Karlsruher Institut für Technologie

Towards Heterogeneous Record and Replay on the ARM Architecture

Masterarbeit
von

BSc. Inform. Simon Veith

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Frank Bellosa
Zweitgutachter:	Prof. Dr. Wolfgang Karl
Betreuender Mitarbeiter:	Marc Rittinghaus

Bearbeitungszeit: 01. Juni 2016 – 31. Januar 2017

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 31.01.2017

Abstract

Record and replay is a technique that enables full-system debugging at high execution speeds. During execution of a virtual machine, all non-deterministic input to the system is saved to a log file. Later, this log file can be used to feed the recorded data back into another virtual machine instance to reproduce the exact execution as in the original system, accurate to the instruction level.

Examining a system's operation on an instruction-by-instruction basis with little runtime overhead enables more insights for the purpose of software verification or security analysis, particularly with regard to the mobile devices and server markets.

While the ARM architecture has been gaining in popularity during the past decade, and the software running on such machines has greatly increased in complexity, most work in the field of record and replay has thus far focused on the x86 architecture. A deterministic record and replay implementation for ARM exists [21], recording non-deterministic inputs at the virtual device level, but it does not support hardware virtualization.

This work evaluates the feasibility of a heterogeneous record and replay solution for the ARMv7 architecture. We identify the sources of non-determinism in an ARM computer system and demonstrate our implementation of a low-level record and replay scheme with QEMU. Our modified virtual machine software can successfully perform homogeneous record and replay of unmodified guests using the Tiny Code Generator (TCG) binary translator.

We have implemented recording for the heterogeneous case, using the Linux Kernel-based Virtual Machine (KVM) as the hypervisor. However, we have identified several issues in the architecture's design and its implementations that complicate a successful replay: Certain instructions exhibit non-deterministic behavior that cannot be recorded; however, with appropriate correction mechanisms, successful replay may yet be possible.

Although runtime overheads of up to 40 % have been observed in the KVM-accelerated recording system, it still operates faster than a TCG-based virtual machine running on high-end x86 machines.

Deutschsprachige Zusammenfassung

Die Verbreitung der ARM-Prozessorarchitektur hat in den vergangenen Jahren stark zugenommen: Der Großteil der heute verkauften Smartphones verwendet einen Prozessor der ARM-Architektur, und auch in der IT-Branche werden ARM-basierte Server aufgrund ihres geringeren Energieverbrauchs und der höheren Integrationsdichte in Betracht gezogen.

Mit der gesteigerten Leistungsfähigkeit aktueller ARM-Prozessoren ging jedoch auch eine erhöhte Komplexität der Rechnersysteme einher. Während frühere Mobilgeräte ein kleines, aber dafür ausgiebig getestetes Betriebssystem verwendeten, kommt auf aktuellen Smartphones Software zum Einsatz, die sich in Aufbau und Größe kaum noch von Desktop- und Server-Anwendungen unterscheidet.

Bei der Verifikation und der Fehlersuche auf diesen Systemen stoßen traditionelle zyklische Debuggingmethoden schnell an ihre Grenzen: Bedingt durch die hohe Interaktivität – etwa in Form von Nutzereingaben auf einem Smartphone oder durch Netzwerkkommunikation bei Servern – können Defekte nicht immer zuverlässig reproduziert werden. In einigen Fällen verhindert die Ausführung mittels eines Debuggers gar die korrekte Funktionsweise des Systems oder führt dazu, dass das Problem durch geänderte zeitliche Abläufe nicht mehr auftritt. Auch ist nicht immer von vornherein klar, ob der Defekt im Betriebssystem, in Funktionsbibliotheken oder in der Anwendungssoftware selbst vorliegt, weshalb gegebenenfalls mehrere Debugging-Durchläufe notwendig sind; der Fehler muss dabei für jeden Durchlauf erneut reproduziert werden.

Das Konzept der Ausführungswiederholung (engl. *record and replay*) bietet sich als eine mögliche Lösung für diese Probleme an. Hierbei macht man sich zunutze, dass Computer im Allgemeinen so entworfen sind, dass sie ein deterministisches Verhalten aufweisen: Die selben Eingaben führen immer zur selben Ausgabe. Zwar verarbeitet ein modernes Computersystem die Eingaben aus unzähligen Quellen, etwa von Zeitgebern oder aus dem Netzwerk, wodurch sich auch unter scheinbar identischen Bedingungen nichtdeterministische Abweichungen im Laufzeitverhalten ergeben. Ein solches System lässt sich jedoch in ei-

ner virtuellen Maschine simulieren, wodurch alle Eingaben präzise aufgezeichnet werden können. Mit dieser Aufzeichnung lässt sich später eine erneute Ausführung der virtuellen Maschine starten, welche die gespeicherten Daten als Eingabe erhält und so exakt dem gleichen Programmablauf folgt.

Die Ausführung in einer virtuellen Maschine bringt jedoch wiederum eigene Probleme mit sich, allen voran die reduzierte Ausführungsgeschwindigkeit bedingt durch die Simulation mittels Übersetzung. Aktuelle Prozessoren verfügen daher über Erweiterungen für Hardwarevirtualisierung, mit denen die virtuelle Maschine zum Großteil direkt auf der physikalischen CPU des Hostsystems ausgeführt werden kann und so eine hohe Ausführungsgeschwindigkeit gewahrt wird.

Die Erweiterungen für Hardwarevirtualisierung lassen sich auch zum Zwecke der Ausführungswiederholung nutzen, um nichtdeterministische Eingaben in einem System aufzuzeichnen, ohne dabei dessen Zeitverhalten zu stark zu verändern. Dieses Konzept wurde in Projekten wie Aftersight [16] und V2E [39] erfolgreich für die x86-Prozessorarchitektur umgesetzt.

Bedingt durch die weite Verbreitung der x86-Architektur lag der Fokus bestehender Arbeiten auf dem Gebiet der Ausführungswiederholung zumeist auf dieser Plattform. Der zunehmende Marktanteil der ARM-Architektur und der damit einhergehende Bedarf an Werkzeugen zur Fehlersuche machen jedoch die Technik der Ausführungswiederholung auch für diese Systeme interessant.

Unsere Arbeit befasst sich daher mit dem Thema der Ausführungswiederholung auf der ARM-Architektur. Das Ziel war es, zu überprüfen ob sich die Architektur für ein solches Aufzeichnungsschema prinzipiell eignet und welche Ereignisse hierfür berücksichtigt werden müssten.

In vorigen Publikationen wurden verschiedene Ereignisklassen identifiziert, welche für die Ausführungswiederholung relevant sind. Diese lassen sich unterteilen in synchrone Ereignisse, die durch Instruktionen in der virtuellen Maschine ausgelöst werden, sowie asynchrone Ereignisse, deren Ursprung außerhalb der virtuellen Maschine liegt. Auf der ARM-Architektur umfassen die synchronen Ereignisse die Coprozessor-Instruktionen sowie Lesezugriffe von Speicherbereichen für Memory-Mapped I/O (MMIO). Asynchrone Ereignisse sind externe Interrupts sowie Schreibzugriffe per Direct Memory Access (DMA) durch virtuelle Geräte.

Für die Ausführungswiederholung ist es von erheblicher Bedeutung, dass die gespeicherten Ereignisse zum exakt richtigen Zeitpunkt in die virtuelle Maschine eingespielt werden, da es ansonsten zu einem unterschiedlichen Ausführungsverlauf kommen kann. Zu diesem Zweck werden sogenannte Landmarken eingesetzt, die einen Zeitpunkt während der Ausführung eines Systems eindeutig kennzeichnen. Die ARM-Architektur bietet hierfür Ereigniszähler an, welche die Anzahl der ausgeführten Instruktionen erfassen können. In Verbindung mit den Virtualisierungserweiterungen können diese Zähler die virtuelle Maschine und das Hostsystem getrennt betrachten. Neben dem Instruktionszähler wird in die Landmarke

auch ein Abbild der aktuellen CPU-Register aufgenommen, sodass fehlerhafte Zählerwerte erkannt und korrigiert werden können. Dieser Mechanismus kann auch verwendet werden, um einen falschen Programmablauf bei der Wiederholung zu erkennen.

Wir stellen in dieser Arbeit einen Entwurf und eine Implementierung eines Systems zur Ausführungswiederholung auf der ARMv7-Architektur vor. Unsere Implementierung basiert auf QEMU, einem Simulator für virtuelle Maschinen, der Codeübersetzung mittels des Tiny Code Generators (TCG) unterstützt, um eine virtuelle Maschine komplett softwaregestützt auszuführen. Darüber hinaus wird auch die Ausführung mittels der Linux Kernel-based Virtual Machine (KVM) unterstützt, welche – eine passende Host-Architektur vorausgesetzt – die Hardware-Virtualisierungserweiterungen für eine beschleunigte Ausführung nutzt.

Beide Ausführungsmodi wurden von uns um eine Aufnahmemöglichkeit erweitert, sodass sowohl eine homogene Ausführungswiederholung möglich ist (Aufnahme und Wiederholung mit TCG), als auch der heterogene Fall (Aufnahme mit KVM, Wiederholung mit TCG). Die Aufzeichnung im KVM-Modus erforderte Änderungen am Linux-Kernel sowie die Verwendung der Hardware-Debugerweiterung, was einige Einschränkungen für das Gastsystem mit sich brachte.

Im Verlauf unserer Arbeit wurden einige architekturelle Probleme festgestellt, die eine hardwaregestützte Aufzeichnung erschweren: Die ARM-Architektur sieht einen eingebauten Zeitgeber vor, der sich weder deaktivieren noch aufzeichnen lässt, wordurch sich der virtuellen Maschine eine Quelle für Nichtdeterminismus bietet. Zwar kann man etwa im Linux-Kernel die Verwendung dieses Zeitgebers deaktivieren; dies ist jedoch nicht generell für alle Gastsysteme möglich.

Des Weiteren verhält sich die *Store Exclusive*-Operation, die für atomare Operationen eingesetzt wird, nichtdeterministisch: Der Erfolg beziehungsweise das Fehlschlagen der Operation wird durch einen unterschiedlichen Rückgabewert angezeigt. Die Spezifikation der ARM-Architektur gestattet es, dass eine Hardware-Implementierung die Instruktion aus internen Gründen fehlschlagen lässt, etwa aufgrund von Verdrängungen aus dem Zwischenspeicher, und in unseren Experimenten beobachteten wir, dass dieses Verhalten in einem Cortex-A15-Kern tatsächlich auftritt. Da eine fehlgeschlagene Store Exclusive-Operation in der Regel einen zusätzlichen Schleifendurchlauf erzeugt, führt dies zu unterschiedlichen Zählerwerten bei der Wiederholung der Aufnahme im TCG-Modus, der dieses nichtdeterministische Verhalten nicht nachbilden kann.

Die Store Exclusive-Operation, die sich nicht durch den Hypervisor abfangen lässt und die auch von unprivilegiertem Code verwendet werden kann, führt somit zu erheblichen Problemen bei der Ausführungswiederholung. Trotz Korrekturmechanismen, die unsere Implementierung beinhaltet, konnte somit bisher keine erfolgreiche heterogene Ausführungswiederholung durchgeführt werden.

Nichtsdestotrotz führten wir für unsere Arbeit eine Evaluierung der Aufzeichnungsgeschwindigkeiten und der dabei auftretenden Datenraten durch. Dabei traten im KVM-Modus abhängig von den durchgeführten Tests Verlangsamungen von 1 % (im Falle von CPU-lastigen Prozessen) bis zu 40 % (bei I/O-lastigen Prozessen) auf. Wir vermuten den Hauptgrund für die Verlangsamungen in der fehlenden Unterstützung für DMA-Operationen in der Implementierung der virtuellen SD-Karte; dennoch war die Aufzeichnung in jedem Fall schneller als eine Simulation im TCG-Modus.

Trotz der aufgetretenen Probleme konnten wir zeigen, dass eine deterministische Ausführungswiederholung grundsätzlich möglich und vorteilhaft ist, wenngleich man die möglichen Gastsysteme einschränken muss und noch weitere Korrekturmechanismen notwendig sind. Unsere Arbeit konzentrierte sich auf den Einzelkern-Fall und die ARMv7-Architektur; weitere Arbeiten sind notwendig für die Aufzeichnung von virtuellen Mehrkern-Systemen oder von Systemen mit der Nachfolgearchitektur ARMv8.

Acknowledgments

Throughout my studies at the Karlsruhe Institute of Technology, I have always felt particularly at home in the Operating Systems division. I have completed many projects and tutoring jobs there and was lucky to be working with very competent supervisors: Konrad Miller, Marius Hillenbrand, and my Master's Thesis supervisor, Marc Rittinghaus. My thanks go out to them for always being understanding, friendly and helpful to us students. It has been a pleasure working with you.

Contents

Chapter 1

Introduction

The ARM processor architecture is gaining popularity due to the increasing availability of processor designs and rapid advances in computing performance. Cloud computing providers are evaluating ARM-based servers for their higher efficiency and integration density, and manufacturers of mobile devices such as smartphones have already established ARM as the standard architecture in both budget and high-end devices.

At the same time, the complexity of those computing systems has increased dramatically in the past 10 years. Where previous handsets have consisted of a comparatively weak CPU and a small but well-tested operating system, today's smartphones generally run a software stack whose size rivals that of desktop PCs. Numerous software bugs in Android's Linux kernel, the system libraries or user applications have been found in the past years, with more critical bugs expected to be as of yet undiscovered. Similarly, on the server side, vulnerabilities such as Heartbleed, which permitted access to critical memory locations, have stayed undetected for a long time.

Detecting or debugging such errors with traditional cyclic debugging techniques has been difficult because of the system's high degree of interactivity, either in the form of user interaction through mouse or touch screen, or through network communication in the case of a server. These interactions cannot always be reproduced faithfully to trigger the bug. Furthermore, the act of debugging the system may mask certain issues which do not occur while the system is being observed (so called *Heisenbugs*) due to different timing.

Observing and reproducing a software defect generally requires a guess in advance as to which subsystem or which layer contains the bug. When doing a post-mortem analysis of an intrusion into a server, it may not be clear which attack vector was used. System log files are of limited use because they may not hold enough detail to understand the issue at hand. It would be helpful to have a method of analyzing different layers of a system—such as the operating system

kernel, the system libraries, or the user applications—without having to reproduce the problem anew.

In high-performance systems, it is often a problem to debug issues which occur only on production servers, or which require a high system load to be triggered. In such a situation, it is neither possible to attach a debugger to single-step through the execution, nor to reproduce the issue on command. A debugging technique would be required that is performant enough to be active over long periods of time without significantly disturbing the productive use of a system.

The concept of *record and replay* promises to address these issues by exploiting the fact that a computer system is, by design, mostly deterministic: Feeding a computer the same inputs multiple times should always lead to it producing the same output. Thus, in theory, when launching a software system under identical conditions, it should follow the same execution flow each time. While modern computing systems are very complex, processing inputs from many sources, it is possible to record these non-deterministic inputs. The recorded data can later be fed back into a running system to make it behave identically to the original recording system.

Originally, such deterministic record and replay systems were implemented using virtual machine emulation software, causing very high overheads in execution time when compared to a hardware system. With the advent of hardware virtualization extensions, high-performance recording has become feasible, as demonstrated in projects such as Aftersight [16].

Given the large market penetration of the platform, it is understandable that most record and replay projects have concentrated on the x86 architecture, despite its many features which make it hard to virtualize [31]. The ARM architecture, however, is rapidly gaining both in popularity and in performance, giving rise to a need for debugging techniques in order to manage the increased software complexity.

Our work seeks to evaluate how the concept of record and replay can be applied to the ARM platform. We examine the architecture to identify the non-deterministic events that need to be included in a recording in order to deterministically replay them in simulation later.

Virtual machines based on binary translation allow for system-level debugging, even across different processor architectures; yet still, execution speed of the debugged system remains an issue. It should be examined if hardware virtualization extensions, such as used by the Linux Kernel-based Virtual Machine (KVM), can be repurposed for record and replay in order to enable a low-overhead recording process. It is well known that the x86 platform had been hard to virtualize before the advent of virtualization extensions, due to subtleties e.g. in trap and exception handling; it shall be seen where the difficulties in the ARM architecture lie and whether they can be worked around.

To that end, we explore the non-deterministic events described in previous work for x86 and map them to the ARM architecture. For each event, we analyze which information must be recorded and how it can be obtained using the platform's virtualization extensions.

In order to feed back the recorded events at the correct time, so called landmarks as defined in [39] need to be used to identify the exact processor state at the time of their occurrence. Hardware performance counters are often used for this purpose; however, previous projects have observed inaccuracies in the performance counters of x86 CPUs [37], raising concerns that similar problems exist in the ARM architecture's performance counters. We test their accuracy to determine whether additional information about the machine's state must be collected to uniquely identify a point in time during execution.

Our contribution to the topic of record and replay is an analysis of the ARMv7 architecture with respect to replayability, as well as a design for a hypervisor-based recording system on this architecture. We demonstrate a working deterministic record and replay system for software-emulated virtual machines, and show that the heterogeneous case—recording through hardware virtualization—is not easily possible on ARMv7 due to non-deterministic behavior of certain instructions. Despite these unresolved issues, the prototype of our heterogeneous system can replay a large part of the boot process of a Linux guest kernel.

We present benchmark results showing that, while hardware-virtualized recording carries a significant overhead compared to execution without it, it is still faster than performing a simulation entirely in software.

1.1 Thesis Outline

Chapter 2 introduces the reader to the topic of virtualization, giving a general overview of the ARM architecture and its extensions for hardware virtualization, as well as presenting previous work concerning deterministic record and replay. The operating principles of the QEMU virtual machine simulator, forming the basis for our implementation, are outlined.

Chapter 3 applies the concept of record and replay to ARM, showing where the events identified in previous work for x86 can be found in this architecture, and how the performance monitoring extensions can be used to obtain a landmark.

Chapter 4 presents the design of our system for record on replay on the ARMv7 architecture, describing the involved components and which information is collected for defining a landmark.

Chapter 5 details our implementation of the design using QEMU, with the Linux Kernel-based Virtual Machine (KVM) used for recording and the Tiny Code Generator (TCG) for replaying. We describe the issues we encountered,

as well as the mechanisms we implemented to correct a mismatched landmark resulting from non-determinism during recording.

Chapter 6 evaluates the performance and correctness of our recording system implementation, as well as the efficacy of the correction mechanisms we implemented to work around non-deterministic behavior.

Chapter 7 summarizes our work, presenting the results as well as suggesting possible improvements for future work.

Chapter 2

Background

This chapter provides an introduction to the concept of record and replay by first giving an overview of virtualization in general, assuming that the reader already has basic knowledge of operating system concepts. The ARMv7 architecture targeted by our work is introduced, highlighting the key differences to the x86 architecture and showing how virtualization is done on ARM. Since the replay system developed in this work is based on the Linux Kernel-based Virtual Machine (KVM) subsystem and the QEMU virtual machine software, their principles of operations are explained before surveying previous work in the field of record and replay.

2.1 Virtualization

In computing, the term **virtualization** refers to the concept of transforming a system using physical hardware, e.g. an operating system running on an x86 processor, such that it no longer uses a physical instance of that resource, but a *virtualized* version instead [36]. These virtualized resources may or may not be directly backed by a physical realization—it is possible to simulate them or, as has become common practice in the cloud computing domain, share a single existing system among multiple virtual users on demand. This transformation allows for greater flexibility in fulfilling the architectural requirements of software and for better utilization of servers. Unlike *abstraction*, however, which intends to simplify an interface to reduce complexity, virtualization aims to present the same interface as before, in order to retain compatibility with existing software [36].

An important use of virtualization is during the development of new computing platforms: Software typically develops at a faster pace than hardware because physical implementations of computing architectures take a long time to design, verify and produce. Virtualizing a system allows for software to be developed for

architectures that do not exist yet. The desired system can be *simulated*, which creates the illusion that a particular machine exists, without actually having to realize it physically [27]. Such simulators are very flexible, in that the virtual machine environment they provide and the **guest system** running in it can be designed entirely different from the physical machine it runs on—the **host system**.

Virtualization is also used for debugging purposes on existing architectures, providing a portable and reproducible platform on which to study a program's execution. Security analysis benefits from virtualization in that malicious software can be isolated from production systems by confining it to a virtual machine where its behavior can be analyzed. The Argos project [32], for instance, uses taint analysis of incoming network packets to detect remote code execution attacks; an approach that would be impractical without the use of virtual machines. As demonstrated in the ReTrace project [38], virtualization allows for the collection of execution traces of the entire machine, including the operating system, which is not possible through traditional user level debugging methods.

A virtual machine simulator works by using the facilities provided by the host operating system—e.g., processes, files, networking, and virtual memory—to create a model representation of the guest system entirely within its own process environment. It keeps track of the virtual machine's state, including the central processing unit (CPU) registers, the contents of the system's memory, pending input and output, and the state of interrupt lines. Typically, it will execute in a looping fashion to determine which architectural action to perform next: decode and execute an instruction, trigger interrupts, or handle input from the host system. It is not required that the virtual architecture and the physical architecture match: for example, if the virtual machine requires more registers than are available on the host machine, the extra registers can be simulated in working memory. Even systems requiring the presence of a memory management unit (MMU) could be simulated on a machine that lacks it.

This flexibility, however, comes at the price of performance, typically causing slowdowns of 30 times the native execution time in the case of x86 [34]. Since code from the virtual machine cannot be executed directly on the host system, for lack of virtualization awareness, it must either be **interpreted** or **binary translated**. All aspects of the simulated architecture, such as the devices and memory layout, must be reproduced in a way faithful to how the corresponding physical hardware would handle them. Privilege levels need to be checked, the processor state (particularly conditional execution flags and status bits) must be updated, and accesses to input/output devices have to be mapped to the facilities of the host system.

A virtual machine using an interpreter analyzes one instruction of the guest code at a time: The next instruction is decoded and a routine in the virtual machine software is invoked to perform emulation of that instruction. While conceptually

simple, this technique allows for little optimization by the executing host CPU due to the complexity of the decoding loop, which has to run after every individual instruction.

A binary translator, as is exemplified in Figure 2.1 for the QEMU virtual machine software [13], reads an entire sequence of instructions to be executed and, via a platform-independent intermediate code stage, generates equivalent machine code for the host system. Virtual machine software typically employs *dynamic* binary translation at runtime because a distinction between guest machine code and data cannot be made statically, and due to the possibility of self-modifying code. The generated machine code can be cached and re-executed later, if necessary, avoiding costly re-compilation.

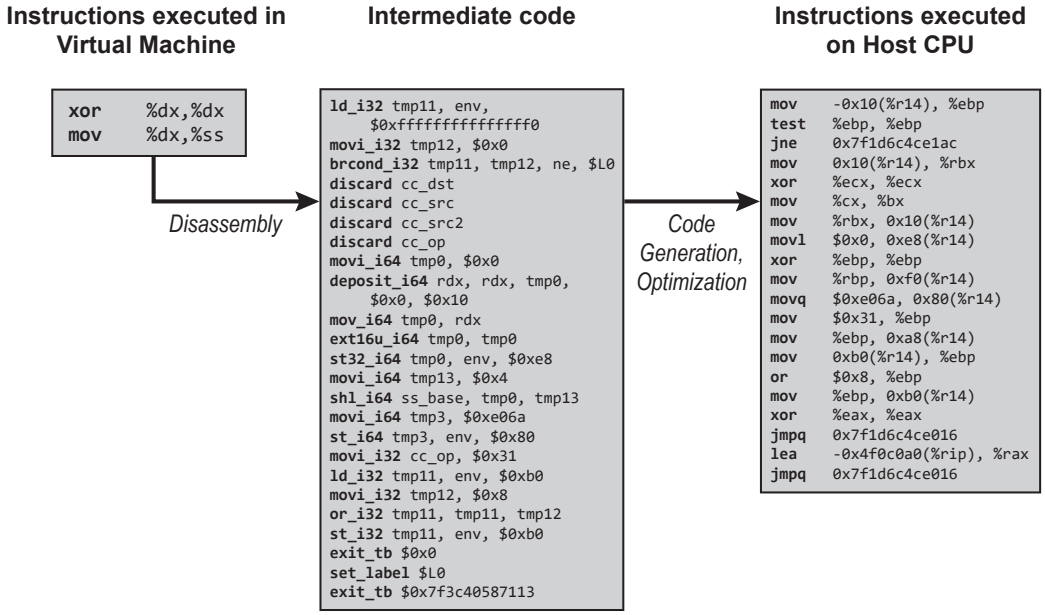


Figure 2.1: Example of x86 guest CPU code executed in a virtual machine hosted by QEMU, binary-translated using the Tiny Code Generator (TCG) to be run on an x86 host system. Notice how even for two simple arithmetic and data transfer instructions, executed on an identical host architecture, the resulting instruction count already increases tenfold, slowing down execution. The executed code in this particular case is from the virtual machine's BIOS boot-up code.

The code generated by dynamic binary translation can be augmented with tracing hooks, which enables, for instance, the detailed analysis of memory access patterns, and allows for profiling software execution. In that case, extreme slowdowns of up to 1000 times the execution time can be observed, depending on the level of detail [34].

While the host architecture and the guest architecture can be radically different from each other, the case where the two are identical is a valid and common use of virtual machines: Running a virtual machine on a host system of the same architecture is done for security reasons—to isolate a piece of software from sensitive resources, e.g., in malware research—or to reduce costs and improve utilization of hardware by way of consolidating servers. It has become cost-effective to no longer rent entire physical machines at a colocation facility, but to acquire and release virtual machine instances as needed.

Executing a virtual machine on a system with a matching architecture enables some of the simulated guest devices to be mapped directly to resources of the host machine, without the large overhead incurred by emulation. A significant part of the instruction set can be executed natively by the host system's CPU without having to resort to binary translation. Only the instructions that are sensitive to the virtualized machine state (e.g. I/O accesses) or that require privileged access must be trapped by the subsystem that controls the virtual machine instances. This subsystem is called the **virtual machine manager (VMM)** or **hypervisor** and is part of the operating system on the host machine. [27] It mediates the virtual machine's access to the host CPU and is called upon whenever a privileged operation or an aspect of the virtual hardware must be emulated.

Enabling efficient virtualization had for a long time been only an academic exercise, because the commonly used computer architectures did not lend themselves well to virtualization. The Intel x86 architecture in particular, perhaps the most prevalent general-purpose computing architecture in use today, has several historically-grown architectural details that make it hard to virtualize. For instance, there exist several instructions which, despite being privileged operations, do not trap when called from non-privileged modes and can therefore not be emulated by a hypervisor. [31]

The need for efficient virtualization was finally addressed by major processor vendors Intel and AMD in the year 2006, implementing extensions for **hardware virtualization** and thereby bringing the concept to the mainstream [31]. These extensions supplement the instruction set by adding operations for *virtual machine entry* and *exit*, thereby providing a straightforward way of switching the execution context from the host to the guest machine and vice-versa. Figure 2.2 shows a high-level sequence of this process.

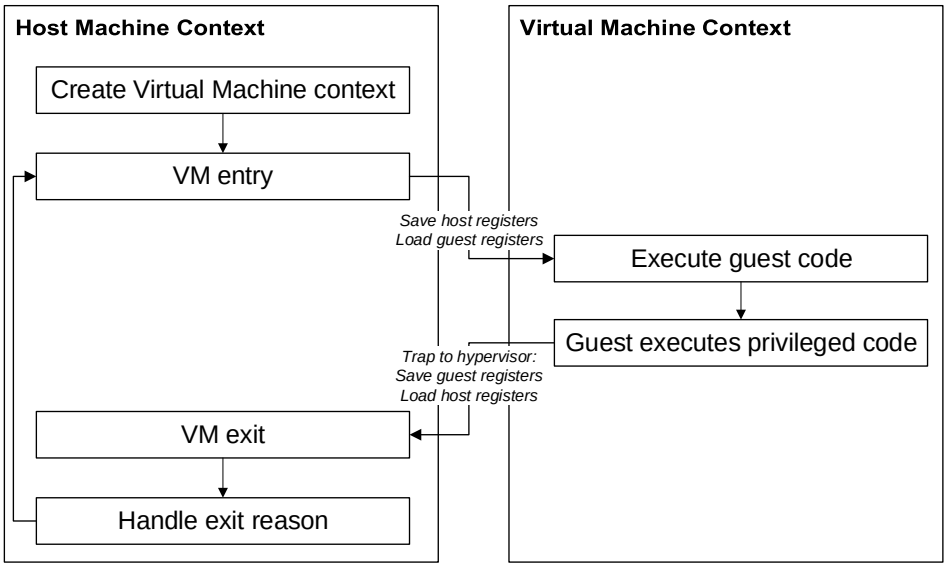


Figure 2.2: Schematical overview of hardware virtualization. A virtual machine context is first created and populated with information about the guest. The hypervisor can then switch to this context, allowing the guest to physically run on the CPU. Instructions that require emulation are trapped by the hardware, causing a virtual machine exit and switching back to the hypervisor.

Later additions to the virtualization extensions allowed for a streamlined handling of memory and I/O accesses, reducing the performance penalty incurred by page faults in the guest machine [26].

2.2 ARMv7 Architecture

While the x86 architecture, as one of the most widely used general-purpose computing platforms, has been at the center of attention in virtual machine research, the growth of the mobile computing market has drawn the interest in virtualization towards the ARM computing architecture. Devices implementing an ARM processor design had previously been low-power designs with small software stacks; the rapid proliferation of high performance ARM processors in smartphones and servers, however, has made the platform a viable target for virtualization.

The ARM architecture traces its roots back to a research project by Acorn in 1985. [30] Conceived as a reduced instruction set computing (RISC) architecture with a data bus width of 32 bits, it started as a low-complexity, low-power design

and has evolved from being a single processor implementation—the ARM1—to a whole family of architecture variants, as well as processor core designs available under license from ARM Holdings. The most recent 32-bit architecture, ARMv7, shall be the target architecture for this work. With mobile devices starting to hit the 4 GB memory limit imposed by 32-bit architectures, the successor platform ARMv8 with 64-bit support is gaining traction.

The ARM architecture generally follows the RISC principle of implementing a simple instruction set that can be executed at high processing speeds, as opposed to the complex instruction set computing (CISC) approach that for instance the x86 architecture has adopted over time.

The key features of the ARM architecture are:

- **Large uniform register file**: The 32-bit general purpose registers R0 to R14 can be used without restrictions, with R15 being the program counter. Certain registers have multiple banked copies between the various processor states. The x86 register file, by contrast, only encompasses 8 registers.
- **Load/Store architecture**: All accesses to memory occur only through explicit `LDR*`/`STR*` instructions, unlike the x86 instruction set which allows direct computations on memory contents.
- **Memory-Mapped I/O**: Access to devices is generally handled via memory-mapped regions using reads and writes. There are no equivalents for the `IN*`/`OUT*` instructions on x86.
- **Simple addressing modes**: The memory addresses used by instructions only depend on register contents and on the values encoded in the instruction word. Indexed addressing is supported, but there is no segmented memory model as there is on x86.
- **Conditional execution**: Most instructions can be made conditional, causing them to only be executed when a previous condition check has passed. Program flow can therefore in some cases be simplified by omitting explicit branch instructions.

While RISC architectures can usually be implemented more efficiently in silicon than CISC designs, the lack of complex operations typically reduces the density of application code, which is a concern particularly in the embedded devices market. To address this issue, ARM processors offer a so-called *Thumb* mode, in which instructions are limited to 16-bit width, supporting more compact code at the expense of certain functionality. The CPU can switch between the two modes at runtime using a "branch and exchange" instruction, allowing for mixed 32-bit and 16-bit code.

The basic ARMv7 architecture supports two different privilege levels: PL0 or *unprivileged execution*, and PL1, which grants access to restricted system functionality and is intended only for use by the operating system.

Implementing these two privilege levels, there are different processor modes that the CPU can be in at any given time:

- **User mode** (PL0), typically when running application code.
- **IRQ and FIQ modes** (PL1), triggered by an external interrupt request (IRQ) or fast interrupt request (FIQ), respectively.
- **Supervisor mode** (PL1), synchronously entered using the `SVC` (supervisor call) instruction from unprivileged modes and used for implementing system calls.
- **Abort and Undefined modes** (PL1), caused by page faults or undefined instructions.
- **System mode** (PL1), not normally entered by architectural events.
- **Monitor mode** (PL1), comparable to System Management Mode on x86. It is part of the security extensions and not relevant to our work.

The currently active processor mode is identified by a bitmask in the **Current Program Status Register** (CPSR). Each of these modes has a banked copy of the stack pointer (R13), the link register (R14) and of the CPSR from which it was entered, simplifying exception handling. Figure 2.3 gives a concise overview of the ARM core registers and their banking behavior. [11, B1.3]

The ARM architecture supports two kinds of asynchronous interrupt requests, IRQ and FIQ, each of which has its own associated processor mode at privilege level PL1. The FIQ mode has five more banked registers, allowing for faster interrupt handling without having to spill registers to system memory, but is otherwise used the same way as IRQ mode. Subsequently, this document will focus on the IRQ type, which is the primary type used by the Linux kernel.

Interrupt requests can be triggered by activating the IRQ line on the CPU, after which the processor saves the CPSR and the program counter and enters IRQ mode. The exception vector table is consulted to find the entry address of the exception handler.

Unlike other platforms such as x86, the ARM architecture itself does not have a notion of interrupt numbers: There is only a single interrupt line. To allow for multiple interrupts to be distinguished and prioritized, ARM processors include a **Generic Interrupt Controller** (GIC) peripheral which queues multiple interrupt events. The GIC register interface is memory-mapped into the processor's address

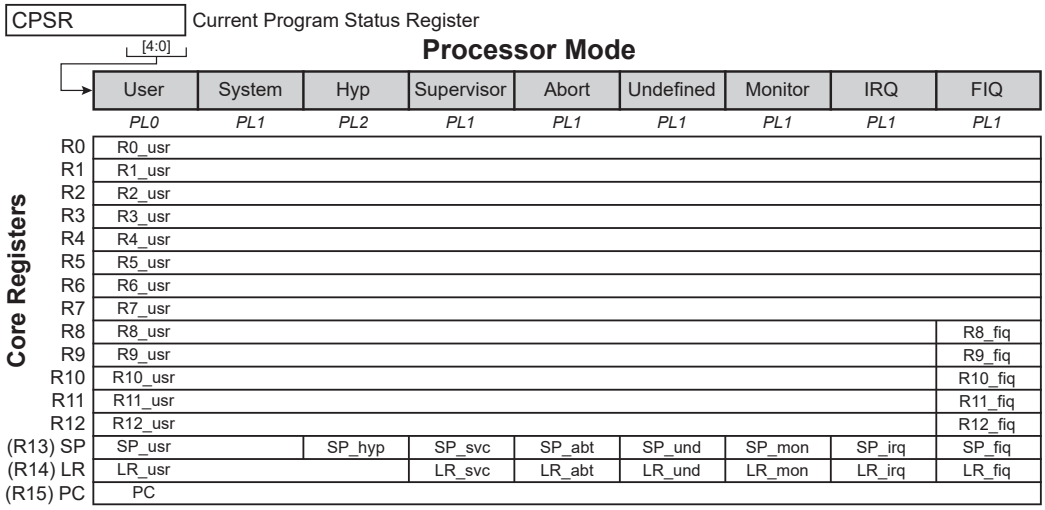


Figure 2.3: Overview of the ARM core registers, the processor modes and their associated privilege levels (PLx). The current processor mode is determined by the lowest 5 bits in the CPSR. Certain registers, such as the stack pointer and link registers, have multiple banked copies across the different processor modes.

space and upon receiving an IRQ, its **Interrupt Acknowledge Register** (IAR) is used to receive the interrupt number and to clear the pending interrupt. [10]

The concept of plug and play has not been popular on the ARM architecture. Instead, the devices available to a machine are typically configured using a device tree binary file (DTB), which contains the memory addresses and configuration information for each device. The device tree is typically passed to the operating system kernel by the machine's boot loader.

For debugging purposes, an ARM processor provides a component called the debug architecture, which offers support for breakpoints and watchpoints and which is accessed through the coprocessor 14 interface.

2.2.1 ARM Virtualization Extensions

A platform implementing the ARMv7 architecture can optionally support the virtualization extensions which, like their counterparts in the x86 implementations, enable support for high-performance virtual machines. To that end, the system architecture is extended by a new, higher privilege level PL2, as well as a new processor mode implementing it, called **Hyp mode**, which has its own virtual address space isolated from the PL0 and PL1 modes. [11, B1.7]

While a basic implementation of ARMv7 will start the CPU in Service mode, a system supporting the virtualization extensions boots up in Hyp mode instead,

which allows the operating system to access the hypervisor control registers only available in PL2. The operating system will then switch to Service mode and continue the regular boot process, later returning to Hyp mode using the `HVC` (hypervisor call) instruction as necessary.

Using the Hyp Configuration Register (HCR), the hypervisor component in the host operating system can trap certain system functions, such as accesses to the coprocessor interface used for system management purposes, to Hyp mode instead, where they can then be emulated or dispatched to the virtual machine management software. Additionally, exceptions occurring while executing a virtual machine can be routed either to their corresponding default processor modes, where they are handled inside the guest machine, or also taken to Hyp mode.

In order to inject interrupts into the guest virtual machine, virtual IRQs and FIQs can be triggered by setting the corresponding bits in the HCR, causing an exception to be delivered once execution of the virtual machine resumes.

On entry to the virtual machine, the hypervisor changes the address translation regime to switch to the correct address space of the guest, sets the return address and target processor mode and uses the `ERET` (exception return) instruction to switch from Hyp mode into the virtual machine. Once executing in PL0 or PL1, the only way of returning to PL2, and thus to the host operating system, is by invoking `HVC` or through an exception delivered to Hyp mode.

2.3 Kernel-based Virtual Machine (KVM)

The aforementioned hardware virtualization extensions enable a hypervisor to effecticely execute virtual machines; however, the instructions provided to do so are privileged operations and can therefore not be used by user code directly. In order to provide virtual machine hosting software with a way of accessing these facilities, the operating system needs to expose these mechanisms through an interface.

In the Linux operating system, this is done through the so-called **Kernel-based Virtual Machine** (KVM) subsystem, which was introduced in kernel version 2.6.20 in 2007. [2] It provides a unified system call based interface to the virtualization extensions offered by several platforms, including Intel x86 and ARM.

KVM follows the UNIX principle of "Everything is a file" by providing a device `/dev/kvm`, which can be accessed by user mode applications that have sufficient file system permissions to open it. [8] There are several classes of operations available using the `ioctl(2)` system call:

- **System ioctls** for managing the KVM subsystem as a whole and, in particular, to create new virtual machine instances.

- **VM ioctls** for managing an individual virtual machine, e.g. to make user space memory from an application available in the guest machine's address space and to create virtual CPUs.

- **Virtual CPU (vcpu) ioctls** for managing the execution context of one of the virtual CPUs, and to schedule it for physical execution.

An application making use of KVM will typically first create a virtual machine instance using the `KVM_CREATE_VM` call. Memory for the guest system can be allocated in user space, e.g., using `malloc(3)`, and is then made visible inside the guest memory space using `KVM_SET_USER_MEMORY_REGION`.

Actual execution of the virtual machine takes place when the application invokes the `KVM_RUN` ioctl: The KVM subsystem switches the host system's execution context to the VM, causing the CPU to natively run the guest machine code. Whenever an architectural event is trapped and must be emulated by the application, the ioctl returns and the application can read the returned `kvm_run` structure to find out the exit reason. One such reason, for example, is `KVM_EXIT_MMIO`, which signifies that the guest machine is accessing a memory address that requires special handling.

Once the application has completed emulation of a trapped event, it executes `KVM_RUN` again to resume the virtual machine.

Since Linux 2.6.22, the basic KVM application programming interface (API) has been finalized; new features can only be added through extensions to this interface, whose support can be queried at runtime.

KVM was initially only available on the x86 architecture, due to the lack of virtualization support in pre-ARMv7 architectures, but has been supported on the ARMv7 architecture since Linux kernel version 3.9. [18]

2.4 QEMU

While KVM provides the facilities that enable high-performance virtualization, it does not by itself emulate the virtual devices provided to the machine. Instead, any access to such devices causes a virtual machine exit, returning control to a user mode application which then performs emulation.

One such user mode application making use of KVM is QEMU, a virtual machine simulator originally written by Fabrice Bellard which supports the emulation of a wide range of architectures including x86 and ARM. It can be used for *full-system* emulation in user space, but also as a translator to execute *user-land* applications from a foreign architecture. This work concerns itself only with full-system emulation of the ARM architecture.

Besides emulating a CPU core, QEMU supports a wide range of virtual devices that can be made available inside the virtual machine. These include an universal asynchronous receiver transmitter (UART) for serial transmission, keyboard and mouse input, graphical displays, network adapters, block devices such as hard drives, as well as basic system hardware such as timers and interrupt controllers. Depending on the kind of device and on the simulated architecture, they can be accessed using I/O instructions or are addressed through memory-mapped I/O.

When running a virtual machine with QEMU, code inside the guest system is by default executed using the **Tiny Code Generator** (TCG). Executed instructions are first disassembled into an intermediate, architecture-independent form to model its effects on the virtual machine state. Multiple architectural instructions are grouped into translation blocks (TBs) until encountering a branch or I/O instruction, making a TB the smallest unit of execution. A second code generation step then performs just-in-time compilation to produce platform-specific machine code that can be executed natively on the host system. Figure 2.1 gives an example of this translation process for BIOS code on x86.

Grouping the executed instructions into translation blocks has the advantage of reducing execution overhead, as it allows the host CPU's branch predictor to operate on larger blocks of code at once, as well as improving cache locality. The boundaries of a translation block are chosen in a way that instructions which change the CPU state, such as branch operations, always occur at the end of a translation block.

For further performance improvements, multiple blocks can be chained together if the next block to be executed is known and translated already: The loop exit instruction of the translation block is patched to jump directly into the next TB, avoiding an exit to the CPU loop in many cases. Since generating a translation block can be expensive, it is stored in a cache for later re-use.

The TCG approach allows for great flexibility in combining host and guest architectures, as is evidenced by the large range of platforms actively supported by QEMU. However, it cannot solve the fundamental problem of emulation: While binary translation speeds up execution compared to simple interpretation of code, it is still vastly slower than native execution on a suitable platform.

Looking for a way to improve the execution speed, an extension to the Linux kernel was developed, called KQEMU. [1] This extension allowed QEMU to execute some guest code natively on the host CPU. With the introduction of KVM into the stable Linux kernel, KQEMU was obsoleted and is now no longer supported. QEMU can now instead use KVM on systems that support it, bypassing the Tiny Code Generator entirely.

2.4.1 Simulation Process

After launching QEMU in full-system emulation mode through the `qemu-system--arm` binary, the general program logic is as follows:

1. QEMU determines which machine type is to be emulated (e.g., the ARM Versatile Express platform) and invokes the corresponding initialization routine.

2. The simulated peripherals are initialized and registered in an internal device hierarchy. These peripherals include system memory, the interrupt controller, UARTs for serial input and output, as well as storage devices.

3. A virtual CPU (e.g., a Cortex-A15) is created and reset.

4. The operating system kernel that has been passed to the `-kernel` command line argument—which can be an executable in the ELF format or a Linux self-extracting bzImage—is loaded into the virtual machine's system memory, and the program counter is updated to point to the entry address. When loading a Linux kernel, a small bootloader is used to pass the kernel command line arguments along.

5. The configured accelerator is initialized, which is either the Tiny Code Generator or optionally, when executing a guest system with the same CPU architecture as the host, the Kernel-based Virtual Machine. In the latter case, a virtual machine context is created in the kernel, and the guest machine's RAM block allocated by QEMU is mapped into the VM address space.

6. A CPU worker thread is started, which begins virtual CPU execution.

   - **When using TCG:**
     (a) The CPU state is checked for a pending interrupt request. If such a request is pending and interrupts are not masked, the CPU state is updated to execute the corresponding exception handler.
     (b) The program counter is used to look up a compiled translation block for the currently executed address.
     (c) If no such translation block can be found, the current instruction is disassembled from ARM machine code to QEMU's intermediate code and subsequently compiled to native host machine code. Instructions requiring special handling, such as memory accesses, are translated into calls to QEMU helper functions.
     (d) The resulting translation block is executed and cached for future use.

- **When using KVM:**
  (a) The `KVM_RUN` ioctl is used to transfer execution to the virtual machine context.
  (b) The Linux kernel KVM subsystem performs the context switch to execute the guest code natively on the CPU, and switches back to the host once a trapped event requires emulation.
  (c) The `KVM_RUN` ioctl returns and QEMU reads the returned structure's `exit_reason` field to determine the cause.
  (d) If emulation is required, e.g., for memory-mapped I/O accesses, the appropriate QEMU helper function is called.

7. The CPU worker thread continues execution in a loop, unless interrupted by QEMU's I/O thread or the simulation is terminated.

2.4.2 Interrupt Handling

The Generic Interrupt Controller (GIC) is responsible for queuing and prioritizing interrupts from multiple sources and dispatching the pending interrupt with the highest priority to the virtual CPU. Both QEMU and an ARM processor supporting the virtualization extensions contain an implementation of a virtual GIC (vGIC), though only one of them is used at a time.

When using TCG, a virtual GIC is instantiated and its configuration registers are mapped into the address space of the virtual machine. Memory-mapped I/O requests to the GIC by the guest operating system, e.g., to query the currently pending interrupt, are dispatched to QEMU's vGIC functions using memory listeners. The vGIC manages the simulated CPU's `interrupt_request` line, which is checked in the CPU loop between the execution of translation blocks to enter IRQ mode.

If KVM is enabled instead, there are two options for where the interface between QEMU's virtual devices and the vCPU can be located, as shown in Figure 2.4.

1. The physical CPU's vGIC, a so-called **in-kernel IRQ chip**, is used. QEMU does not instantiate a vGIC, and interrupt numbers are used as parameters to the `KVM_IRQ_LINE` ioctl to set the state of each external interrupt line. This is the default option when KVM is used.

2. QEMU's internal vGIC is used. The virtual peripherals report their interrupt outputs to the vGIC, as they do in the TCG case. The vGIC performs all multiplexing and filtering in user space and communicates only the status of the single IRQ line to KVM. This behavior is triggered by specifying the CPU option `kernel_irqchip=off`.

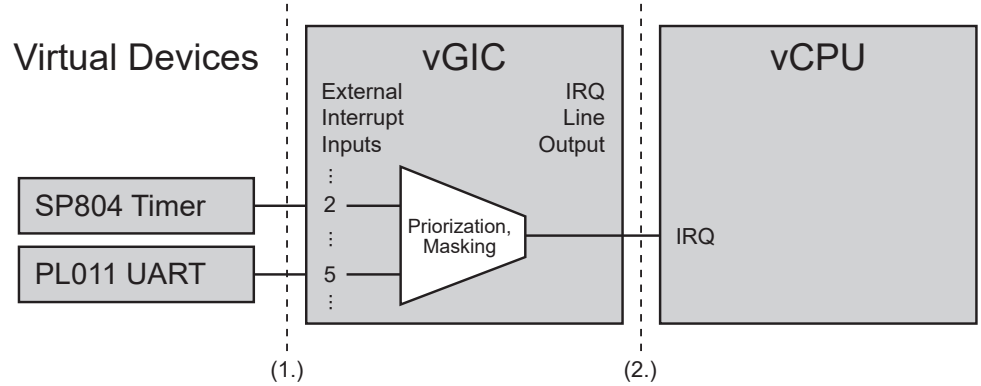


Figure 2.4: Interaction between virtual devices, the vGIC and the vCPU. The vGIC has multiple numbered interrupt inputs and prioritizes them, delivering one IRQ request to the vCPU at a time. The dashed lines denote the two possible interfaces across which interrupts are signalled between QEMU and KVM: (1.) when using the in-kernel IRQ chip, (2.) when using QEMU's internal vGIC implementation.

2.5 Record and Replay

To find and reproduce bugs in a computer program, a technique called *cyclic debugging* is typically used: The program in question is run multiple times, examining its state at certain points and stepping through functions in varying detail to narrow down the possible locations of the error. However, if the software system is sufficiently complex, it may not be clear at first in which part the defect is located—in fact, it may not even be evident if it is an error at the application or at the operating system level.

Furthermore, attaching a debugger to the target program to manually step through execution may mask the error (leading to so-called *Heisenbugs* which disappear when they are observed) or, worse, prevent the system from working at all because the user interface or networking is unresponsive.

To overcome the limits of cyclic debugging, the idea of **record and replay** was conceived. Rather than inspecting the system at the same time as it is running, the whole system's operation is recorded so that an exact sequence of executed instructions can later be replayed and analyzed multiple times.

While a full trace recording every executed instruction would be prohibitive in size and runtime slowdown, practical implementations make use of the fact that computing architectures are mostly deterministic: Given the same inputs, a system behaves identically across several runs. It is therefore sufficient to record only few non-deterministic events and feed them back into a running system during replay.

These sources of non-determinism include:

- Data read from peripheral devices (such as network interfaces)
- System timers (e.g., the `RDTSC` instruction on the x86 architecture)
- Interleaving of multi-processor events
- Timing of interrupt requests

While non-deterministic behavior of existing computer systems cannot be easily eliminated, the results of such non-deterministic operations can be recorded and then, during replay, fed back into the simulated system.

In order to achieve determinism without explicit hardware logging mechanisms, record and replay has commonly been implemented using virtual machines. Initially, these virtual machines operated at the the operating system interface level. ReVirt, for example, is based on a system called UMLinux, which uses para-virtualization to translate a guest operating system's hardware accesses into usermode system calls. [22] As computers became more powerful, it became feasible to use full-system emulators like Bochs to implement record and replay, as used by ExecRecorder. [19]

An implementation of deterministic record and replay by Dovgalyuk et al., which has been included in QEMU, does not directly record actions performed by the virtual machine (e.g. memory reads), but records the inputs to the emulation software itself (e.g. keyboard and mouse events), using instruction counting to replay those inputs at the correct time and to adjust the simulated clocks appropriately during replay. To do so, it requires the use of the Tiny Code Generator; hardware acceleration with KVM is not supported. [21]

The advent of virtualization extensions gave rise to high performance virtual machines, permitting record and replay at moderate costs in speed, as demonstrated in ReTrace [38].

Decoupling the normal operation of a system from the debugging phase has several advantages: The regular functioning of a server or user-interfacing device is not disturbed by breakpoints or analysis delays, preserving interactivity and keeping the system in a usable state. Furthermore, unlike cyclic debugging, the scope of analysis can be determined later on: When performing a post-mortem analysis of an attack on a server, a comprehensive full-system replay can be used to perform multiple analysis passes at different levels (operating system, library or user code) if it is not clear which attack vector was used. The Crosscut replay system [17] can use such a full-system recording to generate multiple *slices* at different abstraction levels.

ReTrace [38] further separates the recording phase from replaying by introducing **heterogeneous** record-and-replay: The execution of a virtual machine is recorded on one system—the VMware hypervisor—and replayed on another—the simulator. This decoupling allows for the replay or analysis to take place on a different physical machine, relieving the original system of analysis overhead. This idea is further expanded on in the Aftersight system [16], even permitting real-time threat analysis and decisions for best effort security.

Aftersight uses a modified QEMU full-system emulator, which unlike the previous schemes allows for heterogeneous record-and-replay across different computing architectures. This approach combines the advantages of a hypervisor-based, fast recording (e.g., using the VMware hypervisor or KVM-accelerated virtualization software) with a replay in a full-system emulator that is not constrained by the recorded architecture.

The two implementations of such a heterogeneous record-and-replay scheme are Aftersight, whose underlying technology was removed from VMware Workstation in 2011, and V2E [39]. Neither of these systems are available publicly in the form of source code. Their work has focused on the x86 architecture, which is by far the most popular platform in the virtualization and software security domain.

Chapter 3

Analysis

There have been various efforts to implement record and replay for the x86 architecture, such as ExecRecorder [19] or more recently PANDA [20]. They permit recording the execution of a simulated virtual machine and later replaying it to analyze various aspects of the executed software or to extract information.

A compelling use case is to employ record and replay for malware analysis and intrusion detection. However, for this purpose, virtual machine simulators are not always applicable: Servers and mobile devices, in particular, depend on a high degree of interactivity. Running them in a simulator is not always possible in production. Some strains of malware, for example, use the system's performance or inaccuracies in the emulation to detect if they are running inside a simulated virtual machine, in that case refusing to become active to prevent analysis [24].

To overcome the performance and accuracy issues, hardware extensions for virtualization can be leveraged to enable recording with acceptable overhead. These recordings can then be fed into a virtual machine simulator, which can be done on a different platform than the original system that has been recorded, giving this approach the name *heterogeneous* record and replay. Aftersight [16] and V2E [39] are implementations of this technique. We have chosen the heterogeneous approach to record and replay as the basis for our work.

The record and replay projects so far have focused on the x86 platform, which is understandable given the popularity of the architecture. However, as evidenced by many vulnerabilities discovered in mobile devices, such as Android smartphones, the ARM platform is no longer protected under "security by obscurity", but has become a lucrative target in the IT security business. Gaming company Nintendo, whose mobile gaming platforms since the GameBoy Advance have been using ARM microprocessor designs, has begun to offer a bug bounty program, offering up to $20,000 in rewards for exploits that can compromise the system's security [7], showing that there is a growing demand for ARM security research.

It is therefore of interest to consider if the concept of heterogeneous record and replay can be applied to the ARM architecture, in order to aid in debugging a system. This chapter discusses the requirements for deterministic replay and the challenges that must be solved to enable a successful full-system replaying.

This work considers the problem of deterministic full-system record and replay on the ARMv7 architecture; a task for which—to the best of our knowledge—no implementations exist yet. While the successor architecture ARMv8 is already available, featuring 64-bit support, it is not currently as widespread as ARMv7, which is still used by many mobile devices and development tools currently on the market. [1]

We focus on recording a single-core virtual machine, because at the current stage—when evaluating the feasibility of record and replay on the ARM architecture in general—multi-core support would add little value and complicate the design.

3.1 Event types

The goal of record and replay is to capture as much information during the execution of a system as necessary to deterministically replay the behavior later. As the behavior of a system is defined by the stream of instructions that it executes, one may separate the architectural instructions into two classes, as defined by Bressoud and Schneider [14]:

- **Ordinary instructions**, whose behavior is determined only by inputs from within the system itself, and which can therefore be deterministically replayed given the current state of the system.
- **Environmental instructions**, which are influenced by a source outside of the system and can therefore not be deterministically replayed without additional information.

For the purpose of record and replay, it is sufficient to record only the environmental instructions and their inputs; ordinary instructions are deterministic already, and can be dropped from the recording, provided that the replaying software implements them in exactly the same way. This omission is in contrast to execution tracing, which aims to save a complete log of all executed instructions.

While environmental instructions are non-deterministic from the system CPU's point of view, they typically communicate with external devices, which follow

[1] Popular development boards using the ARMv7 architecture include the Raspberry Pi 2 (Cortex-A7, released in 2015) and the Odroid-XU4 (Cortex-A7 and A15, released in 2015).

deterministic behavior individually. By including an appropriate model of those devices, their communication with the CPU does not need to be recorded because it can be inferred from the simulation. This approach is used in Cooperative ReVirt [12], which includes the communication partners in the record and replay scheme to reduce the network traffic that needs to be recorded.

Similarly, QEMU's existing record and replay scheme based on the *icount* system does not record the results of environmental instructions, but the non-deterministic inputs to each simulated device [21]. Since the record and replay scheme is homogenous, with both the recorded and the replayed system running in the same virtual machine software, it is feasible to ensure that the virtual devices behave identically in both modes.

When recording on actual hardware (i.e., through KVM), however, developing an exact model of a peripheral is a difficult task and must be done for each of the devices individually. In developing ExecRecorder, de Oliveira et al. considered including the Programmable Interrupt Timer (PIT) in the recording system, but noted that inaccuracies of the virtual machine's software implementation, as well as noise in the system's crystal-controlled oscillator, made the PIT's exact timing behavior unpredictable during replay [19].

It is therefore easier to capture non-deterministic inputs into the virtual machine not for each peripheral individually, but directly at the interface to the CPU. Figure 3.1 illustrates the two approaches.

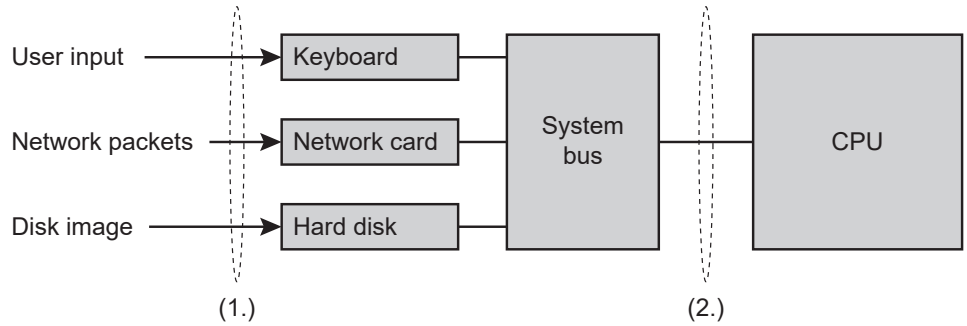


Figure 3.1: Possible interfaces across which non-deterministic input can be recorded. (1.) For each device individually, reducing the size of the recording, but requiring modifications to the virtual devices; (2.) At the CPU interface, increasing the number of recorded events, but simplifying the recording and replaying processes.

Recording at the CPU interface level leads to a higher volume of data that must be stored, yet greatly simplifies record and replay because the virtual devices need not be individually considered—their responses are part of the replay log, and thus they do not need to be emulated. Furthermore, the replay becomes independent

of a particular implementation of virtual devices: Replaying a recording that has been performed at the device level requires the implementations of the virtual devices to match *exactly*, since any change to them—even a bug fix—might lead to a different execution during replay.

Several classes of non-deterministic events have been identified in previous projects [19] [23]:

- **Input events**, occurring in the form of data that is read from a peripheral device into a processor register or main memory.
- **Hardware interrupts**, which are generated asynchronously to the CPU's execution by external peripheral devices.
- **DMA transfers**, taking a special role in that they typically trigger a hardware interrupt and also write data to main memory.

3.2 Timing

In the previous section, events in a system's execution have been classified according to their deterministic properties and their source. In order to accurately replay them, it is not sufficient only that non-deterministic values are recorded, but also that they are played back at the correct time. For this purpose, Dunlap et al. [23] separate replay events into two groups:

- **Synchronous events**, which occur through the execution of an instruction and whose timing is therefore determined by their position in the instruction stream. This includes, for example, reads from system memory.
- **Asynchronous events**, which are triggered by sources outside of the instruction stream, such as hardware interrupts, and whose precise location in the instruction stream can therefore not be inferred.

When replaying synchronous events, their precise timing is unimportant: Being anchored in the instruction stream, it is only their relative ordering that must be preserved when reading the values from the replay log.

The situation is different for asynchronous events: The virtual machine software that replays a recording must interrupt execution and deliver the event at the precise time at which it occurred during the original recording. If the timing is not correct, interrupt handlers, for instance, may run too early or too late, altering the system's behavior.

In order to find the correct point at which to inject an asynchronous event, information must be stored in the replay log by which that point can be identified

in the instruction stream. Yan et al. have used the term **landmark** for this concept in their V2E replay system [39].

The exact contents that make up a landmark vary by project. In V2E, a snapshot of the x86 CPU state is used, consisting of the register contents and the flags register. When used as the only identifier, this can lead to a problem if the landmark happens to match at several points during execution, causing a "false positive" match and leading to the asynchronous event being delivered too early. The authors of V2E acknowledge this issue, but found that, during their experiments, it did not matter in practice because most events were of the synchronous kind [39].

Besides the current CPU state, modern processors typically implement some variant of a hardware counter for architectural events, which can be used to approximate a position in the instruction stream. Using such a counter, it is possible to reduce the chance of a false positive landmark match. Echo, a record and replay project for multi-threaded applications, uses the performance counters of the Pentium 4 processor to count the number of retired instructions for use in a landmark [28]. Other projects like ReVirt [23] use the branch counter for this purpose.

3.3 DMA

In order to enable high-performance data transfers, modern computing systems support the direct memory access (DMA) model, in which the CPU does not need to poll a device for new data, but can instruct it to asynchronously write the result of an operation directly to memory. The CPU can then continue to perform other work and is notified of the transfer's completion through a DMA interrupt. The DMA model is used, for instance, by hard disk drives and network adapters to deliver bulk data read from the disk and incoming network packets.

For the purpose of record and replay, the DMA model poses a challenge in that a hardware DMA controller's behavior cannot easily be recorded, yet can modify memory contents at any time. Even if the device performing DMA is a virtual device under control by the virtual machine software, it may write to guest memory asynchronously to the CPU execution loop. In order to deterministically replay the execution of a system, these operations need to be replayed at the correct time. While regular memory writes issued by the CPU, being ordinary instructions, can be omitted from the recording because they can be inferred from the current CPU state, data written to memory by the DMA controller must be recorded.

Dunlap et al [23] assume, in order to make record and replay involving DMA feasible, that any DMA controller does not become active by itself, but that DMA requests follow a transactional model: The CPU issues a DMA request, specifying a destination for the data to be read, and only reads from that address once the DMA controller signals completion of the request. This is a reasonable assump-

tion to make because the state of the destination data is not well-defined while a request is in progress, and properly written device drivers will only attempt to access the data once the completion interrupt has been received.

To allow for deterministic replay, the recording system must therefore be aware of any DMA requests issued by the guest system, which is possible if all DMA controllers are emulated by the virtual machine software. Furthermore, it needs to record not only the landmark when a request has been issued, but also the content that has been written to memory. Under this model, there are no incremental writes to the destination address, but all of the DMA result's content is written at once.

3.4 Applying Record and Replay to ARM

Having identified several classes of events relevant to record and replay, we now look at how its concepts can be mapped to the ARM architecture. As outlined in the ARM Architecture Reference Manual, the instruction set is composed of the following groups [11]:

- **Branch instructions** to change execution flow and to switch between instruction sets (ARM/Thumb).
- **Data-processing instructions** performing arithmetic operations on register contents only.
- **Status register access instructions** to access the banked versions of the CPSR and, if the virtualization extensions are supported, to move data between banked general-purpose registers across different modes.
- **Load/store instructions** to transfer the contents of general purpose registers to and from memory.
- **Load/store multiple instructions** for reading and storing sets of multiple registers at once.
- **Miscellaneous instructions** including debugging, memory barrier and waiting instructions.
- **Exception-generating and exception-handling instructions** to perform calls into privileged processor modes and to return from them.
- **Coprocessor instructions** for communicating with platform-specific coprocessors, including certain system management registers.

- **Advanced SIMD instructions** implemented in the ARM NEON extensions, supporting vector operations.
- **Floating-point instructions** implemented in the VFP extensions, supporting floating-point arithmetic.

3.4.1 Memory-mapped Input/Output

Among the instruction set groups, the instructions involving memory accesses deserve special consideration. Since the ARM architecture has no explicit `IN`/`OUT` instructions to do port input/output as on x86, access to peripheral devices generally happens by way of memory-mapped input/output (MMIO). For this reason, all instructions reading from memory must be considered environmental instructions.

On the x86 architecture, it is not immediately obvious which instructions are environmental instructions because many of them can operate on memory contents directly. Fortunately, ARM is a load-store architecture, on which operations reading from system memory are easily identified:

- **Load Register**: Reading a 32-bit word from a memory address into a register (`LDR`). Also supported are halfword (16-bit, `LDRH`) or byte (8-bit, `LDRB`) accesses, which applies to the instructions below as well.
- **Load Register Exclusive**: Memory read with support for synchronization (`LDREX`). After using an exclusive read, a subsequent exclusive write operation (`STREX`) succeeds only if no other processor in the system has written to that address in the meantime.
- **Load Register Unprivileged**: A read operation that is carried out as if it was executing in User mode (`LDRT`).
- **Load Multiple**: Read values from consecutive memory addresses into multiple registers at once (`LDM`). Pre- and post-incrementing the source address register and writeback of the final address are supported. This group also includes the `POP` instruction, which implicitly accesses the stack.

We do not consider the write operations (`STR*`) because they are ordinary instructions, requiring no further recorded data to make them deterministic.

In order to trap memory accesses, the hypervisor can simply omit any page table entries for the regions that are of interest for the recording, leading to a virtual machine exit on every access. Regular accesses to system memory (RAM) do not need to be trapped because their results are deterministic; after the first access to a RAM page, an appropriate page mapping is created and the hypervisor no longer needs to be involved when the guest accesses that page.

3.4.2 Coprocessor Access

Besides accesses to system memory, the ARM architecture also supports two coprocessor instructions through which data can be read from one of up to 16 coprocessors in the system:

- **Move to ARM core register from Coprocessor**: The `MRC` instruction issues a command to a given coprocessor, returning the result in a general-purpose register of the CPU.
- **Move to two ARM core registers from Coprocessor**: Like `MRC`, `MRRC` issues a coprocessor command but returns a 64-bit result in two general purpose registers.

As with the memory accesses, we ignore the ordinary coprocessor write instructions (`MCR`/`MCRR`).

The coprocessor interface is typically used for system configuration purposes. Coprocessor 15 (CP15) encompasses platform identification, cache management, virtual memory and performance counting registers. It is also used in the event of instruction and data aborts to provide fault information. Figure 3.2 shows an overview of the subsystems that can be configured through CP15.

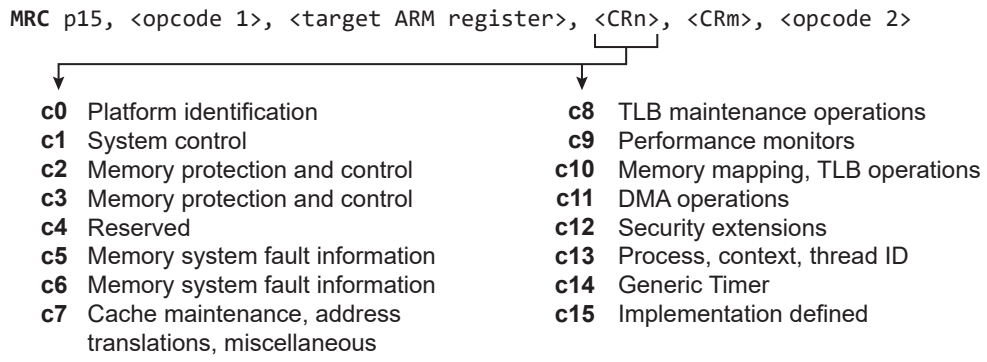


Figure 3.2: Overview of the coprocessor 15 registers, grouped by their subsystem (CRn). Using the Opc1, CRm and Opc2 parameters to the `MRC`/`MCR` instructions, a particular control register can be selected.

If a system implements the optional Generic Timer extension, a system timer can be accessed through CP15, typically operating at 1-50 MHz. This timer fulfills a similar role to the `RDTSC` instruction on x86. The architecture also allows for virtualizing the physical counter value by subtracting a configurable offset, resulting in a virtual counter value register, CNTVCT [11, B4.1.34], intended to account for the time that a virtual machine has spent in a paused state.

Coprocessor 14 (CP14) is used to access the debugging functions of the CPU and controls various aspects of the Thumb instruction set. Coprocessors 10 and 11 are used by the floating point and SIMD subsystems. Other coprocessor numbers are reserved for use either by ARM or by implementors of the architecture.

For the purpose of record and replay, it must be possible to record the result of any coprocessor read operation. Indeed, the virtualization extensions provide support for trapping some of the coprocessor accesses to CP14 and CP15 to Hyp mode. The trapping mechanism for CP15 is configured through the HSTR register [11, B4.1.73] using one bit for each of the subsystems affected by CP15, with the notable exception of the Generic Timer extension: Access to the Generic Timer registers is specifically excluded from any trap mechanism.

This restriction poses a problem for deterministic replay because the Generic Timer introduces non-determinism into the system, and recording the values read from the counter is essential to accurately replay execution. Since reading the CNTVCT virtual counter register cannot be trapped, it is not possible to deterministically replay arbitrary guest software because it cannot be ruled out that it might access this register.

It is therefore necessary to restrict the guest system by making the following assumptions:

1. The guest operating system does not use the Generic Timer.

2. The guest operating system does not allow User mode (PL0) access to the CNTVCT register.

The Generic Timer is an optional extension to the ARM architecture. As such, software cannot rely on its availability. The Linux operating system runs fine on platforms without the extension; in fact, certain platforms like the Samsung Exynos 5422 used in our work do ship with a Generic Timer implementation, but with support for it disabled in Linux' device tree database.

While it is possible to configure Linux not to use the Generic Timer extension, it is not possible for the hypervisor to completely disable it when running guest virtual machines.[2] Therefore, one must trust the guest operating not to make use of the Generic Timer, and thus record and replay cannot be used with arbitrary (untrusted) guest operating systems.

In order to make sure that user software running in the virtual machine does not use the Generic Timer to introduce non-determinism—either accidentally or deliberately to thwart analysis—access to CNTVCT must be disabled from User mode, which is the default setting after reset.

[2]Accesses to the CNTPCT physical counter value register can be disabled using the CNTHCTL.PL1PCTEN bit, but there is no such setting for the virtual counter [11, B8.1.3].

Accesses to coprocessor 14, which provides control registers for the debug and Thumb instruction set subsystems, can be trapped by setting the appropriate bits in the HDCR register [11, B4.1.66].

3.4.3 Interrupts

The ARM architecture supports two different kinds of external asynchronous interrupts: IRQ requests and FIQ requests, differing only in the way that the banked general purpose registers are handled. Once such a request is received by the CPU, and interrupts are not currently masked through the CPSR, the processor generates an appropriate exception and enters the corresponding processor mode (IRQ mode or FIQ mode, respectively).

When in the process of executing a virtual machine, interrupts delivered to the host machine are generally not intended for the currently executing guest operating system, but must be handled by the host operating system kernel. For this reason, settings in the HCR register [11, B4.1.65] allow for IRQs and FIQs to be taken to Hyp mode instead of being delivered to the guest machine. In KVM, this mechanism is used to properly exit the virtual machine, then switch to Service mode with interrupts enabled, causing the interrupt to be delivered again while in the host machine context.

There are two possible ways in which a hypervisor can inject interrupt requests into a virtual machine. Both are illustrated in Figure 3.3 for the case of IRQ requests.

1. Using the virtual Generic Interrupt Controller (vGIC) provided by the virtualization extensions, populating its list registers with interrupt entries [10, 5.3.8]. Each entry in the list registers corresponds to one virtual interrupt and contains the associated interrupt number and priority. Like a physical GIC implementation, the vGIC performs interrupt masking and priorization, and the guest operating system uses memory-mapped I/O to acknowledge the interrupt requests. Interrupts are signalled directly to the vCPU.

2. Using the virtual IRQ and FIQ bits in the HCR register. Once the corresponding bit has been set, an interrupt request will be generated upon entering the virtual machine (unless disabled in the CPSR). In order to acknowledge the interrupt and to get the interrupt number, the guest operating system accesses a GIC that is simulated by the virtual machine software.

In order to perform deterministic record and replay, the timing of interrupt requests must be precisely recorded. Since there are no physical interrupt sources delivered directly to the virtual machine, but all interrupts are of the virtual kind, this appears to be easily done.

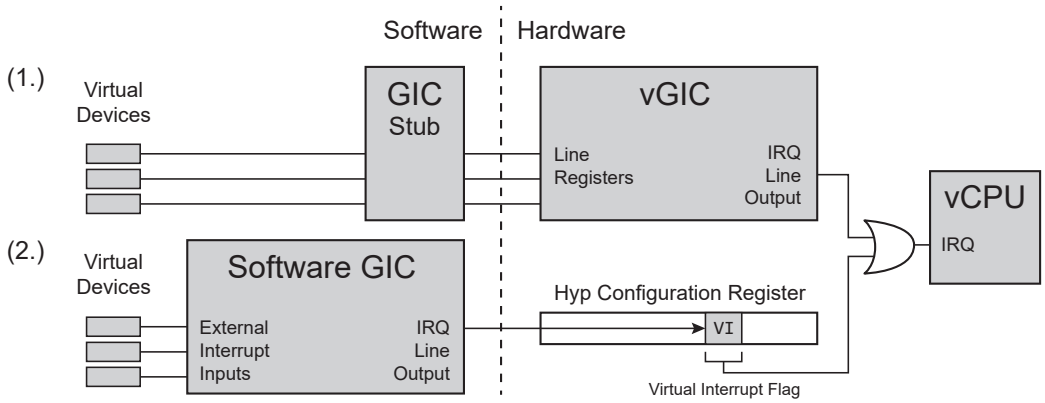


Figure 3.3: Two possible ways of handling virtual interrupts: (1.) Using the vGIC provided by the virtualization extensions, which handles interrupt priorization and masking; (2.) using a software-emulated GIC which sets the virtual interrupt flag directly.

The actual delivery of virtual interrupt requests, however, cannot be trapped by the hypervisor. Virtual interrupts are always delivered directly to the virtual machine at PL0 or PL1, never to PL2 [11, B1.9.11]. Since a vGIC, if used, can trigger virtual interrupts at any time without the hypervisor being aware of it, the recording of those events is not possible.

For record and replay to work, it is therefore required to trigger interrupts only through the HCR.VI and HCR.VF flags, and to record any changes to these registers together with the current landmark. A vGIC must not be used; the virtual machine software must provide a software implementation of a GIC.

While a recording scheme observing changes to the interrupt lines can only record when an interrupt request is *delivered* to the virtual machine, it cannot accurately tell when the interrupt request is *serviced* by the vCPU, as it might have interrupts disabled through the CPSR or another exception might take priority. For the replay, this does not make a difference because the virtual CPU will react to the interrupt requests in the same way.

It is worth noting that not only the act of raising the interrupt line must be recorded, but also when that interrupt request has been cleared again; otherwise, the virtual machine might continue to receive extraneous interrupts.

Once an interrupt request has been delivered to the virtual machine, and an exception has been taken into the corresponding IRQ or FIQ mode, the guest operating system queries the GIC's interrupt acknowledge register (IAR), which provides the interrupt number that has triggered the request. At the same time, its state is changed from *pending* to *active*, meaning that interrupt processing has begun, but is not yet completed. Completion of the interrupt servicing is indicated

by a write to the end of interrupt register (EOIR). Following completion, the GIC will either deliver the next interrupt, or lower the IRQ line to indicate that no more requests are pending.

All communication between the guest operating system and the GIC occurs through a memory-mapped I/O interface, which means that during replay, no special consideration needs to be taken for the emulation of the GIC: Recording MMIO read operations is sufficient.

On replay systems targeting the x86 architecture, a particular instruction set detail complicates interrupt handling: Certain instructions can be augmented with the `REP` prefix, causing a single instruction to repeat until the count register ECX reaches zero. Interrupts recorded in the course of such an instruction must correctly take into account that the instruction has completed only partially [29]. Fortunately, on the ARM architecture, no such repeated instructions exist.

3.4.4 Performance Counters

For asynchronous events, additional timing information must be recorded in the form of a landmark. While landmarks consisting of only the current CPU state may work under certain circumstances, performance counters offer a more reliable way of identifying a particular location in the instruction stream.

The ARM performance monitors were introduced in some ARMv6 processors and have been a part of the architecture starting with ARMv7. Their implementation is optional, but recommended.

Up to 31 performance counters can be supported in a system, plus one fixed-function cycle counter. Access to the performance counters normally occurs through a coprocessor interface available as CP15, but may optionally offer a memory-mapped interface as well.

On a system implementing both the performance counters and the virtualization extensions, the hypervisor can optionally reserve a range of performance counters for exclusive use in Hyp mode (PL2): Setting HDCR.HPMN to a value n allows access from PL0 and PL1 only to counters PMNx with $x < n$. The ARM architecture requires that at least one counter remains available outside of Hyp mode.

Reserving a range of counters for use in Hyp mode makes it easily possible to use the performance counters to monitor events inside the virtual machine, without having to worry about interfering with the guest operating system's own use of the performance counters.

Each of the performance counters can individually count one of 128 possible common architectural or implementation-specific events (of which not all numbers are allocated). Table 3.1 lists some of the defined common events [11, C12.8.2].

Event number	Mnemonic	Description
0x02	L1I_TLB_REFILL	Level 1 instruction TLB refill
0x06	LD_RETIRED	Instruction architecturally executed, condition code check pass, load
0x07	ST_RETIRED	Instruction architecturally executed, condition code check pass, store
0x08	INST_RETIRED	Instruction architecturally executed
0x09	EXC_TAKEN	Exception taken
0x0C	PC_WRITE_RETIRED	Instruction architecturally executed, condition code check pass, software change of the PC
0x0D	BR_IMMED_RETIRED	Instruction architecturally executed, immediate branch

Table 3.1: A selection of event categories commonly supported by the performance counters.

Of particular interest for record and replay is the INST_RETIRED event type, which simply counts each instruction in the instruction stream. Note that this event type does not include the "condition code check pass" requirement: Most instructions in the ARM instruction set can be made conditional, that is their execution depends on the result of a previous test operation. The INST_RETIRED event counts them no matter if they have actually been executed or failed their condition code check. This behavior is ideal for identifying a location in the instruction stream because instructions can be counted without considering the results of condition code checks.

The ARM architecture also provides events for branch counting. Record and replay projects targeting the x86 architecture typically use the branch counter for their landmarks, for instance in ReVirt [23]. This choice is motivated by the observation that the counters for retired instructions are not perfectly accurate on x86 [37] and that a single instruction prefixed with REP can be counted multiple times. Using the branch counter instead provides a less precise, but more robust landmark. ReVirt uses the branch counter to approximate a location in the instruction stream, then single steps until the exact location (as determined by the CPU state snapshot) has been found.

While multiple counters can run in parallel, counting different event types at the same time, only one of the counters can be configured and read at a time. Selecting a counter is done through the PMSELR register. The 32-bit counter value can then be read from the PMXEVCNTR register. Counter values can also

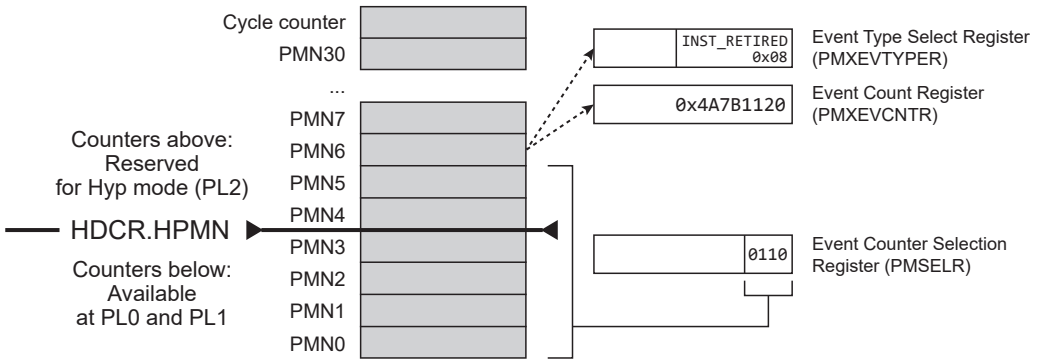


Figure 3.4: Performance counter control registers and their interactions. Using the HDCR.HPMN field, the hypervisor can reserve a number of performance counters for use in Hyp mode only. A single performance counter is selected using PMSELR, allowing the counted events to be configured (PMXEVTYPER) and the actual counter value to be read (PMXEVCNTR).

be written, which is helpful during virtual machine context switches. Figure 3.4 illustrates the use of the most relevant performance counter registers.

Since the PMUv2 version of the performance monitors extension, it is possible to filter events by privilege levels, which is particularly helpful for virtualization: When enabling the PMXEVTYPER.NSH flag, event counting takes place only at privilege levels PL0 and PL1. Hypervisor code running at PL2 is excluded from counting and therefore does not distort the landmarks.

As with the Generic Timer, there is no way for the hypervisor to prevent the guest system from using the performance counters: At least one performance counter as well as the CCNT cycle counter are always available [11, C12.7.2]. However, unlike the Generic Timer, accesses to the performance monitor CP15 registers can easily be trapped and recorded.

3.5 Conclusion

In this chapter, the events relevant for deterministic record and replay have been identified: Input events, interrupt requests and DMA transactions must be recorded in the event log because they can affect the replayed system non-deterministically. For the purpose of uniquely identifying the point in the instruction stream at which such an event must be replayed, the concept of landmarks has been introduced.

In the case of the ARMv7 platform, which will be the target of our replay efforts, the input events that must be recorded are reads from memory associated with peripheral devices (MMIO), as well as transfers from coprocessor registers to the ARM general purpose registers.

The ARM platform has support for two interrupt types, IRQs and FIQs, which can be signalled to the CPU. Changes to the respective interrupt line must be recorded; being asynchronous events, an exact landmark is required in this case for accurate replaying. No special consideration must be given to the interrupt controller because it is mapped through a memory interface, and thus already covered when recording MMIO reads.

Using the virtualization extensions, most of the events that have been discussed can be trapped by the hypervisor and thus recorded. The Generic Timer, however, which is accessed through the coprocessor interface, does not have any trapping mechanism. With no way of capturing the values read from the timer, record and replay must be restricted to guest operating systems that can be configured not to use the Generic Timer. The Linux kernel fulfills this requirement.

For the purpose of recording a landmark, the ARM architecture supports performance counters which allow instructions and branches to be counted, among other events. When combined with the virtualization extensions, events can be filtered between the host and guest machine contexts, making it easy to count only the instructions executed by the virtual machine.

Chapter 4

Design

This chapter concerns itself with the design of our heterogeneous record and replay system, stating the design goals, identifying the system's components, and enumerating which information needs to be recorded to enable replay later. The requirements for the event logging backend are discussed, while keeping the design independent from any particular hypervisor or virtual machine software.

4.1 Design Goals

Our work concerns itself with recording and replaying an ARMv7 virtual machine. While the platform for the guest system is therefore fixed, we do not want to constrain ourselves to replaying a virtual machine's execution only on exactly the same platform where it was recorded. In particular, it should be possible to perform recording on an ARMv7 host machine, leveraging the virtualization extensions for fast recording speeds, but later perform the analysis on an x86 PC. This goal is motivated by the greater availability and higher performance of development machines using x86. While ARMv7 machines are likely to be faster when recording an ARMv7 guest due to the use of hardware virtualization, a regular PC might still outperform them—due to its higher clock speeds and power budget—when software emulation is used for replaying. For this reason, the system should be built upon a virtual machine simulator software that runs on many host architectures.

Enabling high performance recordings is essential to observe a system in productive use. If the system becomes too slow during debugging, it can be difficult to even trigger the bug: The user interface might be not responsive enough to effectively interact with it, or a server may drop connections due to timeouts. The recording scheme should therefore produce only a small overhead, compared to the case where recording is disabled.

While the system should support fast recording, we make no such requirement for the replaying that takes place later: Performance is most critical when the virtual system is running and being interacted with. Since replay and analysis can be deferred until later, and performed on a possibly faster machine, we do not intend to optimize for replay speed. Similarly, it may occur that the virtual machine actually runs faster when replaying because it spends less time waiting for external inputs, or that the replay speed is not consistent within a run. For the designed purpose—to enable debugging and to trace execution of a virtual machine—the speed of the replay matters little, as long as correctness of the machine's execution can be ensured.

A replay system may have bugs, or there may be differences in the implementations of architectural instructions between the recording and the replaying system, which may lead to the replayed virtual machine behaving differently. Depending on how severe these differences are, execution between both machines may diverge, leading to an entirely different execution flow and to being unable to reproduce the system's behavior. In order to detect such a situation, there must be a component which regularly verifies if the state of the virtual machine matches what was observed during the recording, without causing too much overhead.

In Chapter 3, direct memory access (DMA) requests have been identified as a source of non-determinism in a system. Since DMA is an advanced topic, and primarily intended to improve performance, we try to first obtain a working system before implementing DMA transfers. To that end, the use of DMA in the guest system will be disabled where possible, enforcing the use of the programmed I/O (PIO) model which polls the device through the memory bus. Nevertheless, we include DMA in our system model to analyze its impact on recording performance.

The target system that the recorded virtual machine should eventually support will be the Linux kernel, being a widely-used operating system which can also be easily modified, should para-virtualization be necessary. However, Linux is a complex operating system, and it is to be expected that it uses many features of the ARMv7 architecture, making it a challenging task to ensure all of them are correctly replayed.

4.2 System Architecture

Our record and replay scheme for the ARMv7 platform consists of two main components, the recording system and the replaying system, as outlined in Figure 4.1. Each system implements a virtual machine with an ARMv7 central processing unit (CPU), although they do not need to share a common implementation, nor run on the same host computing platform. The two components are only coupled by way of the event log.

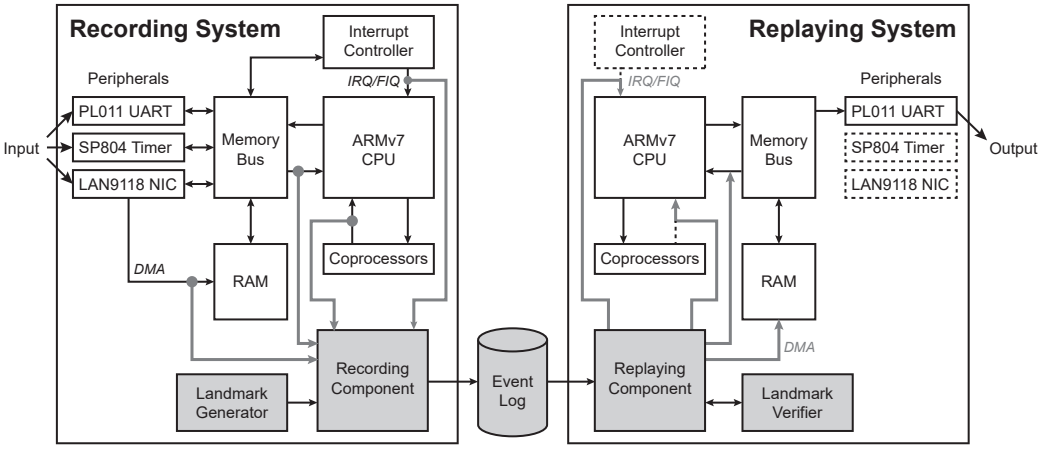


Figure 4.1: System overview for an ARMv7 record and replay scheme. A recording component taps into the memory bus and the coprocessor interface, passively recording data read by the CPU, DMA transfers, and interrupt lines. Events are stored in a log with their corresponding landmarks. In the replaying system, a replaying component reads events from the log and feeds them back into the system, replacing input from the affected devices. Note that in the replay system, peripherals are typically detached from the system such that they do not produce erratic output.

The recording system runs a virtual machine in the configuration that is intended to be debugged, i.e., with an appropriate operating system installed, networking enabled and peripheral devices (such as the serial console UART) connected. Shown in Figure 4.1 are sample peripherals supported on the ARM Versatile Express platform, which is a popular target for emulation. Integrated into the virtual machine software is a recording component which taps into the memory bus and the coprocessor interface, recording the data that is read by the CPU from peripherals, but not from RAM. All data collection is done passively; the recording component does not actively manipulate the execution of the virtual machine.

In order to inject events at the correct time later during replay, each event must be tagged with a landmark identifying its location in the execution stream. For this purpose, the system includes a landmark generator, which may examine the components of the virtual machine (e.g., the CPU registers or the RAM contents) in order to compute a landmark.

All events are stored in an event log, which is typically a file, but may also be a specialized tracing backend such as Simutrace [33].

The replay system is structured similarly to the recording system, but here, the component which controls the replay takes up a more important position: It reads events from the backing store and feeds them back into the running virtual ma-

chine, taking on the role that the peripheral devices normally would. A landmark verifier ensures that the system is in the correct state for injecting an event.

When in replay mode, the virtual devices that would normally be simulated are detached from the system because any data that the CPU attempts to read from them must be provided by the replay log. Writes from the CPU to the devices can be ignored, and in fact, they should be; otherwise, the virtual machine might cause side effects, leading to corrupted virtual disks or producing garbage network traffic.

While most devices should be disconnected, and writes to them should be ignored, there are exceptions to this rule: The coprocessor interface is used for configuring certain aspects of the system's caching and execution behavior and must therefore continue to receive writes, even during replay. The serial port UART is not crucial to successful replay, but it may be desirable to see the emulated system's output.

In addition to data transferred to the CPU, the replaying component also takes control of the virtual CPU's interrupt request (IRQ) and fast interrupt request (FIQ) lines, setting it high or low at the times indicated by the landmark. It is important that no other component in the system changes these lines, as this would trigger unwanted interrupts and prevent a correct replay.

One might consider the idea of making a virtual machine "go live" once there are no more events to replay, a concept described by Bressoud et al. [14] and implemented on x86 by Scales et al. [35] for the purpose of fault tolerance: Once replay is finished, the replay component is disabled, the virtual peripherals are reconnected, and the system continues executing from that point. It cannot be ensured, however, that the virtual devices or the environment are in the correct state to proceed with execution; the guest operating system's network stack, for instance, would find itself trying to continue TCP sessions which are long gone. For this reason, our system is designed to terminate execution once the end of the event log has been reached.

While there have been record and replay projects employing real-time communication between two systems, such as the failover replicas in [14] or the real-time threat analysis in Aftersight [16], our system is intended for offline analysis. As such, data is exchanged only in one direction through the replay log.

4.3 Landmarks

Landmarks, as defined in [39], serve as an identifier for a point in the execution flow of a virtual machine. A replay system can use them in order to know at which time an event must be injected into the virtual machine or to verify whether it executes in the same way as during the recording.

For synchronous events, which are triggered by actions performed from within the virtual machine, a landmark is not strictly necessary: A memory read, for instance, can only be replayed as soon as the corresponding load instruction is executing. The sequential ordering of the events, however, must be preserved.

Asynchronous events, on the other hand, require more information to locate the correct point in the instruction stream because they originated in the environment, and the virtual machine itself does not trigger a synchronous event by which to identify the insertion point. In order to replay asynchronous events such as interrupts, a landmark is therefore mandatory.

Since an ARMv7 CPU can only react to interrupts in between two instructions, defining a landmark as the number of instructions that have been executed by the CPU is sufficient to uniquely identify a point for replaying asynchronous events. On ARMv7 each instruction executes at most once—there is no `REP` prefix as there is on x86. Although execution of an instruction may be skipped because of the conditional execution feature, it makes sense to include it in the instruction counting scheme in that case to make a landmark unambiguous.

The performance counters on the ARMv7 architecture support counting the `INST_RETIRED` event, which implements exactly this instruction counting scheme. Furthermore, platforms implementing the second version of the performance monitoring extensions (such as the Cortex-A15 processor core) can filter events by privilege level, making it possible to exclude hypervisor code running at PL2 from counting and thereby to accurately count events in a virtual machine.

In order to prevent the virtual machine from interfering with instruction counting, a performance counter can be reserved for use by the hypervisor using the field HDCR.HPMN. This counter can neither be seen nor used by the guest system.

Typically, a hypervisor intending to count guest instructions will, upon virtual machine entry, store the current counter value and configure one of the reserved timers to count `INST_RETIRED` events at PL0 and PL1 only. Once execution returns from the virtual machine, the counter value is compared with the stored value to obtain the number of executed instructions. Performance counter values are limited to a width of 32-bits, after which they overflow; this limitation must be considered when obtaining a landmark, but never caused an issue in our experiments.

4.3.1 Verification

In theory, the instruction counter alone is sufficient to uniquely define a landmark. It is expected, however, that the record and replay system has bugs, inaccuracies or unidentified sources of non-determinism, leading to the virtual machine behaving differently during replay.

Previous projects targeting the x86 platform have observed inaccuracies in the instruction counts provided by Intel CPUs [37]. An incorrect instruction count can lead to interrupts being delivered too late or too early, which changes the execution flow of a virtual machine.

As identified in [15], sources of non-determinism abound, motivating the need for verification tools. To overcome replay issues, it should be possible to detect when the virtual machine's execution diverges from that of the recording by including additional information in the landmark.

Such extra information included in the landmark should be:

- Specific enough to help distinguish the correct state from incorrect ones;
- Quick to compute, in order not to cause a lot of overhead during recording;
- Quick to verify, such as to not slow down the replay too much, and
- Small in size, so that the landmark's stored size is not increased too much.

A sensible choice for landmark verification information would be the program counter register (R15): Its value changes after every instruction—except for pathological cases such as in an endless loop branching to itself—and it can reliably detect when the instruction counter is slightly offset. Being stored in a register of the virtual CPU, it can be quickly obtained and verified, and requires only 32 bits of space.

The record and replay scheme is designed so that it records reads from memory-mapped I/O (MMIO). Instructions accessing virtual RAM are not recorded, but assumed to be deterministic: RAM is only written to from the virtual machine or through DMA requests, whose content is also recorded. Bugs or differences in the implementations of instructions can however lead to different data being written to RAM when compared to the recording system; the Store Multiple (`STM*`) family of instructions, in particular, may or may not execute only partially in the presence of a data abort [11, B1.9.8].

Incorrect values written to RAM may lead to errors in the replay that can be hard to detect: They may manifest only several million instructions later when that value is read again and causes a different code path to be taken than in the recording. At that point, it is difficult to tell where the incorrect value originates from.

In order to catch memory errors as soon as possible, checksums of the virtual machine's RAM contents can be computed and added to a landmark. Such a checksum should be quick to compute and verify because a virtual machine's RAM size is typically in the order of gigabytes.

Various implementations exist, ranging from simple consistency checks such as CRC32 (32-bit) to cryptographically-safe algorithms including MD5 (128-bit), SHA-1 (160-bit) and SHA-2 ($\geq$ 224-bits). While all but the latter algorithms have been deprecated for use in cryptography, they are otherwise still suitable for the purpose of landmark verification in our system.

Computing a memory checksum may take a few seconds; for this reason, it is advisable to not add a checksum to every landmark, but only in intervals of around 10-100 million instructions. There is a trade-off to be made between detecting an error early, requiring shorter intervals, and keeping record and replay speeds high, which suggests larger intervals.

4.3.2 Debugging

Including the program counter in the landmark enables the detection—and, under certain circumstances, correction—of a shifted instruction count, as well as to detect if the execution flow inside the virtual machine is different from the recording. The reason for the diversion, however, can rarely be deduced from the program counter alone without looking at the CPU state.

Although primarily used for verification of the recording system, it may be helpful to generally collect more data in the landmarks than necessary, in order to help uncover bugs in the emulator. If new features are added to the implementation later, or the software implementing the replay component is changed, the extra information in the landmarks can be used for regression testing.

For this reason, the landmarks can be augmented by a complete register dump, which on ARM consists of 16 registers. Doing so allows for the debugging of replay errors by comparing the registers from the landmark with the actual CPU registers. To this end, it is helpful if the recording contains many events: The more landmarks appear in rapid succession, the more closely the replay log resembles an execution trace of the virtual machine.

Memory checksums, while helpful for verification purposes, are of only limited use when debugging a replay error: They can only detect *if* any byte in the system's RAM is incorrect, but do not give any information about *where* in memory the first error occurs. With perhaps millions of instructions since the last landmark, it is nearly impossible to trace back the location of the error.

Including a memory dump in a landmark can be a great help during debugging of the replay system, allowing to pinpoint the exact position of an incorrect byte, but with possibly many gigabytes of virtual machine RAM that need to be saved to the event log, they must be used very sparingly.

4.4 Event Types

Having previously looked at which architectural events can occur on the ARMv7 architecture in Section 3.1, we now introduce the six different event types that our record and replay system uses.

4.4.1 Initialization

While not strictly an architectural event, it is helpful to model the beginning of the recording as a special event that always starts the event log. The first event may contain metadata about the recording, for example information on which hardware platform it originates from, to enable platform-specific adjustments and workarounds.

More importantly, if this initialization event already contains a landmark, as is the case for all event types, then it can be used to verify the initial system state upon replay: By including a memory checksum, it is ensured that both the memory size as well as its contents match that of the recording system. This is particularly important because the replay system may have been launched with a different kernel image, or even just a different boot command line, both of which can affect the execution of a guest system.[1]

The initialization event might also include a memory dump that is loaded into the virtual machine RAM at startup, making the replay file self-contained because the kernel image and its command line are already included.

4.4.2 MMIO Reads

The recording system taps into the memory bus and records all values read from memory addresses which belong to a peripheral device (MMIO). Reads from RAM are considered deterministic and are therefore not part of the recording.

Being a synchronous event, replaying a memory read does not require an accurate instruction counter: The event is replayed exactly at the time the CPU issues the corresponding load instruction. The event's landmark can however still be used to verify if the execution flow and processor state are correct. Synchronous events can even be used to adjust an incorrect instruction counter by resetting it to the value contained in the landmark once the load instruction is executed.

Read instructions on the ARMv7 architecture range from an 8-bit width (`LDRB`) up to 64-bit width (`LDRD`); the log event must be able to accommodate values of any such size.

[1] The Linux kernel prints the command line to the console upon startup; so even if it contains just one extra space, the replay will fail at the latest when that extra character is written to the console and distorts the landmarks.

It is not required to store the memory address from which the value was read, since the address is implicitly defined by the instruction that triggered the event. Like the extra information in the landmark, it can however be used for verification purposes.

When recording with the virtualization extensions, memory reads cannot be trapped directly; however, they cause data aborts when the guest accesses pages which are not listed in the translation table. The hypervisor can omit translation table entries for the MMIO regions, causing a trap on any access to a device. Since the recording virtual machine emulates all virtual devices by itself and is responsible for generating a device's reply, it can easily record that value to the event log.

4.4.3 Coprocessor Reads

Coprocessor reads can be 32-bit (`MRC`) or 64-bit (`MRRC`) in size. Like MMIO reads, they are synchronous events, so they can be used for correcting the instruction count. The coprocessor parameters are encoded in the instruction and as such do not need to be stored in the event.

Certain optimizations are possible, for example in the case of the process, context and thread ID registers, which can be accessed through coprocessor CP15 with parameter CRn 13: They provide the guest operating system with space to store execution context information such as the current process ID. Thus, being usable like ordinary registers, they can be treated as deterministic for the purpose of record and replay and omitted from the recording. It must, however, be ensured that they are initialized to the same starting value in the recorded and the replayed system.

Other coprocessor registers, such as those of the Generic Timer interface (coprocessor CP15, CRn 14), are highly non-deterministic and must be recorded in any case. This poses a problem because not all of them can be trapped and recorded with the virtualization extensions, so their use must be disabled in the guest operating system. With no way for the hypervisor to enforce this restriction, the guest operating system must be trusted not to use the Generic Timer; the virtual machine software must provide an alternative timer implementation that is supported by the guest operating system.

The virtualization extensions provide support for trapping most other coprocessor accesses. The hypervisor can, through the use of the HSTR and HDCR registers [11, B4.1.66], decide for each CRn individually if it should be trapped or execute natively.

Not all coprocessor reads which need to be recorded are emulated by the virtual machine host software. In order for the recording system to be able to gather the result of a coprocessor read normally handled in hardware, it must perform a

single-step operation over that particular instruction and record the values from the result register. The ARM architecture provides support for this functionality in the form of the debug extension, whose breakpoint mechanism can be used to implement single-stepping [11, C3.3.5].

After the single-step operation has completed, the results of a coprocessor instruction can be gathered from the current virtual CPU state. The processor registers that need to be saved can be identified using the Hyp syndrome register (HSR), which on the original coprocessor trap event contains information about the `MRC`/`MRRC` instruction [11, B3.13.6].

4.4.4 Set Interrupt Lines

ARMv7 supports two external asynchronous events: interrupt requests (IRQ) and fast interrupt requests (FIQ). Virtual machine software provides an internal representation of such interrupt lines and sets them high or low, checking their value in the CPU loop and taking the virtual CPU to the corresponding exception mode if an interrupt is signaled.

In a virtual machine that executes using the ARM virtualization extensions, interrupts are injected through the HCR.VI and HCR.VF flags or generated through a virtual Generic Interrupt Controller (vGIC).

A recording system must record whenever the virtual interrupt lines change, which is only possible if the virtual machine has full control of them. For this reason, a vGIC must not be used and the GIC must be emulated entirely in software.

Unlike MMIO and coprocessor events, interrupt events are asynchronous and as such depend on an accurate landmark. A shifted instruction count must be corrected through extra information in the landmark, otherwise the interrupt is injected at the wrong time.

4.4.5 DMA Writes

DMA writes occur when a virtual device wants to efficiently transfer a (possibly large) block of data into virtual machine RAM. Since these writes are often the results of I/O operations (e.g., reading a block of data from the hard disk or receiving network packets), their timing is non-deterministic and asynchronous to the virtual CPU's execution.

In a virtual machine, DMA writes to the guest memory must be recorded with an accurate landmark such that on replay, the data block is written to RAM at the correct time; otherwise, the guest operating system, expecting the results of a DMA operation, might read incorrect data.

Care must be taken if the device driver performing the DMA write operates asynchronously to the CPU loop, as is the case if the virtual machine is currently

executing through the virtualization extensions: A virtual machine exit must be triggered in order to accurately obtain the current landmark.

Since the data written by DMA operations must be considered external, non-deterministic information, the entire data block must be saved in the replay log. Due to possibly varying transfer sizes, it is the only event which requires a variable record size in the log.

4.4.6 Trace

The landmarks which are part of the recorded events can be used to detect if the replayed virtual machine's execution diverges from that of the original recording. These landmarks are however only encountered when a non-deterministic event has been recorded. Depending on the workload in the virtual machine, several millions of instructions may be executed without triggering such a replay event, making it hard to pinpoint the source of an error when a landmark mismatch is detected.

For verification and debugging purposes, it is therefore helpful to introduce extra events into the record and replay scheme which have no relevance for enabling determinism, but whose only purpose is to periodically verify the execution state through use of the landmark.

In our system, we use a *trace* event, which can be generated on every virtual machine exit to the hypervisor. When such an event is encountered during replay, its landmark is checked against the current system state. Since virtual machine exits occur far more often than the actually recorded replay events—e.g., on data aborts when the hypervisor is asked to map a guest page—they substantially reduce the distance between landmarks and allow replay errors to be detected more easily.

Trace events are entirely optional, but when used, they depend on an accurate instruction counter. They do not need to store any other event information beyond the landmark itself.

4.5 Logging

After identifying the replay events modeled in our system, we now discuss the requirements for the replay log. The replay log stores a sequence of events in the order in which they were submitted, and later during replay allows for retrieving them in that order.

There are six different types of events in our record and replay scheme. Since the replay should follow the same execution flow as the recording, the relative ordering of the individual events is fixed; i.e., if a MMIO read event E_1 occurrs

before a coprocessor event E_2 during recording, then they will be in the same order during replay. For the log format, this means that the data associated with E_2 will never be required before E_1 has been processed, and thus all types of events can be written sequentially to the same stream.

While all event types can share the same stream, they have different size requirements, which are summarized in Table 4.1. Generally, the data associated with each event has a fixed maximum size, except for the DMA write event whose size is only bounded by the 32-bit address space (although in practice, individual writes of more than a few megabytes are unlikely). Nevertheless, this precludes the use of a fixed-size record, making iteration through the file slightly less convenient.

Event	**Stored information**
All	Landmark: • Instruction counter (32-bit) • Optional: Program counter register (32-bit) • Optional: General purpose registers (15 × 32-bit) • Optional: Memory checksum (32-bit to 256-bit, depending on the algorithm) • Optional: Memory dump (up to 4 GB)
Initialization	–
MMIO Read	• Value (up to 64-bit) • Optional: Memory address (32-bit)
Coprocessor Read	• Value (up to 64-bit) • Optional: Coprocessor number (4-bit) • Optional: CRn (4-bit) • Optional: CRm (4-bit) • Optional: Opcode 1 (3-bit) • Optional: Opcode 2 (3-bit)
Set Interrupt Lines	• IRQ line status (1-bit) • FIQ line status (1-bit)
DMA Write	• Destination address (32-bit) • Size (32-bit) • Content (variable size)
Trace	–

Table 4.1: Storage requirements by replay event. Data marked as *optional* is not strictly required for replay, but may be included for debugging and verification purposes.

The advantages of extra information in a landmark for verification and debugging purposes have been highlighted previously; some of that information should, however, not be part of every single landmark. Computing a memory checksum or adding a memory dump to every event would be excessive, so the log format should allow for adding such information to the landmark optionally on a per-event basis, without wasting that space when the extra information is omitted.

Considering the use of external tools, it may be helpful to decouple navigating through a replay log from the interpretation of the individual events: Adding a length prefix to each log entry avoids having to decode each event structure just to find out how many bytes it occupies before the next entry begins. Such a design is advantageous for performing analysis on replay logs in external tools.

For easy handling of the replay file, it should ideally be self-contained, i.e. not offload event data or memory dumps into a separate file. One might even go so far as to include the virtual machine settings and the kernel image in the replay file, e.g., by way of an initial guest memory dump.

Due to the regular structure of the recording file, with many events expected to contain identical data, compressing the event log data is worth considering. New events are only appended to the end of the log, making it conceptually suitable for streaming or passing through a compression tool.

4.6 Replay

In the replaying system, the log file is used as an input to produce a sequence of events. The replaying component must then use their stored data to feed the events back into the running system. While this is easily done for the synchronous events, since they are triggered through the instruction stream and can thus be processed strictly in order, the asynchronous events require special handling.

To inject an interrupt at the correct time, the replaying system must ensure that only a certain number of instructions are executed, up to the point at which the next asynchronous event must be injected.

Since the log file contains all relevant data for replaying accesses to virtual devices, they do not actually need to be emulated. Devices that interact with the environment beyond the virtual machine, such as an emulated storage device or a network interface card, must in fact be disabled during replay, or the instructions they receive from the replayed system might cause them to produce garbage output. As an exception to this rule, display and serial devices may be enabled so that the output from the replayed system can be observed.

4.7 Conclusion

In this chapter, we have stated the design goals for our record and replay scheme: It should enable recording an ARMv7 system with little overhead and later allow execution of that system to be replayed on a possibly different system. Only the information required for deterministic replaying should be included in the replay file, in contrast to execution tracing systems.

The recording system designed in this work consists of a recording component, recording data from memory-mapped I/O, coprocessors, interrupt requests and DMA writes, as well as a landmark generator which examines the virtual machine's state to uniquely identify a point during its execution. Events are stored in an event log, which can be used by the separate replaying system to feed information back into a running virtual machine.

Each event is augmented with a landmark providing information about the virtual machine's current state, allowing the correctness of the replaying to be verified. A landmark includes at the very least the instruction counter, but may optionally also include the program counter or other CPU registers. Memory checksums and dumps can be added to provide further verification.

Requirements for event storage have been identified: The event log must provide a way to sequentially write (during recording) or read (during replay) a single stream of events. Support for variable-sized events is required because DMA events have no fixed size.

Chapter 5

Implementation

The previous chapters have outlined the concept and challenges of heterogeneous record and replay on the ARMv7 architecture. In Chapter 4, a generic design for a record and replay scheme on the ARMv7 architecture has been presented. While looking in detail at the architectural features that must be considered for a recording, the design was intended to be independent of any particular virtual machine implementation, and thus did not concern itself with the application programming interfaces (APIs) of either the virtualization extensions or the virtual machine software.

The requirement for low-overhead recording of real-world workloads motivated the use of hardware virtualization for running a virtual machine. There are two major hypervisors supporting the ARM platform: Xen, which runs paravirtualized guest operating systems, and Linux KVM, supporting the execution of arbitrary guest systems. While both are open source, only KVM is suitable for system-level debugging of unmodified guest software.

In this chapter, our design for such a replay system is demonstrated in the form of an implementation using the QEMU virtual machine software and the Linux Kernel-based Virtual Machine (KVM). We elaborate on the changes that were necessary in these software projects to allow for deterministic replay and discuss the problems we encountered in enabling heterogeneous replay.

To demonstrate our record and replay scheme, we had to choose a virtual machine software upon which to base our project. We selected the QEMU open source virtual machine emulator [13] because of our group's familiarity with the project, and for its support for ARMv7 as both the host and guest architecture. Multiple existing projects in the field of record and replay have chosen QEMU for their implementations as well [21] [25]. Besides offering software CPU emulation through its Tiny Code Generator (TCG), it also supports running the guest system using KVM, which made it well-suited to our requirements. Both QEMU and the Linux kernel are written in the C programming language.

At the time we started implementing our system, QEMU 2.6.2 was the most recent stable version, and shall be the basis for this work. For the host system kernel, where changes were necessary to support our recording scheme, we used the most recent Linux kernel from the stable development branch, which was version 4.8.10.

5.1 Recording

Any recording system which uses KVM is fundamentally divided by the boundary between the kernel and userspace applications. As such, the recording component from our original design must be split into a kernel recording component within KVM and a userspace recording component within QEMU. Figure 5.1 shows the architecture of our recording system, with the new components highlighted in gray.

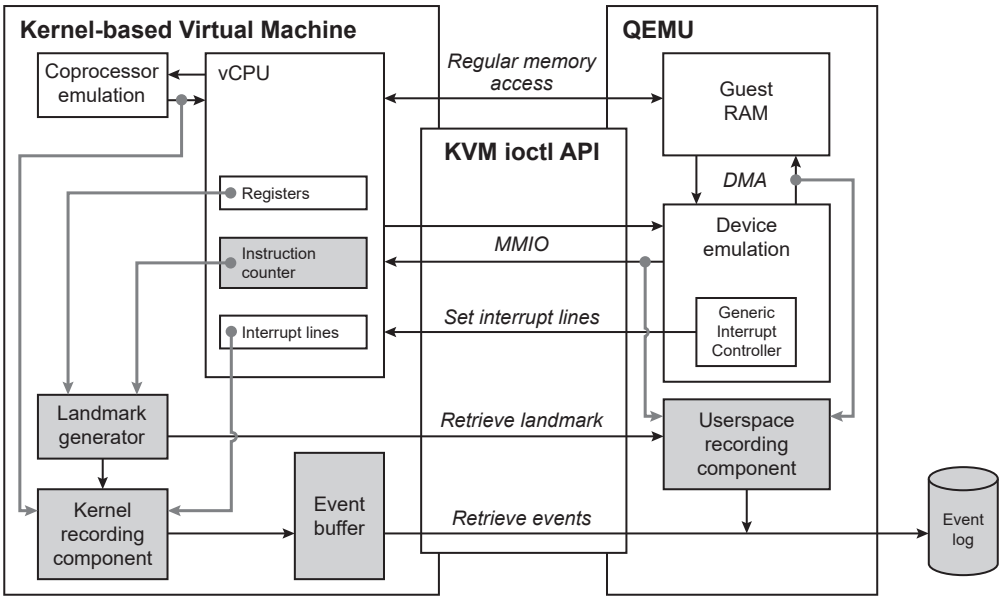


Figure 5.1: Components and their interactions in our KVM-based recording system. Coprocessor events and changes to the interrupt lines are recorded in the kernel, which also generates the landmarks. Memory-mapped I/O events, which are passed to the virtual machine software by KVM's design, are recorded in userspace, along with any DMA events. Events from the kernel event buffer are then merged into the event log.

Our system was designed so that costly context switches between kernel mode and user mode can be avoided. To that end, coprocessor reads and changed interrupt lines are recorded entirely within KVM, and the resulting events are written to a kernel event buffer of a configurable size. Once that buffer is full, a KVM exit to userspace is forced, causing QEMU to flush the buffer and save it to the event log.

All components of our recording system are disabled by default, preserving compatibility with existing KVM API clients and allowing for easy comparisons between regular KVM execution and recording mode.

5.1.1 Instruction Counting

The instruction counter is implemented using the performance counters offered by the ARMv7 platform, specifically the *Instruction architecturally executed* event (`INST_RETIRED`) defined as a common event for the architecture.

Additions were made to the KVM world-switching code such that instruction counting is enabled before a guest virtual machine is entered, and disabled upon returning. The implementation takes the following steps:

1. Select the designated performance counter for configuration by setting PMSELR.

2. Disable the counter through the PMCENTCLR register.

3. Configure the performance counter to count `INST_RETIRED` events at PL0 and PL1 only, excluding hypervisor code at PL2.

4. Load the instruction counter from the internal virtual machine structure into the PMXEVCNTR register.

5. Enable the counter through the PMCNTENSET register.

6. **Virtual machine entry**: Enter the virtual machine using the `ERET` instruction.

7. **Virtual machine exit**: May occur due to a trapped event or an expired time slice.

8. Disable the counter.

9. Write back the counter value to the VM structure's instruction counter.

The instruction counter in the kernel has a width of 32-bit. In order to allow for absolute landmarks without overflowing too soon, they are expanded to 64-bit after being received by QEMU.

Our implementation assumes that the CPU implements version 2 of the performance monitor extensions, which allows for counting guest instructions separately from the host instructions. Instruction counting is theoretically possible without this extension, but would introduce a systematic error through the VM entry and exit instructions that must be corrected carefully.

Having read about inaccurate counters on the x86 platform [37], we were initially concerned that the performance counters on ARM would suffer of similar inaccuracies. In fact, the Architecture Reference Manual is particularly vague on the accuracy requirements of the performance counters, permitting only a "reasonable degree of inaccuracy" [11, C12.2] without defining this criterium.

In order to verify the correct operation and the accuracy of the counters, small test programs were written which tested various instruction classes (arithmetic, branching and exception-generating instructions) on how they affected the instruction count. This task would have been easier if KVM supported single-stepping through a virtual machine, allowing to check the counter value after every instruction; however, unlike on x86 and 64-bit ARMv8, this feature is not implemented in KVM for ARMv7. For this reason, periodic virtual machine exits had to be triggered by performing a memory-mapped I/O operation. Upon exit, the instruction counter could then be read by QEMU.

Fortunately, the instruction counts we observed were extremely accurate. We noticed a systematic error whereby the instruction count was increased by one upon encountering a data abort, prefetch abort or interrupt exception. Our hypothesis is that the processor internally generates a branch instruction to the corresponding exception vector. Nevertheless, being a deterministic error, it is easily adjusted for during replay.

We encountered some minor issues concerning trapped operations, which, by being taken to the hypervisor, are never actually executed and thus not counted by the performance counters. When KVM performs emulation, it skips the trapped instruction by increasing the program counter. For this reason, the counter must be increased by one if `kvm_skip_instr()` is invoked. The `WFI` (wait for interrupt) and `WFE` (wait for exception) instructions are affected by this mechanism, as are data aborts and instructions involving memory-mapped I/O.

5.1.2 MMIO Reads

In our recording system, we assume the contents of RAM to be deterministic, thus requiring only reads from memory-mapped I/O regions to be recorded. From the hardware's point of view, the two cases are seemingly identical at first: The first

access to a page that is not currently in the translation table causes a data abort, which is then taken to KVM.

If KVM resolves the corresponding page as belonging to guest RAM, it creates an entry in the translation table, mapping that page directly to the memory block that the QEMU user process has allocated as guest memory. Execution in the virtual machine can then resume; if further accesses are performed to that page, no traps to the hypervisor occur, keeping execution speeds high.

On the other hand, if the accessed page does not belong to guest RAM, KVM first checks the KVM I/O bus for an internal handler for that address—such as a virtual Generic Interrupt Controller (vGIC)—and if there is none, generates a KVM exit to the userspace application controlling the virtual machine.

Accesses to memory-mapped I/O regions are thus reported to QEMU, which forwards that memory access to its simulated memory bus where it is then dispatched to the corresponding device function. Fortunately, the reads whose result we must record therefore end up in the same subsystem as during software execution using the Tiny Code Generator (TCG).

As this memory subsystem resides in user space, the resulting event can be written directly to the log without an intermediate buffer (as in the kernel recording component). The corresponding landmark can be retrieved through a KVM ioctl call.

5.1.3 Coprocessor Reads

While seemingly simple in theory, capturing the results of coprocessor operations turned out to involve many different approaches. We have identified four classes of coprocessor operations requiring different handling in KVM:

- **Coprocessor 15, CRn 15**: These accesses are already trapped by KVM through the HSTR register, so it was simple to hook into the existing emulation function and create an event on every read.

- **Coprocessor 15, CRn 4 and 14**: Accesses to these coprocessor registers, which include the Generic Timer, can never be trapped. The guest operating system must never use them, or deterministic replay will not be possible.

- **Coprocessor 15, All others**: While it is simple to trap these registers by additionally enabling their respective Tx flag in the HSTR register, they are taken to the same emulation function as CRn 15 registers. KVM does not expect to handle these registers, which means that no emulation functions exist whose result can be recorded.

- **Coprocessor 14, Debug interface**: These include, for example, the debug interface. They are not normally trapped by KVM.

The coprocessor accesses which are not normally emulated by KVM pose a problem: If we trap them to the hypervisor, we cannot obtain their result to record an event. On the other hand, if we let them execute, then the virtual machine will continue to run without generating a virtual machine exit.

We were able to solve this problem for coprocessor 15 by performing a single execution step in the virtual machine. Although KVM does not implement single-stepping support on ARMv7, hardware support for it is available in the form of the ARM debug architecture.

Single-stepping can be done through the debug interface [11, C3.3.5] by performing the following steps on virtual machine entry:

1. Through the DBGDSCR register, enable debug monitor mode and disable interrupts such as not to disturb the single-stepping.

2. Set the breakpoint value register (DBGBVR) to the current program counter value.

3. Configure the breakpoint condition to an *unlinked instruction address mismatch* at PL0 and PL1 only, through the use of the breakpoint control register (DBGBCR). This causes the breakpoint to hit if execution flow moves beyond the instruction that is to be stepped over.

4. Enable HDCR.TDE in order to route the breakpoint event not to the virtual machine, but to the hypervisor.

5. Disallow access to the debug interface from within the virtual machine through the appropriate settings in HDCR.

6. Enter the virtual machine.

7. The breakpoint hits after executing the target instruction, reporting the breakpoint as a prefetch abort exception which is taken to Hyp mode (PL2).

8. Examine the virtual CPU state to gather the results of the single-stepped instruction.

9. Re-enable access to the debug interface.

10. Disable the breakpoint.

As soon as execution returns to the hypervisor, the virtual machine's processor state contains the result of the coprocessor operation in one of the registers. The number of the register that must be saved can be gathered from the Hyp syndrome register (HSR), which is already provided in a decoded form by KVM.

One might consider, when encountering a trapped access to coprocessor 14, to apply the same strategy: Disable traps, enable a breakpoint and step over the instruction. However, the Architecture Reference Manual mandates that when using the debug interface from Hyp mode—i.e., when debug exceptions are taken to PL2 rather than into the virtual machine—access to the debug registers must be disabled [11, B1.8.9], which renders the single-stepping approach unusable.

We were therefore left with no hardware support for safely recording accesses to the debug registers. Since granting the guest machine proper access to the debug registers was not a priority for us at this time, we chose to implement access functions that provide a read-only view of the host register's state. While this was not a true fix for the problem, effectively preventing the guest from using the debug interface, it served as a stopgap solution to continue with our work.

Since coprocessor emulation occurs entirely in kernel space, we chose to record coprocessor read events in a kernel buffer to defer a costly KVM exit to userspace.

While coprocessor recording has generally worked well in our implementation, there were rare cases in which two coprocessor reads in immediate succession, such as when reading the DFSR and DFAR registers to obtain information about a data abort, failed to record the second access. We were unable to reproduce the problem in a test case and therefore do not know whether it is an architectural glitch or a bug in our implementation. As a workaround, if such a case is encountered during replay, the read is dispatched to QEMU's coprocessor emulation while emitting a warning.

5.1.4 Interrupts

Using the ARM virtualization extensions, it is not possible for the hypervisor to be notified when the guest CPU responds to an asynchronous interrupt request. The closest that the hypervisor can get to knowing when exactly an interrupt is taken is by assuming control of the source that generates it. Since the conditions for taking an interrupt are well-defined and deterministic, observing changes to the virtual interrupt lines HCR.VI and HCR.VF is enough.

All other sources of interrupts must be eliminated, which precludes the use of a virtual Generic Interrupt Controller, or *in-kernel* GIC in KVM's terms. This device must therefore be implemented entirely in software; an approach which KVM calls an *out-of-kernel GIC*. QEMU fortunately supports emulating such a

GIC in userspace, although it must be enabled by adding `kernel_irqchip=off` to the virtual CPU options.

The out-of-kernel GIC uses the `KVM_IRQ_LINE` call [8, 4.2.5] to set the status of the virtual machine's interrupt lines. KVM then updates the HCR flags upon each entry to the virtual machine. Upon receiving an interrupt, the guest operating system queries the GIC to acknowledge the interrupt; being a memory-mapped I/O operation, its result is recorded.

While there are two possible locations at which changes to the interrupt lines could be recorded—either in QEMU, tapping into the `KVM_IRQ_LINE` call performed by the emulated GIC, or in the kernel upon VM entry—we opted to record interrupt events in the kernel because it allowed us to capture the interrupt lines at the exact location where HCR is set, taking into account corner cases where, for instance, an interrupt is set while in the process of single-stepping.

5.1.5 DMA Writes

The virtual machine's guest RAM consists of a block of memory that is allocated by QEMU in userspace, then mapped into the virtual machine address space by KVM. QEMU can write to this memory at any time, and the emulated devices make use of this fact for implementing direct memory access (DMA). Instead of polling data from a device one word (32-bit) at a time, the guest operating system can instruct the device to write blocks of data—such as network packets or disk blocks—directly to RAM, which improves performance.

DMA transfers are initiated by the virtual CPU, but their resulting data and the time of their completion are determined by non-deterministic factors. As such, they must be recorded for correct replay later.

Writes to guest memory occur through the `address_space_*()` family of helper functions. They are, however, used both by some device drivers to perform DMA, as well as by multiple helper functions for ordinary guest memory writes, which we do not want to record. Our solution was to change the few cases where the functions were being used for DMA into calls to `dma_memory_*()` and record those instead.

The function `address_space_unmap()` is used for committing the result memory block of a DMA request into memory, and is also recorded.

DMA writes originate from QEMU, and therefore all data is already available in userspace for quick copying to the event log. However, since these requests can occur asynchronously to CPU execution, the virtual machine may currently be running, and thus the last landmark obtained from KVM may not be up to date—causing the DMA write to be replayed too early. In order to ensure accurate timing of the DMA write during replay, the proper course of action would be to force a KVM exit to synchronize the current landmark. In practice, one may assume that

the guest does not read the target memory area of the transfer before it is signaled the DMA completion interrupt, and thus that it does not matter if the response arrives too early. We could not test this hypothesis, however, because the Linux guest system we used for testing did not trigger any DMA operations.

5.1.6 Trace

Trace events are meant for debugging, serving no other purpose than to generate an event whose landmark can be verified to check if the replay of the virtual machine is still correct. They increase the number of landmarks in the log between recorded events, allowing for a more granular debugging.

If enabled in our system, a trace event is generated on every virtual machine exit, no matter if it results in an event being recorded or is merely caused by, e.g., a data abort handled by the hypervisor. In this mode, these trace events occur as a by-product.

We have also implemented an option which causes the virtual machine to operate entirely in single-step mode, causing a trace event to be generated for every instruction and effectively generating a complete execution trace of the virtual machine. This option has proven invaluable to track down some particularly elusive bugs, although, like most execution tracing solutions, it reduces performance extremely.

5.2 Logging

Once an event has been captured, it must be stored in an event log to enable replay later. We have considered different options for implementing the event log in our system.

Initially, our goal was to modify the existing record and replay mechanism in QEMU by Dovgalyuk et al. [21] for our own purposes because it already had an implementation for reading and writing log files, with error handling, command line options and endianess conversion taken care of. The idea was to replace the existing events by that of our own system. QEMU's existing record and replay scheme is however based on a different, higher-level approach, taking close control of the execution loop and the system's timers to ensure that virtual devices behave deterministically. We found that modifying this replay mechanism was a bad fit for our system, so we discarded this idea.

Our own logging implementation uses a single log file per recording for easy handling. Each event is written to the file in the form of a fixed-size structure, shown in Figure 5.2. The structure contains common fields for the event type,

flags and the landmark. Event-specific data is stored in a `union`, causing some overhead but simplifying the implementation.

While the event structure itself is of a fixed size (94 bytes in total), event data that is seldom used or of varying size is appended afterwards; this optional data includes memory dumps, checksums, and the contents of direct memory access operations.

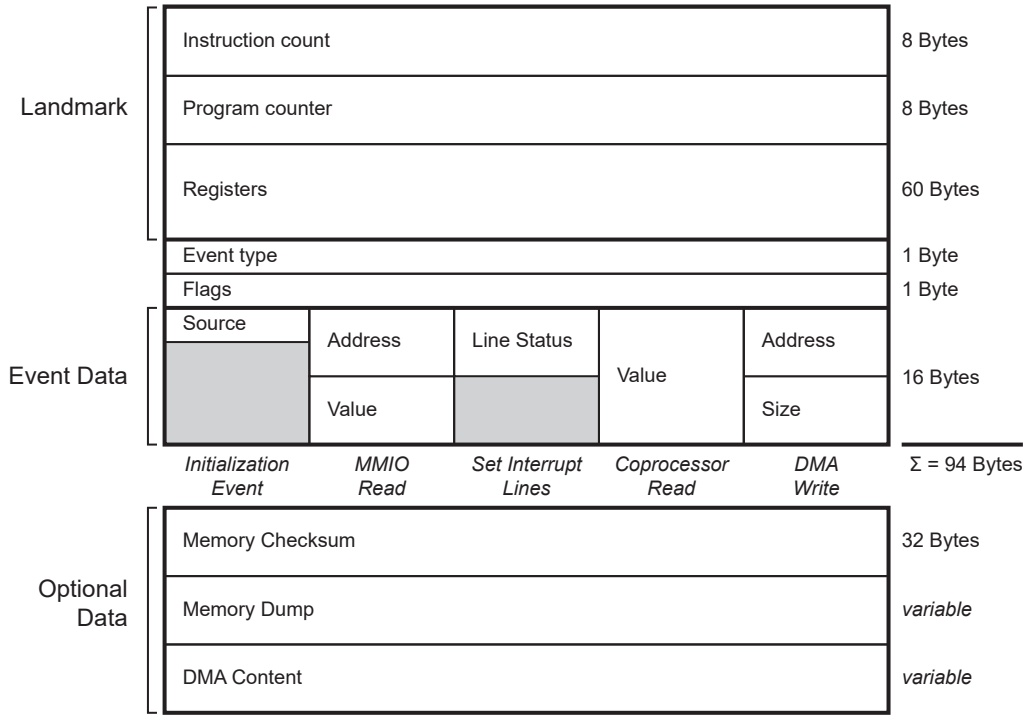


Figure 5.2: Structure of each event record (not to scale).

We have chosen the SHA256 hashing algorithm for the checksum because it was offered by the helper functions in QEMU. By default, a checksum is saved once on the initialization event to check the kernel image and its boot arguments. If desired, an interval can be specified for the number of instructions after which the next checksum should be generated. We have found an interval of 100 million to 1 billion instructions to be an acceptable setting without compromising execution speed too much.

When testing the memory checksum and memory dump mechanisms, we experienced a bug where the memory data read inside QEMU was inconsistent with the guest system's view of memory: The values contained in the memory dump were occasionally stale, whereas tracing the virtual machine's execution revealed that the virtual CPU was in fact reading up-to-date data. The problem only occurred when running simple test kernels which did not use virtual memory; when

running the Linux kernel as a guest, the memory contents were always read correctly by QEMU. We therefore assume that this is a rare caching bug which matters little in practice since QEMU normally does not read the guest's memory.

Improvements to our logging format could be made by packing some fields together or by using compression, as used in Simutrace [33], a specialized storage backend for high-speed tracing. Furthermore, we currently do not consider the endianess of the host architecture, which can be ignored in our case because Linux on ARM defaults to the same little-endian byte ordering as x86. To store the event log in a truly portable way, this issue should be addressed. For our initial efforts in evaluating the feasibility of record and replay on ARM, however, our simple log file approach has been an acceptable prototype solution.

5.3 Replay

We have thus far considered how to capture events when executing through KVM, and how these events are stored to the event log. The event log can then be used by the replaying system to reproduce the execution.

While it has been demonstrated on x86 that KVM can be used to implement such a deterministic replaying system [29], our work targets a virtual machine running in software emulation; in our case, QEMU using the Tiny Code Generator (TCG). The choice for following this heterogeneous approach was made because unlike recording, the replaying process is not time-critical, and implementing it in software allows for easier development and greater insight into the system.

In the following sections, we identify the approaches we took to feed the recorded events back into a virtual machine running in QEMU.

Once we had identified the locations in QEMU at which to replay events from the log, we realized that it was only a small step to enabling recording in TCG mode as well. Primarily for validation purposes, we have therefore successfully implemented homogeneous TCG-to-TCG recording and replaying, which helped in debugging the issues we encountered in the KVM-to-TCG case.

5.3.1 Instruction Counting

The replaying system reads a sequence of events from the recorded event log. Each event is tagged with a landmark, consisting of the number of instructions that have been executed up to that point and must be fed into the running system at the correct time, that is, at the correct number of executed instructions and under the condition that the landmark verification data, i.e., the registers, matches. There are two fundamental classes of events, requiring different approaches in QEMU:

- **Synchronous events**, which in our implementation are the MMIO and co-processor read events. Since they result from an instruction in the execution flow of the virtual machine, an appropriate event is automatically triggered at the correct time (assuming that the execution flow has not diverged). The instruction count in the landmark may be checked, although this is not strictly necessary.

- **Asynchronous events**, consisting of interrupts, DMA and trace events. They occur asynchronously to CPU execution, and thus a mechanism must be in place which interrupts CPU execution at the correct time for the event to be injected. A correct instruction count is mandatory.

A virtual machine software's CPU simulation executes in a looping fashion. One might therefore consider the following basic algorithm for replaying asynchronous events:

1. Obtain the number of instructions executed until now.
2. Check if there is an asynchronous event pending for exactly this instruction count.
    - If yes, replay that event.
3. Execute a single guest instruction.
4. Increment the instruction counter by one.
5. Go to 1.

While this approach would work in emulators performing interpretation, one guest instruction at a time, QEMU's Tiny Code Generator binary-translates several guest instructions at a time into a translation block (TB), which is its smallest unit of execution. A translation block executes completely, or not at all; it cannot be conditionally terminated in the middle.[1] Checking for asynchronous events in between iterations of the CPU loop therefore does not work correctly because an event may fall within the boundaries of a TB. An example where this approach would fail can be seen in Figure 5.3

Fortunately, the problems of counting executed instructions and inserting an event at the correct time have already been solved by Dovgalyuk et al. [21]: The *icount* instruction counting scheme that forms the basis of QEMU's existing high-level replay system allows for events to be injected at exact points within the instruction stream. To this end, TCG execution observes an icount limit at all

[1] It may, however, still be aborted by a synchronous exception.

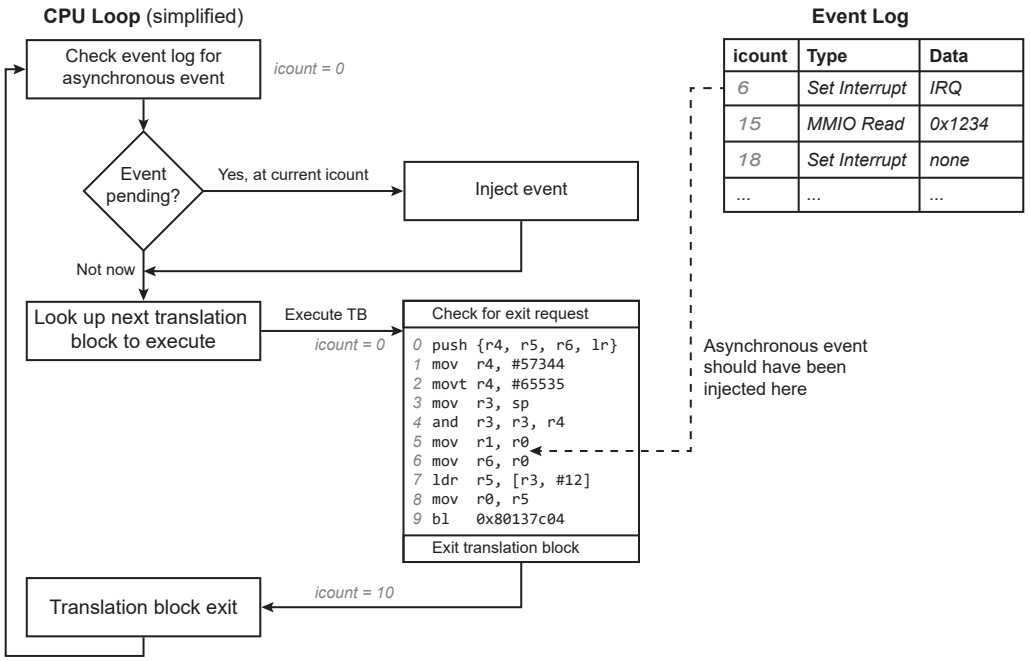


Figure 5.3: The problem of injecting asynchronous events when guest code is executed in translation block units. When entering the TB, the instruction count is 0, thus it is too early to inject the next interrupt from the log. After executing the TB, the instruction count is 10 and the correct time to inject the interrupt at instruction count 6 has been missed.

times. On each entry to the CPU loop, the next event is looked up in the event log to determine how many instructions may be executed until control must be given back to the CPU loop.

The code in each generated translation block is augmented by an icount check at the beginning, which determines if it can be executed as a whole while still falling below the current icount limit. If the translation block contains too many instructions, it exits to the TCG execution loop, signaling that the TB must be shortened. The translation block is then recompiled, taking into account how many instructions to include so that the icount limit is respected. Figure 5.4 illustrates this process. The reason why the icount limit is checked inside the translation block, and not in the TCG execution loop before calling the TB, is QEMU's chaining mechanism, by which successive translation blocks can be patched together without requiring an exit to the execution loop.

In our implementation, interrupts, DMA, and trace events are all considered when determining the instruction count limit. Any trace event from the recording thus lines up with the TB entry locations, allowing for verification of the guest's

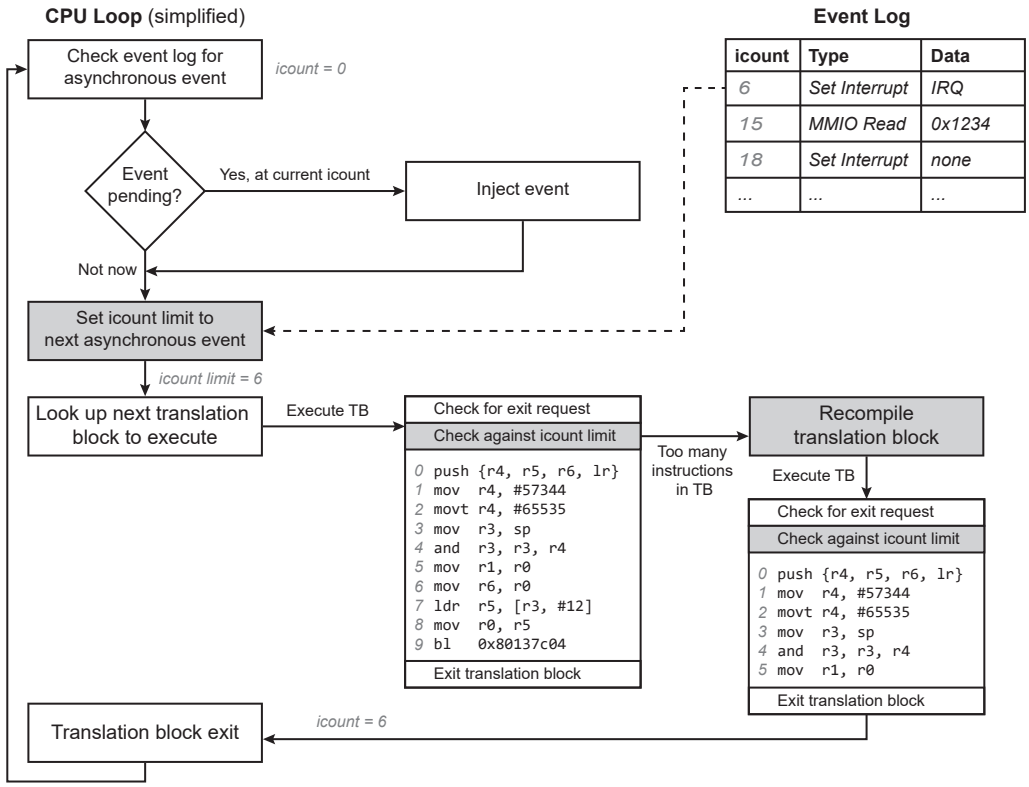


Figure 5.4: Injecting events at the correct time using the icount mechanism. Before executing any TB, the next asynchronous event in the log is looked up to determine how many instructions may be executed (here: 6 instructions). The first TB that is looked up turns out to be too large (at 10 instructions) and is therefore shortened to 6 instructions. After that TB exits, the CPU loop is resumed and injects the pending interrupt on the next loop iteration.

execution flow. Execution speed, however, is reduced because translation blocks are fragmented into smaller pieces, increasing the simulation overhead. When not required, it is thus faster to disable the verification of trace events.

The requirement to look ahead to the next asynchronous event is slightly inconvenient because unlike the designed system, the practical implementation requires seeking forward in the event log, rather than simply taking a look at only the next pending event. It is not sufficient to only consider the very next event in the log because even if that event is synchronous, it may have an asynchronous event waiting before the end of the TB, as exemplified in Figure 5.5.

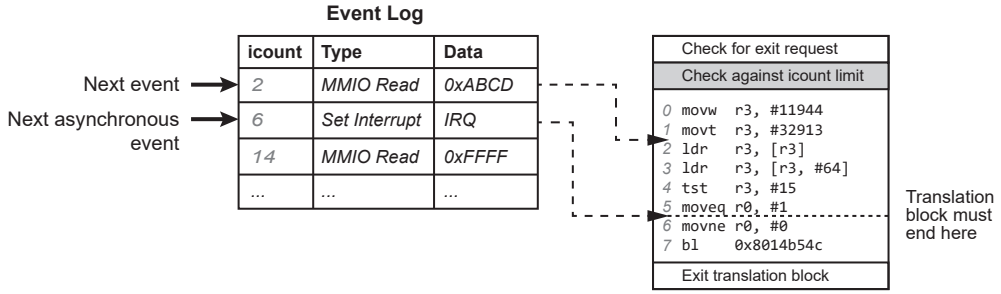


Figure 5.5: A case where a simple lookahead for the next event is not sufficient because an asynchronous event would fall within the same translation block as a pending synchronous event. If this translation block is executed as-is, the correct time to insert the interrupt event will be missed. The icount limit must be set with respect to the next *asynchronous* event, which may be several events ahead.

5.3.2 Landmark Verification

Beyond the instruction count, each event's landmark also contains additional information about the virtual machine's state that can be used to verify if the replay is working correctly.

Synchronous events are triggered as a result of an instruction and therefore match up with the corresponding replay event automatically; however, validating the instruction count would provide another way of verifying correct execution. Again, however, the translation block granularity of TCG complicates the issue: The instruction counter is only incremented upon entry to the translation block. In Figure 5.5, upon executing the MMIO read instruction at icount 2, the landmark in the replaying system would have already been increased to the value of 8 (the total number of instructions in this particular TB).

It is therefore not trivial to verify the landmark of a synchronous event, and not strictly necessary either; knowing the exact instruction count upon encountering a synchronous landmark, however, can allow for the correction of a shifted instruction count, which became important later during our experiments.

The first approach we took was to attempt writing the instruction count back incrementally when a helper call is encountered, which is the case for load instructions and coprocessor accesses. Due to the many convoluted ways in which a translation block can be traversed, including conditional execution and exceptions, we scrapped this approach.

Instead, the relevant memory read and coprocessor read instructions now end the translation block during translation. The instruction count is then still too high when encountering a synchronous event, but the error is always exactly one instruction and can be corrected.

For reasons we were unable to track down, terminating a translation block in the `gen_aa32_ld*()` helper template broke regular TCG execution (i.e., when not recording or replaying), causing endless loops when booting the Linux kernel, but working fine when using the TCG for replaying. Since the mode of operation is known when the TB is compiled, this issue could be easily worked around however and was not investigated further.

Landmarks contain a full dump of all CPU registers, which can easily be compared with the current virtual CPU state. The program counter register, R15, however, requires special handling: Like the instruction counter, it is not always updated after each executed instruction, but only written back at translation block boundaries. To obtain the current program counter value, for the purpose of landmark verification at synchronous events, we had to add various hooks to branch-generating TCG functions such that the up-to-date program counter is written to a field in the CPU state for use during helper calls.

5.3.3 Coprocessor Reads

Unlike in KVM, where recording the coprocessor accesses proved to be the most challenging part, recording and replaying them in QEMU was straightforward: We tapped into the `get_cp_reg()` and `get_cp_reg64()` helper functions to obtain the result or inject it from the replay log.

TCG optimizes its generated code by transforming accesses to constant coprocessor registers into immediate load operations, bypassing the helper function. We had to disable this optimization in order to log all relevant accesses.[2]

We noticed that a large number of accesses involve coprocessor 15, CRn 13, which contains extra registers for process and thread identification. Since the contents of those registers are fully deterministic, being written only by the guest, we always omit them from the recording.

Care must be taken when initializing the default values for these ignored registers: QEMU's default for these registers is 0, whereas KVM initializes them to the easily spotted magic numbers `0xDECAFBAD` and `0xD0C0FFEE`. This discrepancy was quickly discovered through the use of trace events, which contained the magic values in the landmark's register dump.

5.3.4 MMIO Reads

Reads from memory-mapped I/O areas are dispatched through QEMU's userspace memory subsystem, regardless of whether KVM or TCG is used. This design

[2]Even if the register is constant to QEMU, it may not be considered constant by KVM; our helper function must still be called to correctly acknowledge the event when replaying a log file from KVM.

makes them particularly easy to record and replay because it results in the same functions being used.

We selected the `memory_region_dispatch_read()` function for recording and replaying memory reads since it is the first function common to both the KVM MMIO and TCG code paths.

In rare cases, the function is called recursively when reads are performed through TCG, whereas they are directly dispatched to the correct memory region, without recursion, if the read is performed through `address_space_rw()` instead. Figure 5.6 shows a call graph we observed during replay. The recursive call led to problems when performing heterogeneous replay from KVM to TCG due to the different numbers of calls to our recording hook. We worked around the problem by capturing only the result of the outermost recursive call, since it overrides the result of all inner layers.

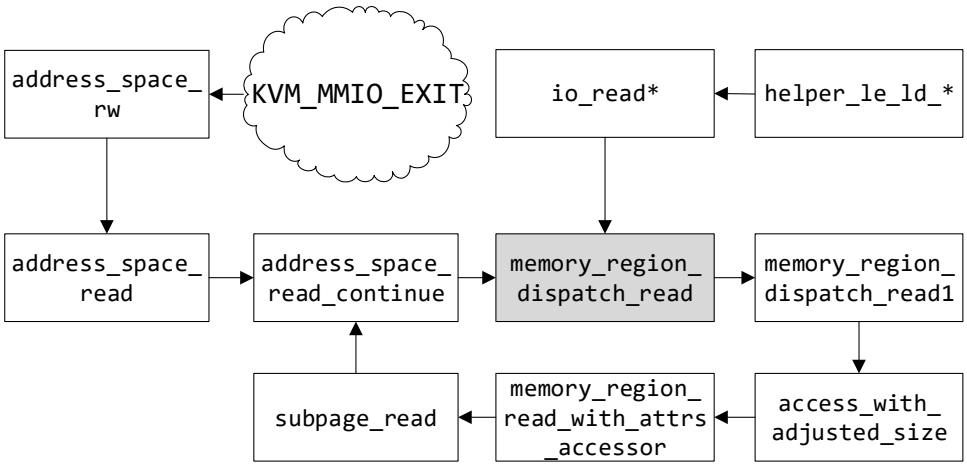


Figure 5.6: A possible call graph for the memory read functions. MMIO reads performed by TCG-generated code arrive through `helper_le_ld_*()`, whereas MMIO exits from KVM are translated into `address_space_rw()` calls. Note that there may be a cycle, depending on the implementation of the accessed memory region, which can lead to extraneous events being generated.

5.3.5 Interrupts

When executing a virtual machine using TCG, the interrupt lines are checked at the beginning of the TCG execution loop. If an interrupt (IRQ) or fast interrupt (FIQ) is signaled, and the CPU is in a state where interrupts are permitted, the CPU

is taken into the appropriate processor mode and branches to the corresponding exception vector.

QEMU's `interrupt_request` field corresponds nicely to the HCR.VI and HCR.VF flags used by KVM, where we record any changes to their state right before the virtual machine entry. If, during replay, an interrupt event is pending exactly at the beginning of the TCG loop, we restore the state of the interrupt lines as stored in the event. As detailed in Section 5.3.1, the icount mechanism is used to ensure that the check for an interrupt is not missed at the desired time of injection.

When recording interrupts in TCG mode, there were two different approaches we considered: Hook all locations that modify the virtual interrupt lines, or at each TCG loop entry, compare the interrupt lines' current state to the last known state. We chose the latter, since it closely resembles the approach we took when implementing recording in KVM, and because it allows for both recording and replaying to take place at the same location.

During development, we initially forgot to discard guest writes to the Generic Interrupt Controller (GIC) while replaying, causing it to unexpectedly modify the interrupt lines. Storing the last known state of the interrupt lines allowed us to detect if an unexpected source generates interrupts during replay, which has helped debug this particular issue.

5.3.6 DMA Writes

DMA writes are considered asynchronous events by our implementation, causing them to be inserted at the correct time by way of the icount mechanism. DMA writes are replayed at the beginning of the TCG execution loop by reading the DMA data that is stored in the event log and writing it to memory using `dma_memory_write()`, the same function that was originally used for recording.

Although we have implemented replaying DMA writes, we could not verify their correct operation since the Linux guest system we used did not use any DMA transfers.

5.4 Differences in Execution

At this point during development, having implemented record and replay for all the events we had identified during the design of our system, we were able to successfully record and replay a full Linux system bootup when performing homogeneous record and replay from TCG to TCG. We were therefore confident that recordings obtained through KVM could also be replayed on TCG.

As it turned out, replaying the non-deterministic events that we had previously considered to be relevant were not the only changes that would have to be made to the system. QEMU's implementation of the ARM instruction set contained several surprising differences to the actual hardware, while still conforming to the rules mandated by the ARM Reference Manual.

This section details the difficulties we encountered when attempting the heterogeneous case of KVM to TCG replaying, and the countermeasures we took.

5.4.1 Load Multiple

Landmarks, with their regular snapshots of the CPU state, provide a reliable way of ensuring that the virtual machine's execution is the same between the recording and replaying system. We occasionally observed cases where, on a replay event, a register in the virtual machine would hold a value different from what was recorded in the landmark.

Tracing the source of that register's value back revealed that it had been read from memory through a Load Multiple (`LDM`) or `POP` instruction, which reads values from multiple consecutive memory locations into a set of registers. It was not immediately obvious how a value in RAM could be different between the recording and replaying system if all previous landmarks had been correct, and especially given that the memory checksums matched.

Eventually, we noticed that the replay events which failed their landmark checks were always coprocessor reads in the data abort handler of the Linux kernel. Examining the CPU state at that point revealed that the abort was in fact caused by the Load Multiple instruction in question, and that the incorrect registers were those that had already been loaded at the time the data abort occurred.

It turned out that the Cortex-A15 implementation of the Load Multiple instruction discards all results when a data abort is encountered. QEMU, on the other hand, writes each value to a register immediately since it simplifies the implementation.

Both variants are correct: The ARM Reference Manual permits instructions which execute a series of sequential accesses to contain undefined register values on a data abort [11, A3.5.3]. Since the data abort handler will generally restart the faulting exception, the exact state of the registers does not matter for correct execution, but it is relevant for our landmark verification.

In order to quickly resolve the issue, we have therefore modified QEMU's implementation to behave in the same way that this particular hardware CPU does. An alternative approach—temporarily disabling landmark verification when a data abort is encountered—may be preferable since it takes into account that the register values, being unpredictable, may vary between implementations of the architecture.

5.4.2 Store Exclusive

When attempting to replay a KVM recording of booting the Linux kernel, we were regularly experiencing landmark mismatches after executing 500 million to 1 billion instructions: An interrupt event was to be replayed at the instruction count specified in its landmark, however upon arriving at that instruction count, the landmark contents did not match the CPU state.

Upon examining the situation at the time that the error occurred, we noticed that the landmark for the instruction where the interrupt should have been injected had already passed. Repeating the experiment showed that almost in all cases, TCG was reporting instruction counts that were too low by five instructions.

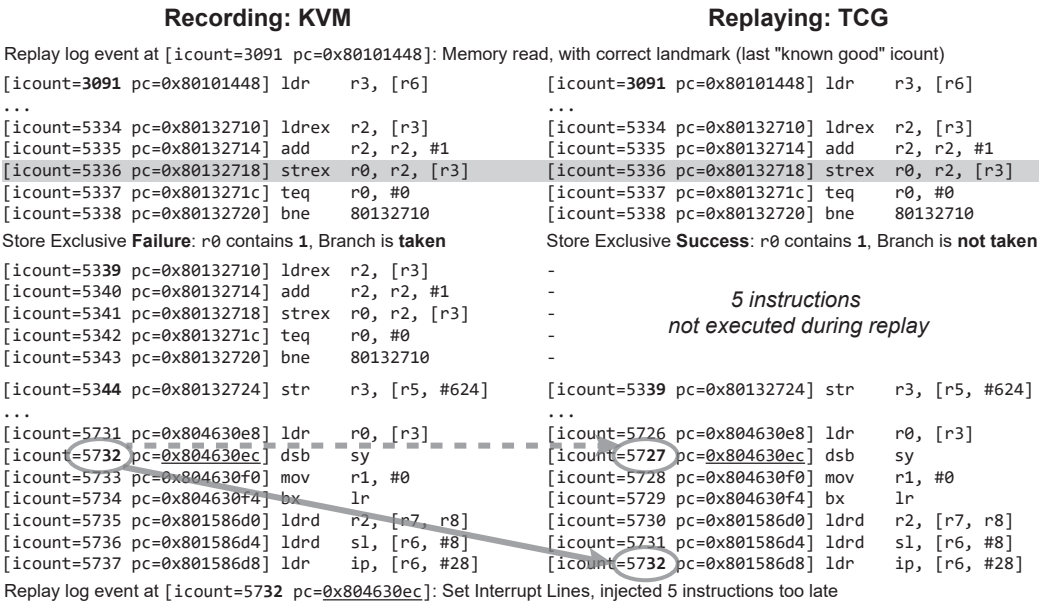


Figure 5.7: Example of an observed Store Exclusive failure, leading to incorrect instruction counts compared to TCG. (Numbers truncated for illustrative purposes.)

Furthermore, the errors always occurred around the same code addresses, suggesting that a particular instruction was to blame. Eventually, enabling single-step mode to create full execution traces revealed a difference between KVM and TCG: A *Store Exclusive* operation was failing in KVM for no apparent reason, but succeeded when replayed in TCG. Figure 5.7 shows an annotated example of such an execution flow.

The Load Exclusive (`LDREX`) and Store Exclusive (`STREX`) operations are used for synchronization purposes. Code that intends to atomically modify a value in memory uses Load Exclusive to read the value into a register, modifies it, then

writes it back to the same location using Store Exclusive. The latter operation only succeeds if no other observer, process or thread has written to that address in the meantime [11, A3.4]. On success, a value of zero is returned in the result register, whereas a failure is signified by a value of one.

QEMU correctly implements the semantics of Store Exclusive, only permitting a store to the location that was previously read. The hardware implementation used during recording, however, caused the instruction to fail, in spite of a correct load-store pair being executed. This behavior is permitted by the Architecture Reference Manual [11, A3.4.5], citing cache evictions as a possible reason for a Store Exclusive monitor to be cleared and the operation to fail with no apparent reason. A context switch, such as occurs on a virtual machine entry or exit, also clears the monitor, by design.[3] We initially thought these context switches to be the reason for the failures, but when we recorded the virtual machine exits and replicated this behavior in TCG using the trace events, the problem persisted.

The execution sequence in Figure 5.7 is typical of how exclusive instructions are used: Read a value, modify it, attempt to write it back, check the result, and retry if necessary—a sequence of five instructions.

Indeed, most asynchronous landmark mismatches we observed were caused by a shifted icount, and if so, it was always the instruction count from KVM that was in advance by multiples of five instructions.

The Store Exclusive instruction thus proved to be the first architectural instruction we encountered to exhibit non-deterministic behavior. Unfortunately, to our knowledge, there is no way to trap this instruction or to record its results.

We looked into the performance counters for events that might enable inferring the result of Store Exclusive operations. The Cortex-A15 processor core we were targeting supports an *Exclusive instruction speculatively executed – `STREX` fail* event; though, since it counts speculative execution, it does not make any guarantees about whether the instruction was actually retired. Even if one were to assume the counter as reliable, with possibly many Store Exclusive instructions in between two landmarks, the replayer would only know the total number of failures, and not their individual locations in the instruction stream.

Similarly, adding the branch counter to the landmark would enable the replayer to detect—at the next landmark—that it has not executed enough branch instructions, but it would not know which of them it would have had to take.

The problem with Store Exclusive could be solved if there was a way of trapping the operation. For lack of such a mechanism, it would be possible to instrument the Linux kernel's use of the instruction to cause a replay event directly

[3] This behavior also means that our single-stepping execution trace generator could never proceed beyond a loop which retries `STREX` until it succeeds. It was sufficient, however, to detect this issue.

afterwards, e.g., by introducing a dummy coprocessor read instruction. However, because the instruction is not privileged, this approach does not work for arbitrary user code running in the guest operating system, which may also use the instruction.

5.5 Landmark Correction

With no apparent way of recording the results of Store Exclusive operations, to proceed with our work, we considered implementing a mechanism to correct the shifted instruction counts. To do so, we had to make a rather strong assumption about the guest code: A failed Store Exclusive instruction does not affect the virtual machine state beyond a few instructions.

In particular, this assumption requires that the guest code does not perform any counting of how often Store Exclusive operations have failed, and does not write any such information to memory. While all code locations that we examined for this property were indeed benign, malicious guest code could deliberately introduce non-determinism into the system to thwart analysis.

If the assumption holds, however, we can use the next replay event that follows a Store Exclusive operation to resynchronize our landmark: Since the `STREX` failure does not have any lasting influence on the virtual machine state, we can simply adjust the instruction counter and continue replay normally.

Of particular importance to this mechanism are the synchronous events: the coprocessor and MMIO reads. Being anchored in the instruction stream, they can be used to accurately compare the current replay instruction counter with the value it should have, as specified by the recording.

The situation is more difficult, however, when considering the asynchronous events, which do require an accurate instruction count. A possible way to resynchronize these events is by single-stepping execution, checking the landmark after every instruction until the correct insertion point is found.

To avoid having to single-step over too many instructions, hindering performance, adjustments of the instruction counter are only permitted within a *resynchronization window* whose size can be configured. Single-stepping only takes place during the last Δ instructions before the next asynchronous event is expected. Figure 5.8 illustrates the approach.

There is no *correct* setting for the size of the resynchronization window, as multiple Store Exclusive operations may have failed since the last correct landmark, causing the error to accumulate. We have found a value of $\Delta = 30$ to work reasonably well, but not in all cases.

Choosing a too large resynchronization window can cause asynchronous events to be injected too early because of a false positive match on the landmark: There

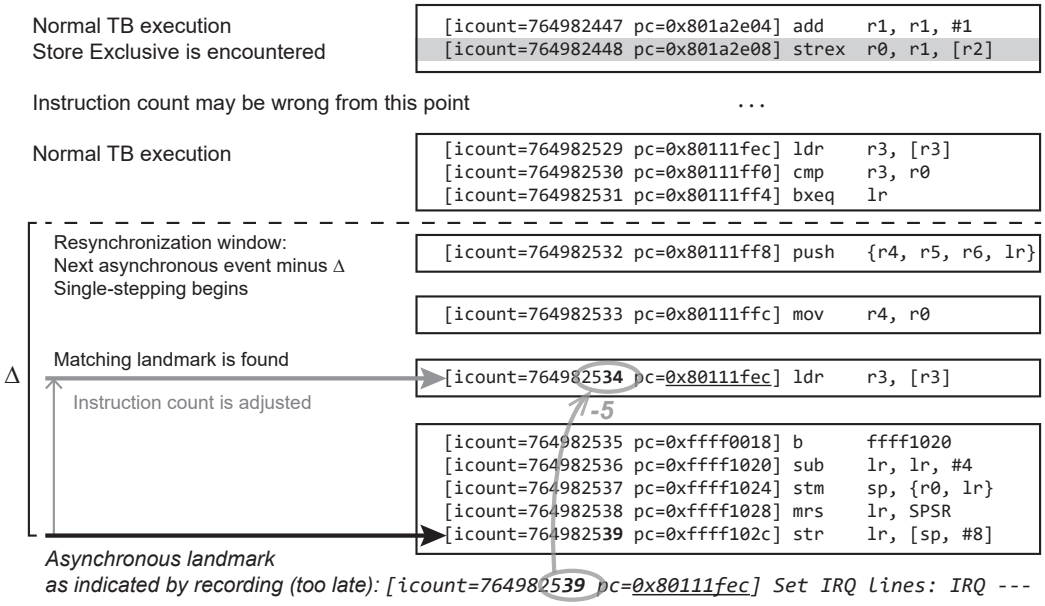


Figure 5.8: Single-stepping to correct a landmark. The asynchronous landmark stored in the recording has an icount that is too high (39), which would cause the event to be injected too late. To find the correct insertion point (34), execution is switched to single-step mode, comparing the current landmark to the one stored in the event. Note that the last block shown in this example already follows the corrected path of execution, with `0xffff0018` being the interrupt vector.

may be multiple points in the instruction stream where the landmark for the event matches, and the replaying system cannot tell in advance which is the correct one.

Curiously, when evaluating different sizes for the resynchronization window, we have occasionally observed interrupts that were obviously injected at the wrong position—sometimes hundreds of instructions in advance—but caused no adverse effects for the subsequent replay. This behavior can be explained by the design of interrupt service routines, which are carefully written to correctly restore the CPU state from the time the interrupt was taken; thus, there may be multiple points where the replayed interrupt may be injected, while still resulting in the same eventual virtual machine state. However, there were also many instances where false positive matches would in fact break the subsequent replay.

In our original design, we had intended to use the extra landmark information—the program counter and other CPU registers—only for verification and debugging purposes since the instruction counter alone is, in theory, sufficient to uniquely identify a point in the execution. The landmark resynchronization mechanism however, which is necessitated by the non-deterministic behavior of `STREX`, depends on this extra information to correct a mismatched icount between the record-

ing and replaying system. We thus had to deviate from our design by making the extra validation data a mandatory part of the landmark.

5.6 Conclusion

In this chapter, we have described the steps we took to implement the record and replay scheme designed in Chapter 4. Our implementation is heterogeneous in that the recording system uses the Kernel-based Virtual Machine (KVM), in order to enable high performance recording, while the replaying system uses QEMU's Tiny Code Generator (TCG).

Extending KVM to support recording non-deterministic events proved challenging due to various architectural details concerning coprocessor accesses. Multiple trapping mechanisms must be used for the different coprocessors, and their interaction with the debug subsystem—which is itself accessed through a coprocessor interface—made it difficult to correctly capture the results of read instructions by way of single-stepping execution. Not all issues have been resolved; for instance, the guest machine's access to the debug subsystem had to be disabled entirely.

Recording interrupts in the KVM subsystem was only a matter of saving and comparing the last state of the virtual interrupt flags. MMIO reads and DMA writes, both being events which are already handled in userspace within QEMU, were easily recorded.

A simple log file format was implemented, allowing for recording all required events into a single log file which can then be copied to another system for replaying. The replaying system in our design was implemented using TCG, allowing for better control and inspection of the virtual machine than would be possible with KVM.

We have noticed some minor differences in instruction implementations between the hardware and QEMU that could be easily worked around. However, we discovered that the Store Exclusive (`STREX`) instruction behaves non-deterministically in hardware, leading to major problems during replay because its behavior could not be accurately replayed in software.

In order to work around the issues revolving around Store Exclusive operations, we assumed that the guest code only uses the instruction in a safe way, not affecting the long-term state of the virtual machine. This assumption, however, precludes the recording of arbitrary guest code, since it could deliberately introduce non-determinism into the system through this instruction. While we had previously constrained that only privileged guest code must be trusted not to use the Generic Timer extension, the restriction concerning `STREX` applies to unprivileged guest code as well.

Under this assumption, we have implemented mechanisms to correct mismatched instruction counts between recorded and replayed events, in order to fix the non-determinism introduced by Store Exclusive operations.

Chapter 6

Evaluation

The previous chapters have detailed the concept, design, and implementation of our heterogeneous record and replay system for the ARMv7 architecture. While we have succeeded in performing homogeneous record and replay using the Tiny Code Generator (TCG), and have made advances towards a heterogeneous solution using the Kernel-based Virtual Machine (KVM), we have yet to show that our implementation works correctly.

In this chapter, we therefore verify the replay accuracy of our system and evaluate its implications for the performance during recording. We describe the mechanisms we have put in place to ensure that the replayed system's execution is correct, analyze the errors that may occur in practice as well as their frequency, and examine if the recording overhead is acceptable for practical use.

6.1 Approach

For a record and replay system to be practical, it must be *correct* in that the behavior of the replayed virtual machine is exactly the same as that of the original machine, and it must offer a higher *performance* than comparable solutions; in particular, the benefits in recording speeds should outweigh the added complexity over simply simulating the system in software.

Our implementation already contains mechanisms to ensure that the replayed system executes the same instruction stream as the original recording system, by way of the landmarks which contain the instruction counter as well as a full CPU register dump each time an event is recorded. Even small inaccuracies in the replay quickly make the two systems diverge, leading to an entirely different program flow which can be detected through a mismatched landmark register value. Optionally, RAM checksums extend verification to the virtual machine's system memory as well.

We demonstrate the functionality of our replay system by recording the execution of multiple benchmarks in a Linux virtual machine and later showing that they can be correctly replayed in the case of TCG. While our system cannot entirely replay a recording obtained through KVM yet, we examine the issues which prevent it from working and determine their frequency.

Performance has been one of the driving factors for hardware-assisted record and replay, and thus we compare the recording time of our system to the case where a regular software-only emulation approach is used.

6.2 Evaluation Platform

Our recording system implemented in QEMU can operate in two alternative modes, either using TCG to execute the virtual machine, or KVM. When recording is performed using TCG, the architecture of the host system does not matter since the virtual machine is simulated entirely in software. Executing the guest system natively using KVM, however, requires a host system implementing the ARMv7 architecture and its virtualization extensions.

Furthermore, our implementation makes use of the performance monitoring unit version 2 (PMUv2), which is required for accurately distinguishing between host and guest instruction counts. For development and testing, we had to choose a hardware implementation which fulfills all these criteria.

Popular ARMv7 implementations including the virtualization extensions are the low power Cortex-A7 and the high performance Cortex-A15 processor cores, which are architectually compatible to each other. Besides being used in many smartphones, they are also part of various embedded single-board computers.

The particular system we used for development and testing was an Odroid-XU3 single board computer [6], which features a Samsung Exynos 5422 processor with four Cortex-A7 and four Cortex-A15 cores in a heterogeneous multiprocessing configuration.

The Linux kernel has built-in support for the Exynos 5422, which means that we could use the most recent stable release of Linux—version 4.8.10 at the time of our development—as the starting point for the recording host system. We had initially tried to base our work on the vendor's official kernel branch for the Odroid-XU3 derived from Linux 3.10, but were experiencing host kernel panics when executing most modern Linux kernels as guests on KVM.

Although all cores in the system are architecturally compatible to each other, KVM only supports running a virtual machine on a core of the same family that is being emulated. Since all CPU cores are treated the same way by Linux, we had to constrain QEMU to only run on a Cortex-A15 core, or it would be randomly assigned to a CPU core of the wrong type.

A suspected hardware bug on this particular series of development boards causes one of the CPU cores to wake up in Service mode, rather than Hyp mode, when resuming from a sleep state. Linux detects this inconsistency between the cores and normally refuses to enable KVM. In order to get KVM working, we therefore had to apply a workaround patch [5].

The development board includes 2 GB of LPDDR3 memory, a Fast Ethernet network interface card internally attached to the USB 2.0 bus, as well as a serial port pin header for console output. Storage can be attached in the form of a microSD card, or optionally an eMMC module.

Throughout developing and testing, we have used an Ubuntu 16.04 LTS installation on the Odroid-XU3. The kernel, which includes our modifications to KVM for record and replay, was loaded to the machine using TFTP at each boot. The Network File System (NFS) was used for providing the root file system, in order to use the microSD card entirely for storing and exchanging replay log files.

The replay log files were stored on a 32 GB SanDisk Ultra microSD card with a Class 10 speed rating, formatted with the ext4 file system.

6.3 Virtual Machine Setup

Unlike the emulation of x86 guest machines, there is no *default* ARM guest platform in QEMU, but a wide range of possible virtual machine implementations of the ARM architecture. Upon closer inspection, however, it turns out that many of these emulated platforms are outdated, rarely used, or do not support all of the required virtual devices.

The two possible options we considered for the virtual machine platform were the `virt` and the ARM Versatile Express (`vexpress-a15`) machines. The `virt` machine uses virtio devices, which provide high-speed paravirtualized input and output when used with KVM; however, since the use of paravirtualization would likely complicate our design, we decided not to use the `virt` machine. The `vexpress-a15` machine, however, is a popular target for emulation, and was thus chosen for our work.

For the guest operating system, we selected the stable Debian GNU/Linux 8 "Jessie" release because it offered up-to-date installation files suitable for use in ARMv7 guest machines, including the Versatile Express platform. We did not modify the guest kernel, but used the official Debian kernel image installed by the setup routine, named `3.16.0-4-armmp-lpae`.

A sparse 16 GB disk image in the `qcow2` format was used for the system drive, stored on the host system's microSD card, and provided to the virtual machine as an SD card image—the only option supported on the Versatile Express machine and unfortunately without any DMA support.

As outlined in Section 3.4.2, for record and replay to possibly work, the use of the Generic Timer must be disabled entirely in the guest system. Fortunately, Linux offers a way to do so using the device tree mechanism, which provides the operating system with information about the devices that a particular platform supports. To disable the Generic Timer, we removed the `arm,armv7-timer` entry from the device tree passed to the virtual machine.

Removing the device tree entry for the Generic Timer is not uncommon: The Linux device tree for the Exynos 5422 platform, for instance, which we used for the host machine, does not contain such an entry, despite the processor hardware offering a Generic Timer implementation.

The Versatile Express platform we used for our virtual machine contains an ARM SP804 Dual-Timer module, which is also emulated by QEMU and automatically used by the guest Linux system to provide a clock source. This timer is made available through the memory-mapped I/O interface, making it easy to capture using our recording scheme.

In order to use KVM, the secure mode of the virtual machine's CPU had to be disabled because it cannot be used in combination with the virtualization extensions. Furthermore, as explained in Section 3.4.3, our record and replay scheme requires that a software GIC is used. Both can be achieved using the setting `-machine vexpress-a15,secure=false,kernel_irqchip=false`.

The virtual machine was provided with 1 GB of guest RAM, which is half of the host memory available on the Odroid-XU3. No swap partitions were used on either the host system or the guest machine.

6.3.1 TCG Reference System

In order to allow our hardware-virtualized system's performance to be compared to the software-only case, we have performed our testing on an x86 host system as well, using the Tiny Code Generator for emulation of the virtual CPU. The system's specifications were as follows:

- **CPU**: Intel Xeon E5-2618L v3 [4], with 8 processor cores running at 2.30 GHz (base speed) to 3.40 GHz (turbo frequency), HyperThreading, and 20 MB cache.
- **RAM**: 4×8 GB DDR4-2133 memory (Micron 18ASF1G72PZ-2G1A2) with Error Correcting Code (ECC).
- **Storage**: 256 GB Solid State Drive (SanDisk SDSSDHP256G) using a Serial ATA 3.0 interface.
- **Operating System**: Ubuntu 16.10 GNU/Linux, 64-bit version.

The system was used exclusively for testing, with no other concurrent users of the machine, thus allowing for the full single-core turbo performance to be used. The same virtual machine image and settings were applied as when using KVM on the Odroid-XU3.

6.4 Benchmarks

For evaluating the performance of our record and replay implementation, we have used the Phoronix Test Suite [9], which allows for automated testing on Linux systems. The particular tests to be run were chosen by their popularity on the OpenBenchmarking.org website, as well as by their testing focus (CPU, file system or RAM).

We have selected the following tests to be performed:

- The **Linux kernel build** benchmark, which compiles a Linux 4.9 kernel and measures the time taken. This is a highly CPU and disk intensive benchmark which is commonly used as a performance gauge for a system.
- The **Apache** benchmark, which is also CPU and disk intensive, but more lightweight than the Linux kernel build and thus quicker to finish.
- The **C-Ray** benchmark, which performs little to no I/O, but uses primarily the floating point unit to raytrace an image.
- The **CacheBench** benchmark, which tests the performance of the RAM and caching subsystem.

For each particular benchmark, we have performed reference runs in both KVM and TCG modes using an unmodified QEMU 2.6.2 emulator, which are identified as the *Running* column in the subsequent test results. The KVM runs were performed on the Odroid-XU3, whereas in the TCG case our x86 testing machine was used.

The *Recording* benchmark passes were performed with our custom version of QEMU supporting record and replay, saving all recorded events to a log file on the local machine. After completion, each recorded file was replayed on the x86 system using TCG.

In order to accurately gather timing information, the output from the virtual machine was timestamped using the host clock such that the effective runtime and replay time of a test could be calculated, independent of any possible clock skew inside the virtual machine and of the guest system's boot time overhead. The same was done for the log file size.

Each test using the KVM mode was performed twice, and the arithmetic mean was used to obtain an average of both results. Due to the high runtimes, we configured the Phoronix Test Suite to perform at most one iteration of each test, and only performed one pass of each benchmark in the TCG modes. For the same reason, we have left the memory checksumming mechanism disabled because it slowed down execution considerably.

While all TCG recordings could successfully be replayed, replaying the KVM recordings always failed during the virtual machine system bootup because of the unresolved issues in the recording system. We added the number of successfully executed instructions to the test results, up to the point at which the replay had to be terminated due to diverging from the recorded system. We also provide the number of landmark resynchronizations that had to be performed up to that point due to inaccurate instruction counts (see Section 5.5).

6.4.1 Linux Kernel Build

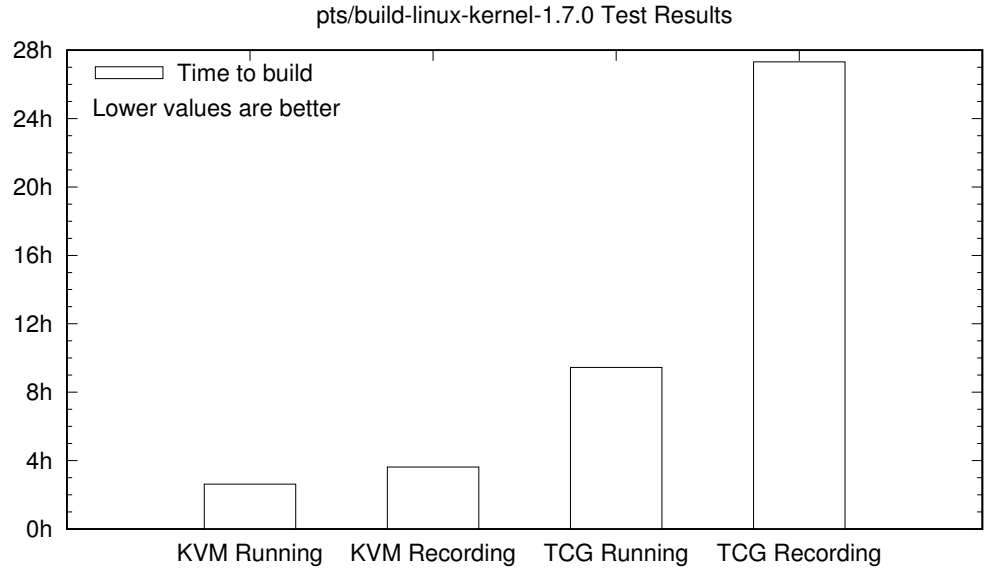
The Linux kernel build benchmark compiles a copy of the Linux 4.9 kernel source and measures the time it takes to complete, which is a popular way of measuring a system's overall performance because it is easily replicated. Figure 6.1 shows our test results for this benchmark.

Compiling the Linux kernel has turned out to be very taxing on the virtual machine. When performing the test on KVM, there is a recording overhead of 38 % on the execution time compared to simply running the test. In the case of TCG emulation, the base running time is already very high at over 9 hours, and performing a recording increases the running time by 189 %.

While both the KVM and TCG tests run the same benchmark, and thus perform the same amount of work, the TCG recording file is much larger at 44.6 GB, compared to the average 22.6 GB of the KVM recordings. The increased file size can be explained by the fact that the virtual machine was running for a much longer time, and thus performed more device I/O operations that needed to be recorded.

The large overhead between the total recording size and the test data itself is mainly caused by the post-test script, which was not counted towards the test itself.

QEMU's emulated SD card uses a FIFO buffer to communicate with the device driver in the virtual machine, which is consistent with our observation that the virtual machine does not use any DMA operation. This communication model, however, is very slow, since the data from the disk must be polled through memory-mapped I/O operations in small chunks, rather than being written directly to RAM in large blocks. With the Linux kernel build being an I/O intensive benchmark, the bottleneck is, in this case, the emulated SD card interface.



	KVM		TCG	
	Running	Recording	Running	Recording
Test result	2h 37m	3h 37m	9h 26m	27h 18m
Test runtime	2h 37m	3h 37m	9h 26m	27h 18m
Recording size		22.64 GB		44.61 GB
Test only		13.33 GB		34.10 GB
Compressed size				
gzip		85.7 %		84.9 %
xz		7.5 %		8.2 %
Replay runtime		–		30h 56m
Instruction count		$5.0 \cdot 10^{12}$		$5.1 \cdot 10^{12}$
Replayed		$5.7 \cdot 10^{9}$		$5.1 \cdot 10^{12}$
No. of resyncs		12		0

Figure 6.1: Benchmark results from running the Linux kernel build benchmark, averaged over two runs in each KVM mode and one run in each TCG mode.

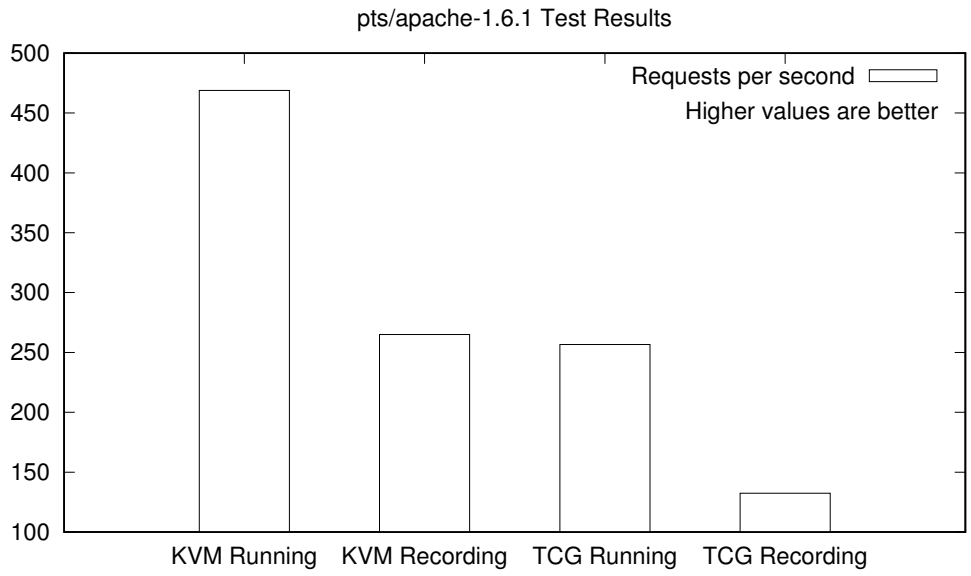
6.4.2 Apache

The `pts/apache` benchmark launches an Apache HTTP server and uses the `ab` benchmarking tool locally to test how many requests per second can be handled. In each run, one million requests are performed at 100 concurrent connections. Figure 6.2 shows the results we have obtained.

At a runtime of 35 minutes in the KVM case, the benchmark is much more lightweight than the Linux kernel build, which took two hours longer. Nevertheless, being also bound by the CPU and I/O performance, it exhibits similar overheads: Recording using KVM reduces the requests that can be served per second by 43 %, whereas the reduction amounts to 48 % when recording in TCG mode.

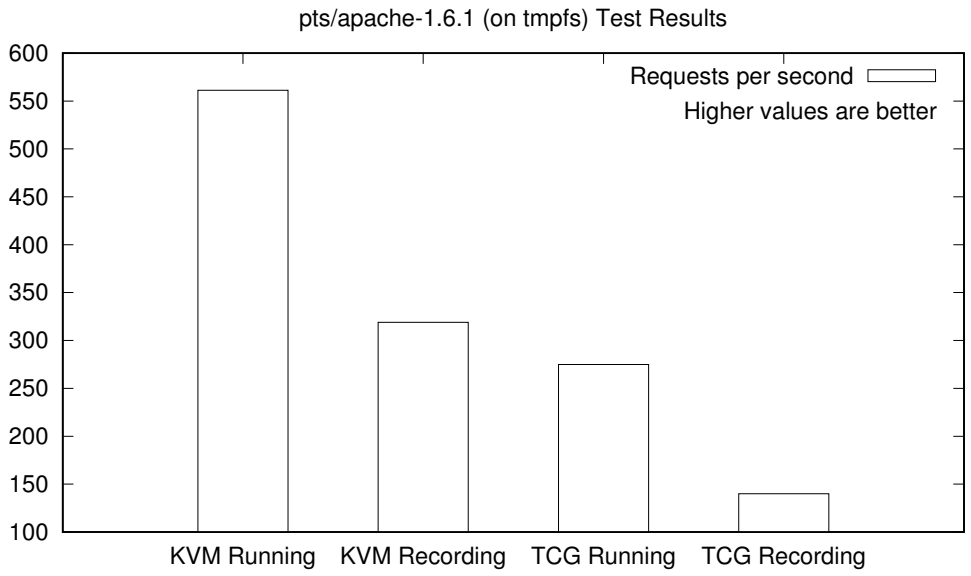
Since we assumed the SD card, on which the test data is stored, to be the major bottleneck during the benchmark, we performed another series of runs in which all benchmark files were stored on a `tmpfs` file system. `tmpfs` is effectively a RAM disk, saving all its data to memory only and thus reducing the data that is read and written to disk. We configured the file system to use up to 768 MB of the virtual system's 1 GB RAM. No swap space was used. We intended to try the same approach for the Linux kernel build as well, but its space requirements were too high for the limited guest memory.

While the resulting speed improvement was not as large as we had hoped, executing the tests from RAM increased the request rate by 20 % (KVM) and 5-7 % (TCG). Figure 6.3 shows the detailed results of the benchmark.



		KVM		TCG	
		Running	Recording	Running	Recording
Test result	[Req/s]	468.8	264.9	256.7	132.4
Test runtime		35m 41s	1h 03m	1h 05m	2h 06m
Recording size			10.87 GB		11.96 GB
Test only			9.41 GB		10.39 GB
Compressed size					
gzip			86.7 %		85.2 %
xz			6.6 %		7.3 %
Replay runtime			–		2h 15m
Instruction count			$2.9 \cdot 10^{11}$		$2.9 \cdot 10^{11}$
Replayed			$5.6 \cdot 10^{9}$		$2.9 \cdot 10^{11}$
No. of resyncs			8		0

Figure 6.2: Benchmark results from running the Apache HTTP server benchmark.



		KVM		TCG	
		Running	Recording	Running	Recording
Test result	[Req/s]	561.3	318.9	274.9	139.9
Test runtime		29m 47s	52m 23s	1h 00m	2h 00m
Recording size			10.42 GB		12.14 GB
Test only			8.54 GB		10.31 GB
Compressed size					
gzip			86.8 %		85.2 %
xz			6.6 %		7.3 %
Replay runtime			–		2h 13m
Instruction count			$2.7 \cdot 10^{11}$		$2.9 \cdot 10^{11}$
Replayed			$3.7 \cdot 10^{9}$		$2.9 \cdot 10^{11}$
No. of resyncs			11		0

Figure 6.3: Benchmark results from running the Apache HTTP server benchmark on a `tmpfs` file system.

6.4.3 C-Ray

C-Ray is a very small raytracing program that uses the floating point unit (FPU) of the processor extensively [3]. In the Phoronix Test Suite, it is used for rendering a 1600x1200 pixel image at 16 threads per CPU core, casting 8 rays per each pixel.

Considering the test results in Figure 6.4, the differences between TCG and KVM mode are striking: While the test completes in under 9 minutes on KVM, executing it using TCG is much slower at 2 hours 19 minutes when running and 3 hours 6 minutes when recording.

This behavior can be explained by the fact that the benchmark uses primarily the floating point unit, with few system calls or I/O operations, and can thus spend most of its time natively executing through KVM. As a result, very few events must be recorded and the recording overhead is only about 1 %.

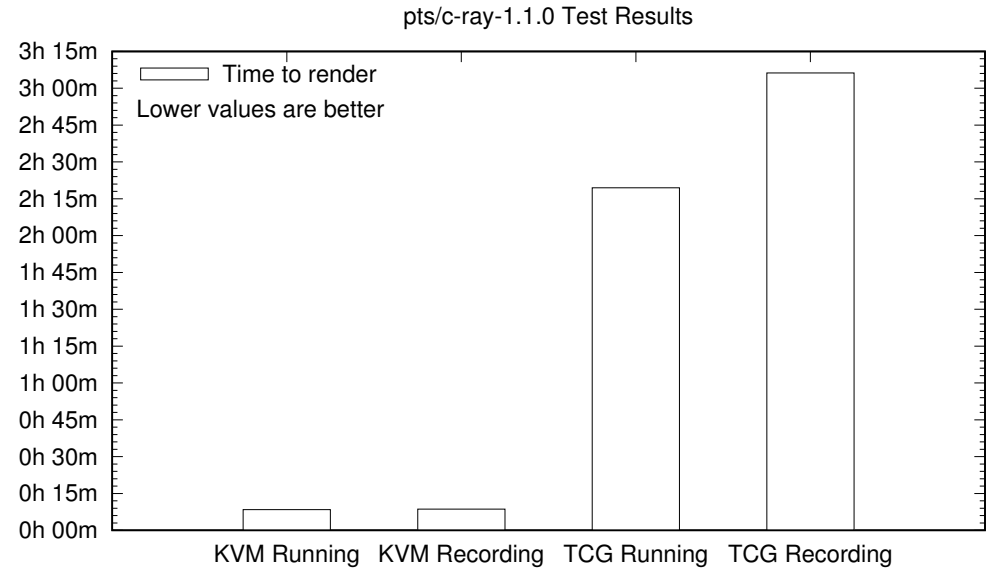
In TCG mode, on the other hand, no hardware acceleration is used, leading to a very high test runtime. Enabling recording in TCG mode causes the test to take 34 % longer. As with the Linux kernel benchmark, the greatly increased size of the TCG recording compared to the KVM recording is not due to events triggered by the test itself, but due to the background activity of the operating system caused by the long runtime of the test.

6.4.4 CacheBench

CacheBench is a benchmark that tests the performance of the RAM and cache of the system. The test runs for a fixed time and measures the read and write rates attained during that time in megabytes per second. Figure 6.5 shows the results of our testing.

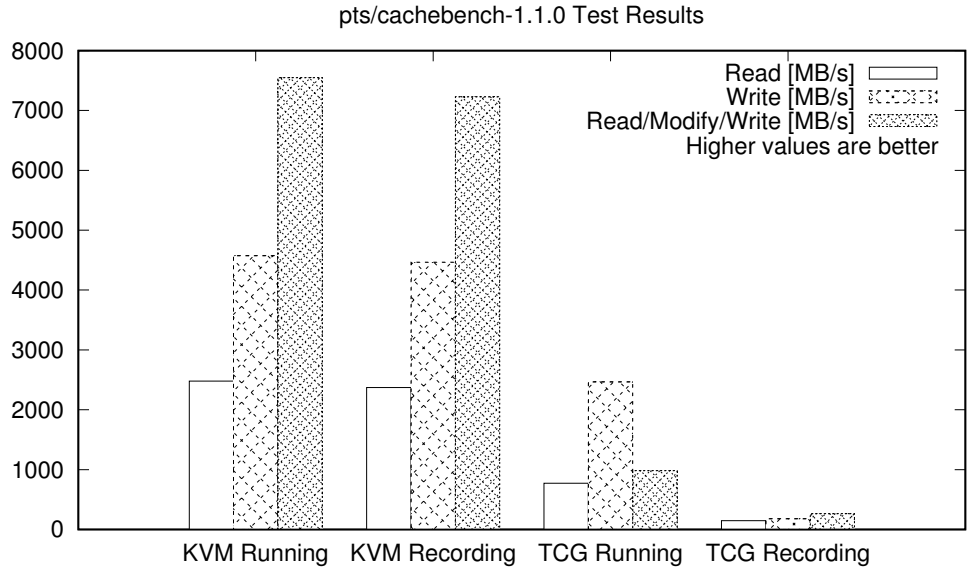
Like the C-Ray benchmark, this test makes little use of I/O because it operates on the guest RAM only, which is directly acessible when executing the virtual machine through KVM. Comparing the transfer rates obtained through the benchmark with other ARMv7 systems on OpenBenchmarking.org showed that they were close to the performance one would expect when executing natively on the CPU. With very few events to be recorded, the resulting recording is very small, with just 70 MB amounting to the execution of the test in either case. Again, the recording overhead when using KVM was about 1 %.

The speeds attained through TCG were much lower due to the many layers of indirections caused by software emulation. Being the only fixed-length benchmark among those we performed, it is worth noting that in this case, the total number of instructions executed in the KVM recordings was 22 times as high as the instruction in the TCG recording; a difference that was not observed in the other benchmarks.



	KVM		TCG	
	Running	Recording	Running	Recording
Test result	8m 23s	8m 36s	2h 19m	3h 06m
Test runtime	8m 38s	8m 53s	2h 19m	3h 06m
Recording size		1.84 GB		4.13 GB
Test only		0.13 GB		2.31 GB
Compressed size				
gzip		88.3 %		87.3 %
xz		5.6 %		6.4 %
Replay runtime		–		3h 16m
Instruction count		$7.3 \cdot 10^{11}$		$7.4 \cdot 10^{11}$
Replayed		$5.7 \cdot 10^{9}$		$7.4 \cdot 10^{11}$
No. of resyncs		9		0

Figure 6.4: Benchmark results from running the C-Ray raytracing software.



		KVM		TCG	
		Running	Recording	Running	Recording
Test result					
Read	[MB/s]	2478.5	2370.3	773.0	148.0
Write	[MB/s]	4572.9	4466.5	2463.5	177.4
R/M/W	[MB/s]	7548.0	7226.0	982.7	263.6
Test runtime		6m 19s	6m 21s	6m 20s	6m 23s
Recording size			1.76 GB		1.80 GB
Test only			0.07 GB		0.07 GB
Compressed size					
gzip			88.7 %		89.4 %
xz			5.5 %		5.0 %
Replay runtime			–		8m 12s
Instruction count			$8.8 \cdot 10^{11}$		$4.0 \cdot 10^{10}$
Replayed			$5.6 \cdot 10^{9}$		$4.0 \cdot 10^{10}$
No. of resyncs			9		0

Figure 6.5: Benchmark results from running CacheBench.

6.5 Store Exclusive

All recordings we performed using TCG could be replayed successfully, without any errors in the landmarks, suggesting that we recorded all the non-deterministic data that is required for replay; at least when using the same QEMU software for replay that was used during the recording.[1]

However, when performing a recording using KVM, there is still the unresolved issue of the *Store Exclusive* failures, which occur non-deterministically and thus cannot be replayed accurately. In Section 5.5, we have therefore made the assumption that those Store Exclusive operations are benign in that they do not have a lasting influence on the virtual machine operation, and thus that the resulting differences in the instruction count may be corrected later.

While this assumption cannot be made about arbitrary user code running in the virtual machine—the `STREX` instruction is non-privileged—it has held up in all code locations that we have checked manually. Figure 6.6 shows four example uses of the instruction.

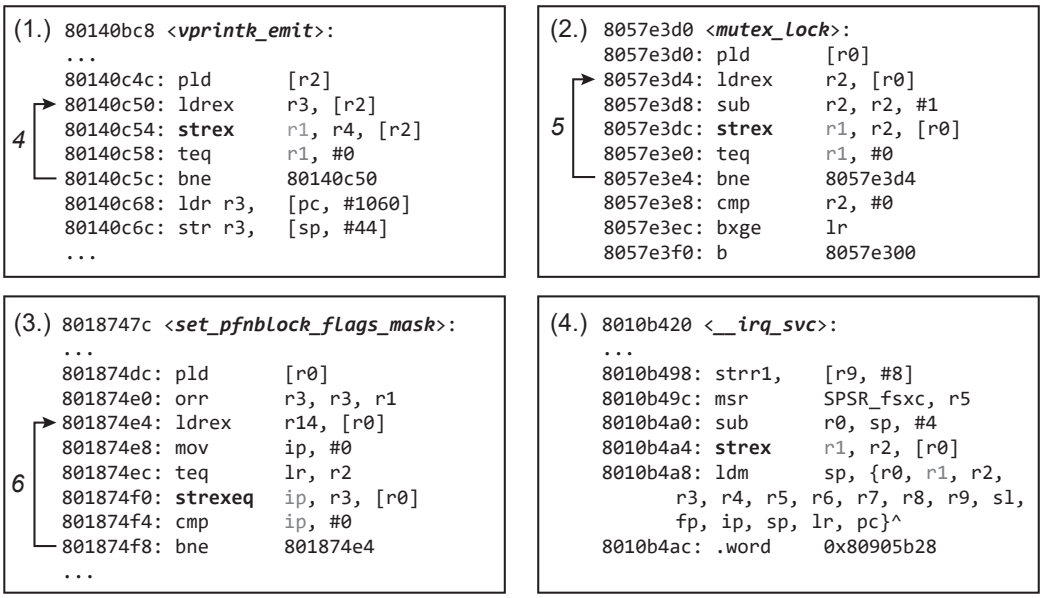


Figure 6.6: Example uses of `STREX` in the Linux 4.8 kernel, showing different sizes of the retry loop. The first register operand of the instruction receives the failure status. Example (4.) is part of a context switch, in which the exclusive monitor is only cleared without caring whether the operation has succeeded.

[1] It is possible that there are other unidentified sources of non-determinism, which are masked by performing homogeneous record and replay on exactly the same emulator platform and would emerge when attempting replay in another virtual machine implementation.

There are different patterns in which the STREX instruction can be used: to implement an atomic store, without using the previously stored data at that location (examples 1 and 3), an atomic arithmetic operation (such as decrementing, used in example 2), and—as required when performing a context switch—to clear any pending exclusive Load-Store pair (example 4).[2] Depending on the operation that is to be performed, the loop which retries the operation on failure comprises between 4 and 6 instructions.

In all cases we have examined, the retry loop is structured in a way such that upon exiting the loop, the result register of the Store Exclusive operation is 0; thus, any failures of the operation have no long-term effects on the virtual machine state. However, lacking a comprehensive analysis, we cannot prove this claim; and software could purposefully violate our assumption in order to avoid being analyzed in the record and replay system.

6.6 Landmark Corrections

We have implemented a mechanism for correcting a shifted landmark, in order to work around the non-determinism introduced by STREX, but have yet to show how it affects the replay.

There are two different types of landmark corrections that can occur in our system, depending on the event that is used for resynchronization: Coprocessor and MMIO read events allow for accurately determining the correct insertion point because the replayed system generates a corresponding event synchronously, whereas an interrupt event with an inaccurate instruction count must be adjusted through single-stepping.

In the recordings that we obtained when performing the benchmarks, we could typically replay the first 5 billion instructions correctly when landmark resynchronization was enabled; this was enough to see the Linux system boot up to the point at which the init process is launched. After that point, execution soon diverged, mostly because of a coprocessor access that hat not been recorded (see Section 5.1.3) or because an asynchronous landmark had been injected too early.

Figure 6.7 is a histogram of how many resynchronization events have occurred among the 64 billion successfully replayed instructions across the KVM benchmark recordings, grouped by the respective number of instructions by which the instruction counter had to be adjusted.

The largest number of resynchronization events occurred with an offset of 30 instructions. This number is not coincidental, as it was the value of our resynchro-

[2] There is a special instruction, CLREX, reserved for this purpose, however its implementation is broken in some revisions of the Cortex-A15.

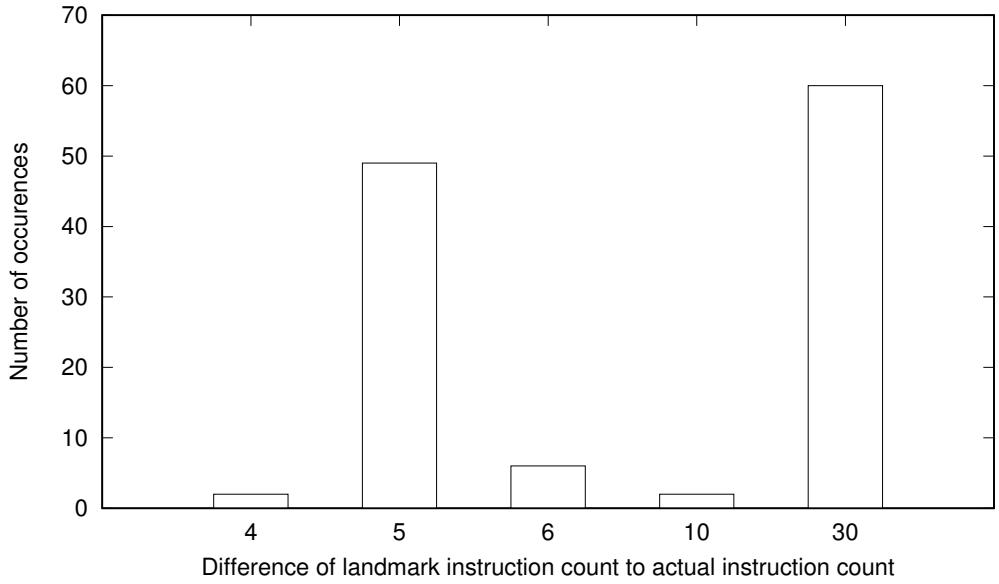


Figure 6.7: Occurrences of landmark resynchronization events across the successfully replayed parts of all KVM benchmarks.

nization window we had experimentally determined beforehand.[3] All instruction count adjustments with an offset of 30 were in fact false positives of our landmark resynchronization mechanism, inserting an interrupt too early. These false positives do not always cause errors in the replay, since they often occur within some kind of polling loop where the exact number of iterations does not actually matter.

Among the other resynchronization events, we can recognize the different sizes of the Store Exclusive retry loop, as previously shown in Figure 6.6. Being the most common variant, the loops with 5 instructions account for most of the synchronous landmark resynchronizations, as well as multiples thereof—since there may be many thousands of instructions between two landmarks, the errors in the instruction count can accumulate.

6.7 Log Size

The different benchmarks generate log files of very distinct sizes. CPU intensive benchmarks like CacheBench and C-Ray generate very few events that must be stored, whereas the Linux kernel build and the Apache benchmark quickly fill

[3] As explained in Section 5.5, there is no *correct* value for this window size due to the non-deterministic nature of the Store Exclusive failures; we chose this value because it worked well in practice.

the log file due to their high I/O activity. Table 6.8 shows statistics about the recordings obtained from the benchmarks and how well they can be compressed.

Benchmark	Runtime	Size	Rate	gzip	xz
apache	1h 03m	9.41 GB	2.49 MB/s	86.7 %	6.6 %
apache-tmpfs	52m 23s	8.54 GB	2.72 MB/s	86.8 %	6.6 %
build-linux-kernel	3h 37m	13.33 GB	1.02 MB/s	85.7 %	7.5 %
c-ray	8m 53s	0.13 GB	0.25 MB/s	88.3 %	5.6 %
cachebench	6m 21s	0.07 GB	0.18 MB/s	88.7 %	5.5 %

Figure 6.8: Log size and test runtime, averaged across all KVM recording runs. The runtime and sizes indicated comprise the test run only, excluding the virtual machine startup and shutdown. The gzip and xz ratios indicate the size of the entire compressed log file (including startup and shutdown) relative to its uncompressed size.

We attribute the rapid log file growth to the lack of DMA support in QEMU's `vexpress-a15` machine, leading to a large number of programmed I/O (PIO) requests which, being carried out through memory-mapped I/O interfaces, generate many recorded events. Since all data that is read from the SD card is thus stored in the replay log in the form of these MMIO events, the card image can be detached from the machine during replay. Nevertheless, storing the same data in larger blocks in the form of DMA events would allow for much smaller overheads.

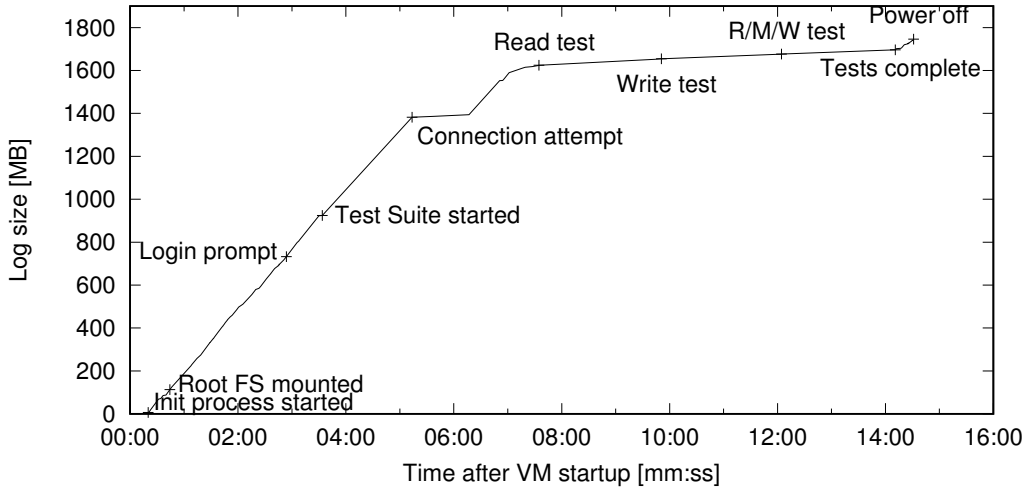


Figure 6.9: Size of a logfile throughout startup, execution and shutdown of a CacheBench run, obtained through KVM.

Figure 6.9 shows the log file growth during the recording of a CacheBench run, including the system startup and shutdown phase. A brief phase at the very beginning can be observed during which the kernel performs low-level initialization, generating very little I/O. As the init process takes over and the operating system boots, the log file grows at a steady rate, suggesting that it is busy performing a lot of I/O. There is a plateau around the 5 minute mark where the test suite attempts to connect to the Phoronix servers, but fails because the network is unavailable; in this phase, the system is idle for a while, causing few I/O events. The tests themselves, again, generate very little I/O because they spend their time accessing the virtual machine RAM, which is mapped directly into the guest's address space.

6.8 Discussion

In this section, we have demonstrated that our implementation can successfully record and replay the execution of a TCG-based virtual machine. Although replay did not succeed in the heterogeneous case from KVM to TCG, we have been able to show the performance of the system to evaluate whether the idea of record and replay is feasible on the platform.

While recording using KVM exhibits high runtime overheads of up to 40 %, it is still in all cases faster than doing so through the use of TCG. Furthermore, we believe that implementing support for DMA or virtio would dramatically reduce the amount of data that must be stored in the log file, while at the same time offering better performance through fewer virtual machine exits.

The key problem which remains as of yet unsolved is that of the Store Exclusive operation; the only instruction we have so far identified to exhibit non-deterministic behavior, which poses a major issue for record and replay. It may, however, be possible that the non-deterministic behavior is specific to the Cortex-A15 processor core, and that other implementations do not behave in this way.

The landmark resynchronization mechanism we have demonstrated is only a stopgap solution to fixing the issues surrounding Store Exclusive operations. There are better ways to approach this problem, e.g., through the use of checkpointing to simulate two different paths of execution based on different outcomes of the Store Exclusive operation.

Chapter 7

Conclusion

The objective of our work has been to evaluate the feasibility of heterogeneous record and replay on the ARM architecture: The non-deterministic inputs that a virtual machine receives—such as interrupts, network packets, disk data—should be recorded such that the execution of the machine can later be replayed accurately, instruction by instruction. Through the use of hardware virtualization, high performance recording should be possible with little overhead. Such systems already exist for x86 [39] [16], but to our knowledge, such a deterministic replay scheme had not yet been attempted before for the ARM architecture.

Our intention was to demonstrate that such a heterogeneous system can be implemented on the widely-used ARMv7 version of the architecture, and that using the hardware virtualization extensions, it is possible to perform recordings at acceptable speeds, allowing for debugging and tracing an entire system at the instruction set architecture level. To that end, we have first analyzed the ARM architecture, identifying the architectural events that require recording, and presented a generic design for performing record and replay specifically on ARM. The performance monitoring unit of the Cortex-A15 processor core was shown to provide highly accurate instruction counters, which can be used as landmarks to replay events at the correct time.

We have implemented a record and replay system based on the QEMU virtual machine software. The system was shown to perform fully accurate replaying in the homogeneous case where the Tiny Code Generator (TCG) is used for emulating a central processing unit (CPU). A heterogeneous recording component was implemented using the Linux Kernel-based Virtual Machine (KVM) hypervisor, which allows for faster recording when hardware virtualization extensions are available on an ARMv7 host machine.

During the implementation of the system, it became evident that there is a certain range of memory synchronization operations, called *Store Exclusive*, which behaves non-deterministically in hardware and cannot be trapped by the hyper-

visor. These instructions have caused major problems during replay, requiring careful adjustment of the landmarks and thus far preventing a successful replay.

Despite not being able to fully replay a recording obtained through the hardware virtualization extensions, we have obtained measurements to evaluate if the performance of such a system would be acceptable. While we had hoped to enable low-overhead recording, the overheads we observed were in the range of a 40 % slowdown compared to executing a KVM virtual machine without recording. We attribute the high overheads to a lack of DMA support in the particular virtual machine configuration we used. Nevertheless, even with recording enabled, execution of a KVM machine was faster than simulating it using TCG on a high-end x86 machine.

7.1 Future Work

The non-deterministic behavior of the `STREX` instruction has caused us major problems, and we do not believe that there is an easy solution to accurately replay it. There may be other CPU features available on ARM, such as CoreSight, which might be of use to work around the problem from the recording side.

It would be possible to perform more complex correction mechanisms in the replaying system, such as to create a checkpoint at each `STREX` instruction in order to attempt simulating both possible return values. This would be a safer approach than to attempt fixing the instruction count later, but was beyond the scope of our work, which was to evaluate the general feasibility of recording.

Our implementation is not yet able to accurately capture all coprocessor events; there is still a problem in capturing two coprocessor instructions in rapid succession, but we believe that this issue could be resolved. Currently, we disable the guest machine's access to the debug registers. It should be evaluated whether it is possible to restore the access without preventing instruction emulation in the hypervisor.

We have focused our efforts on the 32-bit ARMv7 architecture, while at the moment, the 64-bit successor architecture ARMv8 is gaining traction. We have not yet looked into the changes that this update brings to the platform.

Bibliography

[1] KQEMU on the QEMU wiki. http://wiki.qemu.org/Documentation/KQemu.

[2] Linux 2.6.20 release notes. https://kernelnewbies.org/Linux_2_6_20#head-bca4fe7ffe454321118a470387c2be543ee51754, 2007.

[3] C-Ray raytracing benchmark. http://www.sgidepot.co.uk/c-ray.html, 2014.

[4] Intel Xeon processor E5-2618L v3 specifications. https://ark.intel.com/products/83351/Intel-Xeon-Processor-E5-2618L-v3-20M-Cache-2_30-GHz, 2014.

[5] KVM on XU3 with Kernel v4.3. http://forum.odroid.com/viewtopic.php?f=99&t=17015, 2015.

[6] Odroid-XU3 product page. http://www.hardkernel.com/main/products/prdt_info.php?g_code=G140448267127, 2015.

[7] Nintendo: Bug bounty program. https://hackerone.com/nintendo, 2016.

[8] Linux 4.8 KVM API documentation. https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/tree/Documentation/virtual/kvm/api.txt?h=linux-4.8.y, 2017.

[9] Phoronix Test Suite: Open-source, automatic benchmarking. http://www.phoronix-test-suite.com/, 2017.

[10] ARM. *ARM Generic Interrupt Controller Architecture Specification*. Architecture version 2.0 edition, 2013.

[11] ARM. *ARM Architecture Reference Manual*. ARMv7-A and ARMv7-R edition, 2014.

[12] Murtaza Basrai and Peter M. Chen. Cooperative ReVirt: adapting message logging for intrusion analysis. *University of Michigan CSE-TR-504-04*, 2004.

[13] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[14] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 14(1):80–107, 1996.

[15] Anton Burtsev. *Deterministic systems analysis*. PhD thesis, The University of Utah, 2013.

[16] Jim Chow, Tal Garfinkel, and Peter M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX 2008 Annual Technical Conference*, pages 1–14, 2008.

[17] Jim Chow, Dominic Lucchetti, Tal Garfinkel, Geoffrey Lefebvre, Ryan Gardner, Joshua Mason, Sam Small, and Peter M. Chen. Multi-stage replay with Crosscut. In *ACM Sigplan Notices*, volume 45, pages 13–24. ACM, 2010.

[18] Christoffer Dall and Jason Nieh. KVM/ARM: the design and implementation of the Linux ARM hypervisor. In *ACM SIGPLAN Notices*, volume 49, pages 333–348. ACM, 2014.

[19] 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. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 66–71. ACM, 2006.

[20] Brendan Dolan-Gavitt, Tim Leek, Josh Hodosh, and Wenke Lee. Tappan Zee (North) Bridge: Mining memory accesses for introspection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 839–850. ACM, 2013.

[21] Pavel Dovgalyuk. Deterministic replay of system's execution with multi-target QEMU simulator for dynamic analysis and reverse debugging. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 553–556. IEEE, 2012.

[22] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review*, 36(SI):211–224, 2002.

[23] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, pages 121–130, New York, NY, USA, 2008. ACM. `http://doi.acm.org/10.1145/1346256.1346273`.

[24] Peter Ferrie. Attacks on virtual machine emulators. 2008.

[25] Rohan Garg. DéjàVu: Live record-replay of virtual machines for malware analysis.

[26] Matthew Gillespie. Best practices for paravirtualization enhancements from Intel Virtualization Technology: EPT and VT-d. `https://software.intel.com/en-us/articles/best-practices-for-paravirtualization-enhancements-from-intel-virtualization-technology-ept-and-vt-d`, 2009.

[27] Robert P. Goldberg. Survey of virtual machine research. *Computer*, 7(6):34–45, 1974.

[28] Ekaterina Itskova. Echo: A deterministic record/replay framework for debugging multithreaded applications. 2006.

[29] Kurt E. Kiefer and Louise E. Moser. Replay debugging of non-deterministic executions in the Kernel-based Virtual Machine. *Software: Practice and Experience*, 43(11):1261–1281, 2013.

[30] Markus Levy. The history of the ARM architecture: From inception to IPO.

[31] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel Virtualization Technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3), 2006.

[32] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 15–27. ACM, 2006.

[33] Marc Rittinghaus, Thorsten Groeninger, and Frank Bellosa. Simutrace: A toolkit for full system memory tracing. White paper, Karlsruhe Institute of Technology (KIT), Operating Systems Group, May 2015.

[34] Marc Rittinghaus, Konrad Miller, Marius Hillenbrand, and Frank Bellosa. SimuBoost: Scalable parallelization of functional system simulation. In *Proceedings of the 11th International Workshop on Dynamic Analysis (WODA 2013)*, Houston, Texas, March 16 2013.

[35] Daniel J. Scales, Mike Nelson, and Ganesh Venkitachalam. The design of a practical system for fault-tolerant virtual machines. *ACM SIGOPS Operating Systems Review*, 44(4):30–39, 2010.

[36] James E. Smith and Ravi Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, 2005.

[37] Vincent M. Weaver, Dan Terpstra, and Shirley Moore. Non-determinism and overcount on modern hardware performance counter implementations. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 215–224. IEEE, 2013.

[38] Min Xu, Vyacheslav Malyugin, Jeffrey Sheldon, Ganesh Venkitachalam, and Boris Weissman. ReTrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the Third Annual Workshop on Modeling, Benchmarking and Simulation (MoBS 2007)*, 2007.

[39] Lok-Kwong Yan, Manjukumar Jayachandra, Mu Zhang, and Heng Yin. V2E: combining hardware virtualization and software emulation for transparent and extensible malware analysis. *ACM Sigplan Notices*, 47(7):227–238, 2012.