Karlsruher Institut für Technologie

Assessment of Virtual Machine Working-Sets in SimuBoost

Bachelorarbeit
von

Johannes Werner

an der Fakultät für Informatik

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

Bearbeitungszeit: 13. November 2017 – 12. Maerz 2018

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

Karlsruhe, den 13. März 2018

Deutsche Zusammenfassung

Full-System Simulation ist ein nützliches Werkzeug für z.B. Forschung im Bereich Betriebssysteme oder Malware Analysen. Da für eine Full-System Simulation allerdings Emulation benutzt werden muss, wird die Ausführung sehr langsam. SimuBoost versucht dieses Problem zu beheben, indem es einen hohen Grad an Parallelisierung ermöglicht. Das zu simulierende Szenario wird erst in einer hardware-beschleunigten virtuellen Maschine ausgeführt, in der regelmäßig inkrementelle Abbilder des Zustands der virtuellen Maschine erstellt werden. Diese Abbilder werden dann über ein Netzwerk zu einem Server-Cluster übertragen, in dem dann jedes Intervall einzeln in einer Full-System Simulation emuliert wird. Allerdings kann es bis zu mehreren Minuten dauern, bis ein Abbild komplett in solch einem Simulator geladen wurde. Das liegt zum Teil daran, dass der komplette RAM wiederhergestellt werden muss, bevor die Emulation gestartet werden kann.
In dieser Thesis untersuchen wir, ob die Ladezeit verringert werden kann, indem nur die RAM-Seiten wiederhergestellt werden, die tatsächlich in dem jeweiligen Intervall benötigt werden. Dazu messen wir die sogenannte *Arbeits-Menge* von verschiedenen Ausführungsszenarien, wie z.B. einer Linux-Kernel-Kompilierung. Außerdem überprufen wir, ob ein modifizierter Post-Copy Mechanismus die insgesamt benötigte Simulationszeit noch weiter verringern kann, indem nur das Schreib-Arbeits-Set beim Laden eines Checkpoints wiederhergestellt wird und die restlichen Seiten bei Bedarf nachgeladen werden.

Abstract

Distributing checkpoints over a network and restoring them in an FSS can take up to several minutes. This is partly due to the fact that in SimuBoost's current implementation the whole RAM image has to be loaded into the FSS.
In this thesis, we assess whether restoring only the working-set can lead to a decrease in checkpoint loading times. We do this by using an FSS to measure the working-set of different workloads and different interval lengths. The results show that even though the size of the working-set depends heavily on the workload and the interval length, it is still an only small portion of the whole RAM itself with the biggest measured working-set only being 7.94%. We also assessed an optimization technique similar to post copy. Instead of the whole working-set, only the write working-set is restored initially and the remaining pages have to be loaded on demand.

Contents

Chapter 1

Introduction

Full system simulation (FSS) is an important tool in, e.g. operating system research or malware analysis. But due to the required emulation, FSS comes at a high execution time cost. SimuBoost [22] is a project that aims at reducing the overall needed time to finish the simulation by providing a high degree of parallelization.
The workload is first executed in a hardware-assisted virtual machine where checkpoints are created periodically. These checkpoints contain the whole VM state, including the RAM, CPU and miscellaneous device states. The checkpoints are then distributed over a network to a server cluster where each checkpoint is loaded into an FSS individually and the interval is replayed. However, transferring the checkpoints over a network leads to a completely new problem. Restoring the VM state in the FSS can take up to several minutes, which decreases the overall achievable speedup. This is partly due to the fact that in the current implementation of SimuBoost the whole RAM image has to be restored when a checkpoint is loaded.
In this thesis, we explore the possibility of only restoring the actual working-set. The working-set contains all pages that were accessed during an interval by the hardware-assisted virtual machine and since the FSS replays the interval exactly the same, the working-set is sufficient for a successful simulation.
In order to examine whether it will be beneficial for checkpoint loading times to only restore the working-set, we have to measure the size of the working-set for different workloads. We did this by tracing the working-set in an FSS. But due to the induced slowdown of an FSS, the workload could behave differently than it would in a native execution environment and thus our measurements would not represent a realistic working-set behavior. To circumvent this problem we first executed the workload in a hardware-assisted virtual machine and created checkpoints periodically. We then replayed each interval in an FSS where we traced the working-set.

We assessed the working-set for different workloads and different interval lengths and found that e.g., for a kernel build the size of the working-set was only 7.94% of the whole RAM, which was also the biggest working-set we measured overall. We also assessed another mechanism to reduce checkpoint loading times. It is similar to the post copy mechanism in that only the write working-set is restored initially instead of the whole working-set. The missing pages in the read working-set have to be loaded on demand during the simulation.

In Chapter 2 we give an overview of all relevant technologies, such as virtualization in general, SimuBoost, and QEMU/KVM. Chapter 3 looks at two different approaches to measure the working-set of a workload and why exactly we want to measure it in the first place. In the 4th chapter the implementation and design of both approaches are explained. In Chapter 5 we make a final decision which of both approaches we eventually use for tracing the working-set and present our results. In the 6th and last chapter we conclude the thesis by summarizing our approach,observations, and ensuing results.

Chapter 2

Background

In this chapter, we will explore the concepts relevant to our thesis. First, we will look at virtualization in general and then dive into more specific technologies such as emulation, hardware-assisted virtualization, and full system simulation. After that, we will see how checkpointing works which eventually enables us to understand the concept of SimuBoost. In the end, we will look at the specific applications that SimuBoost builds upon, which are QEMU and KVM.

2.1 Virtualization

Virtualization is a technology that plays an important role in modern IT systems. It enables a physical host to run a guest system inside a virtual environment and thus separating the state of both machines without the need for additional hardware. This is especially useful in cloud computing where it is commonplace that multiple guest systems share the same physical host which allowing for dynamically shared resources among users. Otherwise, a cloud computing service would have to provide each customer with its own physical machine, which can be highly inefficient in terms of system utilization [8]. Even in the consumer market virtual machines have become an integral part. Java is a platform independent language which uses the Java virtual machine (JVM) to abstract the running software from the host system. Microsoft uses the same approach for its .NET framework which uses the common language runtime (CLR) as the virtual environment.
In order to virtualize a system, it has to be provided with a virtual environment that has the same interface as a physical machine. To achieve this, a virtual machine monitor (VMM) is needed. The fundamental purpose of a VMM is to control the execution of the virtual machine (VM). It keeps track of the VM's state and emulates the interaction with virtual or physical devices. However, the most important task of a VMM is to make sure that the guest stays within the boundaries of the

virtual environment and cannot alter the state of the host in an undesired way. This is done by trapping instructions that would affect the host system before they are executed so they can be emulated by the VMM.

2.1.1 Emulation

Virtualization creates the possibility to execute guest code that does not match the host instruction set. This enables a user to emulate arbitrary guest CPU architectures independently of the physical hardware. To achieve this, the VMM has to translate the guest code into instructions that can be executed by the host CPU while also making sure that the guest cannot escape the virtual machine.

Intepretation is the most straightforward approach to translate guest code. The VMM fetches the guest instructions one-by-one to decode them individually and creates instructions for the host CPU that match the desired functionality. These instructions modify virtual CPU, memory, and device states which ultimately emulates the behavior of the guest code. However, emulation comes at a very high execution cost, since a single guest instruction can be translated into tens of host instructions resulting in a significant slowdown [25].

Dynamic Binary Translation is a more efficient solution commonly used by popular emulators such as QEMU [5]. Rather than translating each guest instruction individually, the guest code is split into blocks that are translated as a whole. This leads to a higher initial cost but translated blocks are buffered and can be used multiple times which is especially beneficial for loops. The repeated execution of the buffered code blocks amortizes the high translation overhead and leads to a huge performance gain compared to simple interpretation [25].

2.1.2 Hardware-Assisted Virtualization

Hardware-assisted virtualization addresses the huge slowdown of emulation by executing guest code directly on the host CPU without translating or modifying any instructions. This comes with the limitation that both, host and guest, must use the same instruction set. Even though this leads to a huge gain in performance and makes near-native execution speed possible, the x86 platform is hard to virtualize in the first place. Some instructions behave differently when they are executed in user mode than in kernel mode [18]. In order to be able to emulate this peculiarity, the VM has to fall back into VMM when such an instruction occurs. But in user mode, those instructions do not trap and thus makes it difficult to emulate them correctly. To address this and other problems, Intel and AMD introduced processor extensions for hardware-assisted virtualization, namely Intel-

VT [7] and AMD-V [6]. Both approaches are very similar, yet not compatible with each other. They introduce a new operation mode next to kernel and user mode which allows to specifically request that the CPU traps on certain instructions. This mode is called *guest mode* and every time before guest code is executed the CPU switches into this mode [18]. Another addition that was made is *exit reason reporting*. When the CPU switches from guest mode back to the host mode, the CPU reports the reason for this exit and the VMM can react properly [18].

2.1.3 Full System Simulation

Full system simulation (FSS) builds upon emulation and allows to easily implement hooks to trace the behavior of a workload. This makes FSS a useful tool in research and development. It enables developers to implement and test software for hardware architectures that were not published, yet. It also allows developers to prototype and test new hardware architectures before they are actually built. In research, FSS is a useful tool to find security issues by analyzing software and, e.g., searching for possible race conditions. FSS also provides a sandboxed environment where malware can be executed and analyzed while almost eliminating the risk of infecting the host system.
However, simulating a whole system also comes at a cost. With a slowdown factor of 30 to 800 being common [22], an FSS is significantly slower than native execution. This makes FSS unfeasible for long-running workloads and also nearly impossible for a simulation to interact with a nonsimulated instance such as a user or another system. Simulating network interactions becomes especially difficult since communication channels will timeout before the FSS can respond.
Naturally, different use cases for an FSS require a different level of details at which the simulation operates. A trade-off has to be made between functional and timing accuracy. Whereas functional accuracy only focuses on the correctness of the executed instructions, timing accuracy takes specific timing into account, e.g., how many cycles an instruction needs to be finished or how long a cache or RAM access takes. QEMU is an FSS that focuses on functional accuracy [12], which is a so-called functional full system simulator, and thus achieves a relatively fast simulation speed. Simics, on the other hand, is an FSS that takes timing parameters into account. However, implementing a correct timing accurate FSS is nearly impossible, since additionally to the added implementation complexity to account for the timing, seldom all details of a hardware architecture are published. This requires reverse engineering the desired bits or to simply resort to an approximation, which is what Simics does [20].

2.1.4 Checkpointing

Another advantage of virtual machines is the ability to create checkpoints. A checkpoint saves the complete state of a VM at a specific moment in execution, containing the state of RAM, disk, CPU, and other devices [26]. However, current RAM and disk capacities ranging from several Gigabytes to Terabytes makes efficient checkpointing a nontrivial task. Especially in an environment where high frequency checkpointing is required with an interval length of a few seconds or lower.
The basic approach to creating a checkpoint is stopping the VM, creating the checkpoint, and then resuming the VM. To be able to assess different techniques for creating a checkpoint, several metrics have to be taken into account:

Downtime describes the amount of time that the VM is stopped. During this time the VM is unavailable and the VMM creates the checkpoint. The main factors that influence the length of the downtime are the amount of data that has to be saved and the preparation of the next interval. Especially for high frequency checkpointing a short downtime is crucial since the downtime adds to the simulation time of each interval.

Data amount is the total amount of data produced by a checkpoint. Most data is produced by saving the RAM and disk but also includes minor parts such as the CPU state. A low data amount is desirable since huge checkpoints are slow to transfer, e.g., over a network.

Execution slowdown refers to the slowdown factor of the workload due to the checkpointing overhead. This overhead can originate, e.g., from additional page faults or VM exits. We differentiate between downtime and execution slowdown since the length of the downtime generally has no effect on the execution of the workload itself except on the timing. Whereas checkpointing overhead during execution can slow down the workload.

Stop-and-Copy

Stop-and-Copy is the most basic approach to create a checkpoint. The VM is stopped and the VMM saves RAM, disk, CPU, and other device states. This induces a long downtime, however, execution of the workload itself is not slowed down since all the work is done at the end of an interval.

Incremental Checkpointing

Incremental checkpointing is a technique that optimizes the amount of data that has to be saved and consequently also has a positive effect on the downtime. A VM most likely does not modify the whole RAM in a single interval. It rather modifies only a small subset of pages. Baudis measured that for a kernel build only 5-10% of the RAM belong to the actual working-set [10]. This renders creating a complete RAM image for each checkpoint highly inefficient. So instead, only the pages that were modified in the current interval have to be saved and all unmodified pages can be taken from the previous checkpoint. This way a complete RAM image can still be reconstructed. However, a dependency between each checkpoint is introduced which could be undesirable in some use cases. Boehr measured a downtime of 84ms for a kernel build with incremental checkpointing in combination with stop-and-copy [13].

Copy-on-Write

Copy-on-Write (COW) is an optimization to minimize downtime. Rather than creating the RAM image while the VM is suspended, the pages are copied concurrently while execution has been resumed again [26]. During downtime, only the absolute minimum is done like saving the CPU state. However, it is possible that VM modifies pages that have to be copied, but were not copied yet which would lead to a wrong RAM image in the checkpoint that is created. Consequently, the VMM somehow has to detect when such an event occurs which is done by setting the write protections for all pages during downtime. When the VM now tries to modify a page that has not been saved yet, a page fault is triggered and the VMM can save the affected page before the VM modifies it.
Incremental checkpointing and COW can be combined to achieve an optimal downtime. Instead of creating a complete RAM image with COW, we only copy the pages that were modified during the interval. For this approach, Boehr measured a downtime of only 26ms for a kernel build [13].

2.2 SimuBoost

As already stated in Section 2.1.3, full system simulation comes at a very high cost. Simulation time can be up to 33 times slower than native execution on hardware. With memory tracing enabled, the slow down becomes even higher with an execution time increase to the factor 165 [22]. To reduce the overall execution time the simulation has to be parallelized. There are several different approaches to do this. One approach is to simulate each virtual CPU (vCPU) on its own hardware CPU core [16]. But the achievable degree of parallelization is limited by the

amount of vCPUs and, therefore, not feasible for larger scale simulations.
SimuBoost [22] is a project that aims at reducing the time cost of full system simulation by using an approach that allows for a very high degree of parallelization. The core idea is to split the workload into intervals and simulate each interval in parallel. But the problem is that an interval depends on the end state of the previous interval and the simulator would have to wait for the previous interval to finish before it can start simulating the current one. This would make parallelization impossible. To solve this problem, the workload is first executed in a hardware-assisted virtual machine where it is split into intervals. At the start of each interval, a checkpoint is created which saves the current state of the VM. Further, all non-deterministic events (e.g., interrupts) are recorded during execution. The checkpoints make each interval independent from one another since the state of the VM at the start of each interval is now known. Figure 2.1 illustrates the process of splitting the workload into intervals.

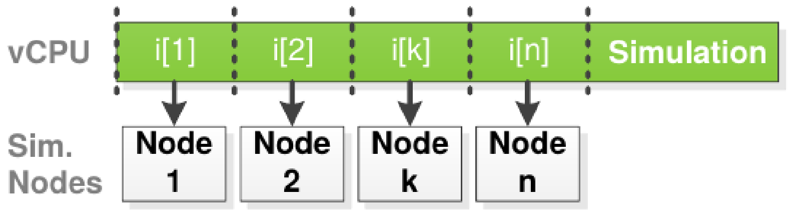

Figure 2.1: The workload is split into intervals which are then distributed to different server nodes [22]

When the execution of an interval has finished, the checkpoint is distributed over a network to a server cluster where the FSS will take place. To ensure that the simulation does not diverge from the original execution, deterministic replay is used which injects all previously recorded non-deterministic events at the correct moment.

2.3 QEMU and KVM

QEMU [4] is a popular and open source full system simulator. It supports many different host and guest platforms, such as ARM, i386/x86-64, PowerPC and MIPS. In total 21 platforms are available. A guest system does not have to be modified in order to run in QEMU. Devices such as display or network devices are emulated as well, thus the guest is provided with a complete virtual system environment. [5].

For full system simulation QEMU uses binary translation to achieve a good performance. At the core of QEMU's system simulation is the tiny code generator (TCG) [11]. The guest code is first translated into an intermediate instruction set. TCG takes this intermediate code as input and performs some basic optimizations on it. Eventually, the intermediate code is translated into target platform code, with the target being the host platform QEMU runs on. The translated code can then be natively executed by the host CPU. However, a single guest instruction can be translated into many host instructions, leading to a big simulation overhead. But this is necessary since it has to be guaranteed that the guest does not modify the host system in an undesired way. Further, analysis functionality is added into the intermediate code, such as tracing hooks. The reason that the guest code is not directly translated into host code, but rather into an intermediate format first, is that an intermediate format allows for a much cleaner code base. It would simply be impossible for a newly introduced platform in QEMU to implement translations for all possible combinations of guest and host platforms. With an intermediate format, only two translations have to be implemented. The first is the translation from the guest instruction set to the intermediate instruction set and the second one is the translation from the intermediate code to host code.
The second operation mode that QEMU supports is hardware-assisted virtualization. This is done using KVM as a hypervisor [1]. KVM is a Linux kernel module available for the virtualization extensions Intel VT and AMD-V. Even though both extensions are incompatible with each other, at the core they are very similar [18]. One big advantage of hardware-assisted virtualization is that it is possible to reach near-native execution speed. In order to achieve this, it is necessary to do address translation from guest virtual addresses (GVA) to host physical addresses (HPA) in hardware. For a given GVA the processor first walks the page tables managed by the guest OS to get the GPA and then uses second level address translation (SLAT) to map the GPA to HPA. Intel introduced in 2007 a technology named extended page table (EPT) as the SLAT extension for the company's platforms. It basically works like a normal page table, except that it maps GPA to HPA and uses different flags in an EPT entry that are special to virtualization. AMD's counterpart was introduced one year earlier in 2007 and is named nested page tables (NPT).

Chapter 3

Analysis

Functional full system simulation is a useful tool in many scenarios. In contrast to pure emulation, it provides an easy way to inspect the virtual system and its run-time behavior on the instruction level. A workload is executed in the full system simulator (FSS) which uses binary translation for emulation and places hooks in the execution to trace the behavior. The workload has to be emulated since tracing on the instruction level is not possible in a hardware-assisted VM. However, this comes at a cost. Rittinghaus et al. measured a slowdown of 31x-810x on average for an FSS compared to a hardware-assisted virtualization [22]. Thus a workload simulated in an FSS can possibly behave differently than it would normally. Furthermore, current implementations of simulators are very limited in the degree of parallelization that can be achieved during simulation which makes an FSS unfeasible for longer running workloads. SimuBoost attempts to solve both problems by using a combination of hardware-assisted virtualization and FSS [22]. The workload is first executed in a hardware-assisted VM where periodically checkpoints are created which save the VM's state. The checkpoints are then distributed over a network to a server cluster where each interval is simulated concurrently. Through deterministic replay, which injects non-deterministic events such as interrupts at the same time they occurred in the original execution, it can be ensured that the simulation behaves exactly like it should. But distributing checkpoints over a network leads to a completely new problem. Push measured that pushing checkpoints to a server cluster directly leads to a checkpoint loading time exceeding several minutes [21]. The checkpoint loading time adds to the overall time that is needed to finish the simulation of an interval and thus the achievable speedup is reduced. Push discussed in his thesis two approaches to decrease the loading time [21]. The first approach is a distributed file system. Each server can see the same data but in order to load a checkpoint, the data has possibly be requested from another server. This method turned out to be not very efficient and thus Push focused on the second approach: IP multicasting. The producer (i.e., the

hardware-assisted VM) pushes the checkpoints to each server at once where they are stored. To restore a checkpoint, the server first looks in it's RAM if the checkpoint is still buffered and if it is not, the checkpoint has to be loaded from disk. This makes the disk, next to the network, another possible bottleneck. But Push could still reduce the checkpoint loading times with IP multicasting by a factor of 43 compared to a distributed file system [21].
Another way to reduce loading times is to reduce the data that has to be loaded. A checkpoint consists of the RAM, disk, and device states whereas disk and RAM amount for the most data. In this thesis, we will analyze whether the RAM data can be reduced.

3.1 Optimizing RAM Loading Time

Currently, the whole RAM image is restored when a checkpoint is loaded in the FSS. This means that several gigabytes have to be loaded and the more RAM the hardware-assisted virtualization had, the more RAM has to be loaded. Since we use deterministic replay, the FSS accesses exactly the same pages as the hardware-assisted virtualization which leads to the obvious optimization, that we could only restore the pages that are actually used in an interval.
The pages that are accessed in an interval are called the *working-set*. The working-set can further be split into the *read working-set* and the *write working-set* which contain the pages that were read or modified, respectively. To be able to assess whether restoring only the working-set instead of the whole RAM can lead to a decrease in loading times, we need to measure how big the working-set of an interval actually is.
Baudis measured that for a kernel build only 5-10% of the whole RAM belong to the write working-set [10][1]. Clark et al. measured that around 0-40% belong to the write working-set, depending on the benchmark [14][2]. Even though only the write working-set was measured, these results let already suggest that the whole working-set is also only a subset of the whole RAM. But to verify this guess, we need to measure the working-set for different workloads ourselves.
Another factor that plays into the size of the working-set is the interval length. The longer the interval, the more data can be accessed. However, this does not necessarily mean that the working-set grows linear to the interval length since for a longer interval the VM possibly accesses the same pages more often as well. Denning proposed a model to predict the ratio between the interval length and the working-set size [15].

[1]2000ms interval, 2GiB RAM
[2]Benchmarks: vpr, gcc, mcf, crafty, parser, eon, perlbmk, gap, vortex, bzip2, twolf

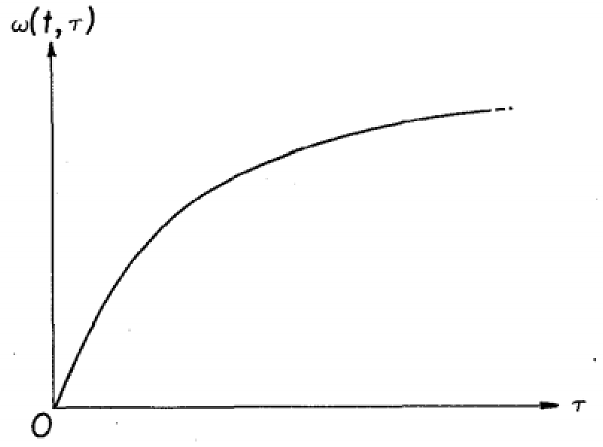

Figure 3.1: The longer the interval length the slower grows the working-set [15]

Figure 3.1 shows how the working set grows relatively to time. The x-axis is the passed time in seconds and the y-axis represents the working-set size in the time interval $(t{-}\gamma, t)$ with t being a time stamp and γ the inspected interval length. The graph suggests that the working-set grows slower the longer the interval is and is slowly approaching the working-set of the complete workload. To verify this prediction of the working-set behavior and how it holds for different workloads we have to measure the working-set for different workloads and different interval lengths.

3.1.1 Optimization Techniques

As already discussed earlier, we want to analyze how we can reduce checkpoint loading times. We focus on reducing the data that amounts for restoring the RAM in the FSS since we showed that RAM offers good optimzation possibilities by only loading the working-set. In the following we present two different approaches to achieve this.

Loading the Whole Working-Set

The intuitive approach is to measure the working-set during execution in the hardware-assisted virtualization and then only send the working-set to the servers. Depending on the size of the working-set this could already cut down loading times significantly. However, the feasibility of this approach depends on the slow-down induced by the overhead in the hardware-assisted virtual machine due to measuring the working-set.

Post Copy

Sending less than the whole working-set to the servers leads to the problem that only parts of the needed pages can be restored by the FSS. This means that some memory accesses cannot be satisfied during simulation and have to be resolved in another way.
An approach that sends less than the whole working-set, but addresses these difficulties, is a slightly modified post copy mechanism. Only the write working-set is transferred over the network to the servers and thus only the write working-set can be restored in the FSS. However, some read accesses can still be satisfied since pages that were modified can also be in the read working-set. Only read accesses on pages that are exclusively in the read working-set will pose a problem. Such pages are loaded on demand. They are read protected when the simulation starts which means that a page fault will be triggered when the VM tries to access them. The VMM then requests this page from the producer (i.e., the hardware-assisted VM) and loads them through the network. The rather big downside of this approach is that a read access possibly requires a network access to download the missing page which could take relatively long, also taking into consideration that the network utilization will already be quite high due to the simultaneous checkpoint distribution. The upside, on the other hand, is that a checkpoint can be faster transferred to the servers since only the write working-set has to be sent and the simulation can thus be started earlier. This in return means that the simulation can finish earlier as well. The deciding factor whether this approach overall speeds up the simulation is how many pages have to be loaded on demand and how long it takes to load a single page. The time it takes the VM to load these pages should not exceed the time it saves by loading only the write working-set initially since then we would end up with an overall worse simulation time. So additionally to measuring the size of the whole working-set we will measure a distinct read and write working-set and identify the number of pages that would have to be loaded on demand.
Another advantage of the modified post copy approach, that we did not mention yet, is that we could use the already existing dirty logging mechanism in KVM to measure the write working-set during execution in the hardware-assisted VM. The same mechanism is also used for incremental checkpointing in SimuBoost. This means we can already be sure that it would not induce any new overhead to the workload execution if we implemented this approach.

3.2 Measuring the Working-Set

In the last section, we discussed two different approaches to optimize the checkpoint loading times and how they would affect the execution speed in the hardware-assisted virtual machine. In this section, however, we will look at how we can measure the working-set of a workload in the first place to make an actual decision on what optimization approach fits best. Since we only want to measure the working-set, it is not crucial that we do get our measurements in the fastest way possible. However, the measuring approach should still be sufficiently fast since we want to measure different workloads with different interval lengths and, therefore, want to be able to complete all measurement in our limited time frame. Further, the behavior of the workload should not be altered significantly by the induced slowdown through the logging overhead. Another condition that a measuring approach should meet is that it should be capable of measuring a distinct read and write working-set.

3.2.1 Traversing the Extended Page Table

The extended page table (EPT) is a data structure that maps guest physical addresses (GPA) to host physical addresses (HPA). Additionally, each mapping contains several flags which signal the state of the corresponding page. The flags we are interested in are the *dirty flag* and the *accessed flag*. These flags are set by the CPU when a page is accessed or modified respectively. A page qualifies as accessed when it was either read, modified, or both.
At the start of an interval, both flags are set to zero for each EPT entry and the simulation is resumed. When the interval has finished, the VM is stopped and during this downtime, the EPT is traversed. For each mapping, the dirty and accessed flags are evaluated and thus the whole and write working-set can be determined. However, with this approach, it is not possible to create a distinct read working-set. Only the exclusive read working-set can be assessed by scanning the EPT for entries which have the accessed flag but not the modified flag set which means that they were only read but never modified. Distinguishing between a page that was modified and read or only modified is not possible which does not fit our requirements for a measurement approach and thus we will not use it in our further research.
However, traversing the EPT is a good method to determine the whole and write working-set during the workload execution in the hardware-assisted virtual machine, as it is needed for both optimization techniques discussed in Section 3.1.1. Traversing the EPT only adds to the downtime when the checkpoint is created and has no effect on the execution of the workload itself other than the timing.

3.2.2 Page Faulting

Just like the approach in Section 3.2.1, this approach also makes use of the EPT. But instead of scanning it at the end of an interval, we set page protections for all pages at the beginning. The EPT allows managing page protections for each page individually. It is possible to revoke read, write, and execute rights, which leads to a page fault when the VM tries to access a protected page.
At the start of an interval, all access rights are removed from each page. When the VM then accesses a page, a page fault is triggered and control is given to the VMM where the access is logged. For a read or execute access the page is added to the read working-set and for a write access, the page is added to the write working-set. In the last step, access rights for future accesses of the same type are given. This means that for a read or execute access both read and execute protections are removed and for a write access only write protections are removed. We can do this since we only need to log each page access once per interval and thus can speed up the execution of the workload by generating as little page faults as possible. Finally, control is returned to the VM and the simulation is resumed.
A special case we have to consider is that we cannot allow write access on a page that is still read protected. This would be an illegal EPT state and would cause the VM to exit [17]. In such a case we have to circumvent this limitation by removing all page protections, execute only the instruction that caused this page fault, and set all page protections again afterwards. This way we are able to log possible read or execute accesses in the future. The downside, however, is that we will need to single-step each write access as long as no read access occurred on the same page.
Using page faults to measure the working-sets is a suitable approach for us since it is capable of creating a distinct read and write working-set, however, Schmidt used a similar approach to trace memory accesses and measured a slowdown factor of up to 1583 [23]. He set all page protections for all pages at the beginning of an interval and thus all memory accesses triggered a page fault. But instead of selectively removing the page protections bit-by-bit, as we do, he single-steps each instruction which allows him to trace multiple memory accesses on the same page.
If we experience the same or a similar slowdown like Schmidt did, it would take an enormous amount of time to measure the working-set and the behavior of the workload could be altered significantly as well. Either of both downsides would make page faulting an unsuitable approach for us. But we do not single-step each instruction and we remove protections as soon as possible which could make our approach sufficiently fast.

For what we have to expect as a minimum slowdown, we can look at the overhead that is induced by KVMs already existing dirty logging mechanism. Schoetterl-Glausch [24] measured the additional overhead for different interval lengths. For a one second interval he measured an overhead of 3% and in total the overhead ranged from 27.5% for 50ms intervals to around 1% for 5000ms intervals.

3.2.3 Tracing in an FSS

We mentioned in Section 3.2.2 that using page faults to measure the working-set could turn out to be too slow. If that happens we will use a fallback approach. The workload is simulated in an FSS with activated memory tracing and from that trace we then can calculate the working-set.
The problem with an FSS is that the slowdown on the workload is significant, ranging from the factor 31 up to 810 [22]. But we already identified a non-significant slowdown on the workload execution as an important property of a measuring approach. To solve this problem we basically use the same method as SimuBoost uses to parallelize a simulation. We first execute the workload in a hardware-assisted virtual machine where checkpoints are periodically created. We then simulate each interval in an FSS with enabled memory tracing and due to deterministic replay, the simulation does not diverge from the original workload. From the resulting memory trace, we simply have to create the final read and write working-set. However, we still get a significant slowdown in the FSS, but it does not matter since the workload is only replayed and behaves like it did in the hardware-assisted virtual machine.
An optimization we can implement further is reducing the number of memory accesses that have to be traced. Creating a full memory trace can be very expensive since each memory access has to be logged. But we are not interested in all memory accesses, we rather only need the first access on a page in an interval. This leads to the possible optimization that multiple accesses of the same type on the same page count just as one. Overall fewer memory accesses have to be logged which, on the one hand, can speed up the simulation itself and, on the other hand, will make calculating the working-set from the memory trace much faster.
The FSS we will use to create the memory trace is QSimu, which is part of the SimuBoost project. This is a very obvious decision since we already mentioned that our measuring approach is very similar to the approach that SimuBoost implements. SimuBoost supports periodic checkpointing, deterministic replay, and memory tracing, which is exactly what we need. We only have to implement the trace optimization ourselves.

3.3 Conclusion

In this chapter, we identified the RAM as a good optimization possibility to reduce checkpoint loading times. Existing work has shown that only a fraction of the whole RAM actually belongs to the write working-set and thus renders restoring the whole RAM image in the FSS as unnecessary and inefficient. The first optimization technique we presented is that only the actual working-set should be restored, and the second technique is a modified post copy mechanism. Only the write working-set is restored in the FSS and the exclusive read working-set is loaded on demand. To make an actual decision which of these optimizations is the better we have to measure the working-sets and how the read and write working-set overlap. To do this we presented three different methods. The first is traversing the EPT during downtime and checking the accessed and dirty flags. But we ruled this approach out since it is not capable of creating a distinct read and write working-set. The second technique we discussed is setting page protections in the EPT and thus triggering page faults when the VM tries to access memory, which enables us to log them. With this approach we had concerns regarding the slowdown and, therefore, introduced a third method as our fallback. This method uses SimuBoost to execute the workload in a hardware-assisted VM and then trace each interval in an FSS.

Chapter 4

Design and Implementation

In Chapter 3 we discussed three different methods to measure the working-set of a workload in an interval. The first method we presented was traversing the extended page table (EPT). But we came to the conclusion that this method is not suitable for us since it is not capable of creating a distinct read and write working-set and we will not focus on it further. The second method was logging with page faults which makes use of the page protections in the EPT and thus creating page faults which allow us to log the memory accesses. The last method was tracing with a full system simulator (FSS) and is our backup method if logging with page faults turns out to be too slow.
In this chapter, we will look at how we implement logging with page faults and tracing in an FSS.

4.1 Logging with Page Faults

Our implementation for logging with page faults consists of a user space part in QEMU and a kernel space part in KVM. The reason we use QEMU and KVM is that they are a popular and open source virtualization software. But instead of using the official versions we use a forked QEMU and KVM that are part of the SimuBoost project. The reason we chose SimuBoost's QEMU and KVM is that they already implement functionality for periodical interruption of the VM which we need to be able to measure the working-set of each interval individually.
As already discussed in Chapter 2.3, the EPT contains the mappings from guest physical addresses (GPA) to host physical addresses (HPA). If a page fault is triggered due to the EPT (e.g., no mapping present) control is given to the virtual machine monitor (VMM) where the page fault is handled. This process is completely transparent to the guest. This means that the guest does not notice whether an EPT page fault was triggered or not. It makes the EPT a perfect fit for our

logging since this enables us to log the working-set without having an impact on the execution of the workload besides the time penalty. We can modify the read-/write/execute bits in the EPT to get notified through a page fault if the guest tries to access its physical memory. And since the EPT is managed in KVM the main portion of our implementation will be done there. QEMU is only responsible for activating and deactivating the logging as well as retrieving the logged data from KVM and save it to disk.

4.1.1 KVMs Dirty Logging Mechanism

KVM already implements its own dirty logging mechanism which is used for VM migration. SimuBoost uses the same mechanism to implement incremental checkpointing. To determine the pages that have to be saved for an incremental checkpoint, dirty logging is activated at the start of an interval and at the end, the created dirty bitmap is retrieved from KVM and evaluated. This sounds promising as we could use the dirty logging mechanism as well, and would only have to implement read logging ourselves.
The EPT does not contain mappings for the whole guest RAM from the beginning. The entries are rather created as they are needed. This means that a page fault can be triggered for a memory access that is perfectly valid but that the EPT does not contain a valid mapping, yet. To handle these kinds of page faults the EPT entry is created and control is returned to the guest. If dirty logging is enabled the corresponding page is additionally logged as dirty but independently of whether the faulting access was a read or a write access. This makes sense for VM migration since a newly accessed page has to be copied to the new VM destination as well as actually modified pages. Even for incremental checkpointing this is the desired behavior. For our purpose, this would be sufficient as well since checkpoint loading time optimizations could be built around the fact that not the actual write working-set is measured. But we want to assess the behavior of the VM as precisely as possible and consequently have to implement our own read and write logging mechanism.

4.1.2 Setting Up KVM

We need to be able to save whether a page was read or modified in an interval. This is done by creating two separate bitmaps in kernel space. The advantage of a bitmap is that it is very space-saving since we only need one bit per page. If a bit in the bitmap is set it means that the page was read/modified and a zero means that the page was not touched in an interval.
We put our bitmaps into the same `kvm_memory_slot` structure as the dirty bitmap of KVM. A memory slot is a chunk of contiguous memory that is mapped

from GPA to host virtual addresses (HVA). The guest RAM is split into several slots which means that for each slot distinct read and write bitmaps have to be created. Consequently, multiple read and write bitmaps exist simultaneously and, to find the correct bitmap to mark a page in, we have to find the memory slot that maps the given GPA to HVA by searching through all slots.

4.1.3 Handling Page Faults

At the start of an interval, if read and write logging is enabled, access rights of all pages are removed in the EPT. This means that the read access bit (Bit 0), the write access bit (Bit 1), and the execute bit (Bit 2) are set to zero. This leads to a page fault every time the guest tries to access its own physical memory [17]. When a page fault occurs, control is given to the page fault handler in the VMM where the memory access can be logged. When the handler returns and control is given back to the guest, its virtual CPU (vCPU) tries to execute the instruction that caused the page fault again. For this reason, it is not enough to only log the memory access but we also have to give the corresponding access rights to enable the vCPU to execute the faulting instruction. This also ensures that we only fault for each read or write access on a page once which makes the whole execution faster. To give the needed access rights we only have to set the corresponding bits in the EPT.

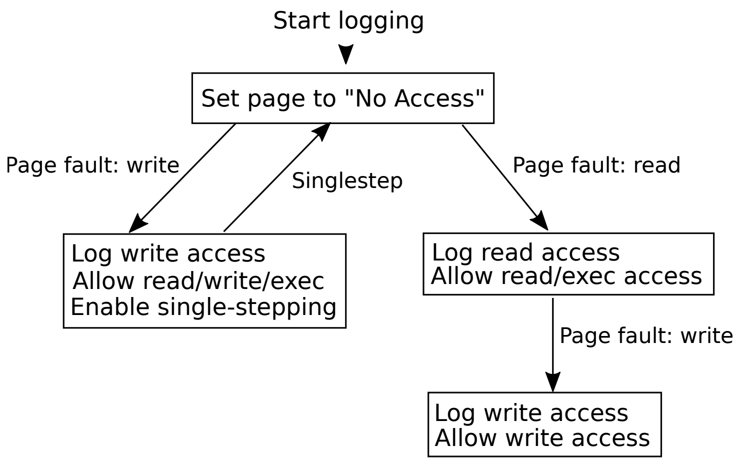

Figure 4.1: After setting a page to "No Access" initially a page access triggers a page fault which allows logging the access

Figure 4.1 shows how a page fault is handled after all access rights of a page were removed initially. If the vCPU tries to read from a page a page fault is

triggered. We log the read access by setting the corresponding bit in the read bitmap and then give read and execute access rights. The reason we also make use of the execute bit in the EPT is that we also want to log instruction fetches as read accesses. After this step no read or execute access on this page will trigger a page fault again. But for a write access, a page fault is still triggered. If this happens we log the write access as usual in the write bitmap and remove the write protection in the EPT. Now the vCPU can read, write, and execute this page without faulting again.
A problem arises when the CPU tries to modify a page that still has no access rights. The page fault is triggered as usual and we log the write access. But now we cannot just give write access rights because it is not possible for a page to have write access rights but no read access rights. This would be an illegal EPT state [17] and will trigger an EPT misconfiguration which cannot possibly be resolved by the VMM and will exit the VM. For this reason, we have to give this page read/write/execute rights and save the page's number. Then we have to enable single-stepping to ensure that the vCPU only executes the instruction that caused the write page fault. To enable single-stepping we have to set the *monitor trap flag* (MTF). This flag ensures that the virtual machine only executes one instruction and then falls back into the VMM. After the instruction was executed, we remove all access rights from the page that we saved previously. Then we disable the MTF so that the VM no longer falls back into the VMM after every instruction and finally return control back to the VM. This way it is possible to still log future read accesses on this page.

Write Access over Page Boundaries

The x86 and x86_64 architecture allow unaligned memory accesses. This means that a memory access does not have to start at an address that is evenly divisible by specific byte amount that can alter between architectures [19]. Such an unaligned access can go over page boundaries and consequently two pages have to be touched which can lead to two page faults caused by the same instruction.
For a read access, this is not a problem since we do not have to preserve any state after a page fault occurred and therefore multiple page faults triggered through the same instruction are still independent of one another. But for a write access, we have to save the page numbers of all the pages that are touched by an instruction so that we can remove the access rights after the instruction was executed. In Section 4.1.3 we already explained that saving the page number is possible as long as only one page fault is triggered per instruction.
Figure 4.2 shows how a write access over the page boundaries from page A to B is handled. When such an access occurs the vCPU first faults on page A. We now take the usual steps, including saving the page number and return control back to

the guest. But the vCPU still cannot execute the instruction because on page B write access is still not allowed. So the vCPU faults on page B again. We handle the page fault as usual, but instead of overwriting the page number we saved previously for page A, we save both the page number of page A and B. Control is returned back to the guest and the vCPU executes the instruction. Due to single-stepping, we fall back into the VMM, disable single-stepping, and since we saved the page numbers of the accessed pages we can remove access rights from page A and B.

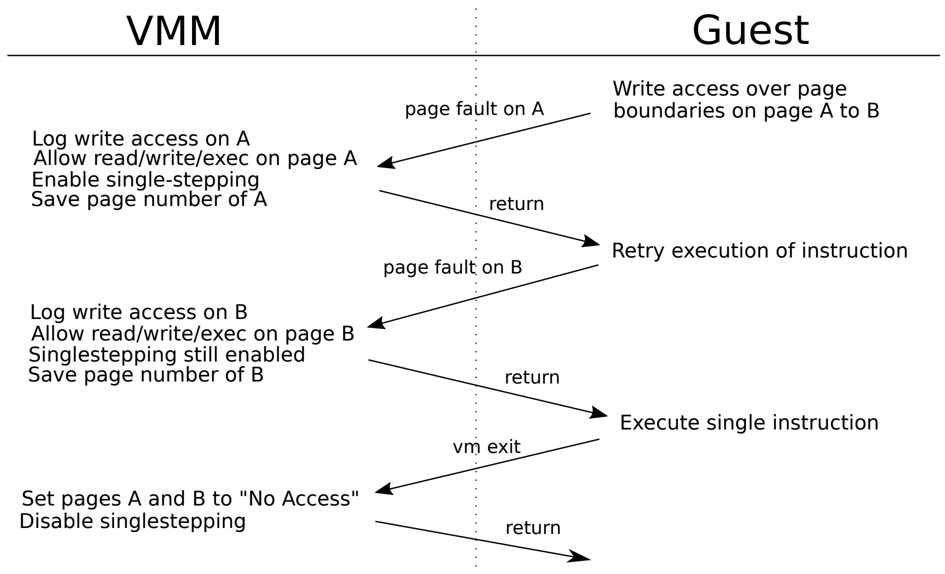
VMM
Guest
Write access over page boundaries on page A to B
page fault on A
Log write access on A
Allow read/write/exec on page A
Enable single-stepping
Save page number of A
return
Retry execution of instruction
page fault on B
Log write access on B
Allow read/write/exec on page B
Singlestepping still enabled
Save page number of B
return
Execute single instruction
vm exit
Set pages A and B to "No Access"
Disable singlestepping
return

Figure 4.2: When a write access over page boundaries occurs the vCPU faults on both pages before the instruction can be executed

Handling Interrupts during Single-Stepping

Interrupts are special in that it cannot be predicted when they occur. So we need to consider the possibility that we just enabled single-stepping and before the vCPU executes the desired instruction an interrupt appears.
Figure 4.3 shows how such a case is handled. A write access leads to a page fault, single-stepping is enabled, and we return control back to the guest. But before the VM can execute the instruction an interrupt occurs. In order to handle this interrupt, a stack frame is allocated instead of executing the instruction. After the stack frame is created the VM falls back into the VMM due to single-stepping [17]. However, it is possible that during the setup of the stack frame multiple memory accesses occurred that could not be logged because it is not possible to single-step the stack frame setup [17]. Back in the VMM, we handle the

VM exit as if the instruction was actually executed and set page protections as well as disabling single-stepping. After returning control back to the VM, the interrupt handler is executed and memory accesses are logged as usual. When the handler has finished, the VM jumps back to the instruction before the interrupt occurred and starts execution from there. Since this is the instruction that caused the previous write page fault and we previously set all protections again, the VM faults again and now we can handle this page fault as usual by disabling all protections and enabling single-stepping.

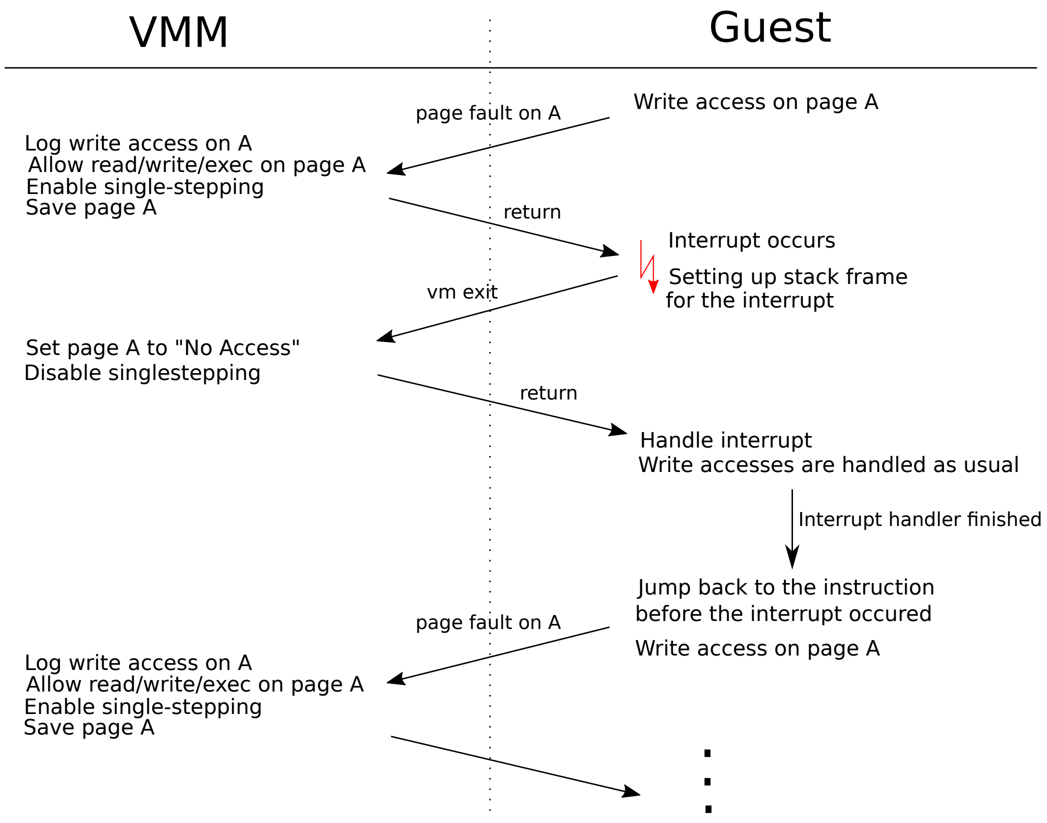

Figure 4.3: When single-stepping is enabled and an interrupt occurs, the VM exits without having the desired instruction executed

4.1.4 Recognizing a Relevant Page Fault

Up to this point, we only discussed how we handle a page fault but not which page faults exactly. We have to consider that page faults can occur which are not relevant to our logging mechanism. So we have to make sure that we only handle the ones that we are actually interested in.

The first category of page faults that is relevant to us are the ones that are triggered due to a nonpresent mapping. Such a page fault is triggered when the guest tries to access a page and the EPT does not contain a valid entry to translate this memory access. However, next to an entry being actually nonpresent we also have to take into account that an entry is considered as nonpresent if all access rights were removed [17]. This means that neither read, write, or execute access is allowed. But such an entry should be considered as present in the context of our logging since we remove all access rights of all pages at the start of an interval without the intention of making them nonpresent. So we have to mark them as present in a different way. This is done by introducing the flag `EPTE_MMU_IS_PRESENT` in an EPT entry. It is set when an entry is created and cleared when the entry is invalidated. On a page fault, we check whether this flag is set or not and if it is set we simply take our steps to log the memory access just as we described in previous sections and return from the handler. If it is cleared a valid entry is created and we log the memory access afterwards.
Into the second category fall page faults that are triggered due to a protection violation. But this is where we have to differentiate between a protection violation due to activated logging or due to the guest trying to access a legitimately protected page. So obviously the kind of page faults we have to ignore are the ones that come from a legit protection violation. Since a page fault carries no information on whether it was triggered due to logging or not we have to somehow remember why the affected page was protected in the first place. For a read protection violation, it is sufficient to know that the page fault was triggered due to a read access since KVM does not set read protections itself and we can be sure that the read protections were set by us in order to catch read accesses. But if the page fault was triggered due to a write protection violation it could be possible that it was not due to logging. KVM implements read-only memory (ROM) for the guest by removing the write access rights of the affected pages. To remember whether a page belongs to ROM or not, an additional flag for an EPT entry was introduced by KVM itself which is already used for the built-in dirty logging. `SPTE_MMU_WRITEABLE` is set if a page is write protected but should still be writable for the guest, which is the case for logging [2]. If the flag is cleared the page belongs to ROM and a write access by the guest would mean a legit protection violation. On a page fault that was triggered due to a write protection violation, we additionally check the `SPTE_MMU_WRITEABLE` flag in the EPT entry. If the flag is set we can handle this page fault and take our usual steps to log it. If it is cleared we ignore the page fault.

4.1.5 Activating and Deactivating Logging

Logging creates overhead and we want to minimize its effect on the overall execution time. So we should be able to activate and deactivate the logging as needed. To do this we make use of QEMUs and KVMs built-in activation mechanism for dirty logging. When QEMU enables or disables KVMs dirty logging, we hook into this process and do the same for our read and write logging.
When activating the logging, space for the read and write bitmaps is allocated and additionally, all page protections for all pages are set. This leads to a page fault for all future memory accesses.
To disable the logging we deallocate the space of the bitmaps but we do not remove protections of all the pages that are still protected due to the previously activated logging. Therefore, the guest will still run into page faults even though the logging was already disabled. Although we are not logging theses accesses anymore, we still give the needed access rights which eventually leads to unrestricted page access bit by bit.
To be able to recognize during a page fault if logging is enabled and, therefore, we have to log the page access, we simply check whether space for the read and write bitmaps is allocated. This way we do not have to manage a global flag which indicates the logging state.

4.1.6 Copying Bitmaps from Kernel Space to User Space

After an interval has finished and all memory accesses during it were logged we need to copy the bitmaps from kernel space to user space to be able to evaluate the gathered data further.
This is done during downtime when the checkpoint is created. Even though it is not possible for the bitmaps to change while they are copied since the virtual machine is stopped during downtime, we have to consider that in another scenario it could not be desirable to stop the VM before it is save to retrieve the bitmaps. So we have to make sure that while we copy the bitmaps, they cannot be changed through a page fault and a subsequent memory access log since this could lead to a bitmap that does no longer represent the accessed pages at the exact moment the copying was issued by QEMU. We ensure this by acquiring the lock for the EPT before we begin to copy. This prevents any other code from making changes to the EPT. If now a page fault occurs and the handler tries to change the access rights in the EPT, as usual, the handler has to wait before it can acquire the lock since the lock is held due to our copying. The important part is that the memory access is logged in our bitmap only after the page protections have been updated. This way the bitmap will be changed only again after the copying has finished.
To prepare the EPT for the next interval the page protections have to be set for all

pages again. This is done right after the copying has finished, but before the lock is released, so we do not miss out on any memory accesses.

4.1.7 Logging DMA

Direct memory access (DMA) is a protocol that allows devices (e.g., mass storage like HDD or SDD) to use the RAM directly without the need of the CPU. This poses a problem since we want to log all accessed guest frames in an interval but so far only mentioned page faults as a way to do this. Page faults are created by the CPU, but because DMA completely bypasses this, a page fault will never be triggered due to DMA. Even though one could argue that on each DMA a corresponding CPU access is required because the data that is transferred through DMA has to be processed somehow anyways. But we want to make sure that our memory trace is as complete as possible under any circumstances and we, therefore, have to log DMA separately.
DMA events are handled in user space by QEMU. To log a DMA event we would simply have to hook into the emulation of it and set the corresponding bit in the bitmap. Since we are in user space and not in KVM anymore we would have to create another set of read and write bitmaps.
However, logging DMA was never implemented in the context of logging with page faults since the slowdown we experienced due to single-stepping was very significant in that the VM was no longer responsive. In Chapter 3.2.2 we already mentioned our concerns regarding the speed of this approach and even though we did not measure the exact slow down of our implementation we assume that we have a similar slow down as Schmidt measured [23].

4.2 Tracing in a Full System Simulator

As already mentioned in Chapter 3.2.2 we had concerns regarding the speed of using page faults to trace memory accesses. The slow down turned out to be very significant as we described in Chapter 4.1.7 which makes memory tracing in a full system simulation to our main approach.
Memory tracing describes the process of capturing all memory accesses during execution and thus allows for later analysis. To capture this data there are several different ways. The approach we chose is executing the workload in a full system simulator (FSS) with enabled memory tracing. As an FSS we chose SimuBoost (with QEMU as the FSS) because it already provides capabilities to trace memory accesses. But full system simulation comes at a very high execution time cost [22] which could make the workload behave differently than it would in a normal execution environment. This makes the approach of directly tracing the workload

in an FSS unfeasible for us. Here comes SimuBoost's support for checkpointing and deterministic replay into play. We first execute the workload in a hardware-assisted virtualization and simultaneously create checkpoints. During execution, we also record all non-deterministic events (e.g., interrupts). When the workload has finished each checkpoint is loaded into the FSS individually and to ensure that the simulation does not diverge from original execution we use deterministic replay which injects each non-deterministic event at the correct moment. During simulation, memory tracing is activated and all memory access events are sent to the SimuTrace server where they are stored.

4.2.1 Reducing System Load

Depending on the chosen interval length we measured that several billion memory accesses can occur in a given time frame. This puts a huge load on the system since for each traced memory access a 32-byte wide structure has to be sent to SimuTrace where it is compressed and saved to disk. Thus SimuTrace needs a lot of resources to do the compression which could hinder the FSS from running at full speed by occupying the majority of the CPU time. Hence it is desirable to reduce the amount of data that has to be sent to SimuTrace.
QEMU is able to emulate guest platforms on different host platforms. The host and guest platform do not have to be the same and therefore QEMU has to translate the binary instructions from the guest platform to binary instructions of the host platform. This is done by first translating the guest code into an intermediate instruction set (IIS). After that, the instructions are translated from the IIS to binary instructions of the host platform [11]. This process makes tracing memory accesses fairly easy. After every instruction that accesses memory in IIS code, a call to a memory trace function has to be added. This function is also translated to host platform binary instructions and will be executed by the host as well. But since we want to reduce the data that is sent to SimuTrace, we have to somehow decide whether a memory access should be traced or not. We do this by making use of the translation lookaside buffer (TLB). The TLB buffers address translations by saving the mapping from virtual page frame numbers to physical page frame numbers. Before the CPU walks the page tables to translate an address it checks the TLB where the desired mapping could already be stored. This way the processor does not have to perform the slow page table walk every time a memory access occurs and consequently the overall execution speeds up. For our purpose, we make use of the fact that when a mapping exists in the TLB we can be sure that the page was already accessed in the past and therefore only have to trace a memory access when its mapping is written into the TLB initially. However, a TLB entry does not carry any information on whether a page was accessed through a read or a write. So we have to save additionally to the mapping the type of ac-

cesses that occurred on a page. To do this we create our own buffer which is kept in sync with the TLB and keeps track of the type of accesses that occurred on the pages that are buffered in the TLB.

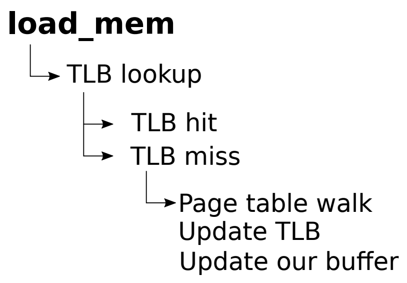

Figure 4.4: QEMU performs on a memory accesses a TLB lookup to translate virtual to physical addresses

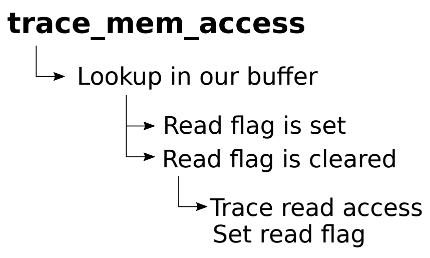

Figure 4.5: To decide whether we trace a memory access we check our own buffer

As an example, we will look at how exactly we make use of the TLB when a read access occurs (write accesses are handled the same way). QEMU first translate the guest instruction that reads from memory into an intermediate instruction, which is shown in Figure 4.4 [1]. To perform the read access, the virtual address has to be translated into a physical address. As we already discussed, instead of walking the page table directly, the CPU first makes a TLB lookup. If the translation is already buffered, it is a TLB hit and the translation can be used. If the TLB does not contain the correct mapping, the CPU has to walk the page table to translate the virtual address. After that, the new translation is saved in the TLB. Due to the limited size of the TLB, an old entry has to possibly be removed. To keep our own buffer in sync we replace the entry that belonged to the old TLB entry with a new one which signals that no memory access for the new mapping in the TLB has been traced, yet. Now the memory access can be executed and afterwards our memory trace function `trace_mem_access` is called, which is shown in Figure 4.5. In the trace function, we now have to decide if we actually trace this memory access. To do this we make a lookup in our own buffer and check whether a read access on this page was already traced in the past. If the read flag is set it indicates that such a trace has already occurred and we do not need to trace it this time. But if the flag is cleared we have to trace the memory access and set the read flag so that we do not trace read accesses in the future.
Even though it might sound like we only log each read and write memory access once and we get the actual working-sets as a result, this is not the case. When

[1]The names *load_mem* and *trace_mem_access* are only used for demonstration purposes and do not represent the names actually used in QEMU's implementation

a mapping in the TLB is displaced, the corresponding flags that were set in our own buffer are also cleared. At a later point, it is possible that the same mapping is placed into the TLB again but we have no way of restoring the flags that we had saved in our buffer previously and thus we will trace memory accesses on this page again. However, Kavita Bala et al. measured that the TLB hit rate can be very high and can range from 70% up to nearly 100% [9] which means that when we access a page we get a TLB hit most of the time which improves the efficiency of the trace.

Tracing Memory Access at Translation Time

So far we only discussed how we trace memory accesses that occur at runtime. But it is also possible that QEMU has to access the guest memory during translation time. Since QEMU still has to translate virtual addresses to physical ones, the page table and TLB are used as usual. So when QEMU accesses memory, first a TLB lookup is done and when no mapping is present, the page table is walked. After that, we simply call our memory trace function and check our own buffer as usual.

We have to put special attention to the memory accesses that occur due to instruction fetching. Instruction fetching is a memory access that obviously also occurs during translation time but with the peculiarity that the instructions are only fetched from the guest memory once even though they could be executed multiple times. Translating instructions is an expensive operation and therefore QEMU buffers translated instructions. This leads to a problem when tracing is activated over multiple intervals. In the first interval, the instructions are fetched, translated and buffered and we have no problem with tracing the memory access. If QEMU uses the buffered instructions multiple times in the first interval, it is still not a problem since we already traced the instruction fetch memory access. However, in the second and all following intervals QEMU still only uses the buffered instructions and we are not able to log the instruction fetches anymore. The solution to this problem lies within our method we use to trace the intervals. We load each checkpoint in its own FSS individually which makes sure that every interval starts with an empty instruction buffer and therefore all instruction fetches can be traced.

The same problem holds true for the TLB optimization in general. Tracing a memory access in multiple intervals becomes not possible until the TLB entry is replaced in between. When a TLB entry that was already accessed carries over from the previous interval into the current one, we are not able to trace further accesses. However, loading each interval in its own FSS is the solution to this problem as well.

4.2.2 Tracing DMA

We already discussed DMA in Chapter 4.1.7 and were presented with the problem that DMA bypasses the CPU and therefore we had to log it separately. For tracing in an FSS we have the exact same problem. We cannot trace DMA by inserting a trace function after every instruction that performs a memory access since these accesses are done by the CPU and not through DMA. Tracing DMA is not as easy as tracing RAM accesses since now we do not have a single place where we can call our trace function. Only a small portion of DMA is done through a defined interface and we would rather have to call the trace function in every place where a DMA access is emulated. To reduce the risk of overseeing parts of the code where such an emulation is done, we resorted to an approximation. We do not log each DMA access individually right when it happens, we rather log whole memory regions that are reserved for DMA and add them to the working-set.

Chapter 5

Evaluation

In this chapter, we will assess the working-set behaviour of different workloads to decide whether further effort in the direction of using the working-set to reduce checkpoint loading times is appropriate.
To do this we will measure the whole working-set size and additionally the separate sizes of the read and write working-set. Further, we want to decide whether the modified post copy method we presented in Section 3.1.1 is practical and could lead to improved loading times. We will do this by measuring the exclusive read working-set. This set contains all pages that were exclusively read during an interval and never modified. With the post copy mechanism, these are the pages that would have to be loaded on demand and the size of the exclusive read working-set is the deciding factor for the achievable speedup.
We will also calculate the Jaccard index for each workload and interval length. The Jaccard index describes the relation between the size of the intersection of the read and write working-set and the size of the whole working-set. This gives us a good insight on the correlation between the read and write working-set and how the workload uses memory in general. We can calculate the Jaccard index with the following formula with A being the write working-set and B being the read working-set:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{5.1}$$

This metric yields a result in the range of $[0, 1]$ and can be interpreted as the size of the intersection relative to the whole working-set in percent.
Lastly, we will review the model we discussed in Section 3.1 which proposed that the working-set grows slower the longer the interval length is.

5.1 Methodology

We already mentioned in Section 4.1.7 that using page faults for measuring the working-set turned out to be too slow. We did not measure the exact slowdown but we estimate it to be similar to the slowdown that Schmidt [23] experienced with his memory tracing. Even though we remove page protections as soon as possible, it is quite unlikely that a read access occurs before a write access and thus many page accesses still have to be single-stepped. This leaves us no other choice but to resort to our fallback approach which is tracing the working set in a full system simulator (FSS). Otherwise, it would not be possible for us to run all benchmarks in our limited time frame.

All measurements were performed on the same machine. The working-set for each workload is measured multiple times but with different interval lengths. If not stated otherwise the interval lengths we use are 500, 1000, 2000, 4000, 8000 milliseconds. We chose these interval lengths according to the model we discussed in Section 3.1 which suggests that the working-set size changes most with shorter interval lengths and grows slower with longer intervals. The exponential growth of the interval lengths allows us to see this behaviour best.

The workload is first executed in a hardware-assisted virtual machine (QSimu with KVM) where the checkpoints are created. This is done for each interval length and workload separately. To ensure that for each workload the state of the VM is exactly the same, we start QSimu additionally with the `-snapshot` flag enabled, which makes all changes to the VM's disk temporary and thus are discarded on exit. Further, no default devices are initialized other than the `std` VGA adapter to ensure that the measurements are not distorted by unnecessary device communication. The RAM assigned to the VM is 2GiB. After the workload has finished, the hardware-assisted VM is shut down and each checkpoint is loaded into an individual FSS instance. Our FSS is also QSimu. The FSS is configured exactly the same as the hardware-assisted VM with the only difference being that hardware-assisted virtualization is disabled and tracing is enabled. The SimuTrace server is the backend of the simulation and stores the memory trace. After the FSS finished with simulating a checkpoint, the memory trace is evaluated and the read and write working-set are calculated. As output, we get two files per checkpoint. One file which contains the page numbers that belong to the read working-set and another file that contains the write working-set. When all checkpoints are completely traced, metrics for each checkpoint are calculated. Eventually, the arithmetic means are calculated to get an average value for each metric for a workload and a specific interval length.

Since SimuBoost is still in development, it is possible that a checkpoint is damaged. This means that the FSS is not able to simulate the affected interval due to a bug in QSimu's checkpointing mechanism. When this happens we simply ignore the damaged checkpoint and move on to the next one. The highest amount of damaged checkpoints we measured for a realistic workload in a specific interval length was only 1% which keeps our measurements representative.

5.2 Test Setup

All measurements were performed on the same machine. The machine had the hardware and software specifications as listed in the table below.

Component	**Specification**
CPU	Intel(R) Xeon(R) CPU E5-2618L v3 @ 2.30GHz (8 cores)
Memory	4x Micron 18ASF1G72PZ-2G1A2 (8GB, 1866MHz)
Disk	SanDisk SDSSDHP2 (256GB)
Mainboard	Supermicro X10SRi-F
Software	**Version**
OS	Ubuntu 17.04 Zesty 64-Bit
Kernel	4.3.0[1]
QEMU	2.6.50[2]
Simutrace	3.4.0

Table 5.1: The hardware and software specifications of our test system.

5.3 Correctness Verification

We need to verify that the optimization for the memory trace we described in Section 4.2 works correctly. We can consider it correct if the resulting working-sets match the pages that the VM read and modified during execution.
However, we cannot prove the correctness of our implementation directly since the checkpoints created in QSimu cannot be loaded into a third party FSS to compare the resulting memory trace. Another approach would be creating an environment where we control each page that is accessed e.g., by writing a minimalistic kernel. Unfortunately, at this point in SimuBoost's development replaying such a kernel is not yet possible.
We have two indirect approaches to proof the correctness. The first approach

[1]With SimuBoost's modifications
[2]With SimuBoost's and our own modifications

is to compare the optimized tracing we implemented to the previously existing unoptimized tracing and see if they yield the same results. The second approach is comparing the number of pages in the write working-set to the pages that were logged as dirty.

5.3.1 Comparing Optimized and Unoptimized Trace

QSimu is capable of creating a full memory trace. But since this mechanism creates an excessive amount of data on a level of detail that we do not need, we implemented an optimization. To proof the correctness of our optimization we can simply compare the resulting working-sets to the unoptimized trace. However, this relies on the correctness of the unoptimized implementation which we assume at this point but cannot prove. Further, the unoptimized trace it is not yet capable of tracing instruction fetches which means that we have to disable this feature in our optimization as well.
We create 10 checkpoints with an interval length of two seconds during a kernel build and trace those checkpoints once with the optimized tracing mechanism and once with the unoptimized one.

	Optimized Trace Working-Set Size		**Unoptimized Trace Working-Set Size**		**Working-Set Deviation [%]**		
ID	Read	Write	Read	Write	Read	Write	Whole
1	10400	7533	10403	7538	0.02	0.06	0.01
2	13167	11559	13175	11563	0.06	0.03	0.01
3	20795	15197	20808	15215	0.06	0.11	0.00
4	10519	7194	10522	7200	0.02	0.08	0.01
5	12269	8051	12274	8075	0.04	0.29	0.01
6	19987	15051	19993	15060	0.03	0.05	0.01
7	30111	25990	30118	26004	0.02	0.05	0.01
8	25771	20840	25775	20845	0.01	0.02	0.01
9	27048	23860	27054	23866	0.02	0.02	0.01
10	26283	21753	26286	21762	0.01	0.04	0.01

Table 5.2: The deviation between the optimized an unoptimized trace is minimal but not zero. The working-set sizes refer to the number of pages in a working-set.

In Table 5.2 we see the size of the resulting working-sets given in pages per working-set. We can see that both the read and write working-set already deviate less than 1% from one another but the complete working-set deviates even less with only 0.01% on average. These results are not optimal as the deviation should be zero. However, the measurements we can make with our optimized trace are sufficiently precise and the gain in simulation speed is worth the precision loss of less than 1%.

5.3.2 Comparing Write Working-Set to Dirty-Set

We already discussed in Section 4.1.1 that the dirty-set does not equal the write working-set. The write working-set is rather a subset of the dirty-set and thus is always smaller. Further, the dirty-set is a subset of the whole working-set which leads to the following inequation that should be true for every checkpoint:

$$|Write\,WS| \leq |Dirty\,WS| \leq |Whole\,WS| \tag{5.2}$$

We get the size of the whole and write working-set from our own measurements which means that the inequation $|WriteWS| \leq |WholeWS|$ is always true since we calculate the whole working-set by merging the read and write working-set. We get the size of the dirty-set by using the debug output of the checkpointing mechanism. During each interval, dirty logging is enabled and at the end, during the downtime, the pages that are logged as dirty are copied into the checkpoint. Since we do not measure the number of dirty pages with our tracing mechanism we can compare the size of the write and whole working-set against the independently measured dirty-set size.
We checked this inequation for 298 checkpoints[3]. We found that for 136 of these checkpoints the inequation is not fulfilled. In 88 checkpoints the measured write working-set was bigger than the dirty set and the excess pages ranged from 25 to 9870. In 48 checkpoints the dirty-set was bigger than the whole working-set and the excess pages ranged from 59 to 26027 pages. These are very concerning results since the inequation should be true for all checkpoints. However, since we currently have no other way of tracing the working-set and do not have enough time to find out where these deviations come from, we will use the tracing approach nonetheless and rely on the plausibility of our measurements.

[3]During a kernel build with 2000 millisecond intervals

5.4 Benchmarks

We want to measure the working-set of different workloads to be able to get an picture of the behaviour of the working-set as broad as possible. Instead of executing the workloads manually and thus introducing inconsistencies between our measurements, we chose the Phoronix Test Suite [3]. It automates the whole execution and ensures that each time we run the same workload, we get comparable results. Since the term used in the Phoronix Test Suite for a workload is *benchmark* we will use these terms in the following interchangeably. As our workloads, we chose a broad spectrum of application scenarios to see whether the kind of workload has a significant influence on the behaviour of the working-set.

- **Linux Kernel Build** This benchmark compiles the Linux Kernel 4.13 and measures the time it takes to complete the build. It is a very CPU intensive benchmark and a popular way to gauge the performance of a system.
- **Apache Server** This benchmark launches an Apache HTTP server and measures how many requests a system can sustain. The Phoronix Test Suite classifies this benchmark as a system test which means that all system resources are used.
- **SQL** This benchmark measures the time that is needed to perform a number of insertions in a database. It resembles a realistic disk heavy scenario which could lead to an interesting working-set behaviour.

Next to real-world scenarios, we also want to assess the working-set of a best-case and a worst-case scenario. In a best case, as little pages as possible are accessed, whereas in a worst case as many pages as possible are accessed.

- **Best Case** The absolute best case would be a VM where no OS or kernel is executed. However, this is not a scenario we are interested in and thus we chose an idle running VM as our best case.
- **Worst Case** In the worst case the VM touches as many pages in an interval as possible. We used the `stress` tool to simulate such behaviour.

For each benchmark, we measure the working-sets for the interval lengths 500, 1000, 2000, 4000, 8000 milliseconds.

5.4.1 Linux Kernel Build

The benchmark `pts/build-linux-kernel` compiles the Linux Kernel 4.13 and measures the time the system needs until the build is completed. Compiling the Linux Kernel is a popular way to measure the overall performance of a system and is classified as a CPU benchmark in the Phoronix Test Suite.
In Figure 5.1 we can see the behaviour of the working-set over the different interval lengths. The working-set grows slower the longer the intervals get and the read and write working-set show the same growth pattern as the whole working-set. However, while the read working-set is almost identical to the complete working-set during all interval lengths, the write working-set is significantly smaller. In 500 millisecond long intervals the kernel build already uses on average 4.65% of the main memory which is about 95MiB. This has grown by 70% at the 8000-millisecond mark to an average of 7.94% accessed memory which is around 162MiB. When we look at the exclusive read working-set, we can see in the Table 5.3 that it is almost constant for all interval lengths. Between the 500ms and the 8000ms interval, it only deviates by 11% and even though the exclusive read working-set slowly grows in absolute size, the size relative to the whole working-set shrinks from 30.7% to 20.77%.
The Jaccard index shows an interesting working-set behaviour as well. Except for the 500ms interval, the intersection between the read and write working-set is 75% of the whole working-set which is quite high considering the sizes of the working-sets.
Baudis [10] also measured the write working-set of a Linux kernel build[4]. He stated that the size of the write working-set ranges from 5% to 10% which is roughly the same result that we got, but this time for the whole working-set. As an average write working-set size we measured 4.93% which does not quite fit Baudis' measurements. The reason for this is that he did not measure the actual write working-set, but the dirty-set, which is a superset of the write working-set and contains additional pages that were accessed through a read. We already discussed this issue in Section 4.1.1. Thus the actual write working-set is smaller than the dirty-set which makes our measurements plausible again. And since the dirty-set already contains a subset of the read working-set, the size difference between the dirty-set and the whole working-set is not very big which explains why we measured roughly the same size for the whole working-set as Baudis did for the dirty-set.

[4]with a 2000ms interval length and a VM with 2GiB RAM

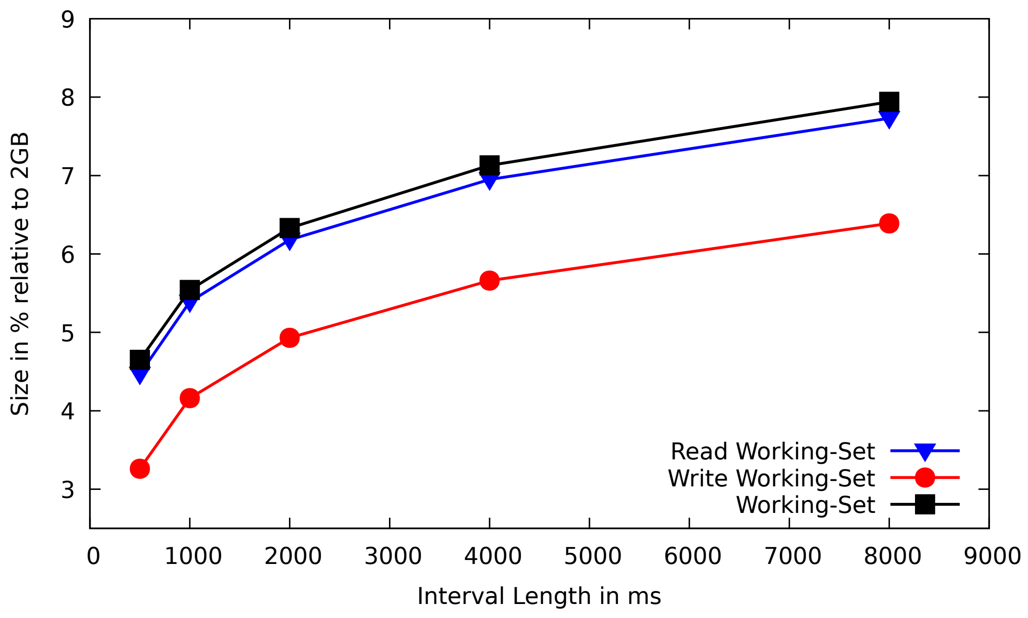

Figure 5.1: The working-set of a Linux kernel build grows slower the longer the interval length is.

Interval length [ms]		500	1000	2000	4000	8000
Working-Set	Aboslute	24,362	29,070	33,177	37,361	41,640
	Relative	4.65	5.54	6.33	7.13	7.94
Read Working-Set	Absolute	23,434	28,234	32,389	36,429	40,512
	Relative	4.47	5.39	6.18	6.95	7.73
Write Working-Set	Absolute	17,071	21,808	25,859	29,694	33,478
	Relative	3.26	4.16	4.93	5.66	6.39
Excl. Read Working-Set	Absolute	7,290	7,261	7,317	7,667	8,161
	Relative	30.70	25.54	22.62	21.12	20.03
Jaccard Index		0.66	0.72	0.75	0.76	0.77

Table 5.3: The measurements for the linux kernel build. The relative values are in % to 2GiB RAM. The absolute values are the total number of pages. The exclusive read working-set is measured relative to the complete working-set.

5.4.2 Apache

The benchmark `pts/apache` uses the `ab` tool to measures the number of requests a system can sustain with 100 requests being carried out concurrently and 1,000,000 requests in total. In the Phoronix Test Suite, this benchmark is classified as a system test.
In Figure 5.2 we can see the working-set sizes we measured for different interval lengths. In contrast to the Linux Kernel Build, the working-set grows nearly linear. To see whether the working-set also grows linear in shorter intervals, we additionally measured the 250-millisecond interval length. This reveals that the working-set has a faster than linear growth rate at very short intervals and only seems drops to a linear growth rate for interval lengths longer than 1 second. What the apache benchmark has in common with the Linux Kernel build is that the write working-set is significantly smaller than the whole working-set, while the read working-set almost matches it again and only drifts apart at longer interval lengths. The exclusive read working-set stays relative to the whole working-set almost constant at around 21% on average while growing 57% in absolute size. The Jaccard index stays almost constant as well. The intersection is averagely 75% of the whole working-set.

5.4.3 SQLite

The benchmark `pts/sqlite` performs a pre-defined number of insertions in an indexed database and is categorised as a disk test.
In Figure 5.3 we can see the behaviour of the working-set. This benchmark is especially interesting since it is a disk-heavy scenario and thus the memory access patterns are different. The first peculiarity is that the working-set grows linear up until the 4000-millisecond interval length and then the growth rate drops rapidly from around 50% to only 30% per interval length. Another interesting behaviour is that the read and write working-set have roughly the same size instead of the write working-set being significantly smaller. At the 8000 millisecond mark, the read working-set is even 14% smaller than the write working-set, which is a behaviour we have not seen, yet. This could be due to the fact that the benchmark performs insertions into a database and thus mostly carries out write accesses.
The exclusive read working-set also does not behave like in the previous benchmarks. It doubles in size between the 500 and 8000 milliseconds intervals but the proportion to the whole working-set drops from 49.06% to only 17.80%. This massive dropoff comes from the fact that the read working-set grows slower than the whole working-set.
Due to the read and write working-set almost matching in size the Jaccard index is also rather low. It ranges from 36% up to only 53% which stays way behind the intersection size of the previous two benchmarks.

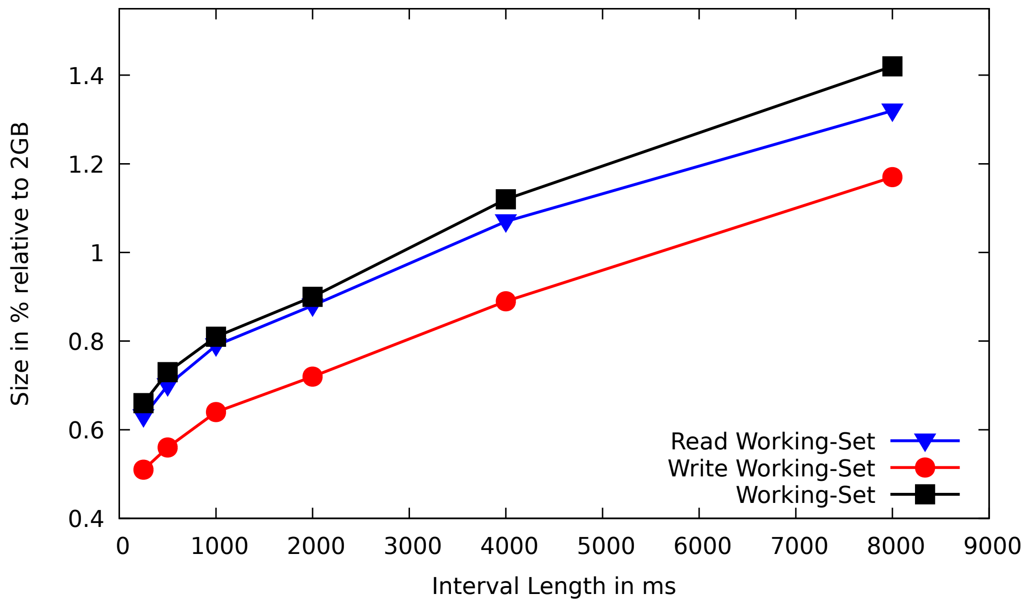

Figure 5.2: The working-set of the apache benchmark grows almost linear.

Interval length [ms]		250	500	1000	2000	4000	8000
Working-Set	Aboslute	3,476	3,809	4,242	4,732	5,867	7,439
	Relative	0.66	0.73	0.81	0.90	1.12	1.42
Read Working-Set	Absolute	3,309	3,663	4,137	4,618	5,596	6,915
	Relative	0.63	0.70	0.79	0.88	1.07	1.32
Write Working-Set	Absolute	2,654	2,956	3,344	3,748	4,661	6,143
	Relative	0.51	0.56	0.64	0.72	0.89	1.17
Excl. Read Working-Set	Absolute	821	852	898	983	1,205	1,296
	Relative	23.66	22.79	21.43	20.80	20.72	17.57
Jaccard Index		0.72	0.73	0.76	0.77	0.75	0.75

Table 5.4: The measurements for the apache benchmark. The relative values are in % to 2GiB RAM. The absolute values are the total number of pages. The exclusive read working-set is measured relative to the complete working-set.

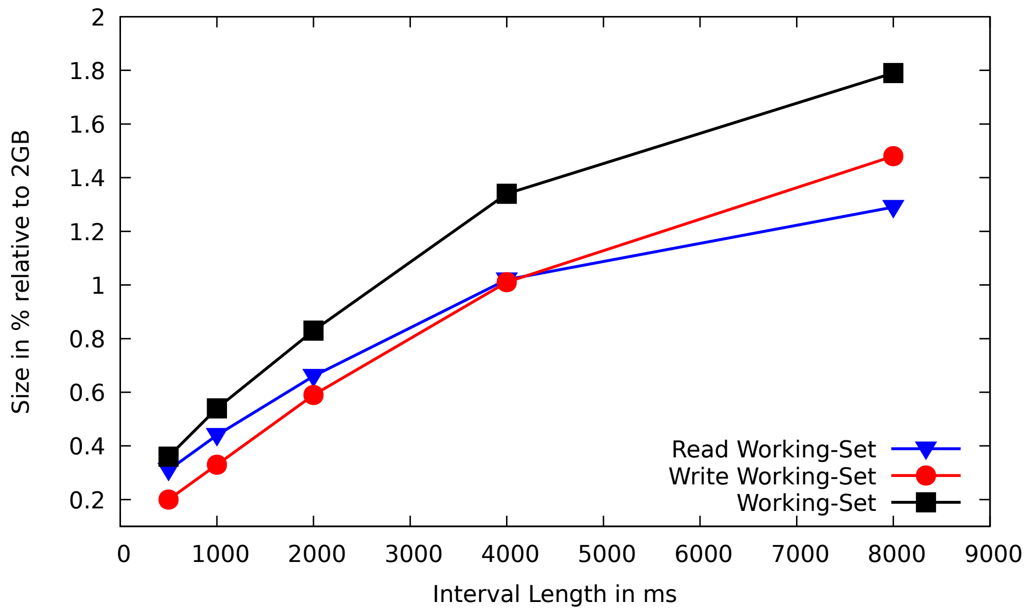

Figure 5.3: The working-set of the sqlite benchmark grows linear up until the 4000ms mark and then the growth slows down.

	Interval length [ms]	500	1000	2000	4000	8000
Working-Set	Aboslute	1,908	2,820	4,341	7,012	9,422
	Relative	0.36	0.54	0.83	1.34	1.79
Read Working-Set	Absolute	1,637	2,322	3,462	5,329	6,772
	Relative	0.31	0.44	0.66	1.02	1.29
Write Working-Set	Absolute	1,045	1,744	3,072	5,280	7,726
	Relative	0.20	0.33	0.59	1.01	1.48
Excl. Read Working-Set	Absolute	862	1,075	1,269	1,732	1,696
	Relative	49.06	39.33	29.11	24.28	17.80
Jaccard Index		0.36	0.41	0.46	0.49	0.53

Table 5.5: The measurements for the sqlite benchmark. The relative values are in % to 2GiB RAM. The absolute values are the total number of pages. The exclusive read working-set is measured relative to the complete working-set.

5.4.4 Idle

We chose an idle running VM as our best case scenario. In a best case, the VM touches as little as pages possible and, even though this most likely will not be a scenario that is relevant to research, it is important to know what amount of memory accesses is added to the workload through the running OS itself.
In Figure 5.4 we can see the working-set size of an idle running VM. It is not surprising that the read working-set almost matches the whole working-set while the write working-set is very small. But the overall size of the working-set stays very small with only 756 pages at the 8000-millisecond mark.
When we look at the exclusive read working-set in Table 5.6 we can see again that the whole working-set is mostly made up of the read working-set with a proportion of 64.51% on average.
The Jaccard index is the lowest we have seen so far. But this comes as no surprise since we already saw that the write and read working-set differ vastly in size which consequently makes the intersection small.

5.4.5 Stress Test

In a worst-case scenario, the VM touches as many pages as possible in an interval. To simulate such a case we used the `stress` tool which we configured to allocate 1512MiB of memory, touch (i.e., read and write) a byte on every page and then deallocate the memory again. This is repeated for as long as the stress test runs.
In Figure 5.5 we can see the working-set behaviour. Disregarding the 500-millisecond interval we see exactly what we would expect. The read and write working-set both match the whole working-set and contain roughly 75% of the 2GiB RAM which are the 1512MiB of memory we configured `stress` to allocate. The reason that we do get different results for the 500 milliseconds interval is that the presented values are only the arithmetic mean of the working-set size of all checkpoints. During most 500ms intervals the VM actually had enough time to touch the whole 1512MiB. But since each time the memory was touched it is deallocated and then reallocated, in some intervals the time was only sufficient to de- and reallocate the memory and then touch only a few pages which ultimately pulls the arithmetic mean down.
The exclusive read working-set, as well as the Jaccard index, show that the read and write working-set match the whole working-set.

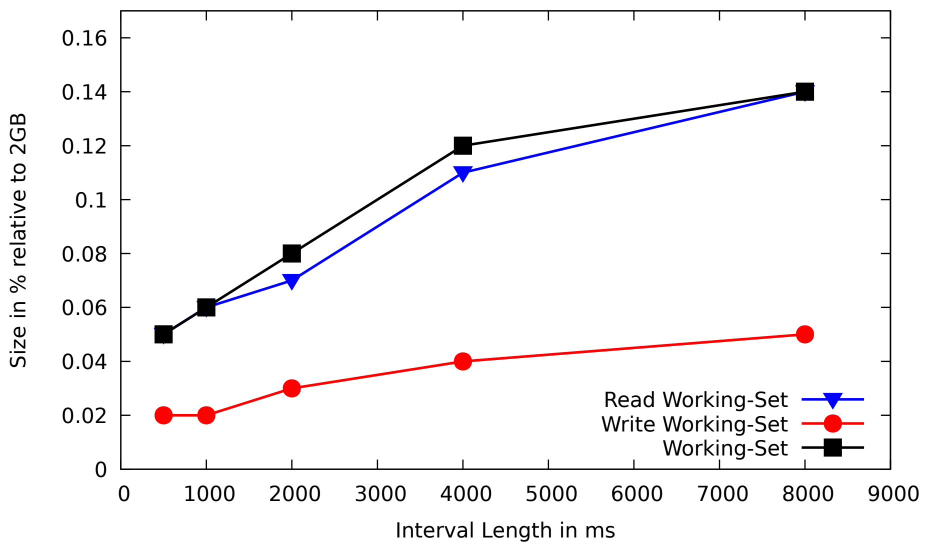

Figure 5.4: The working-set size of an idle running VM is very low.

Interval length [ms]		500	1000	2000	4000	8000
Working-Set	Aboslute	273	320	400	620	756
	Relative	0.05	0.06	0.08	0.12	0.14
Read Working-Set	Absolute	258	306	388	605	741
	Relative	0.05	0.06	0.07	0.11	0.14
Write Working-Set	Absolute	108	118	138	215	254
	Relative	0.02	0.02	0.03	0.04	0.05
Excl. Read Working-Set	Absolute	164	201	262	404	501
	Relative	61.83	62.13	64.75	66.81	67.04
Jaccard Index		0.33	0.33	0.32	0.30	0.31

Table 5.6: The measurements of an idle running VM. The relative values are in % to 2GiB RAM. The absolute values are the total number of pages. The exclusive read working-set is measured relative to the complete working-set.

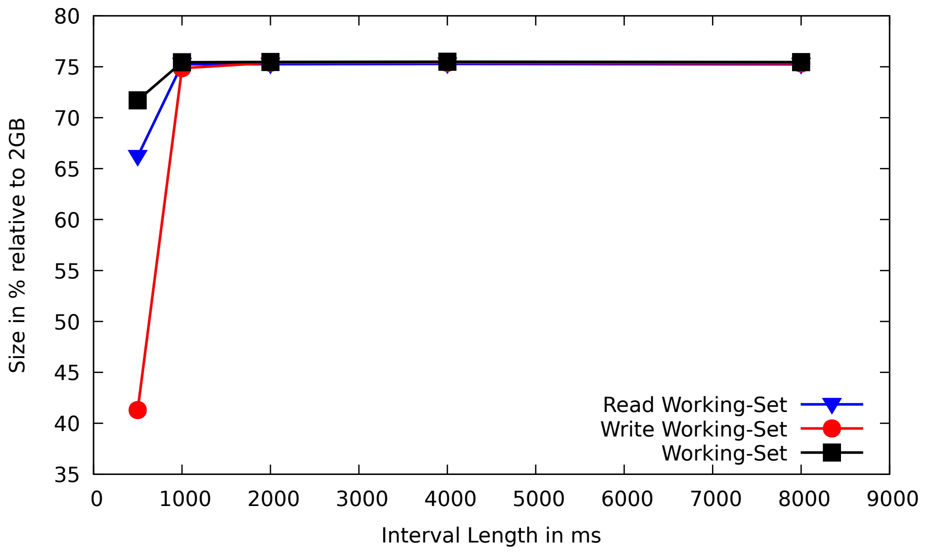

Figure 5.5: The read and wtite working-set of the stress test match the whole working set and make up in total roughly 75% of the RAM.

Interval length [ms]		500	1000	2000	4000	8000
Working-Set	Aboslute	395,977	395,584	395,676	395,818	395,629
	Relative	71.71	75.45	75.47	75.50	75.46
Read Working-Set	Absolute	347,099	394,469	394,545	394,702	394,475
	Relative	66.20	75.24	75.25	75.28	75.24
Write Working-Set	Absolute	216,467	392,478	395,268	395,303	394,967
	Relative	41.29	74.86	75.39	75.40	75.33
Excl. Read Working-Set	Absolute	159,510	3105	407	515	661
	Relative	40.48	0.78	0.10	0.13	0.17
Jaccard Index		0.48	0.99	1.00	1.00	1.00

Table 5.7: The measurements of the stress test. The relative values are in % to 2GiB RAM. The absolute values are the total number of pages. The exclusive read working-set is measured relative to the complete working-set.

5.5 Discussion

The problems we discussed in Section 5.3 are concerning and show that somewhere during tracing a bug still exists. But our measurements showed that the resulting traces are at least plausible. The working-set size we measured for the kernel build fits the measurements Baudis did and the overall working-set behaviour seems to be correct as well. The working-set we measured for the stress test also fits what we configured the `stress` tool to do which is touching 75% of the whole RAM. Thus we conclude that our measurements are at least precise and plausible enough to get a first impression on the overall working-set behaviour for different workloads.

In Section 3.1 we presented a model that predicts that a working-set grows slower the longer the interval length gets. No special time or size parameters are given, meaning that this model tries to predict the overall behaviour. For a realistic workload, Figure 3.1 should reflect the working-set size pretty well since the more time passes, the more pages can be touched by a workload but also the more pages will be touched that were already accessed previously. This leads to less and less newly touched pages which flattens the curve at the end. We can see this behaviour very well in Figure 5.1 for the Linux kernel build. It behaves exactly like it was predicted by the model. For the Apache and SQLite benchmark, we can also see that over time the working-set grows slower even though it seems to grow mostly linear. The reason that we cannot see the predicted working-set behaviour as well as for the kernel build could be that we measured too few interval lengths and 8000 milliseconds is just too short to see how the growth rate slows down. For the idle working-set, we can clearly see how the working-set grows slower even though it does not fit exactly the predicted curve. The stress benchmark is somewhat of a special case as we would need to measure very short interval lengths causing the VM to simply not have enough time to touch the whole RAM.

The concrete working-set sizes we measured for the workloads, except for the stress test, range from 0.05% up to 7.94% relative to 2GiB. The highest value comes from the Linux kernel build with 8 second long intervals, which is a resource heavy workload. Thus we can expect most workloads to use less than 8% of RAM during the same amount of time as we can see for the Apache and SQLite benchmark. The working-sets of both benchmarks do not exceed the 2% mark which leads us to the conclusion that it will be very beneficial for checkpoint loading times to only load the actual working-set instead of restoring the whole RAM. For an interval during which the VM runs idle the time to restore the RAM could be eliminated nearly completely since only very few pages would have to be restored. In our case, it would be less than 1,000 pages for an 8000-millisecond long interval. The stress test is a rather uninteresting case regarding the specific working-set size since nearly all of the RAM belongs to the working-set, just as it should.

We analyzed the size of the exclusive read working-set to decide whether the modified post copy mechanism we introduced in Section 3.1.1 could lead to a gain in the overall simulation speed. We found that the exclusive read working-set is around 20% of the whole working-set most of the time and only grows relative to the whole working-set as the interval lengths get shorter. This means that about 20% less data would have to be transferred to a server initially to restore a checkpoint. In the case of a kernel build and an interval length of 8000 milliseconds, this would save around 32MiB per checkpoint. However, the exclusive read working-set would still have to be loaded on demand which means that in this case the FSS would on average stop 8,161 times per interval, request the page from the producer, download it, and finally resume the simulation. Even with an unrealistically fast estimation that this would only induce a downtime of 1 millisecond per requested page, it still would add a total of 8.161 seconds to the simulation time. This additional overhead negates by far the benefit of only restoring the write working-set. For workloads with a smaller working-set, the same problem occurs. Even though the exclusive read working-set is smaller, the amount of time that can be saved by only loading the write working-set initially gets smaller as well. This renders our modified post copy mechanism as not practical, quite the contrary, it would add to the overall simulation time.

The Jaccard index gave us a good insight into the overall relationship between the read and write working-set. For the kernel build and the apache benchmark, we measured a high intersection size of around 74% on average whereas the SQLite benchmark on the other hand only reached the 50% mark. For an idle running VM, the intersection size barely reached one-third of the whole working-set. This shows that the relation between the read and write working-set varies greatly between different workloads and thus the effectiveness of optimizations beyond simply sending the whole workload (as we discussed in 3.1.1) depends heavily on the workload itself. Such optimizations would have to be tailored for the exact application scenario which is not within the scope of this thesis.

Chapter 6

Conclusion

Full system simulation (FSS) comes at a high simulation time cost. The goal of SimuBoost is to reduce this time by providing a high degree of parallelization. The workload is first executed in a hardware-assisted virtual machine where checkpoints are created periodically. These checkpoints save the state of the VM at the begin of an interval. Over a network, the checkpoints are then distributed to a server cluster where the VM states are restored in independent FSS instances and the interval is replayed.
However, loading a checkpoint in an FSS can take up to several minutes which increases the overall time that is needed to finish the simulation of an interval. Restoring a checkpoint includes restoring the main memory which currently means that the whole RAM image has to be transferred over the network and restored in the FSS. In this thesis, we assessed whether it could decrease the loading time when just the working-set or even only parts of it are restored.
To measure the working-set we introduced two different approaches. The first approach makes use of page protections and thus triggering a page fault for memory accesses allowing us to trace them. However, it turned that this approach induced a very high slowdown on the execution of the workload which is why we used our second approach. For the second approach, we first execute the workload in a hardware-assisted virtual machine, created periodical checkpoints, and then load the checkpoints into an FSS where we trace the working-set.
We found that the working-set grows slower the longer the interval length is. The biggest working-set we measured was 7.94% of 2GiB RAM. This lets us conclude that loading only the working-set instead of the whole RAM image could decrease loading times significantly. Another loading time optimization approach we assessed was a modified post copy mechanism where only the write working-set is restored initially and pages from the exclusive read working-set would have to be loaded on demand. Even though the loading time would decrease further, since only the write working-set is loaded, the time it takes to load the remaining pages

on demand negates the saved time completely. This makes the modified post copy approach not practical and we disqualified it as a possible optimization.

6.1 Future Work

Since the trace mechanism that is already present in the FSS made the simulation very slow, we implemented an optimization. We found that our optimization deviates less than 1% from the unoptimized trace and that the working-set measured with the optimized trace is always smaller than working-set of the unoptimized trace. This means that we are missing out on some memory accesses and thus further work will have to debug at which point this happens.
We also analyzed the relation of our measured write and whole working-set to the dirty-set. Since the write working-set is a subset of the dirty-set and the dirty-set a subset of the whole working-set, this relation should be confirmed by our measurements. However, we found that for a significant amount of checkpoints either the write working-set was bigger than the dirty-set or the dirty-set was bigger than the whole working-set. As this should never happen, further work should find out what causes this error in the tracing mechanism.

Bibliography

[1] Kvm homepage. https://www.linux-kvm.org/page/Main_Page.

[2] Kvm lock overview. https://www.kernel.org/doc/Documentation/virtual/kvm/locking.txt.

[3] Phoronix test suite. https://www.phoronix-test-suite.com/.

[4] Qemu homepage. https://www.qemu.org/.

[5] Qemu wiki. https://wiki.qemu.org/Main_Page.

[6] Amd64 virtualization codenamed "pacifica" technology: Secure virtual machine architecture reference manual. 2005.

[7] Intel virtualization technology specification for the ia-32 intel architecture. 2005.

[8] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. 2010.

[9] Kavita Bala, M. Frans Kaashoek, and William E. Weihl. Software prefetching and caching for translation lookaside buffers. 1994.

[10] Nikolai Baudis. Deduplicating virtual machine checkpoints for distributed system simulation. Bachelor thesis, System Architecture Group, Karlsruhe Institute of Technology (KIT), Germany, November2 2013. http://os.ibds.kit.edu/.

[11] Fabrice Bellard. Tiny code generator. QEMU git repo at commit a6e0344fa0e09413324835ae122c4cadd7890231.

[12] Fabrice Bellard. Qemu, a fast and portable dynamic translator. 2005.

[13] Nico Boehr. Evaluating copy-on-write for high frequency checkpoints. Bachelor thesis, Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, September30 2015.

[14] Christopher Clark, Keir Fraser, Steve Hand, Jacob Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. 2005.

[15] Peter J. Denning. The working set model for program behaviour. 1968.

[16] Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, and Yeh-Ching Chhung. Pqemu: A parallel system emulator based on qemu. 2011.

[17] Intel. Intel 64 and ia-32 architectures software developer manual. 2016.

[18] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. 2007.

[19] Rakesh Krishnaiyer. Data alignment to assist vectorization. 2015. `https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization`.

[20] Peter S Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. 2002.

[21] Andreas Pusch. Checkpoint distribution for simuboost. Master thesis, Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, October26 2017.

[22] 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.

[23] Thomas Schmidt. Evaluating techniques for full system memory tracing. Bachelor thesis, Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, October18 2017.

[24] Janis Schoetterl-Glausch. Intel page modification logging for lightweight continuous checkpointing. Bachelor thesis, Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, October31 2016.

[25] James E. Smith and Ravi Nair. The architecture of virtual machines. 2005.

[26] Michael H. Sun and Douglas M. Blough. Fast, lightweight virtual machine checkpointing. 2010.