Karlsruher Institut für Technologie

Optimizing Continuous Checkpoints for Deterministic Replay

Master Thesis of

Jan Ruh, B.Sc.

at the Operating Systems Group,
Institute of Computer Science & Engineering (ITEC),
Department of Computer Science,
Karlsruhe Institute of Technology

Primary Reviewer:	Prof. Dr. Frank Bellosa
Secondary Reviewer:	Prof. Dr. Wolfgang Karl
Supervising Research Assistant:	Dipl.-Inform. Marc Rittinghaus

Created during: December 01th 2017 – July 15th 2018

Abstract

Functional full system simulation allows monitoring the internal state of a system, including its guest operating system, for detailed analysis. Known functional full system simulators, such as QEMU, have in common that their execution speed suffers considerably, compared to the execution on real hardware. As a result, functional full system simulation cannot be used to analyze interactive and long running workloads.

Rittinghaus et al. propose *SimuBoost* to speed up full system simulation. SimuBoost leverages the almost bare-metal execution speed of a hardware-assisted virtual machine (VM), VM checkpointing, and deterministic record and replay to allow for concurrent, distributed simulation of execution intervals.

The checkpoint mechanism, which creates continuous, incremental checkpoints, is an important component of SimuBoost. We provide an extended analysis of SimuBoost and derive formal requirements for the checkpoint mechanism. We evaluate the existing checkpoint implementation regarding the performance of checkpoint creation and checkpoint loading. We further discuss experiments regarding the working set size of simulation intervals. We find that for a fixed length execution interval of SimuBoost it is sufficient to only load the working set of said interval. As a result, we propose *sparse checkpointing* in order to optimize continuous checkpoints for deterministic replay. Sparse checkpointing leverages access information acquired during checkpoint creation to determine the working set of each checkpoint interval. We use the access information during checkpoint loading to only restore page frames that are in the working set of the respective simulation interval.

Our evaluation shows that sparse checkpointing significantly reduces the average memory footprint of simulations by up to 79 % for a Linux kernel build and decreases the checkpoint loading time by up to 89 %. As a result, sparse checkpointing allows for a higher number of concurrent simulations on a single workstation.

Deutsche Zusammenfassung

Functional Full System Simulation ermöglicht das Überwachen des internen Zustands einer virtuellen Machine (VM), insbesondere des Gastbetriebssystems. Damit lässt sich eine detailerte Systemanalyse durchführen. Bekannte Functional Full System Simulators, wie z.B. QEMU, haben alle gemein, dass ihre Ausführungsgeschwindigkeit nicht an die Ausführungsgeschwindigkeit auf echter Hardware heranreicht. Aus diesem Grund ist Functional Full System Simulation nicht geeignet um lang laufende und interaktive Workloads zu analysieren.

Rittinghaus et al. stellen SimuBoost vor um die Ausführungsgeschwindigkeit von Functional Full System Simulation zu erhöhen. SimuBoost nutzt die fast native Ausführungsgeschwindigkeit einer hardware-beschleunigten VM, VM Checkpoints, und deterministisches Record und Replay um eine parallele und verteile Simulation von Ausführungsintervallen zu ermöglichen.

Der fortlaufende, inkrementelle Checkpoint Mechanismus ist eine wichtige Komponente von SimuBoost. Wir führen eine Analyse seiner Laufzeiteigenschaften durch und leiten formale Anforderungen an den Checkpoint Mechanismus her. Wir werten die vorhandenen Implementierungen bezüglich der Checkpoint Erstellung und des Checkpoints Ladens aus. Außerdem diskutieren wir Ergebnisse bezüglich der Working Set Größe von Simulationsintervallen. Es zeigt sich, dass es für ein Ausführungsintervall von fester Länge ausreicht, wenn wir nur das Working Set des besagten Intervalls laden. Infolgedessen, führen wir Sparse Checkpointing ein um die fortlaufenden Checkpoints für den Einsatz in SimuBoost zu optimieren. Sparse Checkpointing sammelt während des Checkpointerstellens Speicherzugriffsinformationen welche dazu benutzt werden das Working Set für jeden Intervall zu bestimmen. Wir benutzen die Information über das Working Set während dem Laden von Checkpoints um nur Page Frames wiederherzustellen, welche im Working Set des entsprechenden Simulationsintervals liegen.

Unsere Auswertung von Sparse Checkpointing zeigt, dass es den Speicherverbrauch der Simulationen für einen Linux Kernel Build um bis zu 79 % und die Checkpointladezeit um bis zu 89 % verringert. Infolgedessen ermöglicht Sparse Checkpointing eine größere Anzahl an parallelen Simulationen auf einer einzelenen Workstation.

Danksagungen

Ich möchte mich an dieser Stelle bei allen bedanken, die mich während meines Studiums unterstüzt haben, auch bei allen welche hier nicht namentlich erwähnt werden.

Zunächst möchte ich mich bei Marc für die coole Zeit bedanken. Danke, dass du mir die Chance gegeben hast an SimuBoost mitzuarbeiten und mir geholfen hast mich fachlich weiterzuentwickeln. Ich habe durch die Arbeit mit dir sehr viel gelernt und auch viel Spaß an der Arbeit gehabt.

Außerdem, will ich mich bei Anne bedanken, auch wenn wir es nicht immer ganz einfach hatten, hast du mich doch immer wieder daran erinnert, dass es noch etwas anderes gibt :-).

Großer Dank geht auch an meine Großeltern, welche mich wo sie konnten unterstützt haben und für mich da waren. Ich besuche euch immer gerne für eine Auszeit im Garten und eine Runde Rommé ;-).

Zu guter Letzt möchte ich mich bei meinen Eltern bedanken. Danke Mama und Papa für die unzähligen Stunden die ihr mit euren „winkenden Freunden“ verbracht habt damit ich studieren kann. Ihr habt mich immer *bedingungslos* unterstützt, was ich wirklich sehr zu schätzen weiß!

Contents

Chapter 1

Introduction

Functional full system simulation utilizes interpretation or dynamic binary translation (DBT) in order to reassemble a complete virtual computer system. It is possible to leverage interpretation or DBT to monitor the execution of a system simulation, including the guest operating system, without any functional side effects on its execution.

Todays computer scientists utilize full system simulation due to its extended analysis capabilities in order to perform system analysis and computer system forensics [13, 19, 41, 49]. Known full system simulators are, e.g., QEMU [9], Simics [24], and Bochs [1]. They all have in common that their execution speed suffers compared to the execution on bare-metal hardware, depending on the level of detail of the simulation. For example, QEMU utilizes dynamic binary translation and trades a good simulation performance for less accurate hardware emulation, yet it slows down execution by a factor of about $33\times$ [40] depending on enabled analysis extensions. As a result, a workload that is to be analyzed, which runs for one hour on real hardware, takes at least 33 hours to simulate, not considering additional analysis overhead. This renders the use of full system simulation unfeasible for the analysis of long running and interactive workloads.

Rittinghaus et al. [40] propose *SimuBoost*, a deterministic, heterogeneous *record and replay* system that aims to speedup full system simulation. SimuBoost leverages VM checkpointing to split the execution of a hardware-assisted VM into intervals of fixed length. SimuBoost concurrently records non-deterministic events that affect the course of execution of the hardware-assisted VM. The resulting execution intervals are distributed to simulation nodes. Each simulation node utilizes the recording logs to replay non-deterministic events in order to recreate the exact same execution as in the hardware-assisted VM for each finite simulation interval.

An important component of SimuBoost is the checkpoint mechanism that creates continuous, incremental checkpoints of the hardware-assisted VM. Therefore, a series of theses were concerned with the development and optimization of the checkpoint mechanism itself and the storage and distribution of checkpoints [15, 18, 23, 38].

Pusch [38] encountered high checkpoint loading times in his experiments that show the potential of further optimization of the checkpoint mechanism. In contrast to existing work on the checkpoint mechanism of SimuBoost, which focused on checkpoint creation [15, 18], we focus on the optimization of checkpoint loading regarding the checkpoint data amount that is to be restored and the checkpoint loading time.

Therefore, we propose *sparse checkpointing* in order to optimize continuous checkpoints for deterministic replay. We utilize the accessed bits, which are set in the hardware-assisted VM's extended page tables, in order to determine the working set of each checkpoint interval. Furthermore, we implement a checkpoint data retrieval mechanism which only fetches those page frames that are contained in the working set of a specific checkpoint. We found that sparse checkpointing reduces checkpoint loading times up to 89 % depending on the workload. Furthermore, we are able to cut down the average main memory footprint of a simulation by up to 79 %.

Chapter 2 discusses related work and introduces concepts needed to follow the subject matter of this work.

In Chapter 3, we provide an extended analysis of SimuBoost. First of all, we discuss theoretical models describing its runtime performance. Furthermore, we examine the effect of performance properties of the checkpoint mechanism, e.g., downtime, runtime overhead, and checkpoint loading time, on the achievable speedup. Finally, we derive abstract requirements on the checkpoint mechanism and evaluate the existing checkpoint implementation regarding checkpoint creation and checkpoint loading. We continue our analysis arguing about the working set size of simulation intervals by discussing the work of Werner [47]. We conclude that the checkpoint mechanism must not load the complete main memory during checkpoint loading but rather only the working set, which is a fraction of the complete main memory.

In Chapter 4, we discuss the design and implementation of sparse checkpointing for KVM and QEMU.

Chapter 5 provides an evaluation of our approach. We have performed experiments showing the correctness of our sparse checkpointing, the performance of checkpoint creation and checkpoint loading. Furthermore, we have conducted general tests demonstrating the improved main memory utilization that comes with sparse checkpointing.

Finally, Chapter 6 concludes this thesis and provides an outlook of future work.

Chapter 2

Background

This chapter sets the know-how needed to follow the subject matter of the work at hand. We introduce virtual machines (VMs), their implementation, and their utilization in form of VM checkpointing. We establish the concept of deterministic record and replay (RR) and its applications. Finally, we induct into SimuBoost, an acceleration method for functional full system simulation, which leverages checkpointing and heterogeneous RR. Furthermore, we point out its relevance for the field of full system simulation.

2.1 Virtual Machines

Modern computing systems and the software running on top of it rest upon multiple layers of abstraction. Each layer of abstraction hides the complexity of the underlying implementation. At the hardware level a computer consists of a central processing unit (CPU), volatile memory, persistent memory, peripheral devices, and interconnections. The CPU is able to execute a predefined set of instructions depending on its architecture and its mode of operation. Current CPUs provide different privilege levels that restrict the set of instructions available to the program executing on the CPU. The instruction set of a CPU, the CPU's registers, and the way the CPU communicates with devices (memory controllers, peripheral devices) determines the instruction set architecture (ISA) of a computer system.

The ISA describes a well-defined interface which the operating system (OS) utilizes. The OS kernel runs with the highest privileges in kernel mode. It manages the system's hardware and provides software abstractions, such as virtual memory and processes, communication facilities, such as inter-process communication, and resource management, such as task scheduling. Applications run with less privileges on top of the OS in user mode so they are not able to interfere with each other or the OS. User mode applications are not allowed to access hardware

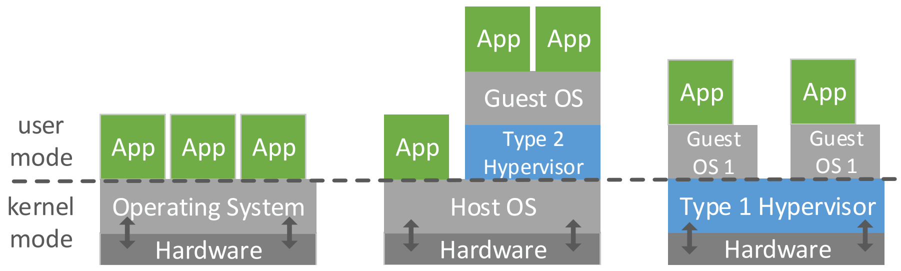


Figure 2.1: Illustrating a general monolithic OS, a type-2, and a type-1 hypervisor. In general, the OS operates in privileged kernel mode and user applications run in unprivileged user mode (left). A type-2 hypervisor depends on a host operating system (middle), whereas a type-1 hypervisor is a standalone VMM running exclusively in kernel mode (right).

directly. As a result, the OS exports functionality, for example, to conduct I/O or allocate memory, to user space through the system call facility.

A virtual machine (VM) allows the execution of an unmodified OS in an encapsulated execution environment. Henceforth, we refer to the system that executes a VM as the host and to the VM itself as the guest. The hypervisor is an application that runs directly on the host and implements the runtime environment for VMs by multiplexing the physical hardware of the host, or emulating devices and providing virtual counterparts for use by the VMs. Therefore, the hypervisor inserts an additional layer of abstraction hiding the underlying host system.

Historically, Goldberg [26] depicts the introduction of privilege levels to system architecture and the resulting novel problems as the main reason for the emergence of VMs in the 1970s. Goldberg describes the part of the OS that provides privileged functionality to unprivileged user space software as the privileged software nucleus. With the operating system running in privileged mode it was no longer possible to execute legacy applications or diagnostic software, which required privileged instructions, since the functionality, previously provided by calling instructions of the bare machine directly, was now exported through the privileged software nucleus. Furthermore, testing and debugging of operating systems was limited, as only one privileged software nucleus is allowed to execute on the bare machine. The only solution was time sharing the machine, requiring repeated shutdown of the computer system and initialization with the desired user software. VMs resolved this problem, allowing for the execution of multiple privileged software nucleuses.

There are different ways to implement a VM. Goldberg [26] defines a virtual machine monitor (VMM) as an application that offers the execution of VMs, whereat unprivileged instructions are directly executed on the host CPU and privileged instruction are handled by the VMM, e.g., by emulating them. In todays terms, a VMM leverages virtualization technology of modern CPUs to provide an almost bare-metal execution speed. There are two different types of VMMs: type-1 and type-2 as depicted in Figure 2.1. A type-1, or bare-metal VMM, implements a small software layer running in privileged mode, that abstracts and multiplexes the hardware, allowing for VMs to run on top. Type-2 hypervisors, so-called hosted hypervisors, rely on a host OS in order to provide virtual devices or multiplex existing devices and to schedule the execution of the VM. For our work only hosted hypervisors are of interest.

An example for deployment of a hosted hypervisor is full system simulation. In this case a simulator, running completely in unprivileged user mode of a host OS, is in control of the VM and modifies its internal state according to the instructions it reads from the guest's main memory. This process goes along with a vast slowdown of up to 771$\times$ compared to native execution [40] which renders it useless for performance critical applications.

Today, there are many areas of application for VMs. Virtualization technology is the foundation of cloud computing. Cloud service provider, like Google or Amazon, deploy several VMs on high performance servers, because VMs offer a high level of scalability. VM instances can be spawned, migrated, and destroyed depending on the number of requests that need to be served. VMs offer extended load balancing facilities depending on respective criteria. Jain et al. [32] provide a survey of current load balancing methods all of which depend on the flexibility of VMs.

On the other hand, VMs offer extended analysis capabilities, as their complete system state is easily observable by utilizing the VMM or a full system simulator. As a result, they are frequently deployed in research. A popular example is intrusion detection. Often malicious code adapts its behavior if it is being monitored by an intrusion detection software to complicate analysis or in general its discovery. Garfinkel et al. [25] leverage the VMM and VM introspection in order to move intrusion detection out of the inspected system itself, so malicious code is not able to detect the analysis software. Schmidt [43] evaluates VM memory tracing mechanisms based on virtualization and full system simulation. Wilhelm [48] utilizes VM memory tracing to uncover security vulnerabilities of shared memory pages that are used for communication between different protection domains of the Xen hypervisor [14].

Besides cloud computing and research, VMs are deployed on desktop systems as well. Desktop virtualization solutions like VirtualBox [12], the kernel virtual machine (KVM) [4] or QubesOS [42] provide VMs for daily use.

The discussed areas of applications of VM's differ in the requirements they post on the virtualization technology that is deployed. A VMM that is in use in a cloud environment needs to fulfill high performance requirements in order to guarantee an optimal runtime behavior of deployed applications. On the other hand, a hypervisor that is applied in research to perform system memory traces has to allow for a high degree of instrumentation, so researchers gain detailed insight into the execution behavior of an investigated software. In this case, runtime performance is subordinate to the objective to gain new insights of a given system.

For the remainder of this section we present different implementations of VMs that differ in their respective performance and the degree in which they can be instrumented for system analysis.

2.1.1 Emulation

Emulation is a way to implement a system VM completely in user mode. In general, emulation aims to execute an application binary compiled for a source ISA on a target ISA [22][1]. The source ISA corresponds to the guest and the target ISA corresponds to the host. For the sake of simplicity, we only discuss instruction set emulation and skip interrupt and exception handling, which is described by Smith et al. [22]. In case of emulation, a user space component is in control of the VM's execution and its complete state, including CPU, main memory, and devices. Smith et al. denote the subcomponent that is control of the VM's execution as emulation manager (EM).

Interpretation is a straightforward approach to emulation. The EM repeats a *decode-and-dispatch* loop that fetches and decodes an instruction from the guest's main memory and modifies the VM's state according to the instruction. As a result, the EM advances the execution of the VM step by step. Interpretation is a simple implementation of emulation, although Smith et al. point out its bad runtime performance due to the general structure of the decode-and-dispatch loop. Bellard [16] states that interpretation is up to 30 times slower than dynamic binary translation (DBT). There are optimizations that increase the runtime performance of interpretation such as *predecoding*, which translates source instructions into a less dense, but easier to interpret format, and *direct threaded interpretation*, which reduces overhead due to indirection. All optimizations of interpretation are described in detail by Smith et al. [22]. He also extensively discusses benefits and drawbacks regarding runtime performance, memory consumption and code portability that come with the respective methods.

Dynamic Binary Translation

Emulation utilizing interpretation leads to a VM execution in which each type of instructions, e.g., basic arithmetic instructions, regardless of the used registers or

[1]The target and the source ISA may differ. In this particular work we are only interested in emulation for which source and target ISA are the same.

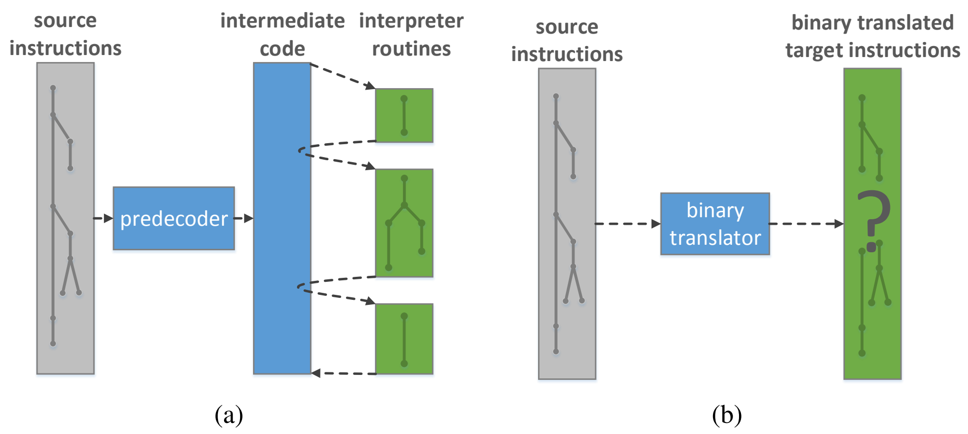


Figure 2.2: Structural comparison of direct threaded interpretation with predecoding and binary translation. (a) The predecoder maps source instructions (gray) to intermediate code (blue) which in turn maps to interpreter routines implemented by target instructions (green). (b) The binary translator directly maps source instructions (gray) to target instructions (green).

memory addresses, is simulated by a dedicated interpretation routine. A single interpretation routine is a self-contained set of target instructions that alter the state of the VM according to the given source opcode and operands. In other words, each dispatch routine for a specific instruction type maps source instructions to a set of target instructions. The execution itself is an interlinking of dispatch routines whereat each subsequent call of the next dispatch routine involves repeated calls to the EM as depicted in Figure 2.2(a).

An alternative approach is *binary translation* (BT) [22], that directly maps a set of source instructions to a set of target instructions. In contrast to interpretation, the execution of a set of binary translated target instruction requires no more interference of the EM as shown in Figure 2.2(b). Therefore BT results in a contiguous set of target code that emulates given source code.

There are two different approaches to BT – static and dynamic BT. First of all, static BT [22] aims to translate the complete set of source instructions before actual system execution. Static translation runs into problems once a jump instruction is to be translated. The target of a jump instruction may depend on the content of one or multiple registers that are not assigned before runtime. Furthermore, ISAs with a variable sized instruction format, e.g., x86, allow for the start of new instructions at every byte boundary, in contrast to ISAs with a fixed length instruction format that guarantee word boundaries for instructions. These facts complicate *code discovery* as it is not clear whether a byte starts a new instruction, ends a previous instruction, or if it is data.

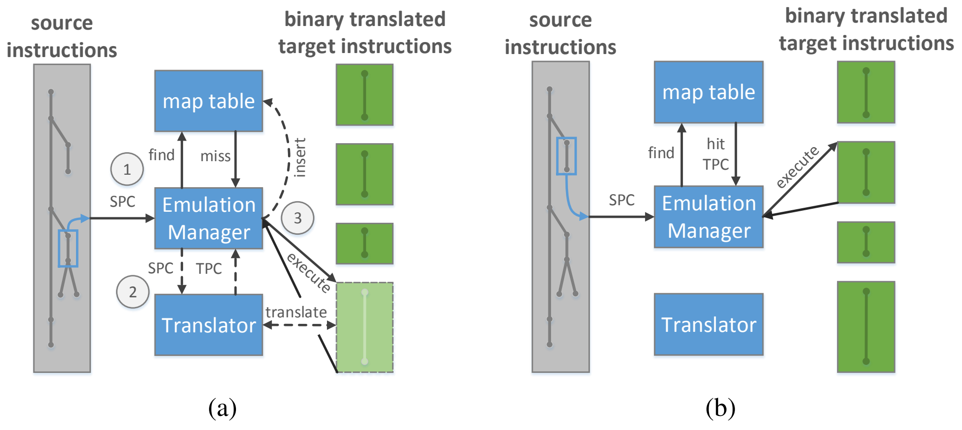


Figure 2.3: Visualizing two cases frequently occurring in dynamic binary translation. (a) If there is no entry in the mapping table for a given source program counter (1), translate the dynamic basic block (2), create a mapping from the source program counter to the target program counter in the table, and execute the translated target code (3). (b) If there already exists an entry in the table for a given source program counter, return the mapped target program counter and execute the associated target code block.

Another problem of static translation is the so-called *code location* problem [22]. BT maps a single source instruction to multiple target instructions. As a result, the target requires an additional program counter, since it generally requires more target instructions to express the same semantic as the original source instruction. Smith et al. [22] refer to the source instruction pointer as source program counter (SPC) and to the target instruction pointer as target program counter (TPC). The EM derives targets of indirect jump instructions at runtime and references the SPC in the target instruction stream. Since the SPC is not valid in the context of translated target instructions, a mapping from SPC to TPC is required. Establishing this mapping statically, before the execution, is not feasible since jump targets may not be determined before system execution.

A solution to both, the code discovery and the code location problem, is to dynamically read and translate blocks of source instructions. This process is henceforth referred to as *dynamic binary translation* (DBT). In context of DBT, a block of source instructions that is translated to a set of target instructions at once is called a dynamic basic block (DBB). A DBB starts with the first instruction after a jump and ends with the occurrence of a jump or branch in the instruction stream. DBT solves the code discovery problem since source instructions are revealed incrementally block by block. Furthermore, DBT allows for the maintenance of a data structure, e.g., a hash table, to keep track of SPC to TPC mappings, thus dealing with the code location problem.

Figure 2.3(a) visualizes the process of translating a new, unknown DBB of source instructions. Once the EM discovers a new DBB, the translator generates the corresponding target instructions and returns the address of the first instruction of the block of target instructions as TPC to the EM. The EM creates a mapping from SPC to TPC and executes the newly translated bock of target instructions. Figure 2.3(b) depicts a case in which there already exists a translation for a given SPC. Therefore, the EM obtains the TPC from the mapping table and is able to directly execute the block of target instructions pointed at by the TPC. DBT enhances performance of the emulation compared to simple interpretation as block of target instructions are reused. This is particularly beneficial for recurrent blocks of source instructions, as found, e.g., in loop constructs.

Memory Virtualization

Besides CPU emulation, an EM has to provide main memory to the VM. From the VM's point of view there needs to be a contiguous physical address space. An EM is a user mode program, hence it operates on a *virtual address space* that is constructed by the underlying OS. The OS establishes and destroys mappings from virtual to physical addresses so dedicated virtual addresses that are actually accessed by any user software are backed with physical memory. This process is transparent to the user space software thus an EM may simply assume there is host main memory of a fixed size available that can even exceed the actual available physical main memory of the host.

We refer to the notion of what the guest expects to be physical memory as guest physical memory. The corresponding addresses are referred to as *guest physical addresses* (GPAs). Furthermore, virtual addresses used by user mode applications of a guest are referred to as *guest virtual addresses* (GVAs). Similarly, we talk about *host physical addresses* (HPAs) if we address actual physical memory and *host virtual addresses* (HVAs) if we talk about virtual addresses in the context of a host application. We presume the deployed target ISA implements virtual memory management using a form of paging, hence the CPU utilizes multiple linked, hierarchical tables to translate virtual to physical addresses. The actual process of translating dedicated addresses on a physical machine is performed in hardware in order to minimize its runtime.

An emulator implements the complete process of address translation in software. This means in case of a memory access, the emulator looks up the active page table given by the page table register (e.g., the CR3 register on x86) and follows the links in the guest page table entries in order to find the GVA to GPA mapping. This process is also called a *page table walk*. Performing page table walks completely in software degrades the performance of a system since main memory accesses are a very frequent operation of a computer system. Therefore, emulators [16] implement a so-called software translation look-aside buffer that improves the performance of GVA to HVA lookups by caching recent translations.

QEMU

A practical implementation of DBT is deployed by the QEMU full system simulator [16][2]. QEMU utilizes the tiny code generator (TCG) to translate source to target instructions. The TCG introduces an RISC-like, ISA independent intermediate instruction format, that is referred to as microcode by Bellard [16]. As a result, one only needs to implement a single translation direction in order to add a new guest or host ISA for all supported guest or host ISAs. For example, if we want to add RISC-V support for all existing host ISAs it is sufficient to implement the translation from RISC-V to microcode. Therefore, the TCG enhances the portability of QEMU. At the moment QEMU supports following guests x86, x86-64, PowerPC, ARM, Alpha, Sparc and MIPS [9] to name only a few. For the remainder of this paper, we are only interested in x86-64 guests running on x86-64 hosts.

Figure 2.3 introduces the concept of a target code cache since the EM maintains a mapping of DBBs to corresponding blocks of target instructions. This mapping is represented by the table that maps SPCs to TPCs. QEMU utilizes such a code cache as well. In the context of QEMU, a block of target instructions is referred to as a *translation block* (TB). A major performance drawback of the conceptional approach to DBT depicted by Figure 2.3 is the fact that the execution returns to the EM after each block of target instructions, even if the subsequent block of target instructions already exists and is determined. Therefore, QEMU implements *TB chaining*. Whenever the EM identifies contiguous DBBs it connects the blocks by replacing the return to the EM with a jump to the subsequent TB. As a result, QEMU accelerates the emulation since big chunks of target instructions can be executed without any interference by the EM.

QEMU internally organizes the guest's main memory as an acyclic graph of nested *memory region* objects. A memory region object is of one of five types [10] – a RAM region, an MMIO region, an IOMMU region, a container simply including other regions, or an alias region. A RAM region is a range of directly readable and writable addresses that appear to a guest like common RAM. A MMIO region on the other hand is implemented as neither readable nor writable thus every access to it is handled by a host callback function. Finally, an alias region allows splitting a region into a set of sections as it always points to another memory region that actually handles accesses made to it.

[2]Bellard identifies the guest system as target in his paper. Therefore, he uses a reverse nomenclature compared to the one we use in this work.

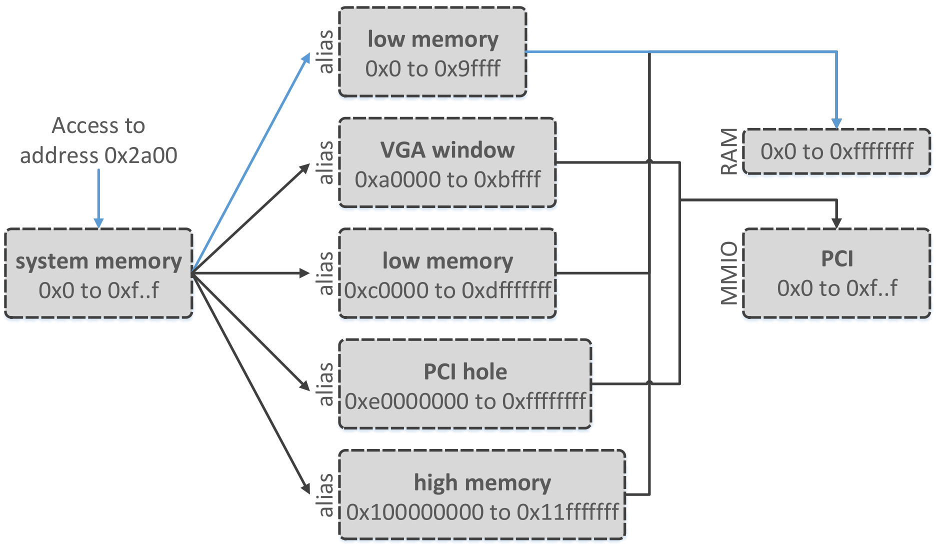


Figure 2.4: Showing the structure of a simplified memory region graph as deployed by QEMU. From the VM's point of view the system memory consists of a contiguous address range. Internally, the system memory consists of five alias memory regions each pointing either to a RAM region or an MMIO region. If an access is made to a specific address (blue), QEMU traverses the nested memory regions in order to determine the target memory region and trigger the corresponding action.

Figure 2.4 visualizes a memory region graph as deployed by QEMU. In this example, the VM is configured with 4 GiB of RAM. QEMU maps the first 3.5 GiB of the VM's guest physical main memory one-to-one to the respective addresses starting with address 0x0, with one exception. The VGA window, containing the frame buffer of the graphics system, overlaps the low memory area at address range 0xa0000 to 0xbffff. As a result, if the guest accesses this region, those accesses are handled by the MMIO region. The PCI region is placed at the 3.5 GiB mark of the guest physical address space. The remaining 512 MiB of RAM are mapped to guest physical addresses 0x100000000 to 0x11fffffff. For simplicity, the MMIO region representing the PCI address range is not shown in detail. GPAs targeting the PCI memory region would trigger respective callback functions that manipulate the corresponding virtual device or general VM state. GPAs that in turn target guest physical memory are mapped to host virtual addresses by simply shifting the GPAs by a fixed offset.

In summary, QEMU's implementation of DBT achieves a reasonable performance compared to other full system simulators. Rittinghaus et al. [40] compare the performance of QEMU to the timing accurate full system simulator Simics. They emulate multiple workloads in VMs provided by QEMU and Simics respectively.

They found that QEMU tends to induce a slowdown of 22–33 times, whereas Simics induces a slowdown of 624–771 times, compared to executing the same workload on a bare machine. Please note, Simics provides additional functionality such as the usage of timing models that justify its high overhead whereas QEMU is optimized for maximal performance. Nevertheless, even though QEMU's performance is reasonably good compared to Simics, it is not feasible to use it for analysis of long running or interactive workloads nor to deploy it for production workloads, e.g., in a cloud environment.

2.1.2 Hardware-Assisted Virtualization

In Section 2.1.1, we have introduced emulation as a way to drive VMs. However, we have noticed the vast overhead that comes with interpretation or even DBT. Therefore, emulation is not applicable in performance critical environments like cloud computing or for the analysis of long running or interactive workloads in research (e.g., memory traces of production systems).

The overhead of DBT stems from the translation of source instructions to target instructions even if the source and target ISA are the same. Therefore, it seems obvious to execute as many guest instructions as possible directly on the host's CPU in order to reduce the runtime overhead. This approach is referred to as *direct execution*. A VMM that implements direct execution processes instructions that do not modify system resources, or depend on the state of system resources, directly on the CPU. All other instructions must trap and being handled by the VMM for example by emulating them. We refer to this process to handle privileged instructions as *trap-and-emulate*.

Smith et al. [22] point out that trap-and-emulate is only feasible as long as all instructions that directly modify system resources are privileged, thus trap if they are executed in user mode. This fact was first described in the work of Popek and Goldberg [37]. They define instructions that directly modify system resources as *control sensitive* and instructions whose behavior or result depends on system resources as *behavior sensitive*. They propose a theorem stating that a VMM for an ISA can be constructed, if the set of sensitive instructions is a subset of the privileged instructions of that ISA. We further refer to instructions that are sensitive, but not privileged, as critical instructions.

An example for an ISA that does not fulfill the theorem of Popek and Goldberg is x86. The `popf` instruction of x86 pops the flags register from a stack in memory, including the interrupt-enable flag that is only modifiable in privileged mode. If executed in user mode, the instruction overrides all flags except the interrupt-enable flag and does not trap. As a result, the VMM does not notice the execution of a `popf` instruction that is intended to be a privileged `popf` and is not able to emulate it leading to a guest OS behavior that is inconsistent with the behavior on real hardware.

Hardware-assisted virtualization aims to solve this problem allowing for the execution of the majority of instructions directly on the host CPU. As a result, the performance of VMs almost catches up with native execution depending on the deployed workload. Hardware vendors, such as Intel or AMD, introduced hardware-extensions to their ISA implementations in order to enable hardware-assisted VM execution.

Intel VT-x

Intel provides its *VT-x* technology [31] for x86 that allows for the native execution of all instructions. Intel achieves this by introducing a hardware extension called virtual machine extension (VMX). With VMX the CPU knows two additional modes of operation, namely VMX root and VMX non-root mode.

In VMX root mode, the CPU is able to use all instructions and to access and manipulate critical system resources. The VMX root mode relates to the normal mode of an Intel x86 processor that offers four privilege levels or so-called rings. Ring 0 corresponds to the highest privilege level and ring 3 to the lowest privilege level. As a result, a VMM, executing on a system with Intel's VT-X technology, runs in VMX root mode in ring 0 so it has full access to all system resources and is able to multiplex them for VMs. The VMs on the other hand execute in VMX non-root mode.

VMX non-root mode is fully transparent to a guest. It also provides the familiar four rings to a VM, so the guest OS can run in ring 0, whereas user applications run in ring 3. Therefore, most instructions of a VM are executed natively. Still, in VMX non-root mode the CPU is not allowed to read or manipulate critical system resources. If a guest tries to access system resources, the VMM interferes and emulates the requested resource. As a result, the VMM running in VMX root mode is in full control of the system, ensuring strict isolation of multiple VMs from each other and the host OS.

Figure 2.5 illustrates the life cycle of a VMM starting a VM that runs a guest OS. The VMM operates in VMX root mode. Whilst in VMX root mode, the VMM can start new VMs by issuing a `vmlaunch` instruction. This triggers a VMX transition from root to non-root mode. Such a transition is also referred to as *VM enter*. On the other hand, a transition from non-root to root mode is called a *VM exit*. For example, a `vmcall` executed in non-root mode or a page fault occurred in non-root mode triggers a VM exit to VMX root mode, hence the VMM context.

VM entries as well as VM exits are controlled by a *virtual machine control structure* (VMCS) [31]. The VMM uses one VMCS for each VM. The processor maintains a state variable, called VMCS pointer that points to the memory region

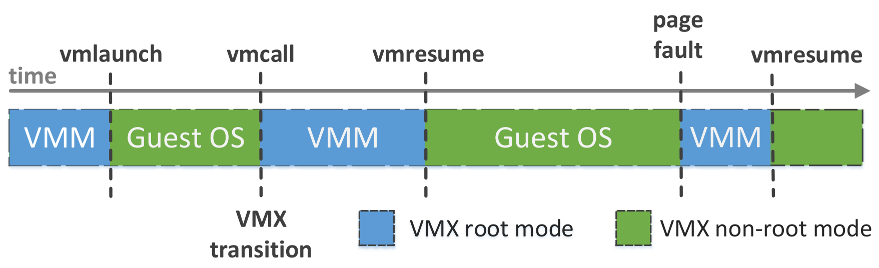


Figure 2.5: Showing the life cycle of a VMM utilizing Intel's virtual machine extension (VMX) [31]. The CPU is already in VMX root mode. The VMM starts a VM by calling the `vmlaunch` instruction. A `vmcall` or critical resource access (e.g., a page fault) invokes the VMM again. If the execution of a VM was interrupted, the VMM can resume its execution with the `vmresume` instruction. A VMX transition describes the change of VM mode from root to non-root or the other way around.

that contains a VMCS. Besides general VM state information and control fields the VMCS contains a *guest-state area* and a *host-state area*. The guest-state area includes processor state of the VM, whereas the host-state area includes processor state of the host, respectively. At the occurrence of a VMX transition, e.g., a VM entry, the CPU switches to VMX non-root mode and stores the current host state in the host-state area of the current VMCS and restores the guest's CPU state from the guest-state area.

Memory Virtualization

Hardware-assisted VMs execute a large portion of instructions directly on the host's CPU. Therefore, they maintain their own sets of page tables since the VMM is completely transparent. Yet, the CPU never uses the page tables setup by a guest because this would break the isolation of VMs and the VMM. The VMM rather maintains its own set of page tables, the so-called *shadow page tables* (SPTs), for each VM. The SPTs contain mappings of GVAs to HPAs and are actually used by the CPU. The VMM must ensure that the SPTs are synchronized with the guest OSs internal page tables at all times. Therefore, the instructions used to manipulate the page tables on an ISA either trap into the VMM or the VMM must write protect the memory area that contains the guests page tables in order to keep track of modifications by the guest OS.

The maintenance of SPTs introduces overhead to VM execution since the VMM traps every time the guest OS manipulates its internal page tables. Hardware vendors introduce a mechanism called *second level address translation* (SLAT) to their architectures in order to counter this performance degradation. SLAT ren-

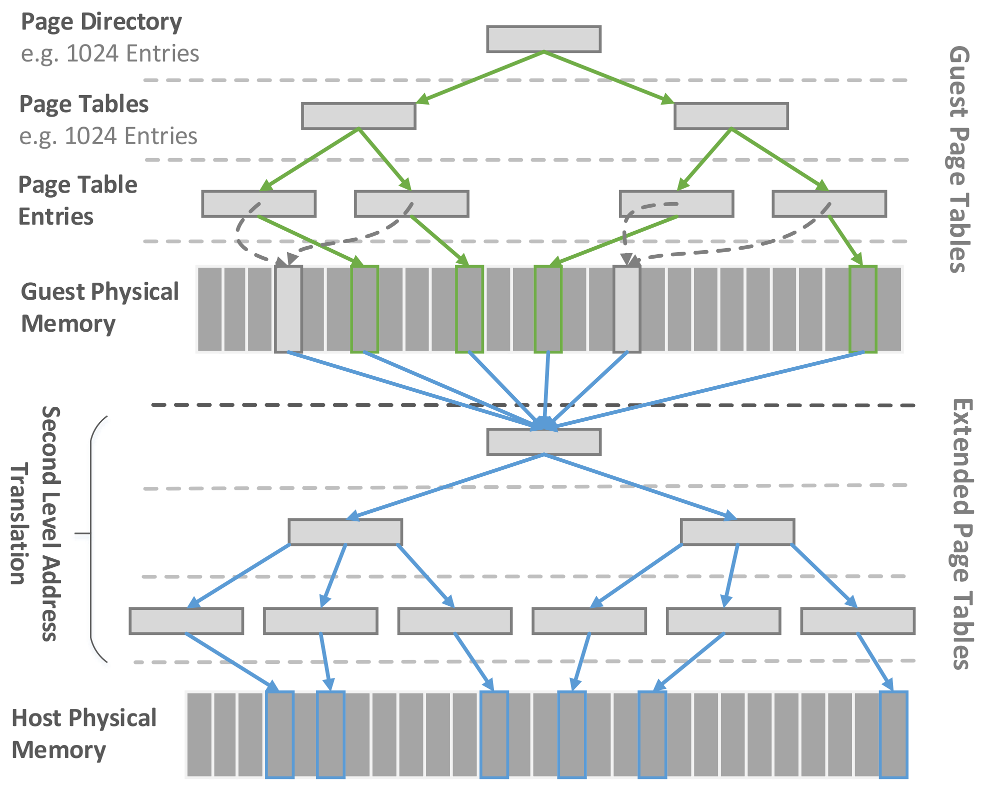


Figure 2.6: Illustration of Second Layer Address Translation (SLAT) – The CPU uses both, the internal page tables of the guest and the extended page tables of the VMM for address translation. In fact, the page table walk of the guest's internal page tables (green arrows) triggers second layer address translations since, e.g., page directory entries reference page tables by GPAs (gray dashed arrows). The GPAs must be translated to HPAs (blue arrows) in order to find the guest's internal page table entries which in turn point to GPAs that are translated to HPAs using the extended page tables.

ders it possible to forgo SPTs and use direct hardware support for guest address translation.

For this work, we go into detail about Intel's implementation of SLAT. Intel introduced *extended page tables* (EPTs) [31] to their x86 CPUs in order to diminish the performance degradation of system VMs due to maintenance of SPTs. If EPTs are enabled the hardware uses both, the internal page tables of the guest OS, which map GVAs to GPAs, and additional page tables that map GPAs to HPAs as visualized in Figure 2.6. If the guest accesses a GVA, the CPU reads the address of the guest's page tables from the CR3 register and issues a page table walk in hardware. During this page table walk, each access to a GPA entails a second layer address translation of GPA to HPA. This mechanism effectively results in an address translation process from GVAs to HPAs that is completely implemented in hardware.

Bhatia [17] provides data regarding the performance gain of Intel's EPTs compared to memory virtualization via SPTs. He executed several micro and macro benchmarks in VMs on the VMWare ESX VMM. He showed performance differences of the VMWare ESX implementation only relying on software-based memory virtualization, compared to the VMWare ESX implementation that leverages Intel's EPT technology. Bhatia found that for memory management unit (MMU) intensive micro benchmarks the gains in performance of EPTs go up to 600 % compared to the software-only solution. For other MMU intensive macro benchmarks the performance gain is still up to 48 %.

Kernel Virtual Machine (KVM)

KVM is a Linux kernel module that provides implementations of Intel's virtualization technologies VT-x and EPTs, in order to provide hosted VMs on Linux. KVM is no standalone VMM as it always depends on a user space counterpart that provides an execution context that is a common process in Linux. This means the VM is transparent to the Linux host. As a result, the VM is subject to the Linux scheduler just as every other process.

KVM exports functionality to user space through a well-defined interface composed of a set of ioctls [5]. These ioctls split into three groups each of which provide a dedicated functionality to the corresponding user mode component. The logical structure of the ioctls is hierarchical, from system, over VM specific, to per virtual CPU (VCPU) ioctls.

In general, the user mode component interfacing with KVM is QEMU. If QEMU is started with KVM enabled, the CPU emulation capabilities of QEMU, namely TCG, are disabled and the VM runs in hardware-assisted mode, leveraging the virtualization extensions provided by KVM. Therefore, QEMU only provides software implementations of certain devices and chipsets depending on the specific configuration of the VM. The device and chipset interfaces are mapped into the VM's address space by QEMU's guest memory abstraction, effectively emulating memory mapped I/O.

Figure 2.7 visualizes the VM creation and kickoff of a QEMU/KVM managed VM. For convenience, the illustration uses slightly altered function names and we omit some intermediate functions, yet in general it reproduces the essential course of events of the interaction of QEMU and KVM. The Linux kernel sets up basic VMX data structures and enables EPT during system initialization. If a QEMU/KVM process is created, the main thread initializes KVM by calling the respective ioctl. The kernel serves the ioctl by creating a new VM context and enabling VMX, hence transferring the CPU into VMX root mode. The main thread

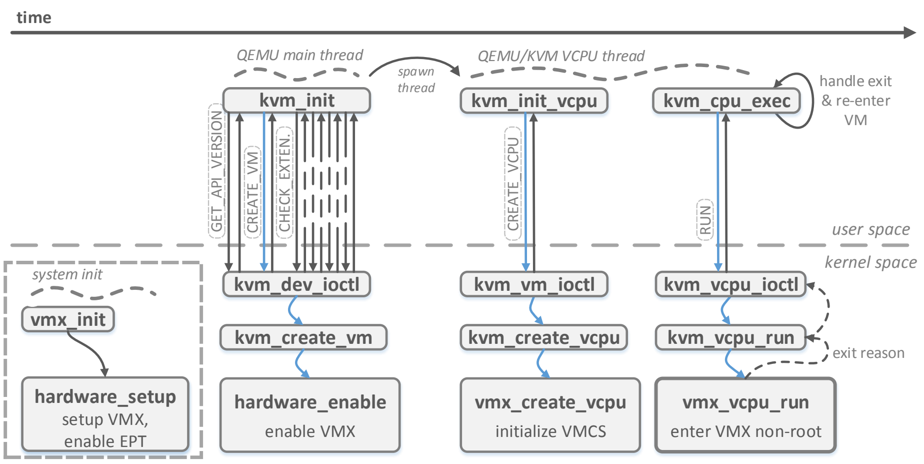


Figure 2.7: Illustrating the creation of a new VM using QEMU/KVM. The dashed box at the bottom left depicts the basic initialization of VMX data structures and activation of extended page tables by Linux during the host boot process. Once a QEMU main thread with enabled KVM acceleration is started, the `kvm_init` *function creates a new VM context and checks for available extensions. Afterward, a new virtual CPU (VCPU) thread is created. The new thread calls* `kvm_init_vcpu` *that creates a new VCPU and starts VM execution by calling* `kvm_cpu_exec`*.*

spawns a new thread that serves as VCPU thread once the VM context is created. The VCPU thread creates and initializes VCPUs by a call to the respective ioctl. The kernel creates new VCPUs and initializes the VCPU's VMCSs in response to the ioctl. The execution continues with the central execution loop. In the loop there exist checks for queued events that need handling in user space, e.g., device emulation or I/O. If there is no event waiting, a `KVM_RUN` ioctl on the VCPU results in the kernel entering VMX non-root mode, hence the execution context of the VM. If the VM exits back to VMX root mode due to an event that requires interference of the VMM, the exit reason is passed back up to QEMU so the VM exit is handled in user space.

Besides the implementation of Intel's VT-x, KVM leverages EPTs to speedup GVA to HVA translation. We have to consider that KVM is not a standalone hypervisor and depends on QEMU as user space component. As discussed in Section 2.1.1, QEMU already establishes a GPA to HVA mapping that may conflict with the EPT mappings created by KVM. Therefore, QEMU must share its guest memory layout with KVM, so KVM is able to create GPA to HVA mappings in the EPTs that are consistent with QEMU's view on guest main memory.

Furthermore, since the QEMU process is a common Linux process, the Linux host may swap or even free guest main memory if there is high memory pressure on the system. For this purpose, KVM registers a so-called *memory listener*. As a result, KVM is being notified if Linux reclaims a page frame that is in use by a VM and KVM invalidates the affected EPT entries.

Rittinghaus et al. [40] performed benchmarks using QEMU/KVM. They found that the slowdown induced by KVM compared to executing the benchmark on a bare machine is negligible for the specific benchmarks they deployed[3]. In contrast, executing the same benchmarks in QEMU with emulation results in a slowdown of factor 22 to 33 compared to the bare machine. Therefore, QEMU/KVM is feasible for applications with severe performance requirements, e.g., cloud computing or superficial system analysis of long-running production workloads in research.

2.2 Checkpointing

Running production software systems in VMs, in particular hardware-assisted VMs, provides several advantages compared to the deployment of a single OS with respective user software on a bare machine. First of all, the VMM is in full control of the execution of its VMs. The VMs running on a single host are encapsulated systems that can interact with each other using predefined, VMM controlled interfaces, yet they cannot interfere with each other or the VMM.

The availability of the complete VM state in the context of the VMM allows for the creation of VM *checkpoints*. A VM checkpoint is composed of the VM's VCPU state, main memory, disk and device states. VM checkpoints introduce a certain flexibility and scalability to the application areas of VMs, especially in the area of cloud computing. Load balancing [32] essentially leverages VM checkpointing as it allows for the *migration* of a complete VM from one physical host to another over a network connection [36].

VM migration as an example for the application of VM checkpointing shows challenges and problems of the technology. The state of a VM is continuously changing as long as a VM is allocated to a host CPU. As a result, the VMM must suspend the CPU allocation of a VM so the VM is in a definite state and the VMM is able to capture it. Once the VM is stopped, a naive approach to creating a checkpoint is to copy all relevant data, including CPU state, main memory, disk image, and device states, to a persistent storage. When the copy process completes, the VMM resumes the VM.

This simple approach is called *stop-and-copy* and raises two interlinked problems. First, the plain amount of data that needs to be copied easily exceeds multiple gigabyte, because of the main memory and the disk image. Secondly, copying multiple gigabytes of data takes a certain amount of time in which the VM is not operational. This so-called *downtime* of the VM is noticeable to end users of

[3]Rittinghaus et al. deploy single CPU VMs so they restrict the benchmark to a single CPU core on the bare machine.

a service which the VM provides. Furthermore, if we do not want to store the checkpoint on disk, but rather migrate it to another host, the downtime even increases as the complete checkpoint data must be send over a network connection. The main focus of attention of VM checkpointing is therefore minimization of VM downtime and reduction of checkpoint data.

For the remainder of this section, we concentrate especially on checkpointing of guest main memory since the data amount of the VCPU and device states are negligible. Disk checkpointing is ignored in this work since it is of no relevance in our analysis and requires additional techniques due to the huge amounts of persistent data stored on disks.

2.2.1 Pre-Copy

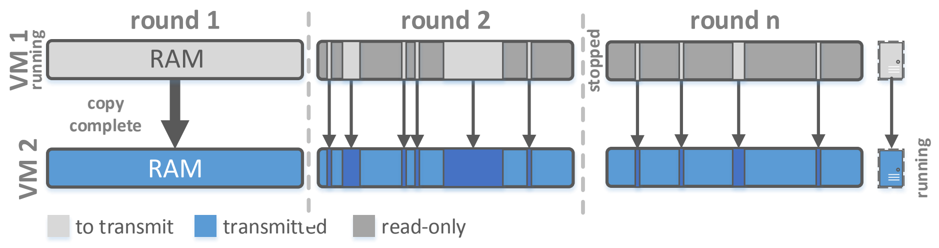
Nelson et al. [36] propose an *incremental* approach identified as *pre-copy*. Clark et al. [20] describe the process of pre-copy in more detail. With pre-copy the VMM transfers the main memory of a VM to another host over the course of multiple rounds.

We visualize this process in Figure 2.8(a). The target VM continues running during the migration process. In the first round the VMM declares the main memory of the target VM read-only and copies the complete RAM to the destination host. Given that the target VM is still running, it writes to pages that were already copied by the VMM. The VMM is notified of this modification in form of an exception since the complete RAM was marked read-only at the start of the first round. In the following rounds, the VMM copies only those pages of main memory that where modified during the previous round, thus the use of the term incremental checkpointing. After a certain number of rounds, a specific threshold is reached and the VMM stops the target VM and copies the remaining page frames to the destination host. Finally, the VMM transfers VCPU and device states and the target VM is resumed on the destination host.

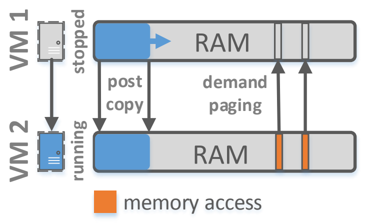
Pre-copy minimizes the downtime by copying a VM's main memory in multiple rounds. Under the assumption that the page modification rate of a VM is less than the copy rate of the checkpoint mechanism, each round must transfer less data to the destination host. As a result, in the final round, when the target VM is stopped, the downtime is kept to a minimum as there is just a small fraction of VM state left to transfer.

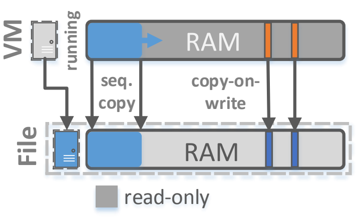
Clark et al. [20] evaluated their pre-copy migration for Xen with three workloads, each with a different requirement profile: (1) SPECweb99, (2) the server of an online multiplayer game, which requires low latencies, and (3) a memory intensive workload. Their test machines are connected by a switched Gigabit Ethernet. They measured downtimes of 210 ms, 60 ms and 3.5 s, respectively.

Nelson et al. [36] evaluated their implementation of pre-copy for the VMWare



(a) pre-copy





(b) post-copy (c) copy-on-write

Figure 2.8: (a) Shows the pre-copy migration mechanism. In round one the complete RAM of VM 1 is marked read-only and copied to VM 2. In subsequent rounds the page frames that have been modified during the previous round are transferred to the destination host. After n rounds, VM 1 is stopped and the VMM copies the remaining state, including a small set of page frames, VCPU and devices states, to VM 2. (b) Shows the post-copy migration mechanism. Post-copy transfers the VCPU and devices states of VM 1 prior to its main memory. Once the host of VM 2 receives VCPU and devices states, the main memory is copied sequentially from VM 1 to VM 2 while VM 2 is running concurrently. If VM 2 accesses a page frame that is not yet available, it is retrieved from VM 1 by issuing a demand-paging request. (c) Depicts copy-on-write checkpoint creation. In contrast to (a) and (b), the checkpoint data is not migrated to a different host, but written to a file. The VM is stopped in order to capture a consistent state of VCPU and devices and to write-protect the main memory. Afterward, the VM continues running, sequentially copying the main memory. If the VM writes to a not yet copied page frame, a page-fault occurs and the VMM saves the page to the checkpoint file before removing the write protection and granting write access to the VM.

ESX Server. Nelson et al measured similar downtimes like Clark et al. For a memory intensive workload, they measured downtimes between 400 ms and 800 ms, up to 6.8 s for different VM main memory sizes, namely 64 MiB, 128 MiB, 256 MiB, and 512 MiB.

Besides the downtime, we have mentioned the importance of the amount of data that must be transferred over a network connection. In general, pre-copy always transfers more data than the actual size of a VM's main memory due to the fact that solely round one transfers the complete guest RAM once. Each subsequent incremental round copies already transferred page frames that were altered by the guest in the mean time. Clark et al. evaluated the total amount of data transferred via the network connection for the migration during the SPECweb99 workload. The VM was configured with 800 MiB of main memory and their pre-copy implementation sent 860.7 MiB to the destination host. That is 7.5 % more than the original VM RAM size.

It is save to say that the low downtime for VM migration using pre-copy is achieved by accepting an increase in the arising amount of data that must be send over the network.

2.2.2 Post-Copy

The accumulated amount of data that must be copied to the destination host if using pre-copy exceeds the raw size of the target VM. Therefore, the total checkpointing time, hence the time it takes from the beginning of the first round till resuming the VM on the destination host, is high.

The *post-copy* approach moves the transmission of the VCPU and device states to the beginning of the migration. In contrast to pre-copy, which transfers the main memory of the VM before the VCPU and devices states, post-copy transfers the main memory of the VM after the transmission of the VCPU and devices states. As a result, the target VM is stopped at the source host and resumed on the destination host once the VCPU and devices states are received. The VM continues executing while the VMM concurrently starts copying the main memory from the source host to the already running VM on the destination host. If the target VM tries to access parts of its main memory that are not yet available, the VMM issues a demand paging request and retrieves the accessed page from the source host. Demand paging induces a performance penalty on the target VM since the request is served by a network connection with a high latency compared to the general paging process occurring on a local system where data is fetched from storage or a shared memory page is mapped in. The complete process is visualized in Figure 2.8(b).

Hines et al. [29] describe their post-copy implementation for Xen [14] that deploys additional optimizations, especially for demand paging. First of all, they deploy a simple pre-paging algorithm in order to diminish performance loss due to demand paging. The pre-paging algorithm is based upon the assumption that the target VM is more likely to sequentially access memory pages than in a random manner. Therefore, if a demand-paging fault occurs they shift the post-copying task to continue sequential copying with the unavailable page frame that caused the demand-paging fault. Furthermore, they extend Xen's ballooning mechanism to avoid transferring free or unallocated pages over the network.

Guest memory ballooning was first introduced by Waldsburger [45] for the VMWare ESX Server. Waldsburger introduced a small balloon module into the guest OS that is implemented as a device driver or kernel module. The balloon component does not interface with the guest OS, but rather with the hypervisor. If the hypervisor intends to reclaim main memory from a VM it requests the ballooning component to inflate. Inflating means the balloon allocates and pins physical pages. This increases memory pressure inside the guest, thus triggering the guest OSs memory management causing the guest OS to return physical pages from its free list. The physical pages allocated by the balloon are returned to the VMM and may be utilized for the creation of additional VMs. VM ballooning is an important mechanism that enables over-commitment of the actually available physical memory in order to maximize utilization of a host.

Hines et al. provide an evaluation that directly compares their post-copy implementation with the existing pre-copy implementation of Xen. They execute various benchmarks in a VM configured with 512 MiB of RAM.

Their measurements show that post-copy significantly reduces total migration time of a VM running the SPECWeb2005 benchmark. The total migration time drops from 11 s with pre-copy to 8 s with post-copy. Furthermore, post-copy reduces the total number of transferred pages for SPECWeb2005 from 180000 with pre-copy to 130000 with post-copy. Yet, post-copy suffers from high downtimes above 1 s compared to Xen's pre-copy implementation that achieves downtimes of at max 600 ms. Hines et al. argue that the high downtime of their post-copy implementation is caused by the page fault tracking mechanism that was deployed for demand paging.

2.2.3 Copy-on-Write

The approaches that we have discussed up till now have all presumed the use case of VM migration for their checkpointing mechanisms. *Copy-on-write* (CoW) is typically exclusively used for creation of persistent checkpoints. CoW aims to trade runtime overhead and a higher checkpoint creation time for a low downtime. The process is depicted in Figure 2.8(c).

For CoW the VMM stops the target VM, and saves VCPU and device states just as it is the case with post-copy VM migration. Afterward, the VMM write-protects the target VM's memory. As the VM continues execution, the VMM starts copying the VM's memory concurrently, similar to post-copy. The VMM removes the write-protection once a page frame is copied to persistent memory. If the target VM accesses write-protected main memory, the execution traps into the VMM. The VMM saves the affected page frame and clears the write-protection.

Sun et al. [28] provide a CoW checkpoint implementation for the Xen hypervisor. In contrast to Hines et al., who rely on a para-virtualized solution for their post-copy VM migration approach, Sun et al. proposed a completely transparent checkpoint mechanism. They implemented the write-protection required for CoW in the VMM by leveraging the SPTs of the target VM that are managed by the hypervisor.

Sun et al. evaluated their solution by comparing it with a modified version of Xen's pre-copy migration mechanism that simply dumps checkpoint data to a file instead of transferring it to another host. They state that CoW decreases the overall downtime by more than 76 %. Unfortunately, they did not provide any data regarding the total checkpoint creation time or the overhead induced by the traps into the hypervisor due to write accesses to protected page frames.

2.2.4 Conclusion

In summary, we notice that the minimization of one of the performance measures of checkpointing (downtime, total checkpoint/migration time, data amount) always is a trade-off with the others. Pre-copy trades a low downtime for a raise in total migration time and the number of page frames transferred over the network. Post-copy on the other hand achieves shorter total migration time and copies less page frames at the cost of an increased downtime. The CoW approach proposed by Sun et al. proofs to significantly reduce the downtime, though it is a checkpoint mechanism only and therefore cannot be compared with the pre-and post-copy migration mechanisms that we have discussed in this section.

2.3 Record and Replay

Besides checkpointing, VMs provide the benefit that their execution, including the behavior of the guest OS, can be observed and analyzed in detail. Therefore, the execution of a VM can be recorded and replayed at a later point in time in the exact same way.

This process is referred to as *deterministic record and replay* (RR). RR is of special interest for security analysis [44] and debugging since it eases the investigation of the behavior of a potential intruder or a misbehaving software and allows for debugging of so-called heisenbugs [27][4] or race conditions.

In general, a computer system is composed of deterministic components. The CPU, for example, always generates the same output given a specific input. As a result, the repeated execution of a computer program always works out in the same way, given we provide identical input each time. In practice this is not the case since non-deterministic events disrupt the deterministic execution on a regular basis. An obvious example for such non-deterministic events are incoming

[4]A heisenbug is a bug that does not occur once the developer tries to observe it.

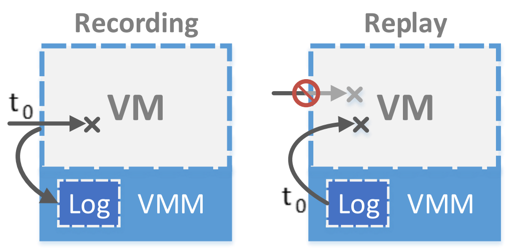


Figure 2.9: Illustrating the two phases of record and replay. The left side visualizes the recording phase. The VMM intercepts a non-deterministic event that occurs at t_0 and writes it to the recording log. During replay the VMM blocks non-deterministic events that do not originate from the log. Once the VM's execution reaches timestamp t_0, the VMM injects the corresponding event.

network packets that need processing. If a new network packet arrives, the network card issues an interrupt that signals the CPU to suspend its current task and call the interrupt handler of the OS in order to process the incoming data. Besides incoming network packets there are other, internal and external, events that cause a disruption breaking the CPU's deterministic execution, for example, user input or reading of time sources like the time stamp counter of the x86 architecture.

The VMM is able to record all non-deterministic events that alter the execution of a VM. For this purpose, it takes an exact location of the occurrence of an event in the execution flow, its type, and linked data (e.g., content of incoming network packets). We further refer to the exact location of an event in the execution flow as the landmark. The mentioned information allows for the exact reproduction of the course of execution of a VM. If a VMM starts a replay, it opens the replay log and starts the execution of the VM. Once the VM hits the landmark of a non-deterministic event, the VMM reads the event information from the log file and injects it into the VM. Furthermore, the VMM must intercept and block all non-deterministic events that do not originate from the replay log. As a result, each run the VM replicates the exact same sequence of internal states. The process of recording and replaying non-determinism is schematically shown in Figure 2.9.

Dunlap et al. [44] provide a fully functional RR implementation, called *ReVirt*, that bases upon UMLinux [30]. UMLinux is an early trap-and-emulate VMM for Linux that executes paravirtualized guests and runs on the x86 architecture. Dunlap et al. deploy ReVirt to analyze attacks. They argue that system loggers for system analysis that execute within an OS that is under attack are insufficient due to multiple reasons.

First of all, they lack integrity since an attacker may tamper with the system logger to cover up the attack. ReVirt is part of a VMM so the integrity of logged data is save as long as we trust the VMM, which has a much smaller trusted code base than a full-size monolithic kernel, since ReVirt is a type-1 hypervisor. Secondly, system loggers, which are part of an OS, often only record a few types of events of interest that may not be sufficient to recreate and analyze an attack in detail. ReVirt on the other hand records every non-deterministic event that affects the execution of a VM, so it is able to exactly recreate its course of execution including the malicious behavior of an attacker.

Dunlap et al. conducted several experiments to evaluate various aspects of ReVirt. They state that logging induces an additional overhead of at max 8 % whereas replaying results in a runtime overhead of at most 3 %. For workloads with long idle phases the replay time even decreases since the replay skips over idle phases. ReVirt proofs that VM record and replay is a feasible approach for system analysis. Nevertheless, Dunlap et al. depend on an paravirtualized Linux kernel for their analysis.

Besides system analysis, RR can be deployed to improve VM checkpointing. The checkpoint mechanisms we have discussed in Section 2.2 all base upon the idea to transfer the VM's main memory while the target VM executes concurrently. Liu et al. [34] propose a novel approach that utilizes RR in order to migrate a target VM from one host to another.

First of all, they stop the target VM's execution and capture its general state, namely CPU and devices. Next, they continue execution and create a CoW checkpoint of the VM's main memory. Additionally, they record all non-deterministic events that affect the VM's course of execution. Once the checkpoint is completed, they start copying the checkpoint file to the destination host while the VMM keeps recording non-determinism affecting the VM. Finally, they iteratively transfer the log file to the destination host, similar to the round-based approach found in pre-copy. The destination host receives the checkpoint file and restores the VM's state.

The VMM on the destination host continuously replays the log files as they arrive in order to synchronize the new VM on the destination host with the target VM on the source host. Once the size of the log file created on the source host falls below a certain threshold the target VM is stopped on the source host. The last log file is transferred and replayed on the destination host, and the target VM resumes execution on the destination host.

Liu et al. base their checkpoint mechanism upon two assumptions. First, the log file growth rate is always less than network transfer rate and secondly the replay rate is higher than the log growth rate. Their measurements show that both assumptions hold in their experiments since the log growth rate never exceeds 1 MiB/s and the replay rate, normalized to logging, is at least $1.05\times$. As a result it is guaranteed for the VM on the destination host to catch up with the target VM on the source host.

2.3.1 Heterogeneous Record and Replay

Dunlap et al. facilitate a single platform for their RR system thus recording as well as replaying is performed on the same hypervisor. This, so-called homogeneous RR, comes with drawbacks. For example, if we choose a hypervisor that is optimized for performance, it often lacks analysis capabilities such as the ability to monitor every main memory access. Therefore, it is impossible to conduct, e.g., extended memory traces using a RR system that bases upon a hypervisor that was optimized for performance.

A solution for said issue is the deployment of different hypervisors for the recording and the replaying phase, so-called heterogeneous RR. With heterogeneous RR, we utilize the performance of a hardware-assisted VMM for recording and the analysis capabilities, e.g., the facilities for memory tracing, of a full system simulator.

Chow et al. [19] introduce *Aftersight*, a heterogeneous RR system that provides offline and online analysis capabilities including intrusion detection and prevention. Aftersight uses the DBT of VMWare ESX Server for more time critical analysis, whereas they utilize QEMU [9] for offline replaying and analysis. In contrast to ReVirt, which is based on UMLinux, QEMU allows for extended system analysis without any modifications to the guest OS. This is mainly due to QEMU's DBT that can be extended and instrumented for analysis.

Chow et al. provide an extensive evaluation of Aftersight's online and offline analysis capabilities. They differentiate three applications for Aftersight – first, running the analysis synchronous with the analyzed workload. Secondly, running heavyweight analysis concurrently with the workload and lastly, performing heavyweight offline analysis. For heavyweight, concurrent analysis they utilize the DBT of VMWare in order to detect malicious memory accesses. For this purpose, they deployed an instrumented guest Linux kernel for their experiments. Their system discovered all write violations occurring in the system. Although, Aftersight's analysis VM executes with a delay of 0.86 s due to the overhead of DBT and analysis. Finally, for their offline analysis they deployed a modified version of QEMU. Their instrumentation of QEMU ensures that every instruction executed by the guest meets a set of memory safety guarantees. This process is very costly resulting in a runtime overhead of $100\times$ compared to the execution on a plain VMWare ESX server.

Chow et al. found safety critical bugs in the VMWare ESX server and the Linux kernel using the aforementioned heavyweight offline analysis, hence justifying the high runtime overhead. In summary, it is save to say that heterogeneous RR is a suitable tool for extended system analysis. Although, the high runtime overhead of about $100\times$ for in-depth analysis using a full system simulator like QEMU is not feasible for long running or interactive production workloads.

2.4 SimuBoost

The work of Chow et al. provides a good example of the high runtime overhead of an instrumented full system simulator. Rittinghaus et al. [40] compare the runtime of VMs executing various workloads in KVM, QEMU with memory trace hooks, QEMU with empty hooks, and Simics with empty hooks. First of all, they state that the slowdown induced by KVM for the evaluated benchmarks is approximately zero. They find a slowdown of factor 22–33$\times$ for QEMU without hooks and a slowdown of factor 113–207$\times$ for QEMU with memory trace hooks. The slowdown of Simics ranges from 624$\times$ to 1036$\times$ with empty trace hooks. This data confirms the results presented by Chow et al. regarding the runtime overhead of an instrumented QEMU.

This extremely high slowdown renders the application of full system simulation for the purpose of analyzing production workloads infeasible. For example, any interactive application cannot be analyzed using full system simulation since the system seems unresponsive to a human user [35]. This unresponsiveness weighs even more if we want to create a memory trace of an application that conducts communication over a network, e.g., a web server or a SQL database server, since its client connections time out. We might argue that the use of heterogeneous RR, as introduced in Section 2.3.1, solves the problem of unresponsiveness and allows for an extensive analysis afterwards. Yet, even with heterogeneous RR the recording of one hour of any workload takes at least four days to replay and analyze due to the high slowdown induced by full system simulation.

Rittinghaus et al. [40] introduce *SimuBoost*, a distributed system to speedup full system simulation that combines techniques presented in this work. SimuBoost as a whole is a classical producer-consumer system as illustrated in Figure 2.10. The system leverages the bare-metal execution speed of a hardware-assisted VM to *precompute* a workload running inside the VM, including the behavior of the guest OS. The VMM on the producer continuously creates checkpoints of the hardware-assisted VM at a fixed interval length of several hundreds of milliseconds up to multiple seconds. Therefore, each checkpoint marks the start of a distinct execution interval that ends at the start of the subsequent checkpoint.

Let us assume a consumer receives checkpoint number j and restores it. At this point the state of the consumer's simulation exactly matches the original state of the producer VM at the same point in execution. The consumer resumes execution starting from the state restored from the checkpoint. In the course of execution, the simulation drifts apart from the original execution on the producer node, due to the effects of non-deterministic events as discussed in Section 2.3. As a result, the state of the VM after the consumer completes the simulation of checkpoint j does not match the VM state contained in checkpoint $j + 1$. This breaks our

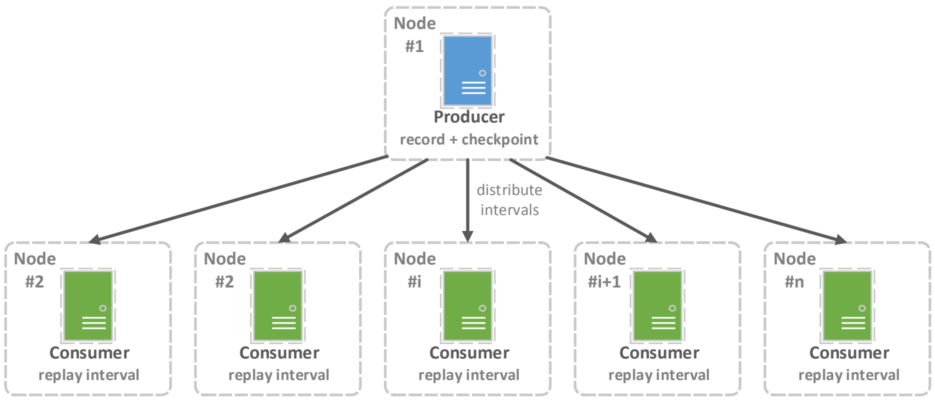


Figure 2.10: Illustrating the general setup of SimuBoost. There is a computation node that serves as single producer. It runs a VMM that executes a hardware-assisted VM. The VM itself executes a workload of interest that runs for a fixed period of time. The total runtime of the hardware-assisted VM can be split into a finite number of self-contained intervals. SimuBoost distributes each interval to a dedicated consumer node. The consumers simulate their respective intervals concurrently.

assumption that the producer VM *precomputes* the execution for the simulation nodes.

For this reason, SimuBoost facilitates heterogeneous RR to counteract the divergence of simulations on consumer nodes and the original hardware-assisted course of execution on the producer. The VMM on the producer node logs non-deterministic events in addition to creating continuous checkpoints. Therefore, a simulation interval now consists of a checkpoint and a log of non-deterministic events. If a consumer node receives a new interval j it restores the checkpoint and starts to replay the log starting at said checkpoint. Therefore, the state after the simulation of interval j matches exactly the state contained in checkpoint $j + 1$, hence guaranteeing consistency between the course of execution of the VM on the producer and the simulations on the consumer nodes.

2.4.1 Continuous Checkpointing

In contrast to traditional applications of checkpointing, such as VM migration or simple backup creation, SimuBoost creates *continuous checkpoints* with a high frequency. Continuous checkpointing must meet specific criteria in order to achieve a maximal speedup of full system simulation using SimuBoost. Baudis [15] analyzed those requirements and implemented a prototype checkpoint mechanism for the use in SimuBoost. SimuBoost shares common requirements with traditional checkpoint mechanisms, which are utilized for VM migration in cloud environments, as we have discussed in Section 2.2.

In general, a checkpoint mechanism should induce a minimal downtime so that the process of migration or rather checkpointing is transparent to the VM. For the application in SimuBoost, the downtime induced by the checkpointing mechanism weighs even more since SimuBoost takes checkpoints at a very high frequency of about one checkpoint per second. For example, if we presume a checkpoint interval length of one second and hypothesize a runtime of one hour for the workload, SimuBoost would create $n = 1\,h \cdot 60 \cdot 60 = 3600$ checkpoints. Furthermore, if we take the minimal downtime measured by Clark et al. [20] for their implementation of pre-copy on the Xen hypervisor as a basis for the downtime (60 ms), we get an accumulated downtime of $t = 3600 \cdot 60\,ms = 216\,s$ for the producer VM of SimuBoost. This already implies a runtime overhead of 6 % under the assumption of a constant downtime. Furthermore, taking checkpoints with a high frequency implies high data rates. The arising data has to be stored and distributed to consumer nodes. As a result, another important requirement on the checkpoint mechanism is the reduction of arising checkpoint data.

Baudis [15] proposes an incremental checkpoint mechanism to meet both requirements. Pre-copy checkpoint approaches already incrementally process data, thus they only copy data that has changed since the previous round. For the application of incremental checkpointing in the context of SimuBoost, Baudis treats each checkpoint interval as an enclosed round so SimuBoost only saves those page frames and disk sectors that where modified by the guest during the last checkpoint interval. Additionally, he implements a deduplication mechanism for checkpoint data in order to further reduce the accumulating data amount beyond the boundaries of incremental checkpointing.

Baudis performed an evaluation of his incremental checkpoint mechanism. He used a checkpoint interval of 2 s, and ran a kernel build as a workload in the checkpointed VM and took 50 continuous checkpoints. His incremental checkpoint mechanism, including deduplication and persisting data using MongoDB [7], achieves downtimes of on average 1–2 s. One reason for the high downtime of his implementation is the simple stop-and-copy approach he takes to copy modified page frame and disk sectors to storage. Baudis further states that 40–60 % of the checkpoint creation time is spent by MongoDB processing checkpoint data. Therefore, Baudis approach was not applicable for SimuBoost, especially if we consider our example that shows that even a hypothetical downtime of 60 ms as achieved by the pre-copy approach of Clark et al. already would induce a runtime overhead of 6 %.

Baudis work showed that a storage solution optimized for the use in SimuBoost is necessary in order to reduce the downtime. Eicher [23] concentrated on reducing the downtime of the checkpoint mechanism by implementing a custom checkpoint store and processing checkpoint data asynchronously. He also presented a first approach to checkpoint distribution which was later adjusted by

Pusch [38], leading to the current checkpoint distribution mechanism based on multicast. Eicher defined an upper bound of 100 ms on the checkpoint downtime, since the work of Rittinghaus et al. [40] presumed the same value and it is perceived as instantaneous by humans [35]. Eicher found an average downtime of 44 ms for an interval length of 2 s, therefore meeting his requirements of a downtime less than 100 ms. Yet, the downtime was not constant, but rather varying depending on the workload and its phase of execution.

As a result, Böhr [18] introduced an incremental copy-on-write checkpoint mechanism for the deployment in SimuBoost. Böhr uses the asynchronous checkpoint processing and storage system implemented by Eicher and replaces the actual process of how checkpoint data is copied to the storage unit. He found that CoW further reduces the downtime to on average 26 ms for a checkpoint interval of 2 s. Additionally, the deployment of CoW results in a relatively constant downtime since, while the VM is stopped, the main memory of the VM is simply being write-protected, whereas the actual process of copying checkpoint data to storage is performed concurrently to the VM's execution.

To this day the checkpoint mechanism of SimuBoost has been further optimized. We have implemented incremental CoW based checkpointing for the virtual disk in addition to the CoW checkpoint mechanism for creation of main memory checkpoints. Furthermore, the process of main memory checkpointing has been divided into two self-contained components that can be combined freely. On the one hand, there is the already discussed mode of the copy process itself that is either copy-on-write or stop-and-copy. On the other hand, there is the dirty logging mechanism (DLM).

We have already discussed incremental checkpointing, which implies the need to track page frames that where modified, hence dirtied, since the previous checkpoint. SimuBoost provides two different DLMs. Dirty logging via write protection (WP) write-protects the guest's main memory in the EPTs just as it is the case with CoW checkpointing. This means, if the guest tries to modify a page frame the access traps into the hypervisor and the hypervisor is able to update its *dirty bitmap* accordingly. Baudis, Eicher, and Böhr used WP as DLM in their respective work. The second DLM performs a page table walk during the downtime of the VM and *scans* the dirty bit of the EPT page table entries that are set by the CPU if a page frame has been written to. The information is in turn synchronized to the dirty bitmap for further processing by the actual data copying mechanism as mentioned before. The dirty bit in the page table entries are cleared before the VM resumes.

In summary, the checkpoint mechanism we currently deploy in SimuBoost has evolved over multiple years. It achieves constant, low downtimes of about 5 ms with WP and CoW and induces a low runtime overhead. Nevertheless, the recent work of Pusch [38] on checkpoint distribution showed that there is further optimization potential for the checkpoint mechanism. In the following Chapters, we will discuss the current state of the art of SimuBoost and perform a detailed analysis of the introduced checkpoint mechanisms in order to uncover remaining drawbacks. Furthermore, we are going to propose optimizations based on our analysis in order to improve the performance of the checkpoint mechanism for the specific application in SimuBoost.

Chapter 3

Analysis

SimuBoost [40] utilizes VM checkpoints and heterogeneous RR to achieve a speedup of full system emulation. Heterogeneous RR deploys a hardware-assisted hypervisor in the recording phase in order to leverage its almost bare-metal execution speed and utilizes full system emulation for the replaying phase due to its extended system analysis capabilities.

The approach of SimuBoost goes a step further as it logically divides the recording in intervals of fixed execution time by periodically taking VM checkpoints. The taken checkpoints are instantly distributed to simulation nodes allowing parallel replay of execution intervals. The speedup that is achievable by distributed heterogeneous RR depends on the performance of the deployed checkpoint mechanism. The checkpoint mechanism should have a minimal impact on the total execution time of the recording VM as well as the replaying VMs on the simulation nodes.

This work aims to optimize the checkpoint mechanism for the application in SimuBoost. Therefore, we further analyze the available checkpoint mechanisms, including checkpoint creation and checkpoint loading. First of all, we specify a set of requirements for a checkpoint mechanism that is deployed in the context of distributed, heterogeneous RR in order to systematically identify room for improvement.

3.1 Requirements on the Checkpoint Mechanism

SimuBoost is a multi-component application that, as a whole, runs on either a computer cluster or a single high-performance workstation. Each particular component, namely, (a) the hardware-assisted VM including logging of non-deterministic events and incremental checkpointing, (b) the checkpoint and recording log distribution, and (c) the emulation on the simulation nodes, must meet definite requirements in order to perform optimally.

We recap the functionality of SimuBoost in detail in order to identify the importance of the checkpoint mechanism for the overall performance of SimuBoost. Furthermore, we determine requirements on the checkpoint mechanism along the way.

3.1.1 Theoretical Model

Rittinghaus et al. [40] provide an abstract model to argue about the theoretical performance of SimuBoost and its functionality. They differentiate two types of computation nodes. One hardware-assisted recording node and a number of simulation nodes.

The recording node executes a VM for a fixed time period T_{vm}. During the execution of the VM, the hypervisor logs non-deterministic events and takes continuous, incremental checkpoints at a fixed rate $r_{cp} = \frac{1}{L}$, where L is the interval length, thus splitting the execution time in n distinct intervals of length L. We assume the logging of non-determinism slows down the execution of the hardware-assisted VM by a factor of s_{rec}. Furthermore, we presume the checkpointing mechanism slows down the hardware-assisted VM by a factor s_{cp}.

The slowdown factor of the checkpoint mechanism s_{cp} depends on the so-called dirty logging mechanism, which is used to keep track of dirty page frames, and the CoW mechanism which causes page faults for write-protected page frames. Therefore, the overall performance penalty for the runtime T_{vm} of the VM is $s_{log} = s_{rec} \cdot s_{cp}$, resulting in the runtime of the recording VM $s_{log}T_{vm} = n \cdot L$.

Additionally, taking a checkpoint provokes a constant downtime t_c of the VM. Note, that the accumulated downtime of each checkpoint adds up to the wall-clock runtime of the recording VM, but not to the effective runtime T_{vm} of the VM. The effective runtime T_{vm} only describes the time the VM is actively advancing the execution of the guest system, thus each constant time period t_c, which the VM is stopped in order to capture a consistent checkpoint, is not added to the effective runtime.

The taken checkpoints and the replay log are instantly distributed to the simulation nodes. Once a simulation node receives a new, not yet replayed, simulation interval and it has replay capacity left, it loads the corresponding checkpoint taking a constant time t_i and starts the emulation of said interval. Figure 3.1 illustrates the distribution of intervals to emulations. We presume the runtime of the simulation compared to the runtime of hardware-assisted VM T_{vm} is slowed down by a factor $s_{sim} = s_{rep} \cdot s_{emu}$, due to replaying non-deterministic events and the emulation itself. Therefore, the runtime of a conventional (non-parallelized) simulation is depicted by $T_{sim} = s_{sim} \cdot T_{vm} = s_{rep} \cdot s_{emu} \cdot T_{vm}$. Accordingly, under the assumption that each interval replays equally long, the runtime of the

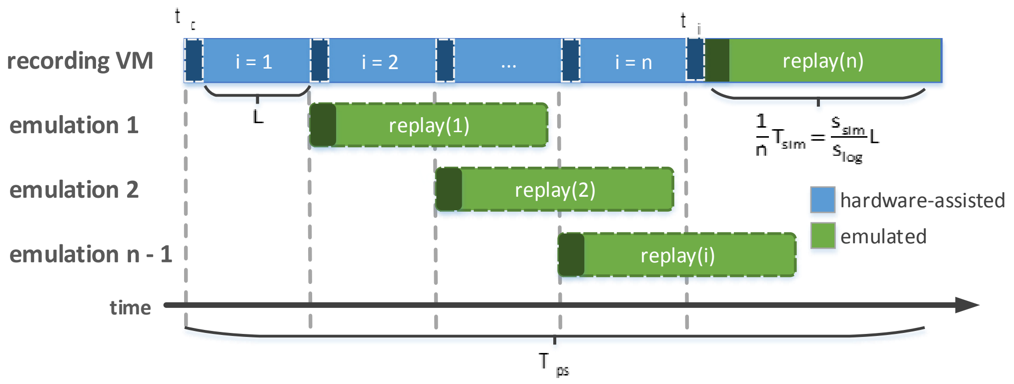


Figure 3.1: Illustrating the chronological sequence of the execution in SimuBoost. Emulation intervals 1 *to* n *map to a set of simulation nodes. The runtime of* $n-1$ *emulations is masked by the execution time of the recording and the replay of the* n*-th interval on the recording node itself.*

simulation for a single interval is $\frac{1}{n}T_{sim}$. If we further assume, that the last interval is replayed on the recording node itself, we get an overall parallel execution time of

$$T_{ps}(n) = s_{log}T_{vm} + n \cdot t_c + t_i + \frac{1}{n}T_{sim}. \tag{3.1}$$

If we substitute $n = \frac{s_{log}T_{vm}}{L}$, we obtain the parallel execution time T_{ps} as a function of the interval length L

$$T_{ps}(L) = s_{log}T_{vm} + \frac{s_{log}T_{vm}}{L} \cdot t_c + t_i + \frac{s_{sim}}{s_{log}}L \tag{3.2}$$

As a result the speedup of the parallel simulation compared to the common sequential simulation is given by

$$S(L) = \frac{T_{sim}}{T_{ps}(L)} = \frac{s_{sim}T_{vm}}{s_{log}T_{vm} + \frac{s_{log}T_{vm}}{L} \cdot t_c + t_i + \frac{s_{sim}}{s_{log}}L} \tag{3.3}$$

Rittinghaus uses Formula 3.3 to find an interval length L_{opt} for which the speedup $S(L)$ is maximal. The interval length in turn affects the number of simultaneously executable simulation jobs. Rittinghaus shows that the number of parallel simulation jobs is determined by the proportion of the simulation completion time $t_i + \frac{s_{sim}}{s_{log}}L$ to the interval arrival time $t_c + L$, under the assumption of an invariant emulation time $s_{sim}T_{vm}$, leading to an required number of nodes

$$N(L) = \left\lceil \frac{t_i + \frac{s_{sim}}{s_{log}}L}{t_c + L} + 1 \right\rceil \tag{3.4}$$

with $L = L_{opt}$.

The maximal number of jobs that can be executed in parallel is determined by the number of available simulation nodes and their hardware specifications. We assume a one-to-one mapping of emulations to available CPU cores on a node.

In our experience, the number of CPU cores or rather the extend of parallelism of the CPU is less of a problem regarding the performance of the simulations than the limited amount of main memory of the host. Therefore, the number of parallel simulation is primarily bound by the RAM size of the producer VM and the available main memory of the consumer nodes, since each emulation instantiates a complete VM, including its complete RAM image. If the recording VM occupies, e.g., 4 GiB of main memory, a consumer node with 16 GiB of main memory is in theory able to emulate three VMs, leaving just another 4 GiB for OS structures, potential analysis software and the checkpoint and recording log storage.

Rittinghaus et al. provide an example scenario with $T_{vm} = 3600$ s, $s_{sim} = 100$, $s_{log} = 1.08$, $t_c = 0.1$ s, and $t_i = 1$ s. They obtain an optimal number of simulation nodes of $N = 90$ for this scenario. If we presume consumer nodes with 16 GiB of main memory each, this implies that we need at least 30 consumer nodes in order to concurrently simulate a VM with 4 GiB of main memory.

Extended Theoretical Model

In practice, we do not have unlimited hardware resources since hardware in general is expensive. Therefore, we need a theoretical model that accounts for a limited number N of consumer nodes. Eicher [23] developed such a model in his work on improving the checkpoint storage of SimuBoost. He showed in his experiments that his model accurately predicts the speedup of SimuBoost compared to sequential emulation in his simple approximation of the original scenario. Yet, Eicher's model must be adapted for todays version of SimuBoost since the checkpoint mechanism has been enhanced since then. In the following, we revise one of Eicher's assumptions and recap his considerations regarding the extended runtime model.

Eicher argues that the accounting of a constant downtime in the unrestricted model of SimuBoost is insufficient since the stop-and-copy based checkpoint mechanism that was deployed in SimuBoost at that time did not guarantee a constant downtime for a fixed checkpoint interval length [15]. Therefore, Eicher states that it is reasonable to replace constant downtime by a slowdown factor s_e[1] that accounts for the difference of accumulated data during a checkpoint interval with length L. As a result, the term $t_c + L$ would be substituted by $s_e L$.

[1]The slowdown factor s_e relates to the slowdown factor s_{cp} that Eicher uses in his work. We substituted it in this work in order to avoid confusion with the completely unrelated slowdown factor s_{cp} which we have introduced in Section 3.1.1.

This substitution of the constant downtime addend is not needed for the current implementation of the checkpoint mechanism and was not needed back then for two reasons.

First of all, the downtimes of a set of checkpoints, which were created using SnC, vary for a fixed interval length. This depends on the arising data amount during each checkpoint interval. A more accurate model may introduce a downtime that depends on the data amount of each individual checkpoint interval in order to yield a more accurate model. Instead, Eicher substitutes the constant downtime addend for a fixed interval by a constant downtime slowdown factor for a fixed interval. This means it would be possible to simply choose a different constant downtime addend in the original model in order to achieve the same that Eicher intends to do. As a result, both accounting methods are equivalent and can be transformed correspondingly $s_e = \frac{t_c+L}{L}$.

Second and most importantly, the current checkpoint mechanism of SimuBoost deploys CoW to copy data to storage. Measurements by Böhr [18] showed that the assumption of a constant downtime holds for CoW checkpoints. Therefore, besides the general misinterpretation by Eicher, today we are save to assume a constant checkpoint downtime. As a result, for further considerations of Eicher's extended runtime model, we disregard his replacement of the downtime addend.

Eicher acts on the assumption expressed in Formula 3.4, namely, that the number of required simulation nodes depends on the proportion of the interval completion rate $t_i + \frac{s_{sim}}{s_{log}} L$ and the interval arrival rate $t_c + L$. He provides an inequation expressing the initial situation in which his extended runtime model is applicable.

The inequality states that the number of available simulation nodes N is less than needed for an optimal speedup as calculated by Rittinghaus et al.[2]:

$$N < N(L_{opt}) = \left\lceil \frac{t_i + \frac{s_{sim}}{s_{log}} L_{opt}}{t_c + L_{opt}} \right\rceil \tag{3.5}$$

In the original theoretical model the total runtime of SimuBoost is given by the sum of the runtime of the recording VM on the producer node and the runtime of the replay of the last simulation interval as illustrated in Figure 3.1. The runtime of concurrently executing simulations on the consumer nodes is masked by the availability of sufficient simulation nodes.

Eicher expresses the total runtime of SimuBoost with a fixed number of simulation nodes in a similar way. Given a fixed number of simulation nodes N and a checkpoint interval arrival rate of $t_c + L$ the Nth checkpoint becomes available after

$$T_{last}(N, L) = N \cdot (t_c + L) \tag{3.6}$$

[2]We drop the addend of one in the original model since it is simply accounting for the producer node. Eicher only considers the simulation nodes.

time units. Replaying a single checkpoint takes $t_i + \frac{s_{sim}}{s_{log}} L$ time units. The total runtime of the recording VM is split into n intervals as discussed in Section 3.1.1. The total replay time $n(t_i + \frac{s_{sim}}{s_{log}} L)$ may be evenly distributed to N simulation nodes. This results in a busy time of

$$T_{busy}(N, L) = \frac{n(t_i + \frac{s_{sim}}{s_{log}} L)}{N} \tag{3.7}$$

for each simulation node. Therefore, the total parallel replay time T_{ps} of SimuBoost given a fixed number of simulation nodes can be expressed by the sum of the time period after which the last simulation node is assigned T_{last} and the time the last simulation node executes T_{busy}:

$$T_{ps}(N, L) = T_{last}(N, L) + T_{busy}(N, L) \tag{3.8}$$

As a result, similar to Formula 3.3, the speedup given a fixed number of replay nodes N and a fixed replay interval L is:

$$S(N, L) = \frac{s_{sim} T_{vm}}{T_{ps}(N, L)} = \frac{N \cdot L \cdot s_{sim} T_{vm}}{N^2 L(t_c + L) + T_{vm}(s_{log} t_i + s_{sim} L)} \tag{3.9}$$

Eicher derives the optimal interval length given a fixed number of replay nodes by solving the equation $\frac{\delta}{\delta L} S(N, L) = 0$ which results in the following function of N:

$$L_{opt}(N) = \frac{\sqrt{t_i s_{log} T_{vm}}}{N} \tag{3.10}$$

In comparison to Eicher's function to calculate the optimal interval length L_{opt}, our function does not depend on the value of the downtime which makes sense since the downtime of the CoW approach used by SimuBoost's checkpoint mechanism is constant.

3.1.2 Conclusion

Eicher's extended model accounts for a limited number of simulation nodes. For the purpose of SimuBoost, we presume that a single consumer node hosts multiple simulations. The exact number depends on the hardware specifications of each individual consumer. As a result, the number of consumer nodes is less or equal to the number of simulation nodes.

At the end of Section 3.1.1, we state that the number of parallel simulations on a single consumer node is primarily bound by the available main memory of the machine since each simulation consumes the same amount of memory than the single recording VM running on the producer node. This means, if we were able to halve the amount of memory required by a single simulation, we could double the number of simulations per consumer, as long as there are enough CPU cores unassigned. Satisfying the requirements on the number of simulation nodes given by the original model in Section 3.1.1 would even allow us to achieve the maximal speedup possible with SimuBoost.

The original model as well as the extended model, include the downtime t_c and the slowdown s_{log} of the VM, which is caused by the recording of non-deterministic events and the dirty logging of the checkpoint mechanism, as a constant factor.

Finally, in Formula 3.4 we notice the effect of the checkpoint loading time on the required number of simulation nodes. The lower the checkpoint loading time the less simulation nodes are required. This is especially highlighted by Eicher's model that embeds the checkpoint loading time directly in the function for the achievable speedup. This is due to the fact that Eicher derived his model from the inequality shown in Formula 3.5.

In summary Formulas 3.3, 3.9 and 3.4 lead to the following abstract requirements on the checkpointing mechanisms.

(a) Downtime and logging slowdown The downtime and the logging slowdown $s_{log} = s_{rec} + s_{cp}$ directly influence the speed up $S(L)$ of the parallel simulation using SimuBoost. Therefore, the downtime and the runtime overhead caused by the DLM should be reduced.

(b) Checkpoint loading time The checkpoint loading time needs to be reduced since it directly influences the number of required simulation nodes. Furthermore, the checkpoint loading time directly influences the achievable speedup in case of a limited number of available simulation nodes as shown by the extended model.

(c) Memory footprint of simulation VMs The number of parallel simulations on a consumer node is bound by the available main memory on the respective node. Therefore, minimizing the memory footprint of an emulation would allow for more simulations per consumer node and improve the performance of SimuBoost for a fixed number of simulation nodes.

3.2 Different Checkpoint Mechanisms in Comparison

In Sections 3.1.1 and 3.1.2, we have derived a set of requirements on the checkpoint mechanism which is deployed in SimuBoost. In this section, we compare the available checkpoint mechanisms conceptually. We use the results of our tests to further argue about the applicability of each checkpoint mechanism in the context of SimuBoost.

3.2.1 Checkpoint Creation

First of all, we are going to recapitulate the current checkpoint mechanism of SimuBoost regarding main memory, disk, and devices. The disk checkpointing mechanism implements an incremental CoW approach. Device states only represent a small fraction of the complete data amount of a checkpoint so they are simply copied to storage during the downtime. Main memory checkpointing leaves more room for discussion. We have already considered the fact that the main memory checkpoint mechanism itself splits into two distinct components, the dirty logging mechanism (DLM), and the actual copy mechanism (see Sections 2.4.1 and 3.1.1 for reference).

The dirty logging mechanism keeps track of dirty page frames by maintaining a dirty bitmap that contains information about which guest page frames were written to during the last checkpoint interval. SimuBoost provides two DLMs – scanning and write protection (WP).

Scanning moves the complete work of acquiring the dirty set of a checkpoint to the downtime of the VM. For that purpose, the VMM performs a page table walk in software while the VM is stopped and tests whether the dirty bit of the page table entries are set. If a dirty bit is set, it looks up the GPA mapped by the corresponding page table entry and synchronizes the information to the dirty bitmap. In contrast, WP performs less work during the downtime of the VM, effectively moving the construction of the dirty bitmap to the execution time of the VM. Therefore, the VMM simply write protects the VM's main memory while the VM is stopped. After the VM resumes, each write access to a page frame triggers a page-fault so the VMM is able to reconstruct the GPA and set the corresponding bit in the dirty bitmap.

The copy mechanism on the other hand utilizes the information of the dirty bitmap in order to processes dirty page frames and copy them to storage. SimuBoost implements two copy mechanisms – stop-and-copy (SnC) and copy-on-write (CoW).

SnC results in high downtimes [15] since it copies the checkpoint data while the VM is suspended. CoW copies dirty page frames to storage concurrently to the execution of the VM, thus reducing the downtime. As a result, CoW is the first choice for the copy mechanism used in SimuBoost.

If we presume the usage of CoW as common copy mechanism and consider the theoretical performance implications of the aforementioned DLMs, we can form following hypothesis.

Scanning should induce a higher downtime than WP since scanning performs more work, namely, walking the EPTs and synchronizing the dirty bitmap during the VM downtime. In return, using WP as DLM should result in an increased runtime overhead compared to scanning, given that write accesses to guest memory trigger VM exits and require handling by the VMM in order to synchronize dirty page frames to the dirty bitmap.

We test our hypotheses by creating continuous checkpoints of a VM equipped with 2 GiB of main memory, using CoW as copy mechanism and scanning or WP as DLM. All tests execute on an *Intel(R) Xeon(R) CPU E5-2630 v3* with *64 GiB of main memory* and a *1 TB Samsung SSD 850*. We run two different workloads in the test VM.

First of all, we execute a Linux kernel build using the Phoronix Test Suite [8] in order to determine the runtime overhead of the respective DLMs. Furthermore, we deploy SPECjbb [11] as a memory and computation intensive benchmark to evaluate the downtimes induced by the DLMs under medium to heavy load. We repeat these measurements for varying interval lengths of 1 s, 2 s, 4 s, and 8 s. The shown values are the mean of ten independent runs. Before each run we flushed the Linux page cache in order to minimize cache related side effects.

Figure 3.2 shows the total runtime (RT) of a VM for different DLMs and varying checkpoint intervals. We consider the RT minus the accumulated downtimes in order to argue about the raw runtime overhead induced by the DLM during VM execution. Therefore, the opaque axis on the right-hand side additionally shows the performance overhead of the respective DLM in percent of the *baseline*. The baseline is given by the RT of an identically configured VM without checkpointing. For a kernel build the baseline is *745.31 s*.

We clearly see that scanning induces less performance overhead than WP. Yet, we notice a convergence of the runtime overhead of both DLMs with increasing checkpoint interval length. Figure 3.3 illustrates the corresponding average downtimes and accumulated downtimes.

The accumulated downtimes plotted on the right-hand side axis are aforementioned accumulated downtimes, which we have subtracted from the total runtime in Figure 3.2, in order to obtain the raw runtime performance overhead of the DLMs. We notice that scanning leads to average downtimes that are nearly twice the number of the average downtimes of WP. Considering this rise of the average downtime the higher the checkpoint interval gets, we may expect the accumu-

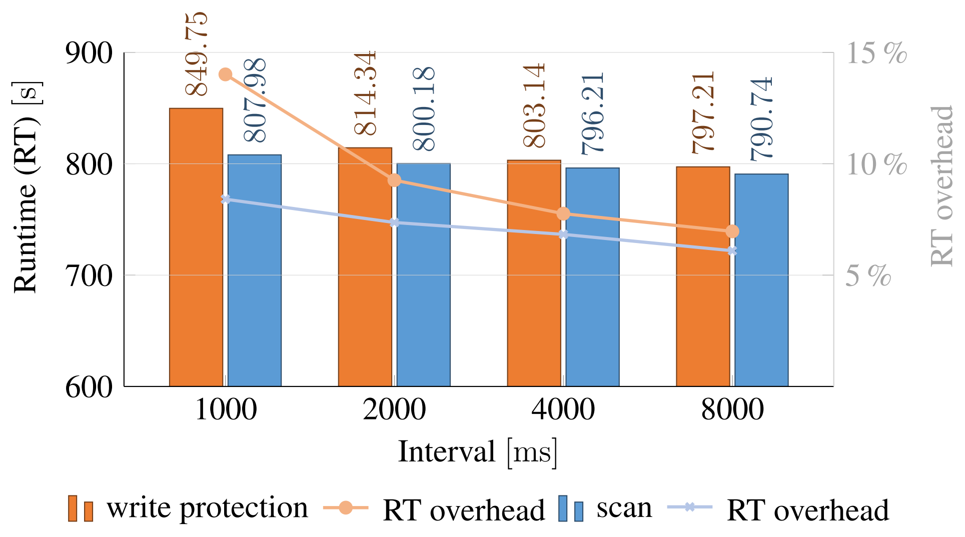


Figure 3.2: The left axis plots the runtime (RT) of a VM executing a Linux kernel build. The runtime is plotted minus the accumulated downtime caused by checkpointing. This means the figure compares the raw runtime overheads produced by the different dirty logging mechanisms. The right axis shows the runtime overhead of the checkpointed VM compared to the non-checkpointed VM in percent. The runtime of a VM without continuous checkpointing totals to 745.31 s.

lated downtime to increase as well. Yet, the accumulated downtime of scanning decreases converging the accumulated downtime of WP.

We explain this effect by the limited growth of the main memory working-set in relation to the interval length. Werner [47] provides an extended analysis of VM main memory working-sets in the context of SimuBoost. He states that the working set in-between two continuous checkpoints grows slower the longer the checkpoint interval is.

For example, Werner provides measurements of the write working set of a Linux kernel build executing in a VM configured with 2 GiB of main memory for interval lengths of 2000 ms, 4000 ms, and 8000 ms. His measurements show an average write working set of 25859 page frames for an interval length of 2000 ms, 29694 page frames for an interval length of 4000 ms, and a write working set of 33478 page frames for an interval length of 8000 ms. This means doubling the interval length does add about 4000 distinct page frames to the write working set. As a result, if we double the interval length, scanning only has to synchronize additional 4000 page frames to the dirty bitmap during the downtime of the VM.

We have identified this additional work in the increasing average downtime in Figure 3.3. SimuBoost takes two checkpoints during a time period of 8 s using a checkpoint interval length of 4000 ms. Therefore, $2 \times 29694 = 59388$ page frames

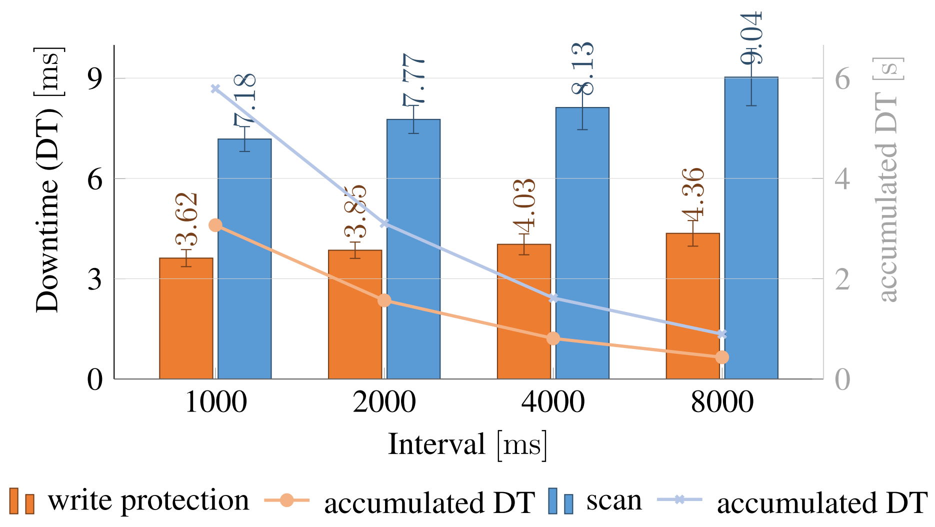


Figure 3.3: The left axis plots the average downtime (DT) of a VM executing a Linux kernel build while creating continuous checkpoints. The right axis shows the accumulated downtime of the checkpointed VM.

are synchronized by the DLM. If we use a checkpoint interval length of 8000 ms SimuBoost only takes one checkpoint during a time period of 8 s so the DLM only synchronizes 33478 page frames.

Besides the convergence of the accumulated downtime, the limited growth of the working set can also explain the convergence of the runtime overhead of WP illustrated in Figure 3.2, since each additional page frame that has been written to during a time period triggers only a single write-protection fault that needs to be handled by the VMM.

In summary, these observations confirm our hypotheses that scanning leads to a higher average downtime than WP since scanning synchronizes dirty page frames to the dirty bitmap during the downtime of the VM. At the same time, the runtime overhead of scanning is less than 9 % for a checkpoint interval of 1000 ms and even drops to 6.1 % for an interval length of 8000 ms.

In contrast, WP must only write-protect the guest main memory during the VM downtime, effectively moving the overhead of synchronizing dirty page frames to the runtime of the VM. As a result, WP achieves a relatively low, constant average downtime. Yet, it results in a higher runtime overhead between 7.76% and 14 % depending on the length of the checkpoint interval.

We further run SPECjbb in order to test the effects of the DLMs regarding the induced downtime over the course of execution. In comparison to a Linux kernel build, which is a workload causing low to medium CPU and memory load and scattered disk I/O activity, SPECjbb passes through multiple phases of execution

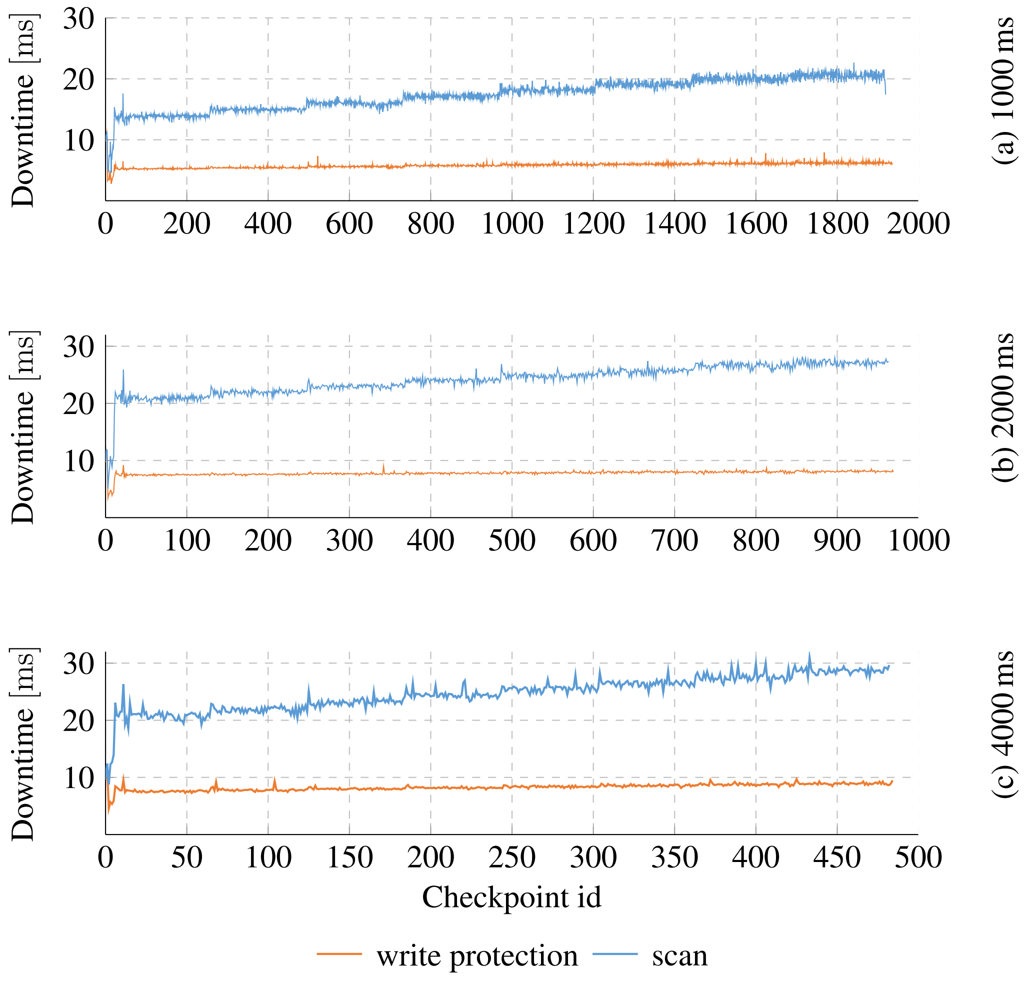


Figure 3.4: Plots (a), (b), and (c) show the downtime for each individual checkpoint for interval lengths of 1000 ms, 2000 ms, and 4000 ms respectively. The workload executing on the test VM was SPECjbb. We deployed write protection and scanning as dirty logging mechanisms.

each of which increases the memory load, hence the dirty page frames per time period.

Figure 3.4 plots the downtime over the course of execution of SPECjbb for checkpoint interval lengths of 1000 ms, 2000 ms, and 4000 ms. We notice oscillating, increasing downtimes for scanning, whereas WP causes relatively constant downtimes of below 10 ms for all interval lengths.

For an interval length of 1000 ms, we are able to identify the phases of increasing memory load of SPECjbb since the downtime for scanning increases every

200 s. Besides a continuously raising downtime for scanning, we realize that the downtime increasingly fluctuates the longer the checkpoint interval is. This observation is confirmed if we compare the standard error of the average downtime of the SPECjbb workload for an interval length of 1000 ms and 4000 ms (Appendix Table A.3). For an interval of 1000 ms the average downtime is 17.36 ms with a standard error of 0.89. In contrast, for an interval of 4000 ms the average downtime raises to 24.68 ms with a standard error of 1.1.

Conclusion

In summary, the downtimes that we have measured for SPECjbb confirm our conclusions from the observations of the Linux kernel build workload. Scanning as a DLM trades less runtime overhead for an increased downtime, whereas WP achieves relatively constant downtimes since synchronization of dirty page frames to the dirty bitmap is performed during VM execution. Furthermore, we do realize that scanning is affected by the memory load induced by the deployed workload compared to WP which achieves constant low downtimes. Additionally, increased interval lengths leads to a convergence of runtime overheads and accumulated downtimes in case of both DLMs.

Our test results illustrate the impact of the DLMs on the downtime and the runtime overhead of the checkpoint mechanism. This allows for further discussion about the preferable DLM for the application in SimuBoost. We find out that WP has an obvious advantage regarding the induced downtime. In general, independent of the interval length, WP achieves a constant downtime that is about half the downtime of scanning. Although, for short interval lengths up to 4000 ms scanning has a slight runtime performance advantage. For longer interval lengths, the runtime overhead of both WP and scanning converge rendering the minimal advantage of scanning negligible.

Nevertheless, Figure 3.3 illustrates that the accumulated downtimes of WP and scanning also converge for increasing interval lengths. Therefore, we argue that the choice of the DLM has only a minimal impact on the speedup of full system simulation that is achievable using SimuBoost.

We reinforce our claim by exemplary applying our measurements of the downtime and runtime overhead caused by the DLMs on the extended model of Eicher. This means we substitute the downtime parameter t_c and the DLM runtime overhead parameter s_{cp} of the total logging overhead $s_{log} = s_{rec} \cdot s_{cp}$. We have further experimentally determined the missing recording overhead s_{rec} by measuring the runtime of a VM configured with 2 GiB of main memory executing a Linux kernel build recording non-deterministic events. We have found a negligible recording overhead of $s_{rec} = 1.0097$ so we set the total logging overhead equals the DLM runtime overhead $s_{log} = s_{cp}$.

Next, we calculate the optimal interval length, given a limited number of simulation nodes, using the adjusted function provided in Formula 3.10. We notice that the optimal interval length depends on the logging slowdown s_{log} which, with respect to our experiments, in turn depends on the used interval length.

Therefore, we rearrange Formula 3.10 in order to get a lower bound for the optimal interval length[3]:

$$L_{opt}(N) = \frac{\sqrt{t_i T_{vm}}}{N} \cdot \sqrt{s_{log}} \implies L_{opt}(N) \geq \frac{\sqrt{t_i T_{vm}}}{N} \tag{3.11}$$

We presume the kernel build scenario with $T_{vm} = 960\,\mathrm{s}$ and the hardware setup of Pusch with two simulation jobs per consumer node. This means we have a total number of $N = 12$ simulation nodes available. We use the simulation slowdown $s_{sim} = 100$ proposed by Rittinghaus et al. and set $t_i = 3.7\,\mathrm{s}$ for the checkpoint loading time. As a result, we obtain $L_{opt}(12) \geq 4.967\,\mathrm{s}$ for the lower bound on the optimal checkpoint interval.

In Appendix A.4, we provide tables of DLM slowdown factors for WP and scanning. Using these tables we substitute slowdown factors s_{log}^{wp} for WP and s_{log}^{scan} for scanning for an interval length of about 5 s:

$$s_{log}^{wp} \approx 1.072 \text{ and } s_{log}^{scan} \approx 1.075$$

Therefore, we are able to approximate an optimal interval length of

$$L_{opt}^{wp}(12) \approx 4.967 \cdot 1.072 = 5.324\,\mathrm{s}$$

for WP and

$$L_{opt}^{scan}(12) \approx 4.967 \cdot 1.075 = 5.340\,\mathrm{s}$$

for scanning.

For the downtime, we presume a worst case scenario, hence we use the average downtime of an interval length of 8000 ms for WP and scanning, respectively. The corresponding data can be found in Appendix Table A.1. With these numbers we obtain a theoretical speedup of $S^{wp}(12, 5.324s) = 11.818$ for WP and $S^{scan}(12, 5.340s) = 11.817$ for scanning.

We realize that in theory the slight difference in logging slowdown of both DLMs has no significant influence on the optimal checkpoint interval length. Furthermore, even the double average downtime of scanning compared to WP does not affect the theoretical speedup. As a result, we can conclude that the choice of the DLM has no considerable effect on SimuBoost's theoretical speedup.

[3]Notice that $s_{log} > 1$ is given since the checkpoint mechanism and recording always induces at least a slight overhead.

3.2.2 Checkpoint Loading Times

In Section 3.1.1, we have pointed out that the DLM and the downtime have no significant influence on the performance of the checkpoint mechanism for the overall speedup of SimuBoost. We have discussed that we are able to evaluate the performance of the checkpoint mechanism in the context of SimuBoost by examining the slowdown of its DLM, the constant downtime, and the checkpoint loading time that directly affects the number of required parallel jobs as illustrated by Formula 3.4. The checkpoint loading times weighs even more if there is a limited number of simulation nodes as illustrated by the extended theoretical model. In this section, we focus on the checkpoint loading time. For this purpose, we recap the work of Pusch [38] on checkpoint distribution, which we have shortly referenced in Section 2.4.

Pusch compared different approaches to checkpoint data distribution. He investigated the write and read performance of direct pull access using TCP sockets and distributed filesystems (FS), such as Ceph FS [2] and GlusterFS [3]. Pusch found that direct distribution of checkpoint data results in high network load since each consumer requests checkpoint data at demand from the producer node. Furthermore, the approaches deploying distributed FS tend to cause high, scattered write delays due to internal reorganization of the distributed FS.

Therefore, Pusch decided to actively push checkpoint data to the consumer nodes using multicast. As a result, checkpoint data is *locally* available on all consumer nodes so in general there is no degradation of checkpoint loading times caused by network delays.

Nevertheless, Pusch finds checkpoint loading times of 1.5 s to 12 s depending on the number of simulation jobs per consumer node and the deployed workload. These checkpoint loading times still seem very high given the fact that all checkpoint data is locally available on the respective consumer node. We investigate his results regarding checkpoint loading times in more detail.

First of all, we shortly sum up his hardware setup and the workloads deployed on the producer VM. The producer node was equipped with an *Intel(R) Xeon(R) CPU E5-2630 v3*, *64 GiB of RAM*, and a *1 TB Samsung SSD 850*. The six consumer nodes were each equipped with an *Intel(R) Xeon(R) CPU E31220*, *16 GiB of RAM*, and a *1 TB Samsung SSD 850*. The guest executing the workload was a *single core VM* with *2 GiB of main memory* running Ubuntu 16.04.

Pusch chose a Linux kernel build [8] and SPECjbb [11] as workloads. He used Eicher's original theoretical model to calculate the optimal checkpoint interval length for his hardware setup. However, he deployed the model without adapting it to the performance characteristics of the current SimuBoost version that we have discussed earlier.

We have noticed that Eicher's basic approach to replace the constant downtime addend in the original model with a slowdown factor does not hold for the current version of SimuBoost. Our further discussion of Eicher's formulas has shown that the optimal interval length for a limited number of simulation nodes does not depend on the downtime of the checkpoint mechanism. Instead, the optimal interval length solely depends on the checkpoint loading time and the logging overhead s_{log}. Pusch could have collected data regarding the checkpoint loading time, the logging overhead, and the downtime of the current SimuBoost implementation beforehand and replace the values of the parameters in Eicher's model in order to obtain a more accurate runtime prediction and optimal checkpoint interval length. As a result, Pusch's results are only in parts suitable to argue about the importance of the checkpoint loading time for the performance of SimuBoost.

Nevertheless, we discuss Pusch's measurement results to get an idea of the issues with checkpoint loading. He measured average checkpoint loading times of 5 s for a Linux kernel build using a checkpoint interval of 4908 ms and two simulation jobs per consumer node. Increasing the number of simulation jobs per consumer node to four and decreasing the checkpoint interval to 2454 ms resulted in average checkpoint loading time of 12 s.

For further experiments he deployed SPECjbb [11] as a workload on the producer VM. Once again he ran tests with two jobs per consumer node and a checkpoint interval of 6720 ms. For this configuration he measured loading times of 1.5 s on average. In a second experiment, he increased the number of simulations per consumer to four and accordingly decreased the checkpoint interval to 3360 ms. This resulted in an average checkpoint loading time of 4 s.

These high values for the checkpoint loading time even exceed the value assumed by Eicher in his work. In general, checkpoint loading times of several seconds degrade the overall theoretically achievable speedup of SimuBoost.

For example, if we consider the configuration executing four simulations per consumer node and the Linux kernel build workload. Pusch states checkpoint loading times of 12 s on average. The theoretically achievable speedup using the adapted extended model derived in Section 3.1.1 is $19.91\times$ compared to sequential simulation. If we were able to halve the checkpoint loading times in this case, the theoretically achievable speedup increases by 7 % from $19.91\times$ to $21.3\times$. This illustrates that it is worthwhile to optimize the checkpoint loading time. Therefore, we discuss possible explanations for the high loading times in order to identify potential to optimize the mechanism.

We have to adhere the fact that checkpoint loading times are independent of the checkpoint interval length. This is because we generally restore the *complete* main memory image of a VM on a consumer node. The main memory image always has the same size that is predefined by the VM running on the producer node. Pusch hypothesizes that the general raise of checkpoint loading times, if doubling the number of simulations per node, is caused by the higher memory pressure that reduces the amount of main memory that is available for the page cache which speeds up disk accesses. Yet, Pusch does not provide data to confirm this hypothesis.

Another possible reason for the high checkpoint loading time, especially in case of the Linux kernel build, is the process of loading disk checkpoint data. In contrast to the main memory image, the amount of disk checkpoint data varies depending on the I/O activity of the respective workload. We notice that the loading times in Pusch's experiments are significantly lower for SPECjbb compared to the kernel build. SpecJBB is a memory and CPU intensive workload that does not cause disk I/O activity, whereas a Linux kernel build constantly reads and writes files from and to disk. As a result, the higher checkpoint loading times of a kernel build compared to SPECjbb may be caused by the time it takes to partially restore the disk image.

We recreate the scenarios of Pusch in order to validate the high checkpoint loading times measured by him and to check for possible reasons. At the moment there is no fully functional, reliable implementation of Pusch's multicast checkpoint distribution approach. Furthermore, up to this point SimuBoost as a whole is still work-in-progress since we have only tested single components, like the RR or the checkpoint mechanism, independently. As a result, we approximate Pusch's scenario to obtain comparable data for the checkpoint loading time. We use the same machine as Pusch used as consumer node. As a remainder, the workstations Pusch used in his experiments are equipped with an *Intel(R) Xeon(R) CPU E31220*, *16 GiB of RAM* and a *1 TB Samsung SSD 850*.

We have preliminary created continuous checkpoints of a VM configured with 2 GiB of RAM, which is executing either a Linux kernel build or SPECjbb, using the same interval lengths as Pusch[4]. We have copied all checkpoint data to our consumer node in order to simulate the locality of the checkpoint data that is achieved by the multicast checkpoint distribution approach. Pusch executed two or four QEMU emulation instances per consumer node. Correspondingly, we deploy two and four QEMU instances on our test machine. In case of two QEMU instances, the first QEMU instance executes the workload that was executed during checkpoint creation in order to simulate a comparable utilization of the machine's resources. Accordingly, in case of four QEMU instance, the first three QEMU instances execute the workload that was executed during checkpoint creation. The second and fourth QEMU instances, are repeatedly restarted in order to guarantee an empty guest state. After each restart we sequentially load the next checkpoint, which we have copied on the test machine beforehand, on the

[4]For a Linux kernel build we used an interval length of 4908 ms for two jobs and 2454 ms for four jobs. For SPECjbb we used an interval length of 6720 ms for two jobs and 3360 ms for four jobs.

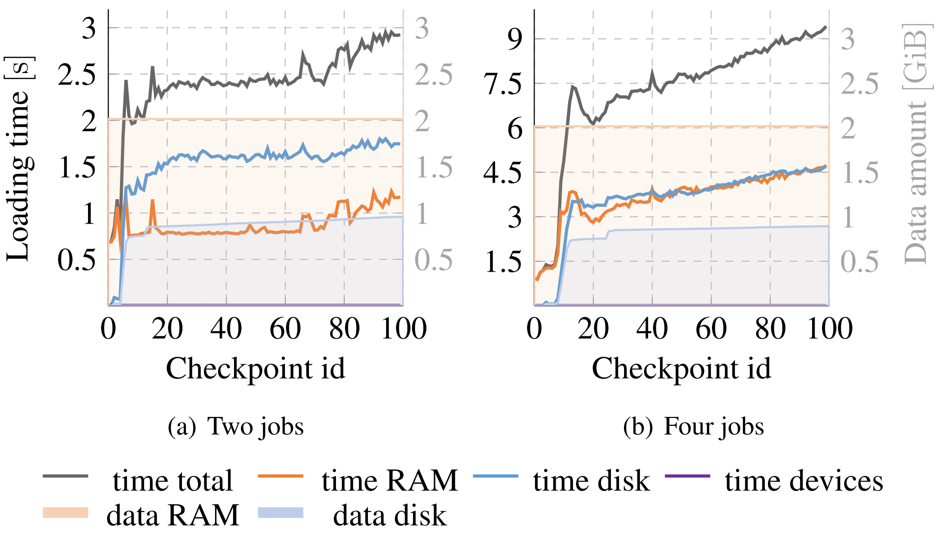


Figure 3.5: Showing the average checkpoint loading times and the average data amount out of ten runs for 100 consecutive checkpoints of a Linux kernel build. The left axis plots the loading time in seconds whereas the right axis plots the data amount that has been restored. The loading time for devices is below 5 ms for both cases so it is only slightly recognizable on top of the respective x-axis.

second and fourth QEMU instance.

We perform ten runs and provide the average loading time of these ten runs for each individual checkpoint. Before each run, we have flushed the page cache of the test machine to diminish cache related side effects between separate runs. As mentioned earlier the total checkpoint loading time consists of the guest's main memory loading time, the disk loading time, and other device loading time.

Figure 3.5 shows the average total loading time for 100 consecutive checkpoints for two and four jobs. The first checkpoint was taken when the VM was booting. Furthermore, we provide the fractions of main memory, disk, and devices on the total checkpoint loading time and the amount of disk data and RAM data that has been restored. The workload executing on the checkpointed VM was a kernel build.

Initially, we notice the aforementioned constant data amount that needs to be restored for the VM's main memory whereas the amount of disk data increases continuously from 700 MiB to about 1 GiB for two and four jobs. Correspondingly, the disk loading time in both cases raises slightly over time as well. Interestingly, for two jobs the disk loading time is significantly higher than the RAM loading time. For four jobs on the other hand, the disk and the RAM loading times are approximately the same. In total, we measure an average checkpoint loading time of 2.4 ± 0.043 s for two jobs, and 7.189 ± 0.21 s for four jobs running

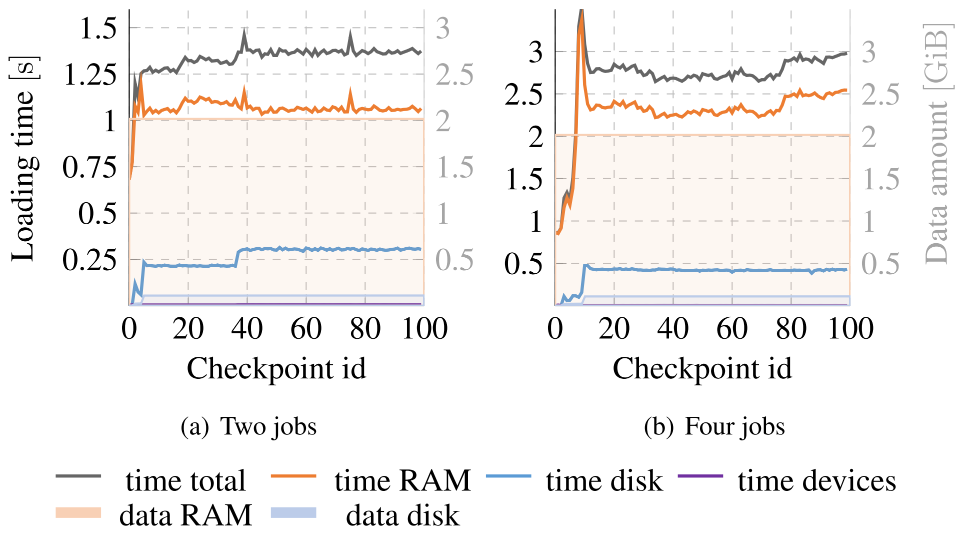


Figure 3.6: Showing the average loading times and the average data amount out of ten runs for 100 consecutive checkpoints of SPECjbb. The left axis plots the loading time in seconds whereas the right axis plots the data amount that has been restored. The loading time for devices is below 5 ms for both cases so it is only slightly recognizable on top of the respective x-axis. The amount of disk data per checkpoint is noticeable above the respective x-axis at about 100 MiB.

concurrently on the test machine.

Figure 3.6 shows the average loading time of ten runs for 100 consecutive checkpoints of SPECjbb for two and four jobs. The total loading time of SPECjbb is with 1.331 ± 0.01 s for two jobs and 2.672 ± 0.045 s for four jobs about half to a third of the loading time of the kernel build. Notably, the RAM loading time of SPECjbb corresponds to the RAM loading time measured for the kernel build for two parallel simulation jobs. The RAM loading time of SPECjbb for two jobs is on average 1.063 ± 0.006 s and the RAM loading time of the kernel build for two jobs comes to 0.882 ± 0.014 s. Yet, the total loading time of a kernel build for two jobs is twice as high as the total loading time of SPECjbb. This is solely due to the high average disk loading time of 1.515 ± 0.037 s of the kernel build compared to on average 0.263 ± 0.006 for SPECjbb.

These disk loading times are also represented in the amount of disk data per checkpoint. For SpecJBB only about 100 MiB of disk data is loaded per checkpoint, whereas the kernel build causes between 700 MiB and 1 GiB to be restored. This confirms our hypothesis that the kernel build workload induces more disk I/O activity than SPECjbb during checkpointing thus increasing the amount of disk data that needs to be restored during checkpoint loading which in turn significantly influences the loading time.

The disk I/O activity provides an explanation for the increasing checkpoint loading times of a Linux kernel build for four jobs compared to SPECjbb. The disk I/O of the three load generating VMs requires additional main memory resulting in increased memory pressure and swapping on the host system. As a result, the checkpoint loading times of the fourth VM degrade over time.

In summary, our measurements are lower than the loading times measured by Pusch, however we have only loaded the first 100 consecutive checkpoints. Nevertheless, we already observe the same effect as Pusch thus doubling the number of simulation jobs per consumer node from two to four increases the checkpoint loading time significantly.

We have been able to identify the increased amount of disk data that is being checkpointed during a Linux kernel build to be responsible for the higher total checkpoint loading time of SPECjbb compared to a kernel build for two simulation jobs running concurrently on the test machine.

We have already discussed the implications of a high checkpoint loading time for the overall feasible speedup of SimuBoost. In the model of Rittinghaus, the checkpoint loading time directly affects the number of simulation nodes that are required to achieve an optimal speedup. Besides, in the extended model, which describes the likely scenario of a limited number of simulation nodes, the loading time influences the optimal interval length and with it the speedup of SimuBoost compared to common full system simulation. Therefore, in the following sections, we discuss approaches to reduce the data amount that has to be restored during checkpoint loading in order to reduce the loading time.

3.3 Sparse Checkpointing

In Section 3.2.2, we have recapitulated experiments performed by Pusch in order to identify reasons for the high checkpoint loading time, which he observed in the evaluation of his checkpoint distribution approach using multicast.

We found that the checkpoint loading time rises with increasing number of parallel simulation jobs per consumer node. This is mainly due to the limited amount of main memory on the test machine. A solution to this problem is simply equipping the consumer nodes with more main memory so they do not run into situations of high memory pressure. Yet, RAM is expensive and it is not always feasible to purchase multiple gigabyte of RAM for several machines. A more sophisticated solution solves the problem of memory pressure and resulting high loading times in software.

In general, the time needed to load data, regardless of the type of storage or the kind of data, has a lower bound given by the *amount of data* that is to be loaded and the performance specifications of the hardware, namely, the disk, main memory, the bus system, and possibly the CPU. We have already stated that it is no option to simply purchase better hardware in means of more RAM or hardware that provides a higher throughput. Therefore, the only adjustable parameters, which influence the checkpoint loading time, are the SimuBoost store, which fetches checkpoint data from disk, and the raw amount of data that needs to be retrieved from disk. This work does not aim to engineer a more efficient storage system for SimuBoost thus we presume the performance of SimuBoost is not an issue.

For the remainder of this section, we discuss two approaches to reduce the amount of data of VM checkpoints in the related field of VM migration in order to gain insights on how to optimize the checkpoint mechanism of SimuBoost, especially in regard to the checkpoint loading time.

3.3.1 Checkpoint Data Reduction

In Section 2.2.2, we have shortly discussed the work of Clark et al. [20]. They performed write working set analysis to identify unused page frames in order to optimize pre-copy.

They state that pre-copy induces a lot of overhead by frequently transferring page frames that are repeatedly modified in subsequent copy rounds. Therefore, they provide a write working set analysis of various benchmarks in order to identify the subset of page frames that are frequently modified and thus, no good candidates for pre-copy. They found that the write working set highly depends on the workload of the VM. Yet, in general there are always page frames that are not being modified during a certain time period and are thus suitable for pre-copy.

Hines et al. [29] aimed to find unused page frames of a VM in order to improve the performance of post-copy based live migration. They modified Xen's ballooning mechanism so it continuously inflates and deflates its balloon depending on the memory utilization of a VM during its life time. Page frames that are claimed by the ballooning mechanism are not transmitted during migration since they are considered to be in the free list of the guest OS, hence not in use by the VM. Their measurements show that the number of pages transferred during migration is significantly less with ballooning enabled. As a result, it seems likely that identifying free or rather unused pages of a checkpoint and skipping them during checkpoint loading leads to lower loading times.

The work of Liu et al. [33] confirms this assumption. Liu et al. extended Xen's checkpoint mechanism to omit free pages during checkpoint creation as well. They modified Xen's ballooning mechanism in order to reclaim free page frames from the checkpointed VM. Yet, in contrast to Hines et al., they did not use a continuous ballooning mechanism. Instead, their balloon inflates and allocates unused page frames just once, right before checkpoint creation in order to reduce the memory footprint of the VM. They evaluated their checkpoint implementation by measuring the checkpoint size, checkpoint creation time, and the restart time, hence the time it takes to load a checkpoint.

Liu et al. found that the checkpoint size for a VM with 1024 MiB of main memory shrinks from about 1024 MiB to 256 MiB. Accordingly, the checkpoint loading time is reduced by about 67 % from 24 s to 8 s.

We do notice that all three approaches, which we have discussed so far, have one thing in common. They all in parts identify the working set of the checkpointed VM and leverage this information in order to systematically omit page frames during checkpoint creation so there are less page frames to be restored during checkpoint loading.

The approach of Liu and Hines et al. depends on a paravirtualized guest kernel that deploys a ballooning module in order to reclaim free page frames from the checkpointed guest. This interference with the execution behavior of the guest OS is not acceptable in the context of SimuBoost. SimuBoost aims to speedup full system simulation in order to render full system analysis of production workloads feasible while only taking a fraction of the execution time of common full system simulation. The deployment of a kernel module to reclaim unused page frames would tamper with the results of potential analysis, e.g., full system memory traces. As a result, the techniques proposed by Hines et al. and Liu et al. are not applicable to improve the checkpoint loading time of SimuBoost. Yet, the general idea to identify the working set of the checkpointed VM and omit those page frames that are not in the working set during checkpoint loading is feasible for SimuBoost.

As a remainder, SimuBoost splits the execution of a VM into simulation intervals of fixed length L by creating continuous checkpoints at a rate of $r_{cp} = \frac{1}{L}$ checkpoints per second. SimuBoost further logs non-deterministic events that affect the course of execution of the VM. Each simulation interval represents a finite entity that is emulated by a dedicated simulation node. A simulation interval i starts with a checkpoint cp_i and ends with the subsequent checkpoint cp_{i+1}. The emulator avoids divergence of the simulation from the original execution by replaying non-deterministic events at matching landmarks.

We realize that each simulation interval represents a *predefined* time interval of the execution of the original VM. In this context, predefined does not only mean that the interval boundaries represented by the checkpoints cp_i and cp_{i+1} are fixed, but also that the complete course of execution of an interval i is determined as we are replaying all non-deterministic events that have been recorded in the original VM. Therefore, during each time interval, the VM is only accessing a predetermined subset of the VM's total RAM – the working set of the individual interval as illustrated by Figure 3.7.

We notice that the working set of an interval i is terminated once the VM

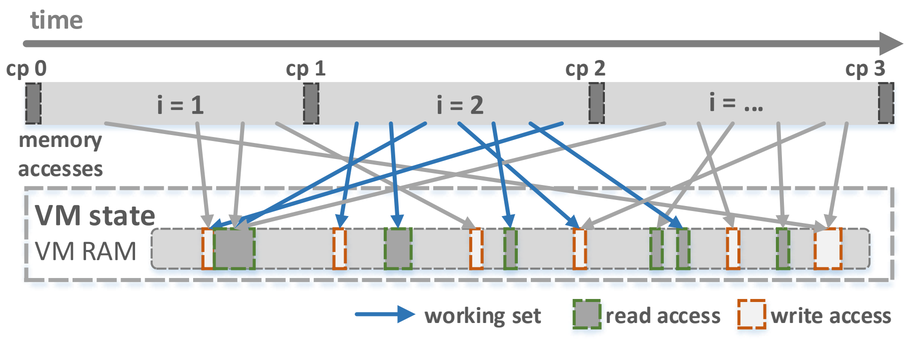


Figure 3.7: Showing a section of the execution of the producer VM. The execution is split in three intervals by the checkpoints cp_0 to cp_3. Over time the VM accesses portions of its mean memory. All memory accesses during a single interval make up the working set of said execution interval.

is stopped for creation of checkpoint c_{i+1}. As a result, it is possible to acquire the working set of each simulation interval while the producer VM advances and stores the working set information of each checkpoint interval with the checkpoint data. During concurrent simulation and corresponding checkpoint loading, we are able to access the working set information of the respective checkpoint and only load those page frames that are in the working set. Henceforth, we refer to the process of only restoring the main memory working set of a simulation interval as *sparse checkpointing*. Analogously, we refer to the checkpoint cp_i, that marks the beginning of simulation interval i and for which we have only restored the working set of the main memory, as *sparse checkpoint*.

We discuss the working set analysis performed by Werner [47] in order to estimate the potential reduction of checkpoint data that is feasible for the checkpoint loading mechanism of SimuBoost.

3.3.2 Working Set Analysis

Werner [47] performed an extensive working set analysis of continuous checkpoint intervals for varying interval lengths. We have already shortly discussed the results of his work in Section 3.2.1 in order to explain the convergence of the runtime overhead and the accumulated downtime of scanning and WP with increasing interval lengths. In this section, we further discuss the experimental results of Werner and its implications for the efficiency of sparse checkpointing.

First of all, we shortly introduce the theoretical model of Denning [21] from the year 1968 that has been referenced by Werner in his work. Denning proposed a theoretical model to argue about the resource demands of a computation system without depending on external hints from the compiler or the user. This allows an

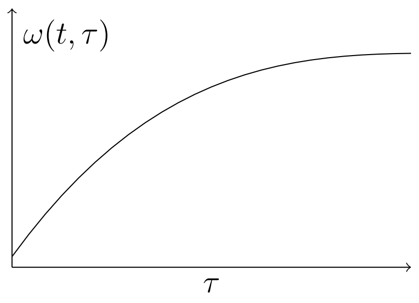


Figure 3.8: Illustrating the behavior of the working set size $\omega(t,\tau)$ in dependence of the execution time interval τ for a fixed t. The working set size growth rate decreases the longer the execution time interval τ gets [21].

OS to determine program behavior on its own and allocate resources accordingly. Denning focused on main memory and processor time as the main computation resources that need to be allocated to programs. He introduced the notion of a program's working set to be a model of the behavior of a program over time. When he talks about the main memory working set he refers to the *working set of information* meaning the "smallest collection of information that must be present in main memory to assure efficient execution" [21]. Denning defines the working set of information of a process at time t to be the data a process references during a fixed execution time interval $[t-\tau,t]$ with $\tau > 0$. Accordingly, he defines the working set size, hence the number of page frames contained in the working set to be $\omega(t,\tau)$. He provides a simple argument which shows that the working set size $\omega(t,\tau)$ is a concave function of the execution time length τ given a fixed point in time t as illustrated in Figure 3.8.

For further discussion we do not differentiate the terms working set and working set size as done by Denning. Instead, we derive the respective meaning of the term working set from the context it is used in.

The model of Denning is directly applicable to theoretically argue about the behavior of the size of the working set if we increase the checkpoint interval length. The execution time interval τ simply corresponds to the interval length L of a simulation interval i. The point in time t, at which the working set is determined, corresponds to the creation time of checkpoint cp_{i+1}, which in turn ends the simulation interval. This means that according to the model of Denning, the working set of the simulation intervals of SimuBoost is supposed to be a concave function of the interval length.

Werner [47] deployed full system simulation and heterogeneous RR to determine the working set for a fixed checkpoint interval. He leveraged and modified the already existing memory trace hooks of SimuBoost's QEMU version in order to derive the read-only working set, the write working set, and the complete

Interval length [ms]		500	1000	2000	4000	8000
Working Set	abs.	24362	29070	33177	37361	41640
	rel.	4.65	5.54	6.33	7.13	7.94
Read Working Set	abs.	23434	28234	32389	36429	40512
	rel.	4.47	5.39	6.18	6.95	7.73
Write Working Set	abs.	17071	21808	25859	29694	33478
	rel.	3.26	4.16	4.93	5.66	6.39
Excl. Read Working Set	abs.	7290	7261	7317	7667	8161
	rel.	30.70	25.54	22.62	21.12	20.03

Table 3.1: Showing the working set sizes for a Linux kernel build as measured by Werner [47]. The absolute value describes the number of pages frames included in the working set. The relative value is in percentage of the test VM's main memory which was 2 GiB. The relative value of the exclusive read working set is in percentage of the complete working set.

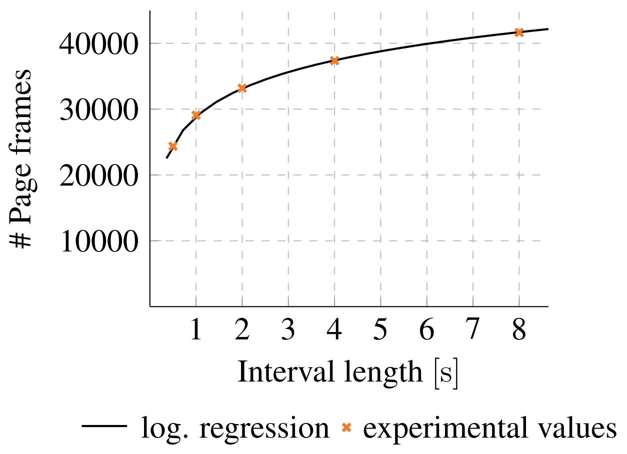
working set. He performed measurements for varying interval lengths of 500 ms, 1000 ms, 2000 ms, 4000 ms, and 8000 ms. His test VM was equipped with 2 GiB of main memory.

Unfortunately, he states that the results of his measurements do not meet central, invariant criteria. He does not provide any assumption what could have lead to the inaccuracy of his data. Nevertheless, he argues that his results provide a valid approximation of the actual working set sizes. For that matter, his measurements of the write working set approximately match the working set measurements performed by Baudis [15] in the course of his work on incremental checkpointing. Furthermore, Werner's working set measurements while running the *stress* workload generator in the test VM showed concrete results.

He configured stress to allocate 1512 MiB of main memory and touch one byte per page frame. This process of touching about 75 % of the test VM's main memory is observable in the size of the working sets he measured. Yet, some numbers he provides do not add up considering the fact that for a checkpoint interval of 500 ms there are 20000 page frames completely unaccounted for (Appendix Table A.6).

Despite this methodical inaccuracy, we suppose his results are adequate to approximate the actual working set sizes of a simulation interval. For a Linux kernel build running on the test VM, he found that the complete working set grows less the longer the checkpoint interval is chosen. He provides the working set size in percent of the test VM's main memory size of 2 GiB. His results can be found in Table 3.1.

We fit a logarithmic linear regression model (LLRM) to the data provided by



$f(x) = a \cdot ln(x) + b$			
coefficients	estimate	std. err	p-value
a	6,181.52	82.55	$5.25 \cdot 10^{-7}$
b	−13,863.1	632.69	$2.08 \cdot 10^{-4}$
multiple r-squared	0.9995		

Figure 3.9: Showing the logarithmic linear regression model derived from the results of Werner [47] for a Linux Kernel build.

Werner in order to predict the working set size for a VM equipped with 2 GiB that executes a Linux kernel build. This allows us to estimate the working set size for an arbitrary simulation interval length which in turn enables us to approximate the potential savings of sparse checkpointing if applied in the scenario of Pusch that we have recapitulated in Section 3.2.2.

Figure 3.9 shows the plot of the LLRM and the original data points of Werner. Furthermore, we provide the quality criteria of our model in order to argue about its validity. First of all, we notice the good visual fit of the model to the original data points of Werner. This good fit is confirmed by the coefficient of determination r^2 that indicates how well observed values are replicated by the given model. The low p-values indicate a high significance of the model. Altogether, this confirms the validity of our model and allows us to use it to estimate the savings we can achieve using sparse checkpointing.

For our discussion of potential loading time savings, we refer to the scenario that we have applied in Section 3.2.2. As a remainder, we were running three emulation instances in parallel, which were generating load on our test machine, while we have loaded 100 consecutive checkpoints in an additional emulation instance in order to recreate the scenario of Pusch's experiment. In our setup we measured total average checkpoint loading times of 7.189 ± 0.21 s, whereat loading of 2 GiB guest main memory took 3.625 ± 0.09 s on average.

The checkpoints were created using an interval length of 2454 ms. We can use our LLRM to estimate the working set size for said interval length and get a working set size of $6181.52 \cdot ln(2454) - 13863.1 \approx 34387$ page frames relating to about 134 MiB of main memory that need to be restored. This means sparse checkpointing could reduce the data amount of main memory that needs to be restored by about 93.5 % from 2 GiB to 134 MiB. If we presume a RAM checkpoint loading time of 3.625 s for 2 GiB we could therefore reduce the RAM checkpoint loading time to 237 ms on average. This leaves us with a total checkpoint loading time of 3.801 s since we still must restore the variable sized disk checkpoint data.

If we recap the functionality of SimuBoost, we notice that a simulation interval does not even require the write working of a checkpoint interval in order to replay properly. This is due to the fact that RR must only guarantee the determinism of the input of the VM. In case of the guest's main memory, this implies that the replay mechanism only needs the read working set of a checkpoint interval since the read working set represents the input to the VM. The write working set on the other hand does not influence the functionality of the replay of a single interval at all. Therefore, by only considering the read working set we could further optimize sparse checkpointing even though the read working set is not significantly smaller than the complete working set according to Werner.

3.3.3 Disk Checkpointing

In general, disk I/O on real hardware is performed by employing direct memory access (DMA). DMA enables a device to communicate directly with the memory subsystem without further involvement of the CPU. The CPU simply has to issue the DMA for a device by communicating with the DMA controller. Once the data transfer completes, the DMA controller triggers an interrupt to notify the CPU. Full system simulators, such as QEMU, emulate DMA in order to be fully transparent to the guest OS running inside the provided VM.

For the purpose of SimuBoost, this allows for another optimization of the checkpointing mechanism. We have already discussed that SimuBoost implements heterogeneous RR hence the producer VM records all non-deterministic events that affect the execution of the VM in order to replay them in the concurrent simulations. The recorded non-deterministic events include DMA of the virtual disk. This log of the DMA communication does not only include metadata, for example, the landmark of the event, yet also the transferred disk contents itself.

As a result, it is feasible to completely omit disk checkpointing during checkpoint creation since all read disk contents are recorded by logging DMA in the producer VM. This in turn means that there is no disk data which we have to restore during checkpoint loading. The disk data is rather directly injected into the simulation when the replay mechanism processes a recorded disk DMA event from the log.

This straightforward approach has only minor effects on checkpoint creation since disk checkpointing induces an additional downtime of less than 1 ms and the disk CoW mechanism operates in a separate independent thread.

Nevertheless, its effect on the checkpoint loading times for workloads with a high disk I/O activity is noticeable if we consider our experiment in Section 3.2.2. We found that the process of restoring disk checkpoint data accounts for the majority of the time of checkpoint loading for a Linux kernel build executing in the test VM. We found disk checkpoint loading times of 1.515 ± 0.037 s if the test machine was executing two simulation jobs in parallel and 3.561 ± 0.123 s for four parallel simulation jobs.

This means checkpoint loading time shrinks to the sparse checkpoint loading time that we have discussed in the previous section plus the negligible device loading time. As a result, by omitting disk checkpointing completely due to RR, we are able to obtain a checkpoint loading time of about 237 ms for a VM equipped with 2 GiB of RAM and an original checkpoint interval length of 2454 ms. This results in potential savings of checkpoint loading time between 90.1 % and 96.7 % compared to the original loading times for a kernel build, depending on the utilization of the consumer node.

3.3.4 Reduction of Main Memory Footprint

At last, we discuss the ramifications that come with the use of sparse checkpointing and circle back to requirement (c), regarding the memory footprint of our simulations. At the end of Section 3.3.2, we have approximated the data amount that would have to be restored if we load a sparse checkpoint. We found that the RAM checkpoint size shrinks from 2 GiB to about 134 MiB.

As a result, if we load a checkpoint we only have to restore 134 MiB of guest main memory for the emulation to be operational for the predefined simulation interval. This could also drastically reduce the memory footprint of a VM since the emulator maps GPAs to HVAs and host virtual memory must not be backed by physical memory. Yet, we have to keep in mind that the memory footprint of an emulation does not only consist of the VM's main memory, but also of the code cache of the DBT and general data structures of the emulator. Additionally, the replay extension of the emulator allocates main memory in order process the event log and corresponding data.

Nevertheless, even if the actual memory footprint of an emulation is $10\times$ the actual sparse checkpoint size, we would be able to deploy more simulations per consumer node compared to non-sparse checkpointing which is beneficial for SimuBoost since the number of simulation nodes is critical for the speedup in Rittinghauses original model.

3.4 Conclusion

In Section 3.1, we have collected three requirements on the checkpoint mechanism of SimuBoost. In the course of this chapter, we have been discussing experiments to further argue about said requirements and to find possible optimizations for continuous checkpointing in the context of SimuBoost.

First of all, we have performed measurements to evaluate the process of checkpoint creation in order to identify room for improvement. We have compared the induced downtimes of the available DLMs, WP and scanning, for different workloads and have measured their runtime overhead. We have found that the accumulated downtime and the runtime overhead of WP and scanning converge the higher the checkpoint interval length is chosen. Furthermore, we have discussed the general theoretical performance degradation of SimuBoost caused by the downtime and the DLM slowdown. We have concluded that their impact on SimuBoost's performance is negligible in theory, despite their perturbation of the execution of the workload.

Furthermore, we have recapitulated the work of Pusch. Pusch observed high checkpoint loading times even though checkpoint data was locally available on the respective consumer nodes. We have conducted experiments and were able to recreate those high checkpoint loading times. We have found that they are mainly caused by main memory exhaustion of our test machine.

Therefore, we have been discussing methods to reduce the checkpoint data amount in order to reduce the loading time. As a result, we have introduced sparse checkpointing as a way to significantly reduce the data amount when restoring the main memory of a VM.

In Section 3.3.2, we have recapitulated the work of Werner [47] regarding the working set analysis of simulation intervals. We have utilized his results to create a logarithmic linear model in order to argue about the savings of checkpoint data amount and with it checkpoint loading time that are achievable by deploying sparse checkpointing.

In summary, we find that sparse checkpointing can reduce the checkpoint loading time by up to 96.7 %. Furthermore, sparse checkpointing might even reduce the memory footprint of SimuBoost's simulations so we can effectively employ more simulations per consumer node which drastically increases SimuBoost's concurrent simulation performance.

Chapter 4

Design and Implementation

In Chapter 3, we have provided a detailed analysis of the existing checkpoint mechanism of SimuBoost. We have identified the checkpoint loading time and the number of simulation nodes to be critical for the speedup of SimuBoost. We have found that there is potential to significantly reduce the data amount during checkpoint loading by deploying *sparse checkpointing*. The idea of sparse checkpointing bases upon the assumption that the availability of working set information of simulation intervals at the time of checkpoint loading can drastically reduce the data amount that needs to be restored for a single checkpoint, thus lowering the checkpoint loading time. Furthermore, the reduced checkpoint data amount potentially lessens the memory footprint of the emulation of a finite simulation interval. This allows for the employment of additional simulations on a single consumer node.

In this chapter, we discuss the design decisions that lead to our implementation of sparse checkpointing for SimuBoost. Furthermore, we introduce our general implementation and point out interesting implementation details.

There are three basic components to our sparse checkpointing approach. First of all, we have to acquire the working set information of each checkpoint interval. The working set information is composed of the page frames that have been accessed during a checkpoint interval. Additionally, we must find a convenient way to store said working set information so it is easily accessible during checkpoint loading. Lastly, when restoring a certain checkpoint, we have to incorporate the working set information of the corresponding simulation interval in order to only restore those page frames that will be accessed during the respective simulation interval.

Our implementation bases upon the Linux kernel 4.3.0 and the QEMU version 2.6.0. We deploy *SimuTrace* [39] in version 3.4.1 as a fast and efficient storage backend. Besides, the implementation has to be compatible with existing functionality of SimuBoost, e.g., the record and replay implementation.

4.1 Acquiring Working Set Information

Denning [21] defines the working set information as the collection of data which a process references during a time interval. In case of SimuBoost, we define the working set as the collection of page frames that have been accessed during a checkpoint interval. We define a checkpoint interval to be the time between two consecutive checkpoints cp_x and cp_{x+1}.

A page frame has been accessed if data was read from it or written to it. SimuBoost already acquires the dirty working set[1] for the purpose of logging dirty page frames for the checkpoint mechanism. As we have already discussed SimuBoost comes with two DLMs – WP and scanning.

In case of WP, SimuBoost's fork of KVM write protects the guest's main memory by setting the read / write flag of the EPT entries of the guest's main memory to read-only access. A write access to a read-only page frame causes a page-fault, thus the VM exits to KVM. KVM keeps track of write accesses and synchronizes the page frame number (PFN) of the written page frames to the so-called dirty bitmap. Each bit of the dirty bitmap represents a PFN. If a bit is set it means the corresponding page frame is dirty. Later on the dirty bitmap is used to identify and copy dirty page frames to storage.

In case of scanning on the other hand, KVM walks the guest's EPTs in software. If it reaches a leaf entry, it tests the dirty bit of the page table entry and synchronizes the dirty information of the page frame to the dirty bitmap. The dirty bits of the page table entries are sticky [31]. This means the hardware does not clear them. As a result, after KVM synced the dirty bit to the dirty bitmap it clears the dirty bit of the respective EPT entry.

We notice that WP is not suited as a mechanism for collection of complete working set information since a read access to a read-only page frame does not trap into KVM. Scanning on the other hand leverages the dirty flag of the EPT entries. Besides a dirty flag, the paging structures of x86 provide an *accessed flag* that indicates whether a page frame has been written to and / or read from by the CPU. We can test the accessed flag of the EPT entries when performing a page table walk to acquire the complete working set of a VM during a checkpoint interval.

At the end of Section 3.3.2, we have briefly considered the possibility to further optimize sparse checkpointing by only incorporating the read working set, since it is sufficient to guarantee the correct replay of a single interval. This is

[1]Note that we have to differentiate between the write and the dirty working set, because the write working set is a subset of the dirty working set on Intel's x86 architecture. This is because Intel's architecture treats every processor memory access, regardless whether it is a read or write access, to the guest's paging-structures as a write hence the dirty bit is set [31]. As a result the dirty set also contains some page frames that have been read from instead of written to.

technically not possible when utilizing the accessed and dirty flags of the EPT entries.

If we want to acquire the read working set, we are supposed to differentiate whether an accessed page frame has been read and then written or only written to so we can omit it. Unfortunately, this is not feasible since the modification of a page frame by the CPU always causes the dirty and the accessed bit to being set in the respective EPT entry. As a result, we would have to single-step each memory access in order to determine whether the accessed bit was set before a write access is performed. This approach induces a significant runtime overhead [47]

As a result, we discard the optimization to only acquire the read working set and leverage the accessed bit of the EPT entries to determine the complete working set of each checkpoint interval.

4.1.1 Synchronization of Access Information

We have extended the existing scanning DLM to also create an accessed bitmap along the way of creating the dirty bitmap for checkpointing. This process includes the adaption of the shared kernel memory buffer that contains the dirty bitmap to also include our accessed bitmap and to synchronize the access information of the EPT entries to said accessed bitmap.

The synchronization of the dirty and access information is performed when the VM is stopped in order to acquire a consistent state of the VM. Henceforth, we refer to dirty and access information as state information. Figure 4.1 shows the components of QEMU and KVM that have been modified to support sparse checkpointing.

QEMU is executing the checkpoint code in a dedicated thread. We presume the dirty and accessed bitmap are already set up. The main memory checkpoint creation starts with a call to `sb_save_ram_sync_cow`.

This function initializes the CoW mechanism for a specific checkpoint so QEMU is able to asynchronously copy dirty page frames once the VM continues execution. To that end, it performs an ioctl that in turn calls the KVM function `kvm_cow_sync_and_protect`, which, in case of scanning, recursively walks the EPTs of the guest in order to synchronize the dirty information of the EPT entries to the dirty bitmap that is used by the checkpoint mechanism.

Furthermore, it synchronizes dirty information from QEMU in user space to the common dirty bitmap. QEMU must maintain its own dirty information in user space since it emulates DMA which directly modifies guest page frames in the execution context of the host. We have extended the recursive EPT walk to not only take into account the dirty bit of an EPT entry but also the accessed bit. We do not have to synchronize access information of page frames that have been read

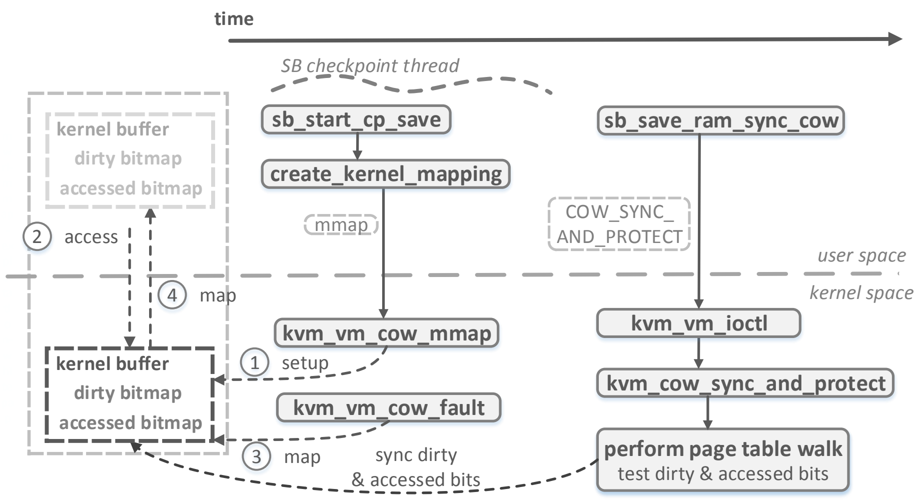


Figure 4.1: Visualizing the functions of QEMU and the Linux kernel that we have adapted for SimuBoost's checkpoint mechanism. The figure shows which functions we have modified to implement sparse checkpointing. The steps marked with numbers from (1) to (4) depict the creation of a kernel memory buffer containing the dirty and the accessed bitmap. The kernel memory buffer is mapped to user space by registering the `mmap`*-handlers* `kvm_vm_cow_mmap` *and* `kvm_vm_cow_fault`*.*

in user mode by QEMU since only data accesses by the virtual CPU are relevant for the replay which are covered by the EPTs

There is a special case for synchronizing dirty and access information to the corresponding data structures in which the dirty and accessed bit in the EPT entries have not been set by the CPU. Just as QEMU emulates certain devices, there are certain CPU instructions that are not directly executed by the CPU, yet rather emulated by KVM. In case of these operations KVM accesses guest memory pages directly in the context of the host so the CPU does not mark affected page frames in the EPT of the guest. KVM handles these cases by directly setting the corresponding bits for the affected page frames in the dirty bitmap. Accordingly, we extend the dedicated functions that read and write guest page frames in the context of the host to directly synchronize access information to our accessed bitmap.

Optimization In Section 4.1, we have ruled out an optimization that plans to only restore the read working set of a checkpoint since it is not feasible to distinguish between page frames that have been read and written and page frames that have only been written, if we only have the state information of the EPT entries available.

The fact that KVM modifies guest page frames in the execution context of the host allows us to, at least partly, omit page frames that are only written. For this purpose, we locate functions KVM uses to directly write to guest memory and skip the synchronization of the access information to our accessed bitmap in those cases. Table B.1 in the appendix lists KVM functions for which this applies.

4.1.2 Sharing and Storage of Working Set Information

KVM acquires the state information in the kernel space since it is a Linux kernel module. As a result, we have to export the state information from kernel space to user space as the checkpoint mechanism is part of QEMU, which runs in user mode.

There are two alternatives for sharing kernel space data structures with user space. We can copy kernel space data to user space or we map the kernel space data directly into the address space of the user mode process.

The process of transferring the dirty bitmap and the accessed bitmap is performed while the VM is stopped since we require a fixed VM state. This means the time it takes to transfer said information to user space adds up to the downtime of the checkpoint mechanism.

As a result, we avoid the copying approach since copying data takes more time than creating a mapping by adding an entry to the page tables of the QEMU process. Note, that in general an access to an unmapped page triggers an expensive page fault. Yet, we only have to handle the page fault that maps the kernel buffer the user space once, whereas the copy approach would have to copy the access and dirty information for each checkpoint. Furthermore, we are able to provoke the page fault once during initialization of the checkpoint mechanism by accessing the buffer in user space, thus avoiding any performance degradation during checkpoint creation.

The left-hand side of Figure 4.1 depicts the setup process of the kernel memory mapping. First of all, SimuBoost's checkpointing thread in QEMU derives the size of the kernel buffer from the amount of main memory that is assigned to the VM. A call to `mmap` issues the kernel to create a memory mapping of the specified size. We have registered an `mmap`-handler for KVM with the Linux kernel that receives and handles the call to `mmap`. Furthermore, we have registered an `mmap` -fault-handler for KVM with the Linux kernel.

The `kvm_vm_cow_mmap`-handler allocates kernel memory for our dirty and accessed bitmaps and some extra bytes for control and monitoring variables, such as the actual size of the dirty bitmap and the accessed bitmap, respectively. If the buffer is accessed in user space it causes a page fault since the virtual memory area that we have allocated in user space is not yet backed by a page frame. This page fault is handled by our `mmap`-fault-handler that returns the pages that contain our kernel buffer so the contained data structures are available in user space.

The accessed bitmap and its size are simply copied to the user data stream of SimuBoost in user space. SimuBoost extracts the binary access information and uses the LZ4 [6] compressor to reduce the size of the accessed bitmap. Afterward, SimuBoost writes the compressed binary data to a file associated to the corresponding checkpoint.

Furthermore, we collect metadata that is contained in additional variables in the kernel memory mapping. This data includes the size of the working set and the number of page frames that can be omitted due to the optimization that we have explained in the end of Section 4.1.1. This metadata is inserted into SimuBoost's checkpointing log file.

4.2 Loading Sparse Checkpoints

In this section, we introduce the functionality of checkpoint loading in SimuBoost. Furthermore, we discuss our design and implementation of the mechanism that utilizes the access information in order to optimize the process of checkpoint loading.

First of all, we shortly introduce how SimuBoost persists checkpoint data on disk. SimuBoost splits data that is relevant for a single checkpoint into two files. There is one `.ckpt`-file for each checkpoint that contains metadata describing the checkpoint. This metadata includes device information and file offsets into a large `.simuboost`-file that contains the actual checkpoint data. If QEMU requests a checkpoint from SimuBoost, SimuBoost reads the corresponding `.ckpt`-file and extracts the device specific metadata including the file offsets into the checkpoint database.

The file offsets are organized in a page-table-like, hierarchical data structure that consists of a directory of file offset tables. We further refer to this data structure as *device map*. The page frame number directly maps to a directory index and a table index in the device map so SimuBoost is able to quickly return database file offsets for given page frame numbers.

In general, SimuBoost restores the complete persisted state of a checkpoint. In case of the guest main memory, this means that SimuBoost reconstructs and returns the complete main memory image. For this purpose, SimuBoost enumerates all entries of the device map and returns the corresponding page frame number and the stored file offsets into the checkpoint database. Afterward, the file offsets are used to fetch the actual page frame data from the database. The data and the page frame number are returned to QEMU which restores the respective page frame to the guest's main memory.

In case of sparse checkpointing, we have transparently introduced a higher level iterator, the so-called *device state iterator*. It either selects the default iterator, which enumerates page frames of a checkpoint if no access information is available, or a custom *sparse iterator*, which incorporates the access information of a simulation interval and only enumerates such page frames that have been accessed during this time interval, if access information is available.

4.2.1 Accessed Bitmap Index

We analyze the distribution of main memory accesses during consecutive checkpoint intervals in order to argue about a suitable implementation of the sparse iterator. Figure 4.2 shows multiple, heatmaps of accessed page frames of randomly picket checkpoints of a guest executing a Linux kernel build. The heatmaps provide an overview of the general locality of memory accesses for a set of checkpoint intervals for this particular workload.

We notice that the more the execution of the kernel build advances the more similar the main memory access patterns of the checkpoint intervals get. Furthermore, the heatmaps depict sparse access patterns meaning that the memory accesses during a checkpoint interval focus on certain memory areas whereas other memory areas, such as parts of high memory, are almost completely omitted.

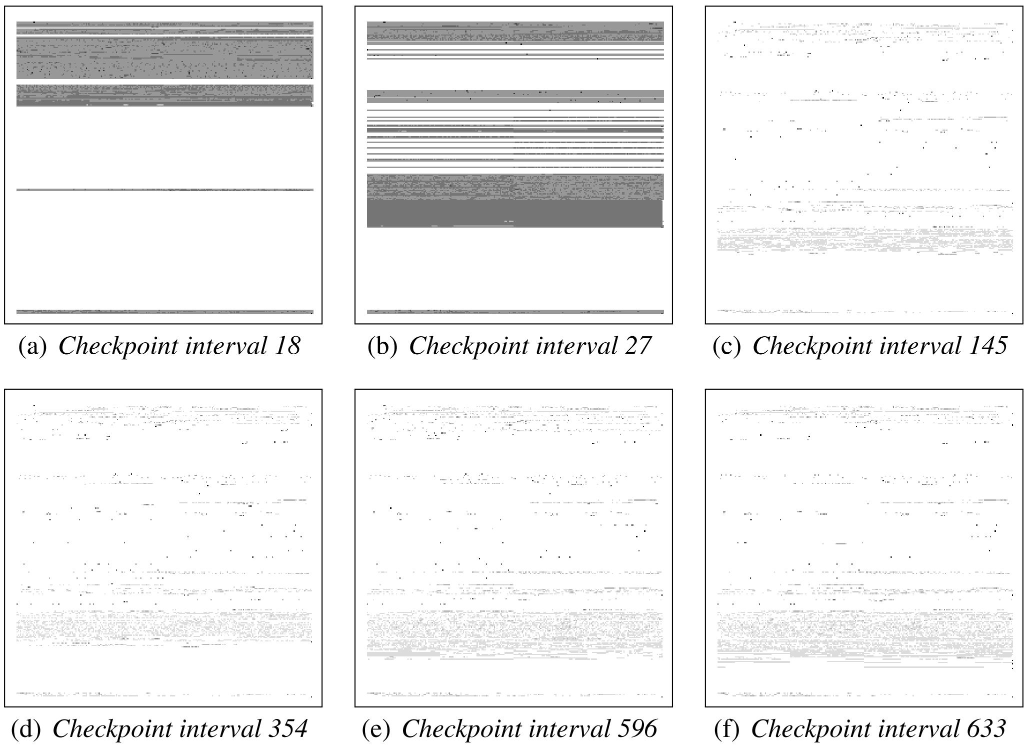
Nevertheless, we have to consider that we are only examining a sample of the set of memory access patterns for a particular workload for a specific checkpoint interval length. Yet, further examination of random samples of memory access patterns of a Linux kernel build and SPECjbb for varying checkpoint intervals reinforce our observation that in general the memory access patterns during a finite execution time interval of a VM focus on self-contained memory areas[2].

The sparse iterator is supposed to return the page frame numbers of page frames whose respective bits are set in the accessed bitmap of the corresponding checkpoint. The returned page frame numbers are in turn used by the device state iterator to return the corresponding database file offsets.

A naive implementation of the sparse iterator tests each bit of the accessed bitmap. If a bit is set, the index of the bit in the accessed bitmap represents the page frame number, which can be used to retrieve the database file offset from the main memory device map. If a bit is not set, the sparse iterator directly moves on to test the next bit.

If we consider our observations of the distribution of memory accesses, we notice that a naive sparse iterator implementation would spend a considerable amount of time iterating areas of the accessed bitmap that are completely zero.

[2]In Appendix B.2 we provide additional heatmaps of randomly sampled checkpoints for varying interval lengths of 1000 ms, 2000 ms, 4000 ms, and 8000 ms for a kernel build and SPECjbb in order to add weight to our claim.


(a) *Checkpoint interval 18* (b) *Checkpoint interval 27* (c) *Checkpoint interval 145*

(d) *Checkpoint interval 354* (e) *Checkpoint interval 596* (f) *Checkpoint interval 633*

Figure 4.2: Showing heatmaps of the memory accesses of a Linux kernel build for a checkpoint interval length of 1000 ms. The VM was equipped with 2 GiB of main memory. For the sake of convenience, each cell of the heatmap represents a cluster of eight page frames with consecutive addresses. For each heatmap the upper left corner represents the lowest page frame number and the bottom right corner the highest page frame number. The presented checkpoint intervals were picked randomly.

Our implementation thus generates an index of the accessed bitmap beforehand in order to enable the sparse iterator to skip zero regions of the accessed bitmap. The index is organized as a key-value-store. We use the standard C++ `std::map` and further refer to it as bitmap index. The bitmap index stores so-called *bitmap entry structs* (BES) that hold metadata of fixed, finite regions of the accessed bitmap.

The BES incorporates five attributes whose short description can be found in Table 4.1. The attribute `startBit` stores the current bit index. Notice, that `startBit` allows us to define an order on the bitmap entries so we use it as the key for the bitmap index. This allows the sparse iterator to quickly find BESs that start at a specific bit position. Furthermore, a BES includes attributes that represent various bit counts. The variable `numBitsSetIncl` counts the number of set bits in the accessed bitmap up to, and including, the current entry. The variable `numBitsSetExcl` on the other hand counts only the number of set

Identifier	Type	Description
`startBit`	word	Start bit of this entry
`numBits`	word	Total size of this bitmap entry in bits
`numBitsSetIncl`	word	Set bits up to this entry (inclusive)
`numBitsSetExcl`	word	Set bits of this entry
`data`	pointer	Pointer to data in the accessed bitmap

Table 4.1: Short description of the attributes of a bitmap entry struct (BES). The attribute `numBitsSetIncl` contains the number of bits that are set in the accessed bitmap up to, and including, the current BES. The attribute `numBitsSetExcl` on the other hand contains the number of set bits of the section of the accessed bitmap that is represented by the current BES, hence excluding the set bits up to this entry.

bits in the section of the accessed bitmap represented by this particular BES. The attributes `numBitsSetIncl` and `numBitsSetExcl` are important since they are used by the iterator to fast-forward the bit position of the sparse iterator in the accessed bitmap. Finally, the `data`-pointer points to the memory region of the accessed bitmap where the bit `startBit` can be found.

For initialization of the index we sequentially iterate the accessed bitmap and test words of 64 bits thus we do not test single bits. We differentiate three kinds of areas in the accessed bitmap.

First, we test whether each bit of a word is completely set to one. If this is the case we instantiate a new BES. We refer to such words as *dense* BES. If the subsequent word is also dense, hence all bits are set to one, we extend the existing BES so it now incorporates two words. This process continues until the initialization hits a word that is not dense thus it contains zero bits. In this case, the dense BES is completed and added to the bitmap index. For words that are not completely zero but contain scattered set bits, we create a new BES instance and add it directly to the bitmap index. Finally, if we hit a zero word we complete the current dense BES, if there is one, and move on to the next word.

4.2.2 Sparse Iterator

Our implementation of the sparse iterator leverages the metadata provided by the bitmap index to quickly find a specific bit in the accessed bitmap that relates to an element position of the iterator. For example, if a component that uses the sparse iterator instructs it to return all elements of the accessed bitmap starting at element 100, the sparse iterator is able to fast-forward to the 100 th set bit in the accessed bitmap by using the metadata of the bitmap index, return the corresponding page frame number, and iterate the remaining set bits in the bitmap.

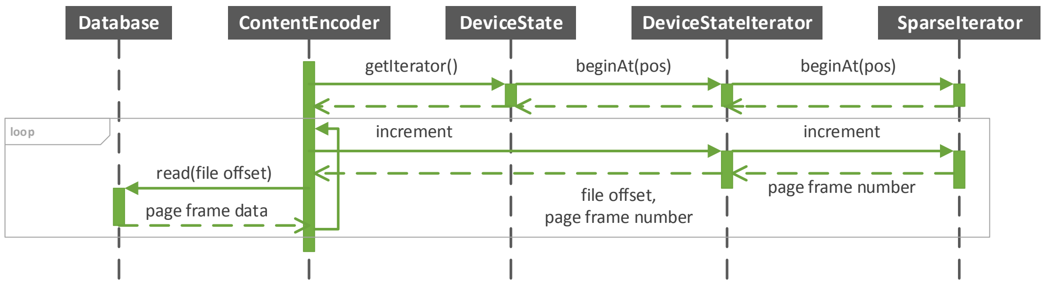


Figure 4.3: Sequence diagram showing the interaction of the components of SimuBoost when loading a checkpoint. The `SparseIterator` *is only used if there is access information available. The deployment of the sparse iterator is transparent to the device state. If there is no access information available for a checkpoint, the* `DeviceStateIterator` *uses a default iterator that simply returns all entries of the device map.*

Figure 4.3 provides an overview of the components that interact with the sparse iterator and their control flow. The `ContentEncoder` retrieves a instance of the main memory device state and instantiates a device state iterator that returns tuples of page frame numbers and corresponding file offsets. The device state iterator in turn deploys the default iterator that returns all entries of the device map or the sparse iterator that only returns entries of the device map, that correspond to the working set of a checkpoint interval, depending on the availability of access information for a particular checkpoint.

We presume that access information is available and the sparse iterator is used by the device state iterator. The content encoder then calls the device state, instantiates a `DeviceStateIterator`, and requires it to return all tuples of page frame numbers and file offsets starting at a specific position. The device state iterator passes this request directly to the sparse iterator. This means that the sparse iterator has to return the bit indices[3] of the accessed bit map starting with the `pos`-th set bit.

The sparse iterator leverages the bitmap index to speed up this process. The sparse iterator goes over the BESs of the bitmap index and tests whether the number of set bits up to this and including this BES `numBitsSetIncl`, is bigger than the requested position `pos`. If this is the case, this means that the sparse iterator has found the correct BES that contains the `pos`-th set bit.

In Section 4.2.1, we have discussed the structure of a BES. A BES may cover multiple 64 bit words if it is a dense BES. Therefore, the sparse iterator continues to find the index into the 64 bit words by testing and masking each set bit of a respective 64 bit word until it hits the requested position of the `pos`-th set bit.

[3]Keep in mind that a bit index of the accessed bitmap directly maps to a page frame number.

The sparse iterator then returns the indices of all set bits beginning at the found start position. Each call to the increment operator of the iterator results in the masking of the next set bit, the return of the corresponding page frame number, and the increment of the index into the BES and, if the end of a BES is reached, the increment of the current BES itself.

The device state iterator takes the returned page frame numbers and looks up the corresponding file offsets in the device map. The device state iterator passes both, the file offsets and the page frame numbers, to the content encoder which reads the page frame data from the database and passes it to QEMU.

4.3 Conclusion

In this chapter, we have discussed the design and implementation of sparse checkpointing for SimuBoost. We have examined the implementation of scanning for dirty logging in the context of checkpointing. We have extended scanning so it does not only acquire the dirty set but also the accessed set of a VM.

This access information is used during checkpoint loading to only restore the main memory working set of a VM for a specific simulation interval. We achieve this by introducing a high level iterator, the so-called device state iterator, that either deploys a default iterator or a sparse iterator, depending on the availability of access information.

The sparse iterator only returns those page frame numbers that correspond to a set bit in the accessed bitmap. The deployment of the sparse iterator is transparent to the device state that holds the device map and to the content encoder that fetches the actual page frame data from the database. As a result, if there is no access information available the device state iterator invokes the default iterator that simply returns all page frames from the device map.

Chapter 5

Evaluation

In Chapter 3, we have discussed the theoretical models of Rittinghaus et al. [40] and Eicher [23] and derived requirements on the checkpoint mechanism of SimuBoost. Therefore, we have compared the DLMs, scanning and WP, and found that the differences in performance during checkpoint creation, regarding the induced downtime and runtime overhead, are negligible for the overall speedup of full system simulation that is achievable by using SimuBoost.

We have further discussed the work of Pusch [38] who measured high checkpoint loading times in his experiments. We were able to approximate Pusch's scenarios and confirm his results. We identified the generally high memory pressure on the machine due to multiple simulation jobs to be the main cause for the high checkpoint loading times. We have concluded that a solution of this problem would be to reduce the data amount that needs to be restored during checkpoint loading.

As a result, we have identified sparse checkpointing as an approach to reduce the data amount during checkpoint loading. We recapitulated the work of Werner [47] on working set analysis of continuous checkpoints in order to estimate the effectiveness of sparse checkpointing. We have found that sparse checkpointing should conceptually be able to reduce the data amount during checkpoint loading by up to 93.5 %.

In this chapter, we systematically evaluate our implementation of sparse checkpointing for SimuBoost. First of all, we introduce our methodology and the evaluation setup. Next, we verify the correctness of sparse checkpoints, thus whether they can be used to correctly simulate a finite execution interval. Simultaneously, we test our hypothesis that sparse checkpointing reduces the memory footprint of emulations, allowing for the deployment of additional simulations on a single machine. This would imply a gain for the achievable speedup of SimuBoost since its simulation performance heavily depends on the number of concurrent simulations as pointed out in Chapter 3. Afterward, we evaluate the performance implications of acquiring the accessed set during checkpoint creation, regarding the induced downtime and the runtime overhead. Finally, we evaluate the checkpoint loading times under high utilization of the test machine and compare our results with the results of the experiments in Chapter 3.

5.1 Methodology

In Chapter 3, we have already discussed the fact that up to now SimuBoost as a whole is work in progress. We have tested single components of SimuBoost, e.g., the checkpoint mechanism, the recording and replay, or the checkpoint distribution via multicast, yet there is no implementation that combines all components to a fully functional distributed, heterogeneous recording and replay system that is SimuBoost.

As a result, we create test scenarios for sparse checkpointing that are as close as possible to the final application of SimuBoost. Therefore, we reconsider the experiments that we have deployed in the analysis, regarding checkpoint creation and checkpoint loading, and argue about their applicability for an evaluation of sparse checkpointing. There are five areas of sparse checkpointing that we evaluate: (1) the correctness of sparse checkpointing including a definition of correctness, (2) the maximal memory footprint of a simulation, (3) the working set size, (4) checkpoint creation, and (5) checkpoint loading.

All experiments were performed using our modified *Linux kernel version 4.3.0*[1] for checkpoint creation, *QEMU version 2.6.0* with recording and replay extensions, and *SimuTrace version 3.4.1* with sparse checkpoint extensions. The deployed VMs are all single core VMs equipped with 2 GiB of main memory and running Ubuntu 16.04 if not stated differently.

5.1.1 Sparse Checkpoint Correctness

First of all, we have to define in which case we call a checkpoint *correct*. In the context of SimuBoost, we use checkpoints, regardless if they are sparse or not, to split the execution of a VM into equally sized execution intervals. A checkpoint marks the start of an interval and the subsequent checkpoint marks the end of an interval. We simulate execution intervals concurrently on multiple simulation nodes. The simulations reenact the execution of the original VM since SimuBoost leverages deterministic recording and replay. Therefore, the simulation reads and replays deterministic input data from the recording log at predefined landmarks. The landmarks depend on the content of registers which in turn may contain data

[1]SimuBoost extensions – checkpoint mechanism and acquisition of working set.

read from main memory. The main memory of the VM itself is never replayed but rather evolves as events from the log manipulate the main memory in a deterministic way.

As a result, if the state of those main memory pages that are used by the VM during simulation is not consistent with the replay, the landmarks of the simulation and the log do not match and the replay misses events, leading to a simulation that diverges from the original VM execution. This implies that a checkpoint is correct in the sense of SimuBoost, if the replay of the corresponding simulation interval succeeds, thus QEMU is able to replay all events of the interval from the log without a mismatch of landmarks or subsequent errors. Nevertheless, notice that this is only a necessary requirement, it is not sufficient to guarantee a correct replay. This means, if the replay of a checkpoint interval fails it is not given that the checkpoint was not restored correctly since there is the possibility that the replay of the recording log is defective.

On the other hand, if a checkpoint, regardless if it is sparse or not, replays correctly, there is still the possibility that the checkpoint was defective yet the discrepancy is minor so it does not affect the control flow of the execution. For example, if data is read from one memory location and written to another, and there is no further access to the written data in the given interval, it will not cause the replay to fail even if the read data was defective.

In summary, a checkpoint is *correct* in the context of SimuBoost, independent of the employment of sparse checkpointing, if QEMU is able to load the checkpoint and replay the corresponding simulation interval. Since this criteria is only a necessary requirement, we compare the correctness rate of sparse checkpoints with the correctness rate of non-sparse checkpoints in order to argue about the validity of sparse checkpointing and our results. If sparse checkpointing achieves the same correctness rate than non-sparse checkpointing it is safe to assume the validity of sparse checkpointing. For the remainder of this paper, we refer to non-sparse checkpoints as *full checkpoints* or *full checkpointing*.

We perform the correctness experiments on an *Intel(R) Xeon(R) CPU E5-2630 v3* with *64 GiB of main memory* and a *1 TB Samung SSD 850*. We employ a 4 GiB VM in order to avoid non-determinism in the setting of accessed and dirty bits in the guest page tables, which the current recording and replay implementation does not capture. We only perform a single run per experiment since the sequential execution of the replay is extremely time consuming.

First of all, we create a complete SimuBoost log, including continuous checkpoints and record non-deterministic events. We create logs for a Linux kernel build using sparse checkpointing and full checkpointing. Afterward, we execute each simulation interval by sequentially loading the checkpoints and replaying the non-deterministic events from the log. At the end, we provide and compare the replay failure rate for sparse and full checkpoints. Additionally, we provide the checkpoint loading time for each checkpoint for sparse and full checkpointing.

We omit correctness experiments for SPECjbb at this point since its runtime is more than twice the runtime of a Linux kernel build. Nevertheless, we perform a concurrent replay experiment at the end of this chapter in order to provide correctness data for SPECjbb. Furthermore, we provide first results of the actual achievable speedup of parallel simulation on a single workstation using SimuBoost.

Note that we do not perform concurrent correctness experiments for the kernel build since our mechanism of acquiring the main memory footprint of the simulation intervals presumes sequential simulation.

5.1.2 Main Memory Footprint

At the end of Section 3.3.4, we argue that sparse checkpointing may reduce the main memory footprint of simulation intervals. We evaluate the maximal main memory footprint of each simulation interval for a Linux kernel build running inside the VM.

Therefore, we extend the experiment introduced in the previous Section 5.1.1. After loading and replaying a given execution interval, we additionally measure the physical main memory consumption of the associated QEMU process. We argue that the lower limit of the main memory consumption of QEMU is given by the size of the working set of the VM for this interval. We know that the working set is maximal at the end of the replay interval since it is a concave function as shown by Denning [21] and illustrated by the measurements of Werner [47]. Besides the VM's main memory, QEMU allocates main memory for the emulation itself, e.g., DBT, device emulation, and I/O activity. If we presume that QEMU approximately uses a constant amount of main memory for emulation of a simulation interval, we can argue that the main memory consumption of QEMU is maximal at the end of a simulation interval.

We measure the physical main memory using the `pmap` command and parse the *resident set size*. In later experiments, we use the maximal size of the average main memory footprint in order to estimate the maximal number of simulations we are able to deploy on our workstation for the concurrent sparse checkpointing correctness experiment of SPECjbb.

5.1.3 Working Set Size

The working set size of a simulation interval affects the data amount that needs to be restored during checkpoint loading and is a good measure for the synchronization overhead of sparse checkpointing during checkpoint creation[2]. We provide data regarding the size of the complete working set and the dirty set. We acquire the complete working set size for a checkpoint interval by counting the set bits of the accessed bitmap. The dirty set size is measured by default by the checkpoint mechanism. Therefore, data describing the working set size and the dirty set size are a side product of our checkpoint creation experiments.

In Section 4.1.1, we have presented a simple optimization that allows us to omit certain page frames thus not adding them to the working set. We provide data regarding the effectiveness of this optimization. Furthermore, we compare our results regarding the working set size with the working set measurements of Werner [47] in order to verify their validity.

5.1.4 Checkpoint Creation

We have already conducted experiments to analyze the performance of SimuBoost's checkpoint mechanisms regarding the induced downtime and the runtime overhead. In Section 3.2.1, we have compared both DLMs, scanning and WP, in order to argue whether they are suited for the application with SimuBoost. For this purpose, we have performed a series of experiments creating continuous checkpoints of a VM. We were running two different workloads, a Linux kernel build and SPECjbb. We have further provided a baseline by executing the same workloads without checkpointing enabled.

We evaluate the checkpoint creation of sparse checkpointing by recreating the experiments of the analysis chapter. Keep in mind that our sparse checkpoint implementation is an extension of the scanning DLM that we have analyzed in Section 3.2.1. Therefore, we are able to argue about the performance implications of sparse checkpointing regarding the induced downtime and the runtime overhead during checkpoint creation by comparing its results to the results of scanning in the analysis and to the baseline that we have acquired in the course of Chapter 3.

As a remainder, for the checkpoint creation experiments we have been using a VM configured with 2 GiB of main memory. The VM is executing either a Linux kernel build or SPECjbb as a workload. We test varying checkpoint interval lengths of 1000 ms, 2000 ms, 4000 ms, and 8000 ms. We perform ten independent runs for each interval length and each workload. Before each run we flush the page cache of the host in order to avoid cache related side effects. We execute the sparse checkpoint creation experiments on a host equipped with an *Intel(R) Xeon(R) CPU E5-2630 v3*, *64 GiB of main memory*, and a *1 TB Samung SSD 850.*

[2]Note that the working set size further is the lower limit for the main memory footprint.

5.1.5 Checkpoint Loading

During the experiments in Section 5.1.1 we have already obtained data regarding the checkpoint loading time of sparse checkpoints. Yet, we recreate the experiments of Pusch in order to evaluate sparse checkpoint loading times on a test machine with very limited hardware resources under high to medium utilization. This allows us to directly compare the results of sparse checkpoint loading with the results of Chapter 3.

The checkpoint loading experiments in Chapter 3 induced high load on the main memory of the test machine. We have claimed that sparse checkpointing is able to reduce the performance degradation of checkpoint loading due to high memory pressure. We verify this claim by recreating those experiments.

We preliminary create sparse checkpoint of a VM equipped with 2 GiB of main memory, executing either a Linux kernel build or SPECjbb. For the Linux kernel build, we use checkpoint interval lengths of 2454 ms and 4908 ms. For SPECjbb, we use checkpoint interval lengths of 3360 ms and 6720 ms. Those interval lengths stem from the scenarios of Pusch and the number of concurrent simulations that are deployed on the consumer nodes. For each workload and interval length, we perform ten runs. The resulting checkpoints are copied to the test machine so the checkpoint data is locally available just as it would be the case with multicast checkpoint distribution (see Section 3.2.2).

The actual evaluation of checkpoint loading splits into two experiments per workload. We perform checkpoint loading experiments executing two and four parallel simulation jobs on the test machine in order to approximate the load on the machine during an application in the context of SimuBoost. For two parallel jobs, we load the checkpoints that have been created using an interval length of 4908 ms and 6720 ms, respectively. For four jobs, we load the checkpoints that have been created using an interval length of 2454 ms and 3360 ms, respectively. We must pay close attention to interval length that was used to create the checkpoints we intend to load since the interval length affects the data amount of a sparse checkpoint.

We mentioned earlier that we have performed ten checkpoint creation runs. Therefore we repeat each checkpoint loading experiment ten times. The presented results are thus the average of those ten runs.

The test machine for checkpoint loading is equipped with an *Intel(R) Xeon(R) CPU E31220, 16 GiB of RAM*, and a *1 TB Samsung SSD 850*.

5.1.6 Parallel Simulation

Sparse checkpointing reduces the main memory footprint of the simulation of an execution interval. It enables us to deploy more concurrent simulation on a single workstation. As a result, it becomes feasible to use SimuBoost on a single machine. We utilize the measurements of the main memory footprint of Section 5.1.2 to estimate the maximal number of concurrent simulations on a test machine.

We then create a complete SimuBoost log, including sparse checkpoints and the recording of non-deterministic events of a VM, executing either a Linux kernel build or SPECjbb. Once the recording finishes, we concurrently load each checkpoint and simulate the corresponding execution interval by replaying the event log.

This setting is only an approximation of a real SimuBoost use case since we decouple the execution of the producer VM and the simulations for the sake of simplicity. Nevertheless, we can argue about the speedup since the runtime of the producer VM is masked by the concurrent simulations. It allows us to evaluate the speedup of SimuBoost with sparse checkpointing enabled.

Keep in mind, that SimuBoost without sparse checkpointing is only able to execute a very limited number of simulations on a machine, since each simulation of a short execution interval instantiates a full-blown VM. Therefore, a machine, such as the one we use for the correctness experiments, which is equipped with 64 GiB RAM, has approximately 40 GiB of main memory available for simulations since the remaining 24 GiB are used by SimuBoost itself and the host system. This means, we are able to run 10 simulations in parallel at max. In contrast, we suppose that sparse checkpointing allows us to deploy significantly more simulations on the same machine.

We use the test machine, which we have also used for the evaluation of correctness, maximal main memory footprint, and sparse checkpoint creation, for both recording and replaying the simulation intervals. The machine is equipped with an *Intel(R) Xeon(R) CPU E5-2630 v3*, *64 GiB of main memory*, and a *1 TB Samsung SSD 850*. We utilize the measurements of the main memory footprint that will result from our correctness experiments to estimate the maximal number of simulations. We realize that SPECjbb generally induces a higher memory load than a kernel build thus we estimate the maximal number of simulations for SPECjbb rather conservative.

We calculate the optimal interval length for creation of a full recording for the maximal number of simulations by applying Formula 3.11. We provide results regarding the correctness, the actual speedup compared to serial simulation, the efficiency of the concurrent simulation, and the average checkpoint loading time. The efficiency of the parallelization is given by the achieved speedup and the number of used nodes [40]:

$$E = \frac{S(N, L_{opt}(N))}{N} \tag{5.1}$$

5.2 Results

In this section, we present the results of our evaluation. We start with the evaluation of the sparse checkpoint correctness and the maximal main memory footprint.

We continue with the working set size and the evaluation of our sparse checkpoint optimization.

Afterward, we present the results of checkpoint creation, and finish with the results of our checkpoint loading experiments under high system load.

Additionally, we acquire the correctness rate of SPECjbb by concurrently replaying a complete recording. This concurrent replay is only feasible since we assume that sparse checkpointing reduces the main memory footprint of simulations allowing for more concurrent simulation instances on a single machine.

We complete the presentation of the results with a discussion of ramifications for the applicability of sparse checkpointing in SimuBoost. We only provide a subset of the plots in this chapter in order to keep the chapter clearly laid out. The plots and data of all experiments can be found in Appendix C.1.

We do not provide runtime related plots or data for all SPECjbb experiments since measuring the runtime of SPECjbb makes no sense as the runtime is constant and the performance is given by specific measures that describe the throughput of the system.

5.2.1 Correctness and Main Memory Footprint

We present the results of our correctness experiments first, so we are able to presume the correctness of the sparse checkpoints for the remainder of the evaluation.

We have performed the correctness tests for a Linux kernel build running in the recorded VM. The recorded VM was configured to use 4 GiB of main memory and the checkpoints have been taken with an interval length of 2000 ms.

Our experiments show that 395 out of 406 sparse checkpoints of a Linux kernel build fulfill our correctness criteria of Section 5.1.1. This is a sparse checkpoint correctness rate of 97.29 %. For full checkpointing, the correctness rate is 100 %. As a result, we have to assume that there are corner cases for which we miss memory accesses of the VM, leading to incomplete working set information during checkpoint loading. We summarize the occurred replay failure reasons in Table C.1 in the appendix.

We have suspected that the sparse checkpointing optimization, which we have introduced in Section 4.1.1, could falsify the acquired working set if there are unanticipated cases for which the optimization does not work correctly. Therefore, we have disabled the optimization and repeated the experiment. In this case, 388 out of 406 sparse checkpoints fulfill the correctness criteria which is 95.57 %.

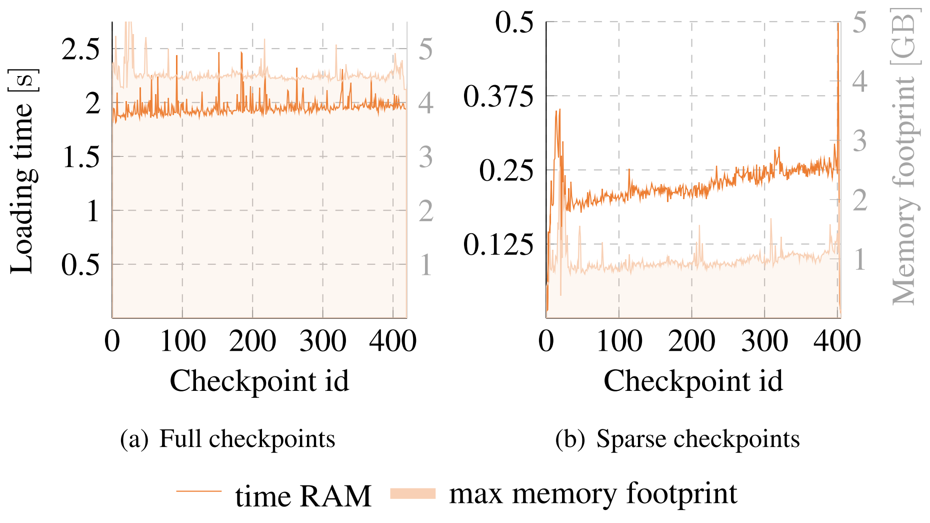


Figure 5.1: Showing the checkpoint loading time and the maximal main memory footprint for each checkpoint of a Linux kernel build for full checkpointing and for sparse checkpointing.

This correctness rate is even less than for the optimized sparse checkpointing implementation. As a result, it is save to assume that the sparse checkpoint optimization does not falsify the working set information, yet the acquisition of access information is incomplete for some basic corner cases.

Besides sparse checkpoint correctness, we have measured the maximal memory footprint of the emulations and the checkpoint loading times for each checkpoint for sparse checkpointing and for full checkpointing. In Figure 5.1, we compare the checkpoint loading times and the maximal memory footprint of sparse checkpointing and full checkpointing. The average checkpoint loading time for full checkpoints is 1.956 ± 0.004 s and the average maximal main memory footprint is 4.521 ± 0.009 GiB. For sparse checkpoints, the average loading time is 0.222 ± 0.002 s and the average maximal main memory footprint is 0.971 ± 0.012 GiB.

This means sparse checkpointing reduces the maximal main memory footprint of each emulation of a Linux kernel build by $78.52\,\%$ on average. We use the average maximal main memory footprint later on to estimate the maximal number of concurrent simulations on our test machine for parallel SPECjbb correctness experiment.

Conclusion

In summary, we do realize that there are some unresolved issues with sparse checkpointing since the checkpoint correctness rate of sparse checkpointing for a Linux kernel build is only $\approx 97\,\%$ compared to $100\,\%$ for full checkpointing. We leave the investigation of why certain simulation intervals fail to replay to future work and argue that a sparse checkpoint correctness rate of 97 % is still sufficient in order to illustrate the benefits of sparse checkpointing for the application with SimuBoost.

Our first measurements show that sparse checkpointing is able to reduce the average checkpoint loading time of a Linux kernel build by $\approx 89\,\%$ and the average maximal main memory footprint by $\approx 79\,\%$. As a result, it becomes feasible to deploy SimuBoost as a whole on a single host.

Let us presume a workstation with 64 GiB of main memory, a CPU with 32 concurrent hardware threads, and an average maximal memory footprint of a single simulation of 1 GiB. Furthermore, we assume that the producer VM uses about 4 GiB of main memory and the SimuTrace storage and other common processes use additional 20 GiB of main memory. This means sparse checkpointing would allows us to deploy up to 40 concurrent simulations on the workstation, if we just consider the main memory as a limiting factor. In contrast, for the same hypothetical setting, full checkpointing only allows for 10 concurrent simulations if we assume an average main memory footprint of 4 GiB.

5.2.2 Working Set Size

In Section 4.1.1, we have discussed the implementation of a simple optimization of sparse checkpointing that omits page frames which only have been written to by KVM in the host context.

Table 5.1 and 5.2 show the average working set size for a Linux kernel build and SPECjbb, respectively. Furthermore, we provide the average number of page frames that have been omitted due to our optimization. This means the shown working set sizes minus the number of omitted page frames yields the effective working set size that is relevant for sparse checkpoint loading. We notice that our optimization saves between $4.69\,\%$ and $7.14\,\%$ of the total working set size depending on the workload and the checkpoint interval length.

These savings do not have any influence on the performance of sparse checkpoint creation since we still have to save all page frames that have been written to in order to guarantee a consistent checkpoint. Nevertheless, those page frames that have been skipped due to our optimization will not be restored during checkpoint loading, therefore reducing the sparse checkpoint loading time and the main memory footprint of simulation intervals.

Interval [ms]	Avg. working set size	Avg. skipped page frames	Percentage
1,000	31,102	1,712	5.51
2,000	33,899	1,904	5.62
4,000	39,065	2,130	5.45
8,000	45,504	2,408	5.29

Table 5.1: Showing the average working set size and the number of page frames that could be skipped due to our optimization for a Linux kernel build for varying interval lengths.

Interval [ms]	Avg. working set size	Avg. skipped page frames	Percentage
1,000	125,356	5,875	4.69
2,000	185,189	11,330	6.12
4,000	199,446	13,516	6.78
8,000	202,848	14,492	7.14

Table 5.2: Showing the average working set size and the number of page frames that could be skipped due to our optimization for SPECjbb for varying interval lengths.

For the remainder of this chapter, if we refer to the working set size we refer to the effective working set size minus the page frames that have been skipped due to our optimization.

If we compare the measurements of the complete working set with the results of Werner [47], we realize that we find slightly higher working set sizes for all interval lengths. This is even true for the effective working set size that already omits a certain number of page frames. We argue that this can be traced back to the incomplete data acquisition of Werner since he finds that the working set sizes he measured using full system simulation are too small as they do not fulfill basic invariants.

5.2.3 Sparse Checkpoint Creation

We present the results of our evaluation of sparse checkpoint creation. We directly compare the results of sparse checkpoint creation regarding induced downtime and runtime overhead with the results of checkpoint creation using scanning as DLM since our sparse checkpoint implementation is an extension of scanning.

In this context, we refer to the implementation of scanning that additionally synchronizes access information as *extended scanning* (ext. scanning) and to the scanning implementation that only synchronizes dirty information as *default scanning* (def. scanning).

First of all, we present the results for a Linux kernel build running in the

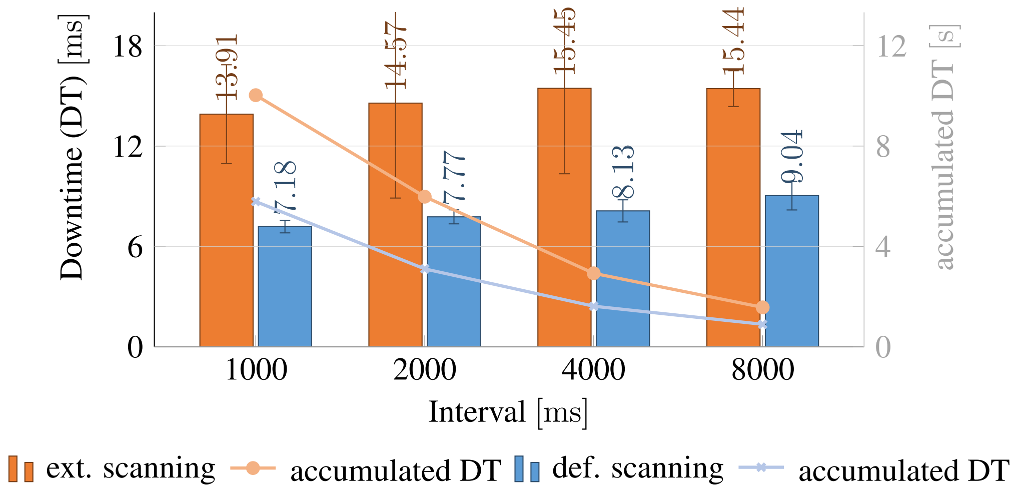


Figure 5.2: The left axis plots the average downtime (DT) of a VM executing a Linux kernel build while creating continuous checkpoints using extended scanning. The right axis shows the accumulated downtime of the checkpointed VM.

checkpointed VM for varying checkpoint interval lengths. Figure 5.2 depicts the average downtime of ext. scanning compared to def. scanning. We realize that ext. scanning induces significantly more downtime than def. scanning.

The rise of the downtime for sparse checkpointing can be explained by the additional work that ext. scanning does by synchronizing accessed bits to the accessed bitmap during the downtime of the VM. Figure 5.3 compares the average dirty set and the average complete working set for different checkpoint interval lengths. The difference between the working set and the dirty set represents the average number of page frames that sparse checkpointing must synchronize additionally[3].

Furthermore, we observe the convergence of the accumulated downtime of ext. scanning and def. scanning the longer the checkpoint interval gets. In Chapter 3, we have observed this convergence of the accumulated downtime when comparing the DLMs WP and scanning. We have argued that this is caused by the limited growth of the dirty set. The same argument is applicable for the convergence of the accumulated downtimes of ext. scanning and def. scanning.

The size of the accessed set is a concave function of the interval length as shown by Werner [47] and demonstrated theoretically by Denning [21]. Our plot of the size of the accessed set as shown in Figure 5.3 further supports this argument. As a result, the additional work for synchronization of access information decreases the longer the checkpoint interval gets, which in turn leads to a reduced

[3]For exact values we refer to Table C.2 in the appendix.

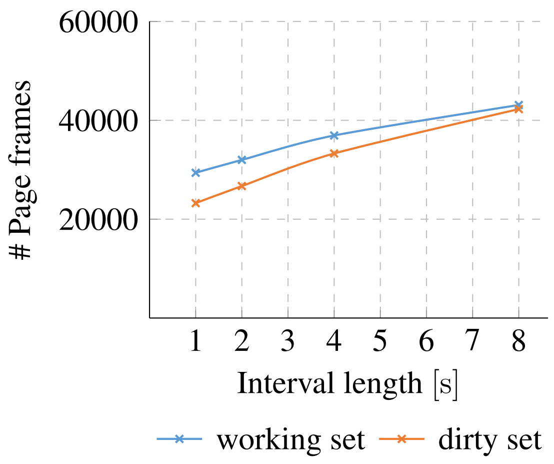


Figure 5.3: Comparing the working set and dirty set measurements for a Linux kernel build that we have collected during sparse checkpoint creation. The VM was configured with 2 GiB of main memory. The plotted values are the average of ten independent runs. See Table C.2 in the appendix for the exact values.

accumulated downtime for sparse checkpointing for longer checkpoint intervals.

Figure 5.4 plots the total runtime overhead minus the accumulated downtime for ext. scanning and def. scanning on the left axis and the runtime overhead in percent of the baseline (745.31 s) on the right axis. We notice that ext. scanning induces only a slightly higher runtime overhead compared to def. scanning. For an interval length of 1000 ms, sparse checkpointing induces an runtime overhead of 9.25 %, compared to def. scanning that induces a runtime overhead of 8.41 %.

The slightly higher runtime overhead of ext. scanning compared to def. scanning for interval lengths of 1000 ms and 2000 ms can be explained by the page frames whose accessed state is not tracked by the accessed and dirty bits of the EPTs as we have discussed in Section 4.1.1. In these special cases, the main memory access takes place in the execution context of the host, during the execution of the host, thus KVM marks the page frames dirty or accessed by setting the respective bit in the dirty or accessed bitmap directly.

In case of ext. scanning this additionally has to be done for page frames that have only been read leading to slightly higher runtime overhead of ext. scanning compared to def. scanning. The effect on the runtime diminishes the longer the checkpoint interval gets since the working set size and the dirty set size converge, as depicted in Figure 5.3, resulting in less additional read accesses that must be synchronized.

Our measurements of the downtime for a kernel build, a workload that induces low to medium main memory load, already shows almost double as high average downtimes for ext. scanning as for def. scanning.

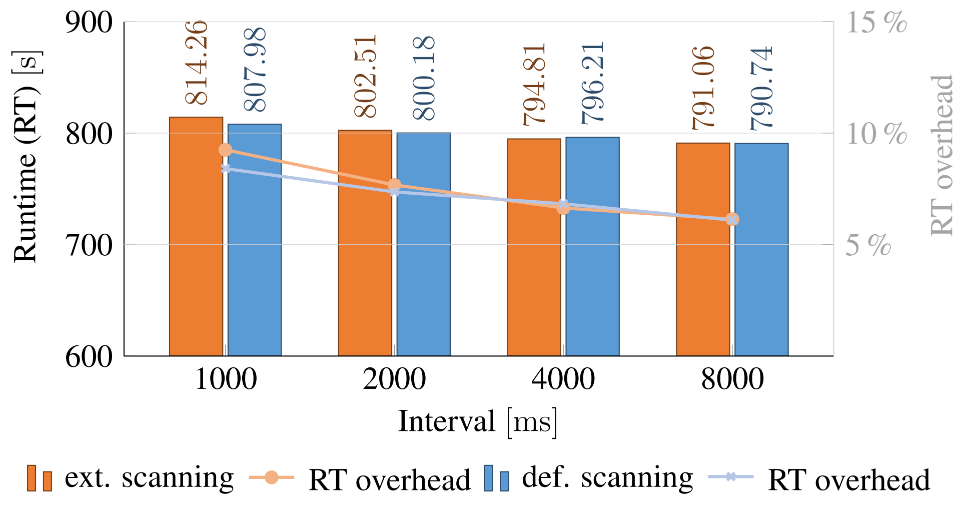


Figure 5.4: The left axis plots the runtime (RT) of a VM executing a Linux kernel build. The runtime is plotted minus the accumulated downtime caused by checkpointing. This means the figure compares the raw runtime overheads produced by extended scanning (ext. scanning) and default scanning (def. scanning). The right axis shows the runtime overhead of the checkpointed VM compared to the non-checkpointed VM in percent. The runtime of a VM without continuous checkpointing totals to 745.31 s.

As a result, we have employed SPECjbb as a CPU and memory intensive workload in order to show the performance of ext. scanning under high load. Figures 5.5 and 5.6 show the downtime for each checkpoint over the course of execution for varying checkpoint interval lengths for SPECjbb and a kernel build. We plot ext. scanning and def. scanning so we are able to compare both mechanisms and visualize the downtime overhead that is induced by ext. scanning.

The downtime plots for each checkpoint confirm our observations of the average downtimes of ext. scanning and def. scanning. Ext. scanning induces constantly more downtime of approximately 8 ms than def. scanning. This is true for a kernel build as well as SPECjbb.

If we compare the respective figures for a Linux kernel build and SPECjbb we see the different characteristics of the two workloads.

SpecJBB (see Figure 5.5) goes through multiple phases of execution each of which increases the load on the system. We are able to identify those phases for both variants of scanning since the downtime raises every 200 s.

A Linux kernel build (see Figure 5.6) on the other hand, induces high memory load at the beginning of the workload as it reads files from the virtual disk resulting in peaks of the downtime. For the rest of the workload, the memory load is evenly distributed leading to relatively constant downtimes.

The exact average downtimes can be found in Table C.3 in the appendix.

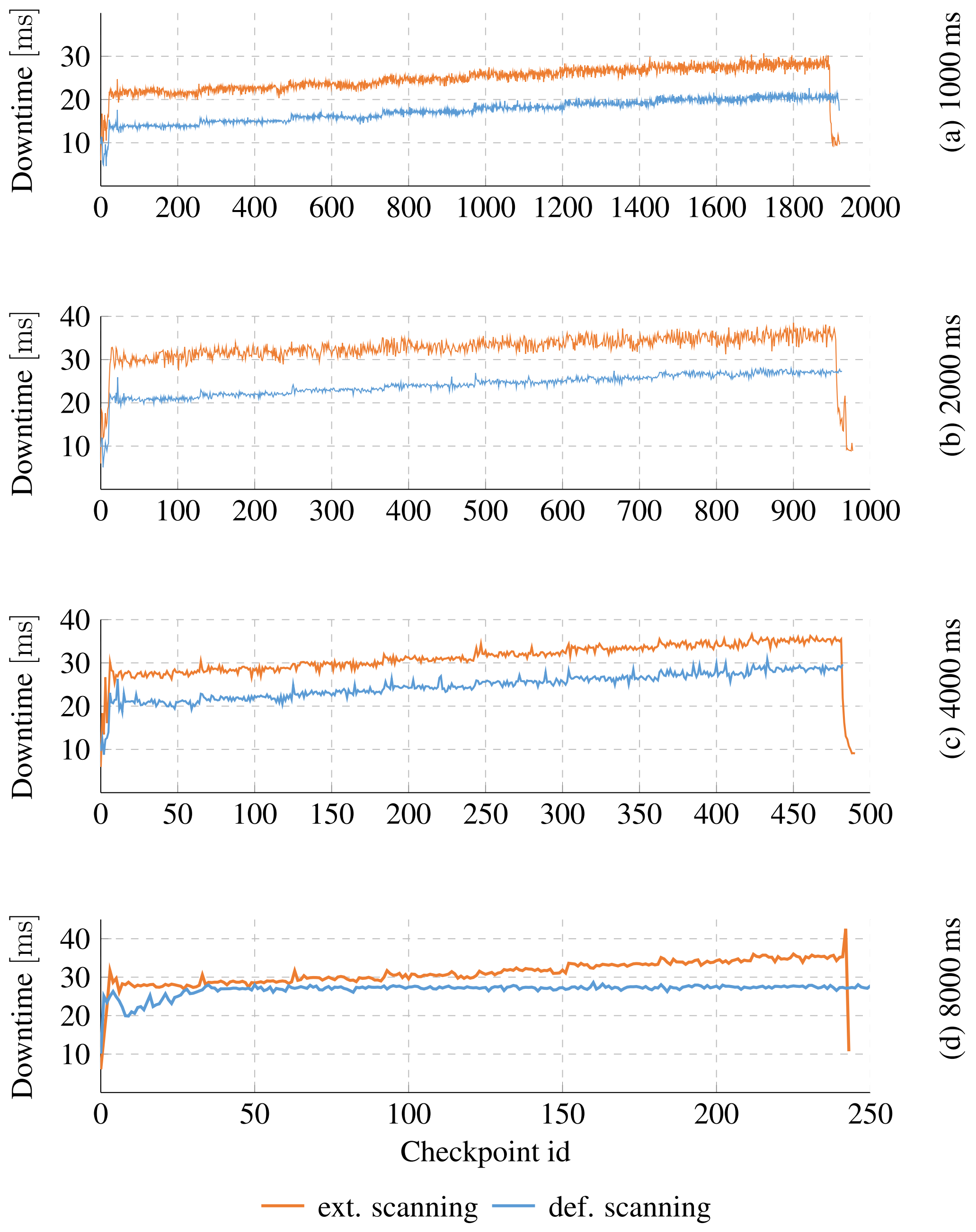


Figure 5.5: Plots (a), (b), (c), and (d) show the downtime for each individual checkpoint for interval lengths of 1000 ms, 2000 ms, 4000 ms, and 8000 ms. The workload executing on the test VM was a SPECjbb benchmark. We compare extended scanning (ext. scanning) with default scanning (def. scanning).

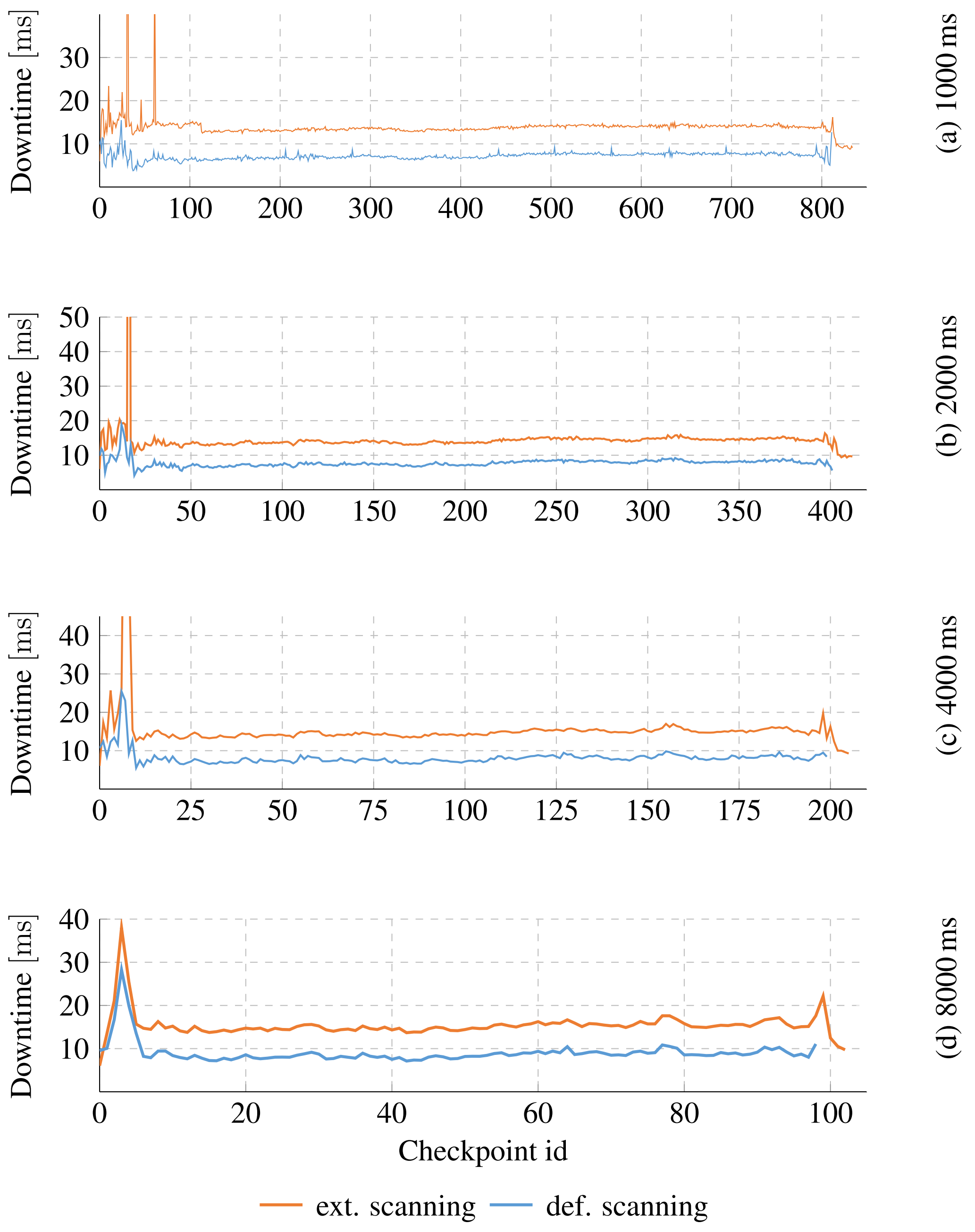


Figure 5.6: Plots (a), (b), (c), and (d) show the downtime for each individual checkpoint for interval lengths of 1000 ms, 2000 ms, 4000 ms, and 8000 ms. The workload executing on the test VM was a Linux kernel build. We compare extended scanning (ext. scanning) with default scanning (def. scanning). The scattered runaway values for an interval length of 2000 ms go up to at max 89 ms.

Conclusion

Our measurements of the sparse checkpoint creation performance regarding the induced downtime and the runtime overhead are coherent with the findings of the analysis regarding the general performance of scanning as a DLM. The direct comparison of ext. scanning and def. scanning illustrates that ext. scanning induces higher downtimes and a slightly higher runtime overhead.

These results are comprehensible since ext. scanning performs more work than def. scanning as ext. scanning not only synchronizes dirty information to the dirty bitmap but also access information to the accessed bitmap.

Our findings of Section 3.4 have shown that the influence of the downtime and the runtime overhead of the checkpoint mechanism are negligible for the speedup that is achievable with SimuBoost. Therefore, it is save to assume that the increased downtime does not diminish the benefits of sparse checkpointing regarding checkpoint loading times and a reduced memory footprint.

5.2.4 Sparse Checkpoint Loading

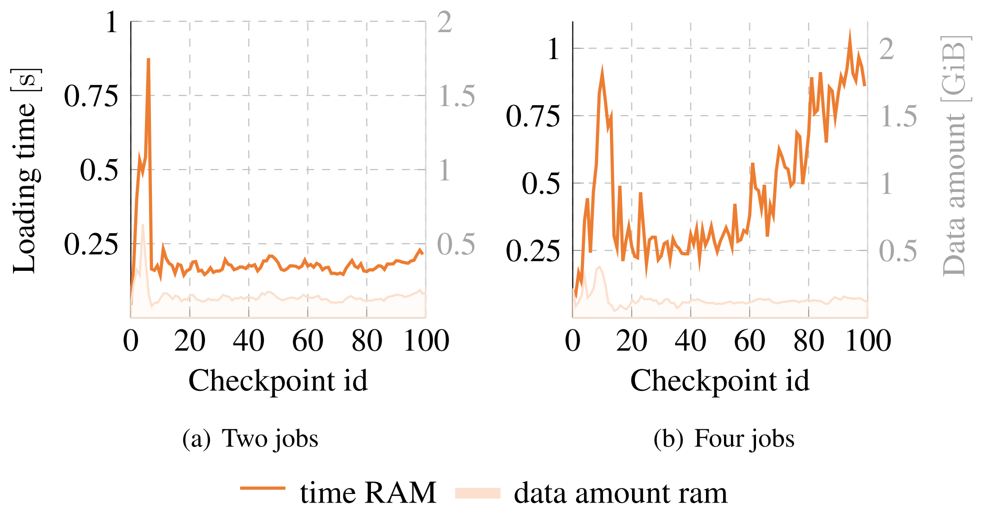
We have performed four experiments that recreate the checkpoint loading experiments of Chapter 3. We have omitted disk checkpoint loading since disk data it is completely contained in the replay log due to the recording of DMAs. Furthermore, we ignore the constant and low device loading times[4] for these experiments for the sake of simplicity. As a result, all figures show only the main memory loading time which represents the approximate total loading time of sparse checkpointing.

Figure 5.7 plots the average checkpoint loading times of a Linux kernel build for 100 consecutive checkpoints on the left axis and the restored data amount on the right, opaque axis. We have conducted the experiment for two and four jobs.

We notice that the checkpoint loading time peaks between checkpoints 0 and 20 with a value of about 1000 ms. In case of two jobs, the checkpoint loading time is henceforth below 250 ms, whereas it increases up to 1000 ms for four concurrent simulation jobs. The average checkpoint loading time for a kernel build is 193.111 ± 9.637 ms for two jobs, and 470.051 ± 24.738 ms for four jobs. The average data amount that is restored for a kernel build is 146 ± 6 MiB for two jobs, and 133 ± 5 MiB for four jobs.

If we compare these results with the results of the kernel build experiments that we have conducted in Section 3.2.2, we clearly see the improvement of sparse checkpointing on checkpoint loading times and the restored data amount. As a remainder, full checkpoints have resulted in average loading times of 2.4 ± 0.043 s for two jobs, and 7.189 ± 0.21 s for four jobs. The average data amount that must

[4]Device loading times are constantly about 5 ms.



(a) Two jobs

(b) Four jobs

Figure 5.7: Showing the average checkpoint loading times and the average checkpoint data amount out of ten runs for 100 consecutive sparse checkpoints of a Linux kernel build. The left axis plots the loading time in seconds, whereas the right axis plots the data amount that has been restored. The loading time for devices is not shown since it is about 5 ms in both cases. We completely omit disk restoring so there is no plot for disk either. As a result, the RAM loading time represents the total checkpoint loading time for sparse checkpoints.

be restored in case of full checkpoints is 2.857 ± 0.020 GiB for two jobs, and 2.779 ± 0.025 GiB for four jobs.

Therefore, the savings of sparse checkpointing regarding the data amount are about 95 %. This value is very close to the prediction[5] that we have provided at the end of Section 3.3.2. As a result, we save on average about 92 % of the checkpoint loading time for two jobs, and about 93 % of the checkpoint loading time for four jobs running concurrently on the test machine.

Nevertheless, we do realize that the average checkpoint loading time for four jobs is twice as high as for two jobs since the curve of the checkpoint loading time for four jobs raises up to 1000 ms, yet the data amount for four jobs is actually slightly lower than for two jobs. For full checkpoint loading in Section 3.2.2, we also observe a raise of the checkpoint loading time. We have argued that the host starts swapping page frames to disk due to the limited amount of main memory of the test machine and the high memory demands of executing four simulations in parallel.

For our sparse checkpoint loading experiments, we have also deployed full-

[5]We predicted a data reduction of 2 GiB to 134 MiB (93.5 %) ignoring the size of the disk checkpoint.

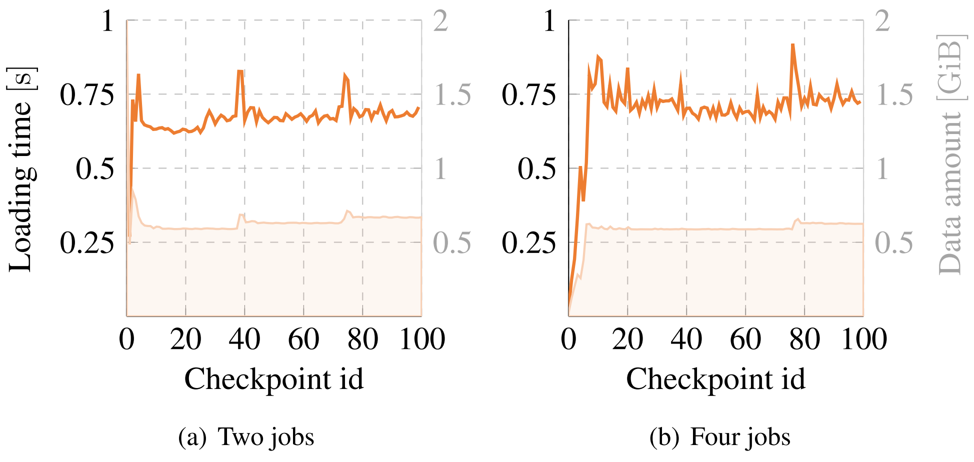


Figure 5.8: Showing the average loading times and the average data amount out of ten runs for 100 consecutive sparse checkpoints of SPECjbb. The left axis plots the loading time in seconds, whereas the right axis plots the data amount that has been restored. The loading time for devices is not shown since it is below 5 ms for both cases. We completely omit disk restoring so there is no plot for disk either. As a result, the RAM loading time represents the total checkpoint loading time for sparse checkpoints.

size QEMU instances to produce load on the test machine that is comparable to the load of a SimuBoost instance. As a result, in case of four jobs, three of those four jobs are default, full-blown QEMU instances, each of which taking at least 2 GiB of memory, whereas only the checkpoint loading QEMU instance is sparse and has a smaller main memory footprint. We argue that the memory footprint of the three load generating QEMU instances in case of a kernel build increases over time since they continuously perform I/O, claiming host resources, such as the host page cache. This leads to displacement of page frames containing checkpoint data, resulting in steadily increasing loading times as the host starts swapping page frames to disk. In case of two jobs, we do not observe this performance degradation for a kernel build due to high memory pressure, since there is only one load generating QEMU instance running.

Figure 5.8 shows the results of checkpoint loading experiments for SPECjbb. For SPECjbb we find average checkpoint loading times of 670.835 ± 5.721 ms for two jobs and 694.493 ± 12.681 ms for four jobs. The average data amount is 647.495 ± 14.501 MiB for two jobs and 576.88 ± 9.847 MiB for four jobs.

The results of the data amount represent the aforementioned higher memory load that is generated by SPECjbb. The average data amount of SPECjbb is four to five times as high as for a Linux kernel build. The same applies to the checkpoint loading times of a Linux kernel build and of SPECjbb for two jobs. Yet, in case of four jobs, we do not notice a raise of the checkpoint loading time for SPECjbb as we have for a Linux kernel build with four concurrent jobs even though there is more data to restore for SPECjbb.

This goes with our explanation of the increasing checkpoint loading time for four jobs for a Linux kernel build. We argue that for a Linux kernel build the load generating simulations claim additional host resources since a Linux kernel build causes I/O activity by reading and writing data from and to disk. This I/O activity passes through to the host I/O subsystem resulting in higher memory pressure on the host. SPECjbb in contrast, causes very little I/O activity, thus there is less I/O activity on the host and there is less performance degradation for four concurrent simulation jobs. As a result, we observe that the checkpoint loading times of SPECjbb for four jobs do not increase over time.

The improvement of the sparse checkpoint loading times for SPECjbb compared to full checkpoint loading times are not as significant as for a kernel build since the disk loading time does not weigh as much and more RAM data is restored for SPECjbb. Nevertheless, for two jobs, we are able to decrease the average checkpoint loading time by 49.6 % from 1.331 ± 0.01 s for full checkpoints to 670.835 ± 5.721 ms for sparse checkpoints. For four jobs, we decrease the average checkpoint loading time by 74 % from 2.672 ± 0.045 s for full checkpoints to 694.493 ± 12.681 for sparse checkpoints. The data amount for SPECjbb for two and four jobs is decreased by about 70 %.

Conclusion

The bottom line of our sparse checkpoint loading experiments has been to demonstrate its performance advantage compared to full checkpoint loading even under the premise of high utilization of the test machine.

Sparse checkpointing is able to reduce the average data amount of a checkpoint by 49.6 % to 93 % of the size of a full checkpoint depending on the deployed workload. As a result, the average checkpoint loading time in our experiments drops by up to 74 % for SPECjbb and up to 95 % for a kernel build.

5.2.5 Parallel Simulation

Finally, we demonstrate the achievable speedup of SimuBoost using sparse checkpointing. The experiment enables us to provide a sparse checkpoint correctness rate for SPECjbb. Therefore, we utilize the measurements of the average maximal main memory footprint of a Linux Kernel build to approximate the maximal number of concurrent simulations on our test machine.

Our measurements in Section 5.2.1 show an average maximal main memory footprint of 0.971 ± 0.012 GiB for a kernel build. We assume that there are 40 GiB of main memory available for the execution of simulations on our test machine, the remaining main memory is used by SimuBoost itself and the host OS. We use a conservative maximal number of simulations of $N = 32$ since the CPU of the test machine only comes with 16 cores and hyperthreading, thus 32 hardware threads.

We perform replay experiments of SPECjbb in order to obtain the sparse checkpoint correctness rate for SPECjbb just as in Section 5.2.1. We use Formula 3.11 to determine the interval length. For the logging slowdown $s_{log} = 1.076$, we use the def. scanning values in Table A.4 since the slowdown of ext. scanning is approximately the same[6]. We set the checkpoint loading time $t_i = 0.69$ s to the worst-case average value that we have measured during the SPECjbb checkpoint loading experiments in the previous section. The runtime of the producer VM is $T_{vm} = 1939.376$ s. We obtain an interval length of 1230 ms. Furthermore, we set the downtime $t_c = 0.02473$ s (Appendix Table C.3) which is the downtime that we have measured for sparse checkpointing for an interval length of 1000 ms.

Our concurrent replay has shown a correctness rate of 96.05 % for SPECjbb which confirms the correctness rate of sparse checkpointing for a Linux kernel build and supports our claim that there are still some unresolved issues with our sparse checkpointing implementation.

We use statistics regarding the replay time of each simulation interval to estimate the simulation slowdown s_{sim}. We derive a value of about $s_{sim} = 39$ for SPECjbb. As a result, we get a predicted speedup for $N = 32$ of $30.99\times$ by applying Formula 3.9.

In comparison, our concurrent sparse checkpoint correctness experiment, which utilizes a worker pool for concurrent execution of simulation intervals, achieves a speedup of $19.06\times$. This speedup value corresponds to an efficiency of 61.5 % which is relatively low.

We are not able to collect memory footprint data for concurrent simulation due to technical restrictions of our measurement method. Nevertheless, we have executed the last 100 simulation intervals of SPECjbb sequentially to obtain at least a rough estimation of the average maximal main memory footprint of a SPECjbb simulation interval. We chose the last 100 simulation intervals since SPECjbb generates the most main memory load at the end of its execution time.

We find an average maximal main memory footprint of 0.975 ± 0.027 GiB for a single simulation. This implies an estimated maximal main memory consumption of about 31.2 GiB for 32 concurrent simulations leaving us with 32.8 GiB of main

[6]We always round up to the next higher interval length for which we provide a value in the table.

memory on the test machine that is free for use by SimuBoost and the host system. This value matches our approximation of SPECjbb's main memory footprint using the data acquired for a Linux kernel build. As a result, we rule out high memory pressure as a reason for the bad speedup efficiency.

This leaves us with two possible bottlenecks that may degrade the performance of the concurrent simulation. First, the test machine is equipped with an Intel(R) Xeon(R) CPU that comes with 16 cores and hyperthreading resulting in an effective number of parallel execution threads of 32. Yet, Wallace et al. [46] show that the performance of hyperthreading does not compare to the performance of a real core. As a result, the simulations alone overload the CPU. SimuBoost's backend adds further load to the CPU possibly degrading the system performance. As a result, we should have used a maximal number of $N = 16$ simulations for this experiment.

Another possible bottleneck is the I/O system of the machine since 32 simulations repeatedly, and in parts concurrently, retrieve considerable amounts of data from disk. This claim is supported by the bad average sparse checkpoint loading times of 3.625 ± 0.0484 s. Keep in mind that even the sparse checkpoint loading experiments of SPECjbb in the previous section, in which case we have completely utilized the test machines main memory, have not resulted in such high sparse checkpoint loading times.

Our observations regarding the average loading time and the speedup efficiency in case of concurrent simulation of SPECjbb mitigate the assumption, which we have made in Section 3.1.1, that the main memory is the primary limiting factor for concurrent simulations. It shows that we must further study the performance implications of SimuBoost in a single machine and a distributed setup in order to maximize its efficiency. Furthermore, we must carefully choose the number of simulations in order to achieve an optimal speedup.

Chapter 6

Conclusion

In this work, we have developed an optimization of continuous checkpoints for the application in SimuBoost – sparse checkpointing. For this purpose, we have performed an extensive analysis of the formal basis [23, 40] of SimuBoost and derived a number of abstract requirements.

We have further investigated the implications of those requirements for the overall performance of SimuBoost by analyzing the existing checkpoint mechanisms of SimuBoost regarding checkpoint creation and checkpoint loading.

We have found that the slowdown of the checkpoint mechanism and the induced downtime have only a minor effect on the speedup of SimuBoost. Yet, we found that it takes 2 s to 3 s to restore a checkpoint on a machine which is under low to medium load and up to 9 s to restore a checkpoint on a machine under heavy load. Furthermore, we have argued that the theoretical models of Rittinghaus et al. [40] and Eicher [23] imply the importance of the number of concurrent simulation jobs for the achievable speedup of SimuBoost. We have identified the main memory consumption of a single simulation to be an important limiting factor for the number of concurrent simulations jobs on a machine.

We have decided to reduce the data amount that needs to be restored during checkpoint loading since it directly influences the checkpoint loading time. Therefore, we have recapitulated the work of Werner [47] on the working set size and composition of continuous checkpoints. Our discussion of his work lead to the approach of sparse checkpointing in order to reduce the data amount during checkpoint loading.

Sparse checkpointing leverages access information in order to only restore those page frames that are in the working set of the checkpoint that is to be restored. Besides, we have noticed the possibility that sparse checkpointing further reduces the main memory footprint of each simulation allowing for more concurrent simulations per machine.

We have implemented sparse checkpointing for KVM leveraging the access information of the EPTs.

The evaluation shows that there are still issues with our implementation of sparse checkpointing, since only 97 % of sparse checkpoints of a Linux kernel build replay correctly. Nevertheless, sparse checkpointing proofs to reduce the average checkpoint loading time by 89 % for a Linux kernel build thus supporting its effectiveness and relevance for SimuBoost. Furthermore, we have shown that sparse checkpointing decreases the performance degradation of checkpoint loading under medium to high utilization of the test machine by 74 % to 93 % depending on the workload.

Our evaluation of sparse checkpoint creation shows a slightly higher runtime overhead for sparse checkpointing compared to full checkpointing. Furthermore, sparse checkpointing induces an average downtime that is almost double as high as the average downtime of full checkpointing. This is due to the additional work of synchronizing access information to the corresponding data structures. Nevertheless, we have shown the minor effect of the checkpoint creation performance on the achievable speedup of SimuBoost in Chapter 3. As a result, we argue that the performance degradation of sparse checkpoint creation compared to full checkpointing is diminished by the performance benefits regarding checkpoint loading.

Finally, sparse checkpointing reduces the average maximal main memory footprint of a simulation from 4.5 GiB to 971 MiB for a Linux kernel build. This allows us to deploy three to four times more concurrent simulations on a single machine. The fact that SimuBoost with sparse checkpointing can be effectively deployed on a single machine facilitates its application in research and increases its practical use since there is no longer the need to setup a cluster of multiple machines to run an accelerated full system simulation using SimuBoost. Yet, our concurrent simulation experiment of SPECjbb has shown that the main memory footprint is not the only limiting factor. The speedup efficiency of SimuBoost suffers from too many simulations on a single machine even if the main memory footprint of the simulations is no longer the limiting factor.

6.0.1 Future Work

Our evaluation in Chapter 5 has shown some drawbacks of our sparse checkpoint implementation that can be tackled in future work. First of all, we have noticed that there are sparse checkpoints that do not replay correctly. This indicates that the sparse checkpoint implementation is incomplete and there exist corner cases in which we miss memory accesses by the guest.

Sparse checkpointing induces higher downtimes than full checkpointing. Future work could aim to reduce the downtime by introducing *pre-scanning*. Prescanning starts with the EPT walk and the synchronization of accessed and dirty information while the VM is still running. As a result, there are less set dirty and accessed bits in the EPTs left to synchronize during the downtime of the VM.

Finally, in our concurrent simulation experiment, we have observed an unsatisfactory efficiency of SimuBoost even though the main memory footprint of the simulations has not been the limiting factor anymore. This is because we have accounted for hyperthreading as if it provides full cores. Therefore, the concurrent simulation suffered from performance degradation due to overutilization of the CPU. As a result, we propose further investigation of the performance implications of SimuBoost running on a single machine and distributed on a cluster.

Appendices

Appendix A

Analysis

A.1 Different Checkpoint Mechanisms in Comparison

A.1.1 Checkpoint Creation

Interval [ms]	DLM	Avg. downtime [ms]	Std. deviation	Std. error
1,000	wp	3.62	0.81	0.26
2,000	wp	3.85	0.77	0.24
4,000	wp	4.03	0.99	0.31
8,000	wp	4.36	1.21	0.38
1,000	scan	7.18	1.18	0.37
2,000	scan	7.77	1.33	0.42
4,000	scan	8.13	2.09	0.66
8,000	scan	9.04	2.7	0.85

Table A.1: Showing the average downtimes of a Linux kernel build executing in a VM configured with 2 GiB of main memory and being checkpointed continuously using varying interval lengths. We have used write protection (WP) and scanning as dirty logging mechanisms (DLMs).

Interval [ms]	DLM	Avg. runtime excl. downtime [s]	Std. deviation	Std. error
1,000	wp	849.75	4.48	1.42
2,000	wp	814.34	2.04	0.65
4,000	wp	803.14	2.93	0.93
8,000	wp	797.21	3.64	1.15
1,000	scan	807.98	2.29	0.72
2,000	scan	800.18	2.62	0.83
4,000	scan	796.21	3.78	1.2
8,000	scan	790.74	0.52	0.17

Interval [ms]	DLM	Avg. runtime incl. downtime [s]	Std. deviation	Std. error
1,000	wp	852.82	4.48	1.42
2,000	wp	815.91	2.05	0.65
4,000	wp	803.95	2.94	0.93
8,000	wp	797.65	3.63	1.15
1,000	scan	813.77	2.33	0.74
2,000	scan	803.29	2.61	0.82
4,000	scan	797.82	3.8	1.2
8,000	scan	791.63	0.51	0.16

Interval [ms]	DLM	Avg. acc. downtime [s]	Std. deviation	Std. error
1,000	wp	3.07	$3.38 \cdot 10^{-2}$	$1.07 \cdot 10^{-2}$
2,000	wp	1.57	$1.38 \cdot 10^{-2}$	$4.38 \cdot 10^{-3}$
4,000	wp	0.81	$1.27 \cdot 10^{-2}$	$4.01 \cdot 10^{-3}$
8,000	wp	0.43	$4.15 \cdot 10^{-2}$	$1.31 \cdot 10^{-2}$
1,000	scan	5.79	$4.62 \cdot 10^{-2}$	$1.46 \cdot 10^{-2}$
2,000	scan	3.1	$5.29 \cdot 10^{-2}$	$1.67 \cdot 10^{-2}$
4,000	scan	1.62	$2.68 \cdot 10^{-2}$	$8.48 \cdot 10^{-3}$
8,000	scan	0.89	$1.62 \cdot 10^{-2}$	$5.12 \cdot 10^{-3}$

Table A.2: Showing the average runtime of a Linux kernel build executing in a VM configured with 2 GiB of main memory and being checkpointed continuously using varying interval lengths. We employed write protection (WP) and scanning as dirty logging mechanisms (DLMs). The upper table shows the runtime exclusive the accumulated downtime that is induced by the copy mechanism of checkpointing. The table in the middle shows the runtime inclusive the accumulated downtime. The difference of the runtime depicted in both tables is the accumulated downtime listed in the lower table.

Interval [ms]	DLM	Avg. downtime [s]	Std. deviation	Std. error
1,000	wp	5.79	0.65	0.21
2,000	wp	7.78	0.59	0.19
4,000	wp	8.28	0.77	0.24
1,000	scan	17.36	2.81	0.89
2,000	scan	24.07	2.92	0.92
4,000	scan	24.68	3.49	1.1
8,000	scan	24.08	3.93	1.24

Table A.3: Showing the average downtime of SPECjbb executing in a VM configured with 2 GiB of main memory and being checkpointed continuously using varying interval lengths. We employed write protection (WP) and scanning as dirty logging mechanisms (DLMs).

Interval [ms]	Slowdown factor
1,000	1.117
2,000	1.096
3,000	1.084
4,000	1.074
5,000	1.072
6,000	1.071
7,000	1.077
8,000	1.067
9,000	1.066
10,000	1.064
11,000	1.063
12,000	1.065
13,000	1.065
14,000	1.063
15,000	1.062
16,000	1.062
17,000	1.059
18,000	1.069
19,000	1.063
20,000	1.069

Interval [ms]	Slowdown factor
1,000	1.087
2,000	1.076
3,000	1.071
4,000	1.069
5,000	1.075
6,000	1.063
7,000	1.071
8,000	1.063
9,000	1.062
10,000	1.06
11,000	1.061
12,000	1.06
13,000	1.059
14,000	1.059
15,000	1.06
16,000	1.058
17,000	1.057
18,000	1.068
19,000	1.058
20,000	1.069

Table A.4: Showing the slowdown factor of the both dirty logging mechanisms of SimuBoost, write protection (WP) and scanning, for increasing interval lengths. The left table shows the slowdown factors of WP, whereas the right table shows the slowdown factors of scanning. The depicted slowdown factors per interval length have been taken by performing a single run thus the runaway values of scanning for e.g., an interval length of 20 s.

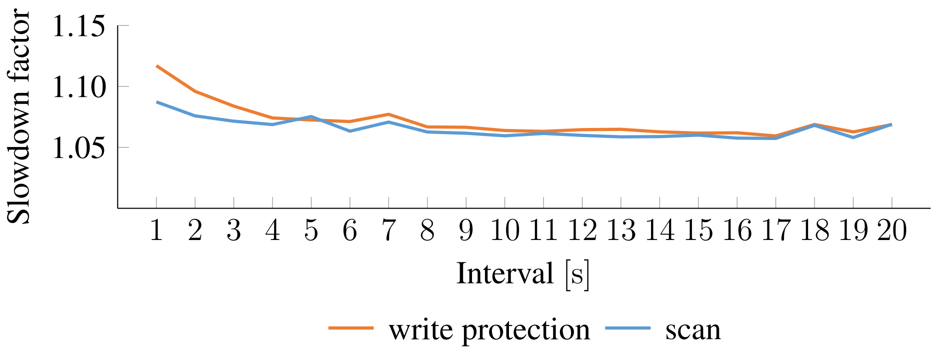


Figure A.1: Plotting the slowdown factor of both dirty logging mechanisms for checkpoint interval lengths ranging from 1 s to 20 s. The corresponding raw data is illustrated in Table A.4.

A.1.2 Checkpoint Loading Times

Workload	# jobs	Avg. total loading time [s]	Std. deviation	Std. error
kb	2	2.4	0.43	$4.29 \cdot 10^{-2}$
kb	4	7.19	2.1	0.21
specJBB	2	1.33	$9.89 \cdot 10^{-2}$	$9.89 \cdot 10^{-3}$
specJBB	4	2.67	0.45	$4.52 \cdot 10^{-2}$

Workload	# jobs	Avg. ram loading time [s]	Std. deviation	Std. error
kb	2	0.88	0.14	$1.44 \cdot 10^{-2}$
kb	4	3.62	0.9	$8.96 \cdot 10^{-2}$
specJBB	2	1.06	$5.55 \cdot 10^{-2}$	$5.55 \cdot 10^{-3}$
specJBB	4	2.28	0.37	$3.72 \cdot 10^{-2}$

Workload	# jobs	Avg. disk loading time [s]	Std. deviation	Std. error
kb	2	1.52	0.37	$3.68 \cdot 10^{-2}$
kb	4	3.56	1.23	0.12
specJBB	2	0.26	$6.47 \cdot 10^{-2}$	$6.47 \cdot 10^{-3}$
specJBB	4	0.39	0.11	$1.07 \cdot 10^{-2}$

Table A.5: Showing the average checkpoint loading time of 100 consecutive checkpoints. The test machine was executing two or four jobs, respectively. The used workloads running inside the restored VM and on the load generating VMs was a Linux kernel build (kb) or SPECjbb. The shown loading times are the average of ten independent runs.

A.2 Sparse Checkpointing

A.2.1 Working Set Analysis

Interval length [ms]		500	1000	2000	4000	8000
Working Set	abs.	395977	395584	395676	395818	395629
	rel.	71.71	75.45	75.47	75.50	75.46
Read Working Set	abs.	347099	394469	394545	394702	394475
	rel.	66.20	75.24	75.25	75.28	75.24
Write Working Set	abs.	216467	392478	395268	395303	394967
	rel.	41.29	74.86	75.39	75.40	75.33
Excl. Read Working Set	abs.	159510	3105	407	515	661
	rel.	40.48	0.78	0.1	0.13	0.17

Table A.6: Showing the working set sizes for stress as measured by Werner [47]. The absolute value describes the number of pages frames included in the working set. The relative value is in percentage of the test VM's main memory which was 2 GiB. The relative value of the exclusive read working set is in percentage of the complete working set.

Appendix B

Design and Implementation

B.1 Acquiring Working Set Information

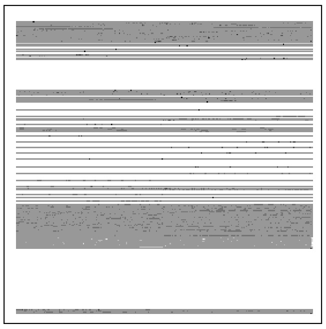
B.1.1 Synchronization of Accessed Information

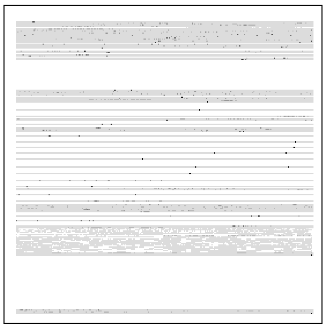
File	Function
arch/x86/kvm/mmu.c	fast_pf_fix_direct_spte
virt/kvm/kvm_main.c	__kvm_write_guest_page
virt/kvm/kvm_main.c	kvm_write _guest_cached

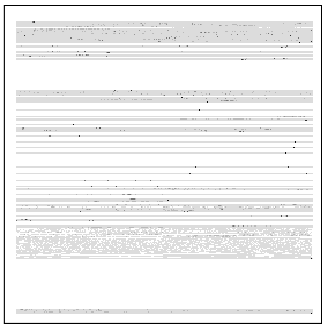
Table B.1: Listing the locations in the source code of the Linux kernel that we have modified to optimize our sparse checkpointing implementation.

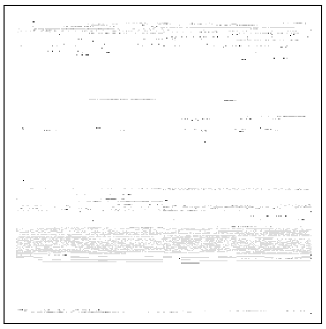
B.2 Loading Sparse Checkpoints

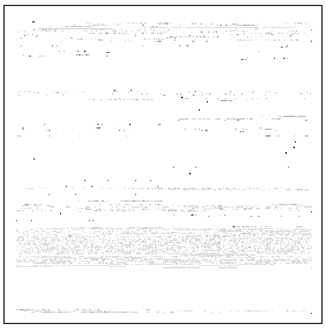
Showing heatmaps of the memory accesses of a Linux kernel build and SPECjbb for various checkpoint intervals. The VM was equipped with 2 GiB of main memory. For the sake of convenience, each cell of the heatmap represents a cluster of eight page frames with consecutive addresses. For each heatmap the upper left corner represents the lowest page frame number and the bottom right corner the highest page frame number. The presented checkpoint intervals were picked randomly.

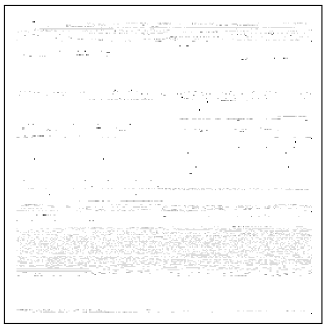

(a) *Checkpoint interval 59*

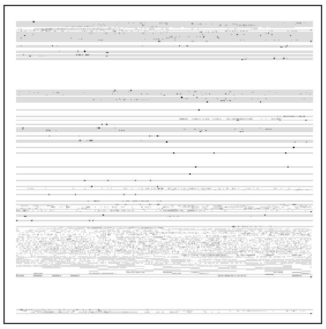

(b) *Checkpoint interval 107*

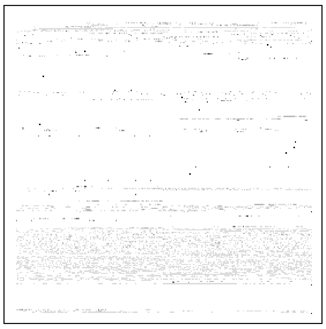

(c) *Checkpoint interval 221*


(d) *Checkpoint interval 236*


(e) *Checkpoint interval 456*


(f) *Checkpoint interval 558*


(g) *Checkpoint interval 558*


(h) *Checkpoint interval 748*

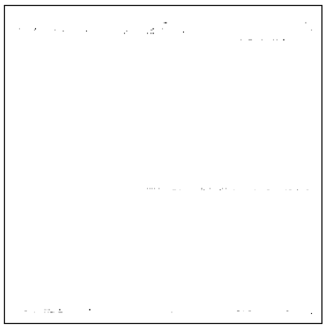
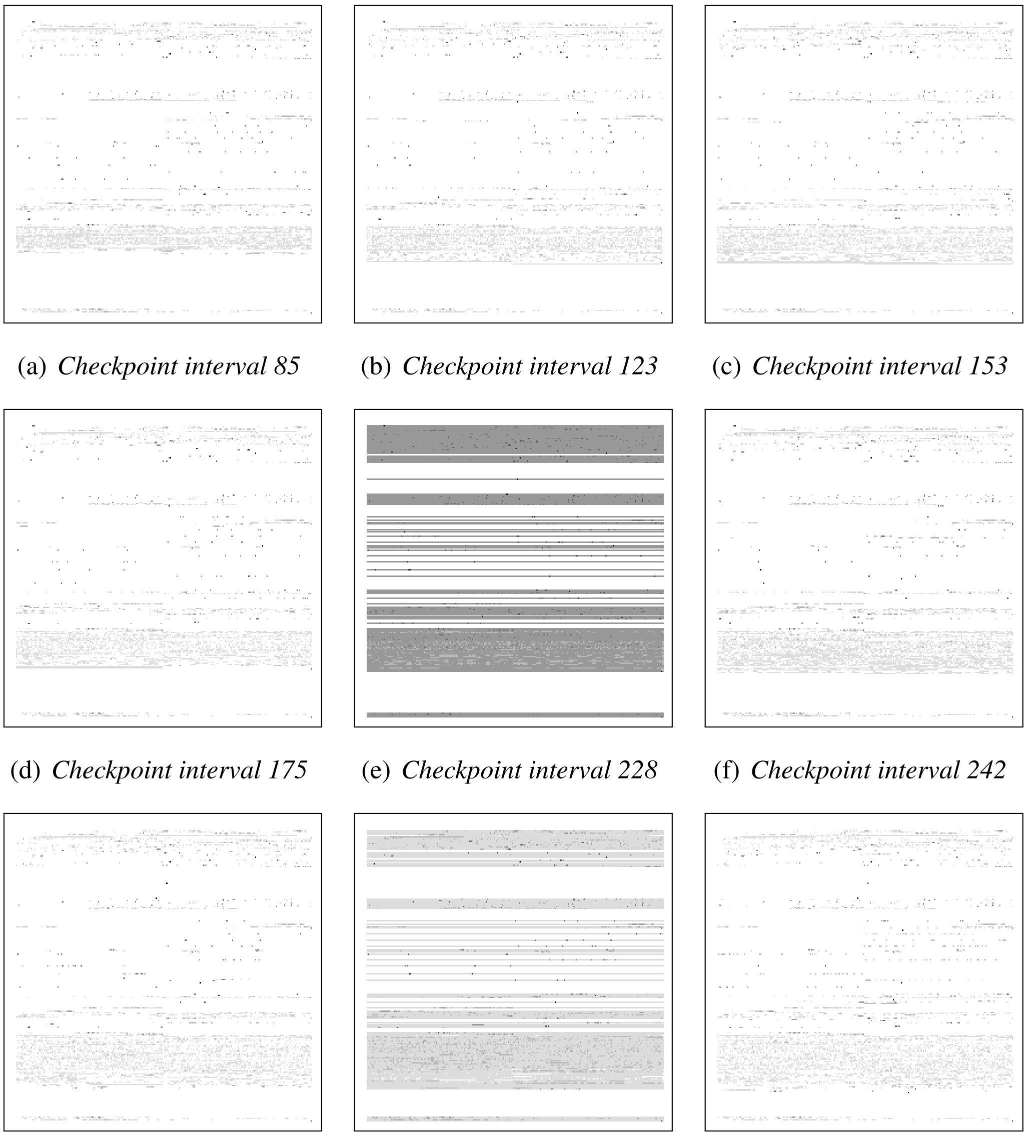

(i) *Checkpoint interval 817*

Figure B.1: Access patterns of a Linux kernel build with a checkpoint interval length of 1000 ms.

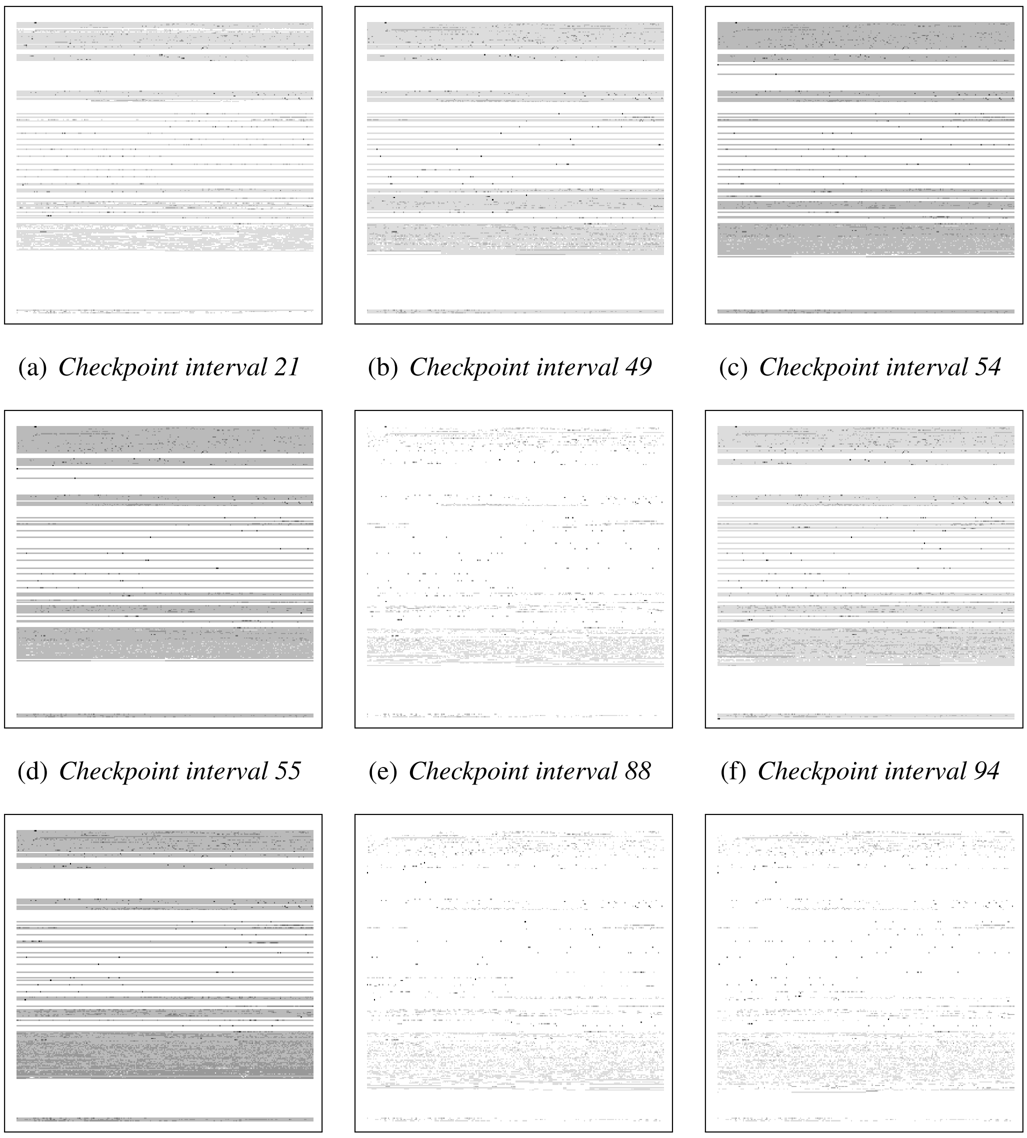


(a) *Checkpoint interval 85* (b) *Checkpoint interval 123* (c) *Checkpoint interval 153*

(d) *Checkpoint interval 175* (e) *Checkpoint interval 228* (f) *Checkpoint interval 242*

(g) *Checkpoint interval 334* (h) *Checkpoint interval 365* (i) *Checkpoint interval 387*

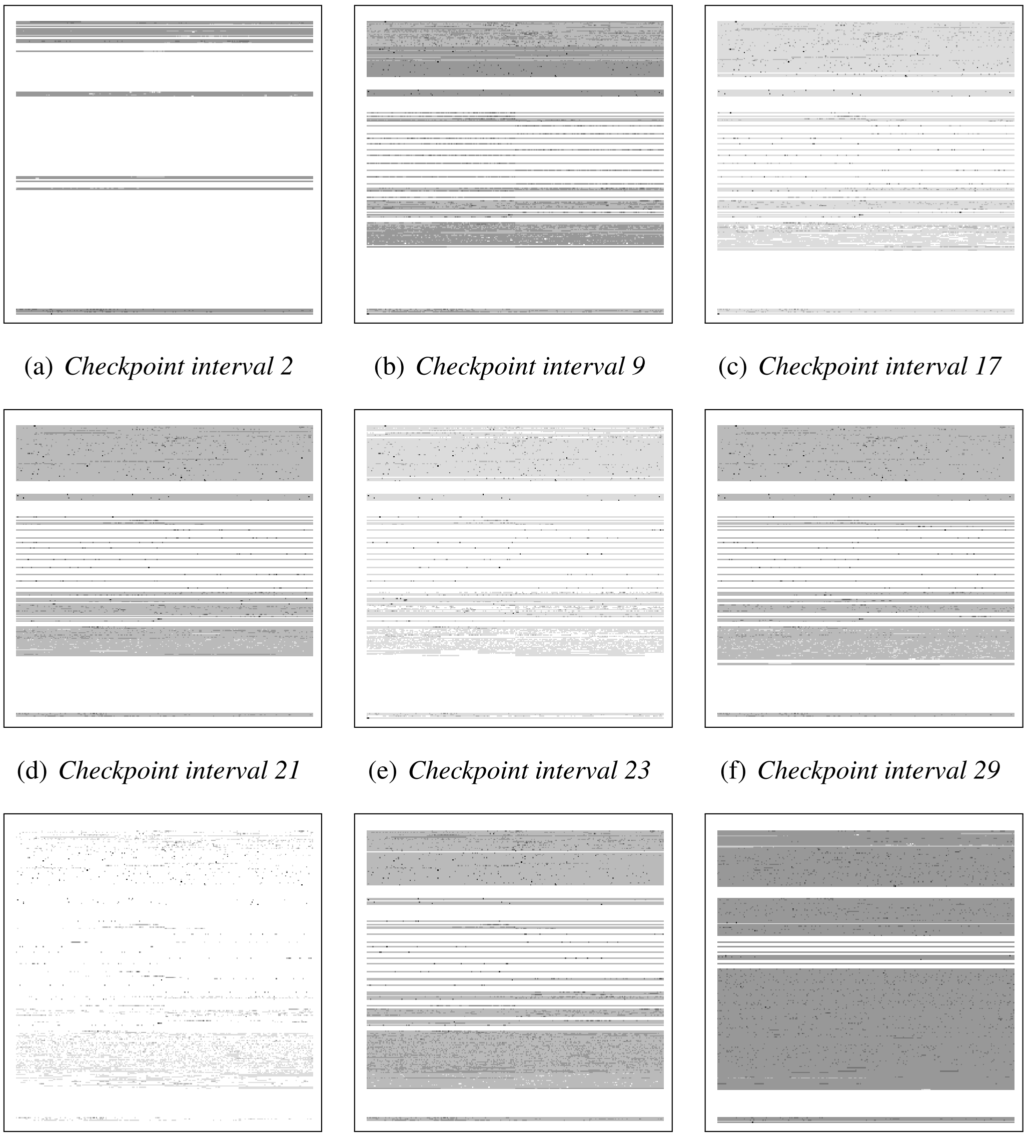
Figure B.2: Access patterns of a Linux kernel build with a checkpoint interval length of 2000 ms.


(a) *Checkpoint interval 21* (b) *Checkpoint interval 49* (c) *Checkpoint interval 54*

(d) *Checkpoint interval 55* (e) *Checkpoint interval 88* (f) *Checkpoint interval 94*

(g) *Checkpoint interval 127* (h) *Checkpoint interval 175* (i) *Checkpoint interval 190*

Figure B.3: Access patterns of a Linux kernel build with a checkpoint interval length of 4000 ms.

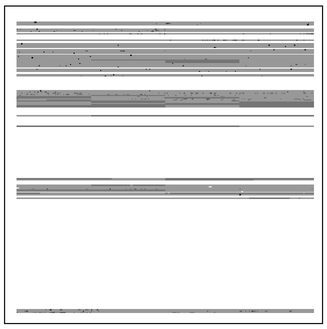


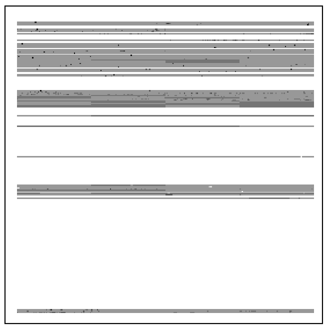
(a) *Checkpoint interval 2* (b) *Checkpoint interval 9* (c) *Checkpoint interval 17*

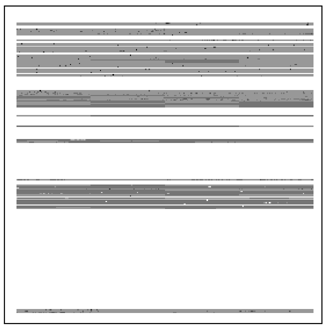
(d) *Checkpoint interval 21* (e) *Checkpoint interval 23* (f) *Checkpoint interval 29*

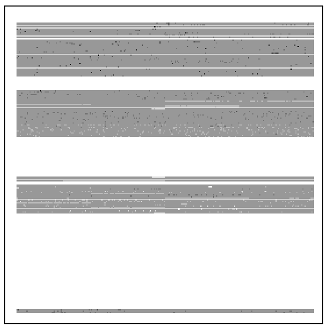
(g) *Checkpoint interval 81* (h) *Checkpoint interval 90* (i) *Checkpoint interval 99*

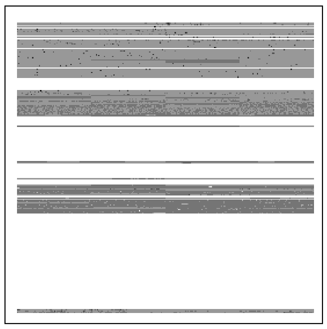
Figure B.4: Access patterns of a Linux kernel build with a checkpoint interval length of 8000 ms.

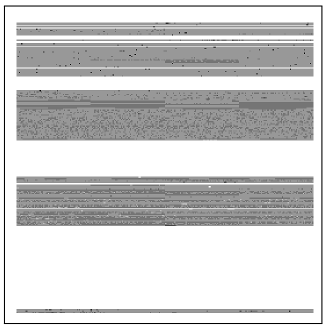

(a) *Checkpoint interval 100*

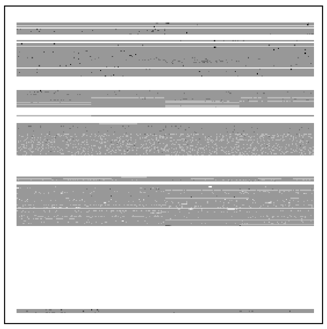

(b) *Checkpoint interval 175*

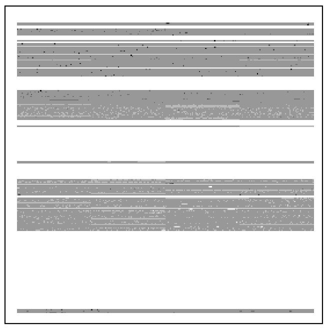

(c) *Checkpoint interval 698*


(d) *Checkpoint interval 864*


(e) *Checkpoint interval 926*


(f) *Checkpoint interval 1229*


(g) *Checkpoint interval 1363*


(h) *Checkpoint interval 1545*

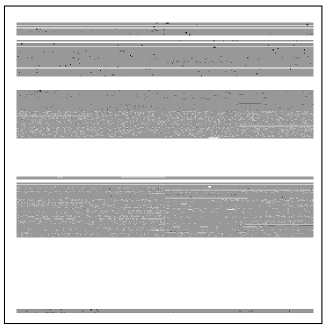
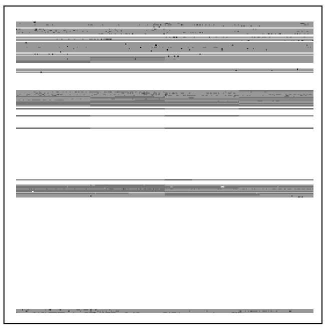
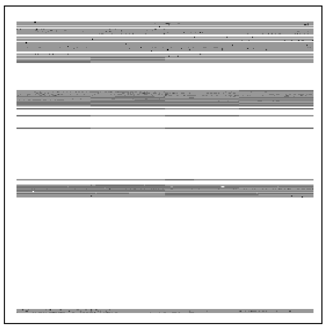
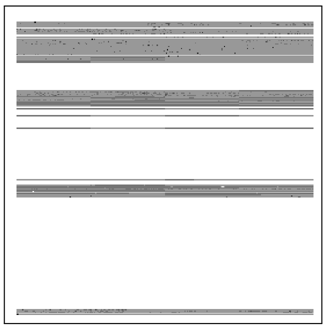
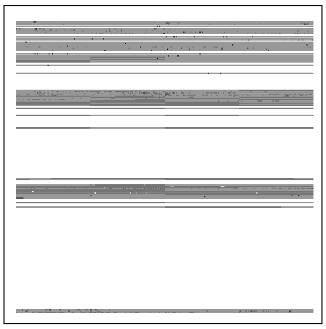

(i) *Checkpoint interval 1859*

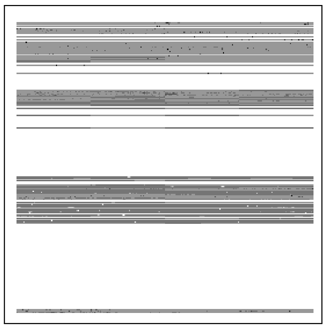
Figure B.5: Access patterns of SPECjbb with a checkpoint interval length of 1000 ms.

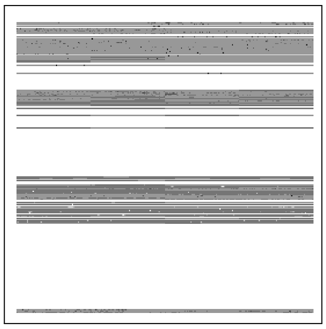

(a) *Checkpoint interval 61*

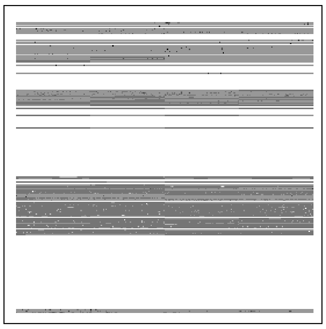

(b) *Checkpoint interval 89*

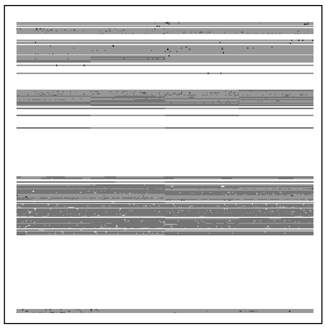

(c) *Checkpoint interval 123*


(d) *Checkpoint interval 216*


(e) *Checkpoint interval 524*


(f) *Checkpoint interval 540*


(g) *Checkpoint interval 743*


(h) *Checkpoint interval 833*

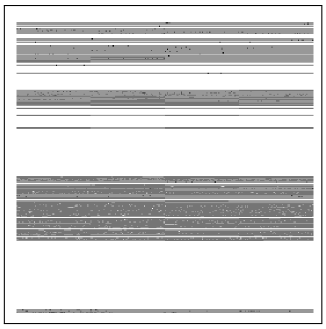
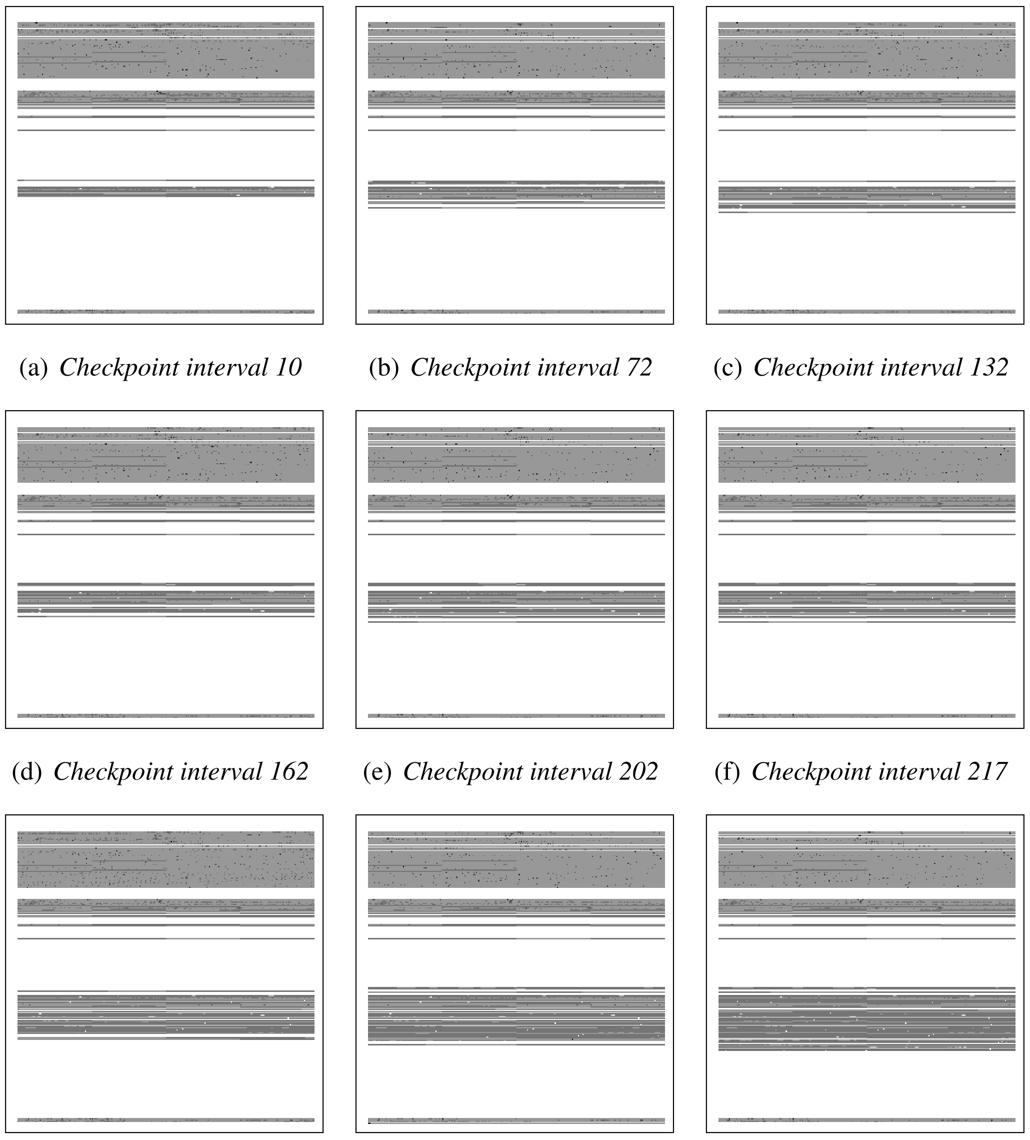

(i) *Checkpoint interval 893*

Figure B.6: Access patterns of SPECjbb with a checkpoint interval length of 2000 ms.

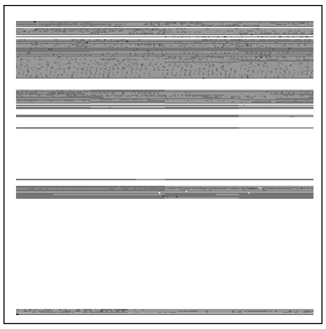


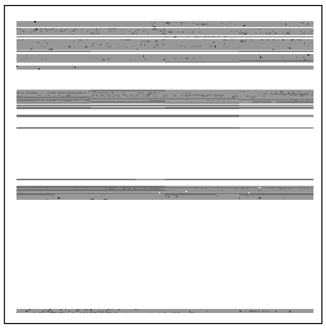
(a) *Checkpoint interval 10* (b) *Checkpoint interval 72* (c) *Checkpoint interval 132*

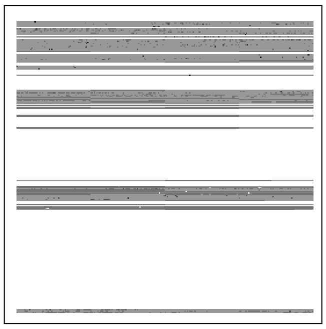
(d) *Checkpoint interval 162* (e) *Checkpoint interval 202* (f) *Checkpoint interval 217*

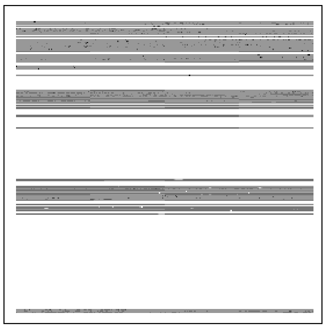
(g) *Checkpoint interval 354* (h) *Checkpoint interval 366* (i) *Checkpoint interval 466*

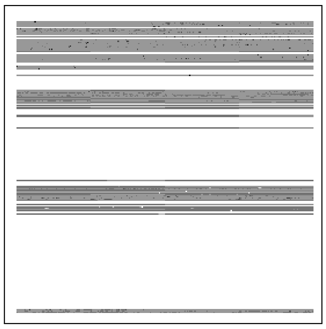
Figure B.7: Access patterns of SPECjbb with a checkpoint interval length of 4000 ms.

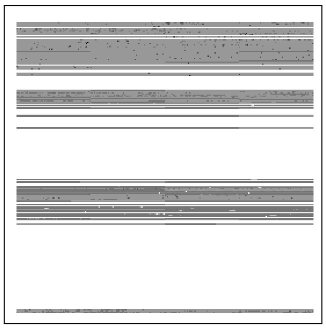

(a) *Checkpoint interval 8*

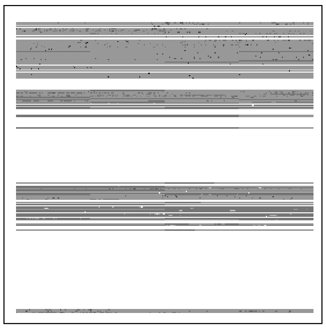

(b) *Checkpoint interval 18*

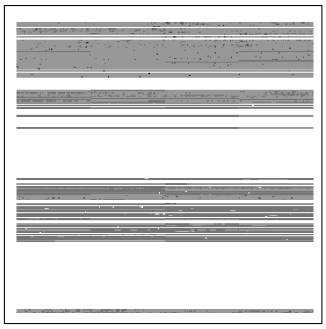

(c) *Checkpoint interval 49*


(d) *Checkpoint interval 68*


(e) *Checkpoint interval 73*


(f) *Checkpoint interval 132*


(g) *Checkpoint interval 164*


(h) *Checkpoint interval 220*

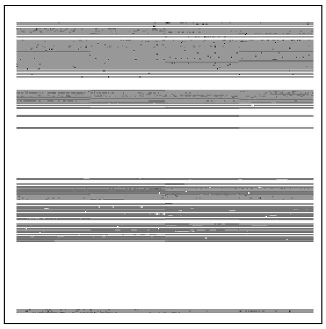

(i) *Checkpoint interval 229*

Figure B.8: Access patterns of SPECjbb with a checkpoint interval length of 8000 ms.

Appendix C

Evaluation

C.1 Results

C.1.1 Correctness and Main Memory Footprint

Checkpoint id	Replay Failure Reason
149, 248, 309, 320, 342, 381, 383	Landmark missed
343	Replay hang up
371, 394	Unspecified

Table C.1: Summarizing the replay failure reasons of the sparse checkpoint correctness experiments for a Linux kernel build. The failed simulation intervals that are categorized as unspecified have not given any indication why the replay failed. In case of a hang up, the replay execution has been aborted because the replay of the specific interval did not complete in a predefined maximal time period of 15 minutes.

C.1.2 Checkpoint Creation

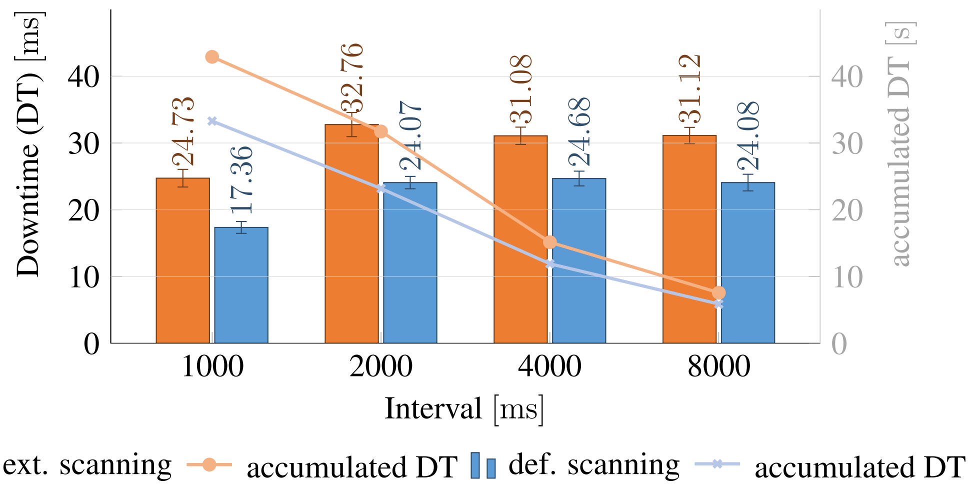


Figure C.1: The left axis plots the average downtime (DT) of a VM executing SPECjbb while creating continuous checkpoints. The right axis shows the accumulated downtime of the checkpointed VM.

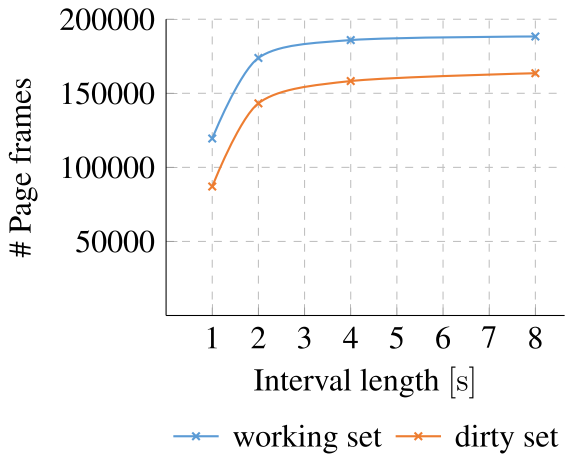


Figure C.2: Comparing the working set (exclusive page frames that have been skipped due to our optimization) and dirty set measurements for SPECjbb that we have collected during sparse checkpoint creation. The VM was configured with 2 GiB of main memory. The plotted values are the average of ten independent runs. See Table C.2 in the appendix for the exact values.

Interval [ms]	Workload	Avg. working set size	Std. deviation	Std. error
1,000	kb	29,389.411	18,794.741	5,943.419
2,000	kb	31,994.772	9,184.181	2,904.293
4,000	kb	36,934.561	14,285.08	4,517.339
8,000	kb	43,095.839	21,677.983	6,855.18
1,000	specjbb	$1.195 \cdot 10^5$	25,941.136	8,203.308
2,000	specjbb	$1.739 \cdot 10^5$	30,198.662	9,549.656
4,000	specjbb	$1.859 \cdot 10^5$	32,611.859	10,312.775
8,000	specjbb	$1.884 \cdot 10^5$	30,353.503	9,598.621

Interval [ms]	Workload	Avg. dirty set size	Std. deviation	Std. error
1,000	kb	23,245.89	25,673.56	8,118.69
2,000	kb	26,701.62	26,248.79	8,300.6
4,000	kb	33,291.03	38,706.49	12,240.07
8,000	kb	42,268.02	52,808.21	16,699.42
1,000	specjbb	87,070.58	17,853.75	5,645.85
2,000	specjbb	$1.43 \cdot 10^5$	26,214.03	8,289.61
4,000	specjbb	$1.58 \cdot 10^5$	27,690.81	8,756.6
8,000	specjbb	$1.64 \cdot 10^5$	30,367.38	9,603.01

Table C.2: Showing the working set size (exclusive page frames that have been skipped due to our optimization) and the dirty set size for a Linux kernel build (kb) and SPECjbb. The working set size was acquired during sparse checkpoint creation and represents the accessed set size. The dirty set size is acquired by the checkpoint mechanism.

Interval [ms]	Workload	Avg. downtime [ms]	Std. deviation	Std. error
1,000	kb	13.91	9.36	2.96
2,000	kb	14.57	17.95	5.68
4,000	kb	15.45	16.15	5.11
8,000	kb	15.44	3.39	1.07
1,000	specjbb	24.73	4.17	1.32
2,000	specjbb	32.76	5.7	1.8
4,000	specjbb	31.08	4.11	1.3
8,000	specjbb	31.12	3.83	1.21

Table C.3: Showing the average downtime of a Linux kernel build and SPECjbb executing in a VM configured with 2 GiB of main memory. We created continuous sparse checkpoints using varying interval lengths.

Interval [ms]	Avg. runtime excl. downtime [s]	Std. deviation	Std. error
1,000	814.258	1.79	0.566
2,000	802.506	5.192	1.642
4,000	794.812	10.316	3.262
8,000	791.063	3.153	0.997

Interval [ms]	Avg. runtime incl. downtime [s]	Std. deviation	Std. error
1,000	824.285	4.012	1.269
2,000	808.487	5.448	1.723
4,000	797.739	10.993	3.476
8,000	792.631	3.173	1.004

Interval [ms]	Avg. acc. downtime [s]	Std. deviation	Std. error
1,000	10.028	2.918	0.923
2,000	5.981	0.351	0.111
4,000	2.927	0.729	0.23
8,000	1.569	$2.307 \cdot 10^{-2}$	$7.294 \cdot 10^{-3}$

Table C.4: Showing the average runtime of a Linux kernel build executing in a VM configured with 2 GiB of main memory. We continuously created sparse checkpoints. The upper table shows the runtime exclusive the accumulated downtime that is induced by the copy mechanism of checkpointing. The table in the middle shows the runtime inclusive the accumulated downtime. The difference of the runtime depicted in both tables is the accumulated downtime listed in the bottom table.

C.1.3 Checkpoint Loading

Workload	# jobs	Avg. ram loading time [s]	Std. deviation	Std. error
kb	2	0.19	$9.64 \cdot 10^{-2}$	$9.64 \cdot 10^{-3}$
kb	4	0.47	0.25	$2.47 \cdot 10^{-2}$
specJBB	2	0.67	$5.72 \cdot 10^{-2}$	$5.72 \cdot 10^{-3}$
specJBB	4	0.69	0.13	$1.27 \cdot 10^{-2}$

Table C.5: Showing the average checkpoint loading time of 100 consecutive sparse checkpoints. The test machine was executing two or four jobs, respectively. The deployed workloads running inside the restored VM and on the load generating VMs was a Linux kernel build (kb) or SPECjbb. The shown loading times are the average of ten independent runs.

Bibliography

[1] The bochs emulator. http://bochs.sourceforge.net/. Accessed: 2018-05-30.

[2] Ceph fs. https://ceph.com/ceph-storage/file-system/. Accessed: 2018-05-14.

[3] Glusterfs. https://www.gluster.org/. Accessed: 2018-05-14.

[4] Kernel virtual machine. https://www.linux-kvm.org. Accessed: 2018-04-06.

[5] Kernel virtual machine, api. https://git.kernel.org/pub/scm/virt/kvm/kvm.git/tree/Documentation/virtual/kvm/api.txt?h=kvm-4.3-1. Accessed: 2018-04-29.

[6] Lz4 compressor. https://lz4.github.io/lz4/. Accessed: 2018-06-11.

[7] mongodb. https://www.mongodb.com/. Accessed: 2018-05-09.

[8] Phoronix test suite. https://www.phoronix-test-suite.com/. Accessed: 2018-05-11.

[9] Qemu. https://www.qemu.org/. Accessed: 2018-04-18.

[10] Qemu documentation. https://git.qemu.org/?p=qemu.git;a=tree;f=docs;hb=master. Accessed: 2018-04-19.

[11] specjbb 2015. https://www.spec.org/jbb2015/. Accessed: 2018-05-11.

[12] Virtualbox. https://www.virtualbox.org/. Accessed: 2018-04-06.

[13] Lars Albertsson and Peter S. Magnusson. Using complete system simulation for temporal debugging of general purpose operating systems and workload. In *Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE Computer Society.

[14] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003.

[15] 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/`.

[16] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, 2005.

[17] Nikhil Bhatia. Performance evaulation of intel ept hardware assist. White paper, VMWare Labs, 2009.

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

[19] Jim Chow, Tal Garfinkel, and Peter M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX 2008 Annual Technical Conference*, ATC'08, 2008.

[20] 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, Berkeley, CA, USA, 2005. USENIX Association.

[21] Peter J. Denning. The working set model for program behavior. In *Proceedings of the First ACM Symposium on Operating System Principles*, SOSP '67, New York, NY, USA, 1967. ACM.

[22] James E. Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. 01 2005.

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

[24] D. Forsgren, J. Eskilson, M. Christensson, F. Larsson, A. Moestedt, J. Hoegberg, G. Hallberg, B. Werner, and P. S. Magnusson. Simics: A full system simulation platform. 02 2002.

[25] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. 3, 05 2003.

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

[27] Michael Grottke and Kishor Trivedi. A classification of software faults. 27, 01 2005.

[28] Michael H. Sun and Douglas M. Blough. Fast, lightweight virtual machine checkpointing. 05 2018.

[29] Michael R. Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09. ACM, 2009.

[30] Hans-Joerg Hoexer, Kerstin Buchacker, and Volkmar Sieh. Umlinux – a tool for testing a linux system's fault tolerance. In *IN LINUXTAG 2002*, pages 6–9, 2002.

[31] Intel. *Intel 64 and IA-32 architectures software developer's manual combined volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4.*

[32] Nidhi Jain and Inderveer Chana. Cloud load balancing techniques : A step towards green computing. 9, 01 2012.

[33] H. Liu, H. Jin, and X. Liao. Optimize performance of virtual machine checkpointing via memory exclusion. Aug 2009.

[34] Haikun Liu, Hai Jin, Xiaofei Liao, Liting Hu, and Chen Yu. Live migration of virtual machine based on full system trace and replay. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, HPDC '09. ACM, 2009.

[35] 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), 1968.

[36] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05. USENIX Association, 2005.

[37] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. pages 412–421, July 1974.

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

[39] 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.

[40] 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.

[41] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen A. Herrod. Using the simos machine simulator to study complex computer systems. *ACM Trans. Model. Comput. Simul.*, January 1997.

[42] Joanna Rutkowska and Rafal Wojtczuk. Qubes os architecture. 02 2010.

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

[44] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Operating Systems Review - SIGOPS*, volume 36, 01 2002.

[45] Carl Waldspurger. Memory resource management in vmware esx server. 01 2003.

[46] S. Wallace and K. Hazelwood. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 209–220, March 2007.

[47] Johannes Werner. Assessment of virtual machine working-sets in simuboost. Bachelor thesis, Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, March12 2018.

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

[49] Chulho Won, Kiwoong Lee, Chansu Yu, Sangman Moh, Kyoung Park, and Myung Joon Kim. A detailed performance analysis of udp/ip, tcp/ip, and m-via network protocols using linux/simos. 01 2004.