Karlsruher Institut für Technologie

Evaluating Copy-On-Write For High Frequency Checkpoints

Bachelorarbeit
von

Nico Böhr

an der Fakultät für Informatik

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

Bearbeitungszeit: 01. Juni 2015 – 30. September 2015

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

Karlsruhe, den 30. September 2015

Abstract

Checkpointing a virtual machine (VM) in high-frequency intervals poses a challenge for checkpointing mechanisms. The stop-and-copy approach suffers from high downtime during checkpointing. That impedes its use in interactive scenarios, for example in workloads that utilize the network or interact with a human user. The pre-copy approach helps to reduce downtime by copying the VM's memory during its execution in multiple copy rounds. However, it is not able to reduce downtime reliably when the VM's writeable working set is too large.

This thesis aims to solve this problem by copying the VM's memory concurrently to its execution. Our approach utilizes copy-on-write to preserve the consistency of the virtual machine memory checkpoint image.

In our evaluation, we have proven this mechanism to provide a predictable, almost constant downtime of 6 ms during a kernel build with a two-second checkpointing interval. We have determined that our implementation yields a small constant overhead while copying pages and is therefore able to keep up with a checkpointing interval length down to 16 ms.

Deutsche Zusammenfassung

Das Checkpointing einer virtuellen Maschine (VM) in hochfrequenten Intervallen stellt Checkpointing-Mechanismen vor eine Herausforderung. Die Stop-And-Copy-Methode leidet an hoher Stillstandszeit während des Checkpointing. Das verhindert den Einsatz in interaktiven Szenarien, zum Beispiel bei Netzwerkanwendungen oder Anwendungen, die mit einem menschlichen Nutzer interagieren. Die Pre-Copy-Methode hilft, die Stillstandszeit zu reduzieren, indem der Arbeitsspeicher der VM während ihrer Ausführung in mehreren Runden kopiert wird. Allerdings ist diese Methode nicht in der Lage, die Stillstandszeit zuverlässig zu reduzieren, wenn das Writeable Working Set der VM zu groß ist.

Diese Arbeit zielt darauf ab, dieses Problem ebenfalls durch das Kopieren des Speichers der VM während ihrer Ausführung zu lösen. Allerdings wird ein Copy-On-Write-Mechanismus eingesetzt, um die Konsistenz der kopierten Speicherseiten sicherzustellen und daher zusätzliche Kopierrunden überflüssig zu machen.

In unserer Evaluation haben wir gezeigt, dass unser Mechanismus eine vorhersagbare, beinahe konstante Stillstandszeit von 6 ms während eines Kernel Build mit einem zweisekündigen Checkpointing-Intervall bietet. Wir haben bestimmt, dass unsere Implementierung kleine Fixkosten während des Kopiervorgangs zur Folge hat und daher in der Lage ist, mit kleinen Checkpointing-Intervallen von bis zu 16 ms Schritt zu halten.

Acknowledgements

I would like to thank my supervisor Marc Rittinghaus. He has invested countless hours of his valuable time to help me throughout all stages of my thesis. Thanks are also due to James McCuller for his help with the IT infrastructure I used for evaluating my work. Special thanks go to everyone who has proof read my thesis and to those who have provided their moral support.

Finally, I wish to thank my parents for their support and encouragement.

Contents

Chapter 1

Introduction

Virtualization has brought a large amount of advantages to computing. One of this advantages is that the state of a virtual machine can be saved at any time and then restored to the exact same state as it was before. This mechanism is known as checkpointing of a virtual machine.

The applications for checkpointing are versatile: Cully et al. have developed Remus [8] that allows for replication of a virtual machine to a backup host for fault tolerance. King et al. [16] presented a time-travelling checkpointing approach for operating system debugging. Rittinghaus et al. [20] have proposed SimuBoost, a technique to accelerate full-system simulators by combination of virtualization and simulation. All these applications have in common that they rely on taking checkpoints of a virtual machine in high frequency.

However, existing checkpointing mechanisms often expose undesireable performance characteristics, especially when applied to high frequency checkpointing. For example, some mechanisms interrupt the virtual machine execution while checkpointing for undesirably long time. This time is called downtime. Long downtime prevents the checkpointing mechanism from being applied to some workloads, for example workloads that require network connectivity or that require interaction with a human user.

The goal of this thesis is to develop a checkpointing mechanism that suits the requirements of SimuBoost and increases its applicability to workloads that are sensitive to long downtime.

The checkpointing mechanism implemented in this thesis extends the existing SimuBoost checkpointing implementation of Eicher [10]. By copying the virtual machine RAM during VM execution, we aim to eliminate a large part of the downtime. A copy-on-write mechanism is utilized to ensure a consistent copy of pages that are written by the virtual machine while the pages are being saved. The existing deduplicating checkpoint storage mechanism in Eicher's implementation is preserved.

Our evaluation proved copy-on-write to be a suitable mechanism for reducing the downtime in high-frequency checkpointing. We were able to demonstrate that copy-on-write exhibits a more predicable and lower downtime of 6 ms in comparison to a stop-and-copy checkpointing approach. We have also determined that our implementation yields a small constant overhead of 16 ms, which then limits the minimum checkpoint interval length to that size

The rest of this document is structured as follows: Chapter 2 presents fundamentals of full-system simulators, virtual machine checkpointing and virtual machine memory managmeent. Chapter 3 analyzes the characteristics of existing checkpointing mechanisms. In Chapter 4, we present the design of our copy-on-write checkpointing implementation. Chapter 5 then describes the implementation derived from our design. Chapter 6 documents the results of the evaulation of our implementation. At last, we conclude our thesis in Chapter 7 and give an outlook on future work.

Chapter 2

Background

This chapter will give a brief introduction to full-system simulators and their current limitations. Afterwards, we will introduce different approaches to checkpointing of virtual machines (VMs). Then, we will discuss important concepts in VM memory management, namely the shadow page tables (SPTs) and two-dimensional paging (TDP).

2.1 Full-System Simulation

A fundamental property of computers is that they are able to simulate other computers. This is true because the theoretical foundation of all computers, the turing machine, allows for the construction of a *universal turing machine*. It accepts a specially encoded form of another turing machine as an input and, given an input to the simulated machine, produces the same output as the original turing machine.

While this property has theoretical importance, it is also of use in practice. A *full-system simulator* simulates a whole computer system including all devices. This is useful because it is often undesireable to run software directly on hardware for several reasons. On one hand, hardware is usually *more complex and more expensive to set up*. It often requires physically setting up devices and connecting them to infrastructure required to run them (for example, a power supply or a network). Apart from the often significant costs of the hardware itself, additional costs arise from setting it up and later running it, for example for specially trained system administrators who manage the systems.

Additionally, hardware is *more difficult to automate*. In contrast to software, it cannot be reconfigured as easily as software, since that often involves physically removing or exchanging components from the system.

Furthermore, running software enables developers to gain *insight in greater detail*. While debugging and analysis capabilities in hardware are limited by what

has been implemented by the manufacturer, Full-System Simulators allow for the inspection of any part of the system, because every part is simulated in software and can thus be instrumented for analysis.

Depending on the amount of insight an user wants to gain on the system, different types of simulation exist. The tradeoff lies within the simulation performance. While the first type of simulator, a *functional full-system simulator*, runs comparatively fast, it also simulates less detail of the system. It mainly focuses on the result of each instruction executed and does not simulate details such as the processing in the pipeline of a processor. QEMU is an example for a functional full-system simulator [5].

The second type of simulator, a *functional full-system simulator with timing*, takes into account architectural details such as processing times of instructions and memory accesses. Because more details must be considered, this type of simulator generally runs slower than these without timing. A popular example for this type of simulator is Simics [17].

The third type of simulator are *architectural full-system simulators*. These are for example useful when the developer wants to analyze the design of the CPU architecture itself, i.e. the pipeline. Gem5 is an example for an architectural full-system simulator [6].

2.2 SimuBoost

However, traditional functional full-system simulators suffer from a performance problem. For example, executing a kernel build workload in QEMU results in a slowdown of 33 times in comparison to execution on hardware [20]. When the number of accessed physical pages is measured during simulation, the slowdown increases to 165 times.

This slowdown limits the applicability of full-system simulation. Especially simulation of long-running workloads takes such a long time that, despite the advantages, simulation becomes unfeasible. For example, the kernel build workload in [20] would run for 1.4 hours on hardware. In contrast, in a simulator, the same workload runs for nearly two days, namely for 46.9 hours.

Motivated by the increasing parallel processing power of current CPUs, it is desirable to parallelize full-system simulation. There are approaches which are able to parallelize simulation by executing the simulation of a number of CPU cores on the same number of CPU cores on the host machine [9]. Even though this can speed up the simulation of multi-core machines significantly, the execution speed of each core core is still low. This can only be improved by parallelizing the simulation of a single core. SimuBoost aims to simulate the same system at different points of time in parallel. For example, to simulate a three-second (s)

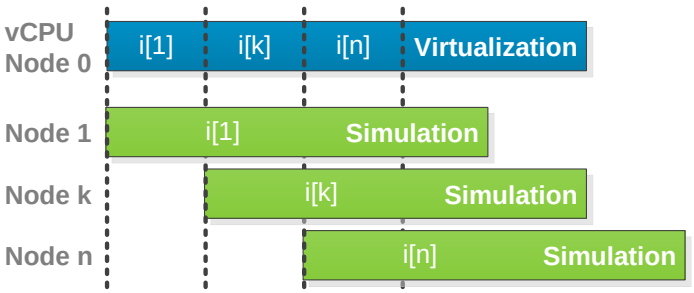

Figure 2.1: SimuBoost simulation parallelization. Simulation runs in parallel for different points in time. Through virtualization, the simulation can be bootstrapped even though simulation of the previous interval has not yet finished. Source: [20].

workload on a dual core system, the simulation would initially simulate the workload for the time intervals $0\,\mathrm{s} \leq t < 1\,\mathrm{s}$ and $1\,\mathrm{s} \leq t < 2\,\mathrm{s}$ in parallel on both cores. Afterwards, it would simulate the workload for $2\,\mathrm{s} \leq t < 3\,\mathrm{s}$.

However, simulation is an inherently difficult problem to paralellize, because every simulated instruction might modify the system's state as well as depend on the previous state. Thus, a parallelization approach that runs the simulation at different points of time in parallel has to know about the previous state of the simulation. However, this previous state might not yet be known, because the simulations are executing in parallel and the simulation of the previous state might still be running. In the example given above, the simulation for $1\,\mathrm{s} \leq t < 2\,\mathrm{s}$ has to know about the end state of simulation for $0\,\mathrm{s} \leq t < 1\,\mathrm{s}$, but since the simulations are executing in parallel, the result is still unknown. Thus, for this approach to work, the simulation would have to take a look into the future.

SimuBoost intends to use modern CPUs' support for virtualization. Because virtualization offers near-native execution speeds, executing a workload inside a VM is much faster than in a full-system simulator. Therefore, virtualization allows to predict the state of the simulation. This state can then be used to bootstrap the simulation of intervals that would otherwise require the state of a still running simulation interval. Figure 2.1 depicts this procedure.

To bootstrap the simulation, the VM's state must be saved regularily. This is called a VM checkpoint. The checkpoint interval length determines the possible speedup through parallelization. Rittinghaus et al. developed a mathematical model to predict the speed up characteristics of SimuBoost [20]. With realistic parameters, this model predicted a checkpointing interval of two seconds to offer a speedup of 84 times in comparison to a traditional functional full-system simu-

lator. Practical experiments have shown that the theoretical model yields accurate predictions [10].

2.2.1 Simutrace

We have already stated that full-system simulation allows a developer to gain greater insight in comparison to the debugging capabilities offered by hardware. This is because simulation is fully software-based and therefore allows the developer to instrument every part of the system. Often, a developer will instrument parts of the system to produce *tracing data* for later analysis. For example, it could be interesting to investigate memory access patterns by a system. This would produce large amounts of data that has to be stored. The developer therefore needs tools to support this task.

SimuBoost is therefore part of a tracing framework for full-system simulation called Simutrace. It handles the collection, reduction and storage of tracing data [19]. Collection is handled by an extension to a full-system simlulator (for example, QEMU). The simulator then acts as a client of the Simutrace storage server, where collected data is reduced (for example, compressed) and stored. A developer can then connect an analysis client to the storage server to inspect the collected data.

Communication between the storage server and its client (be it a collecting client, i.e. a full-system simulator or an analyzing client) happens over a RPC interface. In case the storage server and the client are running on the same machine, data is exchanged over a shared memory segment. This allows for fast zero-copy data transfer. The storage server can also run on a remote machine.

Simutrace calls the atomic entities that are stored *entries*. Each entry has a type, which is used to determine its size. For example, when tracing memory accesses, the developer may want to define a type that contains the address at which the access happened, the type of access (read/write) and, if it was a write, the updated contents of the respective page frame. Applications can then create *streams*, where they can store entries of a single type. By restricting entries to a fixed size, all entries in a stream can be accesed in $\mathcal{O}(1)$ [19]. We will now give an overview on the most important functions of the Simutrace API and how a developer will use them.

When applications want to store entries, they first create a stream for a certain type of entry (`StStreamRegister()`). After opening the stream for writing (`StStreamAppend()`), applications can obtain a pointer to memory where they can store the first entry by calling `StGetNextEntryFast()`. After doing so, the application can obtain a new pointer to the next entry by calling `StGetNextEntryFast()` again and so forth.

It is often difficult to predict the amount of data that is going to be stored in a stream. Simutrace therefore does not know how much space is to be allocated for a stream. To avoid expensive resize operations, the stream is buffered: Simutrace allocates a *buffer* that is divided into *segments*. These segments are of fixed size that suffices to store a number of entries. When the application calls `StGetNextEntryFast()`, the pointer is moved inside the segment. If the application reaches the end of a segment (i.e. there is no more space to store another entry in the segment), Simutrace switches to a different segment in the buffer. It can then start to process the data in the last segment fully asynchronously to the application. Note that Simutrace will only make a call to the storage server when it has reached the end of a segment. Otherwise, it will just increment the entry counter and check if there is enough space in the segment to store the next entry.

2.3 Virtual Machine Checkpointing

A VM checkpoint saves the complete state of a virtual machine for later restore. After restoring a checkpoint, the VM resumes execution as before. Thus, a checkpoint usually involves saving the VM memory, disk and other device states, for example the CPU's state. With disks in the size of terabytes and memory in the size of multiple gigabytes, VM checkpointing quickly becomes non-trival to handle due to the amount of data.

Checkpointing approaches can be classified by the following properties:

Downtime The amount of time the VM is unavailable due to checkpointing. The virtual machine monitor (VMM) stops VM execution during this time to be able to save consistent state for the checkpoint. Downtime occurs once per checkpoint interval. When the downtime becomes too high, interactive use of the VM such as network communication or user interaction can be negatively affected or are no longer possible.

Execution slowdown This describes by how much the VM execution is slowed down due to checkpointing. This slowdown might for example originate from increased memory access latencies by the VM. Even though both downtime and execution slowdown influence the time when a VM will complete a workload, we treat these seperately, because execution slowdown does not impact VM interactivity in general.

Data amount The total amount of data produced. This is important for the efficency of the checkpointing approach and can have implications for the VM downtime, because less data must be copied.

2.3.1 Pre-Copy Live Migration

Live migration moves a VM from one physical host to a different physical host without noticeable interruption of VM execution. Because this involves transfering the VM state, the techniques used for live migration are also interesting for checkpointing.

For live migration, it is important to reduce the downtime to ensure the migration works without noticeable interruption. It is thus undesireable to stop the virtual machine on the source host while transfering all state to the destination host. Instead, *pre-copy live migration* [7, 18] minimizes downtime by copying memory concurrently to VM execution. Since the VM is running during the copy operation, some of the copied pages have become dirty when the copy operation has finished. Thus, after the complete physical memory has been transfered to the destination host, dirty pages still remain to be send. They will then again be transfered concurrently to VM execution. This process is repeated until the number of dirty pages reaches a certain threshold. Then, the VM execution is stopped and the remaining dirty pages are copied. After that, VM execution can resume on the destination host.

This approach works because the number of pages to be copied decreases significantly in each copy round. During the first copy round, where the whole physical memory is copied, it is likely that the working set of the VM consists of a number of pages that is much smaller than the physical memory. In the next round, because less pages must be copied, less time is spent copying the pages and thus it is likely that the working set will be even smaller than before and so forth. After some time, the number of dirty pages will be small enough so the copy operation will have acceptable downtime.

This approach can also be adapted for checkpointing by simply writing the transferred data to a file instead of sending it to a different VM host. Ta-Shma et al. implemented continuous data protection on the Xen hypervisor by means of pre-copy live migration to the same host [23].

2.3.2 Incremental Deduplicating Checkpointing

While live migration only requires a single checkpoint, continuous data protection or SimuBoost require a high frequency of checkpoints. This imposes new challenges on the checkpointing mechanism. For example, downtime imposed by checkpointing would occur at each checkpointing interval, rendering the reduction of downtime even more important. However, because the same VM is checkpointed over and over again, this presents potential for improvements that are not effective with only a single checkpoint. In this section, we will discuss these potentials.

Because not all page frames of a virtual machine are changed in a checkpoint interval, it is unnecessary to copy the whole physical memory of the virtual machine. Rather, it suffices to save the complete physical memory once on the first checkpoint. In all following checkpoints, only frames which were changed by the virtual machine in this checkpointing interval must be saved. We refer to these pages as *dirty frames* or *dirty pages*. The contents of non-dirty frames can then be obtained from the previous checkpoints.

Similarily, only a subset of disk sectors will change in a checkpointing interval and the same strategy can also be applied to the disk image of a virtual machine. Hereafter, we will call this mechanism *incremental checkpointing*.

Incremental checkpointing has found application in a number of cases. Remus [8] and Kemari [24] replicate a VM to a backup host by means of incremental checkpointing to achieve high availability. King et al. [16] employed incremental checkpointing to solve problems in operating system debugging such as non-determinism and influence of the debugging process on the system itself.

In [4], Baudis measured the amount of dirty frames during a kernel build. For a two second checkpointing interval, the number of dirty frames averaged at 25 352. For a virtual machine with 2 GiB of RAM, this reduced the amount of data to be copied to about 5 % - 10 % and thus greatly improved efficency.

Baudis also found that it is possible to reduce the amount of data through *deduplication* of page frames and disk sectors. By hashing the contents of pages and disk sectors, those with duplicate content can be identified and thus must only be saved once. Duplicates could be identified within a checkpoint as well as across checkpoints.

2.4 Virtual Machine Memory Management

An important part of virtualization lies within the memory virtualization. The VM will use the same techniques as it would when it was running on real hardware, that means it will expect to be in full control of the whole physical memory and will set up page tables to manage its memory. However, these page tables will translate from guest virtual addresses (GVAs) to guest physical addresses (GPAs). Because the VM actually shares the host physical address space with other VMs, another translation step is needed: GPAs must be translated to host physical addresses (HPAs).

This section will only take into account virtual machine memory management on the x86 architecture.

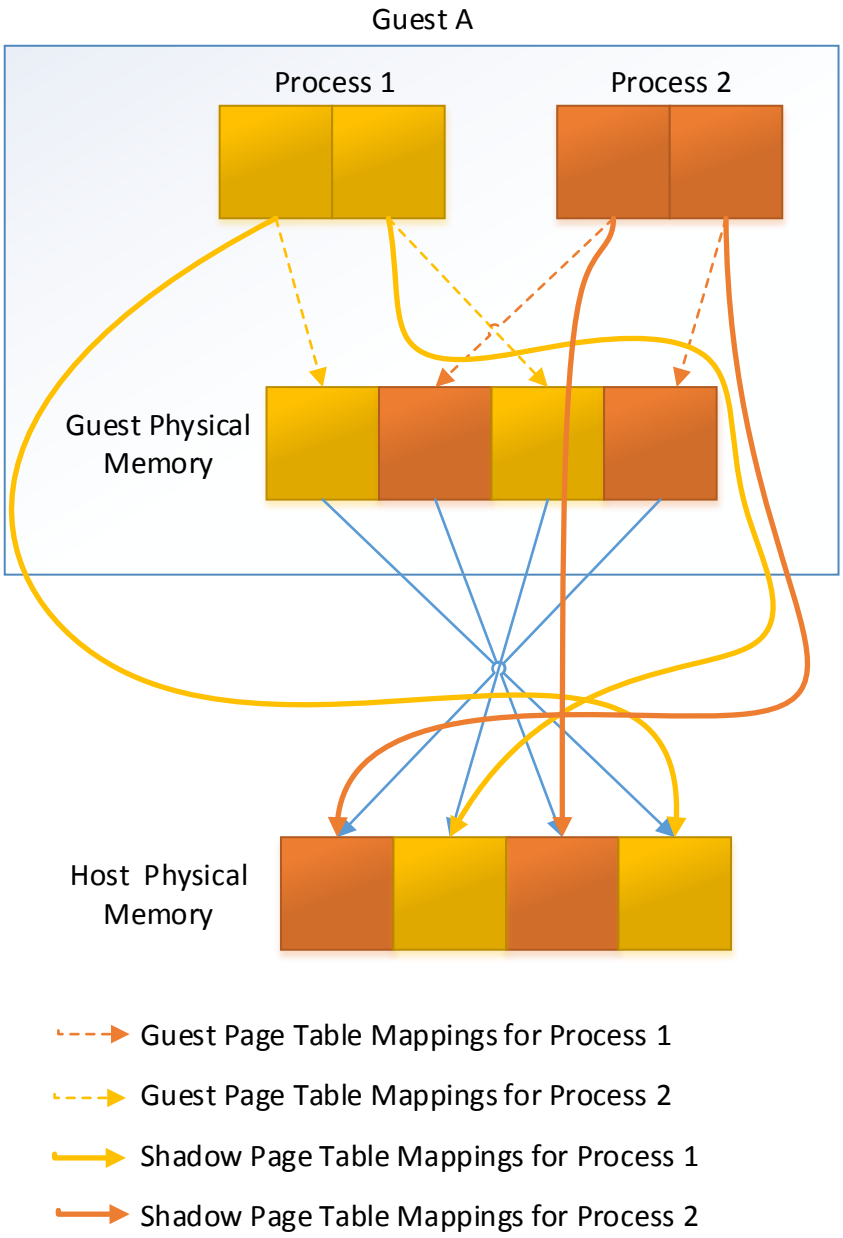

Figure 2.2: Shadow Page Tables: The guest page tables translate from guest virtual addresses to guest physical addresses. However, to ensure isolation, these page tables are not used for address translation. Instead, the VMM sets up shadow page tables which translate directly from guest virtual addresses to host physical addresses.

2.4.1 Shadow Page Tables

For each of its virtual address spaces (i.e. for each of its proccesses), the VM will set up a page table. These page tables will contain mappings from the guest's view, that means they will translate from GVAs to GPAs. When hardware does not support additional mechanisms for VM memory, it will expect these page tables to translate to the host physical address space. Therefore, the mappings set up by the guest are incorrect, because the VM actually shares the host physical address space with other VMs. To preserve isolation between VMs, the VMM must intercept all attempts of the VM to install page tables (i.e. writes to the `CR3` register) and instead install *shadow page tables* (SPTs). Shadow page tables will then contain the correct mappings from GVAs to HPAs [12]. For each virtual address space in each guest (and therefore each page table in each guest), one shadow page table is needed.

The VMM must take care that shadow page tables stay synchronized with guest page tables. When the guest updates a mapping in its page tables, the updated mapping must be reflected in the shadow page tables (with the modification of translating GPA to HPA, of course). Such updates can for example be detected by trapping translation lookaside buffer (TLB) flushes inside the guest [2].

Because page faults of the guest will initially trap into the VMM (and are then maybe injected into the guest), a missing mapping in the shadow page tables is not critical. This allows for lazy build of the shadow page tables: when a page fault occurs, the VMM can walk the GPT and fill the shadow page tables accordingly.

2.4.2 Two-Dimensional Paging

Because a large part of functionality is implemented in software, shadow page tables are not optimal. For example, for each context switch inside the VM, the VMM must be involved to reflect the context switch in the shadow page tables [2].

Processor manufacturers have therefore introduced hardware support for VM memory management in their products. Intel calls this feature "Extended Page Tables" (EPT), for AMD it is called "Rapid Virtualization Indexing"[1]. As both features are conceptually similar, we will use two-dimensional paging (TDP) as a vendor-neutral term. With TDP, additional levels of page tables are introduced that translate from GPAs to HPAs. These page tables are called nested page tables[2] (NPTs).

[1] older documents from AMD sometimes refer to this as as "Nested Page Tables"

[2] In KVM, this page tables are also refered to as shadow page tables. However, as these perform an entirely different translation when TDP is used, the author wants to differentiate between these to avoid confusion. Detailed information can be found in the Linux kernel source in `Documentation/virtual/kvm/mmu.txt`.

These are exposed to the hardware, so in case of a page fault, hardware can first walk the GPT to perform the GVA to GPA translation and then walk the NPT to perform the GPA to HPA translation. Because the hardware performs this additional translation step, isolation can be preserved even when the guest has full control over its page tables. Thus, the VMM can grant write access on the `CR3` register to the VM and the VMM does not need to be involved when a context switch occurs inside a guest.

Benchmarks by VMWare and AMD [2, 12] have generally found TDP to be beneficial for performance. However, because of the additional translation steps introduced, the benchmarks have shown that applications that stress the TLB can suffer from worse performance when TDP is used. This is because TDP introduces additional levels of page tables that must be consulted. However, by using large pages in the NPT and thereby reducing the number of page table levels to look up, this performance regression can be avoided.

Chapter 3

Analysis

In this chapter, we will first discuss challenges that checkpointing techniques must face, namely the amount of data to be saved and the downtime imposed on the VM. For each of these challenges, we will show why they are relevant and then afterwards evaluate existing techniques to tackle these problems. Then, we present their respective advantages and disadvantages with focus on their applicability to high-frequency checkpointing.

3.1 Checkpointing

Checkpointing is the proccess of saving a VM's state in a way that allows for later restore. It is also sometimes refered to as snapshotting. In this thesis, we focus on high-frequency checkpointing that aims to save the VM state in recurring intervals, each with a length of a few seconds. During an *interval*, a *workload* is executing in the VM. When the interval comes to its end, VM execution is interrupted and *downtime* occurs. When downtime occurs, a *checkpoint* is produced. It contains the state of the virtual machine at the end of a checkpointing interval, i.e its RAM, its disk and its device states. By the *checkpointing phase*, we hereafter refer to the time during that the checkpointing mechanism is running, i.e. the time between the first VM downtime and the last VM downtime. This terminology is depicted in Figure 3.1.

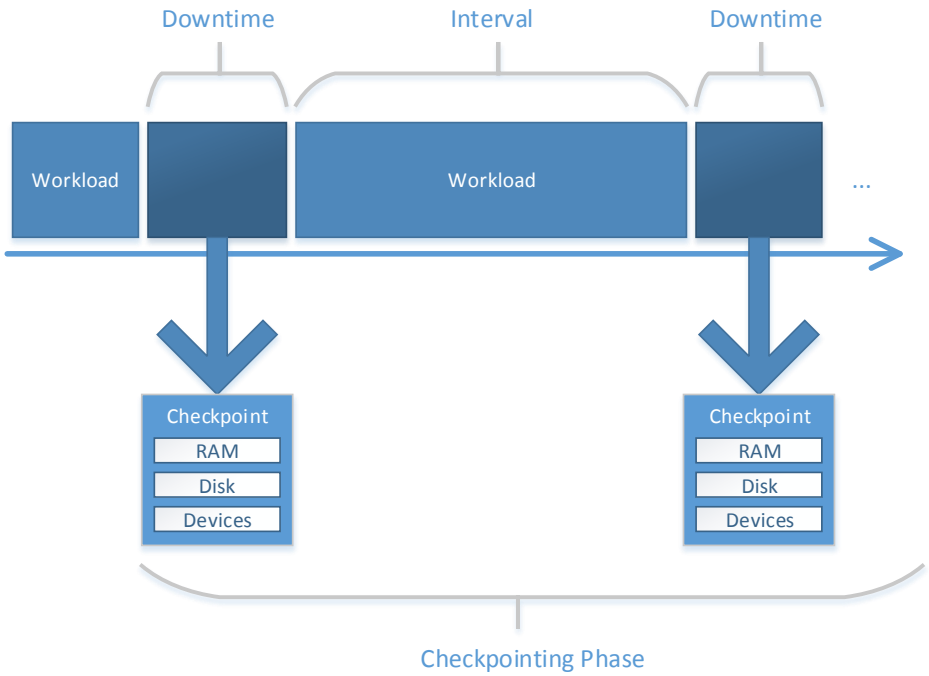

Figure 3.1: Checkpointing terminology. A workload executes in the VM. During the checkpointing phase, multiple instances of downtime occur in the VM. Each downtime produces a checkpoint that contains the VM state at the end of an interval.

3.2 Data Amount Considerations

With today's computers reaching multiple of gigabytes in main memory and terabytes for storage, the amount of data that has to be saved to capture the VM's complete state has become non-trivial to handle. Therefore, an important design goal for checkpointing techniques is to reduce the amount of data that must be saved on each checkpoint. However, it is important to preserve correctness, i.e. the created checkpoints must allow for the complete restore of the VM's state as it was during checkpointing.

3.2.1 Incremental Checkpointing

During VM execution, a significant amount of data will remain static [4], i.e. it will not change or will only change very seldom. This affects disk sectors, but

also memory pages. When a page or disk sector has changed in a checkpointing interval we will call it *dirty* hereafter. Data that was not changed in an checkpoint interval can simply be obtained from the last checkpoint it was changed in. It is therefore unnecessary to save this data when it has not changed. To ensure all data is saved, even when it is not modified in any later checkpoint, all data must be considered dirty on the first checkpoint. We call this approach *incremental checkpointing*.

For incremental checkpointing to work, the VMM must keep track of dirty data. For memory, it can do so by write protecting the guest memory in the shadow page tables or in the nested page tables, respectively. This will notify the VMM (for example, through a page fault) when the guest modifies a page in memory. The page fault initially traps into the VMM where it can mark the page as dirty in its internal data structures and remove write protection from the respective page. Thus, the page fault occurs only on the first write access in a checkpoint interval, minimizing the performance impact on the guest.

Baudis has measured the amount of dirty data in a checkpointing interval [4]. For a kernel build workload, he has found that incremental checkpointing will reduce the amount of memory pages to be saved to about 5 % to 10 % of all pages. In contrast, for workloads with high memory load such as the STREAM benchmark, the benefit of incremental checkpointing was much lower. It decreased the amount of pages to be saved to 54 % - 64 %.

Due to the incremental nature of this approach restoring becomes a more complex operation. A checkpoint that was created by copying disk or memory bit-by-bit for each checkpoint interval can simply be restored by copying the data back into the VM's memory or onto its harddisk. In contrast, for incremental checkpointing, data that has not been changed in a checkpoint interval must be obtained from a previous checkpoint. An index must be maintained to determine where the checkpoint data can be obtained from. When data has remained unchanged for a large amount of checkpoints, old checkpoints might have to be consulted to restore a current checkpoint. This has to be considered when moving old checkpoints to cold storage[1], as they might still be important for restoring relatively new checkpoints. Additionally, data that must be loaded is likely to be distributed over multiple checkpoints. This also means that the checkpoint data is likely also scattered through the host's disk. In comparison to bit-by-bit copies of the memory or disk on each checkpoint interval, more seeking is required on the disk and thus restoring will be slower. Although solid state disks (SSDs) perform better for data that is scattered over the disk, the restore performance will still be worse. This is because checkpoints are potentially scattered over multiple files and there-

[1] large storage that cannot be accessed as fast as local disks, for example tapes

fore more system calls are required for reading and techniques such as read-ahead cannot be as effective.

3.2.2 Deduplication

Incremental checkpointing reduces the amount of data that must be considered for a checkpoint, thus eliminating duplicate data between checkpoints. However, duplicate data that for example originates from the VM writing identical data to two different pages, is still written twice. This problem can be solved through deduplication. In [4], Baudis used a hash function to identify duplicate pages. During a kernel build with a checkpointing interval of two seconds, deduplication could reduce the amount of saved pages by 11 % to 52 %. The disk deduplication rate for the same workload was between 7 % and 71 %.

While Baudis has treated disk and memory deduplication separately, Eicher [10] has introduced deduplication across disk sectors and memory pages. This has reduced the number of disk sectors to be saved. For a kernel build workload, 86 % of all dirty disk sectors could be deduplicated against memory (in a two second checkpointing interval). This is because most disk sectors are initially loaded into memory after reading from disk. This means that disk checkpointing automatically benefits from improvements to memory checkpoining.

3.3 Downtime Considerations

Downtime is the amount of time a VM is unavailable due to checkpointing. It is necessary because the VM might modify its state during checkpointing. The VMM therefore has to stop VM execution to be able to save a consistent VM state.

Some workloads are significantly disturbed by high downtime. For example, human users that interact with a VM during a desktop workload will be significantly disturbed in their activities by any noticeable VM downtime. Because the VM is completely stopped during downtime, there will be no feedback on the user's activities during downtime: for example, the VM will not even update the position of the mouse cursor during that time. Users could think the VM has locked up during downtime, because it does not respond to their inputs.

Workloads that require network connectivity are also disturbed by high VM downtime. When using a TCP connection to transfer data between hosts, a downtime that is higher than or close to the TCP retransmission timeout might lead to retransmits, because the VM cannot acknowledge data during downtime. In [15], these retransmits have been identified as the main factor for increased application response times during VM live migration.

Additionally, some TCP implementations use the round trip time (RTT) between hosts as a congestion indicator. When VM downtime occurs, the VM cannot generate acknowledgement packets (ACKs). It will generate these ACKs when downtime has ended. However, for RTT measurements, the processing time on the destination host is usually assumed to be neglectible. Thus, VM downtime will be accounted to the RTT. The communication partner might incorrectly assume a congestion event and thus reduce its congestion window and transfer less data than the actual connection would allow for.

For these reasons, it is desireable to reduce downtime to a minimum.

3.3.1 Stop-And-Copy

In the simplest case, VM execution is stopped while copying all of the VM's state. Stopping happens even if the VM does not modify memory or disk and therefore a consistent disk and memory checkpoint would be possible during VM execution. This approach is mainly interesting because it comes without runtime implications for the virtual machine: The performance is not affected during VM execution.

With the stop-and-copy approach, downtime is mainly depends on the amount of data to be saved: as all data is copied during VM downtime, more data to copy automatically results in more downtime. Thus, the techniques for reducing the data amount as described in the previous section can improve downtime significantly.

However, the database used for checkpoint storage in Baudis' implementation accounted for a great amount of downtime and thus made it difficult to reach the downtime requirements of interactive applications. In [10], Eicher has shown that replacing the database as used by Baudis with a simple `memcpy` operation can significantly improve VM downtime. With the I/O intensive Bonnie benchmark, memory checkpointing downtimes could be reduced to 166 ms for a two-second checkpointing interval and to 305 ms for a four-second checkpointing interval. However, for larger checkpoint intervals such as ten seconds, downtime increased to 1160 ms. This motivates the search for solutions which further reduce downtime.

3.3.2 Pre-Copy

Pre-Copy is an effective technique to reduce downtime, that is primarily used in VM live migration. VM live migration aims to move a VM from one VM host to another VM host, without noticeably affecting VM execution. This makes the stop-and-copy approach undesireable, because its downtime depends on the amount of data to be saved. Note that live migration does not benefit from incremental checkpointing, as only one checkpoint is required to transfer the VM to a

Interval	Component	Mean Time	StDev	Median
2 s	Disk	1 ms	4 ms	0 ms
	Device states	7 ms	0 ms	7 ms
	RAM	76 ms	297 ms	45 ms
8 s	Disk	4 ms	9 ms	2 ms
	Device states	7 ms	0 ms	7 ms
	RAM	187 ms	818 ms	56 ms

Table 3.1: Stop-and-copy checkpointing: Composition of downtime per checkpoint for two-second and eight-second checkpointing intervals

new host. In this case, incremental checkpointing does not offer any advantage, as there are no previous checkpoints from which non-dirty data can be obtained.

We have performed measurements on Eicher's implementation to analyze the composition of the downtime. Our results for a two-second checkpointing interval are presented in Table 3.1. The table shows that downtime in the stop-and-copy approach mainly originates from copying RAM pages. It is therefore desireable to reduce the time needed to copy memory pages. Usually, the VM will have pages which will not be modified during live migration. Thus, it is unnecessary to stop the VM when these pages are copied: a consistent copy can be obtained while the VM is currently running. In contrast, pages that are modified by the VM during live migration cannot be copied during VM execution because the copy might become inconsistent through a concurrent write access. These pages comprise the writeable working set (WWS). The pre-copy approach strives to minimize VM downtime by copying only the writeable working set during downtime and copying all other pages while the VM is currently running [7].

The problem lies within the estimation of the WWS. The VM's behavior cannot exactly be predicted, but only estimated. Pages that were estimated to be outside the WWS, but are actually in the WWS could be inconsistent, because the copy operation and the VM write might have occured at the same time. However, after copying the estimated WWS, it is easy to find the pages which were incorrectly estimated by obtaining the list of dirty pages. These pages comprise the actual and accurate WWS.

By intersecting the set of pages that were estimated to lie outside the WWS with the set of dirty pages, the VMM is able to determine the set of pages that could be inconsistent. It can then attempt to copy the remaining pages in a further copy round. This is repeated until the number of remaining pages drops below a certain threshold. Because this number is much lower than the total number of pages to be copied, downtime can be significantly reduced. Then, the VM execution is stopped and the remaining pages are copied.

While the pre-copy approach works for VM live migration it is less suited for high frequency checkpointing. The reasons for this are threefold. First, as shown in [15] pre-copy checkpointing can affect the response time of a multi-tier application running inside the virtual machine and can therefore affect results when using checkpoints to bootstrap a simulation, such as in SimuBoost. Second, pre-copy cannot avoid significant downtime for workloads with a large writeable working set. As the VM is normally able to dirty pages at a higher rate than they can be copied, a significant number of pages can remain to be copied during downtime [7, see *diabolical* workload]. At last, the pages that are not part of the writeable working set are already handled by incremental checkpointing: these pages are not dirtied and thus will not be copied on subsequent checkpoints. The optimizations of pre-copy checkpointing will therefore be mainly relevant on the first checkpoint and largely irrelevant on consequent checkpoints.

3.3.3 Copy-On-Write

VM live migration migrates a VM from a host system to a different host system. After the migration, the VM continues to run on the new host system and no longer runs on the old host system. In contrast, VM replication introduces fault tolerance for a VM. It continuously replicates the VM to a backup host, while the VM continues to run on the main host. Upon failure of the main host, VM execution can then transparently resume on the backup host.

Because the replication occurs continuously, it is especially important to reduce the downtime imposed on the VM. Gerofi et al. [11] have utilized similarities in memory pages to speed up the replication and therefore the required downtime. To ensure page contents do not change during similarity analysis, they have employed copy-on-write (CoW) for the virtual machine's memory. Unfortunately, the focus of their work lies within similarity analysis and they do not present details on their CoW implementation and performance.

Sun et al. [22] have implemented CoW checkpointing for the Xen hypervisor. However, their evaluation focuses on long checkpointing intervals (more than 30 seconds). Also, they have not investigated if their checkpointing mechanism slows down the workload running in the VM.

It is therefore desireable to further investigate a CoW-based checkpointing mechanism and its suitability for high-frequency checkpointing.

3.4 Conclusion

In this chapter, we have identified downtime as a core problem of high frequency VM checkpointing. Existing work has shown that reducing the amount of data to

be saved on each checkpoint will improve downtime. For some workloads, for example interactive use by humans or use of the network, these downtimes are still too high and thus have to be improved.

For VM live migration, we found that pre-copy is a promising approach to reduce downtime. However, we found it to be unsuitable for high frequency checkpointing, because significant downtime can remain with particular workloads and the advantages offered by pre-copy are minimal when using incremental checkpointing. This motivates the search for a checkpointing approach that solves the downtime problem and is suitable for high frequency checkpointing.

However, copying pages concurrently to VM execution, as done in pre-copy checkpointing, can help reduce the VM downtime and is therefore an interesting approach to consider for high-frequency checkpointing.

Also, we have determined that through deduplication across disk and memory we can reduce the number of disk sectors to be saved. Our work will therefore focus on memory checkpointing.

Chapter 4

Design

This chapter initially defines the design goals for our implementation and explains their importance. Afterwards, we will derive a suitable checkpointing mechanism from these goals. Then, we present which problems a suitable implementation must solve. Afterwards, we present how we designed the interfaces our implementation will use to communicate with existing implementations, i.e. Simutrace.

4.1 Design goals

The most important aspect of a checkpointing mechanism is that it saves the state of a VM in a way that allows for a complete restore of the saved state without error. For the VM's memory, this can be verified by comparing the memory before taking a checkpoint with the memory after restoring the checkpoint. If there are no differences, the checkpointing mechanism preserved the memory of the VM and is therefore *correct*.

Furthermore, our approach should work with an *unmodified VM workload*, i.e. an unmodified VM operating system and unmodified applications inside the VM. This maximizes the applicability of our approach to all workloads that can run inside a VM: because the workload running inside the VM must not be modified, development costs for the checkpoint mechanism arise only from the initial implementation of the checkpointing mechanism and not for every workload run inside the VM. The implementation is then also independant from OS or application versions running in the VM and new releases of these components do not require adjustments.

As described in the previous chapter, existing checkpointing techniques, for example the pre-copy or stop-and-copy approaches, often suffer from unacceptable downtime. High downtimes make the checkpointing mechanism unsuitable

for interactive use and workloads utilizing the network. For these reasons, it is our goal to *minimize downtime* needed for checkpointing.

Moreover, the implementation should keep in mind that checkpointing can influence the performance of the workload running inside the VM. For example, additional memory accesses for copying the guest memory will slow down memory accesses inside the guest. While such influences can of course not entirely be avoided, the design must keep this in mind. Therefore, our design should have a *limited runtime impact on the workload* running inside the VM.

Lastly, we want our implementation *to be suitable for high-frequency checkpointing intervals*. Thus, our implementation must copy dirty pages fast enough to be finished before the next checkpointing interval arrives.

4.2 Mechanism

In the last chapter, we have seen that incremental checkpointing reduces the amount of data to be considered by a great amount. Our design therefore will make use of incremental checkpointing. As shown before, incremental checkpointing can however still result in significant downtime.

Downtime can however not entirely be avoided. For example, the VM execution has to be interrupted to capture a consistent view of the CPU's state, such as its registers. It is therefore desireable to do only lightweight and fast operations during downtime and perform heavyweight operations that take long time concurrently to VM execution. We have already identified that copying dirty pages comprises the largest part of VM downtime. Consequently, to reduce downtime, copying dirty pages should be performed concurrently to VM execution.

However, this results in a *data loss problem*. Because VM execution continues while copying dirty pages, the VM might modify a dirty page, destorying its original contents. Pre-copy solves the data loss problem by simply performing multiple copy rounds and then stopping VM execution to finally capture a consistent view. So some pages are still copied during VM downtime and this is part of the reason why the pre-copy approach exhibits undesireable performance characteristics for some workloads, such as multi-tier applications.

We want to avoid this influence on applications and therefore need a different solution to the data loss problem. Instead of multiple copy rounds, we want to avoid the occurence of possibly inconsistent copies in the first place. This can be achieved by a copy-on-write (CoW) mechanism: the VMM intercepts write operations of the VM and copies pages before they have been written by the VM. Consequently, there are two cases for a dirty page:

- The page was copied before the VM attempted a write access to the page and thus could be safely copied concurrently to VM execution. We call this case the *concurrent copy case*.

- The page is copied because the VM attempted a write access and thus must be copied to preserve the page contents. We call this case the *copy-on-write case*.

Note that because of the copy-on-write mechanism the checkpoint for an interval i is not available immediately after the VM downtime. Instead, it is constructed during VM execution in the interval $i+1$. The application consuming the checkpoints must take this into account.

Copy-On-Write Case To preserve correctness, it is important in the copy-on-write case to save the contents of a page before the VM actually modified it. Because we want our implementation to work with unmodified VM operating systems, the VMM must be instrumented to intercept write accesses by the VM.

To do so, we can utilize techniques from VM live migration: To allow for pre-copy live migration, most VMMs already keep track of the dirty pages within the VM. This can be achieved by write protecting the VM's pages in the NPT[1]. When the VM modifies a page that is write protected the VMM will be notified[2] and can then mark the page as dirty in its data structures. Instead of just marking the page as dirty, we can instrument this handler to save the original page contents and thereby implement copy-on-write.

Concurrent Copy Case To reduce the amount of data to be saved, our approach will make use of *incremental checkpointing*, i.e. only pages that have been modified in a checkpointing interval will be saved. All other pages are can be obtained from previous checkpoints upon restore. Therefore, we obtain a list of dirty pages at the checkpointing interval. In the concurrent copy case, we can then simply iterate over this list and copy all pages in the list.

When a page is being copied in the concurrent copy case, the VM might concurrently modify it. Therefore, the copy obtained might become inconsistent. An implementation has two choices to handle this:

[1] Alternatively, when not using TDP, the write protection is applied to the SPT. Some hardware also supports logging the dirty pages without write protection, for example Intel CPUs with Page Modification Logging (PML). For more information about PML see `http://www.intel.de/content/dam/www/public/us/en/documents/white-papers/page-modification-logging-vmm-white-paper.pdf`.

[2] How the actual notification is performed is vendor-specific. In the case of Intel's EPT, a `VMEXIT` with the "EPT Violation" exit code is triggered.

- accept the inconsistent copy and fix the problem later. This means that the concurrent copy case will obtain a (possibly) inconsistent copy of the page. But, because the VM modified a page, the copy-on-write case will also be triggered for the same page. Due to the the copy-on-write case interrupting VM execution, the copy obtained by it is guaranteed to be consistent. If a page was copied by the concurrent copy case *and* the copy-on-write case in a checkpointing interval, the VMM can discard the copy made by the concurrent copy case (because it does not know whether this copy is consistent). The advantage is that no synchronization between the copy-on-write case and the concurrent copy case is needed for ensuring a consistent copy.

- allow only one consistent copy. In the first approach, the copy obtained by the concurrent copy case is unnecessary. By introducing a synchronization mechanism that ensures VM execution is interrupted when the VM modifies a page that is currently being copied by the concurrent copy case, the unneccessary copy can be eliminated.

In our design, we have decided for the last option to eliminate unneccessary copies.

Even though VM execution cannot continue during the copy operation in the copy-on-write case, we expect that interactivity can be preserved. On the one hand, with peak memory bandwiths in the range of multiple gigabytes per second, copying a single 4K page is sub-millisecond operation and thus hardly noticeable. On the other hand, we will be copying pages concurrently to VM execution. This means that there is a reasonable chance the page was already copied before the VM attempts to modify it. This reduces the number of pages that would be copied in the copy-on-write case and thus reduces the number of times the VM execution must be interrupted.

4.2.1 Procedure

These considerations lead us to the following procedure for copy-on-write checkpointing:

1. Stop VM execution to copy device states and dirty disk sectors.

2. Obtain a list of dirty frames in the checkpointing interval that has just ended and write protect all frames to ensure write accesses can be intercepted by the VMM.

3. Resume VM execution, intercepting all memory writes.

4. During VM execution, iterate over the list of dirty pages and copy each dirty page.

By write protecting all page frames, we ensure that the VMM is notified of all write accesses to pages. The VMM can then check if the respective page is contained in the list of dirty pages and create a copy if so. Otherwise, the page was not modified and can be obtained from a previous checkpoint. In this case, the VMM will just mark the page as dirty, i.e. as to be saved at the next checkpointing interval. Figure 4.1 illustrates this procedure.

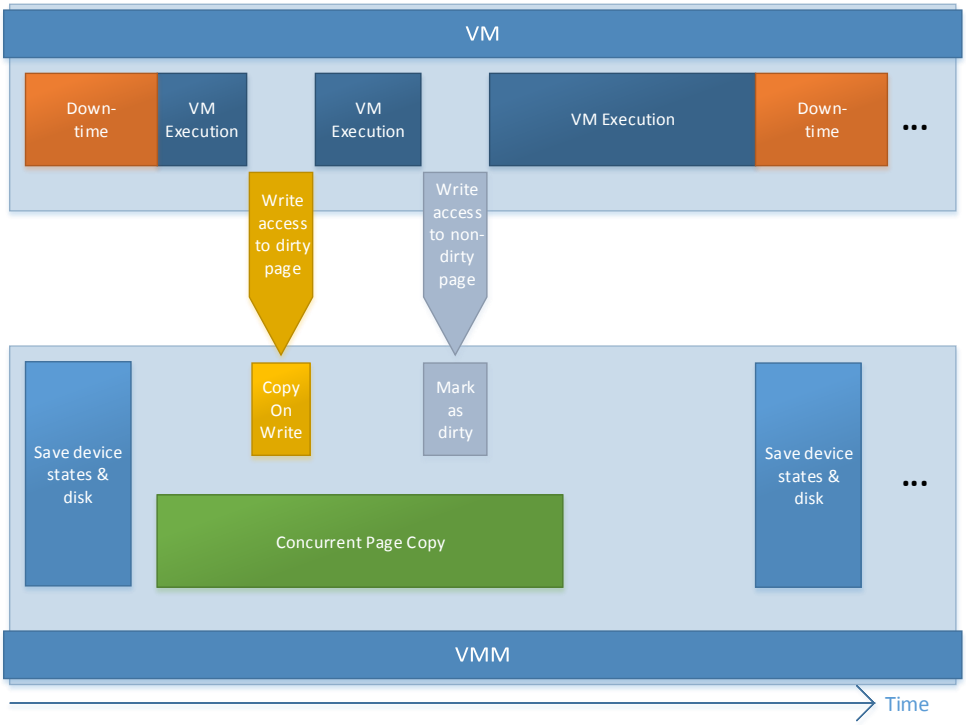

Figure 4.1: Copy-On-Write Procedure: When the checkpoint interval arrives, the VMM interrupts VM execution to save device states and disk. VM execution is then resumed and VM memory is copied concurrently to execution. To ensure a consistent copy, copy-on-write is used.

4.3 Conclusion

In conclusion, downtime cannot be entirely avoided during checkpointing. We have however determined that the downtime characteristics of the stop-and-copy approach can be improved upon by performing only lightweight operations during VM downtime and delaying heavyweight operations until the VM execution continues. Pre-copy follows this approach by copying dirty pages while the VM is running and only leaving a smaller number of pages to be copied during VM downtime.

Because this final copy round during downtime is the main reason for performance problems with some workloads in the pre-copy approach, we have chosen a copy-on-write approach as a different solution to the data loss problem.

Chapter 5

Implementation

In this chapter, we will describe our implementation. To do so, we will first show how we integrated our approach with existing software and Afterwards, we will detail our implementation in kernel space, with focus on the copy-on-write and the concurrent copy case implementation.

5.1 Technology integration

Eicher [10] has already implemented incremental deduplicating stop-and-copy checkpointing. His implementation extends QEMU and Simutrace. QEMU runs the VM and performs the actual checkpointing, while Simutrace is responsible for deduplication and data storage. KVM is employed by QEMU to access the virtualization capabilities of the CPU. To be able to focus on the copy-on-write mechanism and to profit from its optimizations (such as deduplication), we base our implementation on the existing work.

Eicher's implementation uses QEMU version 1.5.50. Its checkpointing mechanism was solely implemented by modifying the user space part of the VMM, the kernel space part could be left unmodified. This is in contrast to our mechanism, that also requires changes to the kernel space part. The reason for this is that, for the copy-on-write case, our implementation must be notified of write accesses by the VM. The Linux kernel does not currently expose a mechanism that allows the user space to be notified of these writes[1]. It is therefore desireable to use a VMM whose source code is freely available and that is compatible with the existing QEMU. We therefore decided to modify Linux Kernel Virtual Machine (KVM) in Linux kernel 4.0.

[1] Andrea Arcangeli is however working on a patch set [3] that allows handling page faults in user space.

5.1.1 Simutrace

In 2.2.1, we have already described how Simutrace achieves zero-copy data transfer from the client to the storage server. For optimal performance, our implementation should try to preserve these zero-copy characteristics of Simutrace. While Simutrace is an user space application, copies of the VM's dirty pages will be made in kernel space. We therefore need a mechanism that is able to transfer the pages from kernel space to user space, while at the same time trying to maintain the zero-copy characteristics.

To avoid additional copies, we must directly access the Simutrace buffer from the kernel. In the kernel, we cannot simply directly access user space memory, because the respective pages may for example have been swapped out. Care must be taken that these cases are correctly handled. We could do so using the `copy_to_user()` function, but this would be costly in terms of performance, because this function has to perform checks, such as the validity of the passed addresses, on each call. In our case, this would mean that these checks are done for each page of the VM we copy during checkpointing. Instead, we decided to establish a persistent mapping of the Simutrace shared memory in the kernel. This way, the performance overhead only occurs once at the first checkpoint and not repeatedly for each copied page.

An user space application would normally just call `StGetNextEntryFast()` after each copied page. Simutrace will then advance the pointer in the segment, only involving the storage server when the segment is full. As the name suggests, this call is normally very fast as long as there is enough space in the segment. If we would use this call after each written page, this would negatively impact performance, because each call would have to cross the kernel space boundary.

To solve this problem, we can just duplicate `StGetNextEntryFast()` in the kernel as long as there is enough space in the segment. We pass its address and size to the kernel, where we write to the segment until it is full. When this happens, we can return to user space to actually submit the entries and to obtain the pointer to the next segment. We call this mechanism a *memory proxy*, because the user space proxies the memory requests of the kernel to Simutrace. The memory proxy is implemented as a QEMU thread that waits in kernel until it either the segment is full or the checkpoint has finished. The action to take is then indicated by the return code. This mechanism is detailed in the next section.

5.1.2 KVM API

This mechanism obviously needs an interface between kernel space and user space, over that this communication can occur. To make this interface fit the existing KVM API, we quickly take a look at how the KVM API is designed.

While system calls are the traditional mechanism for user space applications to use services offered by the kernel, there are several other mechanisms. An important example are I/O controls (`ioctl`s). The main difference between a system call and an `ioctl` is that `ioctl`s are executed on a file descriptor and therefore are associated with that file descriptor. The available `ioctl`s depend on the file to which the file descriptor is associated to. The KVM API consists of a range of `ioctl`s. There are three classes of `ioctl`s in the KVM API [1]:

System `ioctl`s affect the whole KVM subsystem. Also used to create a new virtual machine. These are executed on a file descriptor of the `/dev/kvm` device.

VM `ioctl`s affect an entire VM, for example the VM memory. These are executed on a file descriptor obtained by the `KVM_CREATE_VM` system `ioctl`.

VCPU `ioctl`s affect a single virtual CPU. These are executed on a file descriptor obtained by the `KVM_CREATE_VCPU` VM `ioctl`.

`ioctl`s that take arguments provide a pointer to a struct in user space. The kernel can then copy this struct. As our implementation will operate on the VM's memory, we will implement the following VM `ioctl`s:

`KVM_COW_CHECKPOINT` signals that a checkpoint is to be performed now. To ensure a consistent checkpoint, this `ioctl` should only be executed when user space has stopped the VM. The kernel will then prepare the checkpoint, for example take a snapshot of the dirty bitmap. User space should have saved device states and disk sectors before calling this `ioctl`. When user space resumes VM execution, copy-on-write will be active.

`KVM_SET_MEMCOW_REGION` provides a memory segment to the kernel where it can store dirty pages. Beside error cases, this `ioctl` can exit with the following return codes:

- `KVM_MEMCOW_NEED_MORE_MEM` indicates that the region provided by user space is full. User space is expected to proxy this request to Simutrace.
- `KVM_MEMCOW_CHECKPOINT_FINISHED`: indicates that the checkpoint has been finished and all dirty pages were saved. The last segment provided to the kernel contains data that is to be submitted to Simutrace.

5.2 Kernel Space Implementation

Our implementation requires modification of the VMM page fault handler. Therefore, we have decided to modify the Linux KVM kernel module as included in Linux 4.0.0 to implement copy-on-write checkpointing.

5.2.1 Concurrent Copy Case

The copy-on-write case instruments the VMM page fault handler and therefore must run in kernel space. To make communication between the concurrent copy case and the copy-on-write case easier, we have decided to also move the concurrent copy case to kernel space. In an user space thread that controlls checkpointing, QEMU provides a memory segment to the kernel via the `KVM_SET_MEMCOW_REGION` `ioctl`. This thread remains in kernel space while dirty pages are copied to the segment. When the segment is full, the `ioctl` returns with the corresponding exit code (as described in Section 5.1.2) and QEMU proxies the memory request to Simutrace. When the checkpoint has been finished, the `ioctl` will return with a different exit code.

The actual copy of the dirty pages is then performed while the `ioctl` is in kernel mode. Similar to the Simutrace segment mentioned earlier, we had originally planned to also establish a mapping of the VM's memory in the kernel. However, this approach was proven to be unreliable, because only part of the VM's memory could be accessed this way.

5.2.2 Copy-On-Write Case

We have already determined that the VMM must be notified of changes to the VM memory before the change is actually performed. With Intel EPT, this can be achieved by write protecting the guest memory in the NPT. When the VM attempts to access a page that is write protected in the NPT, the CPU will triger a `VMEXIT` with the EPT violation exit code [13, p. 28-7].

However, some Intel CPUs also support Page Modification Logging (PML), a feature that allows the hardware to log pages modified by the VM without involving the VMM [14]. This is beneficial because avoids `VMEXIT`s and thus improves performance. However, this means the VM can modify a page without the VMM being notified before the change is performed. This is unsuited for copy-on-write. We therefore had to disable PML support in the Linux kernel by permanently setting the `enable_pml` flag to `0`.

In the design chapter we already discussed the necessity for synchronization between the concurrent copy case and the copy-on-write case. The sychronization mechanism must keep the congestion between copy on write case and concurrent

copy case to a minimum: because the copy-on-write case interrupts VM execution, it should run as fast as possible to minimize the slowdown of worloads running in the VM. Additionally, the concurrent copy case should not have to wait for the copy-on-write case if there is no ressource conflict, i.e. if they copy different pages, to ensure that the concurrent copy can finish as early as possible and therefore allow for high frequency checkpointing. We have therefore decided to create spinlocks for each guest frame. This approach trades minimal congestion off against space usage of the spin locks. However, as it can be seen in the Linux kernel source, the spinlock structure is a small data type. On the amd64 architecture (with `CONFIG_DEBUG_SPINLOCK=n`), the spinlock type `spinlock_t` occupies 4 B. For a virtual machine with 2 GiB of memory, the per-page locks would therefore occupy

$$\frac{2\,\mathrm{GiB}}{4096\,\mathrm{B}} * 4\,\mathrm{B} = 2\,\mathrm{MiB}$$

We consider this to be an acceptable trade off.

To further reduce congestion, we do not copy pages directly into the Simutrace segment in the copy-on-write case. Instead, we introduce the *copy-on-write queue* where pages for which the copy-on-write case was triggered are initially stored. After the concurrent copy case has finished, it will pop elements from the queue and then copy each element to the Simutrace segment, until the queue is empty. This completely eliminates the need for synchronization on the pointer to the current offset in the Simutrace segment. Moreover, when directly writing to the segment, it is possible that the segment does not offer enough space for the page. Then, the request would have to be passed to user space, where it would be proxied to Simutrace. The copy-on-write case would have to wait for this to finish, imposing a slowdown on the workload running inside the VM. The copy-on-write queue eliminates this problem.

Instrumentation for Copy-on-write

To find a suitable point where the copy-on-write case could be placed, we analyzed calls to the function `mark_page_dirty()` in the kernel source. It soon became clear that it does not suffice to instrument the EPT violation handler. For example, the Linux kernel can pretend to be a HyperV hypervisor [21]. This allows Windows guests to make hypercalls and thereby become an "enlightened" guest that is aware that is runs in a VM to improve performance. For this communication to work, the Linux kernel has to write the guest's memory. This dirties a page in guest memory without triggering an EPT violation (beacause it runs in the hypervisor context). We therefore analyzed each call to `mark_page_dirty` and searched for the place where the guest memory is actually modified by KVM. In most cases, the `mark_page_dirty()` function is called when the modifica-

tion was already performed and the original contents of the page are therefore no longer available. Therefore, we developed the function `kvm_page_cow()` that takes a GFN as an argument and adds the given guest frame to the copy-on-write queue. Calls to this function were placed at each location where guest memory is modified.

Additionally, KVM also has to emulate memory accesses by the VM. In case of a `CMPXCHG` instruction, the emulation is performed by simply executing a `CMPXCHG` in the hypervisor's context (while adjusting addresses). In the copy-on-write case, this opposes a problem: we must copy the page if the `CMPXCHG` succeeded (i.e., it has modified the guest's memory). But, if `CMPXCHG` succeeds, the page's contents have already been modified, because it is an atomic instruction. However, the original contents of the page are not lost. By examining the instruction arguments, we can determine the offset in the page where the modification occured. The instruction arguments also contain the value which the `CMPXCHG` instruction was seeing at the offset before the modification occured (otherwise, the compare operation would have failed and no modification would have been made). We can therefore copy the page contents after the modification occured and patch the offset in the page to resemble the original contents. The `kvm_page_cow_patch()` function handles this case.

Chapter 6

Evaluation

In this chapter, we are going to evaluate our implementation through measurements. We will first introduce the methology used for evaluation. Our evaluation will be based on the design goals introduced in Section 4.1:

- first, we are going to verify if our implementation is *correct*, i.e. it allows for complete restore of the VM's state.
- next, we will measure the *runtime overhead* of our implementation by executing a workload during checkpointing and measuring its runtime.
- third, we will verify that our approach is able to *reduce the downtime* in comparison to the stop-and-copy approach.
- fourth, we will verify if our implementation is *suitable for high-frequency checkpointing* by examining the time needed to copy the VM's dirty pages.
- at last, we are going to measure the *number of pages the copy-on-write mechanism must be used for* to evaluate the effectiveness of the copy-on-write approach.

At the end of this chapter, we conclude the results of our measurements.

6.1 Methodology

We set up the Simutrace storage server and the modified version of QEMU to run on the same machine. This enables the use of the shared-memory based zero-copy data transfer between QEMU and Simutrace. During checkpointing, all checkpoint data was written to disk as it would be in a real-world scenario.

The VM was set up with a single CPU core, 1 GiB of RAM and 20 GiB of disk space. It was running Debian Linux 8.0 and its default 3.16 kernel. QEMU was configured to run with the `vmware` VGA adapter, unless otherwise noted.

Unless otherwise noted, ten iterations were made for each measurement. In each measurement iteration, a new instance of QEMU and the Simutrace storage server was started. Before starting the storage server, its data directory was completely cleaned. QEMU was always started with the `-snapshot` parameter. This means that QEMU redirects all disk writes of the VM to temporary storage and does not perform them on the real disk image. This ensures a clean and reproducible environment for each measurement iteration.

In each measurement, we ran a build of the Linux 4.0 kernel. A kernel build stresses the VM's memory, disk as well as CPU. It was also used in the evaluation of the checkpointing implementations from [4, 8, 10, 22]. Given a configuration, a kernel build has a constant runtime (within some jitter) and is therefore also suitable for comparing the slowdown of different checkpointing mechanisms. We used a kernel configuration that results in a total build time of about 30 min without checkponting in the VM. All measurements were controlled by scripts. After starting QEMU, these scripts waited until the VM has booted. Then, the checkpointing mechanism was started. Immediately afterwards, the kernel build was started.

6.1.1 Evaluation environment

All measurements were performed on the following machine:

CPU	2x Intel Xeon CPU E5-2630 v3, 2.4 GHz (8 cores with 2 logical cores each)
Memory	64 GiB DDR3, 1866 MHz (4x 8 GiB modules per CPU)
Disk	256 GB SSD
Mainboard	Supermicro X10DRi

The following software versions were used:

Operating System	Debian 8.0, 64-bit
Kernel	Linux 4.0.0 with patches of our implementation
QEMU	1.5.50 with patches of our implementation
Simutrace	3.1.1 with modifications from [10]

6.2 Correctness verification

First, we verify the correctness of our implementation. We denote our implementation correct if the state of a virtual machine after restoring a checkpoint matches the state at which the checkpoint was created.

Because we have not modified Eicher's implementation of device state and disk saving, there is no need to verify their correctness. For memory checkpointing, we verify the correctness by means of a bit-by-bit comparison of the physical VM memory at the checkpointing downtime, hereafter called the *verification image*, with the physical VM memory after restoring that checkpoint.

To obtain the physical VM memory at the checkpointing downtime, we utilize a simple non-incremental stop-and-copy approach. QEMU was modified to copy the VM's memory to a file during VM downtime. When VM execution resumes, our copy-on-write checkpointing will also save the VM's memory. We can then restore the memory obtained by the copy-on-write approach and check if all pages were correctly saved.

This was done by starting a new instance of QEMU. To avoid that the VM execution resumes after restoring the checkpoint, QEMU was configured to freeze the guest CPU upon start (using the `-S` parameter). Then, a checkpoint was loaded and an image of the VM's memory was created by QEMU's `pmemsave` command. This image was then compared with the verification image. We denote a page as incorrectly saved as soon as one bit of the page does not match its contents in the verification image.

The measurements revealed that our implementation is unable to save all pages correctly. Table 6.1 shows the number of pages that was incorrectly saved while checkpointing. In the first checkpoint, 922 pages were incorrectly saved on average (stdev = 186, median = 940). Because a large number of pages must be saved on the first checkpoint (266 255 pages during all iterations), the average number of incorrectly saved pages on the first checkpoint only makes up 0.35 % of all pages that were saved.

In subsequent checkpoints, incorrectly saved pages only occur sporadically in low numbers. The highest number pages occurs in iteration three for checkpoint number two, where 15 pages are incorrectly saved.

A first analysis of the differences between the memory before creating a checkpoint and after restoring the checkpoint, showed that the addresses ranging from `0xB8000` to `0xBFFFF` (8 pages) were always zero in memory saved by our implementation but were non-zero in the verification image. This region of memory contained the text as displayed on the VM's console. This fact suggests the hypothesis that this memory is owned by the guest's emulated video card (the `vmware` video card was used in our tests). Further analysis revealed that QEMU does not provide user space addresses for this memory region to KVM (i.e., no

CP	Dirty Pages	Iteration									
		1	2	3	4	5	6	7	8	9	10
1	266 255	764	750	697	1067	720	1075	1096	1087	813	1151
2	34 665	0	0	15	0	0	4	4	4	0	0
3	14 615	0	0	0	0	2	0	0	0	0	0
4	13 816	0	0	0	0	0	0	0	0	0	0
5	13 314	0	0	0	0	0	0	9	0	0	0
6	12 922	3	0	1	0	0	0	0	0	0	0
7	13 492	0	0	0	0	5	0	4	0	0	0
8	14 911	0	0	0	0	0	1	0	0	7	0
9	13 655	0	0	0	0	0	0	2	0	0	0
10	15 450	0	2	1	0	0	0	0	0	0	0

Table 6.1: Number of incorrectly saved pages. The measurements were performed in ten iterations. During each iteration, ten checkpoints were performed while a kernel build was executing in the VM. Our implementation misses a small amount of pages, mainly on the first checkpoint.

`struct kvm_memory_slot` is set up). This means that our implementation does not copy these addresses, because it does not know that there is data to be copied at this memory region. This makes it clear that an implicit but important assumption of our design does not hold true: copying memory from kernel space alone is not sufficent and more information from user space must be included. Investigation on how this is best achieved is a starting point for future work.

Although the error rates of our implementation make its checkpoints unuseable, the number of errors is very low in relation to the number of pages saved. The highest number of errors occurs on the first checkpoint, where the error is in about 0.35 % of all saved pages. Because this error is minimal, the performance characteristics of our implementation are going to be very similar to an implementation that correctly saves all pages of the VM. In consequent checkpoints, the error rates are much lower and therefore their effects on the performance characteristics are even lower.

6.3 Workload Runtime

Checkpointing a virtual machine introduces overhead, for example through downtime. While our approach aims to reduce downtime, it also introduces additional overhead through the concurrent copy case and the copy-on-write case. To get a

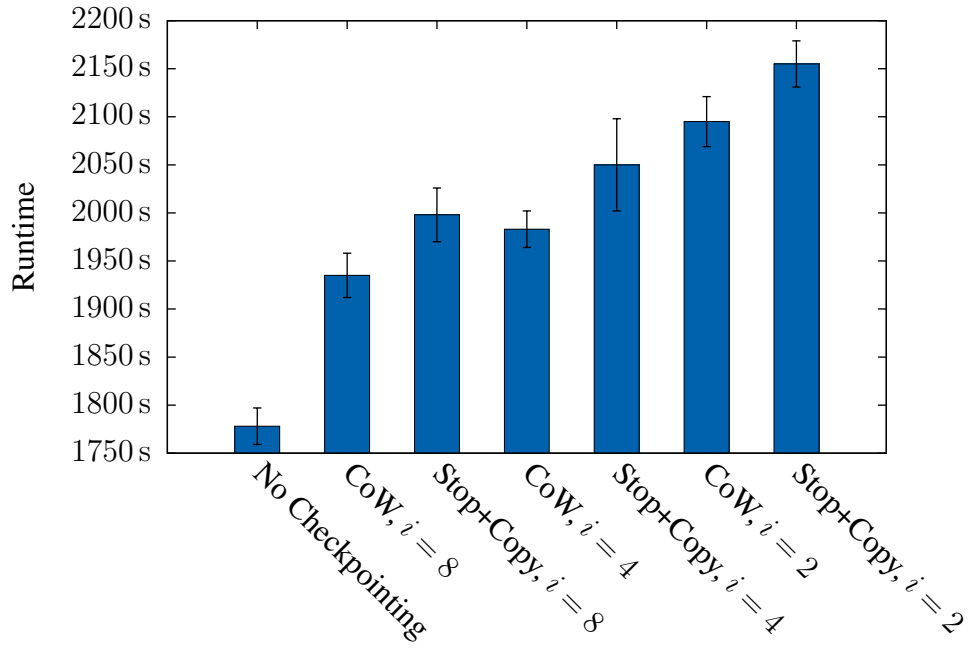

Figure 6.1: Kernel build runtime. The chart depicts the runtime of a build of the Linux 4.0 kernel. i denotes the length in seconds of the checkpointing interval used. Values given are averages over ten iterations. The copy-on-write approach offers a lower slowdown in comparison to the stop-and-copy approach.

high-level overview of the performance of our approach, we execute a workload in a VM and measure the time it takes to run.

Measurement is performed by setting up a virtual serial port to the virtual machine, where the VM is configured to open a text console. A script on the host system starts QEMU and boots the VM, waits until booting has finished, starts the checkpointing mechanism in QEMU and then immediately starts the workload in the VM by sending the appropriate commands over the serial port. The runtime is measured from the host system.

The runtime was measured without any checkpointing, with the stop-and-copy approach from [10] and with our copy-on-write approach. Each measurement was made with checkpointing intervals of two, four and eight seconds. This allows comparison of our approach with no checkpointing and with the stop-and-copy approach. All measurements were made with Linux 4.0.0. The measurements without checkpointing and with the stop-and-copy approach used a kernel with Eicher's instruction count patch [10, p. 49]. This was done to ensure compatibility with Eicher's implementation.

Figure 6.1 shows the results of the runtime measurements. Unsurprisingly, all checkpointing mechanisms slow down the workload. For the stop-and-copy approach, the slowdown is at 11 % for the eight-second checkpointing interval, respectively at 13 % and 18 % for the four-second and the two-second checkpointing interval. With the copy-on-write approach, a lower slowdown can be observed. The slowdown is at 8 % for the eight-second interval and at 10 % for the four-second interval. At the smallest interval of two seconds, the slowdown climbs to 15 %. From these numbers, we can draw the conclusion that the copy-on-write approach impedes the workload less than the stop-and-copy approach in all checkpointing intervals considered.

6.4 Downtime

A main goal of our design is to reduce the downtime during checkpointing. We are going to verify this by measuring the checkpointing downtime during a kernel build. This chapter aims to investivate if further downtime improvements are possible and how efficent our implementation handles downtime.

On the first checkpoint, a large amount of data must be saved and therefore a very high downtime can be observed. For example, for a two-second checkpointing interval, downtime values of 11 484 ms and 17 296 ms were measured for the copy-on-write and the stop-and-copy approach, respectively. For our work, we are mostly interested in the checkpointing mechanism's behaviour while executing a workload. We have therefore left out the first checkpoint from all measurements presented here. The downtime measurements for the first checkpoint can be found in the appendix in Tables A.2 and A.1.

The copy-on-write approach was able to significantly reduce the downtime during checkpointing. While the stop-and-copy approach offered a mean downtime of 84 ms for the two-seconds checkpointing interval, the copy-on-write approach reduced downtime to about one third, namely to 26 ms.

Additionally, the standard deviation of the downtime in the copy-on-write implementation is much lower than in the stop-and-copy approach (Tables 6.2 and 6.3). This hints at a more constant downtime behavior of the copy-on-write approach. In Figure 6.3, we have plotted the downtime behaviour during a kernel build in a single iteration for a two-second checkpointing interval. The graph indeed confirms that the downtime is less prone to fluctuation in the copy-on-write approach.

As the next step, we want to investigate the downtime behavior further. We aim to find potential ways to further improve downtime and to find performance issues in our implementation. Because we have only implemented copy-on-write for RAM, the first area for improvements would be the implementation of copy-

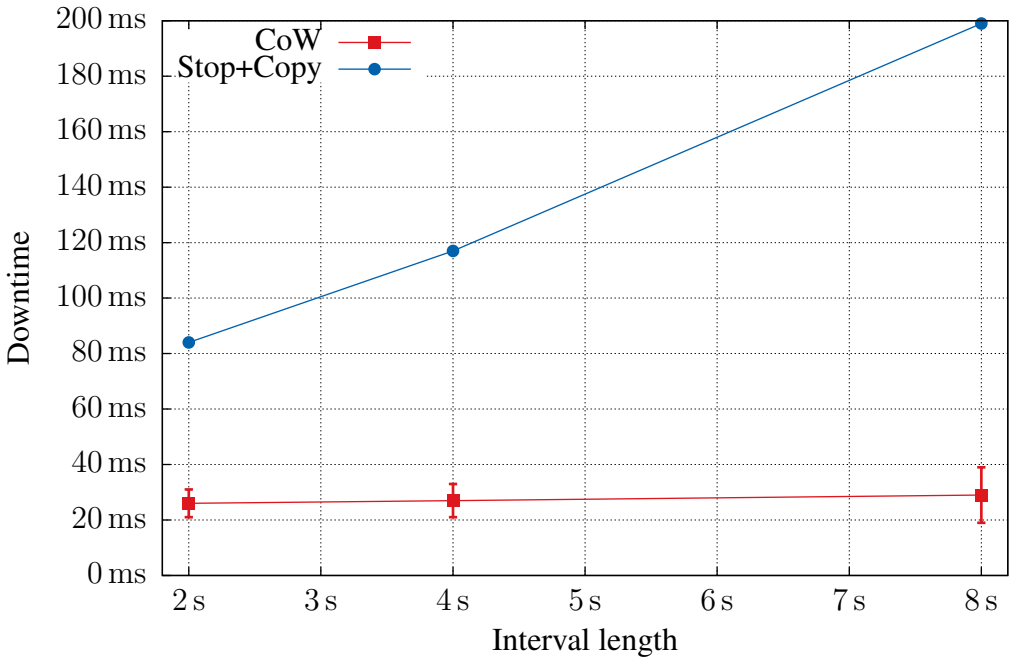

Figure 6.2: Average downtime. The chart depicts the average downtime during a kernel build for several checkpointing interval lengths. Values shown are averages over 10 iterations. The first checkpoint was stripped from all measurements. The standard deviation is very large for the stop-and-copy curve and has therefore been ommitted from the graph. It can be obtained from Table 6.3. The CoW approach offers a much lower downtime, that only slightly increases with the length of the checkpointing interval.

on-write for the disk as well. To investigate the possible improvements, we have listed the average number of dirty sectors and the average downtime in Table 6.2. As a comparison, we have included the same numbers for the stop-and-copy approach in Table 6.3. The average number of dirty sectors in each checkpoint increases with the interval length. Although more sectors are to be saved, the average downtime only increases slightly for CoW. This means that saving the dirty disk sectors only comprises a small part of the VM downtime. Therefore, copy-on-write for the disk image would only yield a small improvement for this particular scenario.

This motivates more detailed measurements on the composition of the downtime. Eicher [10] has already included code for detailed measurements on downtime overhead in his implementation. We have adjusted his code to our implementation to be able to measure the following downtime components:

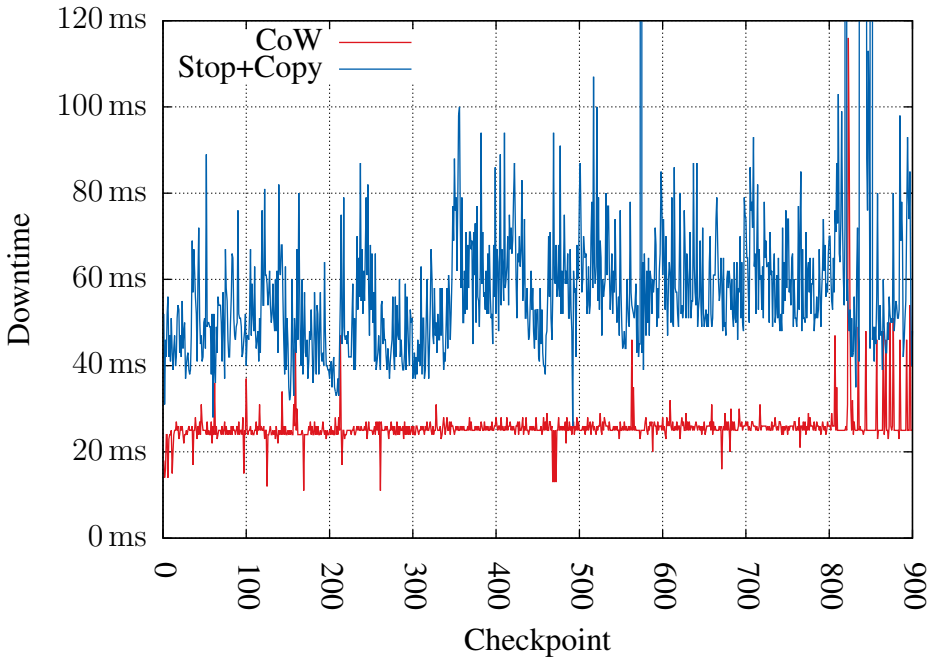

Figure 6.3: Downtime behaviour. The chart depicts the downtime during a single iteration of a kernel build for a two-second checkpointing interval with the copy-on-write and stop-and-copy approach. Some spikes in the copy-on-write curve and the stop-and-copy curve are cut off. The CoW approach results in a steadier downtime curve that stays almost constant.

Device state saving time the time needed to save the states of emulated devices of the VM, for example the video adapter.

Disk saving time the time needed to copy dirty disk sectors.

`ioctl` time the time that is spent in the kernel for the `KVM_COW_CHECKPOINT` `ioctl`. This corresponds to the time needed to set up data structures and to prepare the checkpoint in the kernel.

The measurements were made during a kernel build with a two-second checkpointing interval and with five measurement iterations. Table 6.4 shows the results of these measurements.

First, the measurements confirm the fact that the number of dirty sectors only has limited impact on the downtime with the CoW approach. Only 1 ms is spent with saving dirty disk sectors on average. However, as the standard deviation indicates, spikes to the tens of milliseconds can be observed.

Interval	Downtime			Dirty sectors		
	mean	stdev	median	mean	stdev	median
2 s	26 ms	5 ms	26 ms	75	288	4
4 s	27 ms	6 ms	26 ms	145	391	71
8 s	29 ms	10 ms	27 ms	280	606	27

Table 6.2: CoW Downtime. The table lists average downtime and average number of disk sectors to be saved for several checkpointing interval lengths. Although the number of disk sectors to save increses with the interval length, the downtime increases only slightly.

Interval	Downtime			Dirty sectors		
	mean	stdev	median	mean	stdev	median
2 s	84 ms	299 ms	54 ms	72	239	4
4 s	117 ms	444 ms	59 ms	145	356	71
8 s	199 ms	825 ms	66 ms	279	567	150

Table 6.3: Stop-and-copy downtime. The table lists average downtime and average number of disk sectors to be saved for several checkpointing interval lengths. The number of disk sector as well as the downtime increases with the interval length.

Comparison of the measurement data with the raw data obtained by Eicher [10] showed that in our setup, the device state saving time is much higher. This is the case for the copy-on-write approach as well as the stop-and-copy approach. Analysis revealed that using the `vmware` VGA adapter results in higher checkpoint saving time. Replacing it with the default VGA adapter resulted in a device state saving time below 1 ms. We have, however, not changed our measurement setup and continued all measurements with the `vmware` VGA adapter.

Moreover, the detailed downtime measurements show that the `ioctl` accounts for 18 ms of the downtime on average. Using the kernel's `ftrace` facility, we have profiled the code executed in the `ioctl`. This investigation has shown that obtaining and resetting the list of dirty pages (dirty bitmap) only accounts for a small part of the downtime. No detailed measurements were performed, but the `ftrace` logs showed that this takes around 3 ms. The main performance problem turned out to be an error in our implementation that resulted in a data structure (the `struct kvm_cow_page_stat`) being reset more often than neccessary. We have fixed this bug in our implementation and were able to half the total downtime for a two-second checkpointing interval. It was reduced from 26 ms (ten

Downtime component	Mean Time	StDev	Median
Disk	1 ms	5 ms	0 ms
Device states	7 ms	0 ms	7 ms
`ioctl`	18 ms	1 ms	18 ms

Table 6.4: CoW downtime composition. The table shows composition of downtime for a two-second checkpointing interval. The disk saving time is very low, but the time needed to save device states was higher than expected. We determined the cause to lie within the `vmware` graphics adapter. The time spent in the `ioctl` was also very high, which was due to an error in our code.

iterations, stdev = 5 ms, median = 26 ms) to 13 ms (six iterations, stdev = 5 ms, median = 13 ms).

As noted before, using the `vmware` graphics adapter results in a higher downtime of 7 ms instead of a value below 1 ms with the default graphics adapter. When we account for that increased downtime value by subtracting the resulting downtime overhead we can conclude that we are able to achieve average downtimes of 6 ms.

We have, however, continued our measurements with the implementation that contained the performance regression to preserve comparability.

6.5 Copy Performance

In comparison to the stop-and-copy approach, our implementation introduced additional overhead while copying pages, for example through synchronization and the memory proxy. To investigate the performance impact of this overhead, we analyzed the correlation of the number of pages to be copied with the time needed to copy these pages in a checkpoint for the copy-on-write approach and the stop-and-copy approach. Figure 6.4 illustrates this correlation by displaying the number of dirty pages on the y-axis and the time needed to copy these pages on the x-axis for a two-second checkpointing interval. For each approach, a linear fit was performed on the data and the resulting linear function was also plotted.

The first point to note about the plot is that the data for the stop-and-copy approach seems to be split in two classes: one of them has a higher slope and therefore a shorter copy time and the other one has a lower slope and therefore a longer copy time. Investigation of this phenomenon is out of scope for this thesis and therefore was not performed.

The fit function of the copy-on-write case (hereafter called $c(x)$) is shifted to the right in comparison to the fit function for the stop-and-copy approach (here-

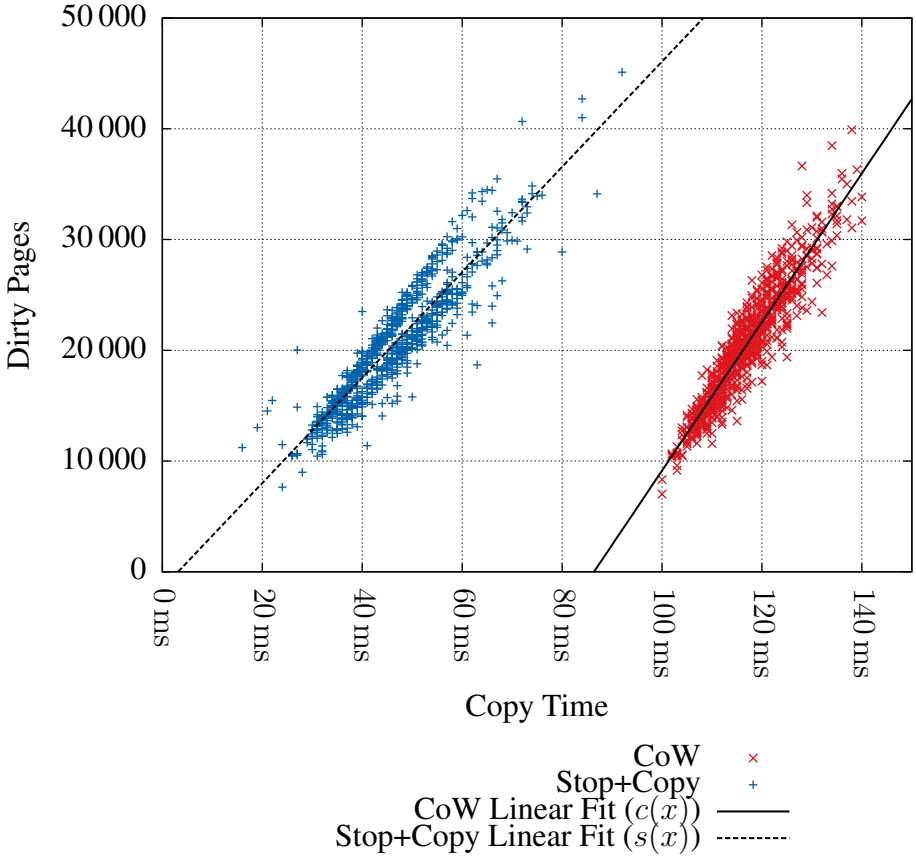

Figure 6.4: Copy Performance. Copy time is plotted over the number of pages to be copied for a two-second checkpointing interval. The CoW fit is steeper than the stop-and-copy fit. The CoW curve is offset to the right, therefore limiting the minimum checkpointing interval length. This was caused by `printk()` calls in our code.

after called $s(x)$). While $s(x)$ crosses the x-axis at about 3 ms, $c(x)$ does so at about 86 ms. This suggests some constant overhead of our implementation that occurs regardless of whether pages are to be copied or not. A possible explanation for part of this overhead could be the `ioctl` that must be called to enter the concurrent copy case.

This constant offset indicates that our implementation is not able to keep up with checkpointing intervals that are smaller than around 90 ms. For checkpointing intervals smaller than this size, saving the dirty pages will exceed the length of the checkpointing interval. Eichers self-regulating mechanism [10] will enlarge the checkpointing interval in this case and checkpoints will effectively be made at longer intervals.

Further analysis of this constant overhead with the kernels `ftrace` framework yielded that the offset is mainly caused by internals of our implementation: for debugging reasons, we had placed `printk()` calls in our code, that were not removed during benchmarking. `ftrace` revealed that each of this calls took around 5 ms in our setup. We have removed these calls and performed a linear fit on the new measurement data ($c_2(x)$). $c_2(x)$ crosses the x-axis at about 16 ms. However, to preserve comparability of our results, we have continued using the slower version of our implementation in the evauluaton.

Moreover, $c(x)$ is steeper than $s(x)$. This means that our implementation is able to copy pages faster than the stop-and-copy approach, even though our design yields additional overhead, for example during synchronization of the copy-on-write case with the concurrent copy case. Detailed investiation on why this is the case is out of scope of this thesis and therefore was not performed.

6.6 Page Handler Distribution

A goal of our design is to save dirty pages of the VM during its execution. The copy-on-write mechanism then ensures that these pages are saved consistently. Saving a page in the copy-on-write case is more expensive than in the concurrent copy case, because a page fault is caused and the VM's execution is interrupted while the page is being copied. The number of pages that can be handled fast, i.e. in the concurrent copy case, is therefore an interesting metric for our approach and evaluates its effectiveness.

Figure 6.5 shows the percentage of pages that could be handled by the concurrent copy case during a single iteration of a kernel build. The values were obtained by averaging over ten checkpoints to smoothen the plot.

Although the plot shows a significant number of spikes, we can see that it does not reach above the 95 % mark. Likewise, except for the last 100 checkpoints, the

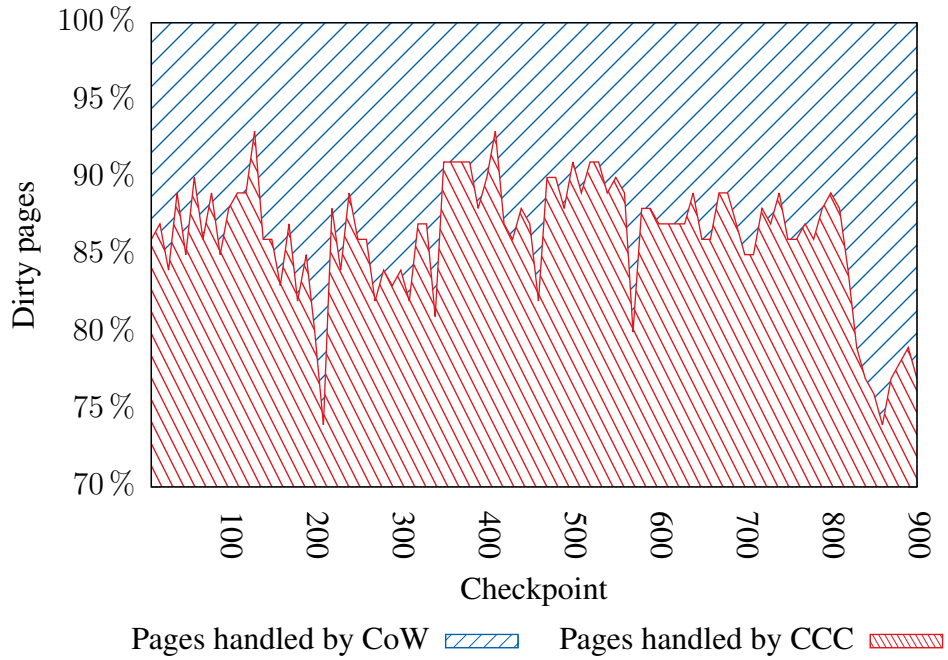

Figure 6.5: Page Handler Distribution. Percentage of pages copied by the concurrent copy case and the copy-on-write case during a single iteration of a kernel build. The plot was smoothened by averaging over ten checkpoints. A majority of the pages can be saved by the concurrent copy case.

plot only sometimes reaches below the 80 % mark. This suggests that a differnt phase, such as the linking phase, of the kernel build starts.

When analyzing the unsmoothened data, in all the checkpoints more than 50 % of the dirty pages were handled by the concurrent copy case. This demonstrates the effectiveness of our approach: in all cases, the majority of pages can safely be copied during VM execution, without involving the copy-on-write mechanism.

6.7 Conclusion

Our evaluation has shown that copy-on-write VM checkpointing offers improvements in comparison to stop-and-copy checkpointing. In Section 6.3, we have demonstrated that the same workload has a lower runtime with copy-on-write checkpointing than with stop-and-copy checkpointing.

Section 6.4 has shown that downtime is lower and more predictable in comparison to stop-and-copy checkpointing. We have also confirmed that implementing

copy-on-write checkpointing for the VM's memory is more beneficial in terms of downtime than copy-on-write checkpointing for the VM's disk. However, we were able to observe a high standard deviation of 5 ms in the disk saving component of the downtime during a kernel build. Future work can investigate if a copy-on-write scheme for the VM's disk image is able to offer further improvements.

However, the evaluation has also shown that our implementation does not fully leverage the benefits the copy-on-write mechanism would theoretically offer: two performance regressions in our code slowed down the checkpointing mechanism significantly. Measurements with a version that included fixes for these issues confirmed that the slowdown was indeed caused by the errors we suspected.

Nevertheless, none of the performance regressions was actually by design, all of them were caused by our implementation. We can therefore conclude that our design is effective with regards to performance.

In Section 6.5, we have identified that the copy-on-write checkpointing mechanism is also suitable for very small checkpointing intervals, such as these used in Remus [8] and is therefore suitable to extend Remus in this regard.

The main benefit of the copy-on-write approach results from being able to copy pages during VM execution. In Section 6.6, we have confirmed that our implementation is indeed able to handle more than half of the dirty pages without involving the copy-on-write mechanism. The variability in the percentage of pages copied by the concurrent copy case, however, hints at variable performance between the checkpoint intervals. The analysis of these spikes is left for future work. Possible improvements may arise from pre-copy rounds before the next checkpointing interval starts or from utilizing a mechanism similar to pre-paging in the copy-on-write case, i.e. to copy nearby dirty pages when the copy-on-write case is triggered for a page.

Chapter 7

Conclusion

SimuBoost aims to speed up full-system simulation. It achieves this by executing the simulation at different points in time in parallel. The state required to bootstrap simulations can be obtained by executing the workload through virtualization and then copying the state by means of checkpointing.

The stop-and-copy checkpointing approach can cause significant downtime of the VM. This limits the its applicability in the SimuBoost concept, because it impedes checkpointing workloads that utilize the network or that require interaction with a human user. As suggested in previous work [11, 22], the copy-on-write scheme is a potential candidate for solving the downtime problem. However, no detailed research on the characteristics of copy-on-write checkpointing existed this far.

Our design resulted in a mechanism that copies the VM's dirty pages during VM execution by iterating over the list of dirty pages. The consistency of the image is preserved by means of a copy-on-write mechanism. Our implementation then resulted in a patch for KVM and QEMU that implements the suggested mechanism. The implementation is based on Eicher's implementation [10] and therefore also stores its checkpointing data in the Simutrace storage server.

While evaluating our approach, we have determined that our implementation offers an almost constant downtime of 6 ms for a two-second checkpointing interval during a kernel build. We have determined the minimum checkpointing interval length that our implementation is able keep up with. It is at 16 ms, making it suitable for high-frequency checkpointing.

7.1 Future work

We have implemented copy-on-write for the VM's memory, but not for its disk image. While our implementation has resulted in a more predictable downtime in

comparison to the stop-and-copy approach, a standard deviation of 5 ms could be observed for the disk saving component in the downtime. Future work has to be done to investigate if a copy-on-write scheme for the VM's disk image is able to offer further improvements.

Further work may also investigate if pre-copy rounds before the next checkpointing intervall starts can offer further improvements, for example for different workloads.

Appendix A

Additional data

A.1 Downtime measurements for the first checkpoint

Interval	Downtime		
	mean	stdev	median
2 s	11 484 ms	186 ms	11 470 ms
4 s	11 462 ms	179 ms	11 474 ms
8 s	11 680 ms	242 ms	11 780 ms

Table A.1: Copy-on-write checkpointing: Average downtime for the first checkpoint.

Interval	Downtime		
	mean	stdev	median
2 s	17 296 ms	444 ms	17 252 ms
4 s	17 403 ms	326 ms	17 475 ms
8 s	17 372 ms	219 ms	17 334 ms

Table A.2: Stop-and-copy checkpointing: Average downtime for the first checkpoint.

A.2 Copy Performance For Improved Implementation

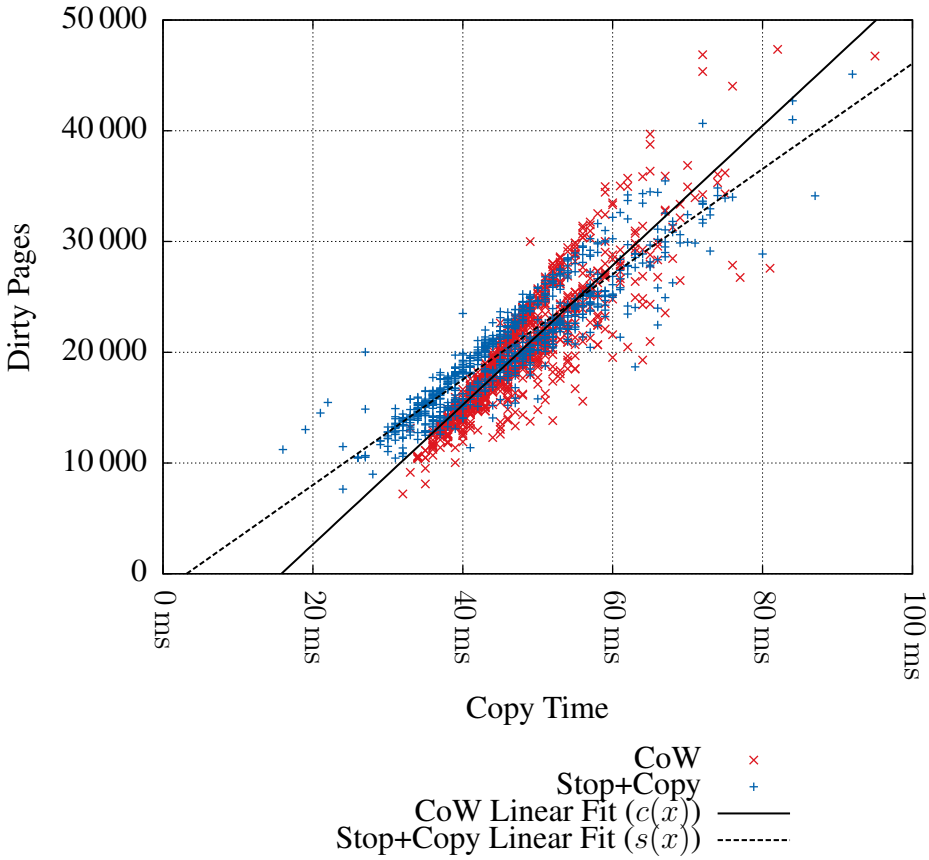

Figure A.1: Copy Performance for fixed implementation. Copy time is plotted over the number of pages to be copied for a two-second checkpointing interval. The CoW fit is steeper than the stop-and-copy fit. The offset of the CoW curve to the right is less significant.

Bibliography

[1] The definitive kvm (kernel-based virtual machine) api documentation. to be found in the Documentation/virtual/kvm/api.txt file in the linux kernel tree.

[2] AMD. Amd-v nested paging. White paper, Advanced Micro Devices Corp., 2008. http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf.

[3] Andrea Arcangeli. Rfc: userfault. Post on the linux kernel mailing list http://thread.gmane.org/gmane.linux.kernel.mm/119732, 2014.

[4] 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.itec.kit.edu/.

[5] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, pages 41–46, 2005. http://www.usenix.org/events/usenix05/tech/freenix/bellard.html.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011. http://doi.acm.org/10.1145/2024716.2024718.

[7] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In Amin Vahdat and David Wetherall, editors, *NSDI*. USENIX, 2005. http://www.usenix.org/events/nsdi05/tech/clark.html.

[8] Brendan Cully, Geoffrey Lefebvre, Dutch T. Meyer, Anoop Karollil, Michael J. Feeley, Norman C. Hutchinson, , and Andrew Warfield. Remus: High availability via asynchronous virtual machine replication. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2008.

[9] Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, and Yeh-Ching Chung. Pqemu: A parallel system emulator based on qemu. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 276–283, Dec 2011.

[10] Bastian Eicher. Virtual machine checkpoint storage and distribution for simuboost. Master thesis, System Architecture Group, Karlsruhe Institute of Technology (KIT), Germany, September 2015. `http://os.itec.kit.edu/`.

[11] Balazs Gerofi, Zoltan Vass, and Yutaka Ishikawa. Utilizing memory content similarity for improving the performance of replicated virtual machines. In *UCC*, pages 73–80. IEEE Computer Society, 2011. `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6123165`.

[12] VMWare Inc. Performance evaluation of intel ept hardware assist. White paper, VMWare Inc., 2009. `https://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf`.

[13] Intel. *64 and IA-32 Architectures Software Developer's Manual*, volume 3C: System Programming Guide, Part 3.

[14] Intel. Page modification logging for virtual machine monitor. White paper, January 2015. `http://www.intel.de/content/dam/www/public/us/en/documents/white-papers/page-modification-logging-vmm-white-paper.pdf`.

[15] S. Kikuchi and Y. Matsumoto. Impact of live migration on multi-tier application performance in clouds. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 261–268, June 2012.

[16] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the USENIX Annual Technical Conference (USENIX'05)*, April 2005. `http://www.eecs.umich.edu/~pmchen/papers/reverseDebug.pdf`.

[17] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, Feb 2002.

[18] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 25–25, Berkeley, CA, USA, 2005. USENIX Association. `http://dl.acm.org/citation.cfm?id=1247360.1247385`.

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

[20] 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 2013.

[21] Vadim Rozenfeld. Kvm as a microsoft-compatible hypervisor. Talk at the KVM Forum, slides at `http://www.linux-kvm.org/images/0/0a/2012-forum-kvm_hyperv.pdf`, November 2012.

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

[23] Paula Ta-Shma, Guy Laden, Muli Ben-Yehuda, and Michael Factor. Virtual machine time travel using continuous data protection and checkpointing. *SIGOPS Oper. Syst. Rev.*, 42(1):127–134, January 2008. `http://doi.acm.org/10.1145/1341312.1341341`.

[24] Yoshiaki Tamura, Koji Sato, Seiji Kihara, and Satoshi Moriai. Kemari: Virtual machine synchronization for fault tolerance. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '08. USENIX Association, 2008.