Karlsruher Institut für Technologie

Accurate Record & Replay of x86 MMU Behavior for SimuBoost

Masterarbeit
von

Benedikt Morbach

an der Fakultät für Informatik

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

Bearbeitungszeit: 14. März 2018 – 13. September 2018

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, den 13. September 2018

Abstract

In order to speed up record and replay of full virtual machines systems like *SimuBoost* employ hardware-assisted virtualization to limit the impact of the recording process on the virtual system's performance. Like all record and replay systems this requires accurate logging of non-deterministic events that occur during the recording and injection of these events during the replay. The deterministic instructions between these events are being replayed directly.

However, current record and replay systems do not account for non-deterministic events that can be introduced by the x86 memory management unit (MMU). The unpredictable behavior of the translation lookaside buffer (TLB) and its potential incoherency with the page tables in RAM can cause the MMU to either set accessed and dirty bits in the page tables or skip this step depending on the TLB's contents. When the recorded system reads the state of these bits, the replay can diverge from the recorded execution if the TLB's contents or its behavior differ between the two, which is almost unavoidable.

We propose a way to record the changes the MMU makes to the status bits in the guest's page tables and implement it as part of the SimuBoost system. We achieve this by protecting the page tables in using the extended page table (EPT) mechanism on modern Intel CPUs and emulating these instructions using the emulation facilities for x86 code within the KVM Linux module. We implement support for replaying these events in the QEMU fork QSIMU that SimuBoost utilizes for heterogeneous replay.

Our results show that this approach can be a feasible solution. While we observe a 350% slowdown during recording for worst-case micro-benchmarks, the impact on some real-world applications is only 50% and CPU intensive workloads as well as the replay of any of the workload are not affected at all. This system is also able to successfully replay workloads the existing system consistently failed to replay before.

Deutschsprachige Zusammenfassung

Systeme zur Ausführungswiederholung (engl. *record and replay*) haben das Potential ein wichtiges Werkzeug zur Fehlersuche in komplexen Softwaresystemen zu werden. Anstatt ein Programm mehrmals ausführen zu müssen um einen Fehler zu reproduzieren und schrittweise ein Areal im Programmcode einzugrenzen das als Fehlerursache vermutet wird reicht es, eine einzige Ausführung des Programms aufzuzeichnen. Anschließend kann diese beliebig oft wiedergegeben und sogar zurückgespult werden um so von den Symptomen des Fehlers zu dessen Ursache zu finden.

Aufgrund der zunehmenden Verbreitung von virtuellen Maschinen in vielen Anwendungsgebieten, allen voran im Serverbereich, wurden diverse Systeme vorgeschlagen die dieses Prinzip der Ausführungswiederholung auf die Maschinenbefehlsebene übertragen und somit das Aufzeichnen und Wiedergeben eines gesamten virtualisierten Systems ermöglichen. Dies ist nicht nur ein hilfreich im Bereich der Fehlersuche, sondern erlaubt auch die forensische Analyse von Schadsoftware nachdem ein solcher Befall entdeckt wurde da das System zur Ausführungswiederholung nicht von der kompromittierten virtuellen Maschine beeinflusst werden kann.

Um Systemressourcen zu schonen vermeiden Systeme zur Ausführungswiederholung es nach der Ausführung jedes Befehls eine vollständige Kopie der virtuellen Maschine zu erstellen. Stattdessen machen sie sich die Tatsache zu Nutze, dass die Ergebnisse der meisten Maschinenbefehle in deterministischer Weise von ihren Eingaben abhängen. Nur der initiale Zustand und solche Ereignisse die inhärent nichtdeterministisch im Zeitpunkt ihres Eintretens oder ihrem Inhalt sind, wie z.B. Unterbrechungen, müssen aufgezeichnet werden. Die restlichen Stadien des aufgezeichneten Systems zwischen diesen Ereignissen können jedoch wiederhergestellt werden indem die deterministischen Befehle auf den vorliegenden Eingabedaten berechnet werden.

Um die Leistungseinbußen durch die Ausführungsaufzeichnung der virtuellen

Maschine in einem akzeptablen Rahmen zu halten verwenden Systeme wie SimuBoost [1] die Hardwareerweiterungen zur hardwareunterstützten Virtualisierung die in die von den meisten aktuellen Systemen unterstützt werden und führt einen Großteil der Befehle des aufzuzeichnenden Systems direkt auf dem Gastgebersystem aus. Dies führt jedoch zu einer neuen Klasse von Problemen. So verwendet die x86 Architektur die von den meisten PCs und Servern verwendet wird verwendet eine Speicherverwaltungseinheit welche die linearen Adressen, die von Software verwendet und referenziert werden, in physische Speicherblöcke im Direktzugriffsspeicher übersetzt. Während dieses Vorgangs aktualisiert die Speicherverwaltungseinheit einige Statusinformationen zu der jeweiligen linearen Adresse in den Seitentabellen, die sie für diese Übersetzung verwendet. Da dieser Übersetzungsvorgang relativ zeitaufwändig ist werden diese Übersetzungen im Übersetzungspuffer zwischengespeichert um Zugriffe auf den Direktzugriffsspeicher zu beschleunigen.

Allerdings ist das Verhalten dieses Übersetzungspuffers bewusst nicht spezifiziert und die Dokumentation verweist explizit darauf, dass er seinen Inhalt jederzeit unaufgefordert leeren kann. Dies hat zur Folge, dass die Übersetzung einer linearen Adresse durch die Speicherverwaltungseinheit entweder die Statusinformationen in den Seitentabellen aktualisiert oder nicht, abhängig davon ob die entsprechende Übersetzung im Übersetzungspuffers zwischengespeichert ist. Wenn dieser Fall während der Aufzeichnung einer virtuellen Maschine auftritt kann dies dazu führen, dass der Ausführungspfad während der Ausführungswiedergabe von dem der ursprünglich aufgezeichneten Ausführung divergiert, wenn das Gastbetriebssystem den nichtdeterministischen Inhalt der Seitentabellen liest und abhängig davon einen Ausführungspfad wählt. Dies führt dazu, dass die Ausführungswiederholung letztendlich fehlschlägt, da die aufgezeichneten nichtdeterministischen Ereignisse nicht mehr zum korrekten Zeitpunkt wiedergegeben werden können, da dieser nur auf dem ursprünglichen Ausführungspfad definiert war, was zur Folge hat, dass auch der deterministische Zustand zwischen diesen Ereignissen nicht korrekt wiederhergestellt werden kann.

Das Ziel dieser Arbeit ist es ein System zu entwickeln, welches eine Ausführungswiederholung inklusive des im vorherigen Absatz beschriebenen Nichtdeterminismus erlaubt ohne vollständig auf die Leistungssteigerung durch die Verwendung von hardwareunterstützter Virtualisierung während der Aufzeichnung verzichten zu müssen und die Leistung dieses Systems sowohl während der Ausführungsaufzeichnung als auch der Ausführungswiederholung zu evaluieren.

Um dieses Ziel zu erreichen werden wir die Ausführungskomponente des SimuBoost Systems, die als Teil des *KVM* Moduls im Linux Betriebssystemkern implementiert ist, erweitern um jegliche Änderung an den Seitentabellen des Gastsystems innerhalb der virtuellen Maschine zu überwachen, gleichwohl ob diese

durch das Gastbetriebssystem oder die Speicherverwaltungseinheit vorgenommen werden. Wir erreichen dies indem wir die Hardwarevirtualisierungstechnologie der geschachtelten Seitentabelle verwenden, die als *EPT* (engl. *Extended Page Table*) in aktuellen Intel x86_64 Prozessoren implementiert ist, um die Speicherbereiche die die Seitentabellen des Gastsystems enthalten als nur-lesbar markieren und jeder Befehl, der auf diese Speicherbereiche schreibend zugreift mithilfe der Emulationssoftware für x86 Befehle, die Teil des KVM Moduls ist, emulieren. Des Weiteren modifizieren wir *QSIMU*, eine Abspaltung des *QEMU* Emulators die SimuBoost zur Ausführung der Ausführungswiederholung verwendet, um diese zusätzlichen Informationen die während der Ausführungsaufzeichnung gesammelt wurden zu verwerten und die gleichen Änderungen an den Seitentabellen auch während der Ausführungswiederholung vorzunehmen.

Unsere Resultate zeigen, dass dieser Ansatz eine vielversprechende Lösung des beschriebenen Problems bietet. Die nötigen Modifikationen am bestehenden Programmcode waren von relativ geringem Umfang und unser modifiziertes System ist in der Lage auch solche Ausführungen zu wiederholen für die dies zuvor unmöglich war. Während wir keinen messbaren Einfluss auf die Laufzeit der Ausführungswiederholung feststellen konnten, variiert der Einfluss auf die Laufzeit der Ausführungsaufzeichnung stark in Abhängigkeit von der Zusammensetzung der Last der aufgezeichneten Ausführung. Berechnungsintensive Ausführungen verlaufen mit der selben Geschwindigkeit wie bei einer Aufzeichnung ohne Unterstützung für nichtdeterministische Speicherverwaltungsereignisse. In einem konstruierten Minimalbeispiel für einen schlechtestmöglichen Fall konnten wir jedoch eine Verlangsamung um 350% feststellen. Für reale Anwendungsbeispiele liegt der Leistungsabfall zwischen diesen Extremen, tendiert jedoch zum besseren Ergebnis.

Contents

Chapter 1

Introduction

Record and replay systems aim to provide an important tool in debugging by allowing the user to record a single faulty execution of a program and subsequently replaying this execution accurately without the replay being affected by any external inputs. This causes any bug that is being tracked down to occur in exactly the same way every time and greatly simplifies the debugging experience. It can even allow the debugger to inspect an earlier state in the programs execution, working backwards in time from the symptoms of a bug to its cause.

Due to the prevalence of virtual machine software in today's computing environments, it has been proposed to apply the same principle of record and replay at the machine instruction level, allowing the recording and subsequent replay of a virtual machine. This can not only be useful for debugging but also as a tool in intrusion detection because the software that records the virtual machine will be unaffected if the virtual machine itself is compromised.

In order to avoid performing a full copy of the virtual machine after every instruction, record and replay systems exploit the fact that most machine instructions are deterministic in nature. Only events that are non-deterministic such as interrupts need to be recorded and the state of the recorded system between these events can be recreated by simple executing the deterministic instructions.

To provide acceptable performance when recording a full virtual machine, record and replay systems for virtual machines such as SimuBoost [1] make use of the hardware-assisted virtualization capabilities of modern processors and execute a majority of the recorded system's instructions directly on the host hardware. While this severely improves the performance of the recording, it also introduces new problems. The x86 architecture that is used by most personal computers and servers utilizes a memory management unit (MMU) to translate linear memory addresses that are used by the software running on the system into physical locations in RAM.

While doing this, it also updates some status information in the page tables that define these translations. Because this operation is relatively slow, the translation lookaside buffer (TLB) is used, which caches these translations to speed up RAM accesses.

However, the operation and contents of the TLB are non-deterministic, which can cause the status of the translations that the MMU manages to either get updated or not, depending on the presence of the given translation in the TLB. If this happens during a recording of a virtual machine, it can cause the corresponding replay to diverge from the code path that the recording took when the operating system in the virtual machine inspects this non-deterministic state in the page tables and chooses different actions depending on their contents. This will then cause the replay to fail when its state deviates from the recording and the following non-deterministic events cannot be lined up correctly.

Our goal in this work is to develop a system that allows a replay of a virtual machine that is faithful to the previously recorded execution, including the non-determinism caused by MMU changes to page tables depending on the TLB's state, while still reaping the benefits of hardware assisted virtualization and evaluate the impact that this has on the performance of the system during both recording and replay.

To achieve this goal, we extend the recording component of the SimuBoost system, which is implemented within the Linux kernel's KVM module to track any change that either the guest operating system in the virtual machine or the MMU makes to the page tables. We achieve this by using the *extended page table (EPT)* hardware-assisted virtualization extension in recent Intel x86_64 CPUs to mark the memory that contains the page tables as read-only and emulating any write to them using the emulation facilities present within KVM. We then modify QSIMU, the fork of the QEMU emulator that SimuBoost uses to perform the replay, to process this additional information gathered during the recording and recreate the same changes to the page tables during the replay.

Our work shows that this approach is a feasible way to record these events, with only small additions to the existing recording code being needed. We confirm that it allows the replay of workloads that would previously fail to replay correctly. While there is no measurable impact on the performance of the replay, our implementation causes varying amounts of slowdown to the execution during recording, with the magnitude of the slowdown greatly depending on the characteristics of the recorded workload. While computationally heavy workloads suffer no slowdown, the worst-case that we evaluated slows down by over 350%, with real world applications falling between these extremes.

Outline

In Chapter 2 we introduce the concepts of paging and the operation of the x86 MMU and TLB and explain more details on the operation of virtual machines, record and replay systems and various implementations of the latter. In Chapter 3 we analyze the origin and impact of the observed problem and the degree to which existing record and replay solution acknowledge or solve it. In Chapter 4 we present the design of our proposed solution and weigh it against alternatives that we considered. After this, we elaborate on some of the details of our implementation in Chapter 5 before evaluating its correctness and performance using multiple benchmarks in Chapter 6. Last but not least we summarize our work, offer a conclusion and suggest potential improvements for future work in Chapter 7.

Chapter 2

Background

In this chapter we will introduce the concepts and mechanisms that build the foundation on which our work builds and which are necessary to fully describe the problem we intend to solve and our proposed solution. First we give a short introduction to virtual memory and the Translation Lookaside Buffer. Next we introduce virtual machines and describe the differences between emulation and hardware-assisted virtualization. Lastly we describe how QEMU and KVM implement these concepts and how Record and Replay of virtual machines can be used to facilitate debugging applications and operating systems.

2.1 Virtual Memory

In order to provide isolation between different processes running on the same system and allow a single system to run more than one instance of a program, modern processors implement what is called paged virtual memory.[2] In this mode of operation the memory addresses that a program accesses are not treated as literal locations in physical memory but are instead treated as so called *linear addresses*. The MMU of the system translates these linear addresses into physical addresses which describe a specific location in physical memory which is then used by the given instruction. These translations — or mappings — of linear to physical addresses are different for each running process and are configured by the operating system's kernel. This prevents malicious or malfunctioning processes from accessing other processes' memory because the kernel only adds translations to physical memory locations that the process is allowed to access. It also avoids conflicts between different instances of the same program because while the addresses that are hard-coded in their binary are identical and thus

both instances intend to use the same memory addresses. However, by using paged virtual memory their mappings refer to different pages in RAM. The set of mappings that define the linear addresses that a program is able to use and which physical memory locations back these addresses is called the program's *address space*.

2.1.1 Page Tables

The translations from linear to physical addresses are described by a set of cascading tables in memory. We will now briefly describe the process by which the MMU uses these tables to translate linear to physical memory addresses on current x86_64 systems. While these tables are organized in levels and the tables on each level have a distinct name that can be seen in Figure 2.1 on page 9, we will refer to all of them as *page tables* unless otherwise noted because they are structured identically and the differences between them will not be of importance to our work.

The physical address of the first of these page tables is recorded in the processor's `CR3` register at all times. The MMU loads this table from memory and selects one of its entries using the first 9 bits of the linear address as the index. Each entry contains the physical address of the next table in the page table hierarchy as well as a set of status flags. The MMU then loads the next table from the physical memory address given in this entry and uses the next 9 bits of the linear address to select an entry in this table. On current x86_64 systems the translation of a single linear address uses up to 4 page tables each of which contain 512 entries. The last PTE contains the address to a physical memory location of a fixed size, called a *page frame*, and the remaining bits of the linear address are used as an offset into this page frame. The specifics of this process can vary depending on the configuration of the processor — e.g. in 32bit mode there are only two or three levels of page tables, they each contain 1024 32bit entries instead of 512 64bit entries and each step uses 10 bits of the linear address to select a PTE — but the general principle remains the same.

To allow the operating system to keep track of which regions of memory are in active use by a process the MMU maintains a set of status flags in each of the PTEs. Two of these flags will be of interest to our work, the *accessed* and *dirty* bits. The accessed bit is set by the MMU whenever it uses the corresponding PTE to translate a linear to a physical address but before it returns the resulting address. The dirty bit only exists in the entries of last level page table and is set by the MMU when the memory that the entry references is written to. Together these two bits allow the operating system to make an informed decision on which contents to evict from memory when the amount of available memory runs low. Data that

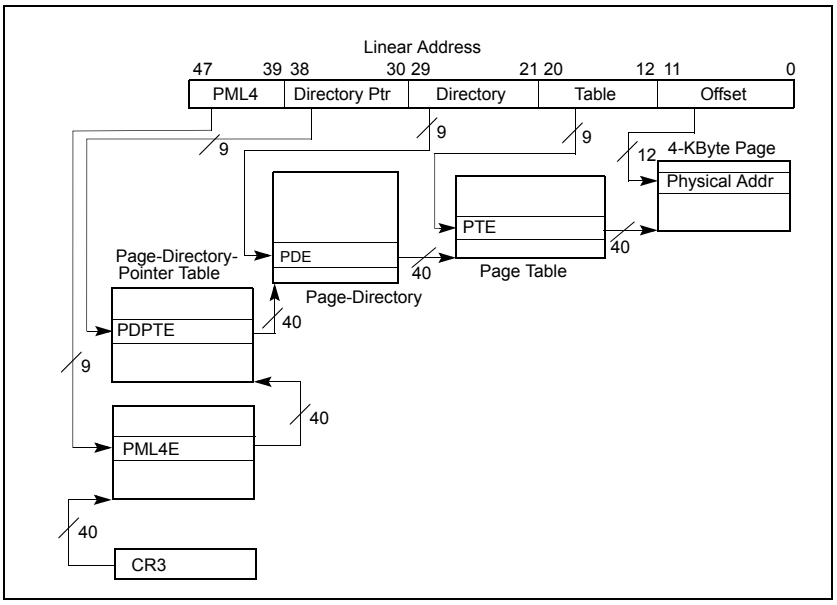

Figure 2.1: Linear Address Translation with 4-Level Paging.[2, p. 124]: Every 9 bit interval of the linear address selects an page table entry in the current page table and 40 bits of the 64 bit sized page table entry are used to locate the next level page table until the final 4 KiB sized page frame is located.

has been read from persistent storage, e.g. a file, and whose PTE does not have the dirty bit set can simply be dropped from memory. Only the information from where the data can be restored when the process tries to access it again needs to be recorded. On the other hand if the memory is dirty it has to be written to persistent storage before being dropped from memory. In both cases the accessed bit helps inform the decision which specific pages to evict from memory. (The operating will usually decide to keep data that has been accessed before under the assumption that it will be accessed again.)

Another status flag and arguably the most important one is the *present* bit which signifies if the page table entry is considered valid. If this bit is cleared the MMU will not use this entry to translate a linear address and will instead upon encountering it raise an exception, the so called *page fault*. This allows the operating system to overcommit memory by clearing the present bit when creating a page table for an application. As long as no linear address that uses the page table entry in questing is accessed the operating system does not need to allocate any memory to hold its contents or even load data from a hard disk to fill this memory. Only when the memory is first accessed will a page fault be raised, invoking the operating system's kernel which will either fill in the corresponding page table entry, allocate a page frame for this linear address and set the present bit to 1 or

terminate the application if it is not allowed to access memory at the given address.

When the operating system determines the currently running process has exhausted the amount of time that it was allowed to run, it determines which process should run instead. Once it has done this, it switches the active address space so that the new process is able to access its memory at the linear addresses that it expects. This is performed by simply writing the physical address of the top-level page table to the `CR3` register. This process is called a *context switch*. The same operation may take place when a process uses a system call to interact with the operating system kernel, except that the kernel's address space is activated, though this depends on the implementation of the kernel.

2.1.2 The Translation Lookaside Buffer

While paged virtual memory brings some important advantages as detailed above, a naive implementation would also come with a significant performance penalty. Accesses to memory are relatively slow compared to data that is already in a processor cache or even a register, with a single access requiring up to two orders of magnitude more time to complete[3] and for each access to memory, up to four additional memory accesses to page table entries have to be performed when paging is being used. This would be prohibitively expensive, especially because every instruction in a running program is also referred to by and loaded from a memory address. In order to avoid quintupling the time it takes to load each instruction and putting more pressure on the CPU caches by having to cache each individual page table entry, the *Translation Lookaside Buffer (TLB)* is used. The TLB caches translations from linear addresses to physical addresses. After the MMU has resolved a translation of a linear address, determined it to be a valid translation and set the accessed and dirty bits in the corresponding page table entries it creates a TLB entry for this linear address. The TLB entry contains the physical address that the translation points to as well as some of the status flags oft the page table entries that were used in the translation, indicating among other thins if the the linear address is read-only or writable, as well es the dirty bit of the last level page table entry.

At the time the TLB entry is created the linear address is implicitly readable because TLB entries are only created for valid translations from linear to physical addresses. Furthermore the state of the accessed bit is not cached explicitly by the TLB because each page table entry used by the corresponding translation already has its accessed bit set by the MMU at the time the TLB entry is created. While the MMU creates TLB entries automatically it is up to the operating system to ensure that they remain consistent with the contents of the page tables. For example when

removing an entry from a page table or marking an entry read-only, the MMU must be notified of this fact. If the operating system does not flush the relevant TLB entries the MMU might use the cached translation from the TLB for a future access to the linear address, potentially ignoring the changed permissions. Whether this happens or the changed permissions from the page tables are used depends on the size and exact behavior of the TLB which is generally implementation defined. [2]

2.2 Virtual Machines

Many computer system resources are underutilized because the applications that run on them are either idle for longer periods of time or only run on demand. However the isolation of different services on individual systems is often desired, for example for security or compatibility reasons. In order to solve this conflict *virtual machines (VMs)* [4] can used to allow these applications to run on the same physical machine while maintaining the required isolation between them. This is used extensively in cloud computing infrastructure to improve hardware utilization while at the same time providing isolation between the individual machines that might be rented out to different customers. A virtual machine allows an unmodified operating system to run as an application on another operating system while keeping it under the illusion that it is running on its own physical hardware. To do this a *hypervisor* or *virtual machine monitor (VMM)* implements the interface (i.e. I/O devices, memory, CPU,...) that a physical machine provides to the operating system while preventing individual virtual machines — also called *guest systems* — from influencing each other or the *host system*. The hypervisor runs the guest system as an unprivileged application and employs either emulation, hardware-virtualization or a combination of both to handle the privileged operations that the guest wants to perform. An example for such a privileged operation is setting and modifying the active page table hierarchy as that allows controlling which parts of memory the running application can access, as we explained earlier. We will now give a brief explanation of both emulation and hardware-assisted virtualization and their use cases.

2.2.1 Emulation

When the hypervisor *emulates* [4] the guest instructions it does not run them directly on the hardware but instead performs different operations while providing the same results to the guest as if its code had been run unmodified. There are multiple reasons why a hypervisor has to emulate some or even all of the guests

instructions. For example the guest might be compiled for a different instruction set, making it impossible to run it directly on the existing hardware. Another reason would be that the instruction can't be executed by the guest if the hypervisor wants to maintain the isolation between guests and host. For example in the Intel x86 instruction set some instructions will cause errors if they are not run in protection ring 0 and some instructions will return different results depending on which ring they are executed from. If the guest were to run one of these instructions and receive an error it would behave differently compared to running on a physical machine so the instruction has to be handled by the hypervisor. However the hypervisor can't run the guest's code in ring 0 because that would give the guest control over the physical system. Instead the hypervisor analyzes the guests code and performs an analogous action that yields the intended result for the guest.

To return to our earlier example, we will explain what happens when the guest wants to change the active page table hierarchy to allow a different application inside the guest to run. This instruction can't be executed as is because page tables are accessed by physical addresses but the physical addresses that are visible to the guest are part of the virtual machine that the hypervisor simulates so the MMU would be unable to find the page tables. To conform with the guests expectations the hypervisor has to inspect the page tables that the guest has constructed and map the guest's physical addresses referenced in the page tables to the actual physical memory on the host that it uses to provide the guest's memory. It then re-creates these page tables using the correct physical addresses and finally informs the MMU about these new page tables. This process is called *shadow paging* and the page tables that are allocated by the hypervisor are referred to as *shadow page tables*.[4] Additionally the hypervisor has to keep track of any additional changes the guest makes to its page tables, mirroring them to the shadow page tables and replace any instruction by the guest that queries the active page table hierarchy to ensure that the guest only ever sees the page tables that it constructed itself. This ensures that for the guest the resulting state of the system is indistinguishable from the state if it's original page tables were being used.

There are multiple approaches to implementing an emulator for a given instruction set. In any case the hypervisor has to parse the guest's instructions and replace those which can not be executed directly by (potentially multiple) native instructions that can be executed directly. If the binary was compiled for a different instruction set every instruction has to be replaced. This can either be done by an interpreter that translates each instruction one by one as it is about to be executed or by translating bigger blocks of the binary to native code at once which are then cached and executed as needed. However both approaches come at a significant performance cost due to the need to decode and translate each original instruction. In the example above a single write to a register to activate a new set of page tables turned

into the hypervisor reading all of the guests page tables from memory, transforming their contents and writing new page tables before performing this single write to a register. While some optimizations can be applied such as caching some of the generated page tables this has to occur on each context switch in the guest system resulting in a significant slowdown.

2.2.2 Hardware-Assisted Virtualization

In order to avoid this slowdown associated with emulation, many processors and chipsets now implement *hardware-assisted virtualization*. Hardware-assisted virtualization adds additional instructions to the processor's instruction set that allow the hypervisor to run a large portion of the guests code directly without emulation. To do this the Intel VT-x hardware-assisted virtualization for example adds two new modes to the processor, *VMX root mode* and *VMX non-root mode*. The hypervisor runs in VMX root mode and can configure the behavior of the VMX non-root mode which the processor enters before executing a virtual machine. In VMX non-root mode the guest's code can be executed directly and the guest is able to use all protection rings 0 – 3. However, when the guest tries to execute privileged instructions it will be suspended and the processor will exit into VMX root operation and supply the hypervisor with information on which instruction the guest tried to execute. The hypervisor is then able to emulate the instruction and allow the guest to resume execution. This scheme of implementing virtual machines is known as *trap and emulate*. Some privileged instructions can be executed by the guest directly without intervention by the hypervisor. For example the guest is allowed to enable and disable interrupts directly but this setting does not affect the host or other virtual machines because it is stored in a structure for the specific virtual machine when the processor exits non-root operation and restored when entering the same virtual machine again. The hypervisor can also configure which instructions cause the processor to suspend the guest which allows the hypervisor a fine grained control over the guests state and can be used for debugging purposes.

While hardware-assisted virtualization offers better performance than emulation it also has some limitations. In general it is only possible to run guest systems that are compiled for the instruction set that the CPU supports natively using hardware virtualization.

The Extended Page Table

While the implementation of hardware-assisted virtualization outlined above describes the initial support for this feature that was available on x86_64 CPUs, different levels of hardware-assisted virtualization have since been added which allow the guest to directly manage some resources that the hypervisor previously needed to emulate, such as the managing of the guest's page tables via shadow page tables.

To obviate the need for shadow paging newer x86_64 processors by both Intel and AMD implement *Extended* or *Nested Page Tables*, respectively. [5] Though the technologies differ in details the basic concept is identical and we will now describe the Extended Page Table (EPT) which we also used in our implementation. The EPT is another set of page tables that is configured by the hypervisor. However unlike the regular page tables it does not map from linear to physical addresses but instead maps from what the guest considers its physical address space to the actual hardware's physical addresses.

When the guest wants to access a physical memory address the EPT is a walked by the MMU exactly like the regular page tables but the indices in the individual tables are derived from the guest physical address instead of the linear address. When the EPT is enabled the guest is allowed to manage its own page tables without intervention from the hypervisor because the `CR3` register that stores the topmost page table's address is now part of the virtual machine's state that is saved and restored by the processor on transitions between VMX root and non-root mode and cannot affect other guests or the host. In order to translate a memory access to a linear address in VMX non-root mode the MMU will now first have to consult the EPT to translate the guest physical address of the top-most page table that is stored in the `CR3` register to a real machine physical address and be able to load the guest page table. Then it will find the address of the next level page table using the normal process described in Section 2.1.1. However this page table is again referenced by a guest physical address which has to be translated using the EPT. This process is repeated up to 4 times after which the guest physical address of the desired memory location has been determined and can be translated to a machine physical address using the EPT. In order to avoid the overhead of loading up to $4 \cdot 4$ page tables/EPT tables from memory the TLB also caches translations from (guest) linear addresses to machine physical addresses as well as translations from guest physical to machine physical addresses when the EPT is enabled.

If they are invalid for the requested access a page fault is raised directly with the guest otherwise the EPT is consulted. The entries of the EPT tables contain permission bits much like the regular page table and if the requested access is not allowed under the given permission (e.g. a write access where one of the

corresponding tables does not have the writable bit set or a read access where the present bit is not set) a VM exit occurs and an *ETP violation* which is the EPT equivalent of a page fault is raised with the hypervisor. The hypervisor is then able to modify the EPT to allow the access or emulate the instruction if it does not want to allow the guest direct access to the memory address or if the address is part of an memory-mapped I/O range of an emulated device.

This greatly improves performance because the hypervisor is not invoked for each context switch in the guest and as long as the location of the guest's memory in the physical address space remains constant (i.e. it is not swapped to disk or migrated to another physical machine) the EPT does not need to change. Later revisions of the EPT also include support for accessed and dirty bits within the EPT entries which allow the hypervisor to make better decisions on which parts of guest memory to write to persistent storage and remove from the EPT under memory pressure similar to how the accessed and dirty bits in the page tables are used by the operating system kernel.

2.3 QEMU/KVM

In this section we will give a high level overview of how QEMU/KVM implement the concepts described above to form a state of the art hypervisor under Linux. As the name implies the hypervisor is really separated into two parts, the *Kernel-based Virtual Machine (KVM)* which is part of the Linux kernel and QEMU which runs in user space.

The KVM kernel module [6] allows the Linux kernel to act as a hypervisor using hardware-assisted virtualization. It contains implementations for multiple architectures while exposing a common interface to the user space application managing the virtual machine. For example for the x86_64 architecture KVM is able to utilize both the AMD SVM as well as the Intel VMX hardware extensions and implements shadow paging as well as nested paging using EPT/NPT with either being used depending on processor support and the configuration of KVM and the virtual machine. It also supports emulation of the targeted instruction sets which is used in cases where direct execution is not possible. However, KVM does not implement any emulated devices. It is instead intended to be used by a user space application that implements the rest of a virtual machine.

QEMU [7] on the other hand implements a hosted virtual machine monitor, i.e. it runs as an application on top of a regular operating system while executing a virtual machine. QEMU handles the emulation of devices and includes among others support for storage, memory, input, graphics and network devices. These

emulated devices each run in their own thread with another thread simulating the CPU of the virtual machine. This thread runs the *Tiny Code Generator (TCG)* [8], a binary translator that allows QEMU to execute both user space programs that were compiled for a different architecture as if they were running natively as well as — as described here — to run a full virtual machine for a guest system. In both cases TCG translates contiguous blocks of emulated application's instructions to an intermediary language, which is then either translated to the host's instruction set and executed directly or — if no backend for the second translation step is implemented — run by a generic interpreter for the intermediary language. If the system supports hardware-assisted virtualization QEMU is also able to utilize this support to improve the performance of the virtual machine, replacing the use of the TCG. On Linux this support is provided by the aforementioned KVM kernel module and execution of the CPU thread is implemented by calling into this module. The CPU thread then blocks while the execution of the guest's code is handled by KVM until it is either interrupted because an interrupt from an emulated device needs to be handled or until KVM returns control to QEMU, for example to handle a memory-mapped I/O request for one of the emulated devices.

2.4 Record and Replay

Often computer programs exhibit bugs that are exceedingly hard to find and thus fix because they happen non-deterministically. This can for example happen due to race-conditions in multi-threaded programs. Depending on how rarely the conditions that cause the bug to trigger appear it can take many invocations of the program until one of them even exhibits the unwanted behavior. However, the most commonly used scheme of debugging is *cyclic debugging* [9] whereby the program is executed multiple times with each execution narrowing down the suspected area in which the bug is located using breakpoints to inspect the state of the program at certain points. Unfortunately the effectiveness of technique is greatly reduced when the bug in question is triggered non-deterministically because it takes longer to find an execution of the program in the invalid state and a single bug can manifest itself in multiple ways at a later point in the execution making it hard to narrow down the scope of the debugging process. Some bugs are even so called "heisenbugs" which either disappear or alter their behavior when the program is being debugged. This can for example be caused by the debugger altering the timing of some operations, making threads of the same program run at a different speed relative to each other which can change the probabilities for race-conditions to manifest themselves.

In order to solve this problem and make it easier or even possible to find this class of bugs, *Record and Replay* [10] techniques can be used. Instead of only

inspecting the state of the program at a given time and trying to deduce the cause of any inconsistencies the complete execution of the program is recorded and thus this exact execution can be replayed later in a deterministic way. This allows inspecting the state of the program can be inspected at any point in time and even work backwards across the timeline of the recording. By using Record and Replay a given bug only needs to be reproduced once and can then be analyzed in depth working backwards from the symptoms to the actual bug that caused them.

Because it would be prohibitively expensive to copy the entire step of the execution for each point in time, requiring both immense amounts of memory and slowing down the program enough to make it unusable, only non-deterministic events that affect the execution are recorded. The execution can then be replayed by injecting these events at the correct time and computing the deterministic operations in-between these non-deterministic events normally. Examples of such events are the time at which a network packet is received or the data that a read from a hard drive returns. It is sufficient to record these non-deterministic events because the replay can reconstruct the state of the program by combining them with the deterministic instructions.

While the same concept of Record and Replay can be applied to whole virtual machines to allow debugging of hard to reproduce bugs even withing the operating system, performance has historically suffered due to the use of emulation until the widespread implementation of hardware-virtualization. Because the virtual machine exposes a different interface to the operating system a different set of non-deterministic events needs to be recorded to ensure that the execution of the whole system can be replayed deterministically. Examples of these events include the timing of interrupts, the data returned from emulated devices (e.g. network interfaces) or the count of the system timer.

Different implementations of Record and Replay for Virtual Machines have been proposed, operating at different levels of abstraction. ReVirt [11] uses UMLinux[1] [13] to run a modified Linux guest system as an unprivileged application on a Linux host, replacing privileged hardware instructions with system calls into the host kernel. This approach is however not very portable due to the requirement to modify the guest kernel. ExecRecorder [14] on the other hand utilizes the Bochs simulator that emulates a full x86 system at the hardware level, though at a significant slowdown.

With the growing prevalence of hardware-assisted virtualization extensions in common x86 hardware there have been efforts to leverage these additional capabilities to lower the performance impact of recording a virtual machine execution increase

[1] Not to be confused with the similarly named User Mode Linux[12]

the compatibility with different guests. ReTrace [15] builds on the VMware Workstation's hypervisor and allows the replay to occur on a different machine from the recording. This allows the recorded system to continue operating normally without being affected by increased load from the replay. Building on top of this, Aftersight [16] uses QEMU to replay the execution, greatly increasing the amount of architectures the system can run on and allows the replay to run in parallel to the recorded execution to allow analyzing the replay, e.g. for intrusions via buffer overflows, while the system is running but without having to run these potentially slow analyses on the production system itself. The approach taken Crosscut [17] on the other hand allows transforming a VMware Workstation recording at machine instruction level into a replay at a different level of abstraction, for example allowing the user to replay a single process that has been identified of being of interest after the recording was created.

Other applications for record and replay have been proposed as well, with V2E leveraging heterogeneous record and replay like Aftersight though in this case the recording is implemented with hardware-assisted virtualization in KVM while the replay is performed in TEMU, a tool for dynamic binary analysis built on top of QEMU [18], to analyze malware without the malware being able to detect and evade this analysis and SMP-ReVirt [19] building on the Xen hypervisor to enable record and replay of multiprocessor virtual machines by guaranteeing the ordering of accesses to shared memory to stay consistent during replay.

Chapter 3

Analysis

It can be difficult to debug and fix hard to reproduce bugs using only cyclic debugging because the cause of the bug is generally not known and in some cases it is even difficult to impossible to reproduce the problem with a debugger attached to the program in question. So called Heisenbugs seem to actively avoid inspection because their occurrence are influenced by subtle differences in timing. Employing record and replay allows analyzing a single execution in depth without changing the timing of external events or needing to reproduce the bug in question in each debugging cycle. It has been proposed [11] to apply the same principle to whole virtual machines to allow easier debugging of such bugs in the operating system kernel or even monitor production systems, which often run in virtual machines today, via recording. This would allow debugging bugs without having to reproduce them even once and has also been suggested as an improvement to intrusion detection [11] because an attacker might be able to alter log files and other traces of his presence on the attacked system but would be unable to hide his presence in the recording of the virtual machine that exists on the host.

Despite these advantages the application of record and replay techniques to virtual machines has not been widely deployed yet and the solution that was commercially available as part of the VMware Workstation hypervisor [15] has even been removed in a later version[20]. It is clear that in order for record and replay of virtual machines to be useful in practice one of the most important concerns if not the most important concern is the accuracy of the replay. In order to keep the amount of data that needs to be recorded as low as possible, all current implementations of record and replay only log non-deterministic events that could otherwise not be reproduced only from the current state of the system at the time of replay. There is however a class of non-deterministic events that can be caused by incoherencies between the TLB of x86 CPUs and the state of page tables in memory that none of the implementations we mentioned in Section 2.4 cover.

In this chapter we will explain the cause of these events as well as their impact on record and replay solutions. In the rest of our work we will describe our implementation to enable recording these events for single-processor hardware-assisted virtual machines as a part of Simuboost [1].

3.1 TLB Induced Non-Determinism in Recording

As mentioned in Section 2.1.1 the MMU does not ensure consistency between the page tables in RAM and the TLB entries that it creates to decrease the overhead caused by virtual memory. When the MMU translates a linear address to a physical address and TLB entry exists for this linear address, the page tables in RAM are not consulted even if their permission bits or the physical address that they reference have been changed. Instead the TLB entry is used directly to resolve the physical address and assert if the current process is allowed to read, write or execute the referenced memory. The same principle applies to the accessed and dirty bits of the page tables. The MMU only sets these bits in the relevant page table entries that were used to translate the current linear address when they are not already set in the structure that it used to perform the translation. If the instruction that caused the translation writes to the linear address and the dirty bit needs to be set it is already set in the relevant TLB entry, it will not be set in the last page table entry. The accessed bit never gets set when a TLB entry is used for the translation because every TLB entry is implicitly considered accessed because the MMU only creates entries in the TLB when translating an access to the given linear address at which point it set the accessed bit in the page table entries.

This implies that if the guest operating system fails to invalidate the TLB when clearing either the dirty or accessed bit in one of its page table entries, the state of these bits after a future access to the linear address depends on implementation defined behavior of the TLB. While this behavior can still be deterministic for a system running on bare hardware, it is certainly non-deterministic when the TLB is being shared between the host system and potentially multiple guests in virtual machines. At any point the guest's execution can be preempted, for example because its scheduled time slice elapsed or because it tried to execute a privileged instruction that trapped into the hypervisor. At this point any amount of host code can run or the host can schedule another virtual machine to run on the same CPU. All of these actions utilize virtual memory and will create new entries in the TLB for any translations that are not already present, possibly evicting the TLB entries that were created from translations in the recorded VM. The host might even sometimes explicitly flush the TLB depending on what operations it performs at this time. Also, according to Intel's documentation [2, p. 1215] on the behavior

of the TLB in their processors, "[a] logical processor may invalidate any cached mappings at any time".

Operating systems generally flush the relevant TLB entries when clearing the dirty bit from page tables because they need to accurately track which pages have been written to. This is needed to avoid data loss from erroneously discarding these memory contents when the system runs out of RAM because they are considered *clean*, i.e. saved to persistent storage. Some operating systems, such as for example Linux since version 3.16 and up to the present day, do not apply the same rigor to keeping the state of the accessed bits consistent with the TLB. [21] When the Linux kernel clears the accessed bit in a page table entry, it does not flush the corresponding TLB entry to avoid the performance hit that would result from the TLB miss if the address is accessed again and the MMU has to read the relevant page tables from memory. The inconsistency between the TLB and page tables is being tolerated in this case because the state of the accessed bits is only used as a heuristic to predict which pages will likely be accessed again in the future and should be kept in memory and the chance of a misprediction due to this inconsistency is considered "relatively low" by the authors of the change due to the TLB being flushed regularly due to e.g. context switches between applications.

The result of this non-determinism is that if the TLB contains an entry for a particular linear address during the recording of the virtual machine but does not contain such an entry during the replay, an additional accessed or dirty bit will be set in at least one page table entry in memory during replay. If the conditions are reversed, a bit will be set during the recording but not the replay. It should also be noted that even if the TLB's behavior was deterministic and an entirely separate TLB could be instantiated for a given virtual machine, the TLB that is used during replay can still differ from the one that was used during recording, such as in a heterogeneous record and replay solution in which e.g. the recording is performed using hardware virtualization and the replay using emulation.

3.2 Impact of Non-Deterministic TLB Behavior on Replay

Some evaluation metrics for implementations of record and replay have been outlined in [22], which include:

- **probe effect**, which measures how much the original execution is slowed down due to the recording compared to a normal execution that is not being recorded

- **log size**, the amount of data per instruction that is generated by the recording and needs to be stored in the replay log
- **replay slowdown**, the difference in run-time between the recorded execution and the replay
- **replay accuracy**, which describes how accurately the replay matches the recorded execution at the chosen level of precision

A record and replay system will generally have to make trade-offs between these variables. For example implementations that record at machine instruction level such as ReTrace[15] or SimuBoost[1] use periodic checkpoints at which the state of the virtual machine is recorded while the states between these checkpoints are computed during the replay. If checkpoints are created more often during recording, both the probe effect and log size increase. On the other hand the replay can proceed faster if it can be parallelized such as in SimuBoost where each checkpoint interval is can be emulated on its own machine, drastically lowering the replay slowdown. A higher number of checkpoints also allows the user of the replay to "seek" closer to the part of the execution that they are interested in without waiting for the intermediate state to be replayed.

As mentioned before, only non-deterministic events need to be logged during the recording. These events can be further separated into two categories according to [19]:

- **Synchronous Events** These events always appear at the same point in the instruction stream but their output is non-deterministic and needs to be recorded. Examples include the x86 `RDTSC` instruction which returns the number of processor cycles since the system was started or the data that devices returned for I/O instructions.
- **Asynchronous Events** These events occur at a non-deterministic point in time but their content can be deterministic. An example for this are interrupts, which cause a deterministic change, e.g. changing certain registers but occur at non-deterministic time.

When using hardware-assisted virtualization to run the guest during the recording, most synchronous and asynchronous events can easily be tracked by configuring the hardware to cause traps into the hypervisor when they occur, which will then log them. Synchronous events can simply be recorded and when replayed in order when the corresponding instruction that caused them is encountered (e.g. when using hardware virtualization the hypervisor would configure the hardware to trap

on these instructions). Asynchronous events on the other hand have to be handled differently. For these events a unique identifier that specifies the exact point in the execution at which they occurred needs to be recorded. Because the instruction pointer is not enough to identify a specific point in the execution (consider the existence of loops for example), this so called *landmark* has e.g. been described in [23] to also include the state of registers and flags, the branch counter or as in SimuBoost the retired instruction performance counter. Only when this landmark is hit — i.e. the current state of the machine during replay matches the landmark — will the asynchronous event be injected into the replay. Implementing a record and replay system that is able to correctly replay any given guest is extremely challenging and it is impossible to ever test every single combination of host, hardware and guest. Because of this it can be helpful to include additional state in the landmark that can be used to verify that the replay did not diverge from the recording.

While the accuracy of the replay is immensely important to the usefulness of a record and replay system, some (theoretical) inaccuracies might be tolerated to improve the other areas of concern. If the difference in accessed bits would not affect the replay accuracy it could be tolerated if the impact on the probe effect of accurately tracking these events was to high. For example writes to the guest's memory could in theory be discarded if it was known ahead of time that the guest will never read this memory location. It has however been observed in [24] that even "single bit in the state of the replayed system can potentially alter its execution path". This problem is further exacerbated by the fact that the divergence between recorded execution path and replay usually does not happen directly but is only later influenced by such a small change.

These differences between recording and replay can not only change the layout of virtual memory because a different page will be evicted or swapped to persistent storage depending on the state of the accessed bits (e.g. Linux uses a complicated algorithm [25] that among other factors takes into account how often a file has been accessed), but also alter the flow of execution. If a process accesses memory that was swapped out in the replay but not the recording, it will trigger a page fault, causing additional code to run to re-load the affected data. This can then lead to landmarks being missed if the counters that are being used no longer match the recorded value, causing the replay to abort because asynchronous events can no longer be injected correctly. The same result can even be expected just because the kernel read a different value from the page table entry at the time a landmark was supposed occur, causing the register contents to differ from the landmark.

Accurately tracking the changes to accessed as well as dirty bits is also important to enable the use of record and replay to debug the guest operating systems memory management code. While we mentioned earlier that the specific behavior of not

flushing the TLB when clearing accessed bits is an optimization and the flush is normally performed after clearing a dirty bit, this of course only applies to guests that behave correctly. To be useful as a debugging tool however, a record and replay implementation must also be able to precisely replay a system that does not behave correctly.

3.3 Preliminary Measurements and Related Work

Our tests with a virtual machine running Ubuntu 16.04.1/Linux 4.8.10 show that depending on the amount of RAM that is allocated to the virtual machine, inconsistencies in the state of the accessed bits happen with different probabilities. They can be observed regularly even just during the boot process of the guest that is being recorded using the current SimuBoost system if the virtual machine is given 256 MiB of RAM. This subsequently leads to landmark misses during replay, while the same does only occur extremely rarely at 4 GiB of RAM. We believe this happens because the Linux kernel's page aging algorithm that clears the accessed bits and later checks which have been set again by the MMU to determine which pages are actively being used is invoked more frequently when the system only has a low amount of free RAM available.

Despite the fact that this issue seems comparatively easy to observe in practice, we are not aware of any performant record and replay solution that aims to track and replay these events accurately. Of the implementations that we mentioned in Section 2.4, only ExecRecorder is able to do so completely, by virtue of using the Bochs simulator that implements the TLB in software and is thus not influenced by any state outside the guest, can behave deterministically for each execution and uses the same TLB implementation for both recording and replay. Implementations like ReVirt that run entirely as a user mode application while re-using the host's resources are unable to track these events because they operate only on virtual memory. The paper that describes SMP-ReVirt is the only work we could find that mentions the problem. It uses shadow paging and avoids any non-determinism by flushing the TLB after performing any changes to the shadow page tables that the MMU uses and being able to mirror the state of the accessed bits deterministically to the guest's page tables. This does cause a recorded execution to behave differently from a normal virtual machine that would nowadays run using hardware virtualization and thus changes the behavior of the guest. For example it would be impossible to debug a problem in the guest's memory management code that is being caused by missing TLB flushes on real hardware because the recorder adds those flushes behind the guest's back.

It is possible that the implementations of record and replay that we examined

simply did not notice this problem because they used a different portion of the guest's state to define their landmarks. However this does not negate the problem's existence but could hide it or make its it less likely to cause a landmark miss during the replay.

3.4 Conclusion

In this chapter we have identified how optimizations of TLB flushes can in some instances introduce non-determinism into the state accessed and dirty bits inside the page tables in RAM. We also discussed how this can affect record and replay systems that aim to accurately replay an entire virtual machine by causing the memory contents of recording and replay to diverge and how even these minuscule changes can introduce larger derivations in the replay or cause it to fail completely by changing the course of the execution of the replay, leading to landmarks being missed.

While this effect can be observed even when recording simple executions such as the boot process of a Linux distribution, we were unable to find any record and replay system that aims to eliminate it by accurately logging and replaying these events. While some systems are unaffected by this problem, either because they operate at a higher level in virtual memory or because they operate at a lower level and emulate their own implementation of a TLB, to our knowledge none of the implementations that replay at machine instruction level and use hardware-assisted virtualization to limit the probe effect are able to handle these events correctly.

Investigating a way to log these TLB induced non-deterministic events using hardware virtualization and evaluating the impact this has on the evaluation metrics for record and replay systems outlined in Section 3.2 will be the goal of this work.

Chapter 4

Design

This chapter concerns itself with the design of our proposed solution to log all writes of dirty and accessed bits by the x86 MMU due to the non-determinism inherent in the specification of the TLB using hardware-assisted virtualization. Our design will not not only be influenced by the problem itself but also by the choice to find and evaluate a solution using the hardware-assisted virtualization capabilities in modern Intel x86_64 CPUs and implement this solution as part of SimuBoost.

Our primary goal is to enable tracking the setting of accessed and dirty bits with very high accuracy for any unmodified guest, even if the guest does not issue TLB flushes on some of the changes it makes to its page tables, with a secondary goal of keeping the overhead of our implementation over the existing record and replay capabilities that SimuBoost provides to an acceptable level.

We will now discuss both the recording and replay of MMU events in detail, explaining our proposed design while contrasting it with other possible solutions that we discarded.

4.1 Recording

The problem of recording non-deterministic events can be generalized as follows: The guest that is being recorded must not be allowed to be aware of any event that the hypervisor is unaware of. In many cases this can be achieved by configuring hardware-assisted virtualization to trap into the hypervisor when these events occur. However even with the virtualization extensions that are available in current x86_64 CPUs it is not possible to configure the virtual machine in a way that causes only the setting of accessed and dirty bits to trap into the hypervisor but not other

accesses to the guest's page tables. This leaves two possible options to achieve the goal stated above:

Either

1. The guest system must be prevented from reading from its own page tables by any means as long as their state is not synchronized with the TLB. This restriction applies both to the currently active page tables as well as any other memory that has been used as a page table and not has been reused in for another purpose before the attempted read. At the same time the MMU can be allowed to freely access the currently active page tables and write the accessed and dirty bits to them.

or

2. The MMU must be prevented from writing to the guest's page tables without the hypervisor being informed. The guest on the other hand can be allowed to freely read the state of the accessed and dirty bits because it is never able to be influenced by an event that the hypervisor has not logged yet.

The first option would allow an optimal solution (in the amount of events that need to be logged) to the problem of non-determinism induced by the accessed and dirty bits to be implemented by to detecting when the page tables are being read by the guest and at this specific point log their state to the replay log, allowing them to be replayed exactly. While this might introduce small differences in the memory contents of recording and replay because the changes to the guest's page tables that are never read do not get logged, we would still prefer it because we believe it offers the best performance overall in all 4 metrics outlined in Section 3.2. As mentioned there, such small differences between the state of recording and replay at a specific point could be tolerated in some cases to improve performance, if it can be proven that they will not cause any changes further down the line. In any case where these bits might either be set or cleared, but are never read by the guest and thus can never influence its state and the execution, they would not necessarily need to be logged, just like the contents of any memory that is only written and never read could be discarded from the replay log.

However, the only way of guaranteeing these conditions on current x86_64 hardware while preserving the isolation of virtual machine and host system that we are aware of is the use shadow page tables. When shadow paging is being used the hypervisor already has to make sure that any changes the guest makes to its page tables are reflected in the shadow page tables and thus needs to protect the guest's page tables. Additionally, it could also protect the guest's page tables from reads and log the setting of accessed and dirty bits when it mirrors them to the guest's

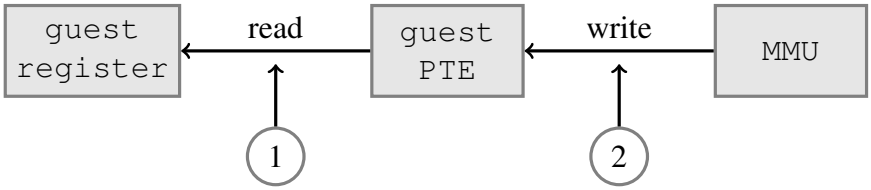

Figure 4.1: Overview of the point at which each of the proposed methods intercept the and log the accessed and dirty bits being set.

page table on demand as described above. Sadly, the use of shadow paging would also come at a significant performance cost. (The use of the EPT mechanism for nested paging has been credited with a performance improvement over shadow paging of up to 48% for MMU intensive workloads [5])

When the EPT is in use, the MMU uses the guest's page tables directly and the access permissions that are set in the EPT apply to both regular guest accesses and MMU accesses. For the first option described above this would imply disallowing all read access to the guest's page tables, which would cause any address translation by the MMU to trap into the hypervisor, resulting in a significant performance loss. This caused us to focus our efforts on the second option, developing a more brute-force system that intends to log every single accessed and dirty bit being set. Some of the steps that are required to successfully log MMU writes of these status bits to the page tables are however similar to those that are needed to implement shadow paging, though less work needs to be done by the hypervisor. We will now give an overview of the individual steps of our design for the recording phase.

4.1.1 Tracking Guest Page Tables

The first necessary task to needs to be completed to accurately record changes to the guest page tables is locating them in the guest's memory and keeping track of any changes that the guest makes to them while it is running.

Context Switches

This requires trapping into the hypervisor when the guest sets the `CR3` register to a new top-level page table and recursively iterating through all the entries that are marked as present in the table it points to, recording the location of each

encountered guest page table.[1] How often this needs to be repeated depends on the current configuration of the virtual machine's CPU, which influences how many layers of page tables are used to translate linear addresses, from 2 layers in 32bit operation to up to 5 layers in upcoming CPUs, support for which is already present in the Linux kernel.

Changes to Active Page Tables

The hypervisor also needs to ensure that any subsequent changes that the guest makes to the currently active page tables also cause a trap, allowing the hypervisor to adjust its list of guest page tables. The reasons for this are twofold.

1. It might have to track additional page table that the guest added at lower levels of the hierarchy by either replacing a present entry or adding a new entry.

2. It is also important to notice when a page of memory is no longer used as a page table. This is most easily implemented at this time because the first point already requires changes to the guest's page tables to trap into the hypervisor.

Making sure that the dropping of page tables by the guest is tracked is important because otherwise unrelated memory will be assumed to contain guest page tables and writes to this memory would cause the hypervisor to assume that the guest changed a page table, interpreting this memory as page table entries leading to unpredictable results. It is important that changes to page tables are handled efficiently as this is a very frequent operation in many operating systems. When a program forks off a child process, their page tables are set up to share all data currently available to the parent process read-only and any writes they perform to their memory will lead to a page fault in the guest to duplicate the affected data and adjust the page tables to allow the write.

Old Page Tables

Consideration also needs to be given to the page tables of the previously active page table hierarchy. It could be advantageous to remember a number of previously

[1] Care needs to be taken not to run into an endless loop because of a common implementation trick in operating system kernels called *recursive page* tables, in which the last entry of the top-level page table points to itself greatly simplifying the implementation of accessing the page tables (which are normally only referenced using physical addresses) from the kernel using linear addresses.

active page table hierarchies and keep them protected, if the guest is likely activate them again in the future. In which case it would be possible to skip walking the entire hierarchy on that context switch. On the other hand, if the guest process they belonged to has been terminated and the guest recycles their memory for other purposes the additional exits into the hypervisor because the memory is still protected would incur additional overhead. Because there is no obviously correct heuristic to decide in which cases this is advantageous and to reduce the complexity of our implementation, we have decided to instead to only track the currently active page table hierarchy.

4.1.2 Logging MMU Updates to Guest Page Tables

When the physical addresses that contain the guest's page tables are marked as write-protected in the EPT, which, as discussed above, is already necessary to detect any changes to them, the MMU will also not be allowed to change any of the status bits in the page table entries. This causes a trap into the hypervisor on any memory access by the guest that is not cached in the TLB and would have to set an accessed or dirty bit in any of the page table entries.

At this point, the access first needs to be distinguished from other accesses by the guest. Thankfully, the EPT violation that causes the exit into the hypervisor when one of the permission bits in the EPT does not match the guest's intent contains both the guest linear and guest physical address that caused it as well as an *exit qualification* describing the type of the access and EPT permissions of the physical address. These allow the hypervisor to identify if

- the access was a write by guest code, a write by the MMU or another access
- the guest physical memory address contains a currently active page table

After the access has been classified as a write to a page table entry by the MMU, multiple options exist for how to allow the guest's execution to continue while logging the relevant changes to the status bits:

Setting Status Bits Manually

The first possibility is to translate the linear address from the EPT violation in software, using using the information from the exit qualification to determine which bits to set in the guest's page tables. Afterwards, the guest is resumed, hopefully continuing it's execution unobstructed. The MMU will then not attempt to write to

the page table entries as the bits it needs to set are already set. In the best case, this is expected to have a low overhead and will add the translation to the TLB, but in some cases the guest will immediately exit again. This can happen for example if both the fetching of the current instruction from memory as well as an access to memory by the instruction cause the MMU to update page table entries. In the case of the `REP` prefix to string instructions, a single instruction can read and write an almost arbitrary amount of memory potentially causing the setting of accessed or dirty bits for each page.[2]

Single-Stepping

A second option is to create copies the guest page tables for the given linear address, remove the write protection in the EPT from the physical addresses containing these page tables. Then the guest is single-stepped for one instruction, after which the state of the page tables is compared and relevant changes added to the replay log so that they are replayed before this instruction. This has the advantage that the instruction is directly executed by the guest and the guest linear address is translated by the hardware MMU, preventing any subtle behavioral changes that might be introduced by performing the translation in software. The translation is also added to the TLB, improving performance on future accesses and ensuring that the behavior is identical with a non-recorded execution. However, this solution likely has a significant performance impact (e.g. [23] showed a 3000× slowdown due to single stepping and this adds additional work due to copying the guest's page tables) and can run into the same problems as the first mechanism if a single access causes multiple EPT violations. Though this could be worked around by copying the complete page table hierarchy, this would further increase the performance hit.

Emulating the Access

The solution that we decided to pursue is to emulate the instruction that resulted in the MMU causing an EPT violation in the hypervisor, setting any accessed or dirty bits as needed during the emulation. While emulation generally also comes with a performance impact, this avoids multiple traps into the hypervisor to execute a single guest instruction. The downside is that the guest addresses that are accessed during the emulation do not get added to TLB, potentially slowing down the guest in the future and causing differences in the non-deterministic behavior of the TLB between a native and recorded execution.

[2]The addition of the `ERMSB` feature flag that indicates increased performance of these instructions on recent Intel CPUs [26] could indicate that their usage will even increase if compilers decide to regularly take advantage of this

4.2 Replay

After the events that we are interested in have been recorded, it is also important to consider if any additional changes to the replay implementation will need to be considered. We will also discuss the format in which the events that are gathered during the recording are logged in this section because we consider this choice to be more influential during the replay.

4.2.1 Replay of Asynchronous Events

As discussed before in Section 3.2, replaying asynchronous events necessitates some sort of marker that identifies a unique point and time in the execution, a so called landmark. This landmark contains some of the virtual machine's state that both identifies such a point uniquely and is fast to compute during replay because it needs to frequently be compared to the state of the replay to ensure that the event can be inserted at the correct point.

Because it is challenging to construct a system that is able to replay every execution perfectly, the landmark should include more information than is necessary to identify the point at which the asynchronous event is injected into the replay. The additional information can then be compared to the state of the replay at the point that matches the landmark to verify that the replay is accurate to the recording.

Because we implemented to logging and replay of accessed and dirty bit events on top of the existing SimuBoost record and replay system, we were able to re-use the existing infrastructure for replaying asynchronous events. SimuBoost uses the `INSTRUCTIONS_RETIRED` hardware performance counter that counts the number of fully executed instructions[3] as a landmark and saves the CPU's general purpose, pointer and flags registers to check the consistency of the replay once the landmark is hit.

4.2.2 Replay of MMU Updates to Guest Page Tables

Compared to the recording of MMU events, replaying them is surprisingly simple. Once the landmark for an MMU event is reached, the replayer only has to set the corresponding bit at a fixed offset in the page table entry. How this is achieved depends on the format of the log entries that is chosen for these events. It would be possible to log the linear address whose access caused the event to be generated,

[3] i.e. only instructions whose results have been fully committed and not rolled back due to a speculative execution misprediction

along with the type of access. However, this would imply that to replay the event the guest's page tables have to be walked again to determine the page table entries that have to be modified. Instead we chose to simply log one event for each modified page table entry, containing its physical address and the type of bit that was modified. While this structure can generate up to 5× more events, if all accessed bits and the dirty bit have to be set for an access, we believe the reduced complexity during replay outweighs the increased size of the replay log and replaying of additional events.

Chapter 5

Implementation

In this chapter, we will describe some of the details of our implementation. As we were able to implement the support for logging non-deterministic MMU events on top of the existing SimuBoost record and replay system, we were able to re-use a lot of its infrastructure and components, only adding the necessary code to log and replay these events. Because of we are building on an existing, working record and replay system, one of the primary goals for our implementation was robustness against both implementation bugs in one of its other components and unexpected guest behavior. This is intended to allow execution to continue both in the recording and replay phase and avoid regressions on the status-quo because as discussed in Section 3.3 depending on the guest configuration the logging of accessed and dirty bits can in practice be optional.

We will now first give a brief overview of SimuBoost's implementation before we explain our changes to implement recording and replay of MMU events.

5.1 SimuBoost

As mentioned before, SimuBoost uses the KVM Linux kernel module and QSIMU, a modified fork of the QEMU emulator, to create a recording of a virtual machine using hardware-assisted virtualization in KVM and emulation in QSIMU in tandem. During replay only QSIMU is used and its Tiny Code Generator (TCG) executes the guest's instructions. A third, custom component called *simustore* handles the storage of the checkpoint data and replay log. Simustore also allows distributing the checkpoints' contents to a cluster of machines and replaying the execution in parallel using a separate QSIMU instance on each machine to replay a specific checkpoint interval, greatly decreasing the replay slowdown.

However, our implementation was able to ignore simustore completely because the existing mechanisms provide a sufficient abstraction and most of our efforts were focused on implementing the recording as part of KVM and only very small changes were needed facilitate the replaying of MMU events in QSIMU.

5.2 Recording MMU Events in KVM

While the two tasks we described in Sections 4.1.1 and 4.1.2 — tracking the location of the guest's currently active page tables and logging the setting of accessed and dirty bits within these by the MMU — are are conceptually separate, they have become somewhat intertwined in our implementation due to our use of emulation for both areas. We will now go over the individual changes that were needed to KVM to support both.

5.2.1 Data Storage

First however, we will describe the central data structure that we use to store both the location of the guest's active page tables and their level in the active page table hierarchy, as well as their status in our implementation.

For each page table that is reachable via the active hierarchy, we store both its guest physical address and its level in the hierarchy inside a *radix tree*, a highly optimized key-value storage structure that is part of the Linux kernel and is used e.g. to track linear to physical mappings for each process in the system. This provides key advantages for implementing the requirements detailed in Section 4.1.1 over some alternative implementations such as marking the guest's page tables by using some of the reserved bits in the EPT entries that map them to machine physical addresses. For example, our system does not need to be aware of the kernel's memory management system swapping out the guest's memory and deleting the EPT mapping. While it is certainly likely that there are other data structures that would deliver even better performance for this specific use case, implementing them to the same level of reliability would not have been feasible as part of this work. We store the instance of a radix tree that we allocate for this purpose within the `kvm_vcpu` struct that holds the state of the guest's virtual CPU because the active address space is also part of the CPU state.

An additional feature of the radix tree structure that we make heavy use of are tags. For each radix tree, up to three tags can be defined. These tags can be applied to the entries and functions are provided that allow efficiently checking if a given tag is applied to any entry in the tree by only inspecting the root as well as iterating

over only the entries that are marked with a given tag. We define the following tags that we make use of to enable important performance optimizations:

- `GPT_MAYBE_DROPPED`: This tag signifies that (one of) parent page table(s) of the the page table that is referenced by the entry has been modified by the guest, but it has not yet been discovered if this memory address still contains an active page table.
- `GPT_MODIFIED`: This tag simply signifies that the referenced page table has been modified by the guest but these modifications have not yet been synced to the radix tree.

We will explain the specific cases in which these tags are used in the next sections, which each outline the major events that our implementation has to react to.

5.2.2 Context Switches

The first modification to existing behavior that we had to introduce is to unconditionally reconfigure the Intel VMX hardware virtualization to trap into the hypervisor when the guest activates a new page table hierarchy by setting the `CR3` register. While this is normally avoided when the EPT is in use, it is inherently necessary to achieve our goal.

When this happens, we first discard our knowledge of the old page table hierarchy. This is done by simply deleting the previous contents of the radix tree. Then we walk the entire new page table hierarchy and mark each table as read-only in the EPT, starting with the top-most table referenced by the new contents of the `CR3` register and recursively descending into all it's children that are marked as present. A performance optimization that we apply here is that before we recurse into the next child page table referenced by one of the current's entries, we check that it has not already been traversed. This saves some function calls during this frequent operation because we observed a Linux guest for example re-using the same page table addresses often, presumably for copy-on-write purposes.

5.2.3 Changes to Page Tables

Because the guest's page tables are protected from writes by the EPT after the last step, an EPT violation now occurs whenever they are written to while the guest is running. While the function that handles this exception normally just invokes a page-fault handler to determine if the guest is allowed to access the physical

address that caused it and potentially create a mapping in the EPT for it before resuming the guest, our implementation adds several additional cases:

- If the EPT violation was caused by the MMU attempting to set an accessed or dirty bit, the access is emulated (more on this below). This can be inferred just from the exit qualification describing the offending guest access.
- If the access is a write access, we check if its target guest linear address points to a guest page table by translating it to a guest physical address using the guest page table and looking that up in the radix tree. If this is the case, the access also is emulated.
  However, before the second check is performed, we check if any of its entries in the radix tree are marked with the `GPT_MAYBE_DROPPED` tag. If this is the case, we first walk the entire guest page table hierarchy, protecting each page table and clearing any tags on the corresponding entries in the tree and then remove any remaining entries that are marked with this tag from the tree. It is not sufficient to traverse the page tables marked as `GPT_MODIFIED` and their children and consider any potentially dropped page tables that are not encountered actually dropped, because they can still be referenced from other page tables.

In both of the cases above, the EPT violation handler returns after the emulation. Either the guest's execution is resumed if the emulation was successful or control is returned to the user space QSIMU process, e.g. to emulate a device if access to a memory-mapped I/O region was attempted.

In any other case, the page-fault handler is invoked as normal to supply the requested guest physical address with a memory mapping in the EPT. Afterwards, only a small fix-up is added performed. The address that was just paged in can still contain a guest page table if the access did not attempt to write to it. To ensure that the resulting mapping is read-only to the guest, we protect the physical address in this case. This is done after the mapping was created to avoid any error-prone to the complex page-fault logic.

5.2.4 Emulation in KVM

in Section 4.1.2 we explained our reasoning for using emulation to log TLB induced non-deterministic events as they occur. To implement this, we were able to reuse KVM's existing support for emulating x86_64 instructions that is used for instructions that the guest isn't allowed to execute directly. As described above, we also chose to track any changes to guest page tables in the emulation code. Both of

these features are implemented in the guest page table walker that emulates how the hardware MMU translates the guest's linear addresses to its physical addresses. Because this is such a central code path that is used for any access to a guest linear address and not just limited to the emulation code, it allows us to perform both MMU and guest changes to page tables within a single translation if both need to occur. It also single-handedly ensures that any code path that accesses guest memory other than the ones we explicitly define in the EPT violation handler causes the necessary events to be logged.

Logging Accessed and Dirty Bits

Because the emulation code needs to use the guest's page tables to access any of the its memory and the guest must not be allowed to notice any divergence between a natively executed and emulated instruction, the emulation code already supports setting the accessed and dirty bits in these page table entries as needed. At the point at which the bits are being set, we simply call a function provided by simuboost, `kvm_vcpu_rr_add_event`, to add an event with the newly defined type `MMU_ACCESSED` or `MMU_DIRTY` containing the guest physical address of the page table entry to the replay log as needed. Because we ensure that all of the guest's active page tables are known to the hypervisor and that the guest can not write to them directly this is the only place in which accessed or dirty bits can be set. This allows us to log literally all instances in which these bits are being set.

Tracking Modifications to Guest Page Table Entries

Any time the emulated page table walker is invoked and has successfully translated the guest linear to a guest physical address and the access to this address is a write operation, we check if the physical address is present in the radix tree i.e. if it is a guest page table. If this is the case we mark the entry in the radix tree with the `GPT_MODIFIED` tag and recursively mark the entries for all of the page table's child tables with the `GPT_MAYBE_DROPPED` tag. Because the level of the page table is stored in the radix tree, we are able to start walking the guest's page tables from the modified table, making this is a fast operation for the common case where a lower level page table is changed during a copy-on-write operation in the guest. If the page table being changed is on the lowest level, only a single tag needs to be set.

5.2.5 Resuming Guest Execution

A downside of this approach to handle the guest's writes to its page tables in the emulation's page table walk is that at this point we are unable to determine what exactly the write will change. It is possible that the guest simply clears an accessed bit or changes the read-only status of the page table entry, which does not change the set of active page tables that we need to track. On the other hand we consider this approach preferable and less error prone to attempting to modify every path in the emulation. It does however imply that we need to actually examine the guest address that was modified and potentially update the radix tree with any changes to the page tables.

We do this as part of KVM's main loop, which is entered when the user space process (QSIMU in this case) passes execution of the guest to KVM and then continues running until the guest traps back into KVM. The loop then tries to handle any exceptions that KVM can resolve itself, such as page faults/EPT violations and passes control back to QSIMU if a device needs to be emulated. Afterwards the guest is allowed to continue executing. At this point, before the guest is entered again, we check if any radix tree entries are tagged `GPT_MODIFIED`. If this is the case, we walk the guest page tables recursively starting at those and clear any tags that are encountered. This ensures that Any modifications that were performed are reflected back in the radix tree and the guest's page tables are protected appropriately.

5.3 Replaying MMU Events in QSIMU

Because we are able to build on the already existing support for replaying various asynchronous non-deterministic events that was implemented in QSIMU as part of SimuBoost, replaying the events caused by accessed and dirty bits was extremely simple. The asynchronous events are handled in the order in which they occurred and the time in the replay at which they are inserted is determined by the landmark that is generated during the recording. For each event, its type defines what data it contains and how it has to be replayed.

When an event of type `MMU_ACCESSED` or `MMU_DIRTY` is encountered, we simply remove it from the queue, extract the physical address that it contains and use the wrappers that QEMU provides to access the guest's memory to read the page table entry from the specified address, set the bit that needs to be set and write it back. Then execution continues as normal.

5.4 Conclusion

In this chapter we have explained our implementation of the design that we presented in Chapter Section 4. The existing abstractions in SimuBoost allowed us to focus entirely on the subject of recording and replaying MMU events.

The bulk of our implementation is part of the KVM kernel module where we implemented the recording of these events. By hooking into some of the most central code paths that deal with reading and writing the guest's memory, both from the guest itself and the host system, we believe to have created a very comprehensive system that should be able to record any changes that occur to the guest's page tables.

We were able to re-use one of the most tested and optimized data-structures that is used as a central part of every Linux system in existence, which fit very well with out intended implementation and allowed us to easily handle the numerous edge-cases that we encountered.

On the other hand, the implementation of the replay system for MMU events within QSIMU was extremely simple as it only needs to distinguish between one event for dirty and one for accessed bits and set the corresponding bit in the page table entry that is referenced in the event struct.

Chapter 6

Evaluation

So far we have introduced the problem of divergence between the recording and replay of virtual machines caused by the non-deterministic nature of the TLB in x86 CPUs and explained the design of our solution to this problem by logging all accessed and dirty bits as they are set in by the MMU in the guest's page tables. We have also given some details of our implementation and motivated them. In this penultimate chapter we will present the results of the tests and benchmarks that we have performed to evaluate our implementation in its accuracy and performance.

We will first describe our test setup, then explain what steps we have taken to ensure our implementation solves the stated problem correctly by accurately tracking all MMU events. After this we will present a set of benchmarks that we have chosen to evaluate our implementation in the other metrics that we presented in Section 3.2 — probe effect, log size and replay slowdown — and explore the results of these benchmarks.

6.1 Test Setup

The specifications of the system that we used to perform the following benchmarks and evaluate the correctness of our implementation can be seen in Table 6.1. While our implementation only aims to log accessed and dirty bits for a single-core VM, the additional performance offered by this machine both by the additional CPUs as well as the fast SSD was useful to ensure that the the creation and storage of checkpoints was not a bottleneck during the benchmark.

The guest system that we have chosen for this evaluation is Ubuntu 16.04.1 (x86_64), with version 4.8.10 of the Linux kernel. This main reason this specific operating system was used is that it has previously been used to evaluate

Component	Model
CPU	2 × Intel Xeon E5-2630 v3 “Haswell” (8 cores/16 threads each)
RAM	64 GiB DDR4-2133MHz
Storage	Samsung SSD 850 EVO 1TB
Host Kernel	Linux v4.3.0 + SimuBoost

Table 6.1: Test Setup

other work based on SimuBoost, which may aid when comparing our results to the results from those works. The guest VM was configured with the default “qemu64” virtual CPU and 4 GiB of RAM.

6.2 Qualitative Evaluation

While we are confident that our implementation is able to intercept and log all changes to the guest’s page tables due to the fact that it is able to hook into the two central code paths in the KVM module that control access to the guest’s memory from either the guest itself or the host — the EPT violation handler and emulated page table walker, respectively — we also needed to test that it works in practice.

While it is next to impossible to prove that the implementation works for all possible guests, we devised two tests to at least show that it solves the problem that are encountered in practice: Non-deterministic TLB behavior causes different contents in the guest’s page tables during replay, which then causes the replay’s execution to diverge from the recording when the guest makes decisions based on the contents of its page tables which ultimately leads to missed landmarks and breaks the replay completely because asynchronous events can’t be replayed.

A key problem when evaluating our implementation is that we cannot take a replay log that exhibits this problem and replay it using our implementation to show that it fixes the problem. This is impossible because the log does not contain the asynchronous events that would be needed to replay the MMU’s behavior. On the other hand, any recording necessarily affects the recorded guest and a recording performed by our implementation will be different from one performed by upstream SimuBoost even if the guest is given the same inputs, if only due to the performance impact that the additional logging has on the guest. This might mask lingering problems instead of outright fixing them as it appears.

However, as we explained in Section 3.3, this problem is easily reproducible by

only allocating 256 MiB of RAM to the guest. This will cause missed landmarks almost immediately regardless of which program is run in the guest and sometimes even during the boot process. When our implementation is used for recording and replay, we could not observe this on either the benchmark system described above or our development system, which uses a CPU from a different micro-architecture generation and is likely to have different TLB behavior.

We were also able to take a recording generated by our implementation that could be replayed without any issues and modify QSIMU in such a way that the `MMU_ACCESSED` and `MMU_DIRTY` events are simply ignored during replay, essentially negating our improvements. When replayed in this manner, the same recording exhibited landmark misses that seem to be caused by differing page table entries.

While the caveats mentioned above still apply, these tests give us confidence that our design is sensible and our implementation performs correctly.

Another important factor that we have not mentioned yet is that the our implementation should not introduce any new bugs or cause the guest to behave abnormally during recording or replay. While we did not see encounter any evidence of this being the case. An automated system that for a given workload compares the results of an unrecorded guest, a recorded guest and the replay would be immensely useful in debugging record and replay systems and strengthening this anecdotal evidence of correct behavior, but is outside the scope of this work.

6.3 Quantitative Evaluation

Now that we assured ourself that our implementation behaves correctly we will proceed to evaluate its performance in the quantitative metrics, probe effect, log size and replay slowdown. Before presenting our results, we will go over our methodology in gathering the data and the chosen benchmarks.

6.3.1 Methodology

We made use of QEMU's snapshot feature that allows executing a guest from a disk image while writing any changes that the guest makes to the emulated disk that is backed by the image to temporary files, preserving the original state of the disk image. This allows running each benchmark from the same original state.

For each run of each individual benchmark, we booted the virtual machine and waited 10 seconds after the boot process was completed to ensure that any services

that are inevitably running in the guest have completed their startup and settled down. Then we started the recording and after the initial checkpoint was created started the benchmark within the guest. We chose this process to ensure that only the benchmark's run-time and only the data that the recording produces while the benchmark is running is measured.

We collect the following measurements for each run:

- **run-time** of the benchmark: It is measured as wall clock time from inside the virtual machine and roughly equals the time that was recorded, minus a small delay between the start and end of recording and benchmark. The length of the benchmarks was chosen to make this difference negligible.
- **log size** of the recording: The amount of data that was generated by the recording.
- **MMU event count**: How many of the `MMU_ACCESSED` and `MMU_DIRTY` events were logged during the benchmark.
- **replay time**: The wall clock time that the recording of the benchmark took to replay.

Afterwards, we repeated the same steps with the unmodified SimuBoost code, collecting the same measurements. (except for the MMU event count of course)

By comparing the metrics gathered from our implementation to this baseline we intend to evaluate how big the impact is that the recording of all accessed and dirty bits has on our record and replay metrics.

6.3.2 Benchmarks

The macro-benchmarks that we have chosen are implemented as part of the Phoronix Test Suite[27], an open source benchmarking framework that automated testing of standardized benchmarks across different platforms.

Linux Kernel Build

This benchmark compiles the Linux kernel in a `make defconfig` default configuration, with one parallel make process for each CPU core (1 in our case). It was chosen because we believe it represents a good mixed load. It runs many individual make and gcc processes, but also produces some CPU load and performs I/O to read source files and write the resulting objects.

Apache Benchmark

We chose this benchmark to represent some of the scenarios that might be experienced on a web server. It uses the apache benchmarking tool `ab` to perform 1,000,000 requests to an apache web server that is started on the local machine using 100 concurrent connections. It then reports the time it took to service all requests and the how many requests the system was able to service per second. Because apache uses multiple processes to respond to these concurrent requests, this benchmark allows us to evaluate the overhead our implementation incurs on workloads with frequent context switches.

SQLite

The *SQLite* benchmark in the PTS simply creates a fresh SQLite database and performs a fixed set of operations on it, reporting the time it took to complete these operations. It was chosen because it completes a lot faster than the other two macro-benchmarks and SQLite is a commonly used in many contexts, especially in desktop and mobile devices. (e.g. the Firefox browser uses it heavily, storing both the history of visited sites and site data such as cookies in SQLite databases)

In addition to these macro-benchmarks that test the performance of real-world applications and libraries, we also decided to include two micro-benchmarks: "mmap" and "stress-ng" to test the worst-case and best-case performance behavior of our implementation, respectively

Micro-Benchmark: mmap

This is a very simple C program (under 50 lines, including white space, formatting and comments) that uses the `mmap` system call to allocate a 256 MiB area of zero filled memory that is not backed by file on the filesystem (i.e. the mapping is created using the `MAP_ANONYMOUS` flag). It also passes the `MAP_SHARED` and `MAP_POPULATE` flags to the kernel to ensure that the memory is already allocated and no page faults will be raised when the memory is written. The program then proceeds to change a single byte in every 4 KiB block of this memory area, which causes the accessed and dirty bits for this 4 KiB page to be set. After this the mapping is deleted using the `munmap` system call. The whole process is repeated 1000 times. This benchmark consists almost entirely of the creation and dropping of page table (entries) by the guest and modification of accessed/dirty bits by the

MMU, making it a worst-case for our implementation that has to emulate all these processes

Micro-Benchmark: stress-ng

The second micro-benchmark we are using is based on the *stress-ng* tool [28] that provides various stress tests for both hardware and operating system interfaces. We chose this tool to simulate a best-case scenario for our implementation by using its `cpu` stress test that runs a wide variety of integer and floating point computations on random data, some of which mirror real-world usage and some of which are entirely artificial. We invoked this tool with these parameters:

```
./stress-ng --class cpu -c 1 --cpu-ops 100000
```

This runs a single instance of the stress test and performs 100,000 iterations, each of which exercise all available computations. This should result in very little overhead from our implementation because there are no almost memory allocations or accesses (causing page table changes) other than the startup of the program and very few context switches because only a single instance is running and the guest is not performing any other actions.

6.3.3 Results

We will now present the results of our benchmarks by focusing on each of our metrics in turn and evaluating how our implementation impacts them compared to the existing SimuBoost code for each of the benchmarks that we performed. We will do this by plotting the fraction of the data gathered with our implementation and data from the run using upstream SimuBoost, representing the increase in probe effect, log size and replay slowdown. We have also included the raw benchmark data that we collected in the Appendix.

Probe Effect

The first metric that we are investigating is the impact that the logging of MMU events has on the probe effect. That is: How much does it slow down the recording compared to the existing logging? As can be seen in Figure 6.1, the workload that is running inside the virtual machine greatly influences the impact that can be expected. Just like we suspected, the *mmap* test case turned out to be a worst-case in this regard, yielding a slowdown of over 400%. While we did not measure if

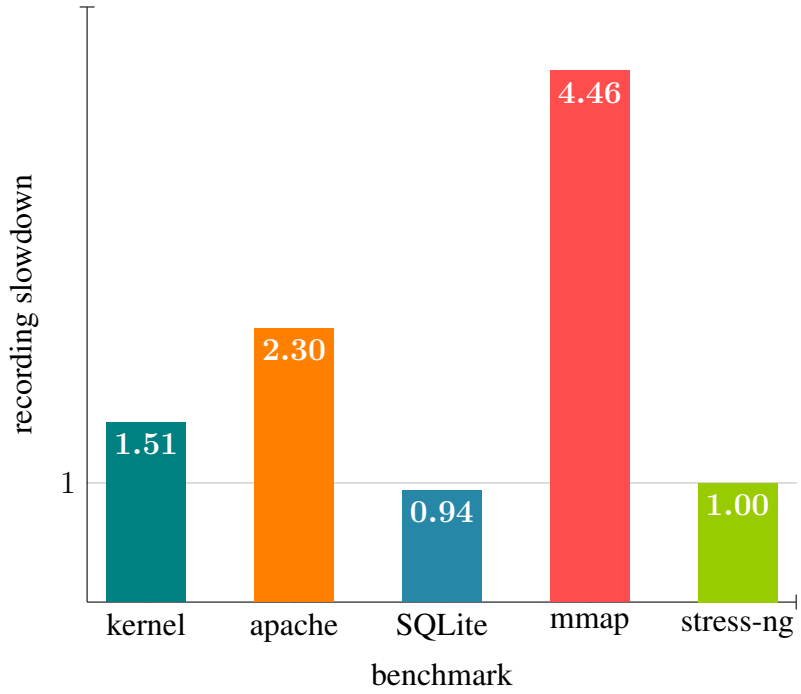

Figure 6.1: Slowdown on the benchmarks' run-time incurred by our implementation, represented by the fraction of each benchmark's run-time while recording with MMU event logging and its run-time while recording without MMU event logging. A value of 1 represents no slowdown compared the existing recording facilities.

this is mainly caused by the page table modifications incurred by the `mmap` system call or the emulation of each write to the allocated memory area, we suspect the latter to be the culprit because none of the other benchmarks showed this extreme behavior. This does however imply that this worst-case behavior is unavoidable when using an emulation based approach like the one we have chosen.

On the other hand, some workloads' performance seems completely unaffected by our changes. Not only our best-case micro-benchmark *stress-ng* yielded the same run-time, *SQLite* did as well (we assume that the slight performance improvement that can be seen in the graph is merely an artifact of our testing setup). While the *SQLite* benchmark is mostly focused on the performance of the system's persistent storage, this confirms that the additional code we had to introduce to check for changed guest page tables before resuming the guest after an exit to user space does not impact performance, at least in this case.

The other two benchmarks, *kernel build* and *apache* exhibit a slowdown between those two extremes. The higher impact on the *apache* benchmark leads us to believe that this slowdown is to a large degree caused by the need to walk all of the guest's page tables on a context switch. The kernel build, which contains higher proportions of CPU intensive computation and disk I/O performs better at only a 50% slowdown. This leads us to believe that a heuristic that would allow skipping this page table walk in some cases as discussed in Section 4.1.1 could be advantageous.

Log Size

As can be seen in Figure 6.2, the size of the replay log was much less affected by the addition of logging support for MMU events and the effects that can be observed are more consistent and less extreme than the impact on the run-time of the benchmarks. In fact, we believe that this effect can largely be attributed to the increased run-time of the benchmarks when MMU events are being recorded, which also leads to an increase in other events that need to be logged and an increased number of checkpoints, which are created at regular time intervals. Figure 6.3, which shows the change in the amount of log data that is generated per second of recording, supports this as well. Even for the *mmap* benchmark, which generated 65,536,025 `MMU_DIRTY` events[1] the log size/second decreased due to the much larger impact on recording time. We believe this to be another good indicator of the performance of our implementation and a good guide of where future efforts to improve it should focus to reap the most rewards. It also shows that for some use cases, the increased absolute log size can be disregarded. If the focus is not on recording a specific amount of work but on recording the last N hours of execution and the impact on the workload's performance can be tolerated, only the amount of data generated in a specific time frame is relevant.

Replay Slowdown

Because the performance of the replay is an important metric for any record and replay system to be useful we are glad that our implementation of replaying MMU events does not adversely impact the run-time of the replay, as can be seen in Figure 6.4. In fact, this could also be presented as a great decrease in replay slowdown, though that would be immensely misleading because it is only caused by the

[1] Curiously it generated only 1087 `MMU_ACCESSED` events. We assume that for the particular invocation of the `mmap` system call that the benchmark performs, the kernel already sets the accessed bit when creating the page table entries, because the `MAP_POPULATE` flag implies that the process will use the memory right away

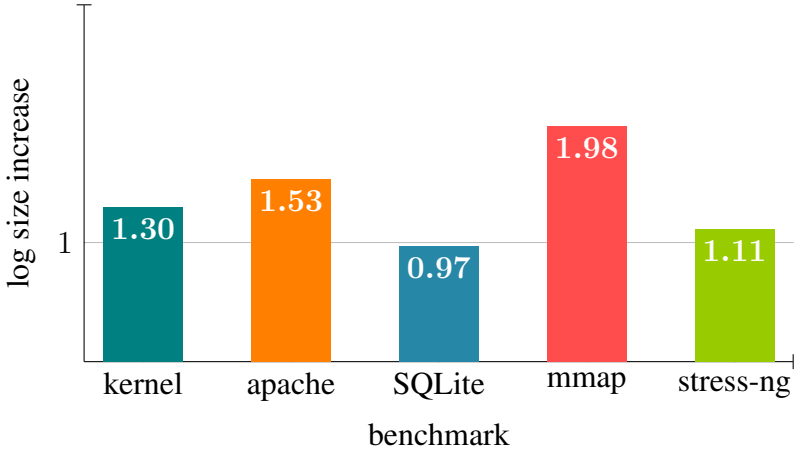

Figure 6.2: Changes to the amount of data generated by the recording, as a fraction of the combined size of the replay log and checkpoints from a recording with MMU event support and those sizes from a recording of the same benchmark without MMU event support. A value of 1 indicates no change in the consumed space.

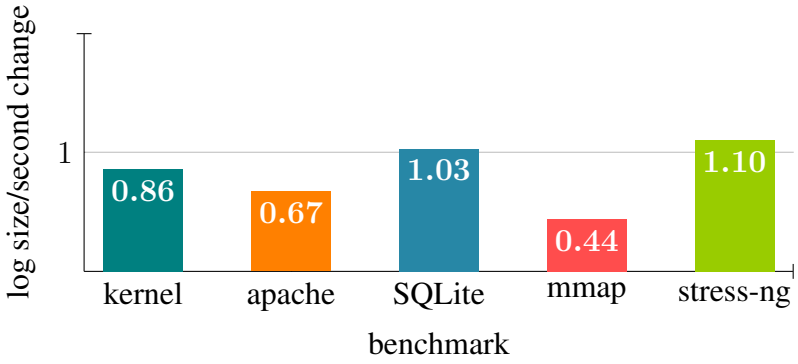

Figure 6.3: Changes to the amount of data generated by the recording per second, due to the addition of logging for MMU events. A value of 1 indicates that the same amount of bytes/s was produced both with and without logging MMU events.

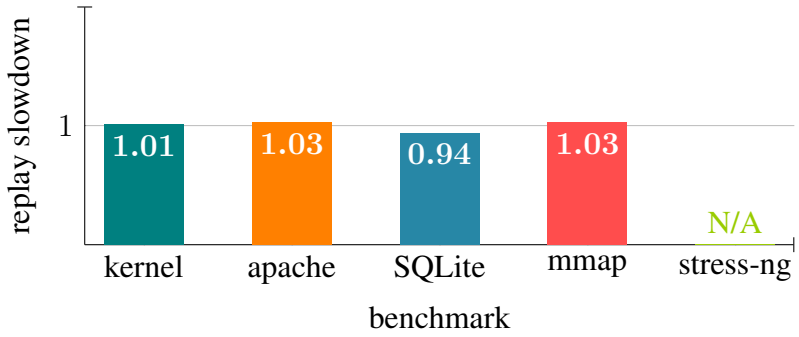

Figure 6.4: Change in the wall clock run-time of the replay of each of the benchmarks, expressed via a fraction of the replay time a recording with logging MMU events enabled and the replay time of a recording without logging of MMU events, for each benchmark.

in some cases greatly increased slowdown during recording. Again, the *SQLite* benchmark seems to actually perform better but we assume this is caused by a slight inaccuracy in manually starting and stopping the recording. These results also validate our decision to choose the simplest possible implementation for replaying MMU events and foregoing any batching of these events during recording as even the staggering number of events generated by the *mmap* benchmark only has a minuscule effect on the run-time of the replay.

The *stress-ng* benchmark failed to replay due to landmark misses both with and without logging of MMU events, which leads us to believe that this is a bug unrelated to our changes to SimuBoost. We were however unable to track down its cause due to time constraints. It should also be noted, that the measurements for the *mmap* and and *kernel* benchmarks had to be slightly extrapolated during replay of the recordings without MMU events as they failed to replay completely due to landmark misses. Thankfully, the amount of time that each checkpointing interval required during replay for these benchmarks was extremely consistent and for the *kernel* benchmark, only the replay time of 5 out of 406 checkpoints (or 10 seconds out of 13.5 minutes of time during the recording) had to be extrapolated. While the number of checkpoints that needed to be extrapolated for the *mmap* benchmark was higher at 55 out of 102, its nature as a micro-benchmark that runs the same operation millions of times in a loop made the measurements even more consistent. The fact that we were unable to observe these failures in any of the benchmark runs we recorded with our MMU event logging further increases our confidence in its correctness.

6.4 Conclusion

In this chapter we evaluated the the correctness and performance of our implementation using multiple benchmarks and determined with a reasonable level of certainty that it is able to record and replay any virtual machine that could be recorded and replayed with the existing SimuBoost code. By correctly recording the non-deterministic behavior of the x86 MMU our implementation is also able to replay recordings that were previously failing due to landmark misses.

Overall, we are satisfied with the evaluation results of our initial implementation in both correctness and performance. We consider the slowdown we were able to measure to be acceptable for an initial prototype using this design. For a mixed workload that does not exhibit certain worst-case behaviors, it could already be used unmodified. Our findings in this chapter also confirmed our suspicions of the remaining bottlenecks that are slowing down the recording compared to SimuBoost. Performance improvements to our implementation are certainly possible and might allow running it on almost all workloads that current record and replay systems can record, with the additional benefit of allowing the replay to succeed in more cases.

Because we re-use the emulation code inside the KVM module, we were apprehensive if our testing would encounter any instructions that are not yet implemented in this emulator. As the x86_64 instruction set contains a huge number of instructions (the manual that documents them spans 2214 pages [29]), they are added to the upstream KVM emulation code only when it is found that they are needed to run a real-world application [30]. Nevertheless we have found it to be sufficient for both a Linux guest and the benchmarks that we used to test our implementation and did not encounter any unimplemented instructions in our testing.

Chapter 7

Conclusion

Existing record and replay systems for virtual machines on x86 CPUs are already able to accurately replay many workloads and only have minimal impact on performance during recording if hardware-assisted virtualization is used. However, some workloads can consistently exhibit problems because the changes that the x86 MMU makes to the accessed and dirty bits in the active page tables can be non-deterministic. Because this behavior is not captured by existing solutions, it can cause the code paths that are taken during replay to diverge from the behavior of the machine during the original recording. In effect, this leads to a complete breakdown of the replay which relies on landmarks — snapshots of the guest's register state — to identify specific points during the replay at which events from the recording have to be injected.

The goal of our work was to implement a system that captures this non-deterministic behavior during recording as part of the SimuBoost record and replay framework and evaluate the feasibility of our approach, both in terms of correctness and performance, using a variety of metrics.

To this end, we modified the KVM hypervisor within the Linux kernel that SimuBoost utilizes during the recording phase to track the location of the guest system's page tables in memory using the EPT hardware virtualization of physical memory. This ensures that no changes to the guest's page tables can be made without the involvement of the hypervisor. We were then able to re-purpose the existing emulation support for x86 instructions in KVM to emulate any instruction that would cause changes to the accessed or dirty bits in the guest's page tables and at the same time transmit these changes to the SimuBoost framework for use during replay. Additionally, we modified QSIMU, the fork of the QEMU emulator used by SimuBoost to perform the replay, to use this additional information during the replay process to perform the exact same changes to the emulated machine's page

tables that occurred during the recording.

When we evaluated this solution we found that it achieved the desired result and was able to replay recordings that would fail to be replayed correctly without our changes, while exhibiting no regressions that we could observe. As expected, depending on the workload that is being recorded, the increased precision of the recording causes a performance hit during the recording. In the benchmarks that we used to evaluate our implementation, this ranged from a 350% increase for a worst-case micro-benchmark to no measurable difference in performance in the best case. Benchmarks of real-world applications yielded results between those values, such as a 50% increase in run-time for a build of the Linux kernel from source. While the amount of data created by the recording increased with our changes, we believe this to mainly be a result of the increased run-time and not the additional logging of MMU events. On the replay side, we were able to show that the addition of support for these MMU events during the replay in QSIMU did not impact the replay's performance in any way, even for the worst-case benchmark that generated just shy of 75000 of these events per second of recording.

7.1 Future Work

While our implementation focuses on supporting the logging of MMU events for uniprocessor guests and evaluating the impact of this change, we believe it could in principle also be adapted to multiprocessor systems with a feasible amount of work. We would however expect a higher performance impact from such a system due to the required synchronization to ensure accurate tracking of the guest's page tables if one CPU is able to modify the page tables that are active on another CPU and due to the increased overhead that the TLB flushes which are necessary when modifying the EPT incur on multiprocessor systems.

Another worthwhile improvement would be evaluating heuristics that would improve the performance of the recording by skipping page table walks and keeping the page tables protected on some context switches, as we briefly mentioned in Section 4.1.1. Our implementation does not yet take advantage of huge pages, which allow a page table walk to terminate at a higher level page table entry, mapping a larger contiguous area of memory and reducing both the depth of the page table hierarchy and the total amount of page tables that are used. While the use of huge pages in the guest would be very advantageous by reducing the amount of work on tracking the guest's page tables and logging MMU events, we had to disable their use in the host because the guest page tables that we write-protect in the EPT are always 4 KiB in size.

Future hardware improvements could also allow decreasing the performance impact that we experienced. For example [31] describes a mechanism that a TLB can implement to indicate its state of synchronization with the guest page tables by adding an additional bit to the EPT entries. As this is currently a very niche application, we do however not expect this to be implemented in hardware any time soon.

Bibliography

[1] Marc Rittinghaus, Konrad Miller, Marius Hillenbrand, and Frank Bellosa. "Simuboost: Scalable parallelization of functional system simulation." In: *Simulation* 1 (2013), p. 1.

[2] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C & D). System Programming Guide*. 2017. URL: https://software.intel.com/sites/default/files/managed/a4/60/325384-sdm-vol-3abcd.pdf (visited on 03/12/2018).

[3] Dr David Levinthal. *Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors*. 2008. URL: https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf (visited on 08/25/2018).

[4] Jim Smith and Ravi Nair. *Virtual machines: versatile platforms for systems and processes*. Elsevier, 2005.

[5] Nikhil Bhatia. "Performance evaluation of Intel EPT hardware assist." In: *VMware, Inc* (2009).

[6] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. "kvm: the Linux virtual machine monitor." In: *Proceedings of the Linux symposium*. Vol. 1. Dttawa, Dntorio, Canada. 2007, pp. 225–230.

[7] Fabrice Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX Annual Technical Conference, FREENIX Track*. Vol. 41. 2005, p. 46.

[8] Fabrice Bellard. *QEMU TCG Implementation README*. URL: https://github.com/qemu/qemu/blob/master/tcg/README (visited on 08/26/2018).

[9] Henrik Thane and Hans Hansson. "Using deterministic replay for debugging of distributed real-time systems." In: *Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on*. IEEE. 2000, pp. 265–272.

[10] Charles E. McDowell and David P. Helmbold. "Debugging Concurrent Programs." In: *ACM Comput. Surv.* 21.4 (Dec. 1989), pp. 593–622. ISSN: 0360-0300. DOI: 10.1145/76894.76897. URL: http://doi.acm.org/10.1145/76894.76897.

[11] 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." In: *ACM SIGOPS Operating Systems Review* 36.SI (2002), pp. 211–224.

[12] Jeff Dike. "A user-mode port of the Linux kernel." In: *Annual Linux Showcase & Conference*. 2000.

[13] Kerstin Buchacker and Volkmar Sieh. "Framework for testing the fault-tolerance of systems including OS and network aspects." In: *High Assurance Systems Engineering, 2001. Sixth IEEE International Symposium on*. IEEE. 2001, pp. 95–105.

[14] Daniela AS 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*. ACM. 2006, pp. 66–71.

[15] Min Xu, Vyacheslav Malyugin, Jeffrey Sheldon, Ganesh Venkitachalam, Boris Weissman, et al. "Retrace: Collecting execution trace with virtual machine deterministic replay." In: *In Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation, MoBS*. Citeseer. 2007.

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

[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*. Vol. 45. 7. ACM. 2010, pp. 13–24.

[18] *TEMU: The BitBlaze Dynamic Analysis Component*. URL: http://bitblaze.cs.berkeley.edu/temu.html (visited on 08/28/2018).

[19] 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*. ACM. 2008, pp. 121–130.

[20] *Goodbye, Replay Debugging...* URL: http://www.replaydebugging.com/2011/09/goodbye-replay-debugging.html (visited on 08/30/2018).

[21] Shaohua Li. *Linux v4.18 Page Aging*. URL: https://github.com/torvalds/linux/blob/v4.18/arch/x86/mm/pgtable.c#L522.

[22] Yunji Chen, Shijin Zhang, Qi Guo, Ling Li, Ruiyang Wu, and Tianshi Chen. "Deterministic replay: A survey." In: *ACM Computing Surveys (CSUR)* 48.2 (2015), p. 17.

[23] Lok-Kwong Yan, Manjukumar Jayachandra, Mu Zhang, and Heng Yin. "V2E: combining hardware virtualization and softwareemulation for transparent and extensible malware analysis." In: *ACM Sigplan Notices* 47.7 (2012), pp. 227–238.

[24] Anton Burtsev, David Johnson, Mike Hibler, Eric Eide, and John Regehr. "Abstractions for practical virtual machine replay." In: *ACM SIGPLAN Notices*. Vol. 51. 7. ACM. 2016, pp. 93–106.

[25] *Linux Page Replacement Design*. URL: https://linux-mm.org/PageReplacementDesign (visited on 09/03/2018).

[26] *Intel 64 and IA-32 Architectures Optimization Reference Manual*. 2016. URL: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf (visited on 09/12/2018).

[27] *Phoronix Test Suite: Open-source, automatic benchmarking*. URL: http://www.phoronix-test-suite.com (visited on 09/11/2018).

[28] *stress-ng - a tool to load and stress a computer system*. URL: http://kernel.ubuntu.com/~cking/stress-ng (visited on 09/11/2018).

[29] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B, 2C & 2D). Instruction Set Reference, A-Z*. 2018. URL: https://software.intel.com/sites/default/files/managed/a4/60/325383-sdm-vol-2abcd.pdf (visited on 09/12/2018).

[30] Stefan Fritsch. *kvm: Add emulation for movups/movupd*. URL: https://github.com/torvalds/linux/commit/29916968c48691c94.

[31] Vyacheslav Vladimirovich Malyugin, Boris Weissman, Ganesh Venkitachalam, and Min Xu. "Synchronizing a translation lookaside buffer with page tables." 9213651. Dec. 2015. URL: http://www.freepatentsonline.com/9213651.html.

Appendix: Raw Benchmark Data

Benchmark Results With MMU Event Logging

benchmark	recording time (s)	log size (bytes)	replay time (s)
kernel	1176.15	13297774	19994.07
apache	1056.52	2375644	3986
SQLite	110.22	894478	306.14
mmap	875.87	630433	2871.78
stress-ng	499.96	395245	N/A

benchmark	#dirty events	#accessed events
kernel	2590	503669
apache	14	1836
SQLite	0	3187
mmap	65536025	1087
stress-ng	1	104

Benchmark Results Without MMU Event Logging

benchmark	recording time (s)	log size (bytes)	replay time (s)
kernel	780.93	10219360	19838.55
apache	458.95	1548252	3860
SQLite	116.72	917537	325.26
mmap	196.595	319060	2789.0475
stress-ng	498.964	357162	N/A