Karlsruher Institut für Technologie

Evaluating Techniques for Full System Memory Tracing

Bachelorarbeit
von

Thomas Schmidt

an der Fakultät für Informatik

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

Bearbeitungszeit: 19. Juni 2017 – 18. Oktober 2017

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

Karlsruhe, den 18. Oktober 2017

Deutsche Zusammenfassung

Die Möglichkeit, Speicherzugriffe aufzuzeichnen, sogenanntes *Memory Tracing*, ist ein entscheidender Faktor bei der Verbesserung der Leistung von Software. Mit der immer weiter werdenden Lücke zwischen Rechenleistung und Speichergeschwindigkeit [27] können durch Optimierung des Zugriffsmusters, zum Beispiel durch Erhöhung der Cache-Trefferraten, die verfügbaren Ressourcen eines Systems besser genutzt werden.

Zwar gibt es viele Tools, die Speicherzugriffe von Programmen auf Basis von *Dynamic Binary Instrumentation* aufzeichnen können, wie Valgrind [29] oder Pin [26], doch sind aber die meisten in ihren Fähigkeiten eingeschränkt, da sie nicht in der Lage sind, die Speicherzugriffe innerhalb des Betriebssystemkerns weiter zu verfolgen, wie z. B. während eines Systemaufrufs. Da dieser entscheidende Teil des Bildes fehlt, können erhaltene Cache-Simulationen irreführend sein, wenn ein Programm häufig die Dienste des Kernels nutzt, was möglicherweise zu falschen Annahmen führt, wie Verbesserungen erzielt werden können. Darüber hinaus ist das Sammeln der Speicherzugriffe des Kernels nicht nur hilfreich, um die Performance von Programmen zu verbessern, sondern kann auch nützlich sein, wenn man versucht, die Interna eines Kernels selbst zu optimieren.

Zudem können diese *Memory Traces* auch nützlich sein, um die Sicherheit der Software zu verbessern, indem sie dabei helfen, kritische Probleme zu finden, die möglicherweise sogar erst von der Toolchain, mit der die Software erstellt wurde, wie zum Beispiel dem Compiler [39], eingeführt wurden. Da die Kommunikation zwischen dem Userspace und dem Kernel über geteilte Speicherseiten gelöst wird, kann ein falsches Zugriffsmuster des Kernels zu einer unerwünschten Privilegienerweiterung führen [39], die es einem nicht autorisierten Programm erlauben, mit den Privilegien des Kernels zu arbeiten.

Diese Arbeit untersucht eine Vielzahl von Möglichkeiten, mit denen man Memory Traces eines Betriebssystemkerns sowie seiner laufenden Software erzeugen kann.

Abstract

The creation of memory traces for full system analysis is very time-consuming, yet it is a vital part of nowadays toolchain for improving software performance as well as for increasing the security of software, by enhancing the understanding of the software behavior.

This thesis examines the use of hardware-assisted virtualization as an alternative to memory tracing based on dynamic binary translation, using Intel's Extended Page Table to restrict the access to memory. Further, both approaches are implemented, comparing them regarding performance and the quality of the data recorded. The experiments show that dynamic binary translation outperforms the proposed system significantly regarding its performance and also has an edge concerning accuracy.

Contents

Chapter 1

Introduction

The ability to create memory traces, that is a set of all the memory accesses a program triggered during its execution, is a crucial part when improving the performance of software. With the gap between processing power and memory speeds widening [27], optimizing the access pattern, by, e.g., increasing cache hit rates, can help make better use of the available resources. While there are many tools out there able to create memory traces for user-space programs based on dynamic binary instrumentation, such as Valgrind [29] or Pin [26], most are limited in their capabilities, as they are unable to trace the memory accesses occurring within the operating system kernel, such as during a system call. With this crucial part of the picture missing, obtained cache simulations could be misleading, if a user-space program is frequently using the services of the kernel, possibly leading to wrong assumptions on how to archive improvements. Further, collecting the memory accesses of a kernel is not just helpful for improving the performance of user-space programs, but can also be useful when trying to optimize the internals of a kernel itself. Moreover, memory traces can also be useful for improving software security, helping to find critical issues, possibly even introduced by the toolchain used to build the software in question, such as the compiler [39]. Further, as the communication between the user-space and kernel does happen to include shared memory pages, a wrong access pattern of the kernel might lead to privilege escalation attacks [39], allowing an unauthorized program to run with elevated privileges.

This thesis will explore a variety of possibilities one might use to create memory traces of an operating system kernel and its running software. Those memory traces can be generated using special hardware tools, which listen in on the computer's internals, such as the CPU pins or memory buses directly, or using simulation-based approaches. While hardware-based tracing might seem superior at first sight, some of its limitations concerning the handling of the vast amount of information as well as the limited insight into the doings of the processor help

software-based tracing shine in a better light. Thus, two software-based tracing approaches will be implemented and further evaluated, one utilizing *dynamic binary translation (DBT)*, and the other based on *hardware-assisted virtualization*, the latter specifically for Intel's x86 processors supporting Intel's *Extended Page Table (EPT)*, allowing to revoke the processor the right to access the memory.

Finally, it will be evaluated how those two methods compare based on their performance as well as on the accuracy of the trace one will receive. It turns out that utilizing DBT has an edge in both those categories.

1.1 Thesis Outline

In Chapter 2, an overview is given of the relevant technologies used in this thesis. Among other things, an introduction to tracing and full-system simulation is provided.

Chapter 3 evaluates the previously introduced technologies and elucidates the reasoning for the choice of the two implementation designs for memory tracing chosen in this chapter.

In Chapter 4, the two previously introduced approaches to memory tracing are implemented, one based on dynamic binary translation and a second based on hardware-assisted virtualization.

Further, Chapter 5 provides quantitative and qualitative results, comparing both previously implemented approaches to one another.

Finally, Chapter 6 concludes this thesis.

Chapter 2

Background

This chapter explains the basics of tracing with a particular focus on memory tracing. Furthermore, it gives a short introduction to full system simulation and some of the possible techniques to implement it.

2.1 Tracing

Tracing describes the process of collecting desired information about the execution of a program based on extensive event logging. Due to the growing complexity of software, it can help developers improve by allowing for a greater understanding of their product based on the recorded data. In practice, this contributes to increasing performance of software [31] and helps to find security vulnerabilities [39]. Moreover, it can be of great importance when developing new hardware, such as microprocessors [21]. The kind of event recorded can vary from being low-level such as instruction traces [8] to higher levels such as tracing communications [28].

2.1.1 Memory Tracing

Traces recording memory accesses of the processor are called memory traces and can be used by hardware designers to create new memory modules [36] or make improvements for future cache architectures [21] with many more possible uses for hardware design alone. Software developers, too, can benefit from memory tracing, indirectly using it in the form of cache simulators to increase the performance of their software [31] or memory error detectors to search for potential security vulnerabilities [39].

One of the many tools for those tasks is Valgrind [1], a software framework shipping with a well-known tool suite for dynamic analysis. However, memory

traces can also be generated in hardware by listening in on the processor's pins directly [15]. Another possible way of creating memory traces in hardware might be to monitor the address and data bus, which is also used by memory-mapped I/O devices to react to requests from the processor. Finally, one can try extracting the desired information from the memory interface, using a specialized monitoring board inserted instead of an actual memory module [4].

There are some crucial challenges to overcome for both hardware and software solutions. Both have to find ways to handle the huge amount of tracing events emitted; this is especially challenging for hardware as the processor might timeout [23] should the tracing implemented to be blocking, as the memory access could be taking too long. In case of asynchronous tracing, the necessary buffer might overflow, and some events could be lost. Further, some software tracers, such as Valgrind, cannot trace memory accesses happening in the kernel space on behalf of the program, e.g., during system calls, as to get those a higher instance is required, such as a simulator.

2.2 Full-System Simulation

A Full-System Simulator can simulate every component of the system, at least detailed enough, so that arbitrary software can run [12]. As one is then running a virtual computer on top of the actual hardware, these kinds of simulations are called virtual machines (VMs) and are sometimes also referred to as guest instances. The simulator itself is known as a virtual machine monitor (VMM) or simply the host.

A common use of simulation is in research and development [12, 14]. When building new hardware, one can try whether the planned project can meet the desired requirements, without the need to develop a prototype, as one could just simulate the hardware. Moreover, the simulator can also be used to start developing software for the hardware before it is available [14]. Furthermore, simulation is vital for security researchers, allowing them to set up save environments, e.g., to study the behavior of malware [30]. Another common use is in the data center, where virtualization is heavily used to make better use of otherwise unused resources through, among other things, server consolidation [22].

To achieve full-system simulation, one needs to simulate the processor, memory and all connected devices. Most domains do only require this simulation to be an abstraction on the instruction set level of the processor, whereas others might need cycle-accurate simulations of all devices, taking heavily into account the implementation details of the desired hardware. Because of that, cycle-accurate simulators are very difficult to build, not only because of the added complexity itself but also because some implementation details are not publicly available and

would need to be reverse engineered [13]. Some simulators, such as Simics [13], approximate cycle-accurate simulators by using a preset cycles-per-instruction value, which still is accurate enough for many computer architects [13].

Although performance is, depending on the kind of simulation, not comparable to native execution [14], with computing resources becoming more powerful and less expensive every year, this downside is becoming less important. Moreover, advances in the way virtual machines can be implemented further help decrease the overhead by multiple magnitudes [14].

2.2.1 Interpretation

The simplest way of implementing a virtual processor (vCPU) for a virtual machine monitor is by using interpretation. It works by setting up a context for the vCPU containing all the registers and other forms of state information required. During the actual execution, every instruction is read and decoded to finally dispatch a function to mimic the behavior of the instruction using a functionally identical reimplementation in code [38]. This approach comes at a significant cost in performance and can be about 1000x slower than native execution [14], as the emulation of every single instruction alone will require tens of actual instructions executed [38].

2.2.2 Dynamic Binary Translation

An improved way of simulating guest systems is dynamic binary translation (DBT). The most significant difference is that instead of reimplementing the instructions in code one maps the desired functionality to native instructions of the host's instruction set, recompiles them as such [38] to finally execute those using the host's hardware directly, still changing the state of the virtual processor, with this alone resulting in a significant speed-up. Moreover, instead of executing the guest instruction-by-instruction, one can execute blocks of instructions and cache their translations [38]. Although the translation might be quite slow itself [38], it amortizes over time as one reuses the cached translations. Aforementioned can be especially useful for loops.

This approach is by multiple orders of magnitude faster, compared to interpretation, with QEMU's Tiny Code Generator, a well-known example further detailed in Section 2.3, only having a slowdown of about 20x [16].

2.2.3 Virtualization

A today commonly used method of implementing virtual processors is using virtualization. Whereas during interpretation one simulates the execution of the instructions, virtualization tries to use the underlying hardware of the host for that directly, even avoiding the additional step of using a binary translation.

The two most prominent ways to achieve virtualization are *paravirtualization* and *full virtualization*, with the latter requiring supporting hardware.

Paravirtualization

Not every instruction set can be used to implement virtualization directly, as most were not designed with this intent [35]. Examples of this are x86 and MIPS, which both only later received support for full virtualization [24]. One of the previous deployed methods was the use of paravirtualization, that is providing adapted interfaces to the virtual machine, to communicate with devices or handle privileged instructions of the CPU, instead of the native ones [35]. As it cannot use the entire native instruction set, the guest's software supposed to run has to be specifically adapted to this environment. An example of this is Disco, a VMM for the MIPS architecture that was demonstrated to run a modified version of the IRIX operating system [9]. Nowadays Paravirtualization plays an important role in improving the performance of spin-locks.

Full Virtualization

Although some instruction sets require the use of paravirtualization, there are others available supporting virtualization natively through either trap-and-emulate or supported by hardware extensions.

Trap-and-Emulate Virtualization based on trap-and-emulate relies on the fact that a CPU has different privilege levels. If the instruction set is well designed, a privileged instruction executed at a lower privilege level should cause a trap [35]. Considering an example where the guest runs entirely on a single privilege level, both its kernel- and user mode, and the VMM is higher privileged so that it can take note of traps caused in the entire guest. Should the guest trap, the VMM can check what kind of instruction caused it and see if the guest, in its current state is, allowed to execute the desired instruction. Then, the VMM appropriately acts on the guest's behavior and emulates the correct response.

Hardware-assisted Virtualization Some instruction sets, such as x86, cannot be virtualized through trap-and-emulate. The reason is that for x86, the `popf` instruction behaves differently, depending on whether it is executed in a privileged or non-privileged mode. As it does not trap, there is no way to correct its behavior [3]. Furthermore, the guest can tell based on its code segment selector that is not running in a privileged mode. Thus it knows that it is virtualized [3].

Due to growing demand, some manufacturers decided to add support for hardware-assisted virtualization to their platforms, using special virtualization extensions. For the x86 platform, Intel and AMD both implemented virtualization extensions that while similar, as both are designed to mimic a classical trap-and-emulate, are not compatible to one another.

Intel Virtualization Extension Intel's Virtual Machine Extension (VMX) introduces a whole lifecycle for the management of virtual machines, making sure those cannot detect they are run virtually and allowing the VMM to choose under which circumstances to exit the guest granularly [19].

Before the guest can be entered, the VMM has first to enable VMX and prepare a virtual machine control structure. This structure contains all the relevant information for a logical processor to enter the guest, as well as necessary information such as under which circumstances to exit the guest. It also stores the state the host was in just before entering the guest [19]. A while after the VMM launched the guest, an exit will occur. Based on the provided exit information the VMM can now determine why the exit occurred, performing actions as it sees fit [19]. Those actions could among other things be I/O operations, page faults, or preprogrammed timeouts of the VMM, allowing for context switching.

Intel Extended Page Table Although all this allows for full virtualization on x86, the first iteration of VMX was not resulting in a universally measurable performance gain compared to at the time available software VMMs based on dynamic binary translation, such as from VMware Inc. [3]. The most significant source of overhead in the VMM at that time was the translation of virtual addresses from the guest (guest virtual addresses) to physical memory addresses of the host (host physical addresses) [3].

As the guest uses virtualized memory, a translation has to happen from the guest's physical memory to the host's physical memory. This translation could just be done using the existing MMU, but as the guest is likely going to use paging, a second translation has to occur from the guest's virtual addresses to guest's physical addresses. As there was no hardware available to support this need, VMMs addressed this problem by implementing MMUs in software, using a so-called shadow page table, which maps guest virtual addresses directly to host

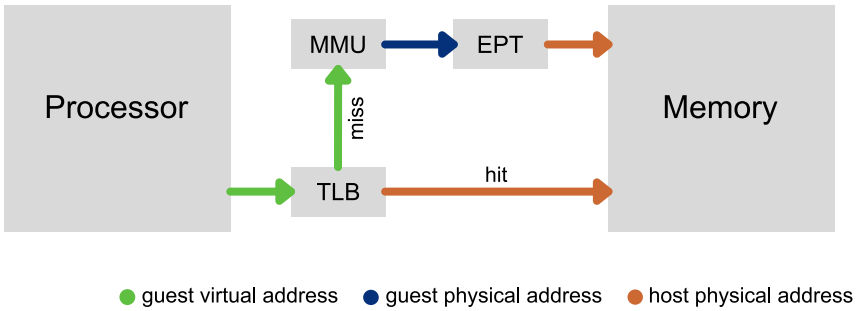

Figure 2.1: On a memory access, the processor forwards the guest virtual address to the TLB for translation. Should the cache not contain a suitable translation, it will trigger a page walk on the MMU and EPT to receive the host physical address necessary to access the memory.

physical addresses, in turn allowing the translation to happen using the existing hardware [35], eliminating the overhead for page walks in software. However, as the Software MMU is a part of the VMM, on every change, control had to be handed over to it by exiting the guest, allowing to adjust the shadow page table accordingly. Concidering that operating systems tend to change their page tables frequently, this results in significant overhead that one can avoid [35].

With the implementation of the Extended Page Tables (EPTs) in a following revision of VMX, this issue was addressed. On top of the usual translation of a virtual address to a physical one by the MMU, a following secondary translation is introduced, before the memory is accessed. This second translation is similar to a regular MMU, as it takes a given address and translates it based on a provided page table. Overall, it allows to grant the VM full access of the MMU, yet the VMM can manage, which parts of the host's memory space the guest will be able to see based on the EPT [19], thus not losing any control.

Considering an example in which the guest is actively executing. To access the memory, a host physical memory address is required. As seen in Figure 2.1, on a memory access the guest virtual address will be used to first loop up for a translation in the TLB. Should it be successful, the host physical address is put out and one can interface the memory. However, should the TLB not provide a cached translation, a page walk of the MMU is necessary to receive the guest physical address. Directly following is a translation of the latter to the host physical address desired, based on a page-walk of the EPT, now allowing to interface the memory correctly.

2.3 QEMU

One example of a full system simulator is QEMU [6], an emulator with its primary focus being running a target's operating system on a host that is already running an operating system. It can emulate the processor, allowing to run a target with a different instruction set than the host's hardware, or use a processor supporting virtualization for the execution for increased performance. Moreover, it can also emulate devices, such as a VGA display or hard drives, and further supports paravirtualization [20].

2.3.1 Tiny Code Generator

Based on dynamic binary translation, the Tiny Code Generator (TCG) is at the core of QEMU when emulating the processor. A guest's code will first be translated at the frontend to an intermediate instruction set format to be further later converted into the host's instruction set at the backend [2]. An example of how this looks like can be seen in Figure 2.2. While one might see this as an undesired overhead at first, it allows for a significantly cleaner codebase, as not every combination of guest-host translations has to be implemented on its own, but only the frontend and backend for each instruction set once.

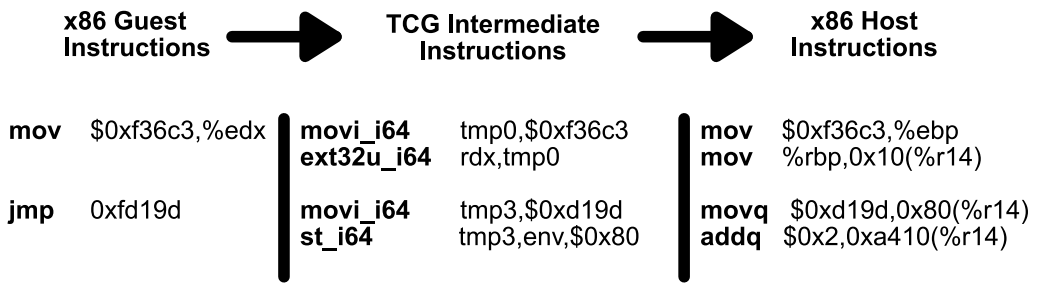

Figure 2.2: The guest's instructions are translated into the TCG's intermediate instruction format. From there on, they are further translated to the instruction set used by the host.

The introduced intermediate instruction set is "RISC-like" [7], meaning there are only a few, straightforward instructions supported directly. This approach can result in a single guest instruction being translated into a set of TCG instructions, possibly resulting in poor performance for complicated ones, which is why there is also a call instruction, allowing to dispatch so-called helpers [7]. Those helpers are functions with their implementation being a part of QEMU itself and thus might be optimized by the compiler, which could be an advantage compared to

coding those based on the TCG instruction set, while moreover, they can access resources available outside of the virtual machine thanks to this.

2.3.2 KVM

Since the advent of virtualization extensions for x86 processors, the Linux kernel received a new subsystem called kernel-based virtual machine (KVM) [22], to create virtual machines using an *ioctl()*-based API. It is important to note that AMD and Intel have integrated their virtualization extensions differently and provide different capabilities [22].

After creation, the VM's virtual processors are not scheduled on their own but require a user-space program, such as QEMU, to call them using the aforementioned *ioctl()*-based API, elevating the running thread into kernel mode. At this point, the kernel will repeatedly cause the virtual processor to run using the virtualization extension. On every exit of the guest, the kernel performs necessary actions, such as handling a page fault. Should the exit be due to a pending signal on the host's running process, or due to an I/O request, possibly caused by an I/O instruction [22] or access of MMIO memory, the kernel will return to the user-space program to handle the case appropriately.

Further, the user-space program has to assign memory to the virtual machine, which it is supposed to use. Each continuous memory region, called *memory slot* by KVM, is a part of the program's virtual address space and can be mapped into the guest's physical address space, but never be resized or removed [11].

Chapter 3

Analysis

Memory tracing can help improve software products by allowing for tools to find security vulnerabilities [39] as well as helping enhance overall performance [31]. Moreover, it is a valuable tool in hardware research [21].

Independent of the actual implementation of the tracing mechanism, there should be a universal hook, passing on the required information about the access itself, allowing one to add the functionality desired, be it recording the trace to disk or running online analysis, as one sees fit. In general, one will be interested in collecting information about a memory access, such as the physical memory address, virtual address, size of the access as well as a point-in-time reference to when it happened, with the latter at best being able to differentiate between each cycle of the processor, although the instruction level would generally suffice.

While being able to create a memory trace for user-level software is easily possible using tools like Valgrind [29], it is more difficult if one is interested in memory accesses of the operating system. For memory tracing to work in such an environment, one needs to be able to control the code execution of the operating system itself using, e.g., simulation or by listening on the hardware directly.

3.1 Hardware Tracing

In general, most hardware solutions for memory tracing will share the same characteristics, as they are off the processor chip and their data source is necessarily the same. For example, they will be unable to access the current instruction pointer, and possibly, they will not even be able to interact with the memory bus directly, as some vendors integrate it into the processor as well [19]. A schematic example of a such-like processor interfacing with memory and devices can be found in Figure 3.1. Thus, one should only assume information accessible by the CPU pins and inferred data is available to a hardware tracer.

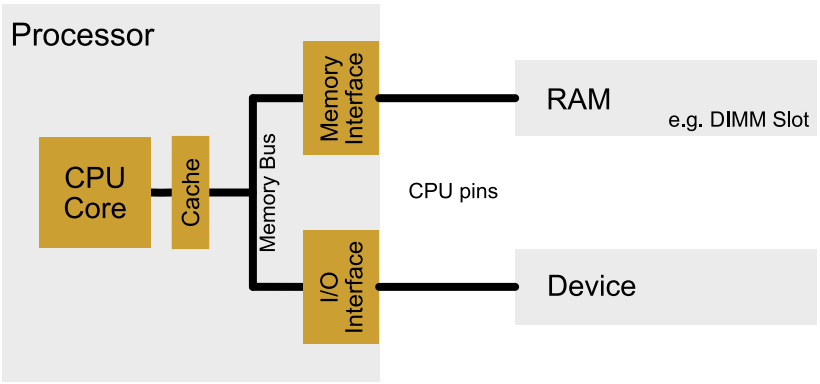

Figure 3.1: Hardware-tracing techniques can only source their information directly from the CPU pins and interfaces inheriting their information from those. They cannot access the CPU core directly to, e.g., obtain the current instruction pointer.

The most significant advantage of hardware tracing is, that, if well implemented, a trace of all physical memory accesses of the processor can be generated without influencing the traced system by slowing it down, allowing for no time distortion [15]. This in turn allows for those full-system traces to more closely match the actual execution. However, such a system is not easy to build due to the high volume of events triggered during memory tracing, as already described in section 2.1.1. Two prominent examples are BACH [15] and HMTT [4], each using a slightly different approach. BACH creates memory traces by listening in on the CPU pins directly, storing the trace events in a local buffer and once reaching a certain threshold, halting the processor to store away the trace, resuming afterward [15]. This halting, which takes about 40 seconds [15], will, however, result in time distortion compared to wall-clock time, which might still influence the behavior of the traced system. In contrast, HMTT uses a hardware board inserted into the DIMM slots of a computer system; the idea is to offload the trace events during the actual tracing process itself, making it unnecessary to halt the system. Although this sounds promising, a more recent report from the same authors written in 2014 [18], makes clear it is tough to scale. Apparently, the bandwidth required to offload the trace for memory modules up to DDR3-800MHz is already 8Gbps [18]. It should be noted that at the time of this writing DDR4 with speeds of up to 3200MHz is available in consumer-grade systems. Furthermore, modern processors support multiple memory channels, increasing the overall memory

bandwidth, making the issue of offloading the enormous amounts of trace events even harder.

However, there also are some disadvantages to hardware tracing, starting with the fact that only physical memory addresses can be recorded directly. Further, should one be interested in tracing all memory accesses of the processor, using a solution that inspects the memory interface, such as HMTT, the solution misses out on MMIO accesses, as can be seen in Figure 3.1. However other hardware tracing mechanisms, such as BACH, might be able to retrieve this information, by either observing the memory bus, or by observing the processors I/O pins should the memory controller be integrated. Moreover, HMTT cannot directly differentiate between DMA memory accesses and the processor's accesses [5], while a system monitoring the processor pins will be unable to detect the first at all. To achieve this, HMTT requires assistance of a hybrid build, using a kernel extension in the traced operating system [5], to know whether a memory access is caused by the processor or a connected device. Such an extension can further help by providing semantic information, such as virtual addresses, but it influences the traced system and thus leads to a different memory trace.

Additionally, to be able to collect all the memory accesses from the processor, one needs to disable all on-chip caches and possibly off-chip caches between the processor and the tracing solution used, which can result in a 10x to a 100x slowdown of the execution [5]. Another concern is portability, as hardware tracers use particular interfaces, such as DIMM slots or a variety of CPU sockets [5], making reuse of the same tracer difficult should the hardware setup change. Last but not least, as this hardware has to be bought or expensively build in-house, there is also a significant cost component to it as well.

3.2 Software Tracing

Simulation-based approaches to memory tracing are an alternative, yet should one require cycle-accurate traces, the slowdown can be from 1000x up to 10000x [4], making this approach very time-consuming. For software development, however, this level of detail is usually not necessary. Using software tracing will cause time dilation, due to the added overhead of collecting the needed information for the event and processing it as required, resulting in this approach, in general, being less accurate than hardware tracers. This overhead causes, e.g., a triggered interrupt to arrive relatively before it would otherwise have, as the virtual processor's execution is slower, effectively making the device seem faster than it is. However, even delaying the interrupt to appear at the right point in time for the execution based on an instruction counter would not help, as devices, such as network cards, might start queuing up events for incoming data that they otherwise

would not have. Moreover, in cases where the processor is simulated the executed instructions might not match the implementation of the hardware accurately, be it due to performance optimizations or because it is proprietary knowledge, possibly causing a different memory access pattern as it would be the case in hardware, introducing further differences to a hardware trace.

On the upside, as the tracing is happening on the same chip, as one is utilizing simulation or more advanced methods like hardware-assisted virtualization, one can access more sources of information than a hardware tracer can access, without the need to add software to the observed system itself. Examples of this are among others virtual addresses, and further, as all devices connected to the observed system will need to be virtualized, MMIO and DMA accesses are traceable without much effort. Should one use simulation to create a memory trace, porting and deploying it to other kinds of computing systems is quickly and inexpensively doable. Although less portable, virtualization-based tracing can also be quickly deployed, as long as the platform supports it.

As the already stated time dilation is unavoidable for software tracing, one's goal should be to minimize it as much as possible. For that, multiple solutions may be plausible.

3.2.1 Simulation-based Tracing

One option would be to use a simulator based on dynamic binary translation, as using a simulator based on interpretation would be unnecessarily slow. Aside from portability, a significant advantage of this approach is that, as it is entirely based on simulation, no hardware extensions for virtualization are necessary.

When executing a guest using dynamic binary translation, every instruction is translated into a set of instructions the host can run natively, as described in Section 2.2.2. To add support for memory tracing, one needs to modify the translation process by adding extra instrumentation for every of the guest's instructions accessing memory. This instrumentation, as it is in line with the guest's execution, can collect all the desired information about the access itself, and pass it on to a hook for further use. However, to be able to get a point-in-time reference, further modifications to the translation process will be required, such as the addition of an instruction counter. For those instructions executing multiple memory accesses, one will need to make sure that their monotonically increasing order will be preserved when recording them.

A possible canidate for interpretation-based memory tracing would be Simics [13], however, the slowdown for an implementation with an empty hook alone can be about 810x slower compared to native execution [34]. A better choice for a tool would be to use QEMU [6], which does not offer a way to trace memory

accesses of the guest, making it necessary to implement this oneself. Without the required instrumentation to trace the memory accesses added, slowdowns of only about 20x compared to native execution [16] can be measured, whereas an implementation supporting memory tracing with an empty hook has a slowdown of about 31x [34], outperforming Simics by order of a magnitude.

3.2.2 Virtualization-based Tracing

Hardware-assisted virtualization allows to significantly improve the performance of a virtual machine, compared to dynamic binary translation, resulting in almost native performance. Thus, the time dilation one will experience for a tracing mechanism could be significantly reduced. Further, with increasing server consolidation in data centers thanks to virtualization, one might even argue that the usual slowdown of a hardware-assisted virtual machines is closer to reality than a bare metal execution.

PinOS Aside from Valgrind [29], another popular framework for dynamic analysis of a user-space program is Pin [26], created by Intel Inc. Based on this framework, an extension was build called PinOS [10], supporting the creation of traces for the whole system, including the kernel space.

PinOS is built on top of Xen [32], which uses Intel's virtualization extension to run guest systems. However, instead of directly executing the guest's code, the VMM executes Pin, which is instrumenting every instruction the guest desires, allowing to keep full control over the execution of the guest on an instruction-level while also being able to use virtualized devices provided by Xen [10].

However, even when not running any analysis on the guest, PinOS seems to be about 50x slower than native execution, based on an average of the benchmarks provided by [10]. This slowdown is due to the overhead of the dynamic instrumentation Pin uses to add tools such as memory tracing.

Pure VMX-based Tracing In contrast to dynamic binary translation or PinOS's instrumentation approach, where the execution is entirely influenceable and controlled by the VMM, using pure virtualization, the VMM only prepares the VM's context, then it hands over control to the processor for the actual execution of the VM, which only returns, e.g., when an interrupt was triggered or the VMM's services are needed to, e.g., handle I/O requests. Due to this loss of control, as one cannot force the VM to exit at any desired point, the tracing of memory accesses is not as straightforward to implement and requires multiple steps.

If the processor needs to access memory and cannot do so for any reason, it will cause a page fault. For processors supporting virtualization with nested page

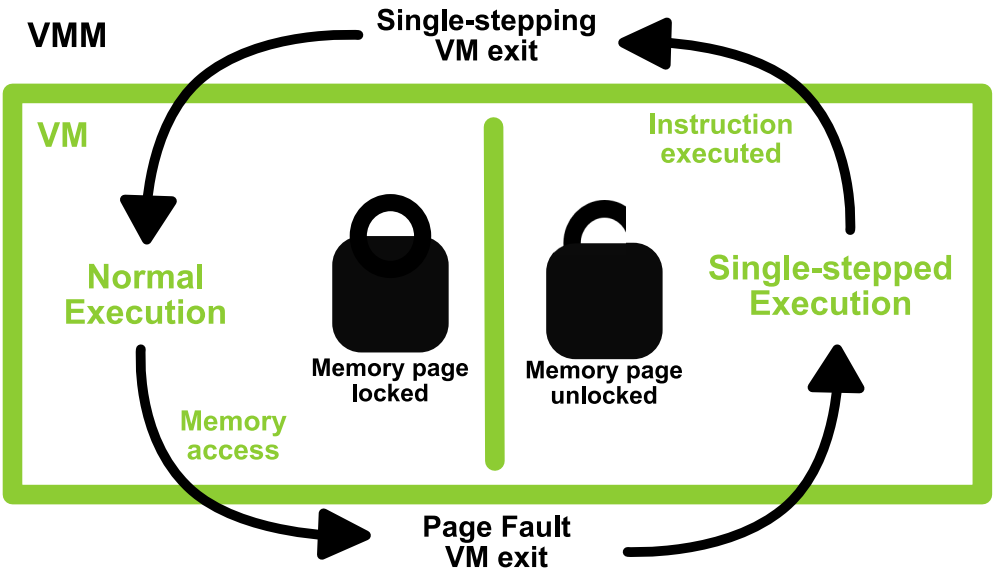

Figure 3.2: Once a memory access occurs, the VM will exit due to the page fault. The page will be unlocked and single-stepping will be enabled. After resuming the VM, it will again exit once the instruction is successfully executed, providing an opportunity to lock the memory page again. Finally, the execution continues normally.

tables, this means a running VM will exit and return control to the VMM, should the page fault be due to the nested page table. Based on this observation, one can get a virtual machine to return to the VMM on every memory access, by setting up every guest physical frame to be not read- and writable.

To achieve memory tracing, first, the entire memory address space will be access protected, by revoking the processor the right to read and write to the individual pages using nested paging. On every page fault that is to be handled by the VMM, one check if the cause of it is due to the set page protections. Should this be the case, the restriction will be temporarily removed on the desired page, depending on the need for reading or writing, while also making a note of the physical and virtual address of the access, its size and other requested information. Furthermore, the guest will be instructed to single-step over the next instruction, exiting again right after its execution, so the page protections can be set in place again, preventing any future accesses from slipping through. After the instruction was successfully executed, all collected information about the possible multiple memory accesses that happened up to this point will be passed on to a hook responsible for the further use of the trace events. At this point, the VM can resume normally. This workflow is visualized by Figure 3.2.

Getting accurate timing information, however, will require the hardware supporting instruction counters. Without this, as the VMM is not controlling the execution of the guest directly, as, e.g., DBT is for its guests, no real assertion can be made, which would only allow for recording the order in which the memory accesses happened. Further, as it is necessary to exit and reenter the VM twice in order to be able to execute a single, writing instruction, it is clear that there is time dilation, which will ultimately result in a slowdown of the execution.

3.3 Conclusion

As the most significant disadvantage to software tracing is time dilation, which even when using hardware tracers might still be an issue, whether directly due to the need to halt the system or because one needs to use slower hardware components, such as memory modules, to slow down the event generation, than one may like, in the following only methods for creating software traces are further discussed.

Concerning the possibly different internals of an instruction, due to a different implementation on the microcode level, one has to admit that those are not set-in-stone and might change for future hardware iterations. Thus the simulator can just be viewed as a different revision of the instructions inner workings, should it result in a different access pattern. Further, such differences will only be detectable on a sub-instruction level, with everything on the ISA level intact and the overall program's flow not influenced.

Based on the previous discussion, it is clear that a purpose-built modification of QEMU, is able to outperform PinOS, as even including the required, additional instrumentation for just adding a hook for memory tracing only results in a slowdown of 31x [34] compared to 50x of PinOS [10]. However, due to the in general significant performance advantage for virtualization compared to dynamic binary translation, as used by QEMU, and due to the lack of information about the performance of pure VMX-based tracing as proposed before, it is appropriate to compare both those approaches.

Chapter 4

Design and Implementation

This part of the thesis is going to focus on how to implement the two different approaches to memory tracing previously described in Section 3.2: one based on dynamic binary translation and a second based on hardware-assisted virtualization. To allow for a similar foundation for the upcoming evaluation, QEMU will be used for both of them, resulting in the use of the same devices and background logic. That way the differences one might experience will be only due to the different technologies involved, namely the Tiny Code Generator (TCG) and KVM, as well as due to the tracing implementations.

4.1 Adapted QEMU by Simutrace

For this thesis, an already adapted version of QEMU is being used, which serves as a reference implementation of memory tracing designed for the use with the Simutrace framework [33]. Simutrace is a tracing backend, allowing for side-effect free recording of memory traces when acquiring the trace using simulation. It is based on a client-server-architecture, having each instance of the simulator create its own session on a server to store the tracing events [33].

In the following, this variation of QEMU, based on v2.6.0, will be referred to as QSimu. However, at this point, QSimu does only support tracing stores to memory, but not loads. Thus, for the upcoming evaluation and implementation, only write accesses will be further considered, as this will be enough to obtain quantitative and qualitative results for the planned comparison.

As described in Section 2.3, QEMU uses the Tiny Code Generator to translate the guest's instructions first to an intermediate format of TCG instructions, to later compile those into the actual instructions executed by the host. Tasks too complicated to implement efficiently based on the TCG's instruction set alone, will call helper functions, that are a part of QEMU itself, similar to a dispatch func-

tion for interpretation-based simulation. To introduce memory tracing, QSimu uses modified TCG instructions, adding a helper method that passes on the information about the access to the Simutrace framework. It is called after the store is performed. Further, helper methods might access the MMU directly. Thus, their interface to the MMU has also been modified by adding a call to the tracing facility.

4.1.1 Striping Dependency on Simutrace

As it seemed the most useful only to evaluate the added overhead of the tracing process itself, instead of also including measurements for possible use cases, such as recording to Simutrace, the QSimu implementation used in this thesis is adapted, leaving just the hook that is supposed to pass on the access event, removing references to the Simutrace framework. This allows limiting the scope of the implementation for the upcoming KVM-based tracing implementation to the tracing facility itself, not requiring the addition of a use case. As shown in Section 3.2.1, it is important to leave the hook as well as the acquisition of the tracing information, as otherwise the slowdown measured will not be accurate.

Thanks to a thoughtful implementation of the tracing hook and the Simutrace developers leveraging the existing *tracetool* of QEMU for their final integration with their framework, one only needs to modify their provided script that usually generates the functions interfacing Simutrace. By removing the contents of those, leaving them empty, we are just left with the hook able to access information such as the kind of memory access and its size, the virtual or physical address that resulted in the access and its data, the current cycle count, as well as the current state of the virtual processor.

Finally, one only has to clean the functions used to initialize the connection to the Simutrace server, as well as those used to setup and teardown the session recording the trace events. All of those are located in *trace/simutrace-qemu.c*. Special care has to be taken for some of those methods, as they expect a valid return value.

4.2 KVM for Memory Tracing

In the following, memory tracing is prototypic implemented using the approach described in Section 3.2.2, to check whether or not this approach to memory tracing is superior to tracing based on dynamic binary translation in term of either execution times or the accuracy of the information acquired.

Based on the discovery that QSimu can only trace write accesses to memory at this point, the scope of the virtualization-based implementation is reduced to just

this functionality as well to ensure better comparability for the upcoming evaluation. Besides, we will restrict this implementation to be only for a single guest processor. Because the virtualization extensions of AMD and Intel are different, as stated in section 2.3.2, the following implementation is further limited to only support Intel's x86 processors with VMX and EPT.

Just like QSimu, the implementation is expected to be able to extract a virtual and physical memory address, the size of the access, its kind (in this case writes only), timing information, as well as the data written to memory for each memory access. However, as this is just a prototype, in some instances, just an idea for an actual implementation is provided.

The following implementation is based on the Linux kernel v4.3, leveraging KVM's existing dirty logging mechanism as far as possible, which is used to efficiently detect, whether a page fault is caused due to set write protections and if so have them removed. On the user-space side, the adjusted QSimu implementation is used, with its only further adjustment being the addition of a monitor command to enable the memory tracing process for KVM's running VM.

4.2.1 New Monitor Command in QSimu

By default memory tracing is disabled. It is necessary for the user to invoke it using an *ioctl*, that is a function to control various aspects of devices on a Linux host [25], such as KVM. The added ioctl `KVM_START_MEMTRACE` does not accept any parameters, although future work might decide to pass on information to limit the scope of the tracing process by regulating whether reads or writes shall be traced, or what data shall be collected.

Within QSimu, a human monitor command was added in *hmp-commands.hx*, referencing a function to enable the tracing process within KVM by calling the previously introduced ioctl.

4.2.2 Architecture of the System

The overall control flow of the memory tracing implementation will be similar to the following: By write-protecting each page of memory, the processor cannot access those directly and will invoke the VMM on a write access. The VMM can then make a note of the access, acquiring the desired information about it, continue by removing the protection of the requested page and finally set up the guest, so it only runs for a single instruction. After this instruction is executed, the VMM will re-protect the page, while also committing any open memory accesses to the receiving hook. The decision to pass on the information to the hook only after returning from the single-stepped instruction will be implemented this way, as an imminent interrupt might change the flow of the program before the VM

gets the chance to execute the instruction. In this case, the implementation will rollback, protecting the previously unprotected memory region and not commit the write access.

The described implementation for the memory tracing is built directly into the KVM module. Respectively some existing structures, like `struct kvm`, representing the state of the entire VM, and `struct kvm_vcpu`, describing a single virtual processor, were modified to contain the necessary state information.

The `struct kvm` was adapted to be able to hold the current state of the tracing process, which for this prototype is just on or off, by adding a boolean. Further, the `struct kvm_vcpu` received a new property revealing whether this implementation's single-stepping is currently activated for that virtual processor, as well as a bounded array of currently uncommitted memory accesses. Those are memory accesses to which the causing instruction has not been completed, yet.

4.2.3 Initialization Phase

Handling the incoming `KVM_START_MEMTRACE` ioctl from QSimu, the kernel will first check, whether the tracing process is possibly already enabled and should that be the case will immediately return.

After that, the function iterates over all the available memory slots, which are mappings of guest physical addresses to host virtual addresses of the invoking user-space program, looking for all valid ones. For those memory slots, all pages will be write-protected using `kvm_mmu_write_protect_pt_masked()`, as provided by the dirty logging mechanism. As this mechanism does not support large pages, support for those is disabled entirely[1]. Furthermore, once done the TLB is flushed, ensuring that no stale translations might be used to bypass the new protections.

Finally, a custom flag in the VM's `struct kvm` is set to reflect the fact that memory tracing is now enabled. Aside from the tracing process itself, only the generation of *shadow page-table entries* will be influenced by this flag. Thus, to ensure every newly created or modified page mapping will still reflect the fact that memory tracing is enabled, write permissions are removed.

[1] This was archived by setting the global variable *largepages_enabled* within the KVM module to false. Thus, it will also influence virtual machines with memory tracing disabled.

4.2.4 Behavior on Write Access

Whenever the processor attempts writing to a memory address, but cannot do so, the guest will exit, invoking the VMM to try resolving the issue. At this point the VMM is provided with an *exit reason* and can access the guest's physical and virtual memory addresses for the fault. This information is provided as part of the *virtual machine control structure (VMCS)*, representing the state of the VM with Intel VMX. The CPU also updates the exit qualification field, which supplies further information on the exit, such as whether a memory access was a read, write, or an instruction fetch. Finally, information about the timing is received using the retired instruction performance counter from Intel's Architectural Performance Monitoring [19].

Memory-Mapped I/O Access KVM itself is unaware of the devices available to the guest. As described in Section 2.3.2, it will return to QEMU in user-space once I/O happens. However, to allow for efficient exits it sets incorrect bits in the EPT's mapping for every page that is assigned to MMIO, causing an *EPT misconfiguration exit*. On this exit, the VMM will verify whether a real misconfiguration happened, it it is because of MMIO. Should it be due to a MMIO write, the access is recorded, should the tracing be enabled.

However, at this point, no direct information about the access size nor the written data is available, as those are not directly provided by Intel VMX. It is, however, possible to get the size of the memory access as well as the source of the data, thus the data to be written, by decoding the faulting instruction. This idea is not implemented, as KVM's integrated instruction emulation does not support the entire instruction set, leaving out vector instructions, which may also access the memory.

Normal Memory Accesses Most of the time a memory access will not be directed to devices, but the RAM. If memory tracing is enabled, every write access will cause an *EPT violation exit*, due to the set write protection, resulting in a return to the VMM. The VMM will consider this a typical page fault and proceed accordingly, eventually assuming the exit occurred because of the dirty logging mechanism, which is utilized for this tracing implementation.

If the page fault is caused due to the write protection, the protection of the guest page in question will be removed. Further, we enable the single-stepping flag of `struct kvm_vcpu`, ensuring that single-stepping will be activated just before entering the VM again. This provides a way to re-enable the write protection, flush the TLB, and commit the write access to the hook after the instruction has been executed. At this point, some information about the memory access has

to be already collected, such as the virtual and physical memory address of the access, as those will not be available again at a later point in time. For this prototype, only the physical memory address is collected, although the virtual address could be acquired as well, by reading it from the VMCS. Further, just like for MMIO accesses, we are not provided with the access size directly nor the data but need to decode the current instruction, which was not implemented.

Before resuming the virtual processor, within the `vmx_vcpu_run()` function, this patch checks the `struct kvm_vcpu` if single-stepping should be enabled. In general, there are multiple possible ways to implement single-stepping. Setting the *trap flag* of the `EFLAGS` register is the most straightforward way, but this would allow the guest to observe this fact, as the guest's registers would need to be manipulated. Therefore, a better choice is the *Monitor Trap Flag (MTF)*, which can be set as part of the VMCS, allowing to single-step the virtual machine without its knowledge [19].

Return after enabling single-stepping Should the next exit, after enabling single-stepping, be due to an *EPT violation* again; the VMM repeats the steps outlined in the previous paragraph for the new faulting address. This, e.g., might happen for unaligned write accesses, where a second memory page will need to be unlocked.

For any other exit reason, the patch starts by re-enabling the protections on the previously unlocked pages, by resolving each access's physical address to the slot they belong to, then using this information to look up all shadow page table entries for this address utilizing the reverse map. Now, the kernel set setting the protection again.

Should the exit be an *MTF exit*, further evaluation is required to see whether the last instruction has been executed by now, as a variety of factors can influence the actual execution [19], however, for every other exit reason, we deem the write to be unsuccessful and rollback, by re-enabling all write protections and not committing to the tracing hook. Using the instruction counter, we can determine, whether any instruction was executed and if so, we know the write was successful. However, *REP*-prefixed instructions will not increase the instruction counter for every iteration, but only the last, which is why we also deem it a successful execution, should both, the instruction pointer and the counter not change. Once we have a successful write, all previously in the `struct kvm_vcpu` accumulated memory accesses can be committed to the tracing hook.

However, the evaluation later revealed, that this heuristic does not behave correctly, should an interrupt be injected into the guest from the outside or should an

interrupt appear from internally at the same time when single-stepping is enabled. As the VM will first try to set up the stack frame for the upcoming interrupt, this patch possibly allows a series of write accesses without tracing those, as setting up the interrupt cannot be single-stepped though in contrast to normal instructions.

VMX handles the particular case of an injected interrupt with single-stepping enabled, by first setting up the stack frame, then returning to the VMM using an *MTF exit* without modifying the current instruction pointer [19]. This implementation will, as explained above, assume this to be a successful write, even though the actual instruction was not yet executed. Although it seems impossible to trace all the memory accesses, leaving the accuracy always being suboptimal, one can improve it by trying to handle the case of injected interrupts separately. As the VMM is aware of all injected interrupts, this should trigger a special execution path that first is to lock all previously unlocked memory pages and ignore previous uncommitted write accesses. Now, single-stepping is to be enabled and every further memory page should be granted access to, up to the next *MTF exit*, as at this point the interrupt is successfully injected, with the only thing left to do is committing the accounted writes and re-protecting the memory pages.

Making matters worse, in case of an internally generated interrupt, consider the interrupt happening right after re-entering the VM with single-stepping enabled. While preparing the stack frame for the interrupt, the processor can simply request further memory pages, which the VMM has to assume are for the current instruction, as there is no signal to the VMM what caused the page fault. Eventually, an *MTF exit* occurs, after the stack frame is prepared, and the instruction pointer will change, without the instruction counter increasing, resulting in all writes being ignored. A possible solution to this would be to simply commit writes under the previously stated conditions, yet, this would also commit some memory accesses of the current instruction, which are not fully executed yet and are in need of re-execution at a later point in time.

Chapter 5

Evaluation

The following evaluation is going to examine the two previously suggested memory tracing techniques. The first is based on dynamic binary translation, adding instrumentation on every write access to allow tracing those. The second utilizes KVM and Intel's EPT, preventing accesses to memory from happening directly, instead invoking the VMM, which can, in turn, trace the write event.

Both will be compared by their adopted implementation, as described in Chapter 4, based on their performance as well as based on the quality of the memory trace one can expect to receive.

The main considerations when choosing a platform for software-based memory tracing is the overall execution time, as time dilation will cause the trace to lose in accuracy. Aside from accuracy, longer execution times are also undesired, should the guest be supposed to be used interactively, e.g., by a customer or just communicating over the network. A high slowdown could possibly lead to timeouts or users getting impatient, thus changing their behavior.

Further, while one of the factors for the quality of the trace is time dilation, it is more important whether, e.g., all the memory accesses occurring are traced and contain all details as desired. In the best case, the resulting trace of write accesses should contain no false-positives and not be missing a single access. Should the quality of the trace be poor, possible analysis of its data could lead to a misleading result.

5.1 Methodology

It quickly turned out the slowdown of the KVM-based tracing approach is significant, making commonly known benchmarks, such as *SPECjbb2005*, impossible to run in the available timeframe.

As the execution does only differ from normal execution for write accesses, as those are the only memory accesses traced, a special benchmark has to be constructed with allows to precisely control how many of the executed instructions are actually writing instructions compared to other instructions. To prevent any side effects from influencing the result, e.g., there should be no interrupts handled. This allows to unbiasedly evaluate the performance of both approaches in this controlled environment, even offering a way to measure the average execution time for an freely chosen percentage of writing instructions.

Thus, a custom benchmark script was written, resembling a minimal multiboot kernel, which once loaded outputted *BOOT* to a serial port and looped for $0x7FFFFFF_{hex}$ cycles as initialization. Then, the script outputs *START* to the serial port and will end the benchmark outputting *STOP*, providing a framework for the aquisition of timing information.

During the actual benchmark, which was written as inline assembly to have full control over the execution, the CPU first loops for an at compile-time configurable amount of times t over a traced write instruction to the same address. After that, a second loop over a dummy read instruction will be executed for r times. The value of r will be computed based on the amount of traced instructions t as well as an upper limit for traced instructions u, by simply subtracting from the upper limit:

$$r = u - t$$

Further, this entire procedure is in itself looped for one hundred million times, to make enough time elapse to get stable measurements of the execution time, yet keep the benchmark quickly executable. A noteworthy benefit of this modeling is that the total amount of instructions executed for each run will stay the same, even should the value of t vary, as long as the upper limit u remains unchanged. This makes it easier to compare different execution times of the benchmark.

Further, this setup allows expressing the percentage of traced instructions executed by the system using a simple formula, providing additional insight into how to interpret the resulting execution times. This formula can be determined based on the knowledge, that the total amount of traced instruction f_t for a given run depends on t, as well as the number of executions of the outer loop, which in this case is a hard-coded one hundred million:

$$f_t = t \cdot 100000000$$

The total amount of untraced instruction f_u depends on all the instructions building both inner loops, the number of times the dummy read was executed as well as the amount of instructions required to build the outer loop.

$$f_t = (\underbrace{5}_{\text{\# ins. outer loop}} + \underbrace{4 * t}_{\text{\# ins. inner write loop}} + \underbrace{5 * r}_{\text{\# ins. inner dummy loop}}) * 100000000$$

Based on this, the following formula for the percentage of traced write instructions was designed:

$$traced_{\%} = \frac{f_t}{f_t + f_u} = \frac{t}{5 + 5 \cdot u}$$

5.2 Test Setup

As it is important to get comparable results for both approaches, the same machine was used to run all performance benchmarks. Figure 5.2 contains all the relevant information about its hardware and the operating system running on it.

Component	Model/Specification
Processor	Intel(R) Xeon(R) CPU E5-2618L v3 @ 2.30GHz
Memory	4x Micron 18ASF1G72PZ-2G1A2 (8GB, 1866MHz)
Operating System	GNU/Linux 4.3.0 (with modified KVM module)

Table 5.1: Setup of the test system

As for the software, the binary of the QSimu variant modified in this thesis, based on QEMU version 2.6.0, was not just used to run the benchmark for the dynamic binary translation tracing approach, but it was also used as the QEMU base in combination with the KVM-based tracing benchmark. QSimu is configured to record the size of the access, the physical address, timing information as well as the data for each access, whereas the KVM-based tracing is limited to only acquiring the physical address and timing information, as the acquisition of all further information would result in further processing overhead and thus would only be of interest, should the KVM-based tracing setup turn out to be faster than the modified QSimu.

All benchmarks were run using the following command-line:

```
sudo qemu-system-x86_64 -display none -k de \
-serial unix:path="{serial}",server \
-monitor unix:path="{monitor}",server -S
```

However depending on the setup, possibly other parameters needed to be added, such as `-enable-kvm` for the KVM-based tracing approach.

5.3 Performance Measurements

The introduced benchmark script was run with the parameter $u = 10000$ and t ranging from 0 to 100, two times for both, KVM-based tracing as well as the modified QSimu, to stabilize the results. Those can be seen in Figure 5.1, showing the averaged execution times of the benchmark for a given percentage of instructions being traced during each run.

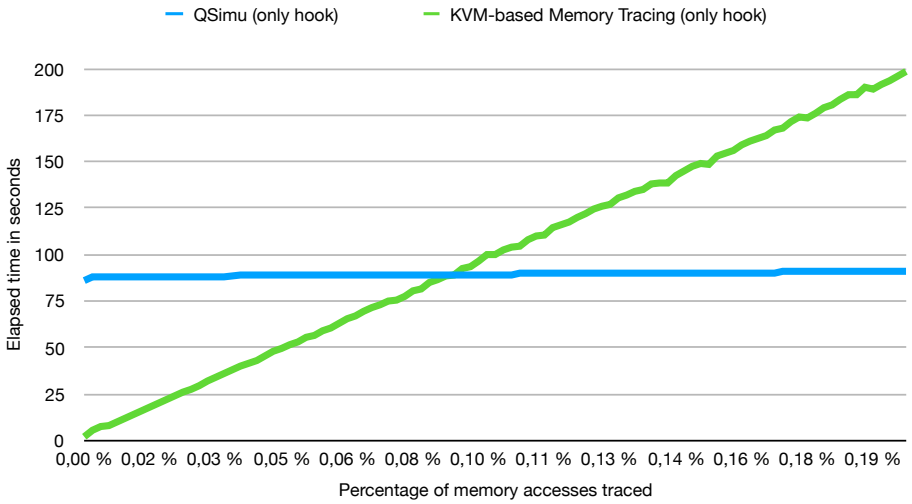

Figure 5.1: The results show the execution times for both approaches to be increasing with the percentage of traced accesses. For workloads with less than 0,09% of write accesses, KVM-based tracing is superior, however, should the proportion be larger, QSimu is faster.

The KVM-based approach was able to execute the benchmark in about 2 seconds, when none of the instructions were traced, compared to QSimu that required 86 seconds. This result is as expected, as the DBT of QSimu comes at a significant cost. Further, as one could also expect, both approaches will take linearly longer for increasing amounts of write accesses, yet the slope for QSimu is nowhere close to the raised slope of the KVM-based tracing. The tipping point, at which QSimu's execution times are the same as for KVM-based tracing is reached at a mere 0,09% of all instructions being traced. From that point on, QSimu is faster than the virtualization-based approach.

According to an analysis from 1998 by J Huang et al., about 9,5% of all executed microoperations on x86 are write operations [17]. Back then there were no vector extensions, so only a few instructions, such as `fsave` have executed more than one write access, thus one can assume about 9,5% of instructions executed to be writing instructions Further, most instructions will execute more than

one micro-operation, resulting in this estimate likely being lower than the actual percentage of write accesses.

Based on this estimation, a second benchmark was executed, three times to stabilize the results, with the goal to simulate an environment in which about 9,5% of all executed instructions are writing instructions. Using $t = 48$ and $u = 100$, this writing rate based on the above formula can be simulated. The results can be seen in Table 5.3, with KVM-based tracing being 1582 times slower than native execution, compared to QSimu just being 38 times slower. Thus, QSimu is executing almost 40 times faster than KVM-based tracing.

Execution Mode	Elapsed time [s]	Slowdown native KVM
native KVM	6	-
QSimu (hook only)	234	39
KVM-based Tracing (hook only)	9501	1582,5

Table 5.2: The execution of the in Section 5.3 described benchmark results in an average slowdown, compared to native execution in KVM, of 39x for QSimu. The KVM-based tracing in contrast is more than 1582x times slower.

5.4 Analysis of the Performance

Those results are not surprising if one considers the significant overhead that comes with this KVM-based tracing approach. First, for each write access, the VM will need to exit and re-enter twice, once to unlock the requested memory page and a second time to enable its protections again. This attached work within the VMM is responsible for further slow down. Additional, the requirement to continuously flush the TLB after every successful write access makes it necessary for the MMU and EPT to repeatedly walk the entire page table. Finally and most importantly, there is a significant, hard to quantify slowdown due to the frequent switching between the VM and VMM, as the processor is unable to fully utilize its superscalar and pipelining capabilities because it cannot start executing instructions of the host or guest, until after their entire CPU context is fully restored.

While the last argument stated above is likely the cause for the largest part of the slowdown encountered, one can still show that even the overhead of the VM entries and exits, as well as the VMM's additional tasks, are sufficient to have QSimu clearly win in terms of performance.

One can calculate an average *cycles-per-instruction (cpi)* value for QSimu, based on the average CPI of x86, which is 2,75 [17], and QEMU's slowdown.

In our case, we are particularly interested in the slowdown including the tracing hook s_{QSimu}, which was measured as being 38 compared to a normally executing KVM, which is close to native execution:

$$CPI_{QSimu} = CPI_{x86} \cdot s_{QSimu} = 104.5cpi$$

On the test machine, according to a VMX benchmarking tool [37], a VM exit will take about 280 cycles, and a VM entry requires 267 cycles, averaged over ten test runs. As we need two full exits followed by reentries to execute a single write operation, at this point ignoring the added overhead of the work required in the VMM, the execution of those entries and exits in total requires 1094 cycles. Using the `rdtsc` instruction, which is a cycle-accurate counter [19], it was further evaluated that the average time spent outside of the VM during memory tracing is about 1212 cycles, averaged over 722 samples. Again, as the VM will exit twice for each write access, the total, additional time spent in the VMM for each memory write is 2424 cycles.

A similar calculation as above for QSimu can now be done for KVM-based tracing, keeping in mind that 9,5% of instruction are write accesses, requiring a significant number of cycles to execute, and other instructions can be executed in their usual time:

$$CPI_{KVM} = 0,095 \cdot (1094 + 2424) + (1 - 0,095) \cdot CPI_{x86} = 336.69875cpi$$

This calculation shows, that even if the KVM-based tracing implementation would not hurt the processor's ability for pipelining and utilizing superscalar technology, due to its significant overhead for each writing instruction alone it would be notably slower than using QSimu for memory tracing.

5.5 Quality of the Trace

Aside from the significantly slower performance for normal use cases, at least the quality of the memory trace should be expected to be superior for virtualization-based tracing, as the underlying hardware is used directly. However, in the limited prototype implemented in this thesis alone many downsides came up, that are going to be addressed in the following:

No direct knowledge of the access size As the current implementation of Intel VMX does not provide a way to access the size of a memory access directly, the VMM has to decode the currently executing instruction in software, which will likely degrade the performance even further. Nevertheless, it might save some

page-faults from happening, as currently, an unaligned write access spanning multiple memory pages will trigger multiple page faults, before the instruction can be executed. Moreover, this will result in multiple memory accesses being recorded, although is was just one. Further, it would allow for easier access to the data written, as in the current implementation the VMM always has to assume the write involves the entire register width, should it wish to perform some optimization based on this, such an reducing the amount of required page faults for unaligned accesses.

Inaccurate for instructions executing multiple writes Although we can force a VM exit on the first write of a guest's instruction, should it be performing multiple write accesses, such as the `xsave` instruction of x86, it is very likely at least some of the following writes will be missed out on [19], as an entire memory page will have to be unlocked. Thus, all the subsequent writes will not be passed through to the VMM to trace, unless they themselves are at least partially on a currently locked page.

Unable to trace all writes for exception handlers While a similar limitation, it is worth pointing out separately, that this tracing approach is unable to trace all write accesses that may happen during the preparation of the stack for an upcoming exception or interrupt handler. While the VMM will be notified on the first write, subsequent writes slip through, should they not be to a different page.

Implementation of accurate read tracing There is a fundamental challenge to overcome if wishes to also trace read accesses using hardware-assisted virtualization and page protections. Considering a reading instruction is next in line to be executed, and the processor still needs to fetch it. Using the proposed single-stepping approach, the instruction can be successfully loaded, but it possibly is already being executed by the time the *MTF exit* occurs, which is the only mechanism allowing the VMM to regain control. This is, as should the location of the read be part of the newly loaded code sequence, it is now able to access the memory, as it is currently unlocked and thus it will be able to perform the read, without the VMM being able to trace it.

5.6 Conclusion

Although the KVM-based approach relies on hardware-assisted virtualization, it is about 40 times slower than using QSimu for memory tracing, most of the slow-

down possibly introduced due to inefficient use of the processors pipelining capabilities and superscalar technology.

Further, as the instruction set does not provide information about the memory access such as the access size, it would be required to decode each writing instruction within KVM. However, as this does not happen in the current implementation, it leaves the memory trace of the current implementation incomplete, yet this could be improved on in the future. Nevertheless, it is not obvious how to prevent other inaccuracies from happening, such as missing out on write accesses for the preparation of stack frames for the interrupt handler or missing write accesses by complex instructions, such as `xsave`. As QSimu is utilizing dynamic binary translation, which allows to simply add instrumentation to trace every memory access, it does not need to deal with suchlike issues.

All in all, the evaluation has shown, that QSimu is better suited for memory tracing than the in this thesis implemented KVM-based memory tracing facility, in terms of both the performance as well as the accuracy of the obtained memory trace.

Chapter 6

Conclusion

Memory traces are an important tool to improve the performance and security of modern software. However, few tools are available to trace the entire system, including the kernel. While hardware-tracers exist, software-based memory tracing is more flexible, inexpensive and thus overall more accessible to a wide range of people compared to hardware tracers, which is why only software tracers were further evaluated during this thesis.

As the inherited slowdown of using a memory tracer based on dynamic binary translation seemed to be avoidable when using a virtualization-based approach, a prototype was designed based on QEMU and KVM using Intel's EPT to control the access to memory. By setting write-protections for all pages, once a write access occurred, it was temporarily granted using single-stepping, then revoked again.

While comparing the prototype of the previously described method to QSimu, an adapted version of QEMU supporting memory tracing, the results show that at this point the virtualization-based approach cannot compete with DBT when it comes to performance, due to the significant overhead for VM exits and entries and the page table manipulations. Aside from the actual cost of a VM exit and entry itself, in addition, it comes with a high penalty, as the processor is unable to fully utilize its pipelining capabilities and superscalar technology. Aside from performance, due to some instructions performing multiple memory accesses, a virtualization-based approach to memory tracing as suggested in this thesis might have inaccuracies. Further, the VMX context does not provide the implementation with all the details one is looking for, resulting in the need to decode the currently executing instruction, which leads to a significant overhead, in contrast as DBT already needs to decode every instruction, the added overhead there is minimal.

Should one perform memory tracing on the 86 platform, implementing such a system based on dynamic binary translation would be a better choice than using hardware-assisted virtualization.

Bibliography

[1] Valgrind. http://valgrind.org/.

[2] Documentation/tcg, Oct 2016. https://wiki.qemu.org/index.php?title=Documentation%2FTCG&oldid=6174.

[3] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *ACM SIGOPS Operating Systems Review*, 40(5):2–13, 2006.

[4] Yungang Bao, Mingyu Chen, Yuan Ruan, Li Liu, Jianping Fan, Qingbo Yuan, Bo Song, and Jianwei Xu. Hmtt: a platform independent full-system memory trace monitoring system. *ACM SIGMETRICS Performance Evaluation Review*, 36(1):229–240, 2008.

[5] Yungang Bao, Jinyong Zhang, Yan Zhu, Dan Tang, Yuan Ruan, Mingyu Chen, and Jianping Fan. Hmtt: A hybrid hardware/software tracing system for bridging memory trace's semantic gap. *arXiv preprint arXiv:1106.2568*, 2011.

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

[7] Fabrice Bellard. Tiny code generator. *qemu git Repository*, (commit b208ac07ea8455878bba250796be38dbe58066db), jul 2017.

[8] Sanjay Bhansali, Wen-Ke Chen, Stuart De Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 154–163. ACM, 2006.

[9] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 15(4):412–447, 1997.

[10] Prashanth P Bungale and Chi-Keung Luk. Pinos: a programmable framework for whole-system dynamic instrumentation. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 137–147. ACM, 2007.

[11] 73 contributors (latest Roman Kagan). The definitive kvm (kernel-based virtual machine) api documentation. *linux git Repository*, (commit d3457c877b14aaee8c52923eedf05a3b78af0476), jul 2017.

[12] Jakob Engblom. Full-system simulation. *Proceedings of the European Summer School on Embedded Systems (ESSES 2003)*, 2003.

[13] Jakob Engblom, Dan Ekblom, and Virtutech Ab. Simics: A commercially proven full-system simulation framework. 2006.

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

[15] K Grimsrud, J Archibald, M Ripley, K Flanagan, and B Nelson. Bach: a hardware monitor for tracing microprocessor-based systems. *Microprocessors and Microsystems*, 17(8):443–459, 1993.

[16] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. Hqemu: a multi-threaded and retargetable dynamic binary translator on multicores. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 104–113. ACM, 2012.

[17] Jer Huang and Tzu-Chin Peng. Analysis of x86 instruction set usage for dos/windows applications and its implication on superscalar design. *IEICE Transactions on Information and Systems*, 85(6):929–939, 2002.

[18] Yongbing Huang, Licheng Chen, Zehan Cui, Yuan Ruan, Yungang Bao, Mingyu Chen, and Ninghui Sun. Hmtt: A hybrid hardware/software tracing system for bridging the dram access trace's semantic gap. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(1):7, 2014.

[19] Intel. Intel 64 and ia-32 architectures software developer's manual, Jul 2017. `https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf`.

[20] M Tim Jones. Virtio: An i/o virtualization framework for linux. *IBM White Paper*, 2010.

[21] Hui Kang and Jennifer L Wong. vcsimx86: a cache simulation framework for x86 virtualization hosts. *Stony Brook University*.

[22] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230, 2007.

[23] Ai Bee Lim and Jack R Johnson. Processor reorder buffer (rob) timeout, Oct 2010. `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/rob-timeout-debug-guide-paper.pdf`.

[24] Imagination Technologies Limited. The mips architecture and virtualization. `http://cdn2.imgtec.com/mips-downloads/m51xx/The-MIPS-architecture-and-virtualization-for-web-download.pdf`.

[25] Linux. ioctl. *Linux manual page*, may 2017.

[26] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

[27] Sally A McKee. Reflections on the memory wall. In *Proceedings of the 1st conference on Computing frontiers*, page 162. ACM, 2004.

[28] Shirley Moore, Felix Wolf, Jack Dongarra, Sameer Shende, Allen Malony, and Bernd Mohr. A scalable approach to mpi application performance analysis. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 12th European PVM/MPI Users' Group Meeting Sorrento, Italy, September 18-21, 2005. Proceedings*, pages 309–316. Springer, 2005.

[29] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[30] David M Nicol. Modeling and simulation in security evaluation. *IEEE security & privacy*, 3(5):71–74, 2005.

[31] Allan Kennedy Porterfield. *Software methods for improvement of cache performance on supercomputer applications*. PhD thesis, Rice University, 1989.

[32] Xen Project. Xen webpage. `https://www.xenproject.org/`.

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

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

[35] Mendel Rosenblum and Tal Garfinkel. Virtual machine monitors: Current technology and future trends. *Computer*, 38(5):39–47, 2005.

[36] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters*, 10(1):16–19, 2011.

[37] Takahiro Shinagawa. The vmx benchmark (vmxbench), oct 2017. `https://github.com/utshina/VMXbench`.

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

[39] Felix Wilhelm. Tracing privileged memory accesses to discover software vulnerabilities. Master thesis, Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, November30 2015.