Karlsruher Institut für Technologie

Exploring Pre-scan, Parallel Copy, and Large Pages for Continuous Checkpointing

Masterarbeit
von

Janis Schoetterl-Glausch

an der Fakultät für Informatik

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

Bearbeitungszeit: 1. Juni 2018 – 30. November 2018

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

Karlsruhe, den 30. November 2018

Abstract

SimuBoost [35] is a concept to speed up full system simulation, which is hampered by its low execution speed. To achieve this, SimuBoost relies on lightweight, continuous virtual machine checkpointing. SimuBoost's checkpointing implementation is incremental [11] and makes use of copy-on-write and concurrent-copy [15] to avoid the high downtime that would be the result of performing the copy while the virtual machine is stopped. Incremental checkpointing only copies those pages written to since the last checkpoint. It therefore requires a dirty logging mechanism. SimuBoost's preferred dirty logging mechanism scans the page tables mapping the guest physical address space to the host physical address space. This occurs during the downtime, increasing it.

We explore if *pre-scan* can decrease the downtime by moving part of the scan outside the downtime We find that, while pre-scan succeeds in reducing the downtime, it can negatively impact the performance of the virtual machine, mostly for interval lengths under 500 ms. At 500 ms, pre-scan causes a sub 1% increase in performance.

The page faults caused by copy-on-write degrade the performance of the virtual machine. We investigate if the number of such page faults can be decreased. We attempt this by parallelizing concurrent-copy. The rationale is to speed up concurrent-copy, so it can save more pages before they can incur a copy-on-write page fault. While our implementation can roughly half the number of page faults, it does not improve performance, instead performance is reduced for intervals shorter than 500 ms. For 500 ms, parallel copy improves performance by less than 1%.

Generally, the use of large pages improves the performance of virtual machines [13, 28, 31]. SimuBoost does not make use of large pages during checkpointing. To do so would increase the amount of memory to capture, the additional overhead reduces the benefit of large pages. A naive, experimental implementation of checkpointing with large pages shows a benefit only for interval lengths of 2 s and higher. The benefit is approximately 5%. For smaller intervals performance is decreased.

Deutsche Zusammenfassung

Diese Arbeit untersucht mehrere Techniken, um SimuBoost [35] zu verbessern. SimuBoost ist ein Verfahren zur Beschleunigung der vollumfänglichen Simulation eines Rechensystems. Dies ist nötig, da der Nutzen einer Simulation, gegeben durch den hohen Detailgrad der Beobachtung, die sie ermöglicht, durch die Langsamkeit, mit der sie erfolgt eingeschränkt wird. SimuBoost beschleunigt die Simulation durch Parallelisierung. Dies erfolgt, indem SimuBoost den Rechner als virtuelle Maschine ausführt und dabei periodisch Abbilder der virtuellen Maschine erstellt. Diese Abbilder dienen als Startpunkte der parallelen Simulation.

Das Erstellen eines solchen Abbildes führt zu einer Unterbrechung der Ausführung der virtuellen Maschine. Um diese zu verkürzen, erstellt SimuBoost Abbilder des Speichers der virtuellen Maschine, der eine erhebliche Datenmenge darstellt, inkrementell [11]. Das heißt, SimuBoost sichert nur die Speicherseiten, die seit dem letzten Abbild von der virtuellen Maschine beschrieben wurde. Dies setzt einen Mechanismus voraus, der verfolgt, welche Seiten des Speichers beschrieben wurden. Um die Unterbrechung der virtuellen Maschine weiter zu verkürzen, kopiert SimuBoost die zu sichernden Seiten nach der Unterbrechung, gleichzeitig zur Ausführung der virtuellen Maschine [15]. SimuBoost schützt die zu sichernden Seiten vor Schreibzugriff, um zu verhindern, dass die virtuelle Maschine sie modifiziert bevor sie kopiert wurden. Wenn ein Schreibzugriff auf eine geschützte Seite erfolgt, wird die virtuelle Maschinen kurzzeitig unterbrochen, die Seite wird gespeichert und danach die Ausführung fortgesetzt. Auch diese Unterbrechungen stellen einen unerwünschten Einfluss auf die Ausführung der virtuellen Maschine dar.

SimuBoost benutzt 4 KiB große Speicherseiten, die Verwendung von größeren Seiten verbessert jedoch, wenn keine Abbilder erstellt werden, die Ausführungsgeschwindigkeit einer virtuellen Maschine [13, 28, 31].

Diese Arbeit untersucht, ob die Unterbrechung zur Erstellung eines Abbildes verkürzt werden kann, indem Teil des Aufwandes, um herauszufinden, welche Seiten seit dem letzten Abbild beschrieben wurden, aus der Unterbrechung verschoben wird. Desweiteren wird untersucht, ob paralleles Kopieren

die Anzahl von Unterbrechungen wegen Schreibschutzverletzungen verringern kann. Außerdem prüfen wir, ob die Verwendung von großen Seiten für SimuBoost Sinn ergibt.

Contents

List of Figures

Chapter 1

Introduction

This thesis explores multiple approaches to improve continuous virtual machine (VM) checkpointing. It does so in the larger context of SimuBoost [35], which utilizes lightweight continuous checkpointing to accelerate functional full system simulation.

Taking a checkpoint requires a brief interruption of the VM's execution, resulting in an undesirable *downtime*. To keep the downtime to a minimum, SimuBoost uses incremental copy-on-write (CoW) checkpointing. Incremental checkpointing captures only those pages that were written to by the virtual machine since the last checkpoint. A dirty logging mechanism is required to know which pages to capture. One possibility is to write-protect the pages to dirty log. A write access to a write-protected page triggers a page fault, so that SimuBoost can mark the page dirty. Another possibility is to scan the address translations mapping the guest physical address space to the host physical address space for leaf page table entries that have their dirty bit set. This occurs during the downtime, increasing it compared to dirty logging using write-protection. However, because it does not cause page faults, scanning ultimately leads to improved performance. Additionally, it allows for accesses to be logged in an equivalent manner. SimuBoost uses this to reduce the amount of data that has to be loaded during simulation. Because of these two reasons page table scanning is the preferred dirty logging mechanism in SimuBoost.

We devise *pre-scan*, in order to create a dirty logging mechanism that combines the low downtime of write-protection with the performance of scanning the page tables. The idea is to perform a scan before the checkpoint is taken, so that the work done by a second scan during the downtime is reduced. We observe that pre-scan is able to cut down the increase caused by scanning to less than half. However, for intervals shorter than 500 ms, performance is

negatively affected. The overhead can be in the order of 10%. For 500 ms, pre-scan shows a performance improvement of less than 1%.

SimuBoost uses copy-on-write to keep the downtime small. Instead of copying the pages modified since the last checkpoint during the downtime, SimuBoost write-protects them and resumes the virtual machine. It then copies the pages concurrent to the execution of the VM. The write-protection serves to preserve the integrity of the checkpoint. If the VM attempts to write to such a page before it has been copied, a page fault occurs instead, allowing SimuBoost to capture the page. Such page faults negatively affect the performance of the virtual machine. At 50 ms checkpointing interval length, they can reduce performance by more than 10%.

In order to reduce this impact, we propose to perform concurrent-copy with multiple threads, so that the bandwidth with which pages are copied is maximized. This would give the VM less time to incur a CoW page fault. We find that parallel copy can reduce the number of CoW page faults by roughly half, however it does not significantly improve performance, instead it decreases it for short interval lengths. The overhead is as high as 8%, for an interval length of 500 ms, we observe an average 0.7% performance improvement.

Virtualizing the memory of a VM requires guest physical pages to be translated to host physical pages. This process can use large pages, which improves performance. SimuBoost, however, uses small pages when checkpointing. As a result, compared to the use of large pages, which are enabled by default, performance drops. To use large pages in conjunction with checkpointing would increase the amount of memory that would need to be captured, since dirty logging would also have to work on the coarser granularity of large pages. Experimentally, we incorporate large pages into checkpointing in order to gain an understanding of the implications of doing so. We find that large pages can improve performance for interval lengths of 2 s and higher. The benefit for 2 s is approximately 5%.

This thesis is structured as follows: Chapter 2 provides background on full system simulation, virtualization and SimuBoost. Chapter 3 analyses the impact SimuBoost has on the virtual machine it checkpoints and identifies aspects of checkpointing that could benefit from optimization. Chapter 4 describes the design and implementation of those optimizations—pre-scan, parallel copy, and the use of large pages. Chapter 5 evaluates if they succeed in improving SimuBoost. Chapter 6 provides a conclusion and describes possible future work.

Chapter 2

Background

Full system simulation is a powerful tool to gather detailed instruction-level run time information that suffers from poor performance. SimuBoost is a project to accelerate functional full system simulation by parallelization. To achieve this, it uses virtualization and virtual machine checkpointing.

This chapter provides background on the relevant topics. Section 2.1 describes full system simulation. Section 2.2 explains virtualization, how it is implemented on modern x86 architectures and reviews virtual machine checkpointing techniques. Finally, Section 2.3 illustrates how SimuBoost combines full system simulation and virtualization, as well as how SimuBoost is implemented.

2.1 Full System Simulation

Full system simulation (FSS) allows simulation of a computer system, alternatively called a machine [18, 41]. It encompasses all parts of the system, including the processor, memory, and devices like input peripherals and networking [18, 51]. Full system simulators are useful tools in a range of cases:

- They allow the development of software without access to hardware, for example when adding support for a new processor to an operating system [18, 19, 27].

- They can be used to test availability by injecting faults and observing the reaction of the system [18, 27]. The versatility of full system simulation allows a broad range of possible faults to be injected because any part of the simulated system can be manipulated. Determinism allows for simple regression testing [18].

- Since FSS encompasses the whole system, it can aid in debugging. The user can not only debug a program but also its interaction with the operating system and the hardware [18, 27]. Bugs often affect security. Bochspwn [24] has identified security vulnerabilities in multiple operating systems by instrumenting the Bochs emulator to analyze memory accesses. Xenpwn [48] has similarly found vulnerabilities in the Xen hypervisor.

- Simulations can replace real hardware no longer in production [18].

- FSS can be used to better understand the behavior of software. SimOS [37] has been used to characterize an operating system and predict its performance on possible future hardware. Additionally, it has been used to improve the output of a compiler [36]

As with all simulations, FSS must rely on a model [18]. The model determines the level of abstraction provided by the simulation [37] and therefore how closely it reproduces the behavior of the real system. *Functional* simulation emulates instructions as defined by the instruction set definition but does not take timing into account [27, 51]. *Cycle Accurate* simulators emulate the internals of a processor, closely approximating a real machine [51].

However, a big drawback of FSS is its slow execution. Its speed depends on the level of abstraction of the model [37] and the implementation [10, 49]. Rittinghaus et al. report slowdowns by a factor of 20–1100 depending on the simulator and instrumentation level [35]. Bochspwn gives a slowdown factor of 32–50 for an instrumented run, resulting in boot times of up to ≈20 min and a completion of the testing workload in less than 24 h [24].

The considerable slowdown of FSS limits its use because the simulation must finish in a reasonable time frame [27]. The author of Bochspwn names performance as a limitation [23]. Additionally, if the slowdown is too high, interactions with the outside environment, e.g. a user, become impractical and unrepresentative of a real system.

2.2 Virtualization

According to Popek and Goldberg [32], a virtual machine (VM) is a *duplicate* of a real machine that is *efficient* and enforces *isolation*. The duplicate is functionally equivalent to the real machine, however, this *guest* machine may differ in timing characteristics and available resources. Efficiency is achieved by running the majority of the VM's instructions directly on the real processor [32]. This, unlike FSS, restricts guests to compatible instruction sets

but also implies much less overhead. All benchmarks evaluated in [35] show an increase in execution time of less than 15% compared to native execution. Directly running instructions also means that virtualization does not allow instrumentation as powerful as that possible with full system simulation.

The virtual machine is an environment created by a virtual machine monitor (VMM) [41, p. 369] running on the *host* machine. The VMM, although it does not appear so to the VM, retains complete control over all resources and enforces isolation, that is, it ensures that the VM can only access resources allocated to it [32]. The VMM must provide the VM with resources including a virtual processor and memory.

A conventional processor supports two different modes of execution—supervisor mode and user mode. In user mode, only part of the instruction set is available. Popek and Goldberg [32] showed that such a processor can be efficiently virtualized by running the VMM in supervisor mode and VMs in user mode, but only if all *sensitive* instructions are *privileged*. That is, if all instructions that, if executed by the VM, would violate isolation, instead cause an exit to supervisor mode, where the VMM can execute them on behalf of the VM while maintaining isolation. Additionally, the processor needs to support a way to determine the VM's view of memory, for example, via addressing relative to a relocation register [32] or page tables [41, p. 397].

2.2.1 Hardware-Assisted Virtualization

The x86 instruction set is not efficiently virtualizable according to the requirements defined by Popek and Goldberg [32] because it includes instructions that are sensitive but not privileged [41, p. 391]. An example of such an istruction is Pop Stack into Flags Register (POPF). When executed in user mode, it does not affect the interrupt-enable flag. The behavior of POPF therefore depends on the mode of the real processor, not on that of the virtual processor. This violates the isolation of the virtual machine. There are techniques to deal with sensitive non-privileged instructions[1], but they increase code complexity and negatively impact performance [41, p. 436].

To alleviate these problems and improve the virtualizability of x86, vendors have released extensions to their instruction sets. Intel introduced Virtual Machine Extensions (VMX) [22, ch. 23], AMD added AMD Virtualization ™ [9, ch. 15]. Because it is more relevant for this thesis, we give an overview of VMX, AMD's extensions work similarly.

VMX adds two kinds of processor operation: VMX root operation and VMX non-root operation. VMX root operation is intented for the virtual

[1] Code Patching [41, p. 391,212]

machine monitor and is mostly the same as processor operation without VMX, except for the availability of VMX instructions. In VMX non-root operation, the behavior of some instructions, like POPF, is changed. For example, they may cause a VM exit, that is, a transition from VMX non-root operation to VMX root operation so that the VMM can enforce isolation [22, ch. 23.3]. VMX root and non-root operations therefore function as supervisor and user mode as understood by Popek and Goldberg [32].

VMX also includes provisions to make implementing a VMM easier and improve the performance of virtualization. One such provision aids in virtualizing memory and virtual to physical address translation.

The x86 protected mode has support for paging to address memory [22, ch. 4]. When paging is enabled, hierarchical paging structures are used to translate linear addresses to physical addresses. In this thesis, all paging structures, regardless of level, are referred to as page tables, their entries as page table entries.

Multiple paging modes exist [22, ch. 4.1.1], all work according to the same principle [22, ch. 4.2]: The CR3 register points to a page table. Part of the linear address is used to index this table, resulting in an entry containing a physical address. This physical address either points to another page table, in which case the process repeats, or, on the last level, is used together with the remaining part of the linear address to derive the result of the address translation. Figure 2.1 portrays the process.

To improve performance, the processor may cache address translations in Translation Lookaside Buffers (TLBs). It may also cache information about page tables [22, ch. 4.10]. System software must appropriately invalidate the caches so changes to the page tables take effect [22, ch. 4.10].

In addition to the physical frame number, page table entries contain bits controlling the page's access rights, as well as bits to track accesses and writes to the page. Whenever the processor makes a memory access, it sets the accessed bit in all page table entries used in the translation. If the access is a write, it also sets the dirty bit in the page table entry lowest in the hierarchy.

The virtual machine monitor must virtualize paging, that is, the guest OS must be able to configure guest paging, with the VMM ultimately in control of host memory allocation (s. Figure 2.2). This can be done without VMX support via the use of shadow page tables [41, p. 399]: Because the access to CR3 is privileged, the VMM retains control over it. When the guest tries to write the address of a page table to it, the VMM instead writes the address of the corresponding shadow page table. This shadow page table contains the translation of guest virtual pages directly to host physical pages. The

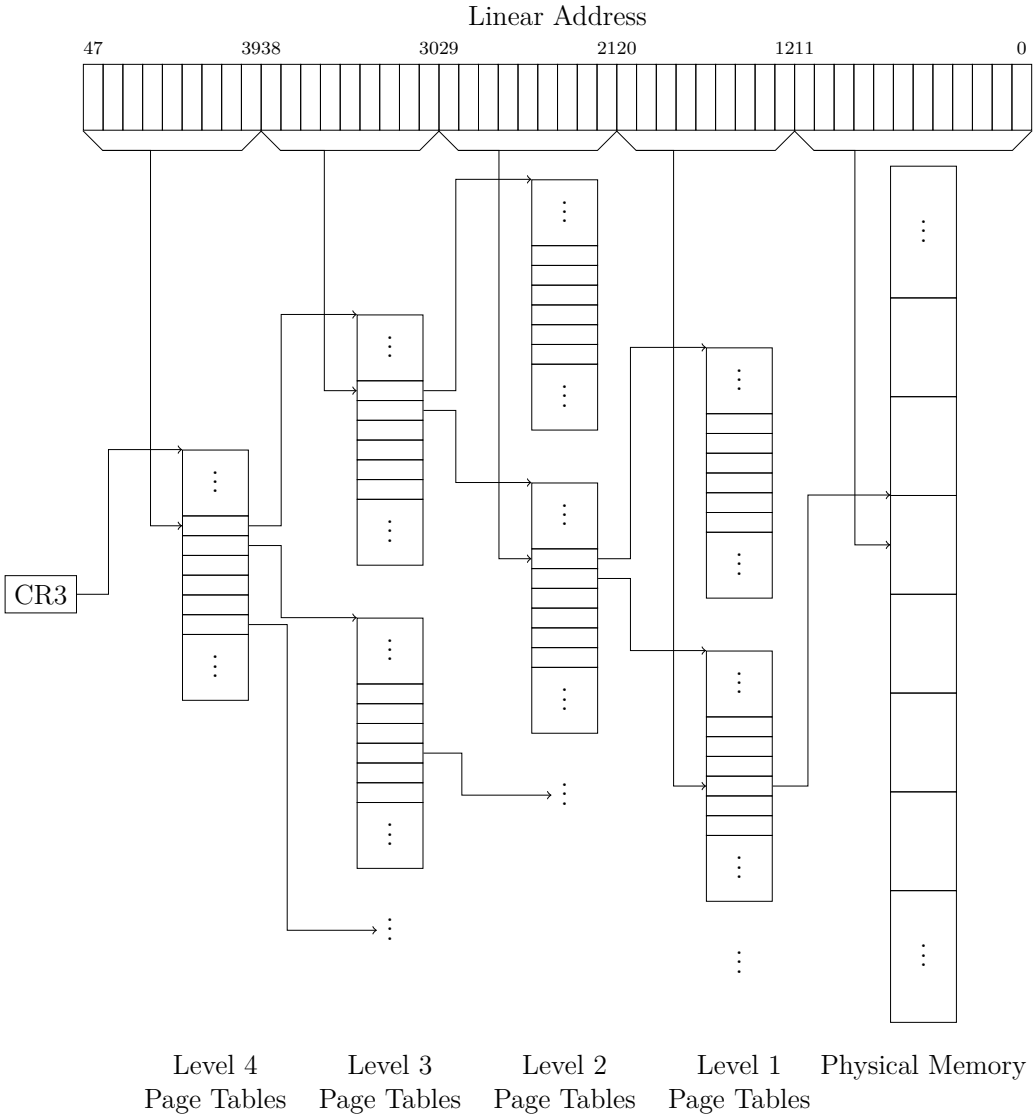


Figure 2.1: IA-32e paging. The linear address is split into offsets used to index hieerarchical page tables. The lowest offset and the lowest page table entry give the physical address.

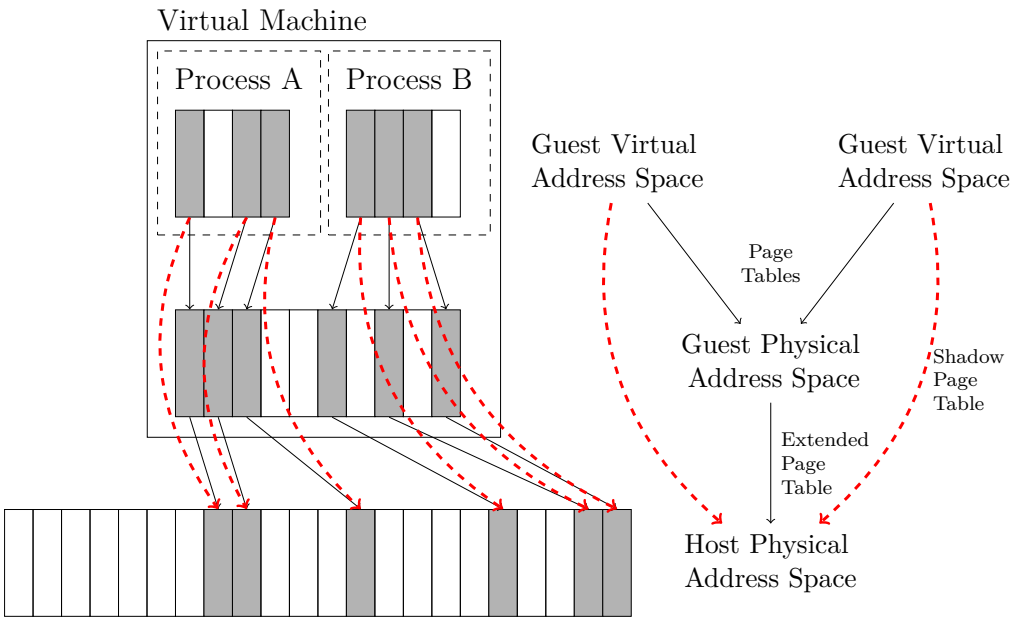


Figure 2.2: Virtualization of paging. The guest OS manages the page tables of processes A and B. The VMM controls the allocation of guest physical memory to host physical memory. If the processor does not support a second level paging mechanism like EPT, the VMM uses shadow paging, replacing the page tables of processes A and B with appropriate shadow page tables.

VMM must take care to keep the shadow page table up to date when the guest OS changes the corresponding page table.

Intel added the Extended Page Table Mechanism (EPT) to VMX [22, ch. 28.2], which removes the need for shadow paging and improves performance [13, 14, 33]. When EPT is enabled, the guest is able to configure paging without VMM intervention. However, the processor does not use guest physical addresses, that is, the (intermediate) results of guest address translations, to accesses memory. Instead, it uses EPT page tables, which work in the same manner as regular page tables, to translate the guest physical to host physical addresses.

As with normal paging, the processor might cache (partial) guest virtual to host physical translations, which must be properly invalidated when necessary [22, ch. 28.3].

EPT may support accessed and dirty bits in the same way normal paging does [22, ch. 28.2.4].

EPT supports 4 KiB, 2 MiB and 1GiB physical pages [22, ch. 28.2.2]. The size of the page is determined by the level of the last page table. The lowest possible level maps a 4 KiB page, that above a 2 MiB page, etc. A bit in the page table entry specifies whether that entry maps a page or points to a lower page table. Using large pages can improve performance by reducing the number of TLB misses, and thereby the number of cycles spend on TLB miss handling [13, 28, 31].

VMX may also support Page-Modification Logging (PML) [22, ch. 28.2.5]. When PML is enabled and the processor sets a dirty bit, it also logs the guest physical address to a region specified by the VMM.

2.2.2 Virtual Machine Checkpointing

A VM checkpoint is an image of the complete state of a VM taken at an instantaneous point in time. As such, it encompasses the state of the processor, memory and devices [42].

When a VM checkpoint is transferred to another physical host, the process is called VM migration. When a running VM is migrated, the operation is designated live migration. Migration and checkpointing in general are useful tools [16, 17]. Migration can be used to balance load. Both can be used for fault tolerance, either by maintaining a fall-back VM on another host [17, 43] or by keeping a checkpoint on the same host, from which execution can be resumed if a fault is detected [47].

Taking a checkpoint affects the execution of the VM. The *downtime*, that is the time the execution of the VM is suspended [42], represents an

interruption of service and must therefore be minimized [16]. Checkpointing can also slow down the execution of the VM.

There are different techniques the virtual machine monitor can employ to create checkpoints. The simplest, **stop-and-copy**, stops the VM and copies the complete state [16, 42]. This has the disadvantage that the downtime is proportional to the amount of of data that needs to be copied. With memory sizes in the order of gigabytes, this can lead to unacceptable interruptions of service [16].

Pre-copy drastically reduces the downtime by copying the majority of data beforehand [16, 17, 42]. While the VM is still running, pre-copy copies the VM's memory in rounds. In each round, all memory pages not yet copied or overwritten by the VM are captured. Eventually, the pre-copy phase ends, the VM is suspended and the pages overwritten since the last round are captured, as well as the processor and device states. Again, the downtime is proportional to the data captured during it. Because only modified pages need to be captured, it can be substantially less than that of stop-and-copy.

Since the VMM needs to know which pages were modified, a dirty logging mechanism is required. Such a mechanism might adversely affect the VM while it is running. Because the same page might be captured numerous times, the total time to create the checkpoint is larger than that of stop-and-copy.

Like pre-copy, **post-copy** moves the majority of work outside the downtime, however, it moves it behind the downtime [17, 42]. This is done by suspending the VM, saving the state of the processor and write-protecting the memory of the VM. Then the execution of the VM resumes. When the VM attempts to modify a page, an exit to the VMM occurs. The VMM captures the page, removes the write-protection so further accesses do not cause an exit, and then continues the execution of the VM. These *copy-on-write* (CoW) VM exits slow the execution of the virtual machine [46]. To decrease the number of CoW exits, the VMM can concurrently copy those pages not (yet) modified by the VM. After having copied a page, it also removes the write-protection, preventing a CoW exit.

When creating multiple checkpoints of a VM after another, the amount of data required to be saved per checkpoint can be lowered by reusing data saved by other checkpoints. This **incremental** checkpointing can be done in two ways:

One approach is to save the complete state of the VM for the first checkpoint and then save only the data changed since the last checkpoint for every subsequent checkpoint [46]. As such, this approach is equivalent to repeating the last step of pre-copy [17]. The restoration of the first checkpoint

is straight-forward, every other checkpoint can be restored by restoring its predecessor and applying the modifications that took place since it.

Another approach to implement incremental checkpointing is to capture—via copy-on-write—the data that will be changed leading up to the next checkpoint [46]. The last checkpoint encompasses the complete state of the VM. A checkpoint can be restored by restoring its succeeding checkpoint, then reverting the modifications via the saved data. Long Wang et al. [47] use this approach to quickly revert the VM back to an earlier state upon detection of an error in the execution of the VM.

Variations on these approaches exist. The first approach can be combined with copy-on-write and concurrent-copy to copy the pages modified before the checkpoint after the downtime, decreasing it [11, 15]. The second approach can be varied by predicting which pages are going to be modified after the checkpoint and copying them during the downtime [46,47]. This reduces the number of CoW page faults and therefore their impact on the execution of the VM.

2.3 SimuBoost

SimuBoost [35] is a project to speed up functional full system simulation by parallelizing it.

An instruction can only be simulated if the state it operates on is known. SimuBoost first runs the workload in a hardware-assisted virtual machine and periodically checkpoints it, namely, it saves the complete state of the VM. Because the execution of the virtual machine is fast, these checkpoints represent fast-forwarded states from which SimuBoost derives starting states for traditional functional simulations. Starting a simulation from a checkpoint does not require waiting for the simulation of a previous interval to have completed, SimuBoost exploits this parallelizability to achieve an overall speed up, as shown in Figure 2.3. As a result, SimuBoost unites the advantages of full system simulation—detailed observability—with those of virtualization—high speed.

The execution of the virtual machine is not deterministic [35]. This is because eventually the non-determinism of the real machine hosting the VM carries through into it. For example, the data of a virtual hard drive must be stored on real storage device. When the virtual processor reads from the virtual drive, the real processor must access the real device. After an unknown duration the device responds and the result of the read is inserted into the VM.

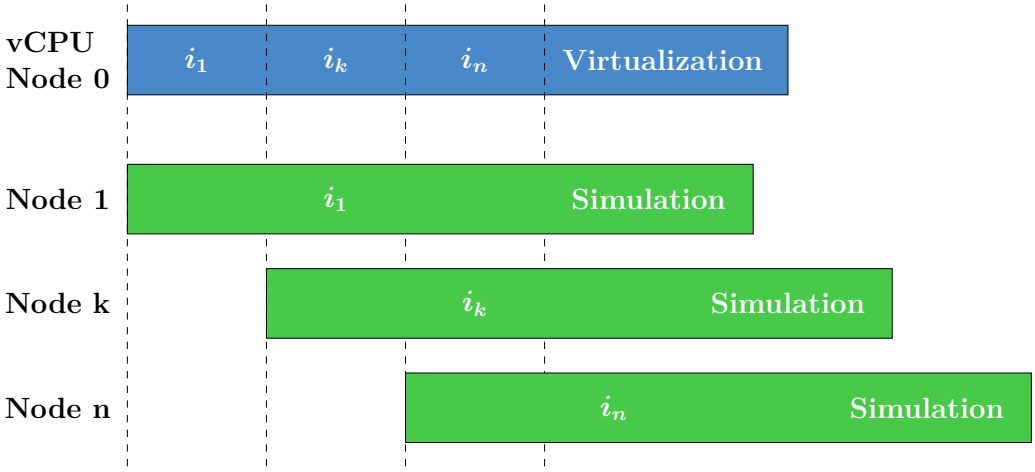


Figure 2.3: Functional simulation accelerated using SimuBoost. The workload runs in a VM. Periodic checkpoints serve as starting states for parallel simulations. Because the simulation overleaps, the last simulation ends earlier than a sequential simulation. [40]

This poses a problem for SimuBoost, because the final state of a simulated interval must be equal to the starting state of the next [35]. Therefore, the simulated execution must behave the same way as the virtualized one. To solve this problem, SimuBoost records non-deterministic events and replays them at the correct time in the simulation.

The downtimes caused by checkpointing influence the behavior of the workload. If an event from a source outside the VM causes a reaction and a downtime occurs between the event and the reaction, it will appear delayed to an outside observer. In this way, the response to a network request may have higher response time and variance. High downtimes can also deteriorate user interactivity, the response to input should be less than 100-200ms [29]. Downtimes should therefore be minimal.

SimuBoost also causes a slowdown of the virtual machine's execution. In part, this is due to the recording of non-deterministic events [35]. Depending on the technique, checkpointing also contributes to the slowdown. Like the downtime, the slowdown should be as small as possible to minimize its influence.

2.3.1 SimuBoost Implementation

The SimuBoost imlementation is based on QEMU [3] and KVM [1]. QEMU is a fast, portable full system emulator with support for a number of vir-

tual devices [12]. Kernel-based Virtual Machine (KVM) is a subsystem in the Linux kernel to expose the virtualization capabilities of the processor to userspace [25]. Both are free software. Userspace can interact with KVM via `ioctl`s to the `/dev/kvm` device. Using these `ioctl`s, a userspace application can create a VM, assign part of its memory to the VM, create virtual processors, and run them [20]. QEMU supports KVM, together they form a VMM with QEMU emulating devices and KVM running virtual processors.

SimuBoost uses Simutrace [34], in form of the Simustore server, as the storage back end. Simutrace provides *streams* of fixed size entries as an abstraction. For each checkpoint SimuBoost utilizes multiple streams, for example it uses one stream for all modified pages, one for device states, etc. A stream consists of segments. After the client has (partially) filled a segment, it submits it to the Simustore server to be appended to the stream. While a stream is an ordered sequence, SimuBoost adds the pages saved during checkpointing in an arbitrary order, with each entry containing the address of the page.

Only one segment per stream can be filled at the same time, because Simustore implicitly closes the previous segment when appending.

To conserve space, Simustore compresses stream data, captured pages are additionally deduplicated.

Figure 2.4 visualizes the components of SimuBoost.

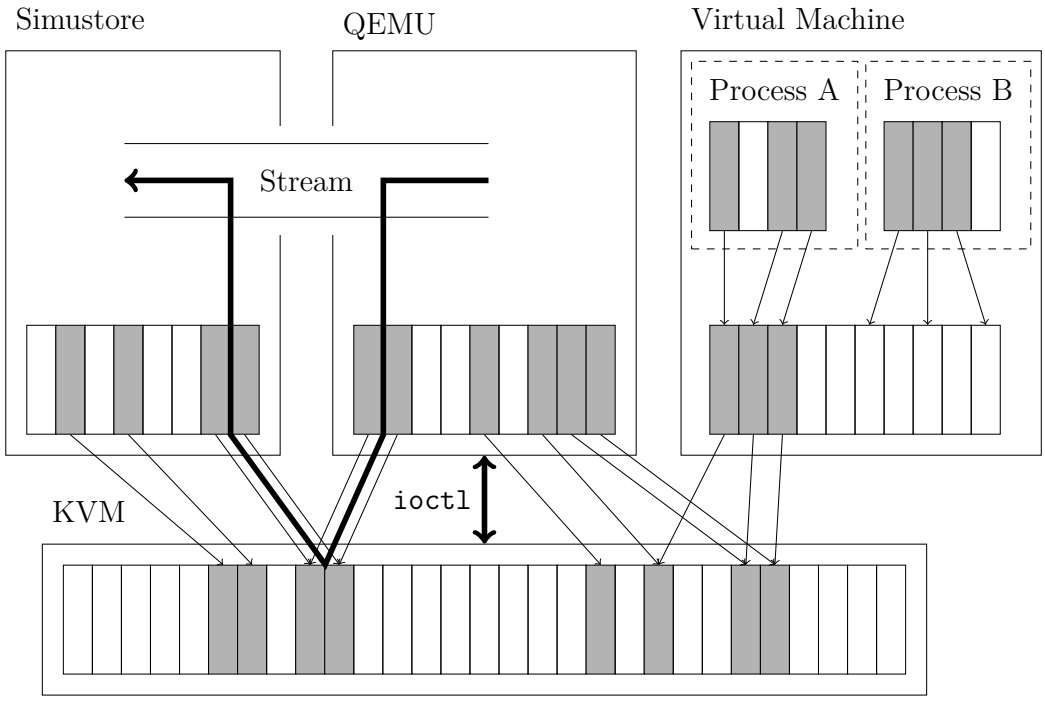


Figure 2.4: Overview of SimuBoost when checkpointing. The horizontal bars represent physical or virtual memory, devided into pages. KVM and QEMU interact via `ioctl`s to create the VM. When checkpointing, QEMU writes the checkpoint data to streams provided by Simustore. When QEMU and Simustore run on the same machine, they can communicate via shared memory, represented by the pages mapped to the same frames. When prompted, KVM saves the VM's memory to an area provided by QEMU—if possible the memory shared with Simustore.

Chapter 3

Analysis

The following chapter analyses aspects of checkpointing in SimuBoost. It shows how checkpointing negatively affects the execution of the virtual machine. These ill effects present opportunities for optimizations presented further on in the thesis.

Section 3.1 compares scanning and write-protection as dirty logging mechanisms. It shows that scanning increases the downtime and would thus benefit from a reduction thereof. Section 3.2 portrays how copy-on-write page faults are distributed and argues that their number could be decreased by increasing the speed of concurrent-copy. Section 3.3 shows that the use of large pages improves performance when no checkpointing takes place but finds that checkpointing with large pages would have to capture substantially more memory. For the benchmarking methodology see chapter 5.

3.1 Impact of Dirty Logging

SimuBoost's preferred checkpointing technique is incremental [11] copy-on-write (CoW) [15] checkpointing. We only consider incremental CoW checkpointing in this thesis. This technique captures only those pages modified by the VM since the last checkpoint. During the downtime the pages to be saved are write-protected, in order to enable copy-on-write. After the downtime, SimuBoost copies the majority of pages concurrent to the execution of the virtual machine. Those pages modified by the VM *before* concurrent-copy could save them are copied via CoW. Since little data is saved during the downtime, it is generally below 20 ms.

SimuBoost must know which pages were changed since the last checkpoint and therefore requires a *dirty logging mechanism*. KVM provides dirty

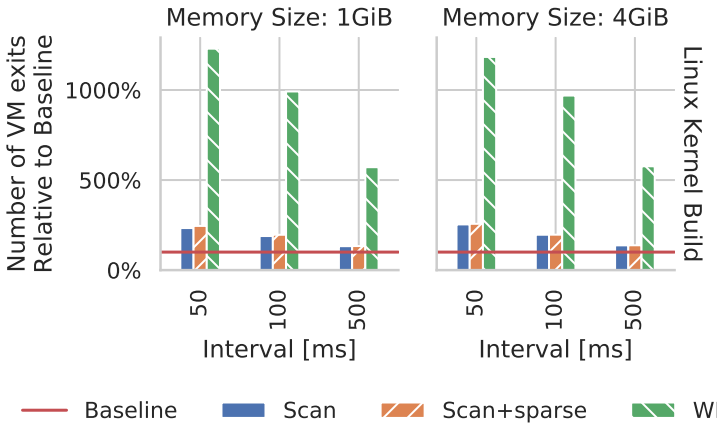


Figure 3.1: Total number of VM exits over the execution of the VM, relative to the baseline. When write-protection is used, the number of VM exits increases substantially, compared to the baseline it is between 5 and 13 times as high. The higher the checkpointing frequency, the higher the the number of VM exits.

logging mechanisms, in order to support live migration. SimuBoost builds on this for its incremental checkpointing implementation.

A mechanism that KVM always provides uses the ability of paging to write-protect pages. It sets the write-protect bit in page table entries mapping pages to be dirty logged. When a subsequent write to a protected page triggers a page fault, KVM marks the page dirty in its dirty logging structures, then makes the page writable.

In the context of dirty logging mechanisms, we refer to this mechanism as *write-protection* or WP. When SimuBoost uses WP for checkpointing, it write-protects the pages to be logged during each downtime.

Figure 3.1 shows that the number of VM exits increase by a factor of 5 to 12 compared to the baseline, where no checkpointing is performed. If the checkpointing frequency is high, the write-protection is re-established often, leading to more exits for short intervals.

Write-protection works with both shadow-paging and EPT.

If the hardware supports it, KVM uses Intel Page-Modification Logging (PML) as dirty logging mechanism. PML lowers the number of VM exits and improves performance [40]. Since the machine this analysis is performed on does not support it, we disregard PML.

SimuBoost additionally supports *scanning* as a dirty logging mechanism. When using scanning, SimuBoost walks the EPT page tables during the downtime. It identifies dirty pages via the dirty bits in the last level page ta-

bles. If it finds a dirty page table entry, SimuBoost marks the corresponding page dirty in its dirty logging data structures and resets the dirty bit. The bit is reset so that a write access during the subsequent interval can set it again. Before the downtime ends, SimuBoost flushes TLBs, in order to make the processor aware of the changes made to page table entries. This causes TLB misses, which can negatively affect the execution speed of the virtual machine.

To decrease the scan time, SimuBoost does not descend through page table entries that have not been accessed, because their children cannot be dirty. We dub this *pruning*. Pruning would be more effective if higher level page table entries contained dirty bits, but the hardware does not support this. Scanning only performs work during the downtime and does not cause additional VM exits while the VM is running. Figure 3.1 shows that this is the case. The small increase in exits compared to the baseline is due to checkpointing in general, interrupting the VM as well as CoW page faults cause VM exits.

To accelerate the loading of checkpoints for simulation, SimuBoost can use *sparse* checkpoint loading [39]. Sparse loading loads only those pages that will be accessed during simulation of the respective interval. In order to know which pages to load, SimuBoost must track which pages are accessed during the virtualized run of the workload. This is only implemented for scanning; in addition to the dirty bits, SimuBoost also logs and resets accessed bits. We call scanning extended in this manner *scanning+sparse*. Since pruning considers the accessed bits in higher levels page tables, not dirty bits, the set of examined page table entries is the same for scanning and scanning+sparse.

The page table walk performed when scanning constitutes additional work not done by write-protection. As a result the length of the downtime increases. Figure 3.2 shows that the mean downtime of scanning is greater than that of WP by 40% to 90%. Scan+sparse has to do additional work logging and resetting accessed bits; this involves atomic instructions. As a result, the downtime is higher, as shown by figures 3.2 and 3.3. Without pruning, the amount of work required for scanning depends on the size of the virtual machine's memory. The bigger the memory, the more page table entries exist which must be considered by scanning. With pruning, the length of scanning mostly depends on the workload and the checkpointing interval. If very little memory is accessed—either because of a light workload or because the checkpointing interval is short—few page table entries must be considered and the downtime is short. Conversely, if a workload aggressively accessed memory and the checkpointing interval is long, pruning becomes ineffective, many page table entries must be examined resulting

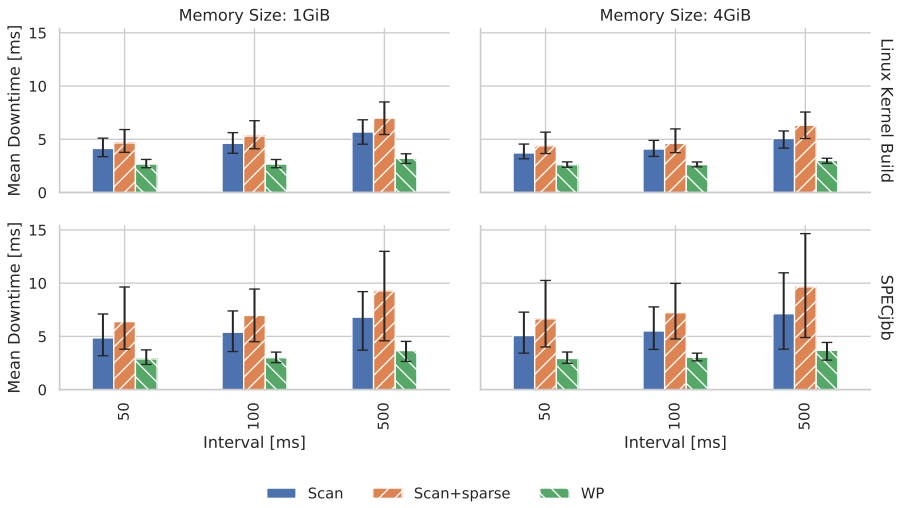


Figure 3.2: Mean downtime throughout a benchmark run. The error bars show the range from the 5th to the 95th percentile. The downtime of scanning and scanning+sparse is approximately twice that of write-protection.

in longer downtimes. Figure 3.2 shows this tendency, the effect is especially pronounced for SPECjbb.

A higher checkpointing frequency implies that more checkpoints will be taken during the time the workload runs, increasing the summed downtime. Figure 3.3 portrays this connection. At an interval length of 50 ms, 1GiB memory, and scanning as dirty logging mechanism, the total downtime for the kernel build benchmark is 98 s, that is, 7% of its execution time is downtime. Reducing the downtime directly improves the performance of the VM. If the downtime of scanning were the same as that of write-protection, the execution time of the kernel build would decrease by 2%.

Scanning and write-protection have opposing advantages and disadvantages. In the end, scanning has a smaller impact on the overall performance of the virtual machine. Figure 3.4 shows that in all scenarios scanning reduces the time to complete the kernel build. At 500 ms its overhead is 15% compared to write-protection's 24%. In the SPECjbb benchmark scanning achieves a higher score in all scenarios. Scanning+sparse's performanc is between that of scanning and WP. In part this is due to the higher downtime of scan+sparse.

While scanning improves the performance, its higher downtimes are still undesirable because they distort interaction with the VM. Figure 3.5 shows the jitter reported by the iperf [6] network benchmark. During this benchmark, the client running inside the VM sends messages to a server running

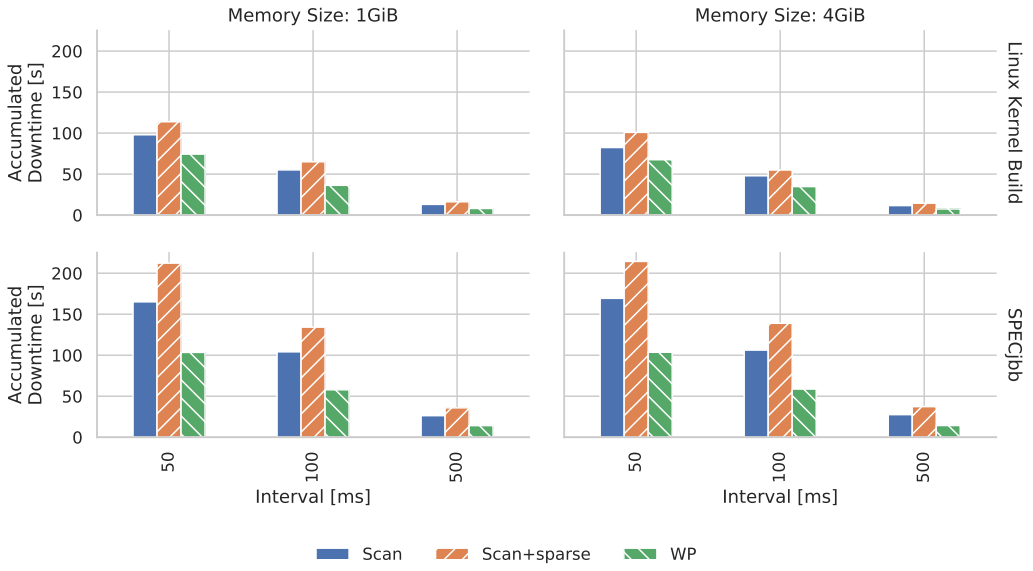


Figure 3.3: Accumulated downtime for each dirty logging mechanism. The total downtime shows the same relationship between dirty logging mechanisms as the average. The total downtime is approximately proportional to the checkpointing interval.

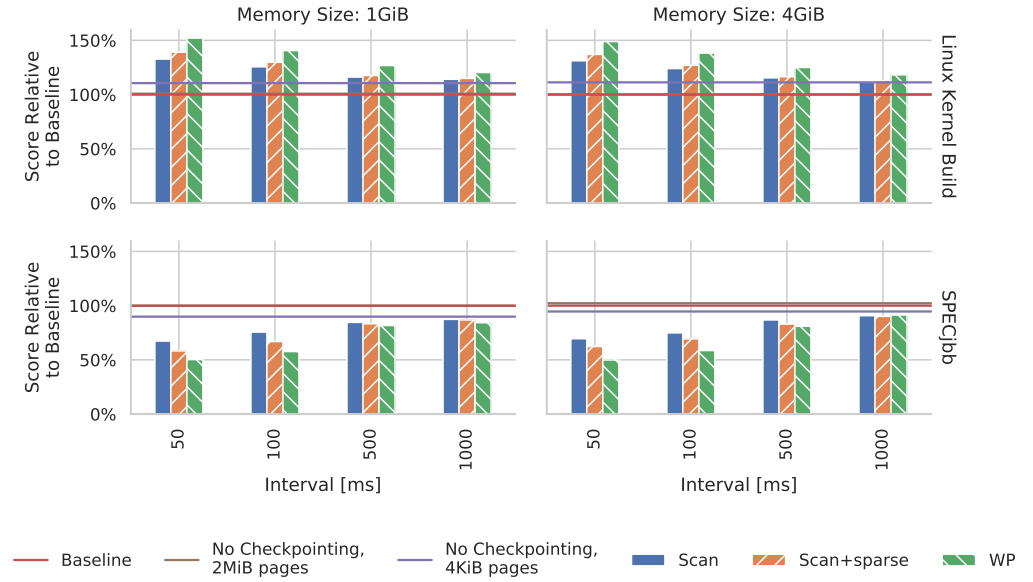


Figure 3.4: Performance for each dirty logging mechanism, relative to the baseline. The baseline constitutes of no checkpointing and the default page size. For the kernel build benchmark the vertical axis denotes the relative execution time, lower is better. For SPECjbb it denotes the relative score, higher is better. The performance of scanning is better than that of WP. As the interval length increases, the performance improves.

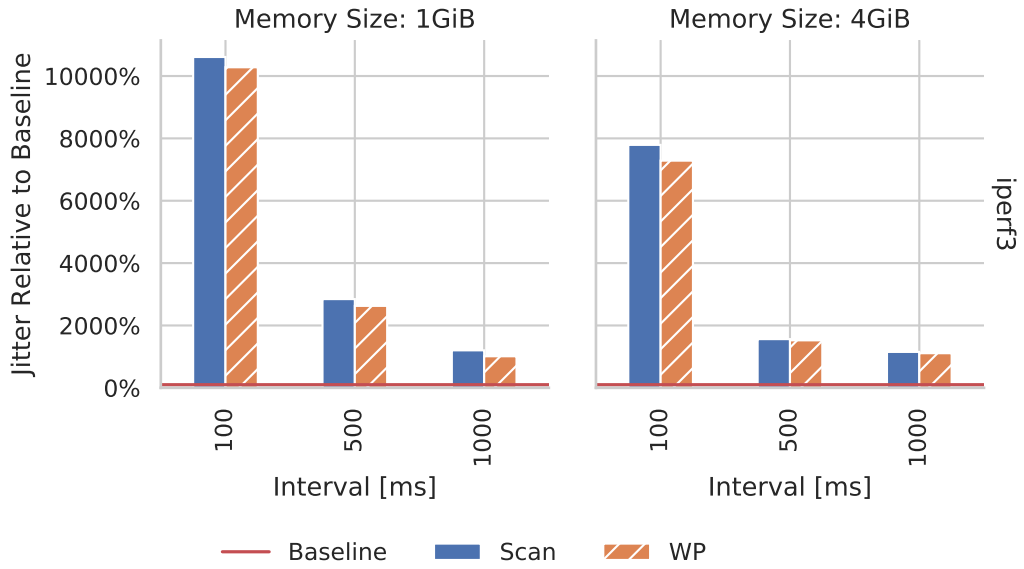


Figure 3.5: Mean jitter observed by the iperf3 benchmark, relative to the baseline. The baseline represents the jitter when no checkpointing is taking place. The jitter is significantly higher when checkpointing is enabled and increases as the checkpointing interval shrinks.

on the host. Jitter is the mean of the differences between consecutive transit times [7]. If a downtime occurs during a transit, this transit time increases, increasing the jitter (assuming no downtime during the previous transit). As a result, the average jitter for scanning is between 12 and 77 times higher than the baseline value. The jitter increases with decreasing checkpointing interval length because downtimes are more frequent. The jitter is slightly higher for scanning than for write-protection. The difference would likely be higher if the benchmark were more memory intensive.

3.2 Impact of Copy-on-Write

Copy-on-write checkpointing negatively impacts the execution of the virtual machine. Each CoW page fault implies a VM exit. While the CoW case is processed by saving the page, the execution of the virtual processor is halted, the workload can therefore not advance for this time. Both VM-μCheckpoint [47] and Speculative Memory Checkpointing [46] try to avoid copy-on-write page faults, citing overhead. Avoiding CoW cases in Simu-

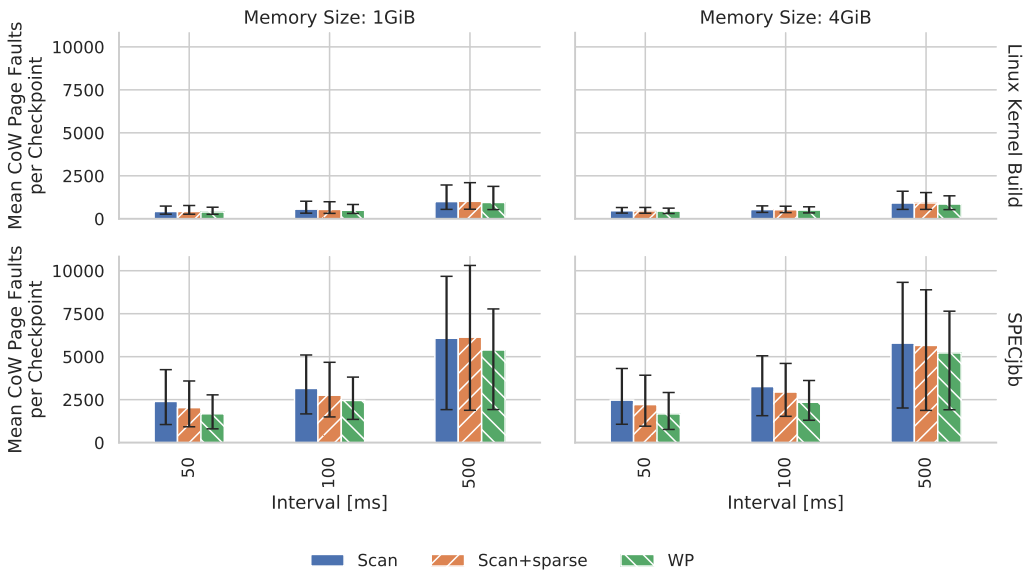


Figure 3.6: Mean number of CoW cases per checkpoint. The error bars show the range from the 5th to the 95th percentile. The number of CoW cases depends on the benchmark and increases with the checkpointing interval.

Boost has the additional advantage that it minimizes the distortion of the workload behavior, making SimuBoost's analysis more accurate.

Figure 3.6 shows the average number of CoW cases per checkpoint. How many CoW cases occur depends on the checkpointing interval and the workload. During a very short interval the workload can modify only few pages. Thus, only few pages must be captured in the next checkpoint, limiting the number of pages potentially captured via CoW. However, as with the total downtime, shorter intervals result in a higher total number of CoW cases. Figure 3.6 shows that SPECjbb causes significantly more CoW cases than a kernel build, indicating that it more aggressively writes memory. There is a small difference between the dirty logging mechanisms. Write-protection has a higher run time overhead than scanning, due to its dirty logging page faults. Comparatively, this speeds up concurrent-copy, leading to less CoW cases.

Figure 3.7 gives an overview over the distribution of CoW cases. For each checkpoint it plots the time CoW page faults occur relative to the end of the downtime. It also depicts when concurrent-copy (CC) has completed. The lower plots of the figure shows the same for a small range of checkpoints. Additionally, they show an estimate of the distribution of CoW cases and CC completion. The figure shows that CoW cases are most likely directly

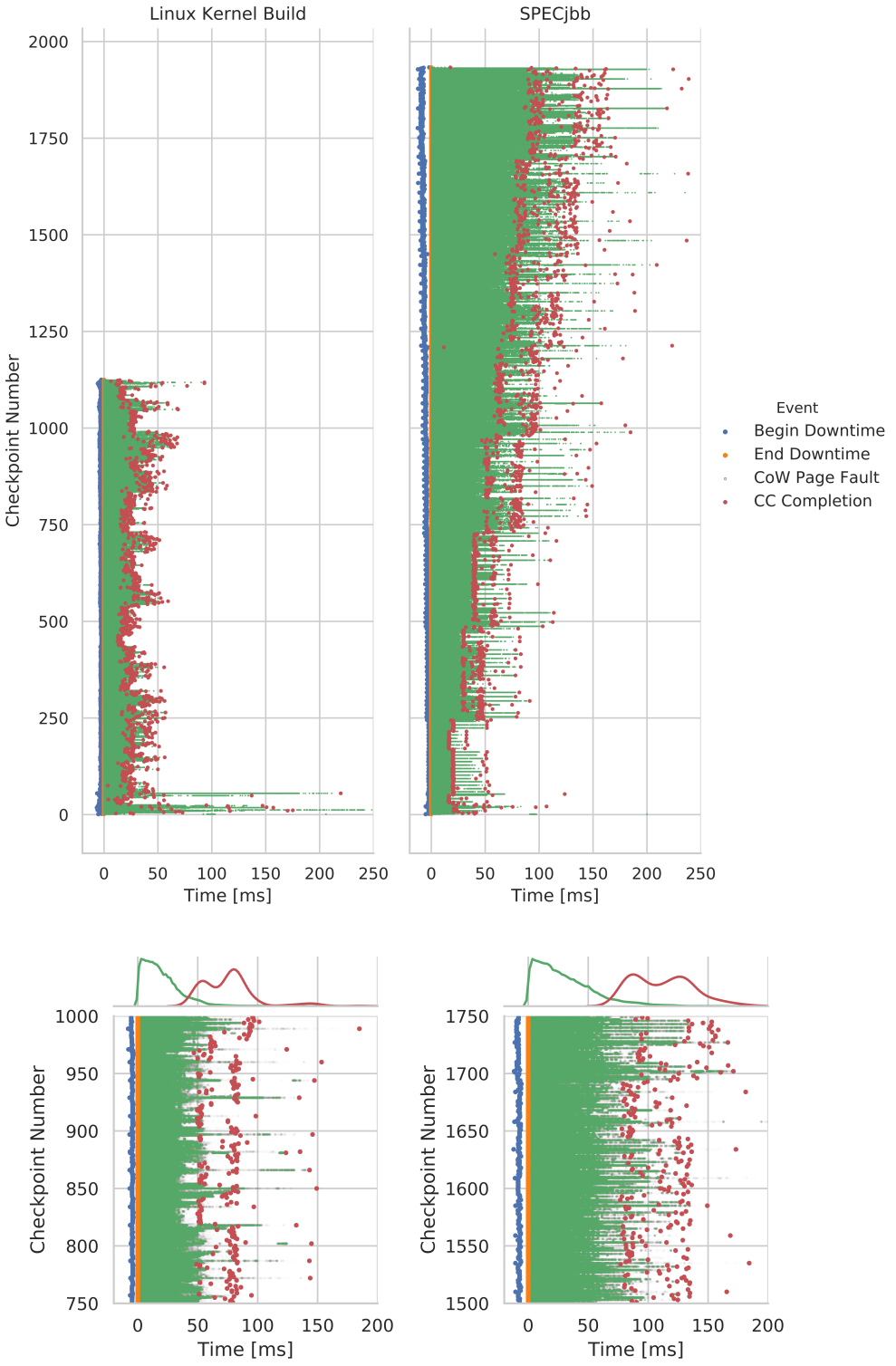


Figure 3.7: Timing of events during incremental CoW checkpointing with interval length 1 s, 4GiB memory. The horizontal axis denotes time since the end of the downtime period. The lower plots shows a smaller range of checkpoints, as well as an estimate of the distribution of CoW cases and CC end points. The plots show that CoW cases occur even briefly before checkpointing completes.

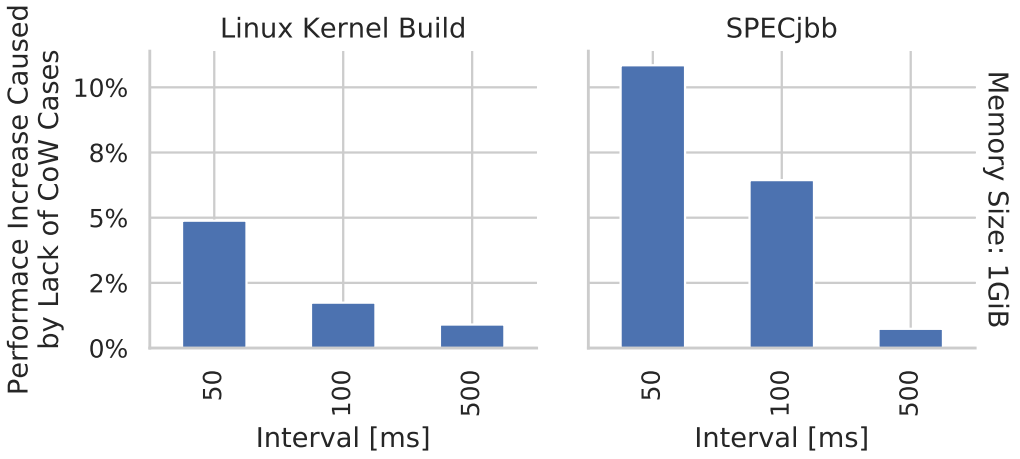


Figure 3.8: Increase in performance if no CoW cases occur, higher is better. For SPECjbb, performance improves by more than 5% for interval lengths less than 100 ms.

after the downtime. This is because concurrent-copy has not yet had time to copy many pages. The effect is more pronounced for the kernel build. As time goes on, the probability of CoW cases drops. After CC has completed, CoW cases can no longer occur. The figure does not show a big gap between the last CoW case and the completion of CC. Reducing the time to complete concurrent-copy by speeding it up would therefore lower the number of CoW cases.

To estimate the possible performance improvement, we experimentally omit write-protecting the pages to be saved. This breaks copy-on-write checkpointing and simulates a 100% reduction in CoW cases. Figure 3.8 shows the resulting change in performance. For 50 ms SPECjbb's score improves by more than 10%, for 100 ms more than 5%. For higher interval lengths the improvement becomes negligible. These values represent the best case, with no CoW cases occurring. However, the improvement is large enough that a reduction of CoW cases by 50% or 30% should cause a noticeable benefit.

3.3 Impact of Page Size

KVM supports large pages [26] but breaks them when dirty logging is active because it logs dirty memory at a 4 KiB granularity. Large pages can be explicitly configured or they are automatically created via transparent hugepages [50]. By default, QEMU requests the use of large pages.

Large pages can increase performance by improving TLB utilization. Figure 3.4 confirms this. For SPECjbb the score of 2 MiB pages is 8% higher than that of 4 KiB pages. A Linux kernel build with 2 MiB pages finishes 10% earlier than when 4 KiB pages are used. Transparent hugepages perform similar to explicitly configured 2 MiB pages. Starting at an interval length of 500 ms, the performance of a kernel build with checkpointing enabled and scanning as dirty logging mechanism is close to that of no checkpointing and 4 KiB pages.

This suggests for long interval lengths, the performance of incremental checkpointing, compared to execution without checkpointing, is limited by its need for a dirty logging mechanism, which currently restricts the page size to 4 KiB.

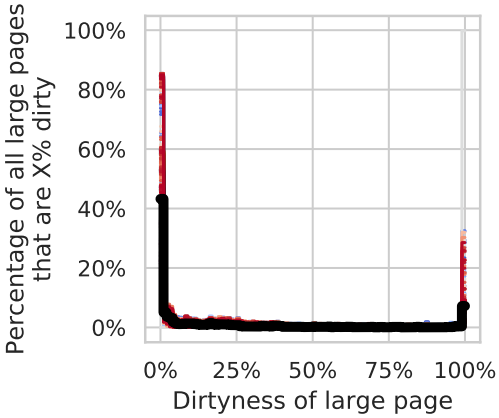
Dirty logging could also be done using larger pages. However, dirty logging at a coarser granularity implies that more memory will be considered dirty, and therefore more memory would need to be captured. This might negate the benefit of large pages.

Figure 3.9 gives an overview of the effect of dirty logging at the 2 MiB level. The Figure visualizes the memory dirtied by SPECjbb at a checkpointing interval of 1000 ms. Figure 3.9b shows the VM's memory at a checkpoint roughly halfway through the execution of the benchmark. Black pixels symbolize a clean 4 KiB page, white ones a dirty page. Figure 3.9d shows the same memory if instead dirty logging was 2 MiB based. A large page is considered dirty if any of the small pages covered by it is dirty. The entire memory covered by the large page must be considered dirty, increasing the amount of memory that must be saved by a factor of ≈ 5.6 in this instance.

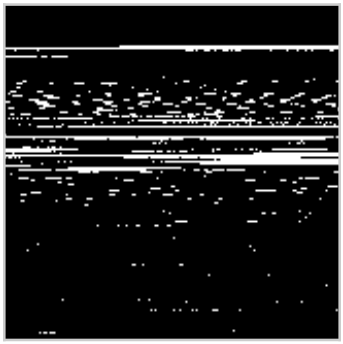
In the following, we assign a dirtiness percentage to each large page, based on how much of its memory is considered dirty by 4 KiB dirty logging. Figure 3.9f shows clean large pages and completely dirty large pages in white, that is, it shows partially dirty large pages in color. The shade of partially dirty pages is determined by their dirtiness. Many partially dirty pages are hardly dirty, these pages contribute the majority of the memory that would be checkpointed needlessly.

The left column of Figure 3.9 gives an overview of large page dirtiness throughout the benchmark. Each plot has one curve per checkpoint, drawn in a different shade. Figure 3.9a depicts the distribution of dirtiness. It shows spikes at 0% and 100%. The values in-between are partially dirty pages, their distribution skews towards little dirtiness. The black curve describe the average over all checkpoints.

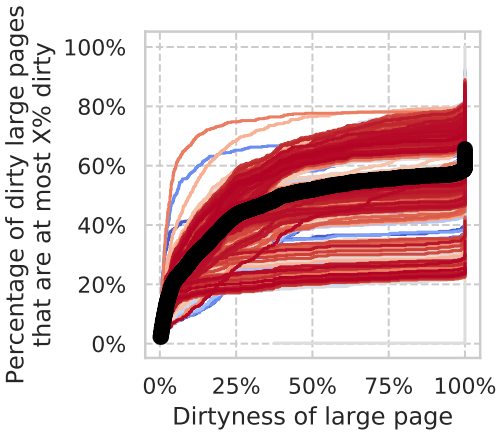
Figure 3.9c displays the percentage of large pages whose dirtiness falls into the interval $(0\%, x\%]$. Figure 3.9e displays the percentage of pages that falls into $[x\%, 100\%]$. Again, the black curves depict the average. Both plots



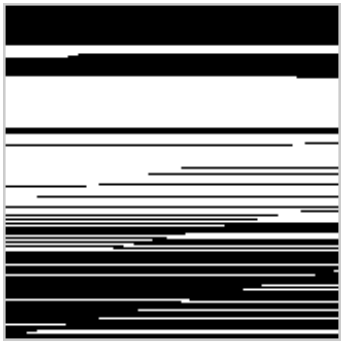
(a)



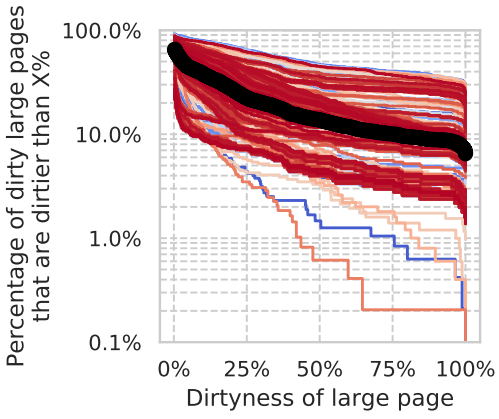
(b) VM memory halfway through the benchmark. Clean 4 KiB pages are black, dirty ones white.



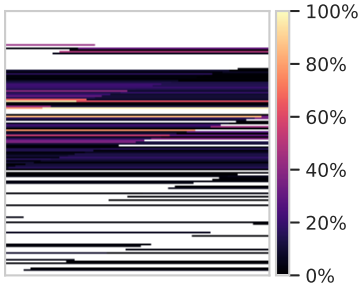
(c)



(d) Equivalent dirty memory if dirty logging had large page granularity. A coarser granularity leads to more dirty memory.



(e)



(f) Partially dirty large pages in color, clean and completely dirty pages in white. Most partially dirty pages had little memory actually modified.

Figure 3.9: Overview of dirty memory throughout an execution of SPECjbb, 1s interval length, 1GiB memory. Figures 3.9a, 3.9c and 3.9e: Histograms showing distribution of dirtiness. One line for each checkpoint, drawn in a different shade. The black lines show the average over all checkpoints. While the distribution varies over the checkpoints, all show a substantial number of large pages that would cause a lot of useless copy work.

show that large pages which are hardly dirty are very common. Between 20% and 60%, 40% on average, of large pages are at most 20% dirty. These pages alone cause 16% to 48% of the VM's memory to be wrongly considered dirty. Very dirty pages are rare, on average only $\approx$ 10% of large pages are more than 60% dirty.

Summed over all checkpoints, dirty logging at large page granularity deems ≈ 3.6 times more memory dirty than dirty logging at the 4 KiB level. This substantial increase would entail additional overhead, in part due to more CoW cases. Because of this, it is doubtful that large pages can increase checkpointing performance.

3.4 Conclusion

We have analysed aspects of SimuBoost's checkpointing implementation and identified sources of overhead. SimuBoost creates periodic downtimes in the execution of the virtual machine. Scanning and scanning+sparse as dirty logging mechanisms increase the length of the downtime compared to write-protection. Additionally, SimuBoost causes copy-on-write page faults, which slow the execution of the VM. SimuBoost also reduces performance by disallowing the use of large pages. These aspects represent opportunities to improve checkpointing that we will explore in the rest of the thesis.

Chapter 4

Design & Implementation

This chapter presents the design and implementation of three optimizations to SimuBoost. The first addresses the downtime increase caused by scanning, the second aims to reduce the number of copy-on-write cases and the third serves to explore if large pages can benefit checkpointing despite the increase in memory to capture.

4.1 Pre-Scan

To counter the increase of the downtime caused by scanning, we propose doing a *pre-scan*. The idea of pre-scan is to shift some of the scanning work before the downtime, when the virtual machines is still running. However, because the VM is running, it can modify pages after the pre-scan has completed, a scan during the downtime is therefore still required. The mechanism by which pre-scan could decrease the downtime is to increase the effectiveness of pruning during the downtime scan. To do so, a pre-scan scans the page tables, logs dirty pages and resets the accessed bits on all levels. If the VM does not access the memory region mapped by a higher level page table entry after the pre-scan, its accessed bit will remain zero, allowing all its children to be skipped when it is scanned during the downtime.

Use of pre-scan must not break dirty logging, care must therefore be taken to coordinate the access to page tables by pre-scan and the running virtual processor. For example, the following sequence must not occur:

1. **Pre-scan:** Observes that a higher level page table entry E was previously accessed

2. **Pre-scan:** Descends and scans E's children, finds no dirty pages

3. **Virtual processor:** Writes to a page mapped by a child of E, sets the dirty bit in the child page table entry

4. **Pre-scan:** Resets E's accessed bit in memory

5. **Virtual processor:** The processor's translation caches indicate that E's accessed bit is already set, therefore, it does not need to be set in memory again

6. **Pre-scan:** Invalidates translation caches

The result of the above sequence is a last level page table entry with set dirty bit and a non-set accessed bit in its parent. The final scan in the downtime does not find the dirty page because it is pruned when the parent page table entry shows a zero accessed bit.

To solve this problem, each pre-scan consists of multiple steps:

Step 1 Walk the page tables. If the accessed bit of an entry is one, set it to zero. Let P be the set of entries modified in this manner. Store enough information to identify P.

Step 2 Invalidate TLBs. After this step, the following statements hold:

- All accesses that took place *before* this step and for which pre-scan observed a set accessed bit in Step 1, are covered by entries identifiable via the information retained in Step 1.
- All accesses that took place *before* this step and which pre-scan was not aware of (e.g., because they occurred after pre-scan read the according page table entry), are covered by page table entries that have the accessed bit set. This is the case because pre-scan did not reset the accessed bit.
- All accesses that take place *after* this step will cause the processor to set the accessed bit. This is because the translation caches no longer contain outdated data.

As a result, all accesses can be identified via the accessed bits in the according page table entries *or* via the retained information.

Step 3 Scan children of entries in P and log dirty bits.

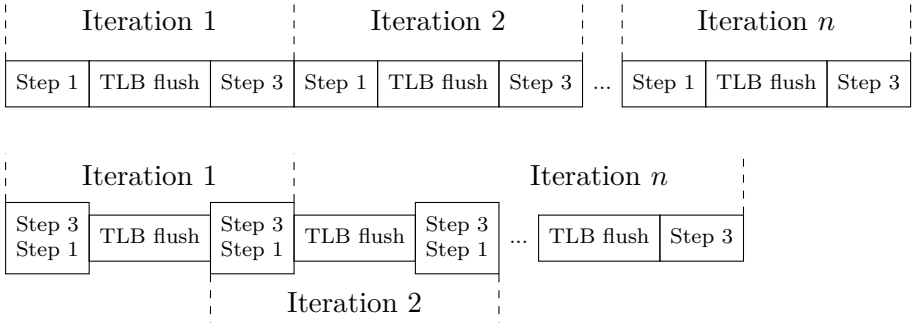


Figure 4.1: Simple and pipelined pre-scan. Pipelined pre-scan combines Steps 1 and 3, shortening the iteration: n iterations with the simple method require $2n$ page table walks, n pipelined iterations only $n + 1$.

A pre-scan followed by a normal scan during the downtime will correctly log all dirty pages.

It is possible to pre-scan multiple times, similar to pre-copy live migration. This might be beneficial due to the same rationale: Each round must perform less work than the previous, completes faster, and thus grants the virtual machine less time to create work for the subsequent round.

During Step 1 we need to retain the information which page table entries are in P. We chose to store this information for each page table entry in one of the entry's unused bits [22, ch. 28.2.2]. In order to identify entries in P in Step 3 and scan their children, we walk the page tables and examine this bit.

As a result, the use of multiple rounds can be further optimized by combining Steps 1 and 3. This has the advantage that entries that would be considered by both Step 1 and 3 are examined only once. We call this approach *pipelined* pre-scan and a pre-scan with separate Steps 1 and 3 *simple*. Figure 4.1 contrasts the two approaches. We define two variants of pipelined pre-scan. During the final step *pipeline-early* logs all dirty pages it finds, while *pipeline-late* scans only pages covered by entries in P. Because of this, pipeline-early might require the scan during the downtime to sync fewer dirty bits. However, they do not differ in which page table entries the downtime scan must examine.

To implement pre-scan, we modify SimuBoost's existing scanning function so that it can be used to perform Step 1, Step 3 and the scan during the downtime. We do this by making the behavior of the function conditional on a number of *flags*. These flags determine which page table entries are descended through during the walk (accessed and/or in P) and which actions are performed (e.g., reset accessed bits and/or perform dirty logging).

When we modify page table entries, we do so atomically in order to prevent interference with modifications by the virtual processor.

The KVM part of SimuBoost is not aware of the frequency or timing of checkpoints. We therefore added a pre-scan `ioclt` to KVM, so QEMU can initiate a pre-scan before the next downtime. The `ioctl` is passed the number of iterations to perform, as well as the method—pipelined-early, pipelined-late or simple.

The time between the pre-scan and the downtime should be minimal in order to give the virtual machine less time in which to dirty memory. To achieve this, the QEMU part of SimuBoost sleeps until a moment before the downtime, calls the pre-scan ioctl and then immediately commences the downtime. As a result, the actual interval length depends on the duration of the pre-scan `ioctl`. To minimize the deviation of the actual interval length from the intended one, we predict the length of the pre-scan `ioctl` to be the exponential average of past invocations, and adjust the moment at which we perform the pre-scan accordingly.

Just as normal scanning optionally supports sparse checkpoint loading, we create versions of pre-scan that additionally sync accessed bits, which we refer to as pre-scan+sparse.

4.2 Parallel Copy

To decrease the number of copy-on-write page-faults, we parallelize concurrent-copy. The intention is to shorten the time until a page is captured by concurrent-copy by maximizing the bandwidth with which pages are saved. If pages are saved quickly, the workload has less time to generate CoW cases.

The current SimuBoost implementation concurrently copies pages in a single thread. This thread requests a segment to fill with pages from the Simustore server. It then calls into KVM via an `ioctl`, passing it the segment buffer. This `ioctl` iterates over the physical memory of the virtual machine and copies pages if they were dirtied during the previous interval and have not yet been saved by copy-on-write. This continues until the buffer is full or CC has completed. Then the `ioctl` returns, informing SimuBoost about the number of copied pages and whether there remain additional ones to capture. Upon return from the `ioctl`, SimuBoost submits the segment to Simustore, in order to append it to the stream that stores modified pages. If more pages need to be saved, SimuBoost repeats the process. Subsequent invocations of the `ioctl` start saving pages where the last call left off, so that all required pages are copied. KVM achieves this by storing the relevant information for each virtual machine.

Multithreaded concurrent-copy requires that:

1. Multiple threads can obtain segment buffers of the same stream
2. KVM supports multiple copy threads
3. The copy threads capture pages at the same time
4. SimuBoost is informed if more pages are to be captured
5. KVM correctly resumes copying
6. Multiple threads can submit segments to Simustore independently

Sharing the copy work between multiple threads, as demanded by requirement 3, necessitates coordination between the threads. A possible design would be to break the VM's physical memory into equally sized parts and assign each part to a thread. However, this design underutilizes threads that have little work to do. After a thread has examined the region assigned to it and found that it does not contain pages to save, it cannot relieve other threads that have actual copying work to do.

Instead, we propose the following design: Whenever a copy thread has no work, it assigns itself a continuous region of pages of a maximum size. These regions are sequential according to an arbitrary order. A shared data structure points to the next region. Access to the data structure is synchronized via a lock.

This design has the advantage that a thread can find new work after it is done with its region. There is a trade-off between small and large maximum region sizes. Large sizes require less synchronization but limit work sharing, small sizes might lead to lock contention.

We implement this design by extending the mentioned `ioctl`, no new `ioctl`s are added. In addition to the segment buffer, SimuBoost passes a thread id to the `ioctl`. Only one thread with the same thread id may execute the `ioctl` at the same time for the same virtual machine. We set the maximum number of thread ids to 128. Each thread id can be associated with a region to save. This is used to track if there is work to resume when the thread enters the `ioctl` again, after it had to leave without completing the region (e.g., because the buffer was full).

Whenever a thread needs more work, it acquires it as per our design. The access to the shared data structure is synchronized via a spinlock. We choose a spinlock because the lock is going to be held only shortly.

If a thread attempts to acquire more work and finds that there is none left, it returns this information to userspace. For concurrent-copy to function

correctly, SimuBoost must keep invoking the `ioctl` for each used thread id until this occurs.

Currently, Simustore supports only one open segment per stream. To implement requirements 1 and 6, we modify Simustore in the following way: We add a *DeferredOrder* flag to the flags associated with a stream. This flag signifies that the assignment of a sequence number to a segment is deferred until the segment is appended, instead of occurring when it is opened. Until this time, the segment is assigned a dummy sequence number, of which there exist a fixed amount. When the segment is appended, it is renamed, that is, it is assigned the next proper sequence number of the stream. Its dummy sequence number can then be reused for a newly opened segment. During an append operation, the client must now identify which segment to close instead of Simustore implicitly closing the last opened segment. This identification is done via the dummy sequence number.

In the course of implementing parallel copy, we also added an optimization to the copy-on-write implementation. Previously, the CoW handler would atomically allocate memory to save the page to. If the allocation pressure was high, these allocations would fail.

Instead, we directly save CoW pages to userspace supplied buffers, as with CC pages. We use the same `ioctl` for the implementation and add a flag to the invocation determining if it is to save CoW pages. Once such a CoW thread enters KVM, it sleeps until it is woken. We use a Linux wait queue [45] to do so. A CoW thread is woken when its buffer is full, an error occurs in the CoW handler or concurrent-copy has completed. This is the case when a concurrent-copy thread exits KVM and there is no more work it can do, nor do there exist copy threads with resumable work. When no CoW thread is available, KVM falls back to atomic allocation. If multiple CoW threads are available, KVM completely fills the buffer of one thread before using another threads buffer. If userspace continuously supplies KVM with buffers to use for CoW, atomic allocations can be completely avoided.

KVM's page fault handler must be non-interruptible, else a deadlock might occur. For this reason, if the copy to an userspace supplied buffer is to be guaranteed to succeed, the virtual to physical translations covering the buffer must exist, they cannot be faulted in by the CoW handler. To assure this, QEMU pins the buffer in virtual memory using `mlock` [21].

4.3 Large Pages

The space of designs incorporating large pages into incremental checkpointing is broad. Such a design must determine which parts of memory are to be

backed by large pages, as it may be detrimental to performance to include areas that will cause many superfluous copy operations. The simple choice is to use large pages for all memory. The alternative is to predict which areas will be dirtied and adjust the use of large pages according to this prediction. Such a design would be significantly more complex. It would need to asses the dirtiness of large pages but could not depend on dirty logging to do so because the dirty logging mechanism itself would work on a large page level. In the context of SimuBoost, the storage back end could provide this information; its deduplication mechanism detects where memory was actually changed. Given this information, there is a wide range of possible policies to decide when to use large pages vis-á-vis small pages.

Incorporating large pages also needs a way to deal with copy-on-write page faults to large pages. One possibility is to copy the large page. However, this would substantially increase the cost of a CoW page fault, as 512 times more memory must be copied. The alternative is to break the large page and establish a 4 KiB page at the faulting address. Then, only this small page must be saved. To benefit the workload, large pages broken in this manner must be re-established. Again, different designs are possible. Re-establishing large pages early allows the workload to make better use of them. If they are re-established while CoW cases are still possible, they may be broken again. Depending on the implementation, this causes additional overhead via page faults and TLB invalidations. The other possibility is to re-establish a large page only if it is not possible for a CoW case to hit the page. This is certainly the case when the checkpoint has been completed. But it can also be guaranteed to be the case when the large page in question has been completely captured by concurrent-copy, rather than all of them. This requires cooperation between re-establishment and concurrent-copy.

We implemented the following simple design:

Large-zap: Use large pages for the entire memory, break them in case of CoW. After a checkpoint has completed, remove all 4 KiB page table entries, subsequent non-present page faults create large pages.

We do not change the granularity of the dirty bitmap KVM uses to store the dirty information. Instead we set 512 bits for each dirty large page. Our implementation covers only scanning as dirty logging mechanism, without sparse support.

Chapter 5

Evaluation

5.1 Methodology

We automated benchmarking by implementing the benchmarks as test cases for the Avocado testing framework [4]. In order to enable the test cases to interact with SimuBoost, we modified avocado-virt [5], a library to create virtualization related tests. Unless otherwise stated, the following applies:

- We disable NUMA balancing.
- We set the CPU scaling governor to performance.
- VCPU threads are pinned to different cores.
- QEMU is started with `-nodefaults`, `-mem-prealloc`, an IDE-HD device, a rtl8139 network device and character devices for the serial console and monitor.
- The baseline value is that of no checkpointing, default page size.
- When we explicitly use large pages, we start QEMU with `-mem-path /dev/hugepages`.
- When we explicitly use 4 KiB pages, we set the transparent hugepage parameters "enabled" and "defrag" to "never".
- We set the `mlock` quota high enough for each CoW thread to use an entire Simustore segment.
- We evaluate parallel copy with scanning without sparse support as dirty logging mechanism.

- We record the number of certain events occurring throughout the execution with the perf [30] utility.
- We record the timings of: start of downtime, end of downtime, CoW case, CC completion with the trace-cmd [38] utility.
- When using the pre-scan version of SimuBoost, we record statistics for each (pre-)scan, also with trace-cmd.
- We start the java process that runs SPECjbb [8] with `-Xmx1G`
- Linux Kernel Build as a benchmark refers to the benchmark in the Phoronix Test Suite [2].
- We perform only one kernel build per benchmark run.
- Values aggregated over each checkpoint (e.g average downtime) exclude the first checkpoint in order to improve their descriptiveness of the common case.

We perform the evaluation on a system with the following specifications:

Processor	Dualsocket Intel Xeon CPU E5-2630 v3 @ 2.40GHz (8 physical cores, 16 logical)
Memory	64 GiB
Disk	Crucial CT256MX1 (System disk)
Disk	Samsung SSD 850 (Simustore output directory)
Motherboard	Supermicro X10DRI-T 1.02

Used software versions are:

Host Operating System	Ubuntu 16.04.5 LTS
Host Linux Kernel	4.3.0+
Guest Operating System	Ubuntu 18.04 LTS
KVM	4.3.0+
QEMU	2.6.50
Simutrace	3.4.1
SPECjbb [8]	2005 v1.07
Phoronix Test Suite [2]	8.0.0
Phoronix Test Suite Linux Kernel Build	1.9.1

Phoronix Test Suite Ramspeed	1.4.1
Phoronix Test Suite iperf	1.0.2
iperf3 [6] (server)	3.0.11

5.2 Correctness

To asses if checkpointing functions correctly with our modifications in use, we compare the memory of the VM immediately before taking a checkpoint with that saved by the checkpoint. We run a workload and save the VM's memory during the downtime, before creating the checkpoint. After the workload has completed, we load every checkpoint and save the resulting memory image. Because saving the complete memory is slow, we use a long checkpointing interval of 20 s. The before and after memory images being identical indicates that checkpointing is correct but does not guarantee it. We find that the images are identical when our modifications are used.

5.3 Pre-scan

The aim of pre-scan is to reduce the downtime. To asses if it accomplishes this, we benchmark a subset of pre-scan configurations. To keep the number of benchmarks manageable, we do *not* explore all combinations of pre-scan methods (simple, pipelined-early, pipelined-late) and iterations up to some limit.

We choose to test pipelined-early with iterations from one to three. By design, the pipelined method should show more strongly if iterating pre-scan can reduce the downtime. We also perform one iteration of the simple pre-scan method (i.e. with separate Steps 1 and 3), to give us a baseline value with which to compare the more advanced pipelined design. We compare pipelined-early+sparse and pipelined-late+sparse, which additionally sync accessed bits. If syncing more page table entries during the last step of pre-scan is beneficial, it should be more apparent with the sparse variant. The rationale is that, if pipelined-early is able to move work out of the downtime, it will move more work for scanning+sparse, creating a bigger contrast between pipelined-early and pipelined-late.

Figure 5.1 shows the downtime averaged over the workload execution for the chosen pre-scan configurations, as well as scanning without pre-scan, and write-protection. It shows that pre-scan succeeds in lowering the downtime, with all configurations being similar. On average over all scenarios, pre-scan without sparse support reduces the downtime to 20% of that of write-

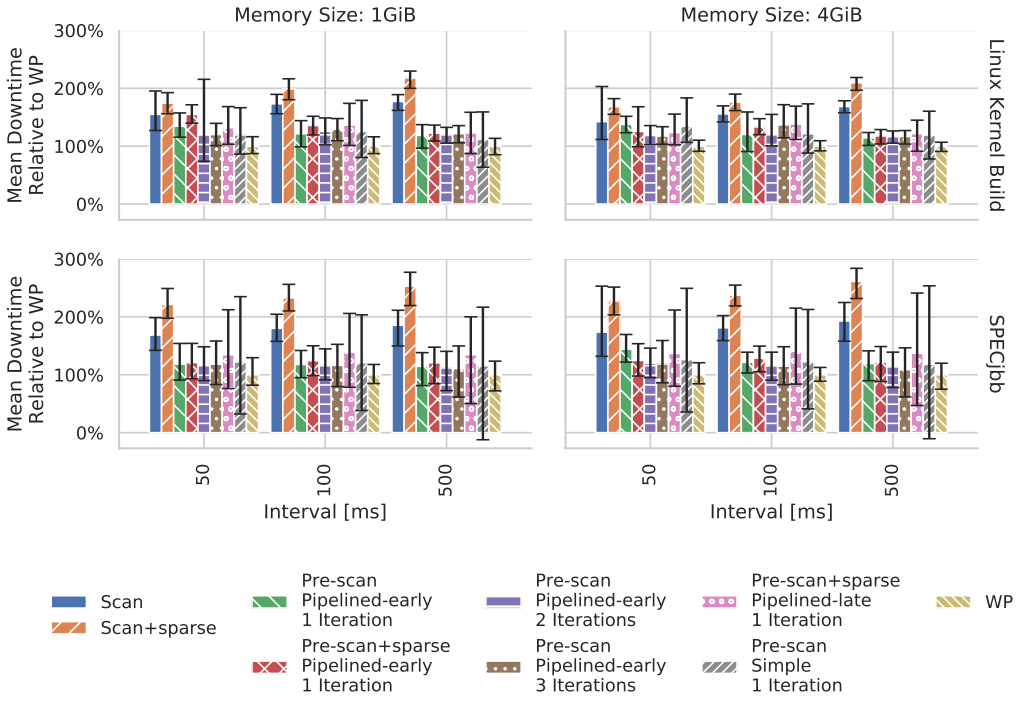


Figure 5.1: Mean downtime for various dirty logging scenarios, relative to write-protection. The error bars show the range from the 5th to the 95th percentile. Pre-scan is able to reduce the downtime and brings it close to that of write-protection. Pre-scan+sparse has downtimes slightly increased compared to pre-scan without sparse support. Pre-scan+sparse pipelined-late has longer downtimes than pipelined-early in most cases. Other than that, there is little difference between the choices of pre-scan configurations.

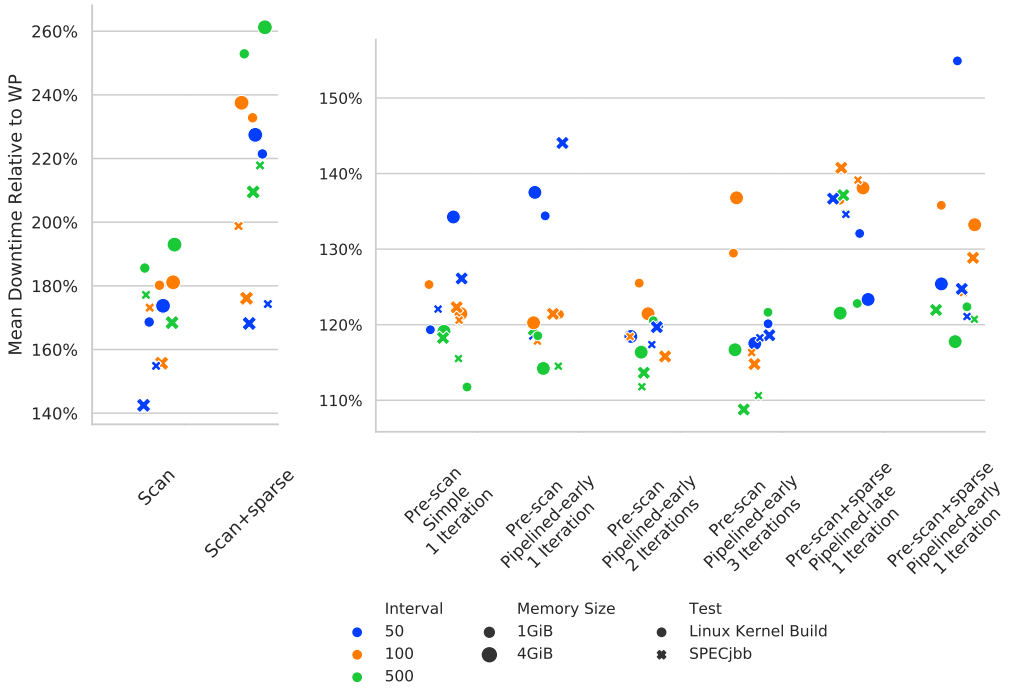


Figure 5.2: Mean downtime for different scanning configurations relative to the downtime of write-protection. For pre-scan without sparse support there is little difference between simple pre-scan and pipelined-early pre-scan regardless of the number of iterations. For pre-scan+sparse pipelined-late 1 iter. (= simple 1 iter.), most scenarios have a downtime more than 30% increased compared to WP. This is not the case for pipelined-early.

protection, whereas scanning without pre-scan results in a downtime 70% higher than that of WP. The same is true when sparse loading is supported, here the downtime is on average 30% higher than WP's, without pre-scan 114%. Pre-scan is therefore able to lower the additional downtime caused by scanning by more than 50%.

There is no clear trend showing that iterating pre-scan can decrease the downtime. While it is the case for some scenarios (e.g. kernel build, 1G, 50 ms), others shows a higher downtime (kernel build, 4G, 100 ms) and some show hardly any difference (kernel build, 4G, 500 ms). Figure 5.2 shows the downtimes relative to WP for all configurations. While for pipelined-early the highest downtimes are at one iteration, the majority of points fall within 110% to 125% of write-protection's downtime, regardless of the number of iterations. Additionally, there are some outliers for three iterations.

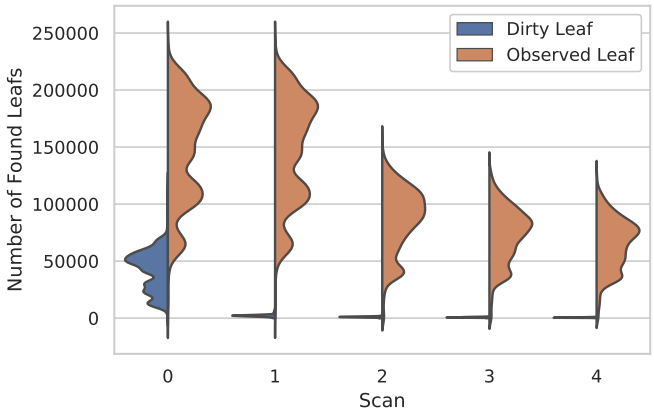


Figure 5.3: Overview of the number of found dirty pages and the number of pages that had to be considered during a pipelined-early pre-scan with 3 iterations. Each scan shows an estimate of the distribution of the number of dirty and considered pages over all checkpoints. The data is collected from an execution of SPECjbb. Scans 0–3 occur during the pre-scan, scan 4 during the downtime. For scan 0, most checkpoints find approximately 50 thousand dirty pages, as shown by the peak. All subsequent scans find hardly any dirty pages. The number of considered pages can only decrease after scan 2, and does so. However, the distributions hardly change thereafter.

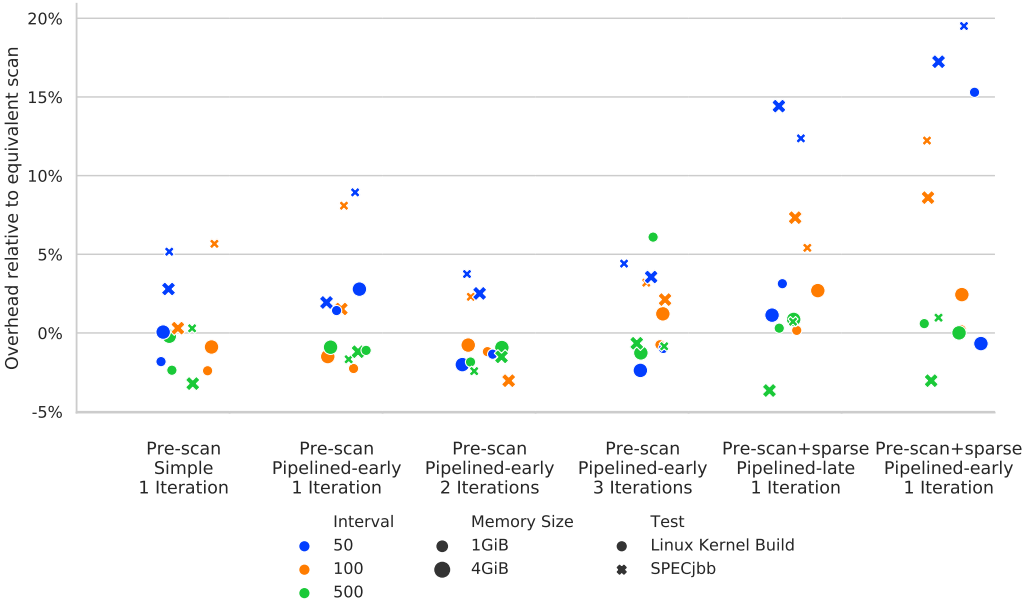


Figure 5.4: Overhead of pre-scan relative to the equivalent scan, that is, pre-scan is compared to scan+sparse iff it supports sparse. For the kernel build it is calculated as the increase in execution time, for SPECjbb as the reduction of the score. Lower is better. While pre-scan is able to improve performance for interval lengths of 500 ms, the performance of smaller intervals mostly suffers. This is especially pronounced for pre-scan+sparse.

Figure 5.3 portrays the work done during a pipelined-early pre-scan with three iterations. For each page table walk, it shows the number of leaf page table entries examined, as well as the number of dirty ones found. The number of entries are shown as the distribution over all checkpoints. The first two scans are required regardless of pre-scan method or number of iterations, the last scan is that during the downtime. Scans two and three represent the additional scans performed by iterated, pipelined pre-scan. Figure 5.3 shows that the first scan takes care of the majority of *dirty* pages, subsequent scans find only few newly dirty pages. This is represented by the narrow spikes close to zero for the scans after the first one. Pre-scan can therefore reduce the number of dirty bits to sync with a single iteration. Additional pre-scan iterations, however, cannot reduce the number of page table entries to *examine*, the distribution of entries observed during scan two is very similar to that of the scan during the downtime.

For one iteration, pipelining conceptually makes no difference. As a result, the downtimes of pipelined and simple pre-scan are similar. Simple pre-scan has a downtime 21% higher that that of WP, pipelined-early 23%.

For SPECjbb, pipelined-early+sparse reduces the downtime more than pipelined-late+sparse. Here, pipeline-late leads to a downtime 37% higher than that of WP, pipeline-early to 24%. For the kernel build, the values are within three percent of each other. Because only one iteration is performed, pipelined-late is equivalent to simple pre-scan.

Figure 5.4 shows the change in performance caused by pre-scan. The values shown are percentages based on the equivalent normal scan, if pre-scan supports sparse it is compared to scan+sparse and vice versa. For the kernel build the percentages represent an increase in execution time. For SPECjbb, the numbers represent the reduction in points. For both, lower is better. The figure shows that for pre-scan without sparse support the overhead lies between -5% and +10%, with most values being within 5% of the equivalent scan. The overhead for most scenarios with an interval length of 500 ms show improved performance. On average, pre-scan+sparse is 0.4% faster than scan+sparse for 500 ms, pre-scan without sparse support 0.9% faster than scan without sparse.

Pre-scan+sparse shows performance degradation of up to 20%, especially for short interval length. The cause of this is unknown as of this time. We have ruled out TLB misses as the culprit by experimentally removing the TLB flushes from pre-scan. This did not substantially change performance.

The pre-scan implementation must protect the page table walk from concurrent modifications by the virtual machine monitor. For example, if a page containing a page table were to be removed from the paging hierarchy and repurposed, pre-scan might read corrupted data. To prevent this, the pre-

scan implementation obtains and holds KVM's `mmu_lock`. While this lock is held, KVM might not be able to process a page fault caused by the virtual machine. To determine if this is the cause of the performance degredation, we modified the pre-scan implementation so that it releases the lock if it is contended. This did not eliminate the performance degradation. It is possible that the degradation is due to the use of many atomic instructions, however, we have not determined if this is the case.

Our implementation may lead to deviations of the intended checkpointing interval and the actual interval. Indeed, we find this is the case for small intervals. For SPECjbb and 50 ms we observe that the average interval length is approximately 20% larger than 50 ms and approximately 40% larger for pre-scan+sparse. The median interval is within 3% of 50 ms without sparse support and approximately 20% with it.

The reason for the increased interval length is that, in our implementation, pre-scan is performed only after concurrent-copy has completed. The write intensity of SPECjbb is high. As a result, too much of the interval is dedicated to concurrent-copy. Additionally, the length of the pre-scan also increases with the intensity. For one iteration it is approximately twice the reduction of the downtime. During the later stages of SPECjbb, the pre-scan can take 20 ms alone. Together with concurrent-copy it is impossible to maintain the intended checkpointing interval.

5.4 Parallel Copy

We designed parallel copy in the hopes of decreasing the number of CoW cases and thereby improve performance, as well as decrease the impact of checkpointing on the execution of the virtual machine.

Figure 5.5 portrays the average number of CoW cases per checkpoint. It shows that parallel copy is able to reduce the number of CoW cases. Because SPECjbb causes more CoW cases, the impact of parallel copy on SPECjbb is bigger than that on the kernel build. However, the impact on SPECjbb is also higher in relative terms. Figure 5.6a shows this. For kernel builds, the number of CoW cases is reduced by 10% to 30% in most scenarios. For SPECjbb, the reduction is between 40% and 70%. It makes sense that the kernel build is less affected because its distribution of CoW page faults is more skewed toward the end of the downtime.

Using more copy threads lowers the number of CoW cases. Averaged over all scenarios, using 8 copy threads reduces the number of CoW cases by another 12% compared to 4 copy threads.

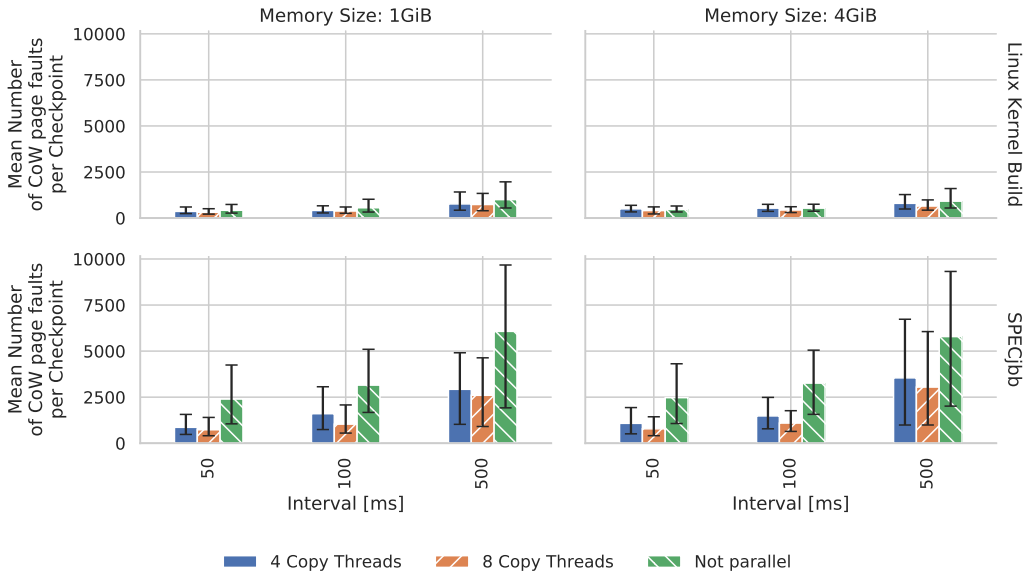
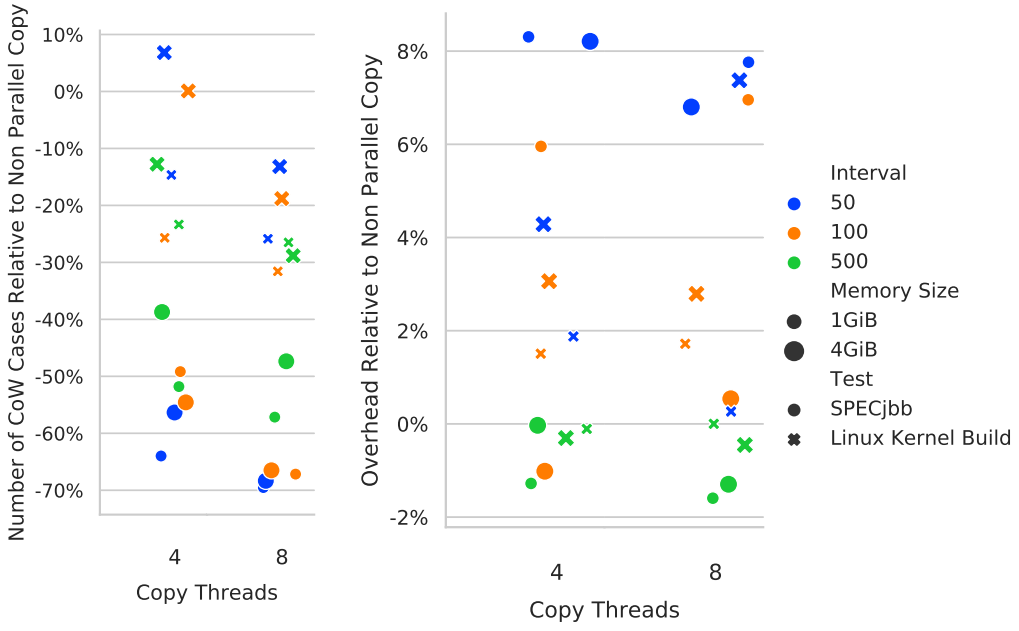


Figure 5.5: Mean number of CoW cases per checkpoint. The error bars show the range from the 5th to the 95th percentile. Parallel copy can reduce the number of CoW cases, this is more pronounced for SPECjbb, which causes more CoW cases to begin with.

Parallel copy is supposed to reduce the number of CoW cases by shortening the time to complete concurrent-copy. However, Figure 5.7 shows that this is not the case. There appears to be a minimum copy time of around 25 ms, as a result it is increased if non-parallel concurrent-copy takes less than that. For the kernel build, parallel copy leads to greatly increased standard deviations. These observations lead us to believe that there is an error in our implementation. The number of pages copied via parallel copy is slightly increased compared with single-threaded copy. The increase is consistent with the reduction in CoW cases, it does not indicate that the parallel copy implementation copies the same page multiple times.

Since the number of CoW cases *is* reduced, the basic mechanism to reduce the number of CoW cases seems to be working. Figure 5.8 confirms this. It displays the timing of CoW page faults relative to the end of the downtime and compares single-threaded and parallel copy. Parallel copy removes CoW page faults occurring late in the interval. The figure does not show any anomalous behavior.

To evaluate the impact of parallel copy on the benchmark, we consider the overhead caused by it. Figure 5.6b shows the results, the values are calculated in the same manner as in Figure 5.4, lower numbers represent



(a) (b)

Figure 5.6: Figure 5.6a: The overhead caused by parallel copy, calculated as in Figure 5.4, lower is better. While parallel copy improves performance for interval lengths of 500 ms, it reduces performance for shorter interval lengths.

Figure 5.6b: Change in the number of CoW cases causes by parallel copy. Parallel copy reduces the number of CoW cases by more than 40% for SPECjbb in all cases, for the kernel build it is less than that. The reduction caused by 8 threads is slightly higher than that of 4.

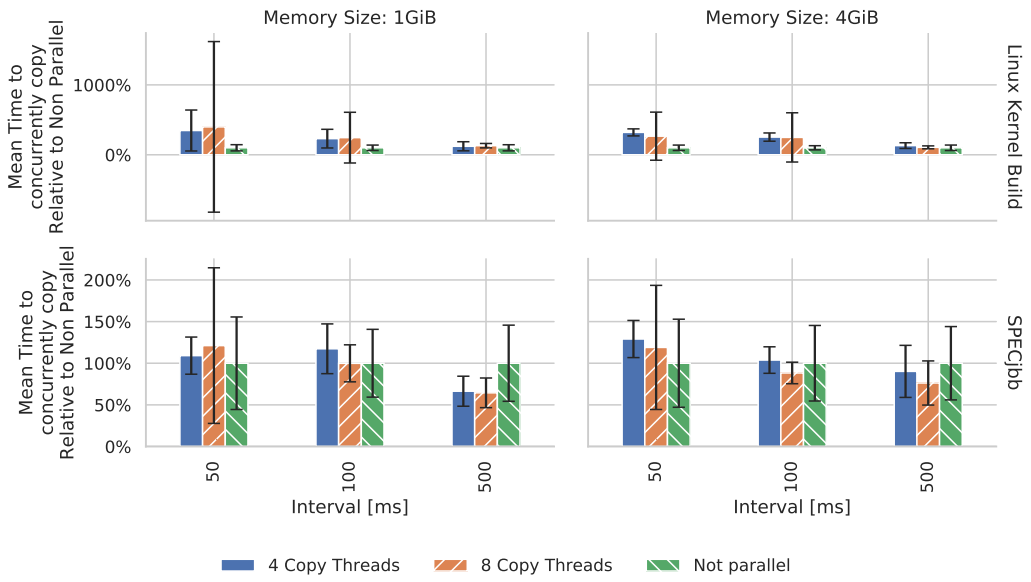


Figure 5.7: Mean time to perform concurrent-copy per checkpoint, relative to checkpointing without parallel copy. The error bars show the empirical standard deviation. Especially for 8 threads and the kernel build, the standard deviation is greatly increased, this is caused by very high outliers.

less overhead. While parallel copy is able to improve performance in some scenarios, the increase is minor. For short intervals, parallel copy degrades performance, it causes overheads as high as 8%. For 500 ms, it improves performance by 0.7%, averaged over all scenarios.

Besides potential implementation errors, parallel copy's decrease in performance is possibly due to the influence of the more intensive concurrent-copy process on the execution of the virtual machine. For example, this could occur via shared memory buses, caches and effects on the processor's frequency. This is, however, hard to quantify.

We evaluated if using userspace supplied buffers for CoW affects performance and did not find that it does.

5.5 Large Pages

We expected the use of large pages to substantially increase the amount of memory to capture. Figure 5.9 shows that this is indeed the case. Even for a target interval length of 50 ms, more than 500 MiB is copied, the increase compared to small pages is tenfold. As the target interval increases, the ratio decreases, at 1 s it is less than four.

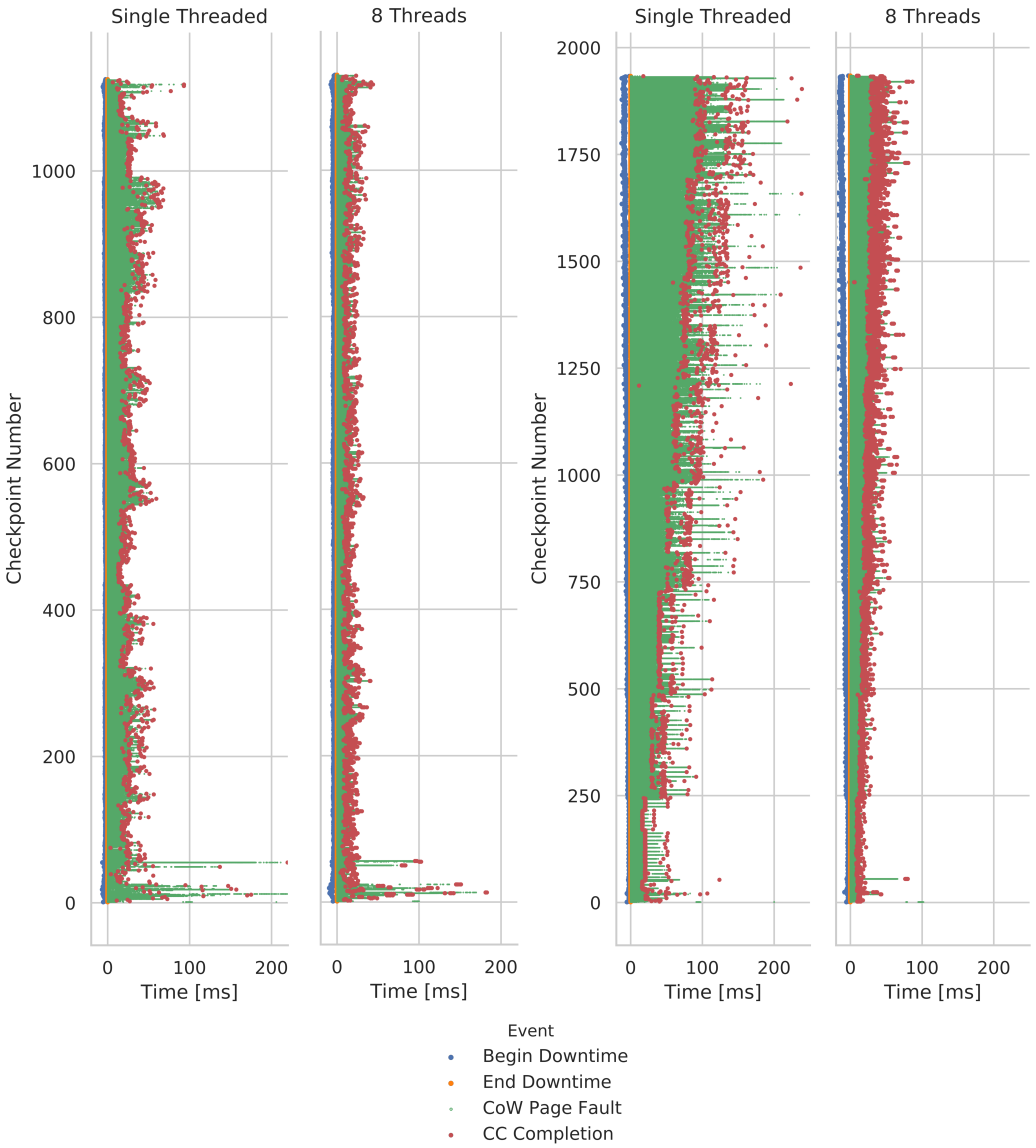


Figure 5.8: Comparison between the occurrence of CoW page faults of single and multithreaded concurrent-copy. The left side shows the kernel build, the right SPECjbb, interval length 1 s, 4GiB memory. The horizontal axis denotes time since the end of the downtime period. The figure shows that parallel copy suppresses late CoW page faults.

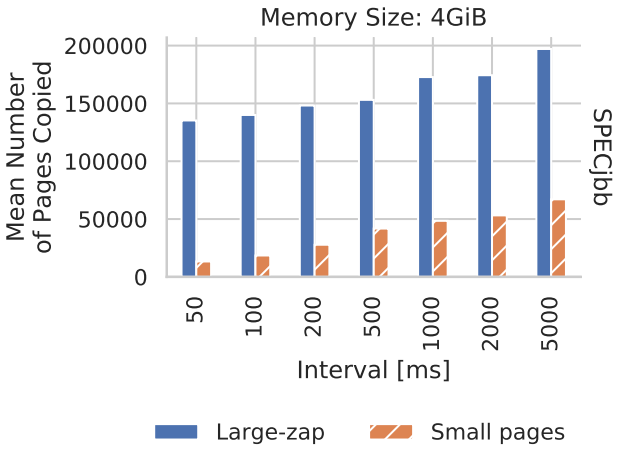


Figure 5.9: Mean number of pages copied per checkpoint. Checkpointing with large pages shows a greatly increased amount of memory to be copied, between 3 and 10 times more than when small pages are used.

As a result SimuBoost is not able to actually adhere to the intended interval length for intervals smaller than 500 ms. This makes the performance comparison harder, because the scores of the target intervals cannot directly be compared. Instead Figure 5.10 plots the mean observed interval length on the horizontal axis. The figure shows that checkpointing with large pages performs worse for interval up to and including 1 s. For 2 s and 5 s performance is better. At 5 s interval length SPECjbb's score is 4% higher when large pages are used. It makes sense that large pages can eventually improve performance. They increase the cost of creating a checkpoint, but as the checkpointing frequency decreases that ceases to matter and the positive effect of large pages prevails.

5.6 Conclusion

Pre-scan succeeded in lowering the downtime close to that of write-protection but affected performance of short intervals negatively. The exact reason for this has not been identified. We have shown that iterating pre-scan does not decrease the downtime further. As a result the conceptually more complex pipelined pre-scan can be abandoned. While parallel copy could reduce the number of CoW page faults, this did not improve performance. The evaluation showed anomalies that indicate an error in the implementation of parallel copy.

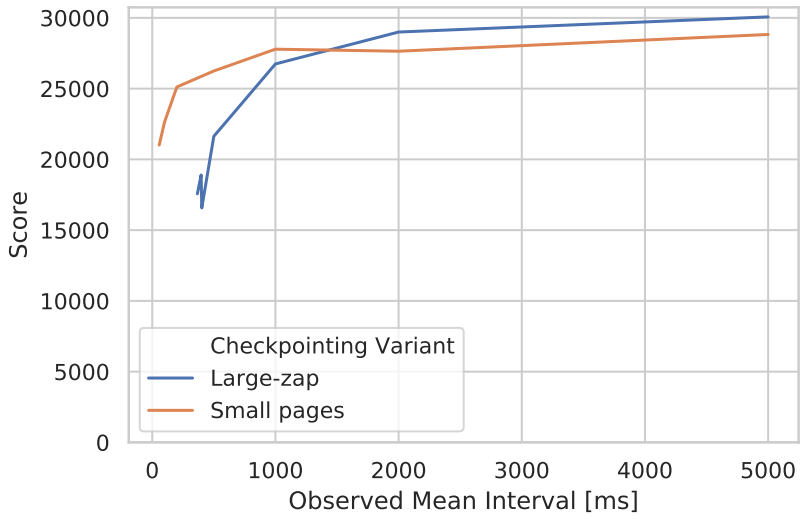


Figure 5.10: Score of SPECjbb, higher is better. The horizontal axis shows the mean observed interval length instead of the targeted interval length. For small intervals, checkpointing with large pages has lower performance than checkpointing with small pages. For 2 s and 5 s, performance is improved by 5% and 4% respectively.

For the simple design we implemented, large pages cause too much memory to be copied to be able to improve performance for interval lengths up to and including 1 s. For longer intervals, we observe a small performance improvement.

Chapter 6

Conclusion & Future Work

We have analyzed SimuBoost's impact on the virtual machine running the workload: SimuBoost causes downtimes, whose length is exacerbated by the scanning dirty logging mechanism. It also induces copy-on-write page faults, which cause overhead during the execution of the virtual machine. Additionally, SimuBoost does not make use of large pages, reducing performance compared to the baseline, which does.

We have explored pre-scan to reduce the downtime when scanning, parallel copy to avoid CoW page faults and investigated if the use of large pages makes sense for checkpointing. We found that pre-scan lowers the downtime but can negatively impact the performance of the virtual machine. To investigate why exactly pre-scan degrades performance and if this can be avoided remains future work. Because the pre-scan takes place after the previous checkpoint has completed, the current pre-scan implementation increases the smallest possible checkpointing interval. In order to avoid this, the implementation must be modified. Instead of performing pre-scan after the completion of concurrent-copy, the two should run simultaneously. One way to do this is to execute pre-scan in a separate thread, starting it at the same point in time as in the current implementation. However, if concurrent-copy completes after the pre-scan, there is a gap between pre-scan and the next downtime, potentially decreasing the effectiveness of pre-scan in reducing the downtime. This could be mitigated by predicting the duration of concurrent-copy as well as that of pre-scan. The next interval length is then set to the maximum of predicted duration of concurrent-copy, pre-scan and the intended interval length. After both concurrent-copy and pre-scan have finished, the next downtime commences.

A further way to improve the accordance with the indented interval is to make pre-scan interruptible. If a pre-scan takes too long, the thread executing it is signaled and the pre-scan is aborted. This could be implemented

similar to the lock release we implemented in our search for the reason of the performance degradation. Instead of checking if a lock is contended, pre-scan would check if the interrupt signal has arrived.

To decrease its duration, pre-scan could also be parallelized by splitting up the guest physical address space and having multiple threads scan the different regions. The current implementation could be extended in this way quite easily, it already supports scanning only a region of the guests memory but does not expose this ability to userspace.

Parallel copy reduces the number of CoW page faults without improving performance. There are anomalies in our implementation that need to be addressed in order to be able to conclude if parallel copy can improve performance or not. Another possibility to reduce the number of CoW page faults is to copy the pages most likely to incur a CoW page fault during the downtime. Pre-scan could provide a heuristic to identify such pages. The scan during the downtime finds pages dirtied since the pre-scan, that is, it finds pages written to shortly before the downtime. Because of temporal and spacial locality, these pages are likely to be written again after the downtime. Capturing these pages during the downtime would prevent CoW page faults and therefore a VM exit and entry. Since there are likely few such pages the downtime would not increase substantially.

Large pages improve performance for long interval only. The increase in dirty memory is a major hurdle to incorporating large pages into checkpointing while gaining performance for small intervals. A more complex design would be required to achieve this. Such a design could for example only use large pages for memory regions it predicts to be very dirty.

It would be interesting to see how parallel copy interacts with large pages. If large pages are broken by CoW page faults and re-established after the completion of concurrent-copy, parallel copy could increase the benefit of large pages by extending the duration large pages are available for. It may also somewhat counter the higher amount of memory that must be captured. Alternatively, more memory to capture could raise the overhead of parallel copy.

Bibliography

[1] KVM. http://www.linux-kvm.org/page/Main_Page, April 11 2016.

[2] Phoronix test suite. http://www.phoronix-test-suite.com/, October 31 2016.

[3] QEMU. http://wiki.qemu.org/Main_Page, April 11 2016.

[4] Avocado testing framework. https://avocado-framework.github.io/, November 30 2018.

[5] avocado-virt. https://github.com/avocado-framework/avocado-virt, November 30 2018.

[6] iperf. https://iperf.fr, November 08 2018.

[7] iperf. https://iperf.fr/iperf-doc.php, November 08 2018.

[8] Specjbb. https://www.spec.org/jbb2005/, November 30 2018.

[9] Advanced Micro Devices, Inc, One AMD Place, Sunnyvale, California, U.S. *AMD64 Architecture Programmer's Manual*, 3.30 edition, September 2018.

[10] Erik R. Altman and Erik R. Welcome to the opportunities of binary translation. *Journal of Parallel and Distributed Computing*, 16:271–275, 2000.

[11] Nikolai Baudis. Deduplicating virtual machine checkpoints for distributed system simulation. Bachelor thesis, System Architecture Group, Karlsruhe Institute of Technology (KIT), Germany, November 2 2013. http://os.ibds.kit.edu/.

[12] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association. http://dl.acm.org/citation.cfm?id=1247360.1247401.

[13] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. *SIGPLAN Not.*, 43(3):26–35, March 2008. `http://doi.acm.org/10.1145/1353536.1346286`.

[14] Nikhil Bhatia. Performance evaluation of intel ept hardware assist. Technical report, VMware Inc, 3401 Hillview Ave., Palo Alto, CA, 2009.

[15] Nico Boehr. Evaluating copy-on-write for high frequency checkpoints. Bachelor thesis, Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, September 30 2015.

[16] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association. `http://dl.acm.org/citation.cfm?id=1251203.1251223`.

[17] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 161–174, Berkeley, CA, USA, 2008. USENIX Association. `http://dl.acm.org/citation.cfm?id=1387589.1387601`.

[18] Jakob Engblom. Full-system simulation technology : Extended abstract appearing in the proceedings of esses 2003 (european summer school on embedded systems), 2003.

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

[20] Alexander Graf, Alex Bennée, Alexey Kardashevskiy, Alex Williamson, Anatol Pomozov, Andre Przywara, Andrew Jones, Andrey Smetanin, Anup Patel, Aravinda Prasad, Avi Kivity, Benjamin Herrenschmidt, Bharat Bhushan, Borislav Petkov, Brijesh Singh, Carlos Garcia, Carsten Otte, Christian Borntraeger, Christoffer Dall, Claudio Imbrenda, Cornelia Huck, David Gibson, David Hildenbrand, Dominik Dingel, Dongjiu Geng, Drew Schmitt, Ekaterina Tumanova, Eric Auger, Eric B Munson, Eric Farman, Fan Zhang, Gabriel L. Somlo, Geoff Levand, Gleb Natapov, Greg Kurz, James Hogan, James Morse, Jan Kiszka, Jann Horn, Janosch Frank, Jason J. Herne, Jason Wang,

Jens Freimann, Jim Mattson, Ken Hofsass, Linu Cherian, Linus Torvalds, Liu Yu-B13201, Luiz Capitulino, Marcelo Tosatti, Marc Zyngier, Masanari Iida, Michael Ellerman, Michael Neuling, Michael S. Tsirkin, Mihai Caraman, Nadav Amit, Paolo Bonzini, Paul Mackerras, Radim Krčmář, Rob Landley, Roman Kagan, Rusty Russell, Sasha Levin, Scott Wood, Shannon Zhao, Stefan Huber, Steve Rutherford, Takuya Yoshikawa, Thomas Huth, Tiejun Chen, Tom Lendacky, Vitaly Kuznetsov, Vladimir Murzin, Wanpeng Li, Xiao Guangrong, and Yi Min Zhao. Documentation/virtual/kvm/api.txt. 84df9525b0c27f3ebc2ebb1864fa62a97fdedb7d `git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git`.

[21] IEEE/The Open Group. *MLOCK(3P) POSIX Programmer's Manual*, posix.1-2008 edition, 2013.

[22] Intel Corporation. *Intel 64 and IA-32 Architecture Software Developer's Manual*, April 2016.

[23] Mateusz Jurczyk. Bochspwn revolutions further advancements in detecting kernel infoleaks with x86 emulation. `https://j00ru.vexillium.org/slides/2018/infiltrate.pdf`, 2018.

[24] Mateusz Jurczyk. Detecting kernel memory disclosure with x86 emulation and taint tracking. Technical report, Google LLC, June 2018.

[25] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. Kvm: the linux virtual machine monitor. In *In Proceedings of the 2007 Ottawa Linux Symposium (OLS'-07)*, 2007.

[26] Avi Kivity, David Matlack, Linus Torvalds, Liran Alon, Masanari Iida, Paolo Bonzini, Peter Feiner, Rob Landley, Takuya Yoshikawa, and Xiao Guangrong. Documentation/virtual/kvm/mmu.txt. 84df9525b0c27f3ebc2ebb1864fa62a97fdedb7d `git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git`.

[27] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, February 2002. `http://dx.doi.org/10.1109/2.982916`.

[28] Timothy Merrifield and H. Reza Taheri. Performance implications of extended page tables on virtualized x86 processors. In *Proceedings of*

the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '16, pages 25–35, New York, NY, USA, 2016. ACM. http://doi.acm.org/10.1145/2892242.2892258.

[29] Robert B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), pages 267–277, New York, NY, USA, 1968. ACM. http://doi.acm.org/10.1145/1476589.1476628.

[30] Ingo Molnar, Kirill Smelkov, and Jiri Olsa. tools/perf/documentation/perf.txt. git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git.

[31] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. Large pages and lightweight memory management in virtualized environments: Can you have it both ways? In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 1–12, New York, NY, USA, 2015. ACM. http://doi.acm.org/10.1145/2830772.2830773.

[32] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974. http://doi.acm.org/10.1145/361011.361073.

[33] Marco Righini. Enabling intel® virtualization technology features and benefits. Technical report, Intel Incorporation, 2010.

[34] 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.

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

[36] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen A. Herrod. Using the simos machine simulator to study complex computer systems. *ACM TRANSACTIONS ON MODELING AND COMPUTER SIMULATION*, 7:78–103, 1997.

[37] Mendel Rosenblum and Mani Varadarajan. Simos: A fast operating system simulation environment. Technical report, Stanford, CA, USA, 1994.

[38] Steven Rostedt. trace-cmd. `git://git.kernel.org/pub/scm/linux/kernel/git/rostedt/trace-cmd.git`.

[39] Jan Ruh. Optimizing continuous checkpoints for deterministic replay. Master thesis, Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, July 15 2018.

[40] Janis Schoetterl-Glausch. Intel page modification logging for lightweight continuous checkpointing. Bachelor thesis, Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, October 31 2016.

[41] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[42] Michael H. Sun and Douglas M. Blough. Fast, lightweight virtual machine checkpointing. Technical report, Georgia Institute of Technology, 2010.

[43] Yoshiaki Tamura, Koji Sato, Seiji Kihara, and Satoshi Moriai. Kemari: Virtual machine synchronization for fault tolerance using domt. Technical report, NTT Cyber Space Labs, 2008.

[44] O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011. `http://www.gnu.org/s/parallel`.

[45] Al Viro, Arnd Bergmann, Christoph Hellwig, David Howells, Frederic Weisbecker, Harvey Harrison, Kees Cook, Linus Torvalds, Luis R. Rodriguez, Masanari Iida, Mauro Carvalho Chehab, Paul Gortmaker, Randy Dunlap, Rusty Russell, Shawn Bohrer, and Tobias Klauser. Documentation/docbook/kernel-hacking.tmpl. 84df9525b0c27f3ebc2ebb1864fa62a97fdedb7d `git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git`.

[46] Dirk Vogt, Armando Miraglia, Georgios Portokalidis, Herbert Bos, Andy Tanenbaum, and Cristiano Giuffrida. Speculative memory checkpointing. In *Proceedings of the 16th Annual Middleware Conference*,

Middleware '15, pages 197–209, New York, NY, USA, 2015. ACM. http://doi.acm.org/10.1145/2814576.2814802.

[47] Long Wang, Zbigniew Kalbarczyk, Ravishankar Iyer, and Arun Iyengar. Vm-μcheckpoint: Design, modeling, and assessment of lightweight in-memory vm checkpointing. *Dependable and Secure Computing, IEEE Transactions on*, 12:243–255, 03 2015.

[48] Felix Wilhelm. Tracing privileged memory accesses to discover software vulnerabilities. Master thesis, Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, November 30 2015.

[49] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '96, pages 68–79, New York, NY, USA, 1996. ACM. http://doi.acm.org/10.1145/233013.233025.

[50] Huang Ying and Mike Rapoport. Documentation/vm/transhuge.rst. 84df9525b0c27f3ebc2ebb1864fa62a97fdedb7d git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git.

[51] Matt T. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *ISPASS '07*, 2007.