет собой файл, в который записываются недетерминированные события, происходящие в симуляторе [2]. Чем меньше информации будет сохраняться, тем меньше будет замедление. Для этого в журнал нужно записывать минимально возможное количество событий, в идеале только события от периферийных устройств, а снимки системы нужно получать уже во время воспроизведения ее работы, когда вызванное этим замедление ни на что не будет влиять.

В процессе воспроизведения порядок выполняющихся инструкций и состояние регистров процессора должны соответствовать порядку инструкций и состоянию регистров при выполнении программы в симуляторе во время записи ее работы.

3. Обратная отладка

3.1 Реализация функций отладчика

Для реализации функций отладки механизм детерминированного воспроизведения должен поддерживать совместную работу с интерактивным

отладчиком. Одним из наиболее подходящих является отладчик GDB [12], он доступен для большинства популярных платформ (Linux, Windows) и целевых архитектур, таких как Intel86, AMD64 и ARM.

Стандартные возможности отладчика позволяют останавливать программу в ходе ее выполнения, двигаться по пути выполнения и изучать состояние программы [13]. Большинство отладчиков могут двигаться только в прямом направлении выполнения. Однако, часто обратное направление движения будет более удобным для пользователя, чтобы найти причину ошибки.

Ядро отладчика GDB содержит поддержку таких функций обратной отладки, как reverse-continue, reverse-nexti, reverse-stepi и reverse-finish наравне с классическими командами continue, next, step, finish.

По команде step отладчик переходит к следующей инструкции, при необходимости заходя внутрь вызовов функций. Подобно ей, команда reverse-stepi переходит на предшествующую инструкцию, возможно заходя внутрь предыдущего вызова функции. Для работы этих команд при обратной отладке используется внутренний счетчик инструкций. При переходе вперед, процессор выполняет одну очередную инструкцию, а при переходе назад происходит загрузка предыдущего снимка системы, а затем воспроизведение кода до инструкции с порядковым номером на один меньше текущего.

Если же для программы доступен исходный код, а не только исполняемый файл, то можно выполнить команду reverse-step, которая переходит не к предыдущей инструкции, а к предыдущей строке кода.

Команда continue выполняет программу в прямом направлении до достижения какой-либо точки останова, аналогично ей, команда reverse-continue делает то же самое, только в обратном направлении. Традиционный отладчик вставляет в исполняемый код в точках останова программы инструкции-ловушки, которые передают управление отладчику при их срабатывании.

В случае обратной отладки симулятор не идет в обратном направлении исполнения кода в прямом смысле. По команде reverse-continue выполнение должно останавливаться в ближайшей предшествующей точке останова, поэтому симулятор загружает предыдущий снимок состояния системы и воспроизводит весь путь выполнения до текущего момента, чтобы найти все точки останова и определить последнюю. Далее он повторно загружает снимок и воспроизводит работу системы до уже известной точки. Если ни одной точки останова не нашлось, то весь процесс повторяется для следующего более раннего снимка.

Команда reverse-nexti аналогично команде reverse-stepi переходит на предыдущую инструкцию, однако в отличии от нее, не заходит внутрь вызовов функций. Если предыдущая инструкция являлась вызовом функции, то произойдет переход к состоянию предшествующему этому вызову, иначе команда работает также как reverse-stepi. По команде reverse-finish отладчик переходит к месту вызова функции, в которой он сейчас находится.

Все точки останова работают и при обратном направлении отладки, что позволяет, например, перейти к предыдущей точке, где была изменена переменная.

3.2 Скорость перехода на определенный шаг

Отладка программы часто происходит итеративно, практически невозможно с первого раза определить место, которое являлось причиной возникновения ошибки. В случае обратной отладки на это требуется много шагов назад. Если каждый из них будет приводить к длительному ожиданию, то весь процесс займет слишком много времени. Скорость перехода к шагу «в прошлом» зависит от удаленности ближайшего предшествующего снимка системы и времени на воспроизведение от этого снимка до интересующего шага.

Переход к предыдущему шагу во время обратной отладки происходит путем загрузки состояния системы, предшествующего интересующему шагу, с дальнейшим воспроизведением до нужного шага. От количества сохраненных состояний зависит скорость, с которой будет происходить переход назад, так как чем меньше интервал времени между ними, тем меньше инструкций нужно будет воспроизводить до искомого шага.

Простейший способ сохранения состояния виртуальной машины – это создание полной копии физической памяти, образа виртуального диска и состояния процессора и периферийных устройств. Однако этот способ является неэффективным по времени и по объему использующейся памяти. Решением этой проблемы может являться использование механизма копирования при записи, подобного применяемому в симуляторе TTVM [6]. Это позволяет существенно сократить накладные расходы на создание снимка виртуальной машины. Когда состояние сохраняется впервые, записывается все содержимое физической памяти, а для следующих состояний сохраняются только изменившиеся страницы. Они записываются в два лог-файла. Первый используется для получения «следующих состояний», то есть хранит изменения состояния n+1 относительно состояния n. Второй используется для получения «предыдущих состояний», хранит изменения состояния n-1 относительно состояния n. Образ виртуального диска хранится аналогичным образом, сохраняются только изменившиеся блоки.

Дополнительные состояния можно добавлять и удалять путем внесения правок в лог-файлы. Если журнал записывается в течение длительного времени, состояний накапливается очень много, так как изначально они сохраняются с фиксированным интервалом. Для экономии дискового пространства некоторые из них нужно постепенно удалять. По умолчанию, чем ближе период к моменту возникновения ошибки, тем больше состояний для него оставляется, так как он более интересен для отладки. Состояния для удаления выбираются таким образом, чтобы расстояние по времени между соседними возрастало экспоненциально в соответствии с удаленностью по времени от момента возникновения ошибки. Для ускорения переходов,

программист может определить более короткий интервал при сохранении состояний для выбранной части журнала, которую он отлаживает чаще всего [6].

Если состояния сохранялись редко, то скорость перехода к предыдущим шагам будет сильно зависеть от скорости воспроизведения. На нее влияют количество записываемых в журнал событий, накладные расходы от реализации механизма воспроизведения внутри симулятора и насколько полно воспроизводится работа всей системы (воспроизводится работа только процессора, воспроизводится работа периферийных, выводится ли изображение от видеокарты в окно симулятора).

Средняя скорость работы команды reverse-continue зависит от величины интервала сохранения снимков системы [14]. При слишком маленьком интервале негативное влияние начинают оказывать накладные расходы от загрузки снимка системы, а при слишком большом - происходит воспроизведение большого участка журнала и теряется преимущество от использования снимков. Оптимальный интервал можно найти эмпирическим путем для конкретной реализации детерминированного воспроизведения и конкретной отлаживаемой программы.

Время, необходимое для работы команде reverse-continue, зависит от количества инструкций до ближайшей точки останова. Симулятор загружает предыдущий снимок системы и воспроизводит журнал с этого состояния до текущего положения в поиске точки останова, если она не найдена, то загружает следующий снимок и снова воспроизводит журнал, процесс повторяется до тех пор, пока не будет найдена точка останова или до начала выполнения программы. Если ближайшая точка останова находится на расстоянии в m инструкций относительно текущей точки, то для ее поиска нужно будет загрузить d снимков системы и воспроизвести столько же участков журнала между соседними снимками состояния.

$$d = \left\lceil \frac{m}{n} \times k \right\rceil, \quad (1)$$

где m - количество инструкций от ближайшей точки останова до текущей точки;

n - количество инструкций от начала журнала на текущей точки;

k - количество сохраненных снимков с начала записи журнала до текущего момента;

Время для загрузки одного снимка и воспроизведение журнала до следующего снимка можно выразить формулой (2).

$$T_d = T_k + t_s, \quad (2)$$

где T_k - время воспроизведения журнала между двумя соседними снимками состояния;

t_s - время загрузки состояния;

$$T_k = \frac{n}{k} \times t_i\ , \quad (3)$$

где t_i - время воспроизведения одной инструкции;

Тогда общее время нахождения ближайшей точки останова можно выразить формулой (4).

$$T_m = d \times T_d = \left\lceil \frac{m}{n} \times k \right\rceil \times (T_k + t_s) = \left\lceil \frac{m}{n} \times k \right\rceil \times \left(\frac{n}{k} \times t_i + t_s \right) \quad (4)$$

После того, как точка найдена, происходит воспроизведение журнала с загруженного состояния до состояния срабатывания точки останова. Требующееся на это время можно посчитать по формуле (5).

$$T_r = \left(\frac{n}{k} \times d - m \right) \times t_i \quad (5)$$

В результате с учетом формул (4) и (5) время выполнения команды reverse-continue равно выражается формулой (6)

$$\begin{aligned} T_m = d \times T_d + T_r &= d \times (T_k + t_s) + \left(\frac{n}{k} \times d - m \right) \times t_i \\ &= d \times (T_k + t_s) + \frac{n}{k} \times d \times t_i - m \times t_i = \\ &= d \times \left(T_k + t_s + \frac{n}{k} \times t_i \right) - m \times t_i \\ &= \left\lceil \frac{m}{n} \times k \right\rceil \times \left(\frac{n}{k} \times t_i + t_s + \frac{n}{k} \times t_i \right) - m \times t_i \\ &= \left\lceil \frac{m}{n} \times k \right\rceil \times \left(2 \times \frac{n}{k} \times t_i + t_s \right) - m \times t_i \end{aligned} \quad (6)$$

Для каждой реализации из формулы (6) можно вычислить оптимальный период сохранения снимков системы.

3.3 Модификация журнала для повышения производительности

Как правило, при отладке удается локализовать проблему, и далее работа идет ни со всем сценарием работы программы, а только с его частью. Поэтому удобно выделить фрагмент сценария, и работать только с ним.

При выполнении команды reverse-continue воспроизведение программы происходит в обратном порядке до первой встретившейся точки останова. Заранее неизвестно когда последний раз выполнялась искомая инструкция или менялась искомая ячейка памяти, поэтому симулятор будет последовательно в обратном порядке загружать сохраненные состояния и выполнять код между ними, чтобы найти место, где ему следует остановиться.

Значительно ускорить этот процесс позволяет индексация выполненного кода. В процессе записи журнала дополнительно сохраняется информация о том, когда какая инструкция была выполнена и какие элементы памяти записала [15]. Весь процесс записи журнала разбивается на небольшие временные промежутки. Для поиска момента, когда был изменен регистр или выполнена

инструкция, используются промежутки, соответствующие выполнившимся базовым блокам машинных команд. Запись о каждом промежутке содержит диапазон адресов выполнившихся инструкций и битовую маску изменившихся в этом блоке регистров. Для поиска изменений в ячейке памяти, вся память разбивается на страницы, для каждой страницы сохраняется своя история из временных промежутков, записи в которой содержат битовую маску измененных ячеек. Когда нужно найти место последнего изменения переменной, вся история проходится в обратном порядке, проверяя битовые маски, а затем загружается нужное состояние.

Если до возникновения ошибки программа работает очень длительное время, то размер журнала станет достаточно большим и может достигать несколько сотен гигабайт. Уменьшить размер файла журнала можно с помощью его сжатия. При воспроизведении его части будут последовательно восстанавливаться. Другим способом сокращения размера журнала является выделение из него фрагмента, относящегося к исследуемой ошибке. Сохранение снимков системы позволяет получить из журнала фрагмент, начинающийся с произвольного снимка, и использовать его в качестве полноценного журнала.

4. Исследование гостевой системы

Использование полносистемного симулятора в качестве основы для реализации обратной отладки имеет как преимущества, такие как возможность отладки компонентов операционной системы и драйверов, так и недостатки. Симулятор выполняет код не только исследуемого приложения, а всей системы целиком. Если приложение собрано с отладочной информацией и доступен его исходный код, то можно указать отладчику путь к исполняемому и исходным файлам. В этом случае выполнять команды отладки и ставить точки останова можно в исходном коде. Если же эта информация отсутствует, то отлаживать приходится ассемблерный код, что более трудоемко.

Если ошибка возникает в каком-то компоненте операционной системы, для которого нет исходного кода, то получить дополнительную информацию о системы и ее объектах можно с помощью анализа дампа памяти и таких инструментов, как интерактивные отладчики. Симулятор с функцией детерминированного воспроизведения позволяет перейти в любое место выполнения программы, а затем с помощью дополнительных инструментов можно исследовать систему.

Дамп оперативной памяти, полученный в момент выполнения программы в симуляторе, можно исследовать с помощью инструмента Volatility Framework [16]. Он включает в себя широкий спектр плагинов для получения списка активных процессов и загруженных библиотек, соответствия физических и

виртуальных адресов, списка модулей/драйверов/потоков ядра и многого другого.

Если гостевой системой является Windows, то для отладки можно использовать отладчик WinDBG [17]. Он позволяет отлаживать пользовательские приложения и драйвера, анализировать аварийные дампы, автоматически загружать символьную информацию для модулей Windows. Поддерживается возможность отладки удаленной машины, для этого в отлаживаемой системе запускается отладочный сервер. Он работает внутри системы, поэтому при воспроизведении интерактивная отладка становится невозможной.

В SDK платформы Android используется симулятор QEMU, на его базе можно реализовать обратную отладку мобильных приложений. Большинство приложений реализованы на языке java, для их выполнения в систему входит виртуальная машина Dalvik. При отладке с использованием GDB симулятор выполняет код этой виртуальной машины, а не java код, что существенно осложняет поиск ошибки. Для отладки java кода в SDK включен отладчик DDMS [18]. Он подключается к порту симулятора или реального устройства и позволяет получить информацию о выполняющихся потоках, процессах и динамических объектах. Отладчик интерактивно взаимодействует с симулятором, посылая ему команды и получая запрашиваемую информацию, поэтому при воспроизведении получение информации, отличной от полученной при записи, невозможно. Воспользоваться DDMS при воспроизведении можно, если остановить симулятор, отключить воспроизведение, получить данные из симулятора, а далее продолжить воспроизведении с приостановленного состояния [19].

5. Предлагаемая реализация обратной отладки

Наиболее используемыми в настоящее время в персональных компьютерах и мобильных устройствах являются процессоры архитектуры i386, AMD 64 и ARM. Немного симуляторов поддерживают все эти платформы, одним из них является симулятор QEMU. Он использует динамическую трансляцию, поддерживает такие целевые платформы как i386, AMD 64, ARM (есть версия для запуска Andriod), PowerPC, MIPS и другие. Имеет открытый исходный код, что позволяет добавлять поддержку новых платформ и периферийных устройств.

Ранее в симуляторе QEMU были реализованы функции детерминированного воспроизведения и обратной отладки [2]. QEMU поддерживает симуляцию не только процессора, а всей системы, включая состояние памяти, периферийных устройств и экрана симулятора. Эта особенность позволяет отлаживать работу драйверов для периферийных устройств даже в том случае, когда само устройство еще не выпущено, для чего можно смоделировать его внутри симулятора.

В основе механизма обратной отладки лежит детерминированное воспроизведение. В процессе работы приложения записывается журнал недетерминированных событий, происходящих внутри системы, именно он позднее позволяет повторить весь сценарий ее работы. К недетерминированным событиям относятся события срабатывания системных таймеров, прерывания и ввод-вывод от периферийных устройств.

При воспроизведении работы системы, события считываются из журнала и выполняются в те же моменты (по отношению к выполнению гостевой программы), в которые они были записаны. Использование журнала событий позволяет в точности повторять сценарий работы системы любое количество раз. Всякое замедление, возникающее при воспроизведении, никак не влияет на работу гостевой системы, так как не происходит реального ее взаимодействия с внешними источниками данных. Это свойство позволяет использовать алгоритмы анализа или снятия трассы, которые обычно дают сильное замедление и не могут быть использованы во время реальной работы из-за возможного изменения поведения исследуемой программы.

Для обеспечения контроля точности используется запись в журнал при необходимости дополнительных событий типа assert. При записи такого рода событий сохраняются значения ряда регистров гостевого процессора. Во время воспроизведения при считывании этих событий происходит сравнение значений регистров из журнала и состояние процессора симулируемой системы в данный момент. Если значения отличаются, то происходит завершение работы симулятора с сообщением об ошибке. Это позволяет избежать зависания или зацикливания при неверном воспроизведении журнала.

В процессе записи журнала событий должно быть сохранено содержимое сетевых пакетов, которые были отправлены или получены от виртуальной сетевой карты. Все сетевые пакеты сохраняются в отдельных файлах pcap, их можно анализировать отдельно с помощью программ анализаторов сетевого трафика подобных Wireshark [20]

QEMU поддерживает совместную работу с отладчиком GDB. Так как отлаживается всегда работа всей системы целиком, исследуемое приложение необходимо загрузить в ОС, установленную в симуляторе. С помощью команд обратной отладки в GDB, можно перемещаться по пути выполнения приложения в обратном направлении. Любая из этих команд требует длительное время для выполнения, поэтому важно рассчитать оптимальный период сохранения состояний системы для конкретной реализации обратной отладки, чтобы максимально сократить время выполнения команд обратной отладки.

Используя формулу 6, для QEMU с функциями обратной отладки можно построить график, изображенный на рисунке 1, зависимости времени выполнения команды reverse-continue от количества сохраненных снимков. Все данные получены для тестового случая загрузки ОС WindowsXP, в их

числе общее количество выполнившихся инструкций от начала записи журнала $n \approx 5 \cdot 10^9$, время воспроизведения одной инструкции $t_i \approx 6 \cdot 10^{-8}$ и время загрузки состояния $t_s \approx 3{,}5$.

На рис. 1 изображена зависимость времени выполнения команды reverse-continue от количества сохраненных снимков для симулятора QEMU. Графики построены для разного значения m – количество шагов до ближайшей точки останова.

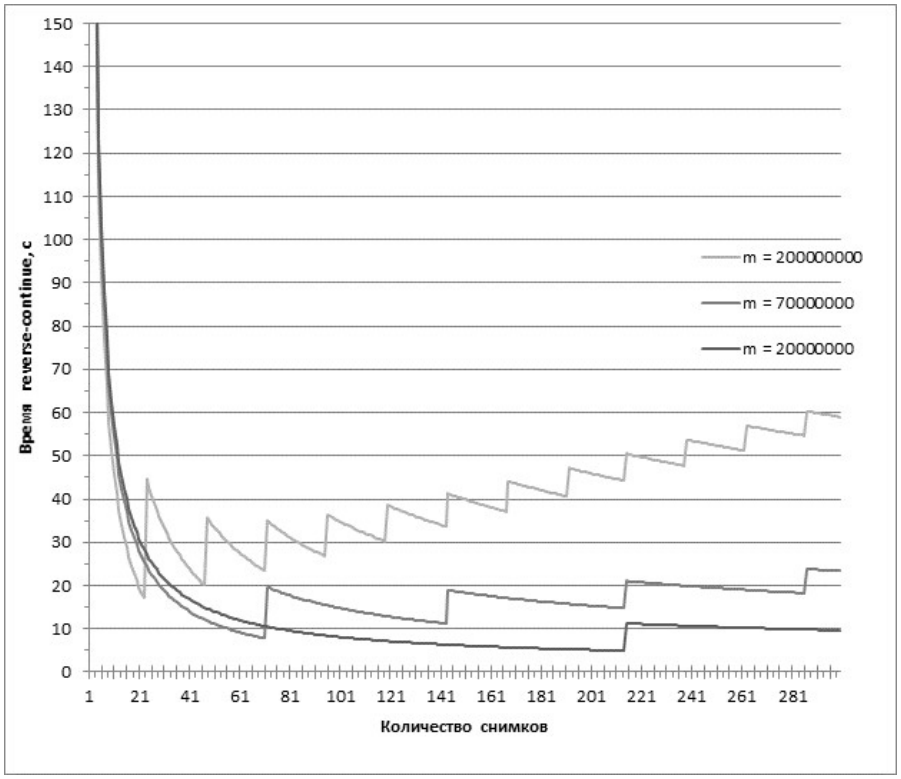

Рис. 1. Зависимость времени работы команды reverse continue от числа снимков состояния виртуальной машины, сделанных при записи ее работы.

Шаг, на котором встретится ближайшая точка останова, не известен, поэтому изображено три графика для различного среднего количества шагов, на которое необходимо вернуться команде reverse-continue.

В табл. 1 приведены результаты вычислений оптимального периода сохранения снимков при записи журнала, при котором время работы команд обратной отладки будет наименьшим.

Табл. 1 Оптимальный период сохранения снимков виртуальной машины при различном расстоянии до точки останова.

Количество шагов до точки останова	Количество сохраненных снимков	Период сохранения снимков, с
$2 \cdot 10^8$	23	5
$7 \cdot 10^7$	71	1,5
$2 \cdot 10^7$	214	0,5

6. Направления дальнейших исследований

Дальнейшая работа над обратной отладкой в симуляторе QEMU планируется в следующих направлениях.

Работа со снимками состояний, перенос их сохранения из этапа записи журнала на этап его воспроизведения, что позволит повысить скорость работы симулятора при записи журнала. Когда есть уже записанный журнал, пользователь сможет менять период сохранения снимков без перезаписи журнала. Использование копирования при записи для сохранения снимков и добавление возможности удаления лишних снимков позволит сократить общий объем хранимых снимков.

Протокол отладчика GDB на данный момент не позволяет эффективно использовать обратную отладку с симулятором QEMU. Когда пользователь хочет выполнить команду reverse-step n, то отладчик посылает симулятору n раз команду reverse-step, что подразумевает загрузку предыдущего состояния и воспроизведение журнала до предыдущей команды. Если бы симулятор получил сразу команду reverse-step n, то за один проход перешел на нужную инструкцию. При выполнении команды reverse-continue можно кэшировать все найденные точки останова, чтобы при повторном ее выполнении, избежать лишнего прохода. GDB имеет открытый исходный код, если его доработать, то связка GDB и QEMU работала бы намного эффективней.

Эффективность обратной отладки напрямую зависит от скорости ее работы, поэтому важно повышать скорость как записи, так и воспроизведения работы программы. Этого можно добиться за счет уменьшения количества событий, записываемых в журнал.

Планируется также добавить возможность исследовать внутреннее состояние системы с помощью инструмента Volatility Framework. Он работает только с файлами дампов памяти, но можно доработать его и интерактивно получать необходимую информацию, подключаясь к работающему симулятору.

7. Заключение

В работе рассматривается обратная отладка с использованием многоплатформенного симулятора QEMU. Применение обратной отладки при создании программных продуктов упрощает нахождение трудновоспроизводимых ошибок, что может существенно сократить время и

затраты на разработку. В настоящий момент на рынке нет реализации обратной отладки, сравнимой по производительности с прямой отладкой.

Описываемый в статье вариант обратной отладки имеет в своей основе симулятор QEMU 2.0 с функцией детерминированного воспроизведения. Для повышения эффективности отладки в работе были предложены методы оптимального сохранения снимков системы, индексации и сжатия журнала событий. Реализованный метод обратной отладки работает с пользовательскими приложениями, ядрами операционных систем. Поддерживается работа с периферийными устройствами из «реального мира», что позволяет отлаживать разрабатываемые и сопровождаемые драйверы устройств.

Список литературы

[1]. Довгалюк П.М. Обратная отладка программного обеспечения: монография. НовГУ им. Ярослава Мудрого. Великий Новгород 2013. 72 стр.

[2]. Довгалюк П.М. Детерминированное воспроизведение процесса выполнения программ в виртуальной машине. Труды ИСП РАН, т. 21, 2011 г. стр. 123-132, ISSN 2220-6426 (Online), ISSN 2079-8156 (Print)

[3]. Dunlap, George W. and King, Samuel T. and Cinar, Sukru and Basrai, Murtaza A. and Chen, Peter M. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. ACM SIGOPS Operating Systems Review - OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation, vol. 36, 2002, pp. 211-224.

[4]. Haikun Liu, Hai Jin, Xiaofei Liao, Zhengqiu Pan. XenLR: Xen-based Logging for Deterministic Replay. In proc. of Japan-China Joint Workshop on Frontier of Computer Science and Technology (2008). pp. 149-154.

[5]. Integrated Virtual Debugger for Visual Studio Developer's Guide. http://www.vmware.com/pdf/ws7_visualstudio_debug.pdf. Дата обращения: 11.06.2015

[6]. Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. Proceedings of the 2005 USENIX Annual Technical Conference, USENIX, 2005, pp. 1-15

[7]. J. Dike. A user-mode port of the Linux kernel. InProceedings of the 2000 Linux Showcase and Conference, 2000

[8]. QEMU - open source processor emulator. http://wiki.qemu.org/Main_Page. Дата обращения: 11.06.2015

[9]. Chia-Wei Hsu, Shiuhpyng Shieh. FREE: A Fine-grain Replaying Executions by Using Emulation. The 20th Cryptology and Information Security Conference (CISC 2010), Taiwan, 2010.

[10]. Bochs -the cross platform IA-32 emulator. http://bochs.sourceforge.net. Дата обращения: 11.06.2015

[11]. Daniela A. S. de Oliveira, Jedidiah R. Crandall, Gary Wassermann, S. Felix Wu, Zhendong Su, and Frederic T. Chong. ExecRecorder: VM-based full-system replay for attack analysis and system recovery. Proc. of the 1st workshop on Architectural and system support for improving software dependability (ASID '06), 2006. pp. 66-71

[12]. GDB and Reverse Debugging. http://sourceware.org/gdb/news/reversible.html. Дата обращения: 11.06.2015
[13]. Bob Boothe. Efficient Algorithms for Bidirectional Debugging. Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2000, pp. 299-310
[14]. Marc Rittinghaus, Konrad Miller, Marius Hillenbrand, and Frank Bellosa. Simuboost: Scalable parallelization of functional system simulation. In 11th International Workshop on Dynamic Analysis, WODA '03, Houston, Texas, March 2013
[15]. Robert O'Callahan. Efficient Collection and Storage of Indexed Program Traces. https://www.cs.purdue.edu/homes/xyzhang/fall07/Papers/Amber.pdf. Дата обращения: 11.06.2015
[16]. Volatility - an advanced memory forensics framework. https://code.google.com/p/volatility/. Дата обращения: 11.06.2015
[17]. Standalone Debugging Tools for Windows (WinDbg). http://msdn.microsoft.com/en-us/windows/hardware/hh852365. Дата обращения: 11.06.2015
[18]. Using DDMS. http://developer.android.com/tools/debugging/ddms.html. Дата обращения: 11.06.2015
[19]. Фурсова Н.И., Довгалюк П.М., Васильев И.А., Климушенкова М.А., Макаров В.А. Способы обратной отладки мобильных приложений. Проблемы информационной безопасности. Компьютерные системы, номер 3, Санкт-Петербург 2014 г, стр. 50-56
[20]. Wireshark User's Guide. https://www.wireshark.org/docs/wsug_html/. Дата обращения: 11.06.2015

Methods to Improve Reverse Debugging Performance

M.A. Klimushenkova <maria.klimushenkova@ispras.ru>
P.M. Dovgalyuk <pavel.dovgaluk@ispras.ru>
Novgorod State University
st. B. St. Petersburgskaya, 41, Veliky Novgorod, 173003, Russia

Abstract. Reverse debugging is software development technique that effectively helps to fix bugs caused by nondeterministic program behavior. Bug elimination usually includes multiple reruns of the program Program executions may be non-deterministic and the debugger may affect program's behavior. Reverse debugging allows inspecting past program's states without re-executing it. The paper describes implementation of software reverse debugging using deterministic replay based on the QEMU emulator. Our implementation of deterministic replay records high-level events (user and network input, CPU interrupts, USB and audio input). Record/replay subsystem saves these events into the log at the recording phase and reads them at the replaying phase. Therefore virtual machine is not connected to the real world in the replaying phase. We present ways to improve

debugging performance by reducing saved data, using copy-on-write snapshots' format and indexing/compressing of replay log. QEMU supports a common user interface for reverse debugging in GDB debugger which allows using reverse-continue (going back to the previous breakpoint or watchpoint), reverse-nexti, reverse-stepi (going back to the previous instruction), and reverse-finish (finding the point when function was called) commands. Time required for these commands' execution depends on taking snapshots frequency in recording replay log. We evaluate shapshotting frequency to get the best reverse debugging performance. In our implementation optimal period for taking snapshots is 3.5 seconds. This paper also presents assessment of snapshots frequency for better performance.

Keywords: reverse debugging; deterministic replay; QEMU; emulator.

DOI: 10.15514/ISPRAS-2015-27(2)-8

For citation: Klimushenkova M.A., Dovgalyuk P.M. Methods to Improve Reverse Debugging Performance. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 2, 2015, pp. 127-144 (in Russian). DOI: 10.15514/ISPRAS-2015-27(2)-8.

References

[1]. Dovgalyuk P.M. Obratnaya otladka programmnogo obespecheniya: monografiya. Novgorod State University. Velikij Novgorod 2013. 72 p. (in Russian)
[2]. Dovgalyuk P.M. Determinirovannoe vosproizvedenie processa vypolnenija programm v virtual'noj mashine. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 21, 2011, pp. 123-132, ISSN 2220-6426 (Online), ISSN 2079-8156 (Print) (in Russian)
[3]. Dunlap, George W. and King, Samuel T. and Cinar, Sukru and Basrai, Murtaza A. and Chen, Peter M. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. ACM SIGOPS Operating Systems Review - OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation, vol. 36, 2002, pp. 211-224.
[4]. Haikun Liu, Hai Jin, Xiaofei Liao, Zhengqiu Pan. XenLR: Xen-based Logging for Deterministic Replay. In proc. of Japan-China Joint Workshop on Frontier of Computer Science and Technology (2008). pp. 149-154.
[5]. Integrated Virtual Debugger for Visual Studio Developer's Guide. http://www.vmware.com/pdf/ws7_visualstudio_debug.pdf. Access date: 11.06.2015
[6]. Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. Proceedings of the 2005 USENIX Annual Technical Conference, USENIX, 2005, pp. 1-15
[7]. J. Dike. A user-mode port of the Linux kernel. InProceedings of the 2000 Linux Showcase and Conference, 2000
[8]. QEMU - open source processor emulator. http://wiki.qemu.org/Main_Page. Access date: 11.06.2015
[9]. Chia-Wei Hsu, Shiuhpyng Shieh. FREE: A Fine-grain Replaying Executions by Using Emulation. The 20th Cryptology and Information Security Conference (CISC 2010), Taiwan, 2010.

[10]. Bochs -the cross platform IA-32 emulator. http://bochs.sourceforge.net. Access date: 11.06.2015
[11]. Daniela A. S. de Oliveira, Jedidiah R. Crandall, Gary Wassermann, S. Felix Wu, Zhendong Su, and Frederic T. Chong. ExecRecorder: VM-based full-system replay for attack analysis and system recovery. Proc. of the 1st workshop on Architectural and system support for improving software dependability (ASID '06), 2006. pp. 66-71
[12]. GDB and Reverse Debugging. http://sourceware.org/gdb/news/reversible.html. Access date: 11.06.2015
[13]. Bob Boothe. Efficient Algorithms for Bidirectional Debugging. Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2000, pp. 299-310
[14]. Marc Rittinghaus, Konrad Miller, Marius Hillenbrand, and Frank Bellosa. Simuboost: Scalable parallelization of functional system simulation. In 11th International Workshop on Dynamic Analysis, WODA '03, Houston, Texas, March 2013
[15]. Robert O'Callahan. Efficient Collection and Storage of Indexed Program Traces. https://www.cs.purdue.edu/homes/xyzhang/fall07/Papers/Amber.pdf. Access date: 11.06.2015
[16]. Volatility - an advanced memory forensics framework. https://code.google.com/p/volatility/. Access date: 11.06.2015
[17]. Standalone Debugging Tools for Windows (WinDbg). http://msdn.microsoft.com/en-us/windows/hardware/hh852365. Access date: 11.06.2015
[18]. Using DDMS. http://developer.android.com/tools/debugging/ddms.html. Access date: 11.06.2015
[19]. Fursova N.I., Dovgalyuk P.M., Vasil'ev I.A., Klimushenkova M.A., Makarov V.A. Sposoby obratnoj otladki mobil'nyh prilozhenij. Problemy informacionnoj bezopasnosti. Komp'juternye sistemy, vol. 3, Saint Petersburg 2014, pp. 50-56 (in Russian)
[20]. Wireshark User's Guide. https://www.wireshark.org/docs/wsug_html/. Access date: 11.06.2015