Karlsruher Institut für Technologie

Intel Page Modification Logging for Lightweight Continuous Checkpointing

Bachelorarbeit
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. Juli 2016 – 31. Oktober 2016

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

Karlsruhe, den 31. Oktober 2016

Abstract

The usefulness of full system simulation is limited by its low execution speed. SimuBoost is a technique seeking to remove this limitation by accelerating full system simulation via parallelization. It builds on hardware-assisted virtualization and frequently creates checkpoints of the virtual machine, which serve as starting points for a distributed parallel simulation. SimuBoost needs lightweight checkpointing. To achieve this, it uses incremental checkpointing, which requires that the virtual machine monitor is informed of changes the guest makes to memory. Traditionally, this dirty logging is implemented via write protection; when active, a write access generates a page fault informing the monitor.

This thesis explores whether SimuBoost's incremental implementation can be made more lightweight by Intel Page Modification Logging, a technology with the aim of reducing the cost of dirty logging.

Deutsche Zusammenfassung

Diese Arbeit untersucht, inwiefern Intel Page Modification Logging die Geschwindigkeit des kontinuierlichen Erstellens von Abbildern einer virtuellen Maschine steigern kann. Diese Betrachtung erfolgt mit Hinsicht auf SimuBoost, einem Verfahren, das dazu dient, die umfassende Simulation eines Rechensystems zu beschleunigen. Solch eine Simulation ist ein mächtiges Werkzeug, um das Verhalten eines Systems im Detail zu beobachten, hat aber den Nachteil, dass die Ausführung stark verlangsamt wird. Diesen Nachteil beseitigt SimuBoost durch die Paralleliserung der Simulation. SimuBoost benutzt dafür hardwarebeschleunigte Virtualisierung; Abbilder der virtuellen Maschine, die in geringen Intervallen getätigt werden, dienen als Ausgangspunkte der parallelen Simulation. Um den Aufwand zum Erstellen eines Abbilds gering zu halten und somit das Verhalten des zu analysierenden Systems möglichst wenig zu verzerren, erstellt SimuBoost Abbilder inkrementell. Dies setzt voraus, dass es über Änderungen, die das in der virtuellen Maschine laufende Programm am Speicher vornimmt, informiert wird. Eine solche Überwachung des Speichers verlangsamt die Ausführung. Intel Page Modification Logging (PML) zielt darauf ab, sie zu beschleunigen. Diese Arbeit reichert die existierende SimuBoost Implementierung um PML an und evaluiert, ob der Einfluss des Erstellens der Abbilder dadurch vermindert werden kann. Dazu werden zwei mögliche Implementierungen mit einer Reihe von Tests in ihren Eigenschaften und Verhalten verglichen. Die Evaluation ergibt, dass eine Verbesserung durch PML nur für kleine Intervalle möglich ist, bei 100ms beträgt die Beschleunigung circa 9%, für eine Sekunde nur 0.5%.

Contents

Chapter 1

Introduction

This thesis explores the viability of using Intel Page Modification Logging to improve the performance of continuous virtual machine checkpointing. It evaluates the chosen approach in the larger context of SimuBoost, a technique for accelerating functional full system simulation.

Functional full system is a powerful tool to inspect the behavior of a computing system. Its compelling capabilities, achieved by fine-grained emulation, come at the cost of greatly reduced speed of execution, limiting its usefulness.

SimuBoost is able to accelerate full system simulation by parallelization. It builds on hardware-assisted virtualization and requires lightweight checkpointing. In order to minimize the downtime and slowdown caused by checkpointing, SimuBoost employs incremental copy-on-write checkpointing. Incremental checkpointing requires the virtual machine manager to be informed of changes to the memory of the virtual machine. This dirty logging, however, slows down execution. Intel introduced an extension — Page Modification Logging (PML) — to its virtualization support in order to mitigate the slowdown caused by dirty logging. This work incorporates PML as a dirty logging mechanism into SimuBoost.

The evaluation compares the behavior of two possible implementations based on a series of benchmarks. It finds that PML can decrease the overhead for small intervals only. At 100ms it causes an acceleration by 9%, at 1s it is 0.5%. Additionally, this thesis contributes an analysis and model of the cost of dirty logging.

The thesis is structured as follows: Chapter 2 provides background on full system simulation, virtualization and SimuBoost. Chapter 3 analyzes the impact of dirty logging and estimates the benefit of PML for checkpointing. Chapter 4 describes the design and implementation of the modifications to SimuBoost. After evaluating the approach in Chapter 5, Chapter 6 summarizes the results of this thesis and provides possible future work.

Chapter 2

Background

The following chapter provides background on full system simulation and a method, called SimuBoost, to accelerate it. Further, it explains hardware-assisted virtualization and virtual machine checkpointing, which the SimuBoost concept is based on. Then, it describes QEMU and KVM, upon which the current implementation of SimuBoost builds.

2.1 Full System Simulation

Full System Simulation (FSS) is a technology which is used to simulate a whole computing system on a host computer. A FSS does not only simulate the processor (CPU) but also devices and memory, such that it is possible to run an existing complete software stack without having to modify it [15]. Some full system simulators are able to simulate a network of multiple computers [22]. In this thesis, FSS is understood to work by using a form of emulation, for example interpretation [22] or dynamic binary translation [9].

In general, simulations are based on *models* of systems so that an analysis of the model's behavior can be generalized to that of a real system [15]. The models used in FSS to emulate the system offer different *levels of abstraction*. Some work on a *functional* level and model the system at the instruction set level. *Cycle accurate* simulators additionally model the timing behavior of a CPU. Even more fine grained models function at a *micro-architectural* level. Microarchitectural and cycle accurate simulations are slower than functional ones, which limits the *scope*, that is the size of the system that can be simulated with a realistic effort [15, 22, 33]. By virtue of being done in software, simulation offers a number of benefits over running a workload directly on hardware: the simulation can easily be parameterized and the simulated system can be completely controlled and inspected [22]. FSS is therefore useful

in a number of academic and industrial cases [15, 22], including:

- Development of software prior to the release of hardware.
- Development of computer components and systems. FSS can be used for easy prototyping.
- Development and research of operating systems (OS). For example, the inspection capability of FSS can be used to trace memory accesses. Miller et al. used FSS to study the properties of redundant memory content to improve deduplication with memory scanners [23].
- Security research. Bochspwn [19] uses the open source x86 emulator Bochs to analyze memory access patterns and thereby identify exploitable race conditions in the Windows kernel. A similar study revealed critical vulnerabilities in the Xen hypervisor [32].

A disadvantage of FSS, however, is that the emulation causes an increase in the execution time of a workload; slowdowns by factors of 30-800 are common [28]. This slowdown limits the usefulness of FSS for analyzing properties of long-running workloads, and it practically prohibits interactive communication with the nonsimulated environment, be it an user or remote host.

2.2 SimuBoost

SimuBoost [28] is a concept to speed up functional FSS and therefore eliminate the limitations of traditional FSS. The reason for the slowness of traditional FSS is that the emulation of an operation is slow compared to its native execution and that it operates on a state which is itself the result of a slow emulation. SimuBoost must break this dependency chain in order to be able to simulate the workload in *parallel*. SimuBoost accomplishes this by using much faster hardware-assisted *virtualization* as a way to *fast-forward* to a state:

The workload is run in a virtualized environment. In periodic intervals, SimuBoost creates a *checkpoint*, that is it saves the current state of the system (CPU, memory, disk, devices). These checkpoints are distributed to *simulation nodes* that run a simulation of the respective interval, starting with the checkpoint as initial state. Nondeterministically occurring events are recorded and replayed during FSS. The parallel execution of the simulations decreases the total simulation time. Rittinghaus et al. employ an analytical model to predict a speedup by a factor of 84 for a one-hour workload under realistic assumptions compared to traditional FSS [28].

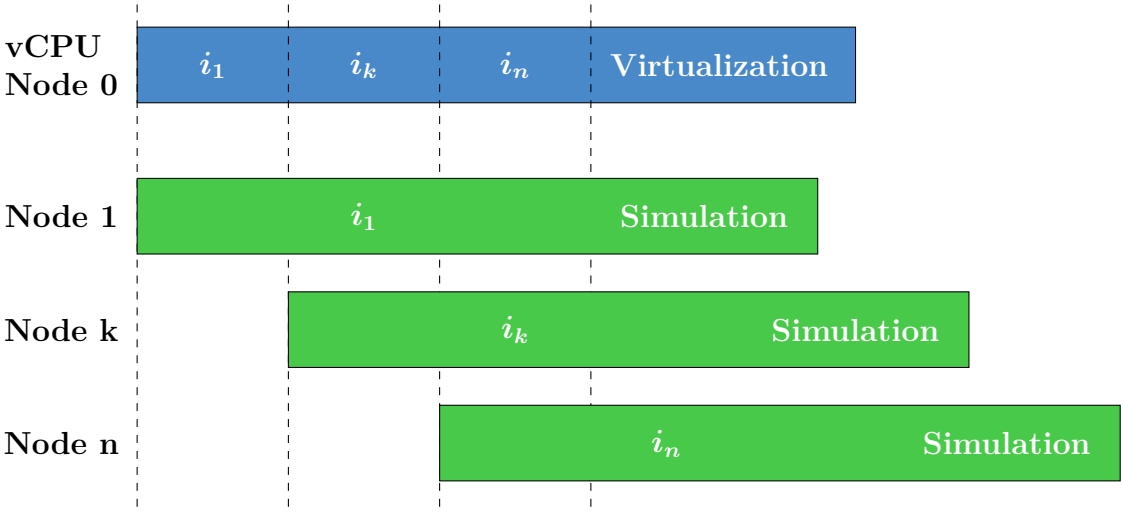

Figure 2.1: Acceleration of full system simulation via SimuBoost. The virtualization node runs the workload and creates checkpoints. Simulation nodes run simulations of intervals, with the respective checkpoint as starting state.

FSS is used to analyze workload behavior. For the analysis to be meaningful, FSS must incur minimal distortion of workload behavior. Creating a checkpoint requires stopping the VM for a *downtime* period. Some checkpointing methods require additional work to be performed during the interval between checkpoints, resulting in a *slowdown of execution*. The downtime and slowdown impact the timing of nonsimulated events. If an asynchronous I/O operation is started before a downtime and completed after, its length appears shorter to the guest than it really was. Additionally, outside entities can observe the guest's downtime, which distorts the results. For example, a remote host connected to the guest via TCP would observe an increase in the round trip time, changing the characteristics of the connection. If downtime and slowdown of execution are too high, they also limit interaction with a human user. For SimuBoost to enable interactivity and accurate simulation results, this means that the downtime during checkpointing and the slowdown of execution must be minimal. To this end, SimuBoost currently employs *incremental* [8] *copy-on-write (COW)* [10] checkpointing. The current SimuBoost implementation is based on QEMU [6] using KVM [3] and uses Simutrace [27] as a storage back end. The storage backend performs deduplication of frames and disk sectors in order to lower the checkpoint size [14].

2.3 Virtualization

A *virtual machine (VM)* is an *efficient, isolated duplicate* of a real machine [26]. It is an environment for programs, provided by a *virtual machine monitor (VMM)*, which approximates the original machine [26], that is programs run in the VM behave functionally identical. They may only incur a minor decrease in performance; Rittinghaus et al. observed a 14.3% increase in the time it takes to complete a Linux kernel build [28]. This can be achieved only if most of the program's instructions execute directly on the real processor without software intervention by the VMM [26]. Programs running in a VM must therefore necessarily use the same instruction set and are modeled on a functional level only.

The VMM is in complete control of system resources; programs running in the VM can only access resources allocated to them and the VMM can take back control over allocated resources [26].

For efficient virtualization to be possible, all *sensitive* instructions must be *privileged* instructions. Sensitive means that if the instruction was executed on the CPU, it could alter state outside the VM or its behavior would depend on outside state. A privileged instruction traps in user mode but not in system mode [26]. An example for a privileged x86 instruction is MOV when writing to a control register [18, ch. 3]. An example for a sensitive one is *Pop Stack into Flags Register (POPF)*, which ignores overwrites of the interrupt-enable flag if the real CPU is in user mode, regardless of the state of the virtual CPU. Its behavior, therefore, depends on state outside the VM. POPF is not privileged and thus poses a hindrance to virtualization. Nonprivileged, sensitive instructions are called *critical* [29, p. 391]. There are techniques to handle critical instructions, for example the VMM can scan the instructions about to be executed for critical ones and patch them to trap or jump back to the VMM. These techniques lead to increased code complexity and reduced performance [29, p. 436].

If all sensitive instructions are privileged, efficient virtualization can be implemented by *trap-and-emulate*. Here, nonprivileged instructions run on the real CPU and privileged ones are emulated by the VMM [26, 29].

Special considerations must be made regarding address translation. The guest OS maintains page tables for translating guest virtual to guest physical addresses. These page tables can not, however, be used by the hardware since the guest physical address space is not the same as the host physical address space. One way to solve this problem is using *shadow page tables*. The guest is left to believe that the page tables it is maintaining are used by the hardware. The hardware actually uses shadow page tables maintained by the VMM. When the VMM detects that the guest updated a page table, that

is the guest updated the translation of a guest virtual address `gva` to guest physical address `gpa`, it will resolve `gpa` to the host physical address `hpa` and update the shadow page table so `gva` is translated to `hpa` [20]. The VMM must take care to set the correct access flags. If the guest OS sets pages read only it expects writes to generate a page fault. The VMM must also mark the corresponding page entry in the shadow page table read only, so it can intercept and inject the page fault into the guest. As is to be expected, shadow paging incurs a performance overhead.

2.3.1 Hardware-Assisted Virtualization

To improve its virtualizability, x86 vendors have introduced extensions to the x86 instruction set.

Intel Virtual Machine Extensions (VMX) is Intel's variant of such an extension [18, ch. 23]. This thesis focuses on Intel — AMD's extensions work similarly [7, ch. 15]. VMX adds two modes of processor operation: VMX root operation and VMX non-root operation. The VMM runs in root operation, guest software runs in VMX non-root operation. When a sensitive instruction is executed in non-root mode, a VM exit, that is a transition to VMX root mode, occurs, informing the VMM of the exit reason, in order for the VMM to be able to emulate the instruction.

Extended Page Tables (EPT) are part of VMX and provide hardware support for translating guest virtual to guest physical to host physical addresses [18, ch. 28]. Without EPT, guest virtual addresses would have to be translated to guest physical addresses in software. EPT can include support for accessed and dirty flags for pages. These sticky flags are set by the processor when the respective event occurs. If software clears these bits, it must flush affected TLB entries, else they may not be set on subsequent accesses [18, 28.2.4].

Intel Page Modification Logging (PML) is an enhancement of EPT with the goal of improving the performance of dirty logging, that is the tracking of write accesses to guest physical pages. Without a feature like PML, a VMM instead has to use write protection to be notified via page faults. Doing so degrades performance.

When dirty flags for EPT and PML are enabled, write accesses to clean pages are recorded to a 512 entry long *page-modification log*. If the log is full, the CPU raises a *page-modification log-full event* as well as a VM exit [18, 28.2.5]. As a result, there is no page fault and subsequent VM exit for every page that is written to, but up to 512 writes to distinct pages can be consolidated into one VM exit.

2.4 Virtual Machine Checkpointing

To create a checkpoint of a virtual machine means to create a consistent image of the full state of the VM [30]. The state of the VM constitutes the state of the CPU, memory, and other devices such as disks, network adapters, etc. VM checkpointing is a subset of VM migration, where the state of a VM is transferred to another physical machine so execution can resume there. VM migration is done to perform load-balancing and system maintenance, as well as accomplish fault tolerance [11, 12]. If checkpointing is performed periodically with high frequency (in the order of seconds or lower), it is called *continuous checkpointing*.

SimuBoost [28] requires continuous checkpointing to achieve highly parallel FSS. Remus [12] and Kemari [31] use continuous checkpointing for fault tolerance, they continuously replicate the state of a VM to another physical machine to accomplish failover without disruption of service.

With memory sizes in the order of tens of gigabytes and disk sizes in the order of hundreds to thousands of gigabytes, a checkpoint encompasses a lot of data that has to be saved or transmitted [30]. Different techniques exist to handle this amount of data while achieving high performance. These techniques differ in their characteristics, notably they result in different *downtimes* and *slowdowns of execution*.

While the VM is running, its state changes constantly and the state of some devices, for example that of the CPU, changes with very high frequency. For this reason, the VM must be stopped once per checkpoint so a consistent snapshot of the VM can be taken. The time that the VM is suspended is called *downtime* [11]. For a user to perceive the response of an action like a key press to be instantaneous, the delay must be no longer than 100-200ms [24], which means the downtime must be less than this to preserve interactivity.

The slowdown of execution is the negative effect a checkpointing method may have on performance outside of the downtime period. Some checkpointing techniques involve keeping track of memory the VM has written to. One way to perform this dirty logging is to write protect all pages and mark them dirty when a page fault occurs. This leads to additional VM exits and work done by the VMM causing a slowdown of execution depending on how much the workload writes to memory.

Stop-and-copy checkpointing is the simplest checkpointing method. The VM is stopped and the VMM saves all state during the downtime. As a result, stop-and-copy causes a long downtime, it is proportional to the amount of data that needs to be saved [11]. Stop-and-copy does not result in a slow-

down of execution, because no additional work is done in the interval between downtimes.

Pre-copy checkpointing moves the majority of saving/transmitting before the downtime period, thereby achieving much better downtimes compared to stop-and-copy [11, 12]. Pre-copy is a method commonly used in *live migration*.

Pre-copy saves the memory of the VM iteratively in *rounds* while the VM is still running. In each round, the VMM concurrently saves those pages that were dirtied in the previous round. All pages are saved in the first round. If the VM dirties less pages during a round than those that have to be saved in this round, the next round will be shorter, giving the workload less time to dirty pages. The process is continued until it converges or a fixed number of iterations is reached. The VMM saves the pages written to in the last round, as well as the state of the CPU and devices, using stop-and-copy [11, 12, 30].

Pre-copy achieves substantially shorter downtimes proportional to the write intensity of the workload, typically they are under 100ms [11, 12]. If saving is done in a separate thread which does not compete with those of the VM, the slowdown of execution is that caused by dirty logging.

Incremental checkpointing is a possibility of avoiding work which is applicable when doing continuous checkpointing. Since checkpoints are created often and with little time in between, the memory and disk content of consecutive checkpoints will largely be the same. Incremental checkpointing saves only those pages that did change and is equivalent to performing just the last step of pre-copy for every checkpoint, that is an incremental stop-and-copy [12].

Baudis [8] estimates that incremental checkpointing reduces the amount of data to be saved to 5-10%; Böhr [10] measured a downtime of 84ms for incremental stop-and-copy during a Linux kernel build [1]. As with pre-copy, the dirty logging required for incremental checkpointing introduces a slowdown of execution.

Copy-on-write (COW) checkpointing shifts the majority of checkpoint creation to after the downtime period.

Initially, the VMM suspends the VM and saves the state of the CPU, as well as that of devices. Then, it write protects all pages and resumes the execution of the VM. While the VM continues to run, the VM's memory is saved concurrently. If the VM tries to write to a page that has not yet

[1] 1GiB RAM, 2s checkpointing interval

been saved, the write protection will cause a page fault handing control back to the VMM, which saves the respective page and resumes execution of the VM. The checkpoint is complete when all pages have been saved either by concurrent copy (CC) or COW [12, 30].

COW checkpointing attains small downtimes [12]. The downtime is still proportional to the amount of memory, because the VMM must write protect all pages, but by a much smaller factor. The slowdown of execution depends on the rate of COW cases.

COW checkpointing can be combined with incremental checkpointing, so only dirty pages are saved. The current SimuBoost version implements this combined approach, for which Böhr [10] gives a downtime of 26ms during a Linux kernel build [2].

2.5 QEMU & KVM

QEMU [6] is a free software full system simulator. QEMU supports simulation of multiple instructions sets [3]. It uses dynamic binary translation to achieve high performance and is easily portable [9]. QEMU supports virtualization via Kernel-based Virtual Machine (KVM) [3], a virtual machine monitor driver in the Linux kernel that exposes hardware-assisted virtualization to QEMU in userspace. QEMU interfaces with KVM to create, run and manage VMs [20]. To do so, QEMU opens the `/dev/kvm` device node and interacts with it via the `ioctl()` syscall. KVM provides a number of `ioctl`s, including ones that perform the following functions [20]:

- Creation of a new VM
- Creation of a virtual CPU for the VM
- Assigning physical memory to the VM
- Running a virtual CPU

[2] 1GiB RAM, 2s checkpointing interval

[3] including x86, PowerPC, ARM and Sparc [9]

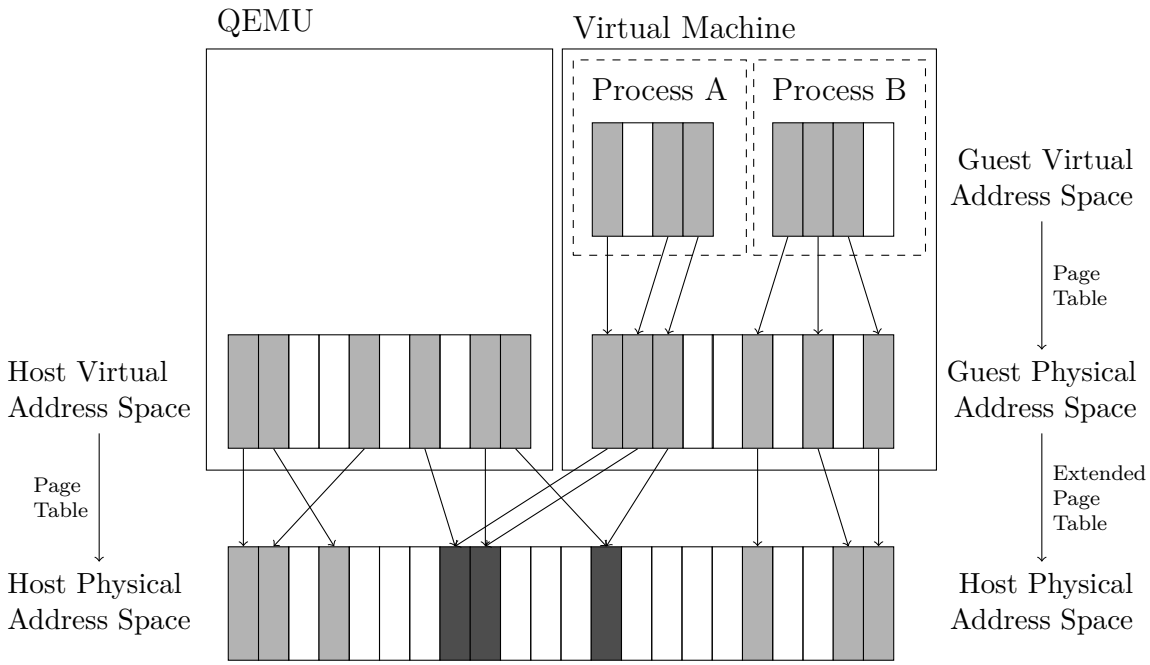

Figure 2.2: The KVM memory setup [20]. Guest virtual addresses are translated to host physical addresses. These translations are managed by KVM using two-dimensional paging if available (as shown), shadow page tables otherwise. QEMU can control the memory of the virtual machine by directing KVM to map the physical memory backing some region of QEMU's virtual address space into the physical address space of the guest (■).

The guest's memory is separate, but accessible from that of the process which created it [16,20]. The translation from guest physical to host physical addresses is managed by KVM, using two-dimensional paging like EPT if available and shadow page tables otherwise [20, 21]. QEMU's independent translations are maintained by the OS. KVM does not manage guest I/O, but leaves this part to a userspace VMM like QEMU. A typical structure for such a VMM, then, is to setup the VM and, in a loop:

- have KVM run the virtual CPU
- upon return of the `run ioctl` inspect the exit reason and handle it

Chapter 3

Analysis

SimuBoost relies on virtual machine (VM) checkpointing in order to accelerate functional full system simulation (FSS). The following chapter analyzes how dirty logging changes the behavior of the VM, as well as how the performance of checkpointing might be improved by Intel Page Modification Logging (PML).

When creating a checkpoint, all information necessary to restore the state of the VM must be saved. The simplest checkpointing technique, stop-and-copy, stops the VM and saves all state in the resulting period of *downtime*. Due to its long downtime of up to multiple seconds, stop-and-copy is unsuitable for SimuBoost. Short downtimes can be achieved by shifting work outside the downtime period, thereby performing it concurrently to VM execution. Examples for such techniques are pre-copy and incrementel copy-on-write checkpointing, which require keeping track of dirty pages, that is they need to know which pages the VM has modified during a certain interval. Traditionally, this is done by write protecting all pages. When the VM tries to write to a page, a page fault occurs and the virtual machine monitor (VMM) marks the page dirty before allowing the write. This additional VM exit, processing and VM entry causes a *slowdown of execution* of the VM. To alleviate this overhead, Intel introduced the PML extension to its virtualization support. When PML is enabled, the hardware logs write accesses to a region in memory. Instead of exiting the VM for every write access, PML does so only when the log is full, therefore incurring less overhead.

3.1 Impact of Dirty Logging

Estimating the benefit of using PML for incremental checkpointing requires an understanding of how dirty logging impacts the performance of a workload

and how PML compares against write protection. In order to quantify the impact of dirty logging, we added a command to QEMU that periodically queries the dirty bitmap. This is done in a separate thread and with a configurable interval between queries. The following benchmark was run with a four core Intel i5-6500 as host CPU. Figure 3.1 shows the impact of

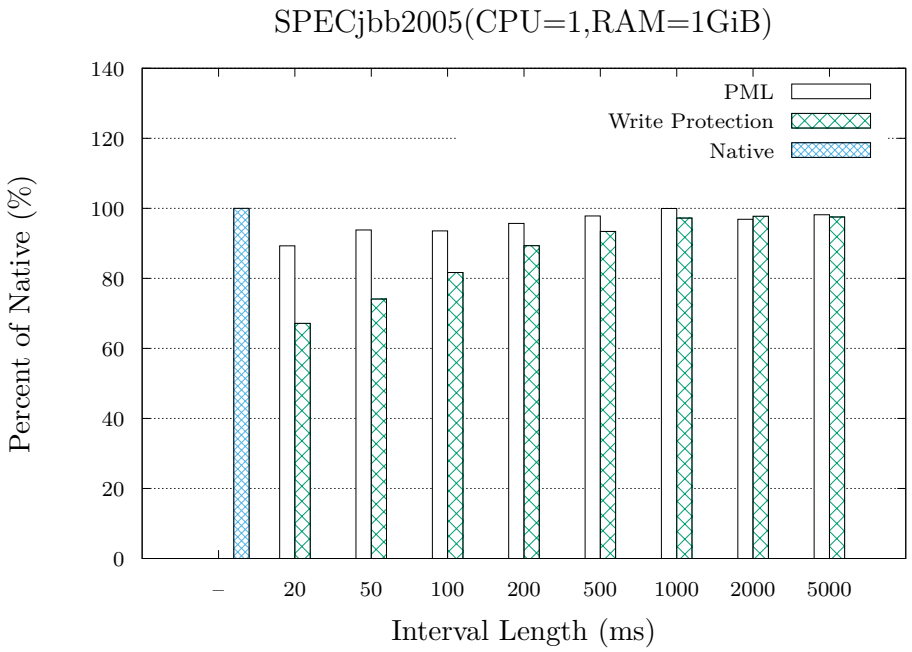

Figure 3.1: Comparison of performance between no dirty logging and dirty logging with PML or write protection. The benchmark measured is SPECjbb2005 run on one core. The horizontal axis denotes interval length between polling the dirty bitmap in milliseconds. The vertical axis denotes score in SPECjbb2005, compared to native, higher is better. All shown values are the average of three runs. PML leads to an increase in performance compared to write protection for short intervals. For intervals $>$ 1s, the benefit is neglectable.

dirty logging on the score of SPECjbb2005, run with one virtual CPU. For an interval of 20ms, the score with PML is 89.3% of that of native execution in the VM, where native refers to running in the VM without performing dirty logging or checkpointing. The score with write protection is 67.2%, thus PML can increase the score by around 22% compared to write protection. As the interval size increases, the benefit of using PML diminishes. At 1s intervals PML's score is as high as native's and write protection's is 97% — a difference of 3%. The reason that small intervals benefit more form PML

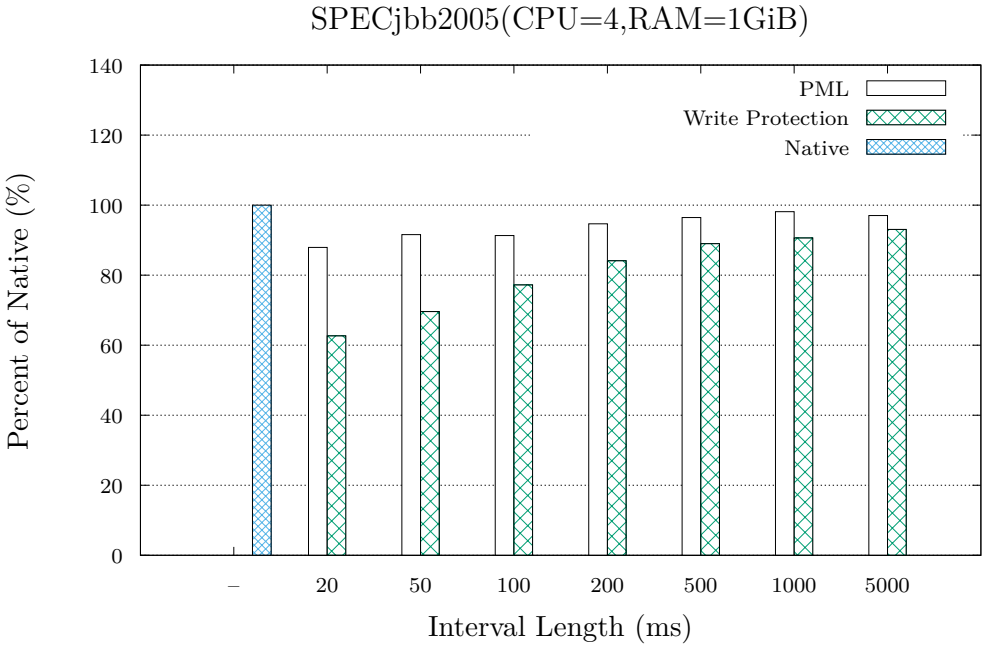

Figure 3.2: Comparison of performance in SPECjbb2005 running with four cores between no dirty logging and dirty logging with PML or write protection. The horizontal axis denotes interval length between polling the dirty bitmap in milliseconds. The vertical axis denotes score in SPECjbb2005, compared to native, higher is better. The results are very similar regardless of the number of CPUs.

is that they, in total, cause more dirty pages. Longer intervals clear the dirty bits more rarely and through this, increase the probability that writes access pages that are already dirty. For these pages, no additional work is performed, which is why the effect of PML is stronger for small intervals.

Figure 3.2 shows the results of running SPECjbb2005 with four cores. Multiple cores increase the intensity of SPECjbb2005. Using four cores, the gain of PML ranges from 22.1% at 20ms to 7% at 1s. In a similar setup, Kai Huang [17] achieved a speedup of 5% for four cores and 1s. The version of SPECjbb he used, however, is unknown; SPECjbb2005 had been outdated by at least two years at that time. His tests ran on a 16 core CPU, ours on a 4 core CPU. Using four cores instead of one increases the absolute score, but hardly the relative increase caused by PML.

We additionally analyze the impact of dirty logging with another benchmark, the Linux kernel build of Phoronix Test Suit [1]. Figure 3.3 shows the results for one CPU. The results are similar to those of SPECjbb2005, the benefit of using PML diminishes with increasing interval length and there is

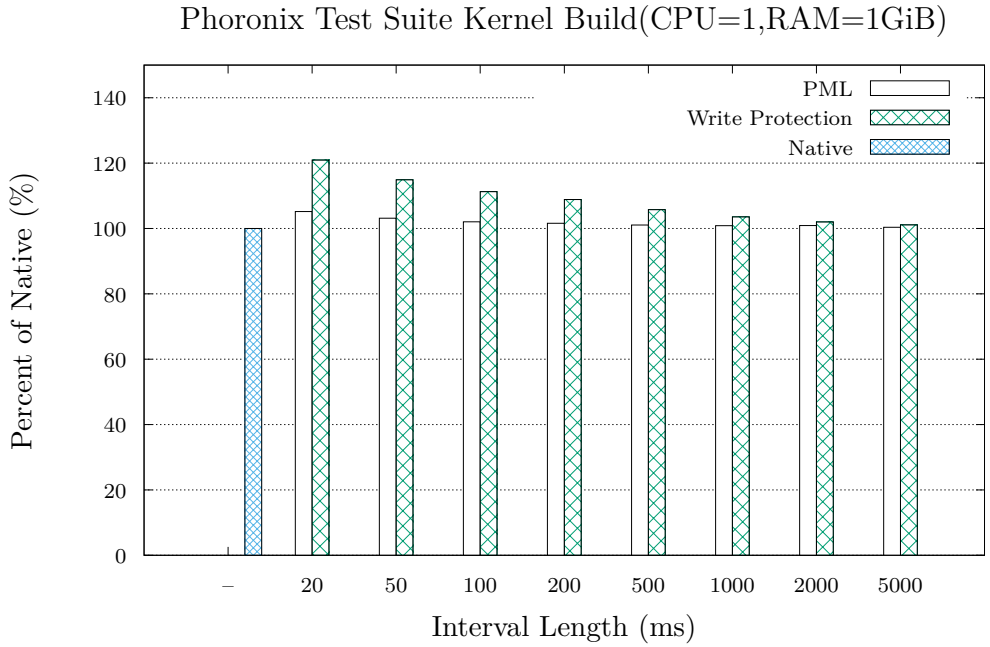

Figure 3.3: Comparison of time to completion of a Linux kernel build between PML and write protection. The horizontal axis denotes interval length between polling the dirty bitmap in milliseconds. The vertical axis denotes time to complete the benchmark, measured on the host, lower is better. The results show a similar progression to that of SPECjbb2005.

little difference in the performance gain across CPUs (see Figure 3.4). For one CPU and 1s interval length it amounts to 3%. Starting with an interval length of 0.5s, the time to build the kernel with PML is close to native (within 1%). For write protection, it is the case for an interval length of 5s.

To gain a better understanding of why PML and write protection cause overhead, we measure the number of VM exits, page faults and PML full events.To perform dirty logging, write protection generates additional page faults, PML introduces PML full events. Both cause a VM exit that has to be processed by the VMM. Note that the VMM used — KVM — also flushes the PML buffer on VM exits not caused by the buffer being full.

Table 3.1 gives the number of events counted during a kernel build. The table shows that the number of VM exits when performing dirty logging via write protection is much higher than without dirty logging. As the interval length increases, the number of dirtied pages decreases along with the number of VM exits. The same pattern holds for the number of page faults. When

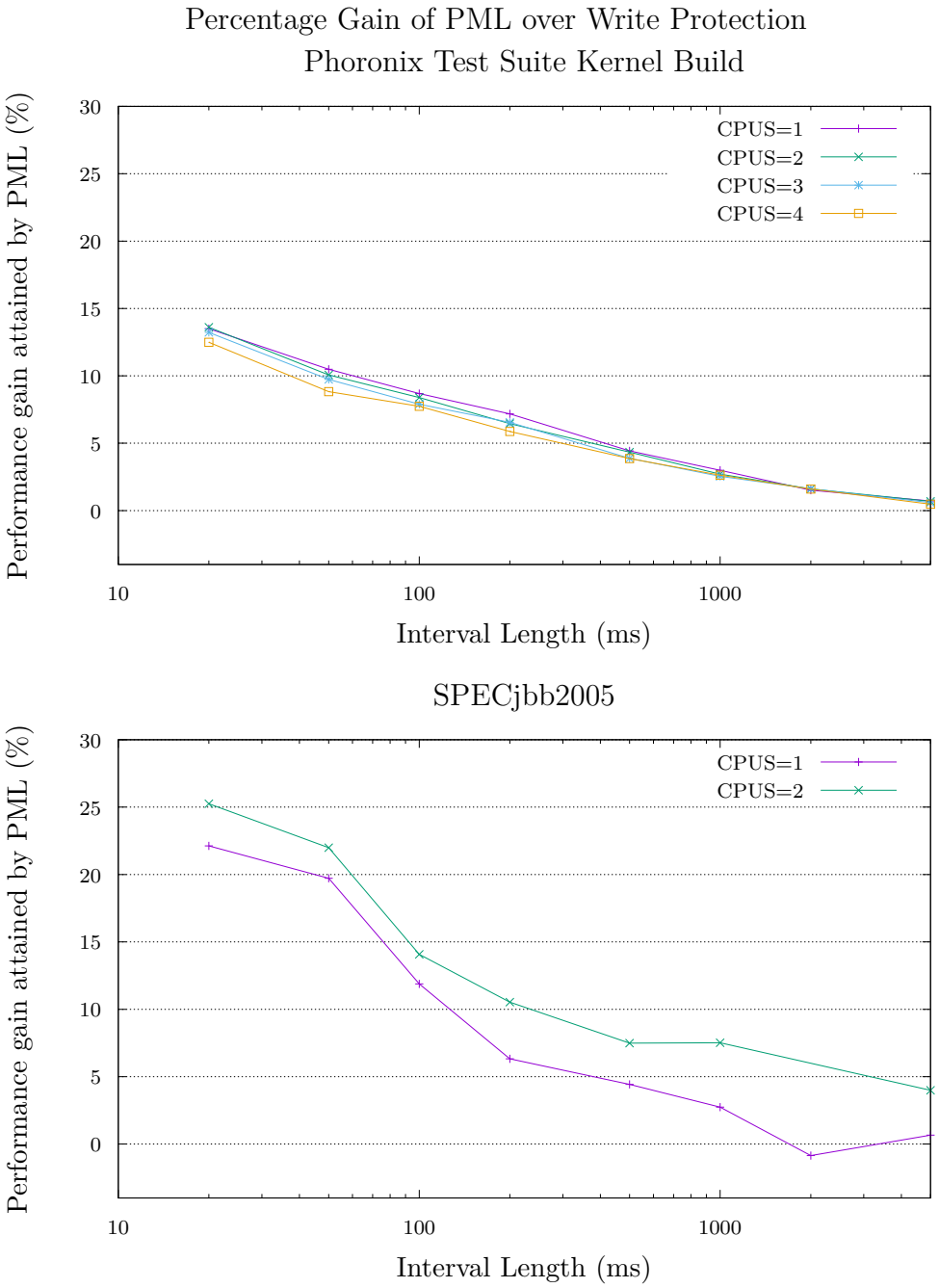

Figure 3.4: Overview of the performance increase created by PML. The horizontal axis denotes interval length on a logarithmic scale, the vertical axis denotes gain of PML over write protection in percent, higher is better. For interval lengths $\geq$ 100ms, the increase varies between 14% and -1%. At an interval length of 2s and 1, CPU PML causes a decrease in performance, for reasons unkown.

Native

VM Exits	Page Faults
$4.6 \cdot 10^6$	$2.8 \cdot 10^5$

Interval Length (ms)	Write Protection		PML		
	VM Exits	Page Faults	VM Exits	Page Faults	PML Full Events
20	$8.8 \cdot 10^7$	$8.3 \cdot 10^7$	$5 \cdot 10^6$	$4 \cdot 10^5$	42,594
50	$6.3 \cdot 10^7$	$5.8 \cdot 10^7$	$4.8 \cdot 10^6$	$4 \cdot 10^5$	28,994
100	$4.9 \cdot 10^7$	$4.4 \cdot 10^7$	$4.8 \cdot 10^6$	$4 \cdot 10^5$	19,902
200	$3.8 \cdot 10^7$	$3.4 \cdot 10^7$	$4.7 \cdot 10^6$	$4 \cdot 10^5$	12,787
500	$2.5 \cdot 10^7$	$2 \cdot 10^7$	$4.7 \cdot 10^6$	$4 \cdot 10^5$	6,573
1,000	$1.7 \cdot 10^7$	$1.2 \cdot 10^7$	$4.7 \cdot 10^6$	$4 \cdot 10^5$	3,677
2,000	$1.2 \cdot 10^7$	$7.3 \cdot 10^6$	$4.7 \cdot 10^6$	$4 \cdot 10^5$	2,088
5,000	$8 \cdot 10^6$	$3.7 \cdot 10^6$	$4.7 \cdot 10^6$	$4 \cdot 10^5$	959

Table 3.1: Results of tracing events for kernel build, run with one CPU. Numbers are averages of three runs. The number of VM exits shows the same tendency as the performance in the benchmark. PML is able to reduce the number of page faults and therefore VM exits.

performing dirty logging via PML, a different behavior can be observed. The number of VM exits and page faults is static across the interval length. Both the number of VM exits and the number of page faults are within an order of magnitude of those measured for native execution. As the number of page faults decreases with increasing interval lengths in the case of write protection so does the number PML full events in the case of PML. PML is able to reduce the number of page faults by up to a factor of approximately 200 at 20ms interval length; at 1s the factor is about 40. The factors for VM exits are smaller at about 17 at 20ms and 4 at 1s.

PML reduces the number of page faults but introduces PML full events. The ratio $R(l)$ between additional page faults and PML full events (Figure 3.5) is not constant, but increases with the interval length.

The fact that $R(l)$ ranges from 2000 to 4000, when the PML buffer can hold 512 entries at most, is counterintuative at first. There is, however, no contradiction, because the number of entries flushed by KVM on unrelated

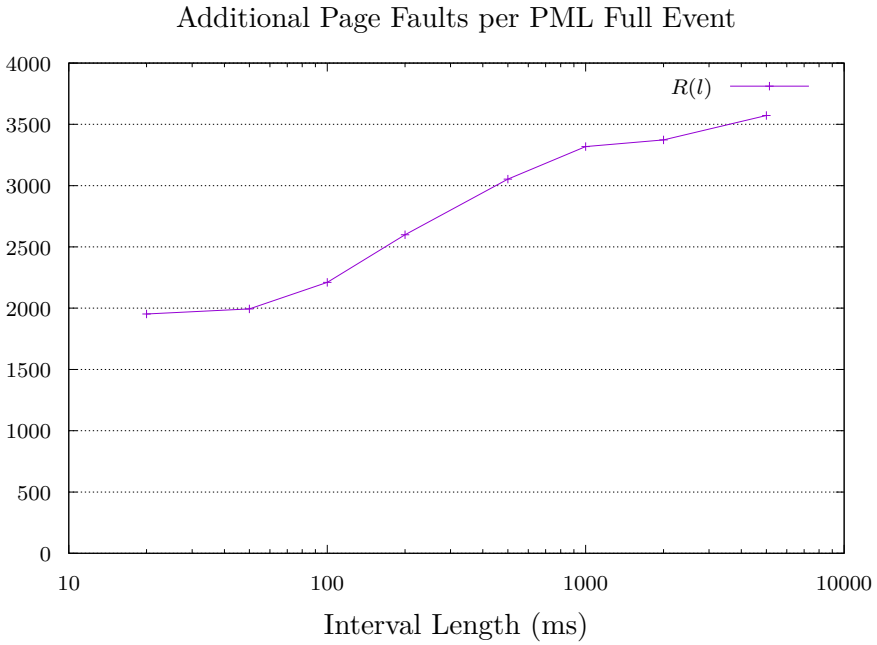

Figure 3.5: The ratio $R(l)$ of additional page faults when performing dirty logging via write protection per introduced PML full event when using PML. The horizontal axis denotes interval length. Values given are for one CPU. The ratio increases with the interval length and can exceed the PML buffer size.

VM exits is also accounted for by $R(l)$. In consequence, $R(l)$ can and does exceed the size of the PML buffer. With growing interval length, the probability rises that the PML buffer is flushed before it reaches its capacity, leading to the observed increase in $R(l)$. The results for multiple CPUs are similar.

When performing dirty logging via write protection, we expect the increase in time to complete the workload to be a function of the number of page faults. Figure 3.6 shows the number of additional page faults caused by dirty logging via write protection and the increase in time to completion. The progression of the number of page faults matches the progression of the time overhead well. Therefore, we estimate the time to completion with write protection $T_{WP}(l)$ as $T_{WP}(l) = \alpha_{WP}\,(P_{WP} - P_{native}) + T_{native}$, where P_{WP} is the number of page faults using write protection, P_{native} is the number of page faults without dirty logging and T_{native} the time to completion without dirty logging. α_{WP} represents the cost of one page fault, we set $\alpha_{WP} = 1.25 \cdot 10^{-6}$s. Figure 3.7 shows a similar consideration for dirty

logging using PML. Instead of additional page faults, we examine the relationship with PML full events. Compared to write protection, the curves are not as similar, especially for longer intervals. Similar to write protection, we estimate $T_{PML}(l) = \alpha_{PML} P_{PML} + T_{native}$, deriving $\alpha_{PML} = 2.4 \cdot 10^{-3}$s. α_{PML} is bigger than α_{WP} by a factor of approximately *1900*, showing that PML flushes are more costly to process than page faults. This is because a page fault is caused by a write access to a read only page and the VMM must only perform work for this single page. In the case of a PML flush, the VMM has to process every address in the buffer. However, since there are a lot less PML flushes than page faults, PML yields a net benefit. Figure 3.8 shows the relative errors of T_{WP} and T_{PML} calculated with the given αs compared to the measured values. For both write protection and PML, the error is within 1%.

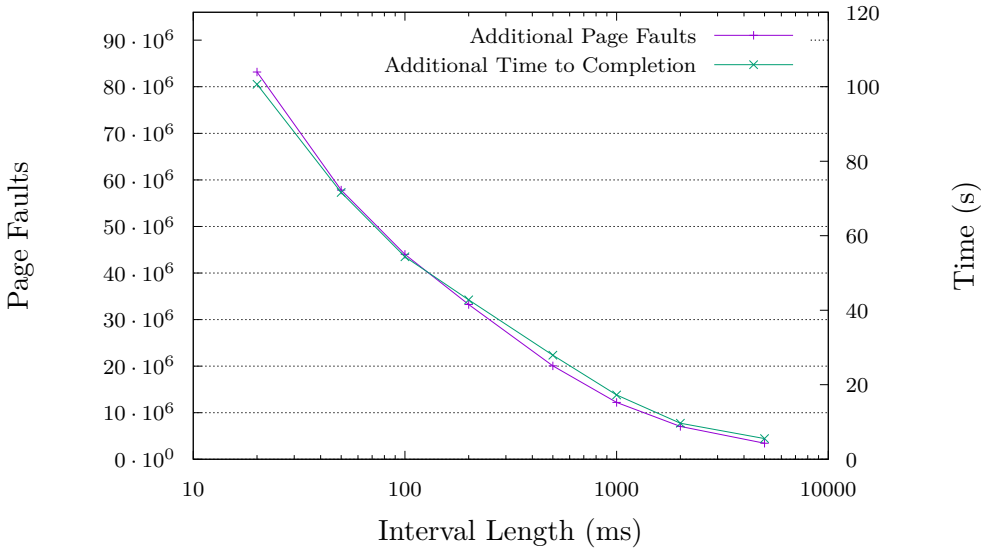

Figure 3.6: Comparison of page faults caused by dirty logging via write protection to the increase in time to complete a kernel build. The number of page faults is given by the left y-axis, the time to completion on the right. Values given are for one CPU. The curves show a similar progression, indicating that one can be predicted from the other.

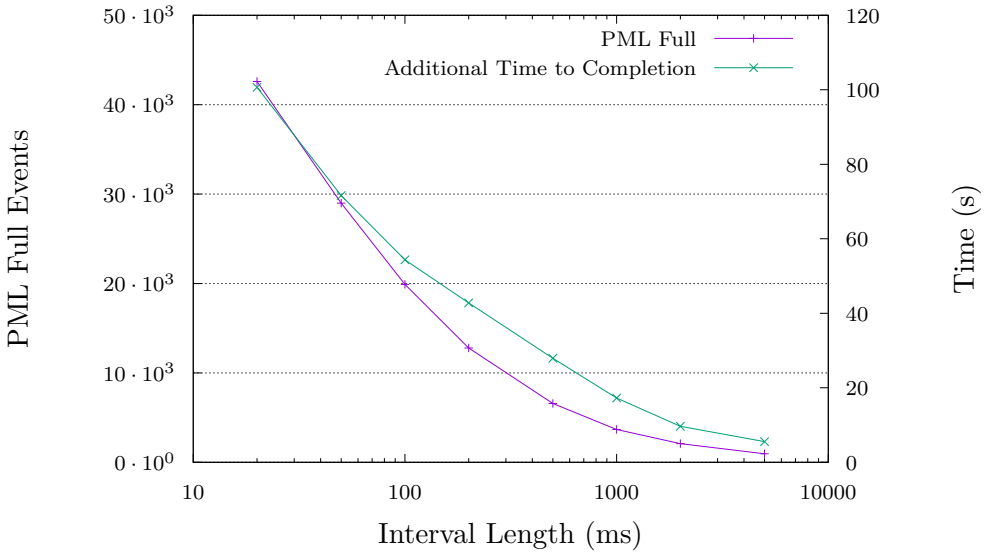

Figure 3.7: Comparison of PML full events introduced by dirty logging via PML compared to the additional time to complete a kernel build. The number of PML full events is given by the left y-axis, the time to completion on the right. Values given are for one CPU. Overall, the curves are similar but not as close as in the same consideration for write protection.

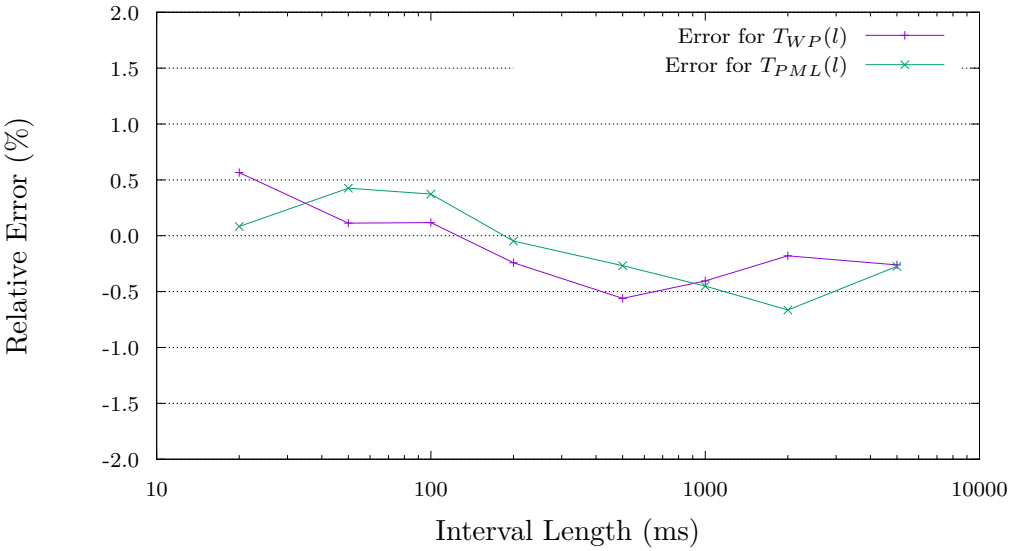

Figure 3.8: The relative errors of T_{WP} and T_{PML} to the measured values. Both are accurate to 1%.

3.1.1 Conclusion

We were able to replicate a performance increase by using PML for dirty logging. The gain is small but measurable, at 1s interval length, it is a small one digit percentage. Smaller interval lengths benefit more from PML, hitting two digit percentages. We have confirmed that PML can reduce the number of page faults by multiple orders of magnitude, thereby also reducing the number of VM exits. The overhead of dirty logging can be predicted by the number of additional VM exits. The number of virtual CPUs had little influence on the benefit of PML. This is because, even though the absolute number of dirty pages increases with more CPUs, the number of dirty pages per CPU does not. Since each CPU can handle page faults and dirty logging via PML independently, no additional overhead is introduced.

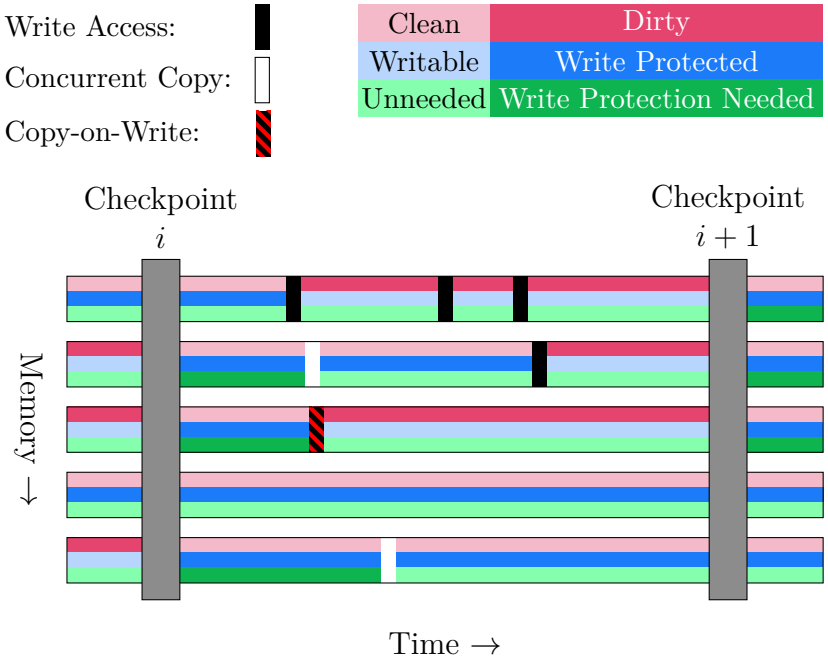

Figure 3.9: Overview of incremental COW checkpointing using write protection for dirty tracking. Write protection is enabled whenever a page is clear.

3.2 Incremental COW Checkpointing Using PML

After having established that PML causes a performance benefit for pure dirty logging, we estimate the potential for using PML as a dirty logging mechanism for incremental COW checkpointing. The following section initially explains incremental COW checkpointing on a conceptional level, emphasizing the different states memory pages can be in. Then it describes the changes necessary to enable PML and presents a model to estimate the expected performance gain.

3.2.1 Incremental COW Checkpointing using Write Protection

The current SimuBoost implementation employs incremental COW checkpointing and uses write protection for dirty logging. Figure 3.9 gives an outline, showing which states memory pages can be in over time.

The horizontal axis denotes time. The vertical bars represent two adjacent

checkpointing downtime periods during which VM execution is suspended. Outside of these periods, the VM executes normally. The rows represent memory pages, they are marked to show which state they are in at every moment in time.

For clean pages ( ) dirty logging must be active. When a write access ( ) to a clean page takes place, the dirty logging mechanism marks it dirty ( ). Subsequent write accesses to dirty pages do not need to be logged. Since write protection is used for dirty logging, all clean pages are write protected ( ). When a write access to a clean page occurs, the page fault handler marks it dirty and disables write protection for it.

During the downtime, the VMM removes the dirty status from dirty pages. For copy-on-write to work, these formerly dirty pages must be write protected ( ) until they have been copied, either via concurrent copy ( ) or via COW ( ). If a COW case occurs, the VMM marks the page dirty. Pages clean before the downtime remain write protected for dirty logging.

3.2.2 Conceptional Changes for Checkpointing Using PML

When using PML instead of write protection for dirty logging, the VMM removes the dirty status from dirty pages during the downtime and write protects them for COW, as before. These pages do not need to be dirty logged until they have been copied. If a write to them takes place before they have been saved, the COW case occurs, the page is marked dirty and no further dirty logging is required until the next checkpointing downtime. In order to use PML for dirty logging, only pages that were saved via concurrent copy need to be considered. For them, write protection must be disabled and PML enabled. Nothing needs to be done for pages that were clean and for which PML is already active.

Multiple designs are possible which differ in the timing of when they disable write protection and enable PML. One possibility is *early PML activation*. Early activation enables PML immediately after a page has been saved via concurrent copy, and disables write protection. Another possibility is *late PML activation*. Late activation does not switch PML and write protection immediately but defers it to some point later in time, the next checkpointing downtime seems an obvious choice.

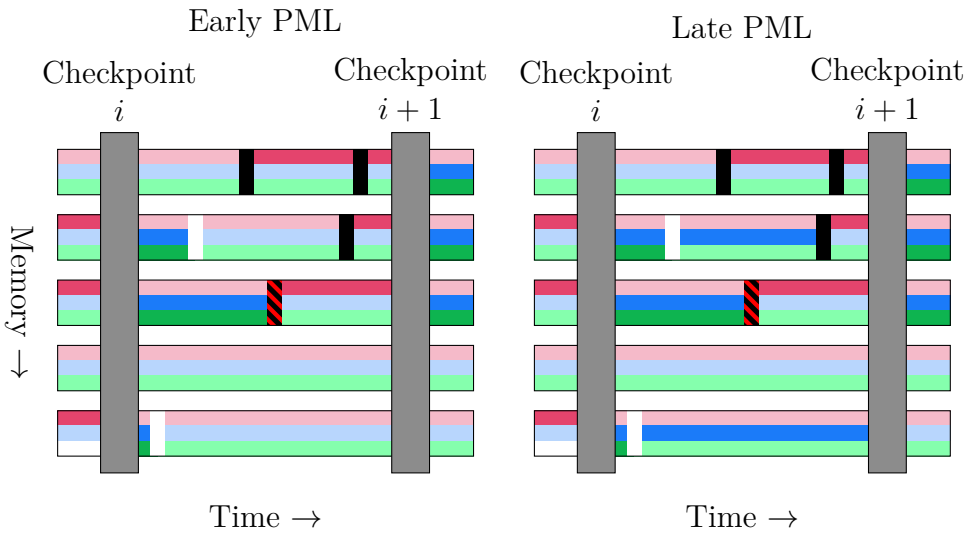

Figure 3.10: Overview of early and late PML activation. When using early activation, write protection is only enabled (■) when it is necessary for COW (■), whereas it is enabled until the next checkpoint or next write access when using late PML activation.

3.2.3 Estimated Performance Gain of PML

In order to estimate the performance gain of late PML activation, we traced a kernel build while performing continuous checkpointing with write protection (Figure 3.11). The time to build the kernel shows the same development as the results of dirty logging. At 20ms, the workload takes approximately 720s to complete; the increase to an interval of 50ms causes a drop to about about 600s and as the interval length increases further, the run time decreases ever more slowly. We measured the time in and outside the virtual machine. The time measured outside includes the downtimes as well as some setup work performed by the benchmark. When taking one checkpoint, we found that the clock inside the VM did not observe the downtime, we therefore interpret the increase in run time measured *inside* as that caused by the slowdown of execution (due to page faults). When performing incremental copy-on-write checkpointing, one part of the slowdown will be due to dirty logging and one due to COW. COW cases occur regardless of whether PML or write protection is used for dirty logging. Thus, we expect the slowdown caused by COW to be comparable when using either method. For pages copied via COW, no explicit dirty logging is performed. Therefore, the number of concurrently copied (CC) pages is the same as the number of pages which were explicitly dirty logged in the previous checkpointing interval — the pages that would

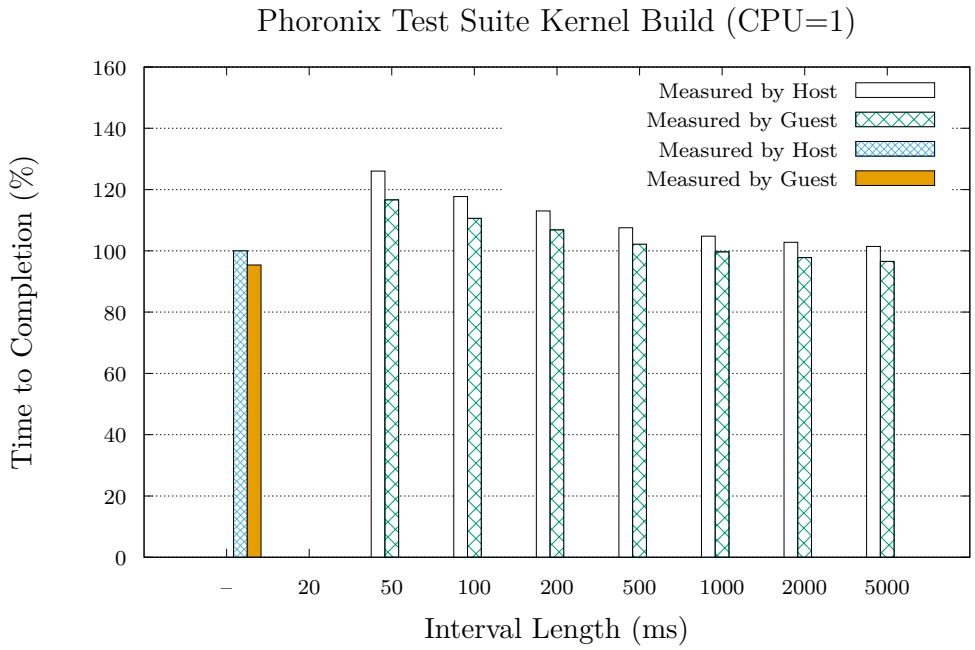

Figure 3.11: Time to complete a kernel build while performing checkpointing using write protection as dirty logging mechanism. Values are given in percent of native, measured by host, lower is better. Because of limited hardware, accurate measurements for an interval length of 20ms could not be obtained, as a result the value has been omitted

Interval Length (ms)	$B(l)$
50	$6.7 \cdot 10^{-6}$
100	$6.7 \cdot 10^{-6}$
200	$8 \cdot 10^{-6}$
500	$7.5 \cdot 10^{-6}$
1,000	$8.4 \cdot 10^{-6}$
2,000	$7.2 \cdot 10^{-6}$
5,000	$7.4 \cdot 10^{-6}$

Table 3.2: $B(l)$ calculated for the given interval lengths. All values are within an order of magnitude of one another. The value for 20ms has been omitted, because accurate measurements were not possible.

benefit from a faster dirty logging method. For write protection, the number of CC cases corresponds to the number of additional page faults. For PML, the number of page full events can be estimated as the number of CC cases over $R(l)$, the ratio of page faults per PML full event. Since we know the cost of dirty logging, we can derive the cost of COW. We assume that for the checkpointing run time inside the VM $T'_{WP}(l)$ holds:

$$T'_{WP}(l) = \alpha_{WP} CC + \beta COW + T_{native} \tag{3.1}$$

$$\iff \underbrace{\frac{T'_{WP}(l) - T_{native} - \alpha_{WP} CC}{COW}}_{=:B(l)} = \beta \tag{3.2}$$

In order to determine β, we trace the number of CC and COW cases. We calculate $B(l)$ for all interval lengths; the results are shown in Table 3.2. The values of B range from $6.7 \cdot 10^{-6}$ to $8.4 \cdot 10^{-6}$, with an average of $\beta := 7.4 \cdot 10^{-6}$. Having derived α and β, we can estimate the time to complete a kernel build measured inside the VM when using PML as dirty logging mechanism as:

$$T'_{PML}(l) = \alpha_{PML} \frac{1}{R(l)} CC + \beta COW + T_{native}$$

Table 3.3 shows the predicted run time for checkpointing with PML, figure 3.12 visualizes the results. The model predicts the benefit of PML to be highest for small intervals, 11% at 50ms. As the interval length increases, the predicted gain decreases to 0.9% at 5s. We conclude that PML is not just a means to improve the performance of dirty logging, but is also applicable to copy-on-write checkpointing, where a mixture of page faults and PML full events occur.

Interval Length (ms)	Time to Completion (s)	Performance Gain (%)
50	510	10.9
100	491.5	8.7
200	479.1	7.4
500	469.8	4.5
1,000	464.8	3
2,000	461.9	1.7
5,000	459.8	$8.5 \cdot 10^{-1}$

Table 3.3: The predicted time to complete a kernel build when checkpointing using PML as a dirty logging mechanism, as well as the expected gain. The predicted values show the same progression as those measured for PML: Small intervals show a benefit, for large ones it is neglectable. The value for 20ms has been omitted, because accurate measurements were not possible.

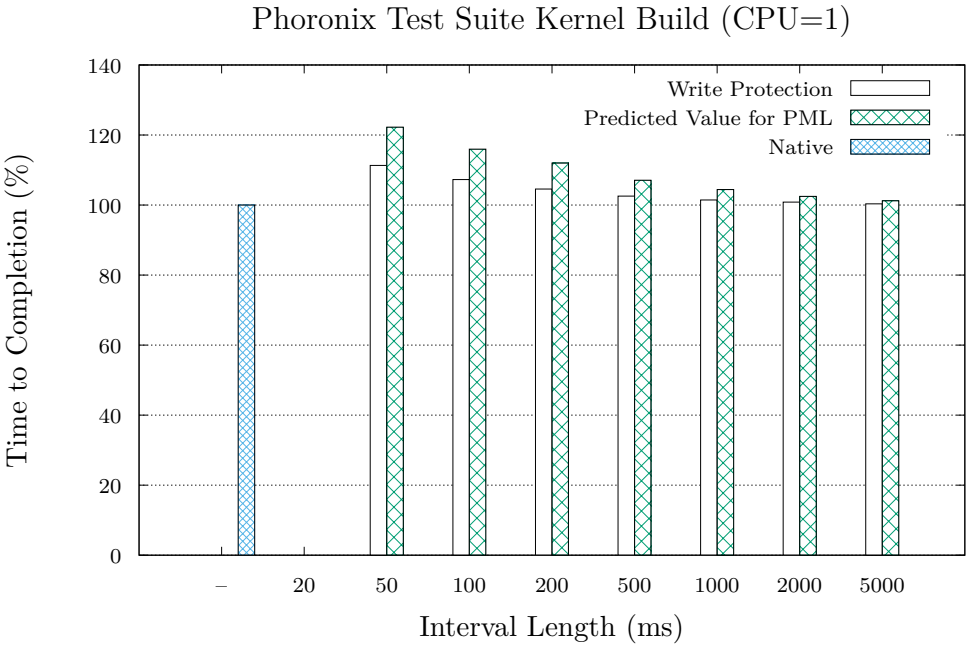

Figure 3.12: The time to complete a kernel build, while checkpointing, measured inside the VM, compared to the predicted time for PML. Values are in percent of native, lower is better. The value for 20ms has been omitted, because accurate measurements were not possible.

Chapter 4

Design & Implementation

The following chapter describes the design and implementation of an incremental copy-on-write (COW) checkpointing mechanism for virtual machines (VM) that uses Intel Page Modification Logging (PML) for dirty page logging instead of write protection.

In order for PML to log write accesses to a page, its dirty flag in the EPT paging structure must be 0 [18]. Clearing this dirty bit requires the entry to be flushed from the translation lookaside buffer (TLB), else the hardware may not notice the change and not log accesses. Full TLB flushes introduce additional overhead because subsequent page translations do not find a cached entries and must perform a page walk.

There are multiple possible designs for replacing write protection with PML as dirty logging mechanism. *Early activation* exchanges write protection for PML as soon as possible, whereas *late activation* defers doing so to the next checkpointing downtime. These approaches have different costs and advantages. Switching early from write protection to PML raises the probability that write accesses are logged with PML and, therefore, should reduce the rate of page faults compared to late activation, which gives the workload a bigger window during which a write causes a page fault. Late activation does not cause additional TLB flushes, since during the downtime period the TLB must be flushed anyway in order to establish write protection. Early activation, however, results in more TLB flushes and thus has a source of cost not present in late activation.

To compare the performance of the two approaches, we have implemented them both. Additionally, we added a number of trace events to the kernel in order to inspect the behavior of checkpointing.

4.1 Goals

A checkpointing implementation must be *correct*. That is the state of the virtual machine after loading a checkpoint must exactly match the state at the moment in time the checkpoint was taken. For our implementation to achieve this, we must incorporate PML in such a way that it logs all write accesses previously logged by write protection, without affecting the write protection necessary for COW.

Our implementations shall improve performance under as many circumstances as possible, that is it shall measureably reduce the slowdown of execution; if this is not possible, it shall do so at least for small intevals. Our implementation may not have a significant negative impact, regardless of interval size and workload.

4.2 SimuBoost Implementation

The SimuBoost implementation consists of an userspace part, a modified QEMU, and a kernel part, a modification of Linux Kernel-based Virtual Machine (KVM). Figure 4.1 provides an overview of its incremental [8] copy-on-write [10] checkpointing implementation. Our implementation modifies it in order to incorporate PML as a dirty logging mechanism.

KVM separates the guest's physical memory into *slots*, represented by the `kvm_memory_slot` structure. This structure contains the dirty bitmap, where the nth bit of the bitmap specifies whether the nth page in the slot is dirty. QEMU maintains a separate dirty bitmap, because it can modify the guests memory without involving the kernel (e.g., during I/O operations with virtual devices). When SimuBoost performs an incremental copy-on-write checkpoint, QEMU calls into KVM via the `KVM_COW_SYNC_AND_PROTECT ioctl` while the VM is suspended. This `ioctl` identifies which pages have been modified and write protects them. `KVM_COW_SYNC_AND_PROTECT` synchronizes those two bitmaps and re-enables dirty logging for dirty pages, so future accesses to those are logged. Böhr statically disabled PML [10]. Therefore, pages are write protected, which is necessary for COW *and* serves as a dirty logging mechanism. The write protection is removed when a page is dirtied, which is why the `ioctl` has to re-enable it. On the first checkpoint, all pages are marked dirty, because there is no previous checkpoint to create an incremental checkpoint against. Per VM instance, SimuBoost maintains the state of guest pages (i.e., clean, dirty, currently being copied) in the *copy map*. After the `ioctl` has finished, QEMU resumes the VM and calls the `KVM_COW_CHECKPOINT ioclt`. This `ioctl` saves the dirty pages until the

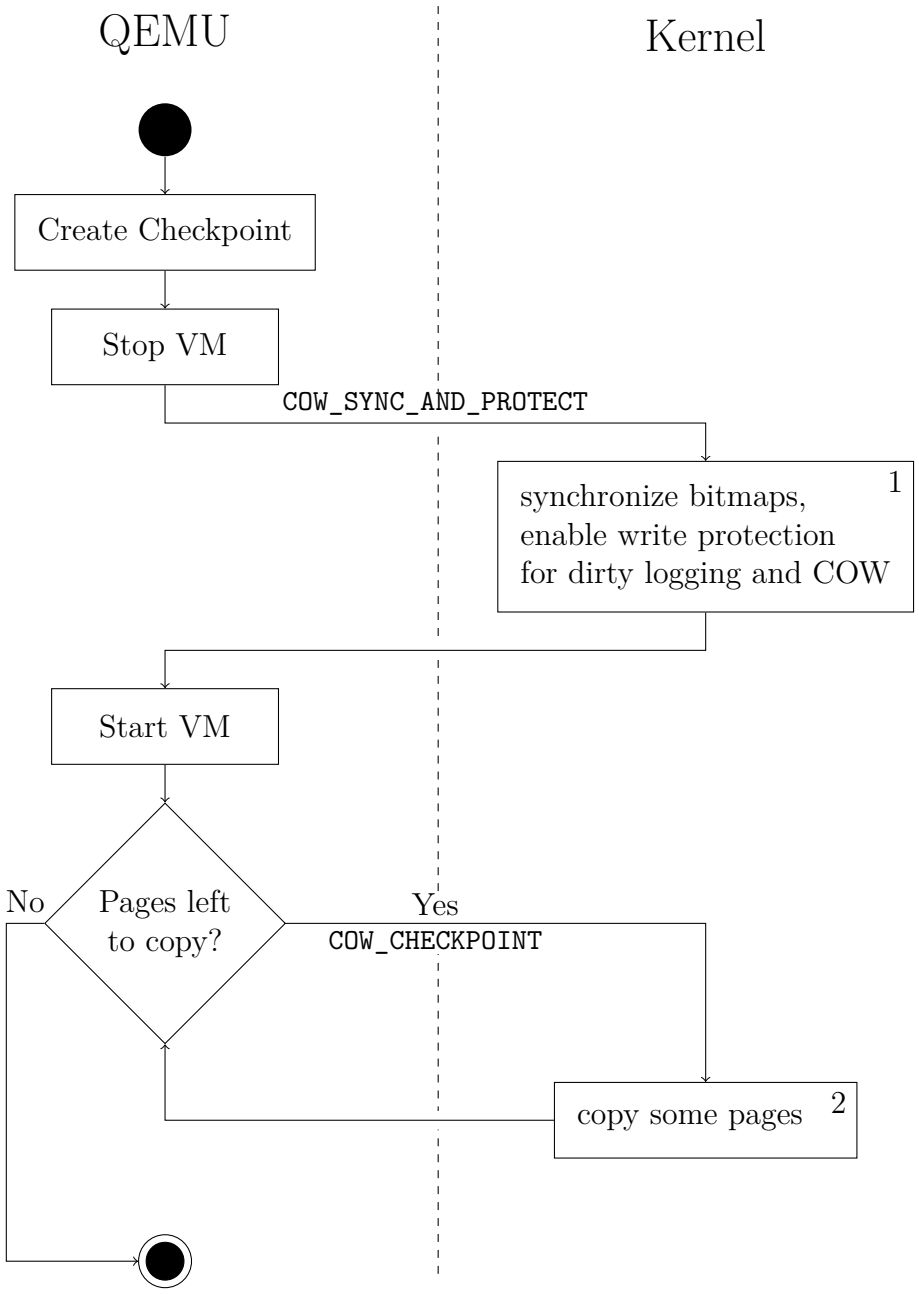

Figure 4.1: Overview of SimuBoosts checkpointing implementation. Elements on the left execute in user space, elements on the right in kernel space. 1 constitutes the work performed during the downtime. 2 is the majority of work, performed asynchronously to workload execution.

64MiB buffer provided by QEMU is full. QEMU calls it in a loop and provides it with a new buffer to fill until all pages have been saved. This is done in a separate thread, simultaneously to VM execution.

4.3 Functionality Common to Early and Late Activation

Both early and late activation need to enable PML and disable write protection. KVM has support for dirty logging via PML [17]. This includes functions to enable PML for a virtual CPU, as well as (re)activating PML for specified frames by resetting the dirty bit in the Extended Page Table (EPT) [18] 2.3.1. Functionality to selectively disable write protection for some frames does not exist in KVM, requiring us to implement it.

KVM uses the same function to set write protections when using a shadow MMU as when using two dimensional paging such as Intel EPT. With EPT, write protecting a guest frame is straight forward: one clears the writable bit of the page table entry in the second table (i.e, EPT) that translates guest physical to host physical frames 4.2. If two guest virtual pages map to the same guest frame, neither would be writable, because of the second translation. When using shadow paging, there do not exist two levels of paging, both are collapsed into the shadow page table. If multiple guest virtual pages map to the same guest frame, the shadow page table will have multiple translations from guest virtual pages to host frames. Protecting a guest frame requires all these translations to be write protected. KVM does so by finding all page table entries responsible for these translations via a *reverse map*. The same mechanism works for EPT, because the reverse map points to the EPT page entry.

Late and early PML activation require the write protection to be removed and, therefore, these changes to be reverted. Some guest frames always need to be write protected, for example KVM write protects those guest frames on which guest page translation tables reside, in order to be informed of changes to them [20].

Although these frames are only protected when emulating the MMU via shadowing and not when using EPT, we decided to exercise caution and not remove write protection unconditionally. We studied the behavior of the page fault handler in order to discover under which circumstances it removes write protection. Our function to remove write protection[1] performs the same checks as the page fault handler and closely mirrors the existing function for

[1] `kvm_mmu_clear_write_protect_pt_masked`

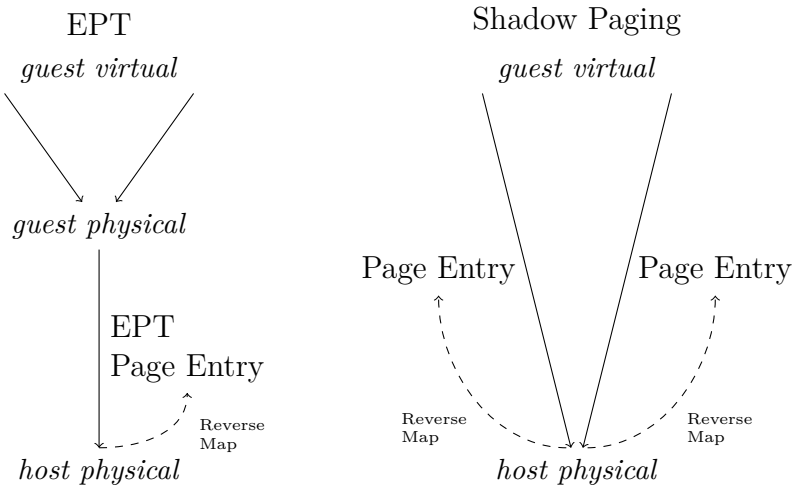

Figure 4.2: Comparison between two level paging (EPT) and shadow paging, when changing write protection. In the case of EPT, the page entry in the *guest physical* → *host physical* table must be modified. In the case of shadow paging, all entries in *guest virtual* → *host physical* tables must be modified. Going backward via the reverse map automatically performs the right action.

setting write protection[2]: For a given guest frame, it obtains the *reverse map*, which points to all the page table entries that translate to the frame, and sets their writable flag.

When testing our function, it *appeared* not to work. No reverse mappings were found and no writable bits were set. This turned out to be the case for two reasons:

1. When dirty logging is activated, before the first checkpoint, a function is called for readonly slots. This function[3] flushes translations that point to frames in the given slot. On x86, the implementation of this function ignores the slot parameter and flushes *all* translations. For this reason, there were no page translations which our function could set writable *while we were inspecting* its functionality.

2. Linux tries to keep data in memory close to the CPU using it (NUMA balancing), which similarly flushes translations.

When we disabled NUMA balancing and inspected our implementation at *a point in time* where the translations flushed by case 1 were *reestablished*,

[2] `kvm_mmu_write_protect_pt_masked`
[3] `kvm_arch_flush_shadow_memslot`

we observed write protection being removed as intended. Pages which are non present, for example after they have been flushed by 1 or 2, inevitably generate a page fault when they are accessed. Our function doing nothing for non present pages is the correct behavior as the page fault cannot be avoided by PML.

4.4 Late Activation

With late activation, the VMM, during a checkpointing downtime, activates PML for those pages that are clean now, but were dirty at the last checkpoint. Figure 4.3 gives an overview of the implementation.

We do not need to activate PML for pages it is already active for, and avoid the cost of doing so. In order to know which pages to activate PML for, we add a second dirty bitmap `old_dirty_bitmap` field to the `kvm_memory_slot` structure. This bitmap holds a copy of the original bitmap's state at the previous checkpoint. Doing so results in one bit per 4KiB additional memory (in total $+3 * 10^{-5}\%$ of VM memory), which we deem neglectable.

Our modified `KVM_COW_SYNC_AND_PROTECT` implementation[4] iterates over all guest frames and calculates which pages to enable PML for. These pages are those for which the corresponding bit in `old_dirty_bitmap` is one and zero in `dirty_bitmap`. Pages with a set bit in `dirty_bitmap` require write protection for COW and PML is not activated for them. The `ioctl` calls a function that executes the respective operations, then copies `dirty_bitmap` to `old_dirty_bitmap` and resets `dirty_bitmap` to all zeros.

Late activation is compiled-in conditionally and can be enabled via the `KVM_SB_COW_LATE_NATIVE_DIRTY_LOG` kernel config.

4.5 Early Activation

Early PML activation enables PML for a page immediatelly after it has been copied. Figure 4.3 gives an overview of the implementation, which consists of modifications to `KVM_COW_CHECKPOINT` [5].

In KVM, it is custom for operations that operate on many guest frames to be implemented by a function which takes a guest frame number `gfn` and a bitmap. This function executes the operation for all frames for which the respective bit is set in the map, starting with frame number `gfn`.

[4] `kvm_vm_ioctl_cow_sync_and_protect`

[5] `kvm_cow_copy_increment`

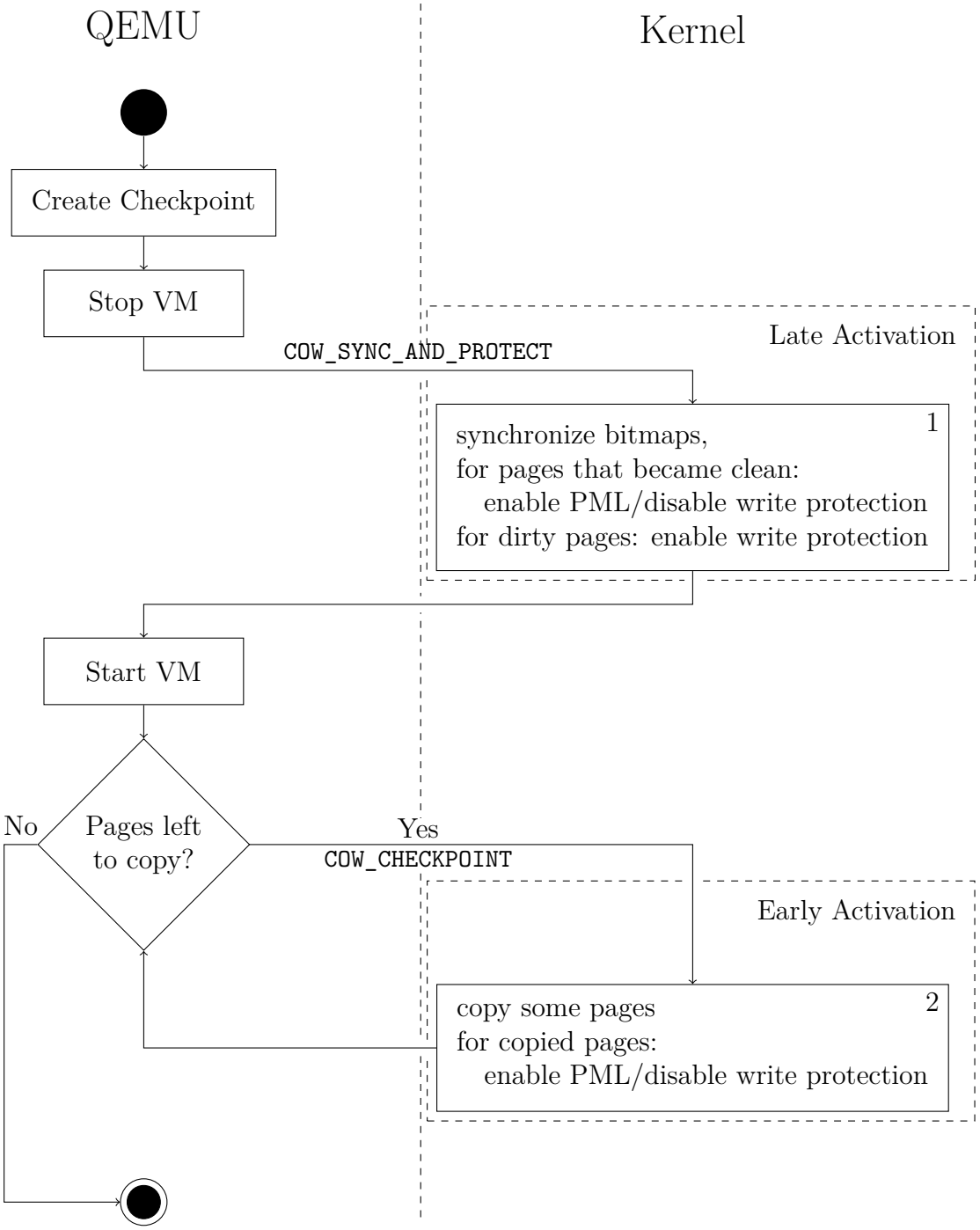

Figure 4.3: Overview of our modifications for PML activation. Late activations only requires changes to [1]; early activation changes to [2].

For example, the function for enabling dirty logging is set up in this way and we implemented our function for removing write protection in the same style.

Although it is not necessary from a design point of view, we add a field `clear_buffer` to `kvm_memory_slot`. When `COW_CHECKPOINT` has concurrently copied a frame, we mark the respective bit in `clear_buffer`. `COW_CHECKPOINT` does not save individual frames, but multiple consecutive ones. Before `COW_CHECKPOINT` completes, we take the content of `clear_buffer` as bitmap to our function, one `long` at a time. Doing so allows us to remove write protection for $BITS\_PER\_LONG$ frames with one call to our function, while keeping the code simple.

Early activation is compiled-in conditionally and can be enabled via the `KVM_SB_COW_EARLY_NATIVE_DIRTY_LOG` kernel config.

4.6 Tracing

In order to facilitate an evaluation of our implementation, we augmented the kernel with a number of trace events, which can be enabled via the `KVM_SB_TRACING` kernel config. The following events were added:

- `kvm_pml_flush` traces the number of entries flushed from the PML log.
- `kvm_sb_inc_diff` traces the number of frames that lost/gained their dirty status compared to the last checkpoint.
- `kvm_sb_cow` traces the number of COW cases that occurred in the kernel for every checkpoint.
- `kvm_sb_cc` traces the number of CC cases for every call to `COW_CHECKPOINT`

Chapter 5

Evaluation

This chapter evaluates whether PML can improve the performance of copy-on-write checkpointing, and examines how the implemented variants behave. Additionally, it assesses the model proposed in Chapter 3.

5.1 Testing Framework

Evaluating our implementation requires experimental data created by executing test cases. Multiple different tests are run with various parameters and repeated to reduce variance. For this reason, experiments can run for days and create many results. Because manually executing every test would be cumbersome, we created a set of scripts to run tests with as little user input as possible and to easily manage the results. Additionally, using scripts makes test runs more *deterministic*, improving comparability. Tests generally will start QEMU, boot an operating system (OS) and then run some command in the virtual machine (VM), while performing *measurements* and *logging* information of interest. Our scripts for running tests are mainly written in Python [5]. We created classes abstracting common functionality:

- OS management: Classes for OS management provide the boot medium and provide methods for booting the OS, logging in and setting up the network and starting ssh so test results can be extracted.

- QEMU: The QEMU class is responsible for starting and stopping the QEMU VM. It loads an OS configuration and sets up two properties `serial` and `monitor`. These properties allow interaction with the VM's serial port and QEMU's monitor command interface. Interaction takes place via the *pexpect* [4] library; it is done by sending text and waiting for a certain response.

- Measurement and Logging: Measurement and logging are provided by one class. The fundamental concept of this class is the nesting of tests. Users of the class can begin a test case and run sub-tests as children of the outer test case. The class automatically replicates this tree of tests in the file system, creating a unique logging directory for every test. Users of the class can easily start and stop measuring the duration of a task. Test results are saved in JSON [2], facilitating easy post processing.

- Tests: The test class provides methods for actions commonly performed during testing, including up-/downloading files to/from the VM, starting and stopping dirty logging and checkpointing, as well as other common tasks such as setting up `simustore` and the tool used to account Linux kernel events, `perf` [25].

Our testing framework logs all relevant information, including the version of the KVM module and its configuration, the source code of scripts, the output of scripts, the interaction with the OS running in the VM, the interaction with the QEMU monitor, the standard output and error of the QEMU process, kernel messages, trace events, and result files created by benchmarks inside the VM.

In addition to scripts for running tests, we created a tool to manage the resulting data. When starting a test, the user gives a comment describing it. Our tool lets the user list all run tests, printing their comments. The user can select a test run and conveniently access its files via a symlink in the current directory, or delete it. Another set of scripts helps with post processing the measured data.

5.2 Evaluation Environment

All benchmarks where run on a machine with the following specifications:

CPU	Intel Core i5-6500 @ 3.2Ghz (4 cores)
Memory	15GiB
Disk	223GiB SSD (OCZ ARC 100)
Mainboard	ASUSTeK Z170-A

The software used and their versions are:

Operating System	Ubuntu 16.04 LTS
Linux Kernel	4.3.0
KVM	4.3.0 [1]
QEMU	2.6.50 [1]
Simutrace	3.3.0 [1]
SPECjbb	2005
Phoronix Test Suite	1.6.0
Linux kernel build	

During testing, the following settings were in effect:

- All virtual CPUs pinned to a physical CPU
- Transparent huge pages disabled
- Numa balancing disabled
- C states disabled

5.3 Correctness

We find our implementation correct, if the state restored by loading a checkpoint is bitwise identical to the state of the VM at the beginning of the checkpointing downtime. The modified QEMU used in SimuBoost has support for dumping the state of the VM at the beginning of the downtime and after loading a checkpoint. This is compiled-in conditionally. Since we did not touch the implementation of device checkpointing, we omit verifying it and only validate that the physical memory of the VM before a checkpoint is identical to that after restoration. We implemented a test case for verifying the correctness of our implementation. With dumping enabled, this test case creates checkpoints while running a memory intensive benchmark inside the VM. Because dumping the state of the VM is slow, the interval length is set to one minute. Then it loads each created checkpoint, causing the state after loading to be dumped. After both before the checkpoint and that after restoration have been dumped, it compares the files for differences. We find that the files are identical when using incremental COW checkpointing with our modifications for PML enabled.

[1] SimuBoost's modifications and our changes

CPUs	Write Protection Downtime (ms)	PML Late Downtime (ms)	PML Early Downtime (ms)
1	2.8	2.3	1.9
2	4.5	3.6	3.1
3	6.1	4.2	3.3
4	8.5	–	3.4

Table 5.1: The downtime averaged over all intervals. Our modifications did not raise the downtime.

5.4 Downtime

We set our implementation the goal to not deteriorate performance. Thus, our implementation may not substantially increase the length of downtimes. To evaluate our success in this regard, we measured the downtimes while running the Linux kernel build of Phoronix Test Suite. The results in Table 5.1 show that our implementation did not raise the downtime. To the contrary, the downtimes for PML are smaller. We cannot offer an explanation for this other than a difference in the test setup. Early PML executes the same code as write protection and late PML performs additional work to activating PML for some pages.

5.5 Slowdown of Execution

In order to examine the impact of PML on checkpointing and to find out whether PML can reduce the slowdown of execution, we ran the kernel build benchmark of Phoronix Test Suite. Figure 5.1 shows the time to complete the benchmark, measured outside the VM. All checkpointing implementations, regardless of the dirty logging method, show the same progression as the pure dirty logging performed in Chapter 3: small intervals cause the biggest performance penalty and as the interval length increases, the overhead of checkpointing lessens ever more slowly. Figure 5.2 shows that the gain of PML is similar regardless of whether one considers the time measured inside or outside the VM. Late PML achieves hardly any benefit compared to write protection; the gain is between -0.1% and 1.4%, tending to zero as the intevals increase in length. Early PML, however, does achieve a noticeable benefit. It is approximately 11% for the shortest interval lengths, but decreases very quickly and starting at an interval length of 0.5s tends slowly to zero.

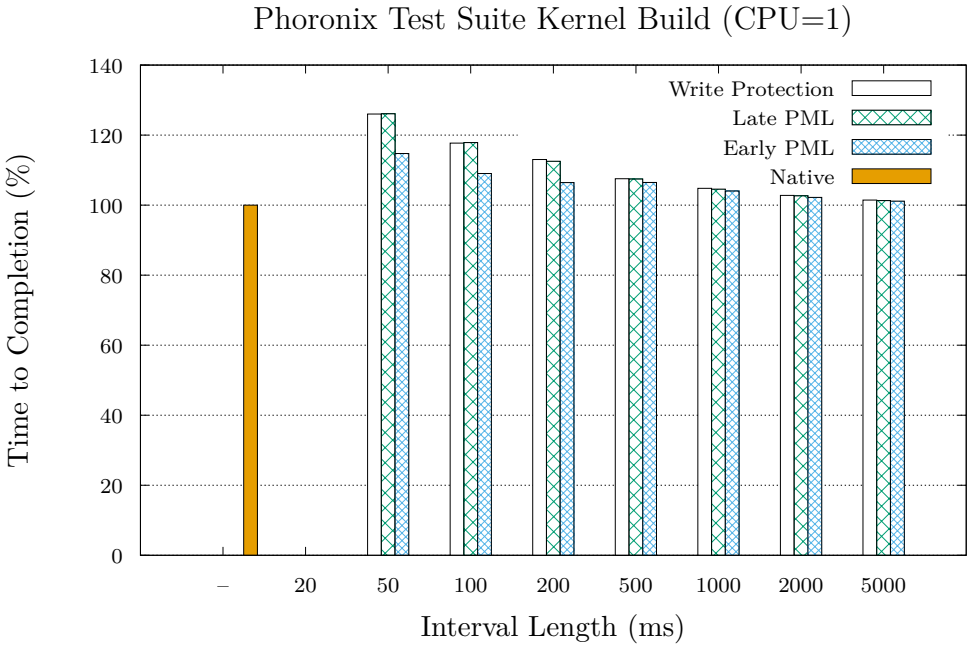

Figure 5.1: The measured time to complete a kernel build for all implementations. Values are in percent of native, lower is better. Write protection and late PML activation are very similar, early activation is able to decrease the run time.

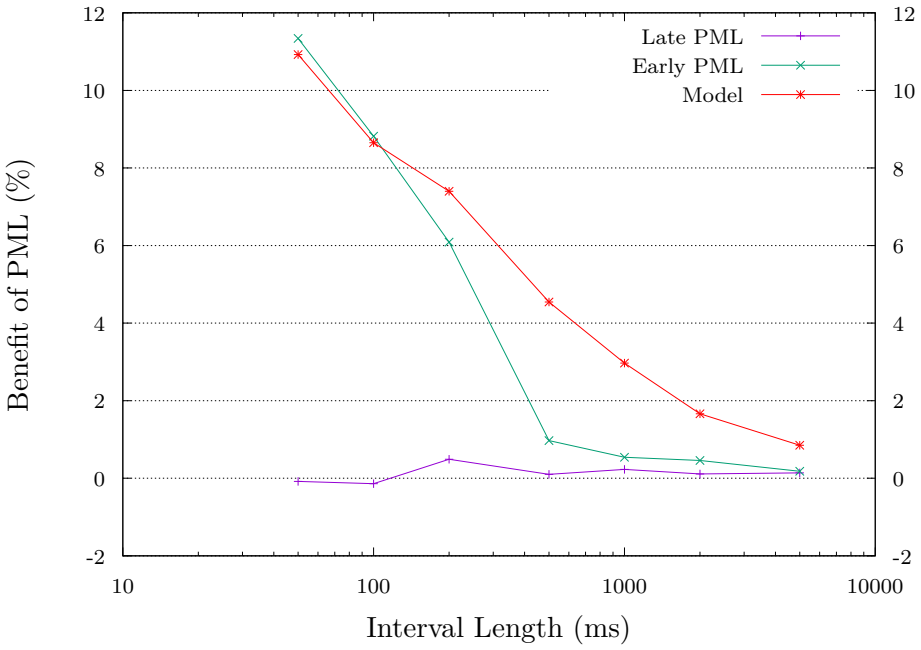

Figure 5.2: The benefit of late and early PML over write protection, as well as the predicted gain. Early activation achieves an improvement, the benefit of late PML is neglectable. The model severely overestimates the benefit for intervals $\geq$ 500s.

Interval Length (ms)	Write Protection TLB Flushes	Late PML TLB Flushes	Early PML TLB Flushes
50	12,701	12,693	22,242
100	5,682	5,686	10,527
200	2,732	2,721	5,531
500	1,041	1,040	2,924
1,000	507	506	1,528
2,000	249	248	813
5,000	98	98	370

Figure 5.3: The number of TLB flushes observed during a kernel build. Early PML activation creates approximately twice as many as late activation.

Our model does not predict such a steep decline and overestimates the benefit of PML for intervals greater than 200ms. The curves differ the most in the area where the number of PML full events diverge from the additional run time in Figure 3.7, which causes the effective cost of a PML full event α_{PML} to be underestimated. As with dirty logging, we assess the number of VM exits, page faults, and PML full events as shown in Table 5.2. All three checkpointing methods show the same pattern of the VM exits declining with rising interval lengths. For write protection, most VM exits are due to page faults. Late PML activation reduces the number of page faults and therefore VM exits. At 50ms it decreases the number of VM exits by 47%, however, the reduction shrinks to only 6% at 5s. Early activation is able to reduce the number of VM exits significantly more, from 80% at 50ms to 11% at 5s. This difference can be explained by the discrepancy in PML full events. The number of PML full events in the case of late PML is surprisingly small, in the order of hundreds, whereas early PML reaches values of up to 25000 for 50ms. Early PML leaves a smaller window during which a write access creates a page fault. The fact that early PML logs far more pages via PML than late PML, indicates that pages dirtied in the interval preceding a checkpoint are likely to be written to before the next checkpoint, which is consistent with the principle of locality commonly attributed to computer programs [13].

We expected early PML activation to generate more TLB flushes, which might have negated the advantage of having PML enabled for a higher proportion of time. We measured the number of TLB flushes 5.3, and confirm that early PML increases it. The number of TLB flushes in the case of late

Native		Write Protection		
VM Exits	Page Faults	Interval Length (ms)	VM Exits	Page Faults
$4.6 \cdot 10^6$	$2.8 \cdot 10^5$	50	$6.5 \cdot 10^7$	$6 \cdot 10^7$
		100	$5 \cdot 10^7$	$4.6 \cdot 10^7$
		200	$3.8 \cdot 10^7$	$3.4 \cdot 10^7$
		500	$2.5 \cdot 10^7$	$2.1 \cdot 10^7$
		1,000	$1.7 \cdot 10^7$	$1.3 \cdot 10^7$
		2,000	$1.2 \cdot 10^7$	$7.5 \cdot 10^6$
		5,000	$8.2 \cdot 10^6$	$3.8 \cdot 10^6$

	Late PML			Early PML		
Interval Length (ms)	VM Exits	Page Faults	PML Full Events	VM Exits	Page Faults	PML Full Events
50	$3.4 \cdot 10^7$	$2.8 \cdot 10^7$	472	$1.3 \cdot 10^7$	$8.6 \cdot 10^6$	25,312
100	$2.8 \cdot 10^7$	$2.2 \cdot 10^7$	547	$1 \cdot 10^7$	$5.6 \cdot 10^6$	17,993
200	$2.4 \cdot 10^7$	$2 \cdot 10^7$	575	$1.1 \cdot 10^7$	$6.8 \cdot 10^6$	8,799
500	$2 \cdot 10^7$	$1.6 \cdot 10^7$	357	$1.9 \cdot 10^7$	$1.5 \cdot 10^7$	722
1,000	$1.5 \cdot 10^7$	$1.1 \cdot 10^7$	249	$1.3 \cdot 10^7$	$9.1 \cdot 10^6$	210
2,000	$1.1 \cdot 10^7$	$6.4 \cdot 10^6$	205	$9.8 \cdot 10^6$	$5.4 \cdot 10^6$	107
5,000	$7.7 \cdot 10^6$	$3.3 \cdot 10^6$	115	$7.2 \cdot 10^6$	$2.9 \cdot 10^6$	37

Table 5.2: The number of VM exits, page faults and PML full events for all implementations. The values decrease for bigger intervals. PML can reduce the number of page faults, early activation more so than late activation.

PML activation very closely resembles those of write protection. For all, the number of PML flushes caused by early PML is roughly double that of late PML/write protection. Since early PML achieves a measurable benefit despite more TLB flushes and outperforms late PML, we conclude that TLB flushes are not prohibitively expensive.

Chapter 6

Conclusion

We modified SimuBoost's checkpointing implementation to use PML for dirty logging. Two variations were added, one that activates PML as early as possible and one that does so during the next downtime. The early variant achieves a measurable performance benefit for small intervals only; at 100ms interval length, the gain is approximately 9%. The performance gain of the late variant is neglectable. Smaller intervals benefit more, but cause a greater absolute slowdown, making them impractical. Longer intervals see hardly any benefit, for 1s it is about 0.5%.

In our analysis of the impact of dirty logging, we created a model to predict the benefit of PML for checkpointing. This model captures progression of the measured time to complete a kernel build, but underestimates it for intervals larger than 100ms. The improvement of this model is left as future work.

We attribute the advantage of early activation to the tendency of a program to reference pages accessed shortly before. A better understanding of the access pattern the workload exhibits might aid in creating a superior model. Our analysis only used accumulated values; an observation of the development throughout the benchmark could provide further insights. Additionally, the results should be reaffirmed by employing other benchmarks, such as SPECjbb, which has a higher intensity, and more powerful hardware, so very small intervals can be considered.

While we found that the additional TLB flushes caused by early activation did not disallow a benefit, its exact effect is yet to be determined.

Bibliography

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

[2] Json. http://www.json.org/, October 4 2016.

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

[4] pexpect. https://github.com/pexpect/pexpect, September 30 2016.

[5] Python. https://www.python.org/, September 30 2016.

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

[7] Advanced Micro Devices, Inc, One AMD Place, Sunnyvale, California, U.S. *AMD64 Architecture Programmer's Manual*, 3.26 edition, April 2016.

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

[9] 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.

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

[11] 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.

[12] 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.

[13] Peter J. Denning and Stuart C. Schwartz. Properties of the working-set model. *Commun. ACM*, 15(3):191–198, March 1972. http://doi.acm.org/10.1145/361268.361281.

[14] Bastian Eicher. Virtual machine checkpoint storage and distribution for simuboost. Master thesis, Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, September04 2015.

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

[16] Alexander Graf, Alex Bennée, Alexey Kardashevskiy, Alex Williamson, Anatol Pomozov, Andre Przywara, Andrey Smetanin, Anup Patel, Avi Kivity, Benjamin Herrenschmidt, Bharat Bhushan, Borislav Petkov, Carlos Garcia, Carsten Otte, Christian Borntraeger, Christoffer Dall, Cornelia Huck, David Gibson, David Hildenbrand, Dominik Dingel, Ekaterina Tumanova, Eric Auger, Eric B, Eric Farman, Gabriel Somlo, Geoff Levand, Gleb Natapov, James Hogan, Jan Kiszka, Jason Herne, Jens Freimann, Linus Torvalds, Liu Yu-B13201, Marcelo Tosatti, Marc Zyngier, Masanari Iida, Michael Ellerman, Michael Neuling, Michael Tsirkin, Mihai Caraman, Nadav Amit, Paolo Bonzini, Paul Mackerras, Rob Landley, Rusty Russell, Sasha Levin, Scott Wood, Stefan Huber, Takuya Yoshikawa, Thomas Huth, Tiejun Chen, and Xiao Guangrong. Documentation/virtual/kvm/api.txt. git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git.

[17] Kai Huang. [PATCH 0/6] KVM: VMX: page modification logging (PML) support. http://permalink.gmane.org/gmane.comp.emulators.kvm.devel/132072.

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

[19] Mateusz Jurczyk and Gynvael Coldwind. Identifying and exploiting windows kernel race conditions via memory access patterns. In *Bochspwn: Exploiting Kernel Race Conditions Found via Memory Access Patterns*, page 69, 102F Pasir Panjang Road, 08-02, Singapore 118530, 2013.

[20] 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.

[21] Avi Kivity, David Matlack, Masanari Iida, Paolo Bonzini, Rob Landley, Takuya Yoshikawa, and Xiao Guangrong. Documentation/virtual/kvm/mmu.txt. `git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git`.

[22] 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`.

[23] Konrad Miller. *Efficient Main Memory Deduplication Through Cross Layer Integration*. PhD thesis, Karlsruhe, Karlsruher Institut für Technologie (KIT), Diss., 2014, 2014.

[24] 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`.

[25] 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`.

[26] 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`.

[27] 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.

[28] 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.

[29] 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.

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

[31] 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.

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

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