Virtual Machine Checkpoint Storage and Distribution for SimuBoost

Master Thesis
by

Bastian Eicher

at the Department of Computer Science
Operating Systems Group
Karlsruhe Institute of Technology

Supervisor:	Prof. Dr. Frank Bellosa
Supervising Research Assistant:	Dipl.-Inform. Marc Rittinghaus

Created during: April 1st 2015 – September 4th 2015

I hereby declare that this thesis is my own original work which I created without illegitimate help by others, that I have not used any other sources or resources than the ones indicated and that due acknowledgment is given where reference is made to the work of others.

Karlsruhe, September 4th 2015

Abstract

While full-system simulation enables detailed analysis of workloads it is much slower than hardware-assisted virtualization, with slowdown factors ranging from 30 to 1000. The *SimuBoost* concept aims to combine the benefits of full-system simulation and virtualization. Checkpoints of a virtual machine's state are created in regular intervals and used to seed parallelized distributed simulations. To reach an optimal degree of parallelization, checkpoints have to be created in short intervals with low downtimes.

In this thesis we evaluate the viability of the *SimuBoost* concept. We improve an existing checkpointing prototype to minimize the downtime by replacing the underlying storage mechanism and performing asynchronous deduplication. We also implement a cluster-distribution solution in order to measure the actual speedup achievable by *SimuBoost*.

The evaluation shows that the new storage mechanism and performing deduplication asynchronously greatly reduce the downtime caused by checkpointing.

Deutsche Zusammenfassung

Full-System Simulation erlaubt eine detaillierte Analyse von Arbeitslasten. Allerdings ist die Ausführungsgeschwindigkeit bedeutend langsamer als die von Hardware-beschleunigter Virtualisierung, um Faktoren von 30 bis 1000. Das *SimuBoost*-Konzept zielt darauf ab, die Vorteile von *Full-System Simulation* und Virtualisierung miteinander zu kombinieren. Checkpoints des Zustands der Virtuellen Maschine (VM) werden in regelmäßigen Abständen erfasst und als Grundlage für parallelisierte verteile Simulationen genutzt. Um einen optimalen Grad an Parallelisierung zu erreichen, müssen die Checkpoints in kurzen Intervallen mit möglichst kurzen Downtimes (Ausfallzeiten) der VM erstellt werden.

In dieser Thesis evaluieren wir Tauglichkeit des *SimuBoost*-Konzepts. Wir verbessern einen existierenden Checkpointing-Prototypen, um die Downtime der VM zu minimieren. Wir tauschen das zugrundeliegende Speichersystem aus und führen asynchrone Deduplizierung durch. Wir implementieren zudem eine Cluster-Verteilungslösung, um den von *SimuBoost* tatsächlich erreichbaren Speedup (Beschleunigung) gegenüber klassischer Simulation messen zu können.

Die Evaluierung zeigt, dass das neue Speichersystem und die asynchrone Durchführung der Deduplizierung die durch Checkpointing verursachte Downtime deutlich reduzieren.

Contents

Chapter 1

Introduction

Full-system simulation is a useful analysis and development tool. For example, individual memory accesses can be traced and recorded. The *Undangle* [1] project uses a modified *QEMU*[2] version to record execution traces and allocation logs. This data can then be analyzed to detect the creation of dangling pointers leading to software vulnerabilities. The *Bochspwn* [3] project uses the instrumentation API provided by the *Bochs* [4] CPU emulator to identify *Windows* kernel race conditions via memory access patterns.

Unfortunately the execution speed of full-system simulators is often 30x to 1000x slower than that of hardware-assisted virtual machines [5]. Therefore, only short workloads can be simulated within a reasonable timespan. Interactive control of workloads often becomes impossible due to long delays responding to user input.

SimuBoost [5] aims to increase simulation speed using parallelization. A workload is executed in a virtual machine while taking checkpoints in frequent intervals and recording non-deterministic events. Based on these checkpoints each interval of the workload is then simulated separately. By using multiple simulation nodes the simulation can be parallelized. The authors of [5] present a formal model to predict the appreciable speedup compared to traditional simulation.

Baudis [6] presents a prototype implementation of a checkpointing system for *SimuBoost*. *QEMU*[2] was modified to create checkpoints and store them in a *MongoBD* [7] database. This work does not cover recording non-deterministic events or distributing checkpoint data to simulation nodes.

The existing work presents two main obstacles to evaluating the viability of the *SimuBoost* concept. First, the prototype suffers from relatively long VM downtimes when checkpoints are taken. The formal model of [5] indicates that the length of the checkpointing downtime has a significant impact on the speedup. Second, the formal model assumes the number of available simulation nodes are effectively infinite. It therefore cannot be used to predict the performance of a system with a less than optimal number of available nodes which limits its real-world applicability.

These limitations motivated our main objectives: We created a modified version of the formal model that takes the number of available simulation nodes into account. Then, we expanded upon the existing prototype to reduce the downtime and implemented a mechanism for distributing the checkpoints. This allowed us to perform measurements that could be compared to the predictions made by the new model.

We were able to significantly improve the write speed of checkpoints by replacing the original prototype's *MongoDB* storage with an append-only flat file. Additionally, we modified the prototype to perform deduplication asynchronously rather than while the VM is paused.

The *SimuBoost* concept achieves a speedup by distributing parts of a simulation across multiple nodes which will generally manifest as several physical machines in a compute cluster. To analyze this aspect of *SimuBoost* we implemented a system for distributing the checkpoint data across a cluster as the prototype records it.

We executed a number of different workloads using different checkpoint interval lengths and measured a number of performance metrics such as the VM downtimes and the time required to transfer a checkpoint to a remote machine. This enabled us to identify potential network, storage or processing bottlenecks.

This work presents a functioning prototype of *SimuBoost*'s checkpointing and cluster distribution concepts. We demonstrate the ability to create checkpoints with an interval of 2s with average downtimes of approximately 50ms, significantly improving upon the existing prototype. Using real-world workloads such as *Linux* kernel builds we were able to achieve speedup factors of 3.9 on clusters with 4 nodes. Our measurements closely tracked the predictions made by the formal model.

The remainder of this work is structured as follows: Chapter 2 provides an overview of virtual machine checkpointing and introduces the *SimuBoost* concept in more detail. In Chapter 3 we introduce the improved formal model and analyze the existing prototype implementation. In Chapter 4 we design an improved checkpointing prototype based on the analysis and evaluate a number of possible designs for checkpoint distribution. Chapter 5 details our implementation of capturing, storing and distributing checkpoints. An evaluation of our implementation using multiple workloads and a comparison with the formal model predictions is performed in Chapter 6. Chapter 7 concludes this work and provides an outlook of future work.

Chapter 2

Background

In this chapter we explore the concepts of virtual machines, full-system simulation and checkpointing. This then provides the basis for an explanation of the *SimuBoost* concept.

We also present specific applications used in our analysis and implementation such as the Virtual Machine Monitor *QEMU* and *Simutrace*.

2.1 Virtual Machines

Virtual machines simulate the hardware of an entire computer system, allowing the execution of an unmodified operating system.

Commonly used terms are:

Virtual machine monitor (VMM) the software running and controlling the virtual machines

Host operating system the operating system the VMM is running on

Guest operating system the operating system running within the virtual machine

There are many use-cases for virtual machines, such as reducing hardware costs by consolidating multiple physical machines into a single one running multiple virtual machines, testing and developing new operating systems and applications without exposing the underlying system to possible side-effects and analyzing malware by executing it in a tightly controlled environment [8].

2.1.1 Emulation

When the architecture of the underlying hardware differs from the architecture of the virtual machine the VMM fulfills the role of an emulator. It must translate instructions from the VM's instruction set to the host-machine's instruction set. Performing this translation on the fly is called Dynamic Binary Translation [9]. *QEMU* [2] and *Bochs* [4] are examples of simulators that use Dynamic Binary Translation. This process generally causes a significantly slower execution compared to running on native hardware.

When the hardware architecture of the underlying hardware is the same as the architecture of the virtual machine most of the guest code can be executed directly. However, the VMM must take care to intercept any sensitive instructions that could escape the VM.

The x86 architecture realizes different privilege levels using so called *rings*. Kernel code executes in ring-0 while user-code executes in ring-3. Therefore, running x86 VMs usually means running ring-3 code unmodified but patching ring-0 code to jump to the VMM for emulation instead [10]. *VirtualBox* [11] and *VMware Workstation* [12] are examples of VMMs that use this kind of Dynamic Binary Translation for x86.

In addition to handling sensitive instructions VMMs need to virtualize the Memory Management Unit (MMU). The guest operating system maintains its own page tables for each process, mapping guest logical addresses to guest physical (virtual) addresses. Each guest address corresponds to a specific host address. In order to handle virtual addresses in the VM using the physical MMU the VMM maintains a shadow page table [13] for each guest page table, mapping guest logical addresses directly to host physical (non-virtual) addresses.

2.1.2 Hardware virtualization

Hardware-assisted virtualization can be used to improve the performance of virtual machines. Instead of scanning for sensitive instructions and replacing them a trap-and-emulate approach can be taken: the CPU is configured to trap any privileged instructions and return control to the VMM instead of executing them. The VMM can then emulate the effect of the instruction before returning control to the VM. *Intel VT-x* [14] and *AMD-V* [15] add hardware virtualization support to the x86 architecture. These extensions add an additional mode of operation to the CPU where such traps can be set up.

Hardware-assisted VMs are sometimes called *Hypervisors*: [16]

Type-1 native hypervisors running directly on the hosts hardware, eliminating the need for a host operating system, for example *Hyper-V* [17].

Type-2 hosted hypervisors running on a conventional operating system just as other computer programs do, for example newer versions of *VMware Workstation* [12].

2.1.3 Full-system simulation

While hardware-assisted virtualization is mainly aimed at fast execution full-system simulation is intended to provide accurate reproduction of physical hardware and additional analysis tools. This generally results in considerably slower execution. In the following we mainly concentrate on functional full-system simulation which, unlike microarchitectural full-system simulation, does not simulate the internal implementation details of a CPU.

An advantage of functional full-system simulation over hardware-assisted virtualization is the ability to perform analysis on the level of individual instructions. Although a similar granularity may be achievable using hardware features such as page protections the resulting overhead is high enough to warrant simply switching to full emulation anyway.

Unlike normal emulation (see Chapter 2.1.1) full-system simulators are design for easy access to analytical data, e.g., by registering hooks for memory accesses. Another criteria is the existence of a cycle counter to be able to place captured events in time.

Bochs [4] is an example of a full-system simulator that provides these features. *MARSSx86* [18] is an open-source full-system simulator based on *QEMU* which has been extended to support cycle-accurate simulations of x86 systems. *Simics* [19] is an example of a commercial full-system simulator that additionally simulates memory access delays and caching effects.

2.2 Checkpointing

Virtual machine checkpoints capture the entire state of the encapsulated system. This consists of:

RAM The contents of the VM's system memory.

Disk The contents of the VM's disk. Usually a copy-on-write system is used rather than creating a complete copy of the underlying disk image (e.g., with `qcow2` on *QEMU*).

CPU The internal state and registers of the virtual CPU.

Devices The state of any other devices provided by the VMM, such as graphics adapters, network interfaces or sound cards.

Checkpoints can serve as a mechanism for suspending a VM to on-disk storage. The VM can then be resumed at a later point in time, potentially on a different physical host. Checkpoints are primarily used for migration and replication of VMs in data centers. By automatically creating checkpoints in short intervals the state of a VM can be duplicated to another location and be kept in sync.

Migration describes the process of moving a virtual machine from one host to another while it is still running. The goal is to minimize the interruption of any services running within in the VM. The two main techniques for VM migration are called pre-copy and post-copy memory migration.

In pre-copy memory migration the VMM copies all RAM pages from source to destination while the VM is still running on the source. If some pages change (i.e., become dirty) during this process they are re-copied until the page dirtying rate reaches the copying rate. Next, the VM is stopped on the source and the remaining dirty pages are copied to the destination. Then the VM is resumed on the destination. The downtime between stopping the VM on the source and resuming it on the destination ranges from a few milliseconds to seconds depending on the executed workload [20].

Post-copy VM migration starts by suspending the VM on the source. Then CPU and device state is transferred to the target where the VM is resumed. At the same time the source transfers the RAM pages of the VM to the target. At the target, if the VM tries to access a page that has not been transferred yet this causes a page fault. These faults are trapped at the target and redirected to the source which responds with the corresponding page [21]. Bradford et al. [22] applies an approach similar to pre-copy to the disk state. The write speed is throttled as needed to keep it lower than the transfer rate.

Post-copy sends each page exactly once over the network while pre-copy can transfer the same page multiple times if it is dirtied repeatedly during the migration. On the other hand, if the destination fails during migration pre-copy can continue executing the VM whereas post-copy cannot. Clark et al. [23] present a live VM migration strategy achieves downtimes of 60ms.

Liu et al. [24] present an alternative approach to VM migration. Rather than re-transferring RAM pages that are dirtied during the initial copy phase *ReVirt* [8] is used to record all non-deterministic events that occur within the VM. This log is then replayed on the target causing the same pages to be modified in the same way, thereby re-synchronizing the states of the two VMs.

Surie et al. [25] also employ logging and replay to synchronize the state of a source and target VM during migration. However, rather than creating a complete log of all non-deterministic, this approach only records UI interaction. This requires less network bandwidth than transmitting complete traces. Any remaining divergence is then handled by classic migration techniques. The paper mentions that this approach is unsuitable for scenarios that affect the persistent state of remote machines via a network connection. For example, replaying interaction with an e-mail client could cause a duplicate e-mail to be sent to the recipient in addition to recreating the state inside the VM.

Replication of VMs can be accomplished by creating checkpoints and transmitting them to one or more remote machines. These machines load the checkpoints but hold off the execution of the VM until a failover event occurs. The difficulty lies in ensuring there is no "gap" between the state captured by the last checkpoint and the state at the time of the failure.

Remus [26] is a checkpointing system that handles this by "hiding" the externally visible state of a VM (i.e., delaying network traffic) until the next checkpoint has been created and transferred. After creating a checkpoint the main VM continues execution in a mode called *speculative execution*. Network output, disk persistence, etc. is held back. At the same time the target VM loads the checkpoint and resumes the execution. The output of speculative execution is only released if migration to the target fails.

VM-microCheckpoint [27] is a framework for high-frequency checkpointing and rapid recovery of VMs. By applying various optimizations, such as those discussed below, *VM-microCheckpoint* is able to create checkpoints as often as every 50ms while incurring an average slowdown of 6.3%. Unlike *Remus* this system uses volatile storage on the same machine rather than persistent storage on remote nodes for the checkpoints. It is therefore targeted at handling transient disk failures rather than complete node failures.

2.2.1 Optimizations

Some of the presented checkpointing use-cases require checkpoints to be created at a high rate. Therefore, the downtime caused by taking the VM offline to copy state becomes a major issue. Additionally, the amount of disk space required to store entire VM states, usually containing complete RAM images in the gigabyte range, can grow too quickly.

Incremental checkpoints avoid having to copy the entire state of a VM every time. These checkpoints only store the state that has changed since the previous checkpoint was created (e.g., only the RAM pages that were modified). This reduces both the downtime and required storage space. A common method for determining which RAM ranges have been written to (dirtied) during a checkpoint interval is activating page protection to catch RAM write accesses [28].

Ta-Shma et al. [29] present a concept for creating checkpoints that capture consistent RAM, disk, CPU and device state using *Continuous Data Protection* (CDP). CDP provides a copy-on-write disk storage layer below the file system that allows rollbacks to snapshots. The paper presents an architecture consisting of an VMM-agnostic CDP server that keeps tracks of CDP snapshots and other checkpoint data as well as a VMM-specific IO interceptor. This provides for fast incremental checkpointing of VMs' block storage.

Deduplication is another technique for dealing with the high volume of checkpoint data that needs to be persisted on-disk. By identifying identical sets of data such as RAM pages both within individual checkpoints as well as across checkpoints these duplicates can be replaced with pointers to already persisted data.

libhashckpt [28] is hybrid checkpointing solution that uses both incremental checkpointing with page protection and hashing on GPUs to detect and deduplicate identical RAM pages.

Shrinker [30] extends the deduplication concept across the boundaries of a single VM. When migrating multiple VMs from one data center to another via a WAN connection the high degree of duplication between VMs running the same operating system is exploited to reduce the total amount of data that needs to be transferred between the two sites.

Introspection uses knowledge of the guest OS' internal state to reduce the size of checkpoints. For example, by determining which RAM pages are not currently allocated by the guest's memory management system these pages can be skipped when storing a RAM image [31].

Park et al. [32] describe a method for deduplicating RAM pages against copies that were already written to disk storage by the guest operating system itself. By tracking the relation between the guest operating system's page cache and disk sectors the amount of RAM data that needs to be copied for a checkpoint is reduced drastically.

2.3 QEMU

QEMU [2] is a popular open source VMM. It runs on Linux, Mac OS X and Windows as the host operating system. Guest operating systems do not need to be modified in order to run inside *QEMU*.

QEMU can operate in one of two modes: software emulation using a component called the *Tiny Code Generator* (*TCG*) [33] to perform dynamic translation of code from instruction set to another or hardware-assisted virtualization using the Linux Kernel module *KVM* [34].

TCG works by translating blocks of instructions from one instruction set to another dynamically at runtime. This enables *QEMU* to simulate a wide range of processor architectures on completely different hardware. *TCG* also rewrites instructions when the source and target instruction sets are the same e.g., when running an x86 VM on an x86 host machine. Here, *TCG* rewrites ring-1 instructions to ring-3 instructions among other things (see Chapter 2.1.1).

In addition to simply reproducing the functionality of one hardware architecture on another *TCG* can also be extended to introduce instrumentation code to the blocks it translates. This enables use cases such as adding counters to all instructions that perform memory access (see Chapter 2.1.3). This makes *QEMU* in *TCG* mode a candidate for *SimuBoost*'s simulation component.

KVM is a Linux Kernel module that exposes a CPU's native virtualization features (see Chapter 2.1.2) to processes running in user-space. When running in *KVM* mode *QEMU* no longer performs any CPU instruction translation, instead instructing *KVM* via the `/dev/kvm` device to switch the physical CPU in and out of a "virtualized" mode. Execution in this mode requires the host and guest systems to share the same architecture. Additional instrumentation on an instruction level is not possible. The execution speed is significantly higher than in *TCG* mode. This makes it a candidate for *SimuBoost*'s virtualization component.

QEMU manages a set of virtual devices such block devices backed by disk images and virtual network interfaces. Other than the CPU the virtual hardware presented to a guest operating system is the same when operating *QEMU* in *TCG* or *KVM* mode. This is what makes *QEMU* particularly interesting for *SimuBoost*: VM states generated in a virtualized environment can potentially be transferred to an equivalent simulated environment.

Disk images containing the guest operating system are usually stored in the `qcow2` format. This format supports sparse storage, i.e., it only takes up disk space that the guest operating system actually uses. This way, an emulated 100 GB disk might occupy only a few hundred megabytes on the host. The `qcow2` format also allows the creation of overlay images that record the difference from a base image file. This provides the possibility for reverting the VM disk's contents to an earlier state [35].

QEMU can save the entire state of a virtual machine in a so-called snapshot and restore it a later point in time. This and many other actions can be controlled from the *QEMU* Monitor console. This console provides a command-line interface called the *Human Monitor Protocol* (HMP) with commands such as `stop`, `cont`, `savevm` and `loadvm` [36].

QEMU creates snapshots by iterating over each virtual device and instructing it to serialize its current state. For most devices such as the network interface this consists of a simple copy of the memory used to hold its state. For the VM's RAM, which is also represented as a device in *QEMU*'s architecture, a complete copy is dumped. *QEMU*'s default snapshot feature does not support incremental RAM snapshots. For block devices the snapshot feature relies on the underlying image format to preserve the state of a specific point in time e.g., `qcow2`'s copy-on-write functionality. The other snapshot data is then embedded within the same image file, grouping all data required to resume the VM's execution later in a single place.

2.4 Simutrace

Simutrace [37] aims at establishing technologies to overcome current limitations inherent to full-system simulation, such as a lack of detailed tracing capabilities and slow execution speeds.

2.4.1 Storage Server

The main component of *Simutrace* is a framework for efficient tracing of memory accesses and other system events.

The framework uses a client-server architecture. The client is implemented as an extension of an existing full-system simulator, which traces the desired events and submits them to the server. The server, called *storage server*, handles tasks such as compression, storage and data retrieval for analysis.

The storage server uses a modular design, which makes it easily extensible:

Streams represent continuous sets of elements such as trace events. New element types with fixed or variable lengths can be registered. They are identified by *Globally unique idendifiers* (GUIDs).

Encoders process the elements of a stream. They can perform tasks such as compression or deduplication. New encoders can be added to the storage server. They are associated with specific stream element types.

Stores are responsible for the on-disk persistence of streams. New stores can be added to the storage server. They are addressed by *storage specifiers*, consisting of a prefix identifying the store to use and the file path controlling where the store places its files.

2.4.2 SimuBoost

Another concept of *Simutrace* is *SimuBoost* [5]. *SimuBoost* aims to overcome the execution speed limitations of functional full-system simulators.

Full-system simulation is generally not parallelizable because the specific instructions executed at any point of the simulation as well as the input and output data for these instructions may depend on the results of previous instructions. However, by running a workload using hardware-assisted virtualization (see Chapter 2.1) and creating checkpoints in regular intervals (see Chapter 2.2) starting points for multiple parallel simulations can be efficiently "precomputed".

Each of these simulations then re-executes the same instructions that were executed by the virtualization during a specific interval of the workload, now optionally performing additional tracing. Figure 2.1 illustrates how multiple simulations are executed in parallel distributed across nodes in a cluster.

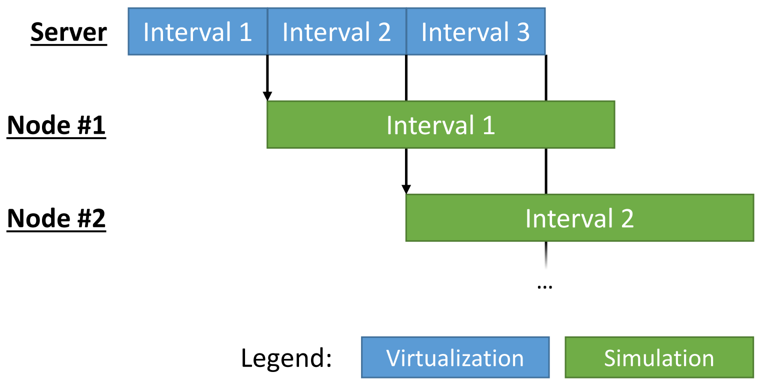

Figure 2.1: Checkpoints are created at the boundary of each interval and seed new simulations.

In order to ensure that these simulations to not diverge from the original virtualization any non-deterministic events that occur in the virtualization need be recorded and then replayed at precisely the same time in the simulations.

There currently does not exist an implementation of *SimuBoost*. The *SimuBoost* paper [5] provides a formal model that predicts the speedup achievable by an implementation.

Let n := the number of intervals, L := the checkpointing interval, s_{log} := the slowdown factor caused by logging non-deterministic events in the VM, t_c := the constant VM downtime for a checkpoint, t_i := a simulation's initialization time, T_{vm} := the workload's run-time with virtualization and T_{sim} := the workload's run-time with conventional full-system simulation.

The total run-time of a parallelized simulation T_{ps} is assumed to be the run-time of the entire virtulization including checkpointing followed by the simulation of the last interval. Figure 2.2 visualizes this relation.

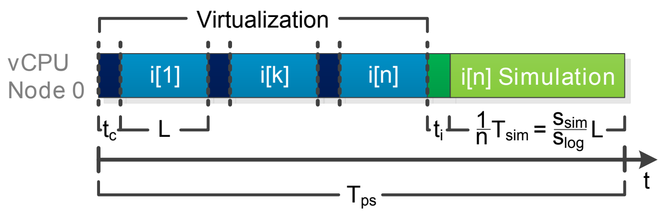

Figure 2.2: "Overview of Parameters. The simulation of the last interval is scheduled on the virtualization node." [5]

This can be expressed as:

$$T_{ps}(L) = s_{log}T_{vm}(\frac{t_c}{L} + 1) + t_i + \frac{s_{sim}}{s_{log}}L \tag{2.1}$$

The achievable speedup would then be:

$$\begin{aligned} S(L) &= \frac{T_{sim}}{T_{ps(L)}} \\ &= \frac{s_{log}T_{vm} \cdot s_{sim}L}{s_{log}^2 T_{vm}(t_c + L) + s_{log}t_iL + s_{sim}L^2} \end{aligned} \tag{2.2}$$

The paper calculates the number of nodes required to achieve an optimal degree of parallelization as:

$$\begin{aligned} N_{opt} &= N(L_{opt}) \text{ with} \\ N(L) &= \left\lceil \frac{t_i + \frac{s_{sim}}{s_{log}}L}{t_c + L} + 1 \right\rceil \text{ and} \\ L_{opt} &= \sqrt{\frac{s_{log}^2 T_{vm} t_c}{s_{sim}}} \end{aligned} \tag{2.3}$$

As a guiding sample the paper's authors chose $T_{vm} = 3600s$, $s_{sim} = 100$, $s_{log} = 1.08$, $t_c = 0.1s$ and $t_i = 1s$. Using these values N_{opt} is calculated to be 90 with a speedup of 84. Baudis [6] presents a prototype implementation of the checkpointing component of *SimuBoost* to help determine a real-world value for t_c.

Chapter 3

Analysis

In this chapter we analyze the *SimuBoost* concept presented in the *SimuBoost* paper [5] and the checkpointing prototype presented by Baudis [6]. We identify bottlenecks in the current implementation and compare possible alternatives.

3.1 Requirements

SimuBoost uses hardware-assisted virtualization and checkpointing at regular intervals to quickly generate snapshots of multiple points in time during the execution of a workload. These snapshots can then be distributed across a cluster of worker nodes to perform full system simulation. This effectively parallelizes an inherently sequential task (see Chapter 2.4.2).

The formal model presented in the *SimuBoost* paper [5] predicts the speedup achievable compared to sequential execution of the entire workload in a full system simulator.

Downtimes caused by checkpointing prolong the execution of the virtualization, which is a non-parallel process. This in turn delays the execution of simulations thereby reducing the achievable speedup. Increasing the interval length and thereby reducing the number of checkpoints reduces the relative impact of downtimes. However, the degree of potential parallelization is lowered. Figure 3.1 visualizes this relation.

This motivates our goal of minimizing the downtime caused by checkpointing. We aim to achieve a downtime lower that 100ms because:

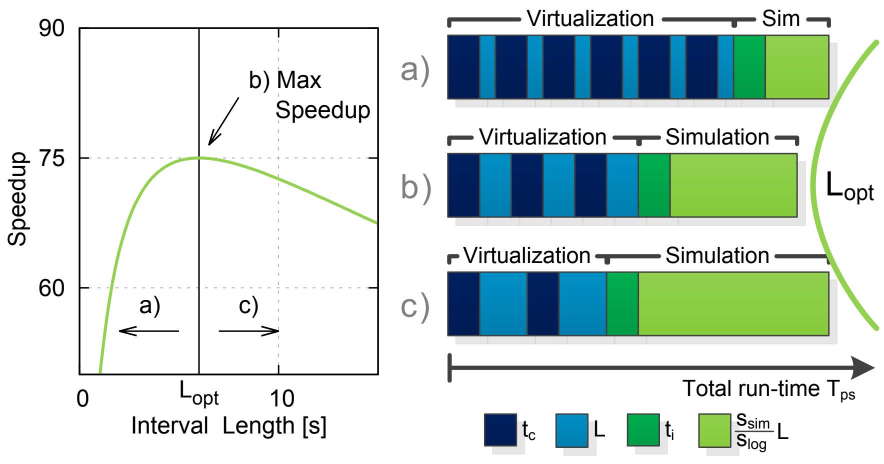

Figure 3.1: "The right interval length is crucial for an optimal speedup (b). With too short intervals (a) the VM downtime dominates and the speedup rapidly decreases. Too long intervals (c) do not parallelize optimally." [5]

- The original *SimuBoost* paper [5] assume a 100ms downtime in their theoretical model, based on the 100ms upper bound demonstrated in *Remus*' [26] implementation of a VM checkpointing system.
- Downtimes below 100ms would generally be perceived as instantaneous by humans [38], allowing interactive use of a system. This is generally not possible with full system simulation.
- *Remus* [26] shows that downtimes below 100ms generally do not affect a VM's network connectivity.

Thus, an effective implementation of *SimuBoost* depends on the ability to capture the state of a VM in as little time as possible and then efficiently transfer it to remote machines.

The recorded checkpoint data may be intended for deferred use rather than immediate seeding of simulations. This necessitates long-term persistent storage.

Virtual machine RAM sizes are often in the gigabyte ranges. Therefore, storing entire RAM snapshots on-disk is cost prohibitive. Also, the disk IO required to capture these magnitudes of data will likely not complete within the 100ms timeframe we allocated.

This motivates a secondary goal of our design: The amount of on-disk storage used by a *SimuBoost* implementation must remain within "reasonable" limits.

3.2 Queuing

The formal model for *SimuBoost* presented in Chapter 2.4.2 assumes that there are a sufficient number of simulation nodes available to meet any demands without queuing (N_{opt})). In this chapter we present a modified version of this model that accounts for a limited number of available simulation nodes. This allows us to evaluate the concept with a more constrained hardware setup and shows whether it remains viable in such scenarios.

As shown in Figure 3.2, to achieve maximum parallelization new simulation nodes are needed until the first interval simulation finishes. This means that, let $N :=$ number of available simulation nodes, the following condition must be met:

$$N \geq N_{opt}(L) = \frac{t_i + \frac{s_{sim}}{s_{log}} L}{t_c + L} \tag{3.1}$$

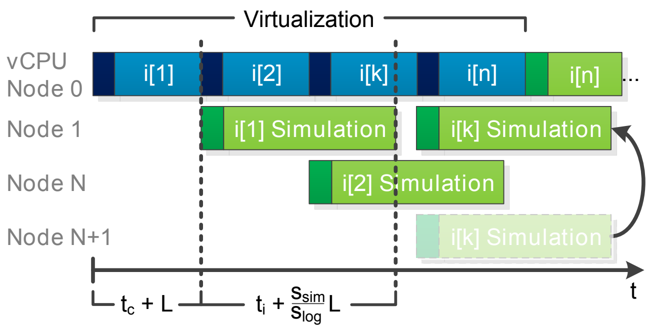

Figure 3.2: "New simulation nodes are needed until the first interval simulation finishes. Subsequent intervals can be scheduled onto previously allocated nodes." [5]

This equation assumes that the checkpointing downtime always remains constant. However, incremental checkpointing approaches as used by the prototype presented by Baudis [6] invalidate this assumption. Therefore, we decided to replace the constant downtime factor t_c with a checkpointing slowdown factor s_{cp}. This allows us to accommodate for a checkpointing mechanism with a downtime that depends on the interval length L. This represents the increased number of dirty pages that accumulate during a longer interval. All instances of $t_c + L$ are replaced with $s_{cp}L$.

Applying this replacement and assuming the number of nodes is insufficient we get the following condition for the applicability of our modified model:

$$N < N_{opt}(L) = \frac{t_i + \frac{s_{sim}}{s_{log}} L}{s_{cp} L} \tag{3.2}$$

Figure 3.3 illustrates how simulation jobs could be queued and executed when the number of nodes is insufficient.

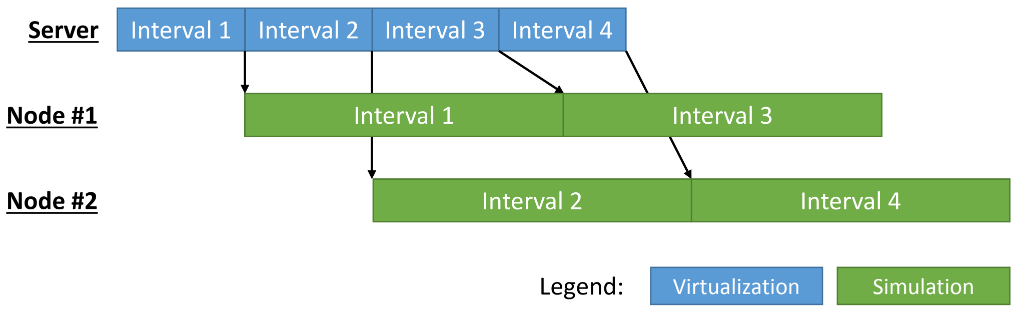

Figure 3.3: When a new simulation job becomes available it is dispatched to a free node in the cluster. If there are an insufficient number of nodes available this can cause a backlog to form. Compare to Figure 2.1 where we assume an unlimited number of nodes.

Since new simulation jobs are now generated faster than they can be processed we can assume that the workload will be distributed roughly equally among the available nodes. This means each individual node will be busy for a duration of:

$$\begin{aligned} T_{busy}(N, L) &= \frac{n(t_i + \frac{s_{sim}}{s_{log}} L)}{N} \\ &= \frac{\frac{s_{log} T_{vm}}{L}(t_i + \frac{s_{sim}}{s_{log}} L)}{N} \\ &= \frac{T_{vm}(s_{log} t_i + s_{sim} L)}{N \cdot L} \end{aligned} \tag{3.3}$$

The last node will start working when the Nth checkpoint has become available at:

$$T_{last}(N, L) = N \cdot s_{cp} L \tag{3.4}$$

The simulation is finished when the last node is finished. Therefore the parallel simulation time for a constrained number of nodes is:

$$\begin{aligned} T_{ps}(N,L) &= T_{last}(N,L) + T_{busy}(N,L) \\ &= N \cdot s_{cp}L + \frac{T_{vm}(s_{log}t_i + s_{sim}L)}{N \cdot L} \end{aligned} \tag{3.5}$$

The achievable speedup can then be computed as:

$$\begin{aligned} S(N,L) &= \frac{s_{sim}T_{vm}}{T_{ps}(N,L)} \\ &= \frac{N \cdot L \cdot s_{sim}T_{vm}}{N^2 \cdot s_{cp}L^2 + T_{vm}(s_{log}t_i + s_{sim}L)} \end{aligned} \tag{3.6}$$

To find the optimal interval length depending on the number of available nodes N we solve $\frac{\delta}{\delta L}S(N,L) = 0$ for L and exclude negative results:

$$L_{opt}(N) = \frac{\sqrt{s_{cp}s_{log}T_{vm}t_i}}{N \cdot s_{cp}} \tag{3.7}$$

In order to determine for which values of N our assumption of an insufficient number of nodes holds we define the following helper function:

$$N_{demand}(N) = N_{opt}(L_{opt}(N)) - N \tag{3.8}$$

Positive values for N_{demand} indicate our model is applicable, negative values indicate the original model presented in Chapter 2.4.2 is applicable.

Figure 3.4 plots $L_{opt}(N)$ and $N_{demand}(N)$ using sample values for the previously introduced constants. This illustrates that the optimal interval length quickly drops as the number of available nodes increases.

This analysis only optimizes the total simulation runtime. It does consider the downtime requirements specified in 3.1. While very long intervals may increase parallelizability for a limited number of nodes they could also lead to noticeable delays during checkpointing, limiting the interactivity of the system.

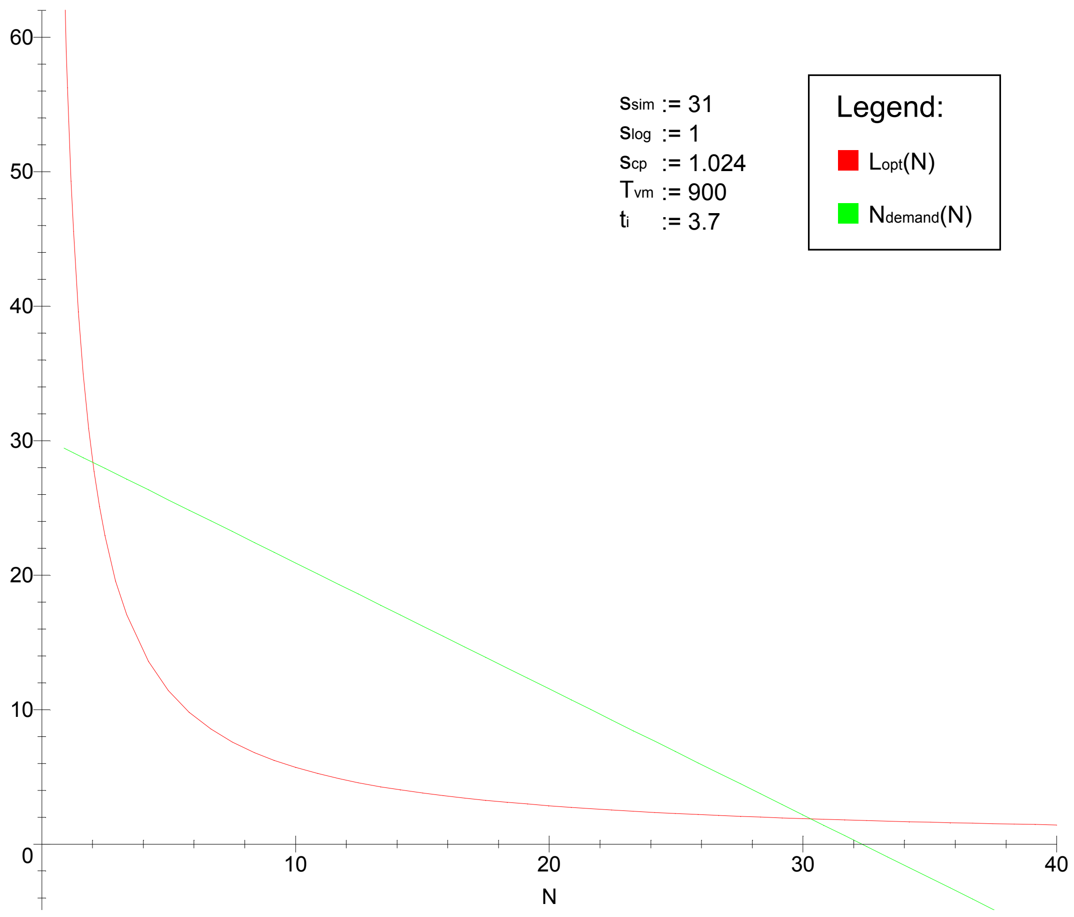

Figure 3.4: Given the sample values, our model is valid for up to 32 nodes ($N_{opt}(32.3) \approx 0$). Optimal interval lengths based on the number of available nodes range from $L_{opt}(1) \approx 57$ to $L_{opt}(32s) \approx 1.8s$.

3.3 Prototype

The checkpointing prototype presented by Baudis [6] is implemented as a modification of *QEMU* [2].

The prototype runs *QEMU* in *KVM* mode [34] to generate checkpoints. A checkpointing thread pauses the execution of the virtual machine in regular intervals and creates an incremental checkpoint. This is accomplished by iterating over all RAM pages and disk sectors that have been modified since the last checkpoint and storing them along with a cumulative view of the current state. The presented implementation deduplicates the pages and sectors using an in-process hashmap (see Chapter 3.3) before using *MongoDB* [7] as an external database for storing them.

3.3.1 Incremental checkpointing

Incremental checkpointing greatly reduces the amount of data that needs to be stored for each checkpoint. This serves two of the requirements identified in Chapter 3.1: keeping the VM downtime short and limiting the amount of on-disk storage required.

Write accesses to RAM pages and disk sectors are tracked while the VM is running to avoid time-consuming checks during VM downtimes. Baudis [6] shows that on average only 5 to 10% of a VM's RAM needs to be saved for a checkpoint.

Using incremental checkpoints, however, makes loading an arbitrary checkpoint more difficult since data from all previous checkpoints may be required to reconstruct a complete VM snapshot. A linear walk through all checkpoints, applying the recorded changes one by one, is unsuitable for *SimuBoost* since this would cause a linear increase in simulation setup time.

Baudis [6] solves this problem by storing a complete map of the VM's RAM and disk states for each checkpoint. Rather than embedding the actual RAM pages and disk sectors in these maps, the prototype stores keys that uniquely identify the underlying data. This approach makes it possible to completely reconstruct a checkpoints state without having to scan through all previous checkpoints. By storing keys instead of the actual data for each checkpoint the space savings of the incremental checkpointing approach remain intact. Figure 3.5 illustrates the employed data structure.

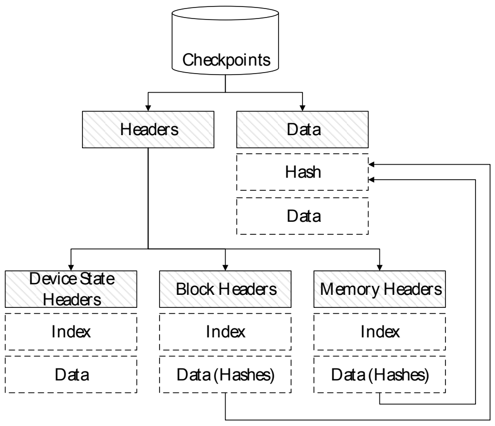

Figure 3.5: "The database layout. Headers are accessed by a sequentially increased index and reference data entities, which are accessed by their hash." [6]

3.3.2 Deduplication

Chapter 3.3.1 introduced the concept of RAM and disk state maps that store keys pointing to data rather than the data itself. Baudis [6] uses hashes of RAM pages and disk sectors (using a hash function that is assumed to be collision-free for this purpose) to generate these keys.

Using hashes over the actual contents of the pages and sectors enables additional deduplication beyond the deduplication already achieved by excluding data that was not changed since the previous checkpoint. Baudis [6] shows great potential for deduplication of RAM pages and disk sectors both within single checkpoints and across checkpoints. Using the incremental data an additional 15 to 55% could be eliminated using inter-checkpoint deduplication.

Park et al. [32] demonstrate that there is also significant potential for deduplication RAM storage against disk IO. This may effectively reduce the checkpointing problem to efficiently capturing RAM state.

3.3.3 Database

The RAM pages and disk sectors need to be stored in a way that meets the following requirements:

Addressable The pages and sectors need to be addressable by a key (i.e., their hash value as stored in the state maps) so that checkpoint states can be reconstructed as described in Chapter 3.3.1.

Linear lookup speed Retrieving a page or sectors must be possible in linear time to avoid checkpoint reconstruction becoming a bottleneck once a significant amount of checkpoints have been recorded.

Network accessible The pages and sectors must be retrievable via the network because the *SimuBoost* design executes simulations on remote worker nodes in a cluster.

These requirements lead to choosing a database management system (DBMS), specifically the subgroup of key-value stores. Baudis [6] uses *MongoDB* for this purpose.

Prior to communicating with *MongoDB* via TCP the prototype uses an in-process cache in form of a red-black tree for fast lookup. If an entry already exists in the tree it can be pruned. If it is new the corresponding unhashed data is stored in the database.

We measured the average downtime of presented implementation with our reference hardware and workloads used in Chapter 6 to be approximately 500ms. This fails to meet the requirements described in Chapter 3.1.

Baudis [6] shows that the downtime during checkpointing is dominated by communication with the database, accounting for 40%-60% of the time per checkpoint. In Chapter 3.4 we explore alternatives to *MongoDB* as potential solutions for this issue.

3.3.4 Distribution

Once a checkpoint has been stored the execution of the virtual machine is resumed.

Each checkpoint can then be distributed to a worker node running *QEMU* in *TCG*[33] mode. The nodes acquire the checkpoint data directly from the *MongoDB* instance via the network. Figure 3.6 illustrates the communication between the components involved.

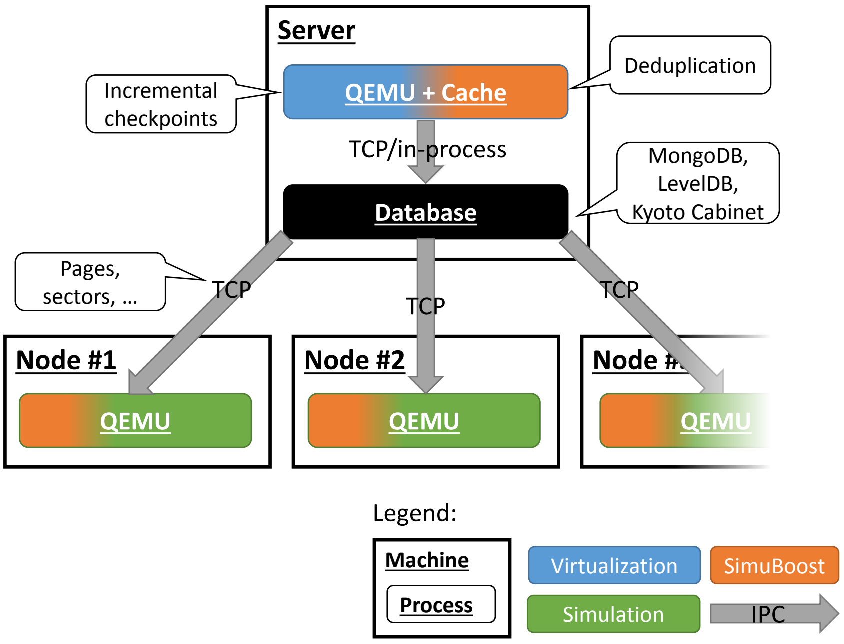

Figure 3.6: The checkpointing implementation presented in [6] implements deduplication inside the *QEMU* process. It stores the deduplicated checkpoint data in a database running on the same physical machine. Simulation nodes can reconstruct checkpoints by retrieving the relevant data from the database via the network.

The prototype implementation does not handle the automatic dispatching of newly generated checkpoints to worker nodes. It also does not attempt to record and replay non-deterministic events. Furhtermore, the underlying *QEMU* version is not capable of loading snapshots generated in *KVM* mode when in *TCG* mode.

3.3.5 Goals

The analysis of the presented checkpointing prototype motivates the following goals for this work:

We aim to reduce the downtime causing by VM checkpointing to ensure a high speedup can be reached.

We wish to implement automatic dispatching of checkpoints to worker nodes in order to measure the interactions between simultaneous checkpoint creation and loading.

We wish to evaluate the overall feasibility of *SimuBoost* by performing complete virtualization+simulation runs and comparing the achieved speedup with the theoretical predictions of [5].

3.4 Databases

The incremental checkpointing (see Chapter 3.3.1) and deduplication (see Chapter 3.3.2) concepts have proven to be sound. However, as mentioned in Chapter 3.3.3, the presented checkpointing approach suffers from too long downtimes. In this chapter we analyze possible alternatives to the existing storage mechanism to reduce this downtime.

The deduplication concept encodes checkpoints as maps of addresses (RAM pages or disk secors) to hash values. This means means we need to account for two types of storage: State maps and a hash table.

The state maps are a growing set of tables which are immutable once written. The maps in the set must be quickly addressable by the checkpoint index they are associated with. The complete contents of individual maps will be read sequentially when restoring checkpoints.

The limited number of individually accessible and sequentially addressable data blocks map naturally to the concept of files in a file system. A distributed file system such as the *Network File System* (*NFS*) could be used to share them with distributed simulation nodes.

The hash table maps the hash values to the actual deduplicated data. New entries are constantly added, necessitating fast insert operations. Each incremental checkpoint provides only a subset of all pages and sectors required to reconstruct a complete snapshot of a virtual machine's state (see Chapter 3.3.1). Therefore, fast loading of checkpoints requires fast random-access when reading from the hash table.

Since each entry in the hash table needs to be individually retrievable classic file system are unsuitable for the task. Their performance is usually severely degraded when storing billions of individual files in a directory. Databases on the other hand are capable of handling such large data sets while providing fast insert and random-access retrieval operations. Many DBMS' also provide network interfaces. Therefore a database would be suitable for storing both the state maps and the hash table.

The amount of RAM of the host system may not be sufficient to hold the entire set of checkpoints for the duration of the simulation, requiring some form of on-disk persistence. Conservative estimations based on data provided by Baudis [6] show that data from 30 minutes of checkpointing could easily exceed 16GB despite deduplication. Additionally, the simulations may be spawned at a later point in time rather than during the creation of the checkpoints. The system storing the checkpoints may be taken offline in the meantime, again necessitating on-disk persistence. Therefore, a DBMS to be used for *SimuBoost* must not be an in-memory only solution.

3.4.1 MongoDB issues

Baudis [6] determined that storing checkpoint data in *MongoDB* accounted for 40%-60% of the downtime. Possible explanation for this observation might be:

MongoDB is a document database [39] rather than a plain key-value store. The additional data structures associated with each entry in the database may cause a certain overhead.

MongoDB encodes data using *BSON* (a binary-encoded serialization of *JSON*-like documents [40]). While *BSON* is designed to be lightweight and efficient, the necessity to wrap every single RAM page and disk sector in an additional data structure may contribute to the slowdown.

The journaling mechanism employed by *MongoDB* ensures on-disk consistency in case of unexpected termination, e.g., caused by power failure [39]. While this is usually a desirable trait for a database, it serves no purpose for *SimuBoost* since any failure during the checkpoint writing phase would interrupt the execution of the virtualization as well. *MongoDB*'s journal can thus be deactivated.

Baudis [6] executed *QEMU* and *MongoDB* on the same physical machine. However, the two processes communicated via TCP/IP on the loopback interface rather than using a low-overhead IPC method such as shared memory.

3.4.2 Alternatives

Since we have established that databases, key-value stores in specific, are a good match for our storage requirements in principle but *MongoDB* has proven to be unsuitable, we attempted to identify potential alternative databases. We applied the following criteria for our candidates:

Simple key-value design The database should be a simple key-value store rather than a complex document store in order to avoid overhead caused by additional data structures or encodings.

On-disk persistence The database must provide on-disk persistence rather than operate solely in-memory. This is required to enable deferred use of checkpoints rather than immediate seeding of simulations (see Chapter 6).

Swapping to disk In addition to simply duplicating its entire state to the disk the database should also be able to swap out currently unused parts of its working set to reduce the amount of physical RAM required.

In-process In order to avoid the need for IPC altogether, the database should be a library embeddable into a host application.

Network accessible The database should provide its own network interface. Otherwise we would have to implement our communication scheme to transfer RAM pages and disk sectors to remote worker nodes in a cluster.

In Table 3.1 we compare a set of commonly used key-value stores against these criteria.

Redis is not a suitable candidate due to its inability to swap parts of its working set to on-disk storage. The same holds true for *Memcached*.

LevelDB meets all criteria except providing a network interface. *Kyoto Cabinet* meets all requirements. The in-process component is accompanied by a stan-

	Simple	Persistence	Swapping	Network	In-process
MongoDB [7]	No	Yes	Yes	Yes	No
Redis [41]	Yes	Optionally[42]	No	Yes	No
Memcached [43]	Yes	No	No	Yes	No
LevelDB [44]	Yes	Yes	Yes	No	Yes
Kyoto Cabinet [45]	Yes	Yes	Yes	Yes	Yes

Table 3.1: Comparison of the following database properties: Simple key-value design, on-disk persistence, swapping to disk, network accessible and in-process execution.

dalone server called *Kyoto Tycoon* [46].

3.4.3 Measurements

We replaced MongoDB in Baudis' [6] implementation with *LevelDB* and *Kyoto Tycoon* and measured each database's respective impact on the downtime.

We executed a sample workload consisting of a Linux kernel build in *QEMU* creating checkpoints in two second intervals using each of the databases. This workload is similar to one of the workloads chosen for the evaluation in [6] and taxes both RAM and the disk.

We also ran a modified version of the experiment where we write the deduplicated RAM pages and disk sectors directly to an append-only flat file. Here, the databases were only used to store the state maps. Writing the data to a flat file provides a lower bound for the databases' IO. This enables us to evaluate the overhead caused by the databases on-disk indexing structures.

Finally, we ran another modified version of the experiment where we performed a simple `memcpy` of all modified RAM pages and disk sectors rather than deduplicating and storing them. The databases were again only used to store the state maps (see Chapter 3.3.1). This scenario serves two purposes: First, by copying all pages without first deduplicating them we are able to determine whether delaying the deduplication until after the VM has resumed execution is feasible or causes the copy phase to take too long. Second, `memcpy` provides a lower bound for iterating over all modified pages and sectors.

The results are displayed in Table 3.2.

MongoDB, in its configuration as used in [6], took approximately 1.5s. By deactivating the journal (see Chapter 3.4.1) we were able to reduce this to approximately

	Deduplicate + store	Deduplicate + flat file	`memcpy`
MongoDB (with journal)	1 475 ms	-	-
MongoDB (no journal)	1 004 ms	861 ms	327 ms
LevelDB	13 167 ms	896 ms	307 ms
Kyoto Cabinet	1 078 ms	987 ms	324 ms

Table 3.2: The average VM downtime using different databases and varying the way the data is processed.

1s.

LevelDB, despite being a simple key-value store and running in-process, turns out to be considerably slower than *MongoDB* for this use-case, taking over 13s on average. This can likely be attributed to *LevelDB*'s on-disk structure being lexicographically sorted [47]. While this improves the lookup speed for random keys it considerably slows down write access and is not well suited to keys of identical length such as hashes.

Kyoto Cabinet achieves performance similar to *MongoDB*. This indicates that switching away from a document-oriented database and using in-process execution rather than loopback TCP/IP communication does not impact the performance of our use-case significantly.

When using a flat file to store the actual RAM page and disk sector data the choice of the database has only a negligible influence on the downtime. This is to be expected since the database is now only handling state maps. Creating checkpoints in this fashion requires approximately 900ms on average for all databases. This value is lower than those achieved by using any of the databases to store the actual data. The databases' on-disk indexing structures are stressed much more strongly when storing key-value pairs for every RAM page and disk sector instead of just state maps. Therefore we can conclude that these indexing structures contributes significantly to the VM downtime with the *MongoDB* scenario taking 14% longer when the database is used without a backing flat file. *Kyoto* performs slight better with an 8% slowdown, while *LevelDB*'s lexicographically-sorted on-disk format suffers the worst requiring 93% more time.

When performing a `memcpy` of all dirty pages and sectors without performing deduplication first the choice of database also becomes largely irrelevant. The process takes approximately 320ms on average. This result shows us that simply copying all modified data during the downtime and performing the deduplication later would significantly shorten the downtime.

3.5 Conclusion

Chapter 3.2 shows that the *SimuBoost* concept remains theoretically viable even with a limited number of available simulation nodes.

Chapter 3.3 elaborates that the existing incremental checkpointing and deduplication schemes are highly effective and contribute to reducing the checkpointing downtime.

However, as shown in Chapter 3.4, existing database solutions do not meet the performance requirements specified in Chapter 3.1.

Performance measurements of flat file writing compared to databases indicate an intrinsic overhead caused by database on-disk structures. While simply appending data to a file optimizes write-performance databases also manage data structures such as indexes to allow fast retrieval of specific datasets. Additionally databases often provide network connectivity, enabling simultaneous retrieval of datasets from multiple machines.

Performance measurements of `memcpy` indicate that simply copying the affected RAM pages and disk sectors during the downtime for asynchronous processing could decrease the downtime significantly.

Chapter 4

Design

Based on the conclusions drawn in Chapter 3.5 we determined that the incremental checkpointing and deduplication approaches are sound. However the database employed by the prototype does not meet the performance requirements.

Our checkpointing design is based off of the existing prototype described in Chapter 3.3. The primary goal of our new design is to reduce the length of the VM downtime to 100ms or less to ensure *SimuBoost* achieves the desired speedup. Based on our findings in Chapter 3.4 processing the checkpoint data asynchronously is indicated to achieve this goal.

Using a simple append-only on-disk representation rather than a classic database also promises to be beneficial. While this may harm read-performance, Chapter 3.1 shows that write-performance has the greatest impact on the achievable speedup. Since the reasoning for using a database presented in Chapter 3.3.3 still holds we need a solution that provides equivalent functionality (e.g., network accessibility).

As an additional goal we wish to evaluate the viability of distributing simulations across a cluster. The design must be able to determine when a checkpoint is complete (i.e., available for simulation) and efficiently retrieve all data associated with a checkpoint on a remote machine.

4.1 Checkpointing

The checkpointing prototype presented by Baudis [6] pauses the VM's execution in regular intervals. This allows for capturing a consistent snapshot of the VM's state.

Checkpoints are created incrementally by storing only RAM pages and disk sectors that have changed since the last checkpoint (see Chapter 3.3.1). Our design keeps this concept unchanged.

Deduplicated RAM pages and disk sectors as well as maps of the RAM and disk states for reversing the deduplication (see Chapter 3.3.1) are sent to a database for storage and later retrieval.

4.1.1 Asynchronous processing

Baudis' [6] design performs the following steps during the VM downtime:

1. iterate over all dirty RAM pages and disk sectors
2. calculate hash values for all dirty elements
3. deduplicate RAM pages and disk sectors
4. send the data to the database

In Chapter 3.4 we showed that copying all dirty RAM pages and disk sectors to a separate area of RAM is considerably faster than performing deduplication and then writing to a database.

Baudis [6] suggest using a copy-on-write mechanism to perform such a copy while the VM has already resumed execution. For simplicity, our design uses the VM downtime to copy all modified data and then immediately resumes the VM. The data can then be deduplicated and stored asynchronously.

Figure 4.1 compares the previous design with our new design. This change moves most of the slow work (deduplication and database access) out of the critical path (i.e., the VM downtime) and replaces it with a very lightweight operation, `memcpy`.

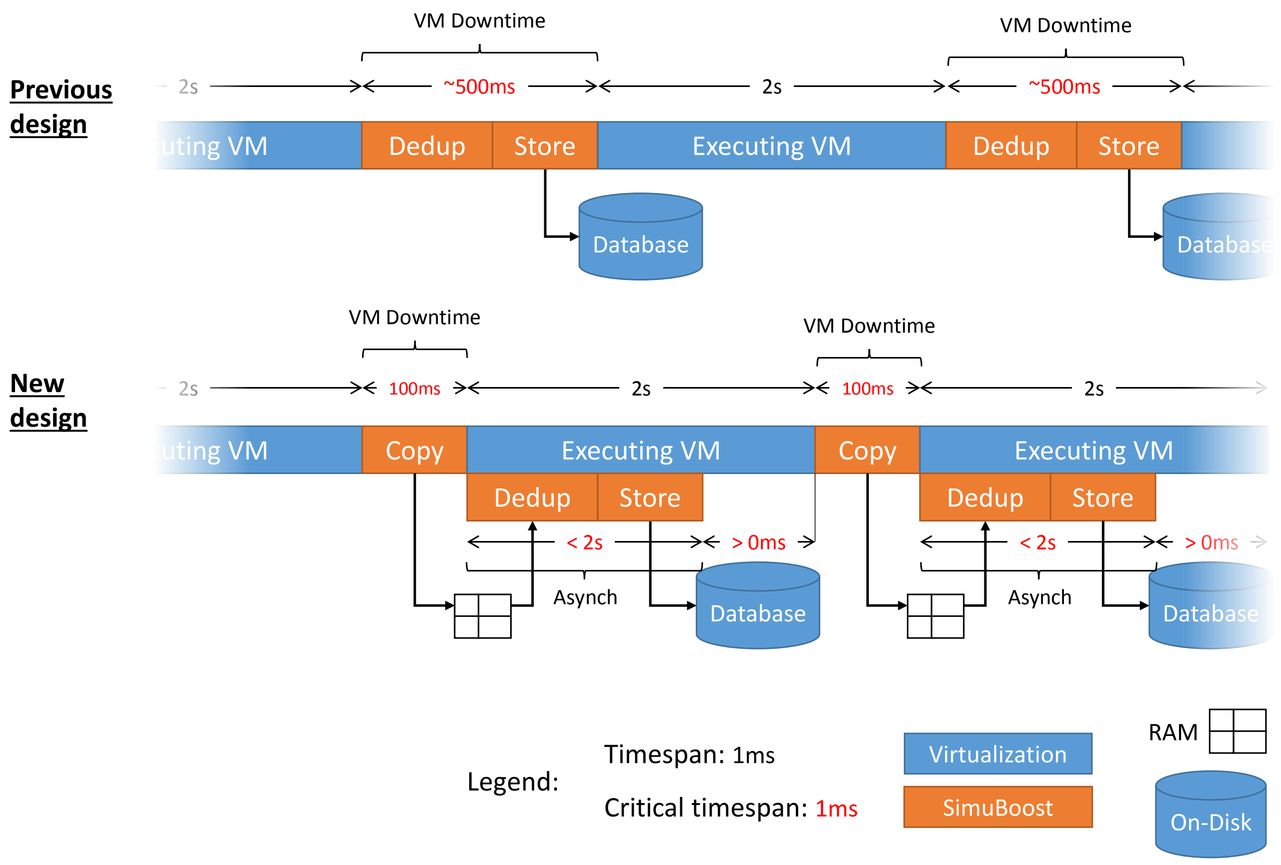

Figure 4.1: The previous design [6] halted the execution of the VM to deduplicate dirty RAM pages and disk sectors and then send them to the database for storage. The new design instead copies the dirty pages and sectors to a separate region in RAM and then resumes the execution of the VM. The deduplication and database storage now happens asynchronously.

Performing the checkpoint processing asynchronously introduces a new critical path: The deduplication and storage should be completed before the next checkpoint is created. There are a number of possible strategies for handling cases where the previous checkpoint has not finished processing when the next one is due.

We could accumulate checkpoint data for processing in a queue and handle it using the producer-consumer pattern. This would ensure the checkpoint intervals and VM downtimes remain uniform. If the problem persisted this might cause a backlog of unprocessed checkpoints to form.

Alternatively, we could pause the VM and wait for the previous checkpoint to finish processing. This would prevent a backlog from building up. However, the increased downtime would have negative side effects: The speedup predicted by the *SimuBoost* paper [5] would be lowered, interactive usage of the VM by human operators might be impaired due to the VM becoming unresponsive at unpredictable times, and network connections from inside the VM might be dropped due to timeouts.

Finally, we chose an approach that prioritizes keeping the downtime short. We keep the VM running and delay the next checkpoint until the previous one has finished processing. This sacrifices the uniformity of checkpoint intervals for relatively constant downtimes. This approach also acts as a self-regulating mechanism that effectively reduces the checkpointing frequency when the underlying system is unable to cope with the volume of data captured. While it avoids queuing unprocessed checkpoints, the amount of data changed within a single checkpoint created after a longer interval may be higher.

The asynchronous checkpoint processing can potentially compete for resources with the execution of the VM. This can be easily mitigated by using a multi-core system (*SimuBoost*'s design is only intended to handle single-core VMs [5] leaving the other cores free to use) and using separate disks for storing the VM's disk image and checkpoints.

4.1.2 Storage

Chapter 3.4 indicates that while common databases provide interfaces suitable to *SimuBoost*'s architecture, such as key-value stores, their performance characteristics are unsuitable. We determined that writing to an append-only flat file instead would achieve the desired performance.

Chapter 3.3.1 introduced the concept of RAM and disk state maps. These data structures store deduplication keys for pages/sectors rather than the actual data.

Baudis [6] used the hashes of pages/sectors as keys, allowing for easy deduplication. A key-value store provides the means for storing the pages/sectors and addressing them by their key.

If we were to apply the concept of flat file storage directly to the existing design we would need to introduce an additional index data structure. This structure would keep track of the offsets within the flat file that correspond to specific hash keys. The index would need to be stored alongside the flat file. Figure 4.2 illustrates such a set up.

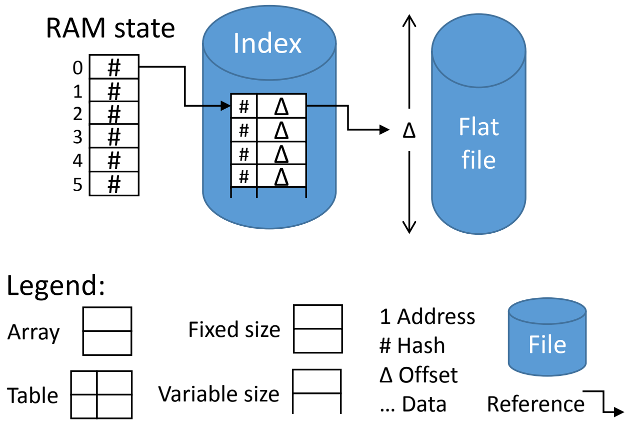

Figure 4.2: The RAM state map is a fixed-size array that stores the hash key for every VM RAM page. The index is an associative array mapping hash keys to file offsets. The flat file contains all RAM pages in a continuous stream. The process is equivalent for disk sectors.

We simplify this design by directly storing the offsets instead of the hashes in the state maps.

The hash table still needs to be kept in memory to be able to perform deduplication. It makes it possible to quickly determine whether a specific entry has already been encountered.

Our design does require the ability to reconstruct checkpoints after the system was offline and volatile memory has been lost (see Chapter 3.4). However, we do not need to continue recording checkpoints after the system was restored. Therefore, there is no need to persist the hash table on-disk for later usage by the read phase.

Figure 4.3 illustrates the modified checkpoint writing process.

Elements (RAM pages or disk sectors) that were modified since the previous

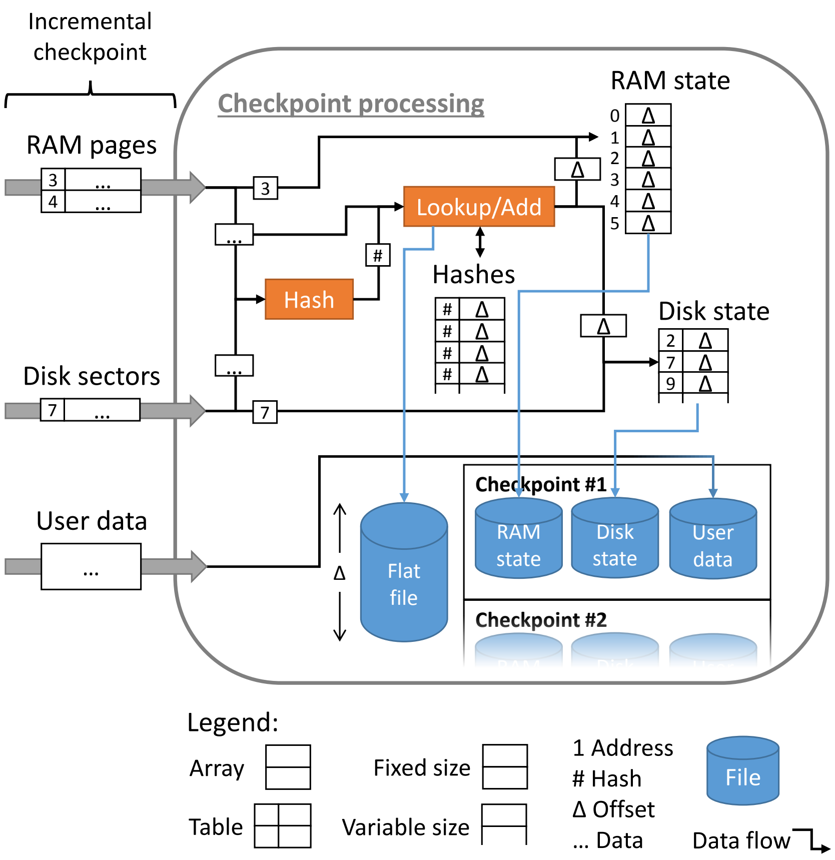

Figure 4.3: The checkpointing process deduplicates RAM pages and disk sectors and stores them in a flat file.

checkpoint are hashed for deduplication. If a new element is added it is appended to the flat file. The offset the element was stored at is recorded at the appropriate index in the state map and added to the hash table. If, on the other hand, an existing element is encountered its offset is retrieved from the hash map. This offset is then recorded at the appropriate index in the state map.

Once all elements of a checkpoint have been processed and applied to the state maps a copy of these maps is stored on-disk. Creating these per-checkpoint file copies of the state maps allows them to be easily restored when a checkpoint needs to be loaded again. It also frees the in-memory representation of the state map for further modification by the next incremental checkpoint.

Our design also includes a per-checkpoint storage space for "user data". This is intended to hold any additional data that the VMM may require to completely restore a VM's state. This generally comprises various device states (see Chapter 2.2) and VM metadata such as RAM size. Baudis [6] showed such data to have a negligible size compared to other checkpoint data. Therefore, we apply no deduplication to it and simply store the data unprocessed.

Baudis' [6] design treats RAM pages and disk sectors in isolation from each other. Park et al. [32] show that many RAM pages in a VM's working set do not need to be captured for a checkpoint because they have already been persisted to non-volatile storage (i.e., disk sectors). The design presented by Park et al. detects such deduplication opportunities by monitoring the state of the guest operating system's page cache.

We intend to achieve a similar deduplication of RAM pages against disk sectors. However, unlike [32] our design does not use any knowledge of the inner workings of the guest operating system. Instead we simply compare the hashes of dirty sectors and pages against each other.

The size of a RAM page usually does not match the size of a disk sector. For example, on the x86 architecture the page size is 4 KiB while the traditional size for disk sectors is 512 bytes. In order to deduplicate these differently sized data structures against each other we:

1. hash each RAM page as a whole for deduplication against other RAM pages,
2. split each RAM page into equally sized parts (4 KiB / 8 = 512 Bytes) and
3. hash each of these parts separately for deduplication against disk sectors.

Due to the flat file and offset storage scheme introduced in Chapter 4.1.2 offsets can reference parts of the file as disk sectors while another offset references the same part as a RAM page. See Figure 4.4 for an illustration.

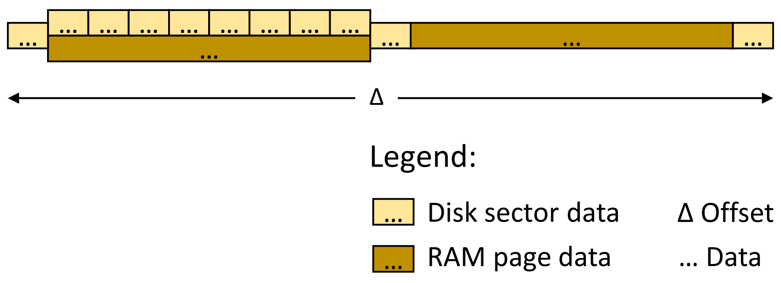

Figure 4.4: We store deduplicated RAM pages and disk sectors in a continuous flat file. An offset within the file can point to a RAM page, a disk sector or both.

4.2 Distribution

SimuBoost's design calls for a mechanism for distributing simulation jobs across a cluster (see Chapter 2.4.2). We use the following terminology for our distribution design:

Server The machine running the hardware-assisted VMM producing checkpoints. This machine is not part of the cluster.

Nodes The machines running simulation instances consuming checkpoints. These machines are part of the cluster.

We distinguish between two main use-cases:

Immediate parallelization Simulations are dispatched to worker nodes as soon as a checkpoint from the virtualization has been created and processed. Virtualization and simulation run in parallel.

Deferred parallelization The virtualization is executed and checkpoints are recorded without dispatching any simulations. The data is stored for parallel execution on a cluster at a later point in time. The system may be taken offline in the meantime. The recorded data may also be transferred to a different machine.

4.2.1 Transmitting checkpoints

The prototype presented by Baudis [6] stores the recorded checkpoints in a database. This allows nodes to retrieve checkpoints via the database's network interface.

First, the state maps for the specific checkpoint pointing to the deduplication keys (see Chapter 3.3.1) are loaded from the database. Then every RAM page and disk sector referenced in the maps is request from the database.

To avoid having to issue an individual database request for every single RAM page (e.g., 262144 requests for 1GiB with 4KiB page size) and incur the associated network roundtrip costs Baudis [6] uses:

Pipelining Combining multiple retrieval requests in a single database query.

Caching Keeping a cache of elements loaded from the database on the nodes.

In Chapter 4.1.2 we introduced flat file storage as an alternative to classic database systems. Since we can no longer use a preexisting network interface we need to design our own communication scheme. We considered a number of possibilities.

We could implement functionality equivalent to the database chosen by Baudis [6]. The node request individual RAM pages and disk sectors from the server. Pipelining and caching serve to reduce the number of requests sent over the network. This option allows the network load to benefit from the checkpoint deduplication performed earlier (see Chapter 4.1.2).

we could instead keep track of each node's cache state on the server. The server can then pair nodes with checkpoints in a way that minimizes the amount of uncached elements that need to be transmitted. This option reduces the network load by grouping the execution of similar checkpoints together.

Finally, we could reconstruct entire snapshots of the VMs' state on the server and send them to the nodes en bloc. This option minimizes the number of network roundtrips required as well as the amount of additional state that needs to be held.

We chose to reconstruct entire snapshots on the sever because this design makes no assumptions regarding which checkpoints have how much state in common with other checkpoints. It minimizes the amount of logic and additional RAM required on the nodes. It also does not require fine-grained control over the distribution of jobs to specific nodes. This makes it easier to use existing cluster management tools.

However, this design does require a high-bandwidth network connection between the server and nodes: For each simulation the system needs to transfer data equivalent to at least the VM's entire RAM size.

4.2.2 Loading checkpoints

Figure 4.5 illustrates process of loading checkpoints and reconstructing an entire snapshot of the VM's state. Compare this to the checkpoint creation process illustrated in Figure 4.3.

The first step for loading a checkpoint in our design is to retrieve the RAM and disk state maps (see Chapter 3.3.1) for the corresponding checkpoint from on-disk files.

These state maps then provide the offsets needed to retrieve the actual data from the flat file. By seeking in the flat file and copying the stored RAM pages to a contiguous set of memory we can reconstruct the VM's RAM.

When loading the VM's disk state our design also intends for the reconstructed snapshot to contain all relevant sectors rather than only the ones modified since the last checkpoint. However, due to the very large size of disk storage compared

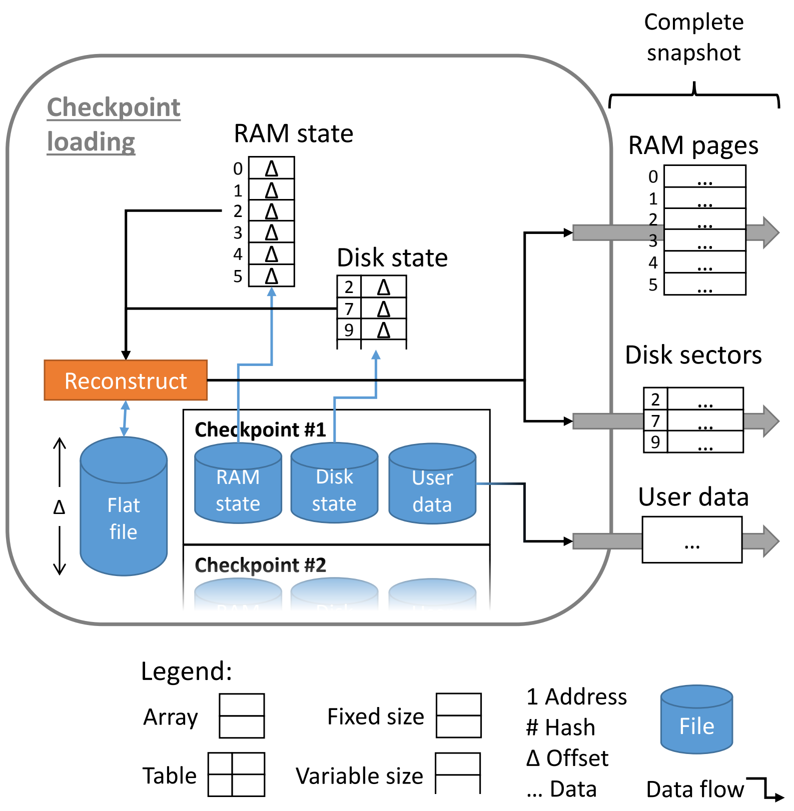

Figure 4.5: Checkpoints are reconstructed from RAM and disk states pointing to flat file offsets.

to RAM, our design transfers a sparse set of sectors to the nodes rather than a complete image as it used for the RAM state.

This sparse set contains all sectors that were modified since the start of the checkpointing process but not necessarily all sectors contained in the original VM's disk image. Therefore, all cluster nodes must have access to an identical copy of the disk image used by the server when originally creating the checkpoints.

4.3 Conclusion

The design we presented in Chapter 4.1.1 reduces the VM downtime caused by checkpointing by performing the deduplication asynchronously. This contributes to the speedup predicted by the original *SimuBoost* paper [5].

The flat file storage scheme introduced in Chapter 4.1.2 improves the disk write speed. The deduplication of disk sectors against RAM pages further reduces the amount of data that needs to be stored. These improvements help ensure the asynchronous portion of the checkpointing process can be completed before a new checkpoint is due.

The checkpoint distribution system designed in Chapter 4.2 allows us to expand the checkpointing-only prototype to a full cluster-based solution. This enables us to evaluate the complete *SimuBoost* concept and to observe the interactions between the different components of the system.

Chapter 5

Implementation

This chapter details how we implemented the design decisions from Chapter 4 and what difficulties we came across in doing so.

We use *QEMU* [2] as the VMM generating the checkpoints. This allows us to use the existing implementation from Baudis [6] as a starting point.

In Chapter 4.1.2 we chose a flat file as an alternative to using a database such as *MongoDB* for storage. In Chapter 4.2.1 we noted that we still need a network interface for retrieving checkpoints on remote nodes.

We decided to implement our flat file scheme as an extension to *Simutrace*'s existing storage server (see Chapter 2.4.1). This allows us to use a preexisting network infrastructure rather than having to develop our own.

Simutrace's storage server supports communication via shared memory. This means that sending checkpointing data to the server involves a `memcpy`. Since our design for asynchronous processing of checkpoints in Chapter 4.1.1 already requires such a copy step, we decided to move the deduplication logic from *QEMU* to the storage server. This way *QEMU* can quickly offload all dirty RAM pages and disk sectors to the storage server, which then, running as a separate process, can deduplicate the data in parallel to the resumed VM.

We also use *QEMU* to run the simulations on the nodes. This simplifies loading the checkpoints since no additional steps need to be taken to convert the VM metadata used by the VMM (referred to as "user data" in our design) to a different format.

In Chapter 4.2 we describe a concept for distributing simulation jobs across nodes in a cluster. We use the *Slurm* [48] workload manager to manage the distribution of these jobs. *Slurm* is widely used in computer clusters, making it a representative choice for real-world usage.

Figure 5.1 illustrates the communication between the components of our implementation.

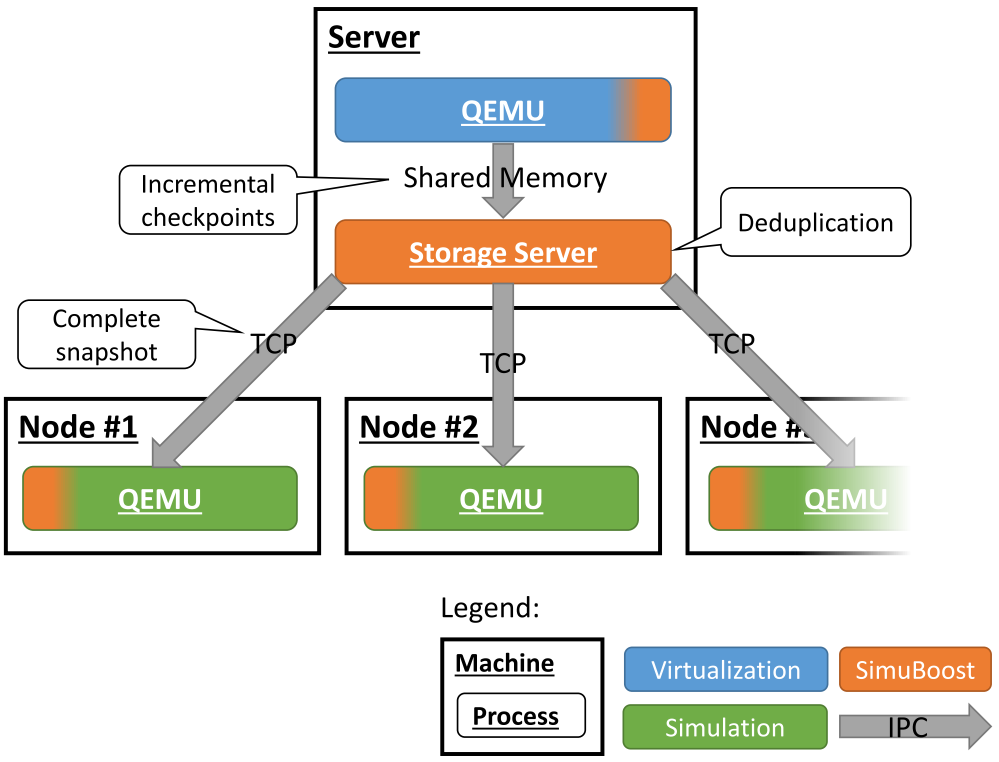

Figure 5.1: The *QEMU* instance running on the server is modified to send incremental checkpoints to the storage server using shared memory. The storage server performs deduplication and writes the data to disk. The *QEMU* instances running on the cluster nodes are modified to retrieve complete snapshots reconstructed from the checkpoints by the storage server via TCP/IP.

5.1 Virtual Machine Monitor

Our design calls for a Virtual Machine Monitor (VMM) for executing the hardware-assisted virtualization. This VMM is to be extended with our checkpointing design.

We based our implementation on the modified version of *QEMU 1.5.1* presented by Baudis [6]. In principal, a different VMM could have been used if it supports:

- running VMs with hardware acceleration,
- pausing and resuming of VMs,
- capturing and restoring VM device states and
- tracking modifications to guest RAM pages and disk sectors.

5.1.1 Control

"When QEMU is running, it provides a monitor console for interacting with QEMU." [36] Baudis' [6] implementation added the following new commands to this monitor:

- `start-cp` starts a checkpointing thread, which runs for the entire duration of the checkpointing. It pauses the execution of the VM in regular intervals to capture its state and then resumes it.
- `stop-cp` stops the checkpointing thread.
- `load-cp` loads the state of a previously captured checkpoint into the virtual machine.

We extended these commands to take additional arguments such as the total number of checkpoints to record before terminating and where to store/load the checkpoint data.

The monitor can be accessed from within *QEMU* or redirected to `stdin/stdout`. We redirected the monitor in order to be able to automate the creation of checkpoints on the server and the loading of checkpoints on the nodes.

5.1.2 Dirty tracking

QEMU's snapshot feature does not support incremental RAM snapshots (see Chapter 2.3). However, *QEMU*'s VM migration feature uses a data structure called `migration_bitmap`. Whenever a write access to a guest page occurs *QEMU* sets the corresponding bit in the migration bitmap. This makes it possible to quickly identify all pages that have been modified within the last interval.

We enable this feature when checkpointing is active. During the downtime we iterate over all pages indicated by the bitmap and copy them for later processing. We then reset the bitmap before resuming the VM's execution. For the first checkpoint we consider all bits in the bitmap to be set. This ensures that the entire snapshot of the VM's RAM is available.

As mentioned in Chapter 4.2.2 our checkpointing design does not require checkpoints to store an entire snapshot of the VM's disk storage. Instead, only the sectors that have been changed since the checkpointing process was started need to be handled.

QEMU's default disk image format, `qcow2`, supports sparse images (storing only sectors that were actually written to) and copy-on-write (COW) for snapshots [35]. We use these features as follows:

1. We create a `qcow2` base image on the server containing the workload to be simulated.
2. We copy the image to each node.
3. When we are ready to activate the checkpointing feature we launch *QEMU* with the `-snapshot` option, telling it to redirect all further modifications to a temporary overlay file.
4. We configure *QEMU* to mark any modified sectors in the overlay image as dirty.
5. We reset the dirty flags after each checkpoint.

This greatly simplifies tracking sectors that were modified during the execution of an interval. We only have to iterate over the sectors allocated in the overlay image when checking for dirty flags. Additionally, the initial checkpoint only needs to include any sectors that were modified since the VM was started rather than all allocated sectors.

Figure 5.2 illustrates the relation between the base image, the temporary overlay image and the data stored for checkpoints.

5.1.3 Instruction count

Our design does not cover deterministic record and replay of events as it is required for a full implementation of the *SimuBoost* concept [5]. Therefore, execution in VMs loaded from checkpoints is likely to quickly diverge from the original execution.

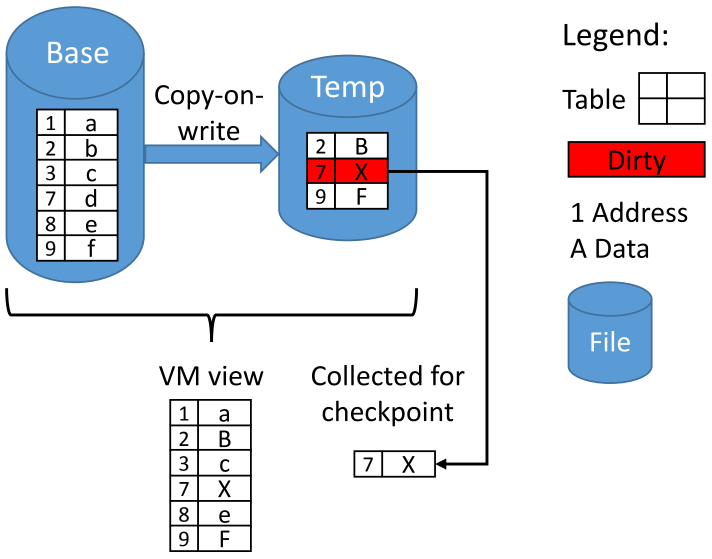

Figure 5.2: *QEMU* supports sparse disk images that only contain sectors that were actually written to. The `-snapshot` option tells *QEMU* to redirect additional modifications to a temporary overlay image. When searching for dirty sectors to collect for a checkpoint only sectors contained in this overlay image need to be considered.

QEMU running in *TCG* mode could act as a simulation system that can be extended with instrumentation code as described in 2.4. However, *QEMU* currently does not support loading snapshots in *TCG* mode that were originally created in *KVM* mode.

In order to be able to evaluate the *SimuBoost* concept we need to approximate the execution of full-system simulations of each checkpoint interval. Executing instructions in a simulator generally takes much longer than execution with hardware-assisted virtualization. We decided to measure the number of CPU instructions executed within the VM during each checkpoint interval. We use this data as a basis for predicting whether the execution times of the simulations is roughly constant for constant virtualization interval lengths.

We modified the Linux *KVM* kernel module [49] to count the number of CPU instructions executed while in guest mode. We modified the `vmx_vcpu_run()` function, which manages the context switch between host and guest mode. We use the MSRs (model specific registers) for fixed-function performance counters on the Intel *Core* architecture to enable the retired instruction counter [50]. We capture the state of the counter immediately before *KVM* enters guest mode and immediately after it leaves it again, accumulating the difference.

We measured the number of instructions executed between capturing the performance counters and actually switching from and to guest mode and determined the value to be constant at 69. By subtracting this value we gain an accurate measure of the number of instructions executed by the VM.

Since *KVM* is a kernel module it cannot directly use the file system. Therefore, we use *KVM*'s `ioctl` interface to pass the results back to *QEMU* in userspace.

5.2 Storage

In Chapter 4.1.2 we chose a flat file as an alternative to using a database such as *MongoDB* for storage. In Chapter 4.2.1 we noted that we still need a network interface for retrieving checkpoints on remote nodes.

We decided to implement our flat file scheme as an extension to the *Simutrace* storage server (see Chapter 2.4.1). This allows us to use a preexisting network infrastructure rather than having to develop our own. The *Simutrace* storage server is designed to store traces generated during full system simulations. It enables the selection of different storage backends that determine how the data is stored on-disk.

We removed the deduplication and caching logic implemented in *QEMU* by Baudis [6] and implemented it as new storage backend in the storage server instead. This makes it possible to reuse the deduplication code with VMMs other than *QEMU*.

Baudis [6] uses *CityHash* [51] as a fast, low-collision hash function for deduplication of RAM pages and disk sectors. We use *FarmHash* [52] instead, which has been designated as the official successor to *CityHash*.

5.2.1 Timeline

Clients can communicate with the storage server using either shared memory or a TCP/IP connection.

We use the shared memory functionality for communication between the *QEMU* virtualization instance and the storage server. This removes the overhead of a network stack when writing checkpoint data, the most time-sensitive part of our design.

We use TCP/IP for communication between the storage server and the *QEMU* simulation instances running on worker nodes. Using shared memory is not suitable here because the nodes are generally physically separate machines.

Figure 5.3 illustrates how we process checkpoint data in parallel.

1. The checkpointing thread in *QEMU* pauses the execution of the VM in regular intervals (see Chapter 5.1.1).
2. We copy any dirty (i.e., modified) RAM pages and disk sectors to the shared memory used for communication with the storage server.
3. We use *QEMU*'s built-in snapshot feature to capture any additional device states and VM metadata.
4. We resume the execution of the VM.
5. We "send" the collected data to the storage server. By virtue of using shared memory this happens basically instantly, only requiring a buffer to be exchanged.
6. Our new storage backend in the storage server begins deduplicating and storing the checkpoint data.
7. The checkpointing thread in *QEMU* blocks until the storage server finishes processing the checkpoint.
8. The checkpointing thread sleeps the time remaining until the next checkpoint is due.

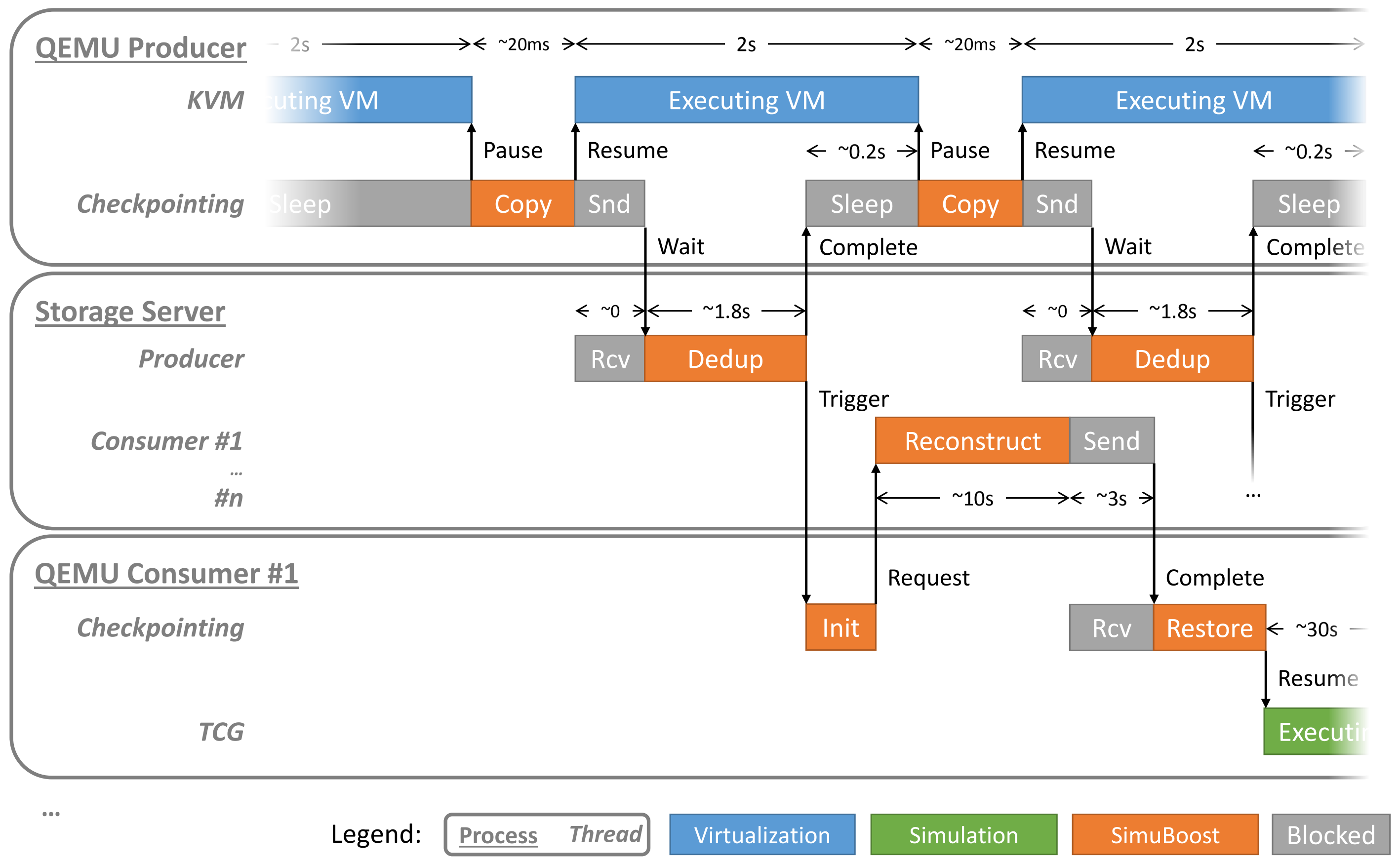

Figure 5.3: We pause the execution of the VM in regular intervals for short periods. The storage server performs deduplication in parallel to the execution of the VM. Simulation nodes are served by separate threads in the storage server, which reconstruct the checkpoints.

If the deduplication process in the storage server takes longer than the intended checkpointing interval the blocked checkpointing thread causes the next checkpoint to be delayed, as intended by the design in Chapter 4.1.1.

Each simulation node is served by a separate thread on the storage server. This makes it possible to reconstruct the checkpoints and send them over the network for execution in parallel.

5.2.2 Streams

The *Simutrace* storage server uses data structures called *streams* for communication between the server and clients. Each *stream* represents a continuous set of elements of a specific type (e.g., trace entries or in our case RAM pages).

When a client connects to the storage server it uses a *storage specifier* to control how and where the data is persisted. The *storage specifier* consists of a prefix identifying the storage backend to use and the file path controlling where the backend places its files.

Our new backend is identified by the prefix `simuboost`. It supports the following stream types:

RAM pages This stream contains pairs of RAM page indexes (RAM addresses divided by the page size) and the contents of the corresponding pages.

Disk sectors This stream contains pairs of disk sector indexes (disk addresses divided by the sector size) and the contents of the corresponding sectors.

User data This stream contains an undifferentiated set of bytes. We use it store *QEMU*'s built-in snapshots capturing any additional device states and VM metadata.

Figure 5.4 illustrates how we represent checkpoints in these streams.

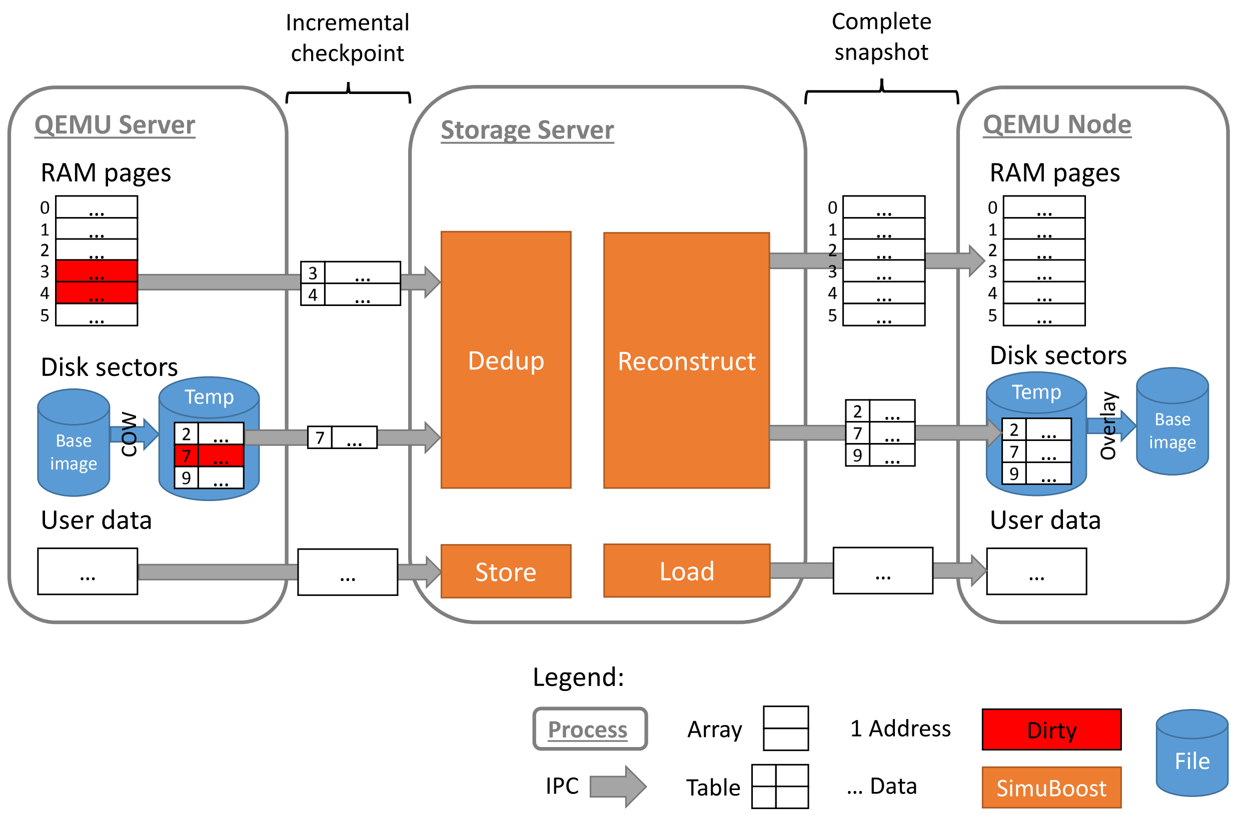

Figure 5.4: We transfer checkpoints as streams of RAM pages and disk sectors. When sending data from the server creating the checkpoints we only transfer dirty pages and sectors. When transferring checkpoint data to nodes for simulation we send complete snapshots.

Simutrace streams are split into equally-sized segments. The default segment size is 64MiB. Data from a stream is queued until a segment has filled up before it is sent to the storage server.

Our current implementation processes incoming segments in the storage server synchronously. This also causes the sending thread in *QEMU* to block until the processing is complete. This does not block the execution of the VM since sending segments is triggered on the checkpointing thread in *QEMU* after resuming the VM's execution (see Chapter 5.2.1).

However, if the amount of data accumulated by *QEMU* for a checkpoint exceeds the size of a single segment this causes segements to be sent and execution to be blocked before the VM is resumed. This would negate our parallelization efforts and severely impact the downtime.

Therefore, we changed the segment size to accommodate the upper limit of the expected number of dirty RAM pages for one checkpoint interval. Based on the figures provided by Baudis [6] we chose a segment size of 256MiB. Optimizing this value for different workloads may require a certain level of a priori knowledge.

The first checkpoint is an intentional special case. Since it always captures all RAM pages it is usually guaranteed to exceed the segment size. Keeping the VM paused while this initial checkpoint data is deduplicated avoids overloading the system with a backlog of data right at the beginning. Since this is a one-time delay rather than occurring once for every checkpoint it does not contribute to the slowdown significantly.

5.2.3 Data structures

We implemented the deduplication index described in Chapter 4.1.2 using the *C++11* data structure `std::unordered_map`. This data structure provides a hash table with an amortized $\mathcal{O}(1)$ runtime for insert and lookup operations.

During early benchmarks we noticed periodic peaks in the checkpoint processing time correlating with the workloads being executed in the VM. We were able to determine that these peaks were caused by `std::unordered_map` reaching an internal element count limit which triggered the creation of additional buckets. The resulting rehashing of all previously inserted keys caused the process to stall for up to 10s.

Switching to `std::map`, a balanced binary tree, would avoid these periodic delays. However, the accompanying $\mathcal{O}(n \log n)$ cost for inserts makes this choice prohibitive for our insert-heavy usage pattern.

We were able to work around the issue by preallocating a sufficient number of buckets for the `std::unordered_map`. Choosing an optimal value for the number of buckets to preallocation requires a priori knowledge of the number of unique (after deduplication) pages and sectors produced by a given workload. However, conservative estimates should be sufficient for practical application.

5.2.4 IO

In Chapter 4.1.2 we described a scheme for writing the deduplicated RAM pages and disk sectors to an append-only flat file. This serves to optimize IO throughput by minimizing seeking. We use memory-mapped files to access the flat file.

When writing checkpoints this allows us to avoid issuing a system call for each RAM page or disk sector to be stored, using simple `memcpy` calls instead.

When reading checkpoints any sectors that have not been swapped out to the backing flat file by the operating system because of memory pressure will be accessible instantly without having to issue a system call.

5.3 Cluster management

In Chapter 4.2 we describe a concept for distributing simulation jobs across nodes in a cluster.

First, we need a tool that generates simulation jobs in response to the storage server (see Chapter 5.2) finishing processing individual checkpoints.

We use the Linux daemon *inoticoming* [53] as a simple solution. *inoticoming* monitors the file system for newly created files matching a specified pattern and executes a custom command whenever one is found. We configured *inoticoming* to detect the per-checkpoint state map files created by the storage server.

Next, we need a tool for managing the actual cluster. We use the *Slurm* [48] workload manager to manage the distribution of the jobs created by *inoticoming*. *Slurm* is widely used in computer clusters, making it a representative choice for real-world usage.

In addition to using *QEMU* as our VMM for generating checkpoints (see Chapter 5.1) we use *QEMU* to run the simulations. This simplifies loading the checkpoints since identical file formats are used for storing snapshots. Future implementations that add full-system simulation capabilities to the simulation stage could operate *QEMU* in *TCG* [33] mode. These implementations would benefit from the virtual hardware present in the virtualization and simulation stages being very similar (see Chapter 2.3).

On the nodes *Slurm* starts instances of our modified *QEMU* version. These instances then connect with the storage server and retrieve the data for the checkpoint they were assigned. Each node has access to a copy of the VM's original disk image (see Chapter 5.1.2).

Chapter 6

Evaluation

In order to evaluate the implementation presented in Chapter 5 we performed a number of measurements.

We analyzed the characteristics exhibited by different workloads during checkpointing. This provides information on how different real-work use patterns impact the performance of *SimuBoost*.

We measured the performance of our checkpointing implementation using criteria like the VM downtime caused by checkpointing and the deduplication rate. This data allows us to check whether the requirements set out in Chapter 3.1 have been met.

We also measured the performance of our implementation when loading checkpoints, i.e., the time required to distribute checkpoint data to simulation nodes. This helps determine the kind of storage and network hardware required for our use-cases.

Finally, we measured the total execution time of the distributed simulations. This shows us the actual speedup achieved. Comparing these values with the predictions made by the formal model presented in Chapter 3.2 allows us to assess the viability of *SimuBoost*.

6.1 Methodology

As mentioned in Chapter 4.1 our implementation does not allow checkpoints created by *QEMU* in *KVM* mode to be resumed in *TCG* mode. This restriction is not due to a fundamental limitation but rather a thus-far unimplemented feature of *QEMU*. For example, *V2E* [54] presents such an implementation.

In order to evaluate our implementation without such a *KVM*-to-*TCG* solution we:

1. completely transfer the reconstructed checkpoint data to the simulation nodes. This ensures our evaluation accurately captures the IO interaction of writing and reading checkpoints as well as any possible network contention.

2. artificially mark the nodes as busy for a precomputed time period. Rittinghaus et al. [5] measured the slowdown factor of full-system simulation with *QEMU* compared to hardware-assisted virtualization with *QEMU* to be 31 on average. Therefore, we use $31 \cdot L$ as the artificial busy time for each node, with L being the configured interval length.

In Chapter 4.2 we introduced two main use-cases for our design:

Immediate parallelization Simulations are dispatched to worker nodes as soon as a checkpoint from the virtualization has been created and processed. Virtualization and simulation run in parallel. To evaluate this use-case we schedule a *Slurm* job for each checkpoint as soon as it is created.

Deferred parallelization The virtualization is executed and checkpoints are recorded without dispatching any simulations. The data is stored for parallel execution on a cluster at a later point in time. To evaluate this use-case we record all checkpoints and then shut down the storage server and purge the operating system's file system cache (e.g., by executing `echo 3 > /proc/sys/vm/drop_caches` on Linux). This recreates the conditions after a cold-start and ensures our measurements are not distorted by caching effects that would not be applicable in a real-world deferred scenario. Finally, we restart the storage server and schedule *Slurm* jobs for all checkpoints in one go.

During the execution the individual components (our modified *QEMU* and the storage server) each write statistics to their own comma-separated values (CSV) files. We use shell scripts to orchestrate the execution of the components and collect all relevant CSV files at the end.

6.2 Evaluation setup

We installed our modified *QEMU* version, the storage server and *inoticoming* on our designated server computer. We used a set of shell scripts to automatically launch and connect the three processes.

We used a preexisting *Slurm* cluster. We installed a minimalistic *SimuBoost* client on the nodes. This client downloads a single checkpoint via the network and then terminates. We used a shell script to add the artificial delay described in Chapter 6.1. Figure 6.1 illustrates the relation between the components.

6.2.1 Hardware and OS

We ran our benchmarks with a single server and a cluster of 5 nodes. All computers were connected via a 10GBit Ethernet network. Table 6.1 lists the specifications of the individual computers. Table 6.2 lists the operating systems used.

	CPU	RAM	Disk
Server	4x Intel Xeon E5-2630	64GiB	256GB SSD
VM Guest	virtual single-core CPU	1 GiB	10GB virtual HDD
Node 1	Intel Xeon E3-1220	16 GiB	500GB HDD, 128GB SSD
Node 2	Intel Xeon E3-1230	16 GiB	128GB SSD
Node 3	Intel Xeon E3-1230	16 GiB	128GB SSD
Node 4	Intel Xeon E3-1230	16 GiB	128GB SSD
Node 5	Intel Xeon E3-1230	16 GiB	128GB SSD

Table 6.1: The server is a stand-alone machine connected to the cluster for our experiments. The cluster consists of 4 identical nodes and one head node hosting the *Slurm* controller.

	OS	Kernel version
Server	Linux Mint 17	3.13.0-45-generic
VM Guest	Linux Mint 17	3.13.0-24-generic
Nodes	CentOS 6.6	2.6.32-504.12.2.el6.x86_64

Table 6.2: The server is an updated Linux Mint 17 installation. The guest OS used for the workloads is a vanilla Linux Mint 17 installation. The cluster nodes share an identical CentOS 6.6 installation.

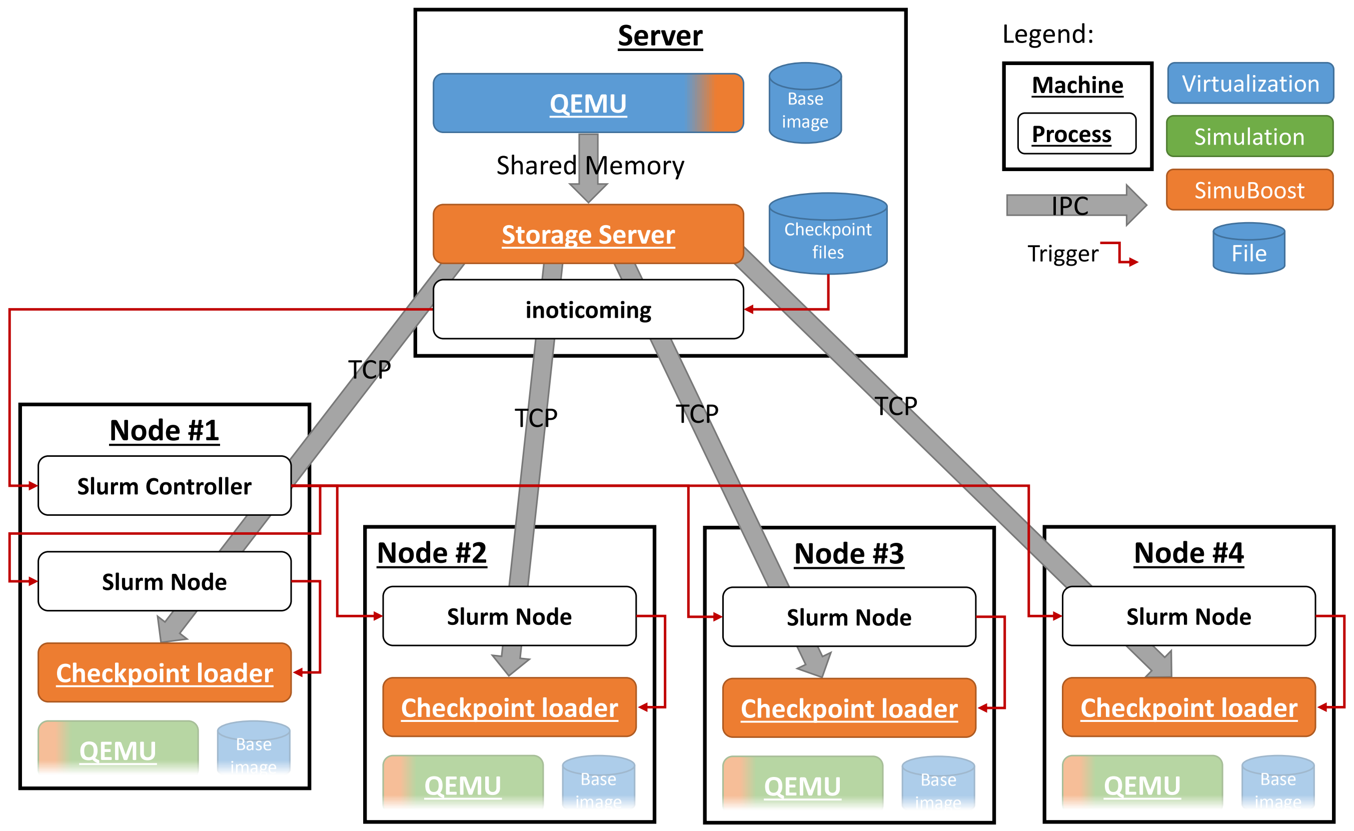

Figure 6.1: *inoticoming* runs on the server and detects checkpoint files created by the storage server. It sends jobs to the *Slurm* controller (`slurmctld`) running on the first node. The *Slurm* controller manages the distribution of the jobs across the Slurm nodes (`slurmd`). The checkpoint loader instances retrieve data from the storage server via TCP/IP.

6.2.2 Workloads

We chose the following workloads for our evaluation:

Kernel build We perform a full build of the Ubuntu kernel 3.13.0. Kernel builds tax both the system's memory and disk.

Bonnie++ benchmark We run `bonnie -x 1000` in an infinite loop. This benchmark measures a computer's hard disk performance. We execute it to simulate heavy disk IO.

STREAM benchmark We run `stream -W 1000` in an infinite loop. This benchmark measures a computer's RAM throughput. We execute it to simulate heavy memory usage.

These workloads cover a broad range of load behaviors and are closely aligned with the workloads used in baudis' [6] evaluation. Park et al. [32] and Liu et al. [24] use similar workloads to evaluate their checkpointing mechanisms.

Each workload includes an approximately 20s operating system boot phase. After booting the respective benchmark is executed automatically. The workloads are run for 10 minutes each, not including checkpointing-induced downtimes.

6.3 Results

In this chapter we present the results of our measurements. First, we compare the chekpointing-relevant characteristics of different workloads. Next, we present a number of performance metrics for our checkpointing implementation. Finally, we look at the cluster distribution and the resulting total runtime.

6.3.1 Workload comparison

The amount of data that has to be captured for a checkpoint depends on the number of RAM pages and disk sectors that were modified since the last checkpoint was taken (see Chapter 5.1.2).

These numbers vary based on the workload executed and the interval in which checkpoints were taken. They strongly influence the VM downtime as well as the runtime of the asynchronous processing (see Chapter 4.1.1).

Figure 6.2 and Figure 6.3 depict the number of dirty RAM pages and disk sectors respectively that accumulate for varying workloads. The first ten checkpoints show nearly identical values for all workloads. This is due to the 20 second boot phase mentioned in 6.2.

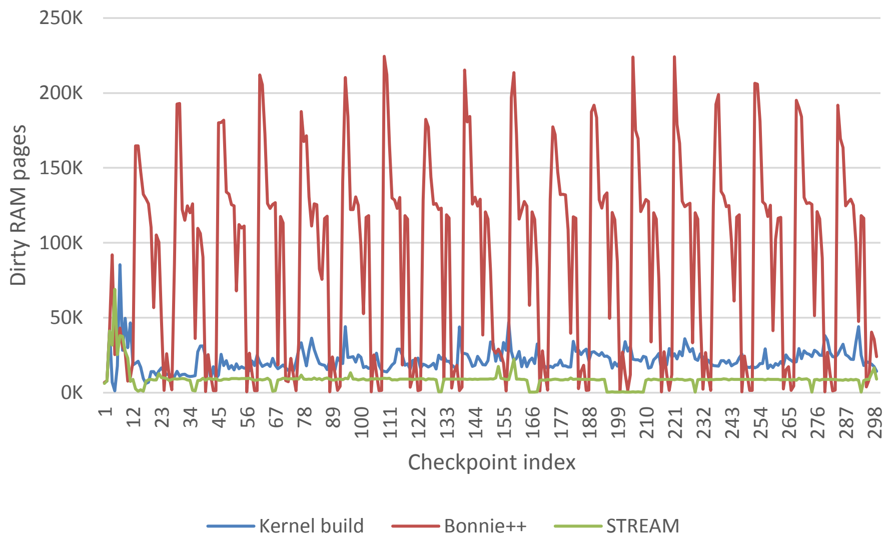

Figure 6.2: Dirty RAM pages per checkpoint for varying workloads with interval length 2s

The *kernel build* workload produces a relatively constant number of dirty pages and sectors during its run, averaging at 21,710 pages and 29 sectors per checkpoint. The *Bonnie++* and *STREAM* benchmarks show periodic spikes due to the same benchmark being executed repeatedly in a loop.

The *Bonnie++* benchmark's spikes repeat approximately every 32 seconds and reach 224,400 pages and 13,770 sectors per checkpoint. These values are significantly higher than the averages of 89,500 pages and 4,000 sectors. This benchmark produces a large number of both dirty pages and sectors since it generates a large set of random data in RAM and then writes it to disk.

The *STREAM* benchmark's considerably less pronounced spikes repeat approximately every 45 seconds, reaching 14,120 pages per checkpoint compared to an average of 8,900 pages. Although this benchmark is the most memory-intensive workload in our evaluation it produces the lowest number of dirty pages. This indicates that the same relatively small memory area is written to repeatedly within each individual checkpoint interval. There is no noteworthy disk activity causing dirty sectors in this benchmark.

Figure 6.4 charts the average dirty RAM pages per checkpoint that accumulate

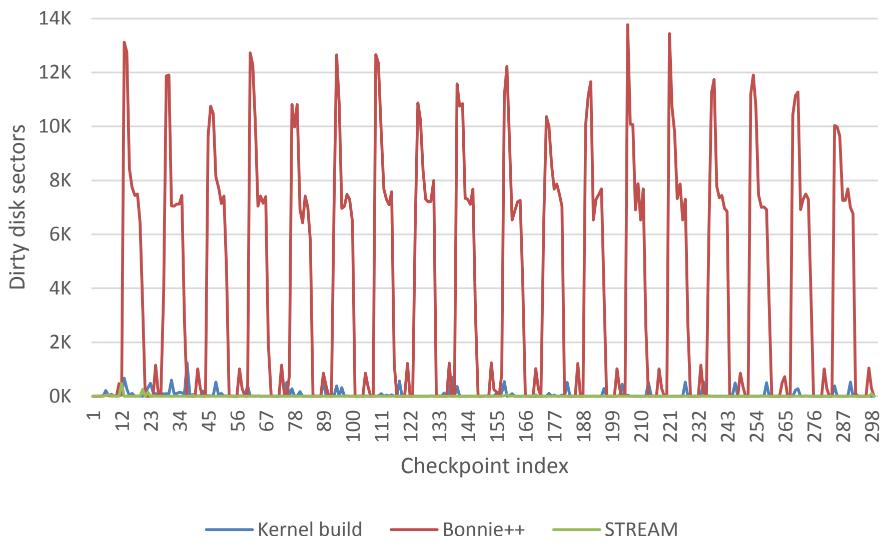

Figure 6.3: Dirty disk sectors per checkpoint for varying workloads with interval length 2s

for varying workloads and interval lengths. An increased interval length leads to a limited growth of the number of dirty pages. Initially, more modified data accumulates due to more instructions being executed per interval. Then a dampening effect sets in when the same page is overwritten multiple times during a single interval. For the same number of changes to a page fewer states are actually captured. This growth damping sets in slower for the *Bonnie++* benchmark than for other workloads since its larger data set is distributed over a larger number of distinct pages.

Figure 6.5 charts the average dirty disk sectors per checkpoint that accumulate for varying workloads and interval lengths. The *kernel build* workload produces relatively few dirty sectors. The *STREAM* benchmark, being a RAM throughput test, produces virtually none. The *Bonnie++* benchmark produces dirty sectors per checkpoint with a roughly linear relation to the interval length. Unlike with the RAM pages there does not seem to be a significant dampening effect. This may be attributed to the file system preferring to allocate previously unused sectors over recently freed space.

In Chapter 6.1 we assume the slowdown of switching from virtualization to simulation is a linear factor. We recorded the number of CPU instructions that were executed during each interval. Since the performance of simulations is mostly CPU-bound the variance of this value indicates how well our assumption holds for the chosen workloads (see Chapter 5.1.3).

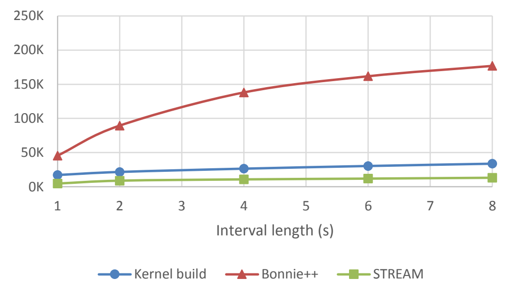

Figure 6.4: Average dirty RAM pages per checkpoint for varying workloads and interval lengths

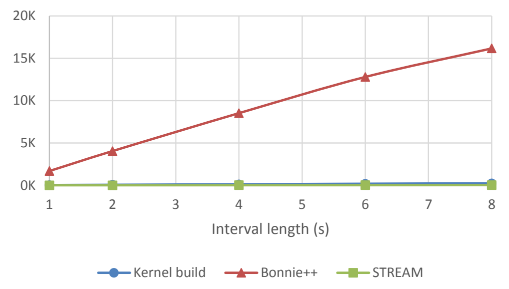

Figure 6.5: Average dirty disk sectors per checkpoint for varying workloads and interval lengths

Table 6.3 list the average the number of CPU instructions executed within the VM during a single interval for varying workloads. Figure A.1 provides a visualization of the number of CPU instructions over time for complete workload runs. The *kernel build* workload executes a relatively constant number of instructions during its run. However, the *Bonnie++* benchmark shows periodic spikes in CPU instructions that correlate with the number of dirty RAM pages charted in Figure 6.2. This variation in the number of CPU instructions that are executed per interval may invalidate our assumption of uniform interval simulation time made in Chapter 6.1 for some workloads.

Workload	Instruction count
Kernel build	7,281,163,691
Bonnie++	3,091,970,490
STREAM	5,253,607,823

Table 6.3: Average number of CPU instructions executed within the VM during a single interval for varying workloads with interval length 2s

6.3.2 Checkpointing performance

In this chapter we take a look at metrics relevant for the performance of creating checkpoints. First we look at the deduplication rate, then the amount of required disk storage space and finally the amount of time spent on processing during the VMs downtime vs. the amount of time spent on asynchronous processing.

Deduplication rate: We recorded the rate of deduplication achieved for each checkpoint. This indicates how effective our deduplication strategy is. It impacts both the speed with which checkpoints can be taken and the amount of on-disk storage required (see Chapter 4.1.2).

Table 6.4 lists the deduplication rates for varying workloads with interval length 2s. We split the workload runs into three distinct phases:

1. The OS boot process, which is the same for all workloads,
2. the first run-through of a benchmark and
3. all subsequent repetitions of the benchmark. This is combined with 2. for the *kernel build* workload because it is not repeated in a loop.

	RAM			Disk		
Workload	Boot	First pass	Rest	Boot	First pass	Rest
Kernel build	24%	17%		64%	76%	
Bonnie++	23%	67%	74%	78%	99%	97%
STREAM	21%	74%	86%	78%	89%	93%

Table 6.4: Deduplication rates during different phases for varying workloads with interval length 2s

The overall deduplication potential is significantly higher for the *Bonnie++* and *STREAM* benchmarks than for the *kernel build* workload. After a largely identical boot phase the deduplication rate for the first run-throughs of the benchmarks is approximately 50% higher than for the *kernel build*, presumably due to the synthetic nature of the workloads. The subsequent run-throughs show an additional 10% increase in deduplication rate due to similar or identical data being generated.

Figure A.2 and Figure A.3 provide a visualization of the deduplication rate over time for complete workload runs. The benchmarks once again exhibit periodic spikes correlating with the benchmark loop.

Figure 6.6 breaks down the disk sector deduplication per checkpoint for the workload *kernel build* into not-deduplicated, deduplicated against other disk sectors and deduplicated against RAM pages (see Chapter 4.1.2). A large percentage of the disk sectors were deduplicated against RAM pages. This is because, data that is to be written to the disk is usually stored in RAM (e.g., in a file system cache) first.

Figure 6.7 and Figure 6.8 plot the average deduplication rates for RAM pages and disk sectors respectively for varying workloads and interval lengths. There is no clear correlation between interval length and deduplication rate.

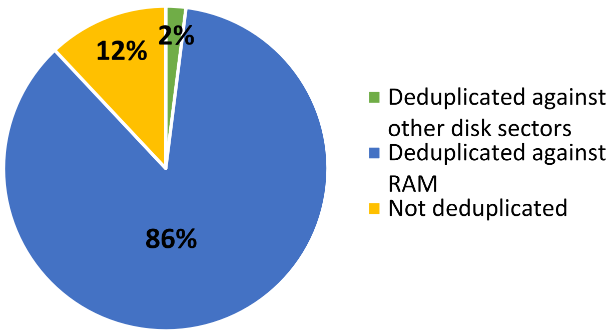

Figure 6.6: Disk sector deduplication breakdown per checkpoint for workload *kernel build* with interval length 2s

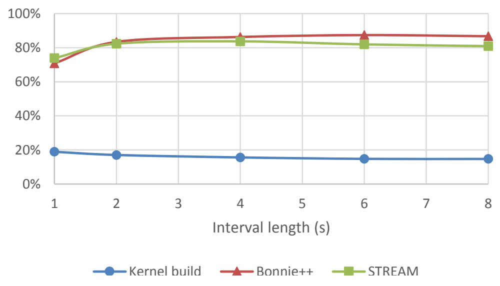

Figure 6.7: Average RAM page deduplication rate for varying workloads and interval lengths

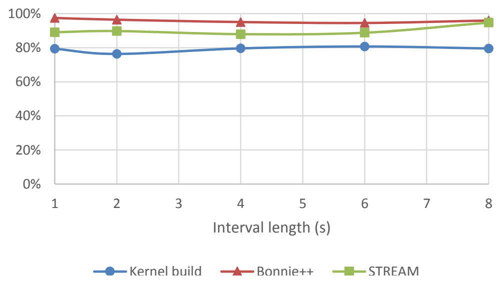

Figure 6.8: Average disk sector deduplication rate for varying workloads and interval lengths

Storage requirements: We recorded the amount of on-disk storage required for each individual checkpoint and all checkpoints in total. This determines the storage requirements for keeping checkpointing data for prolonged periods of time to enable deferred or repeated simulations.

Table 6.5 breaks down the on-disk size of an individual checkpoint's metadata. This consists of the device states as captured by *QEMU*'s snapshot feature, a fixed-size RAM state map (proportional to the amount of RAM in the VM) and a variable-sized disk state map (see Chapter 4.1.2). Figure A.4 plots the on-disk size of individual checkpoints' metadata over time for complete workload runs.

A constant value of

$$\frac{1GiB \text{ RAM size}}{4KiB \text{ page size}} \cdot 16byte \text{ address size} = 4MiB$$

is consumed for the RAM state map. Device states and *QEMU* metadata consume around $80KiB$.

	kernel build	Δ	Bonnie++	Δ	STREAM	Δ
RAM state	4,130 KiB	0	4,130 KiB	0	4,130 KiB	0
Disk state	111 KiB	0.8	1,062 KiB	50, then 0	16 KiB	0
Device state	79 KiB	0	79 KiB	0	79 KiB	0
Sum	4,320 KiB	0.8	5,271 KiB	50, then 0	4,225 KiB	0

Table 6.5: On-disk sizes of specific individual checkpoints captured during different workloads with interval length 2s and average growth rates during complete workload runs

The size of disk state map remains largely constant for the *STREAM* benchmark since it performs no disk writes. The disk state map slowly grows for the *kernel build* workload for each new file created by the build process. The *Bonnie++* benchmark continually creates and deletes files, causing the disk state map size to rise rapidly until a point is reached at which previously dirtied sectors get reallocated by the file system.

A number of peaks occur where the device state size temporarily increases from its usually constant value to $207KiB$. This may be attributed to additional transient state in the virtual disk controllers caused by heavy IO.

Figure 6.9 plots the total on-disk size of all checkpointing data after 10 minutes for varying workloads and interval lengths. Increasing the interval length reduces the total data size with diminishing returns. While individual checkpoints capture more dirty pages and sectors the longer intervals cause more intermediate writes to be "missed", thus reducing the total amount of state recorded.

Figure 6.2 and Figure 6.3 show that *Bonnie++* generates many dirty pages and sectors. However, due to the periodic nature of the benchmark's execution the data is highly repetitive and can therefore be deduplicated very effectively, as seen in Table 6.4. This causes *Bonnie++* to require less total on-disk storage than *kernel build* despite creating more memory and disk traffic.

Downtime: We recorded the downtime caused by each checkpoint. This impacts the speedup achievable by *SimuBoost* and determines whether the workloads remains interactively usable.

Figure 6.10 plots the average VM downtime per checkpoint for varying workloads and interval lengths. Figure A.5 shows the VM downtime over time for complete workload runs. Increasing the interval length also increases the average downtime. This growth correlates with the number of dirty RAM pages accumulated per checkpoint (see Figure 6.4) because the downtime is dominated by iterating over

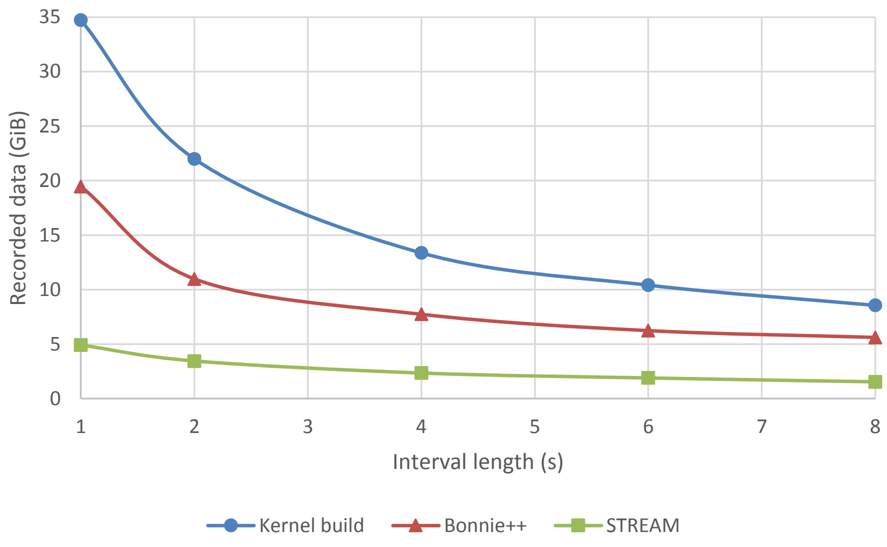

Figure 6.9: Total on-disk size of all checkpointing data after 10 minutes for varying workloads and interval lengths

and copying dirty pages.

The downtime for the workloads *kernel build* averages 44ms for an interval length of 2s, remaining well below our target threshold of 100ms and thus ensuring interactivity. This is a checkpointing slowdown of approx 3%. We gained similar results for the other interval lengths we measured. The *STREAM* benchmark, generating less dirty pages, approximately halves the downtimes compared to *kernel build*.

The *Bonnie++* benchmark oscillates between downtimes similar to *kernel build*'s 44ms and extremely long downtimes reaching up to 5s. This happens when the dirty pages accumulated in a single interval exceed the allocated segment size of 256MiB causing the virutalization to freeze until the pages have been deduplicated and stored (see Chapter 5.2.2).

Figure A.6 compares the VM downtime between immediate and deferred simulation for the *kernel build* workload. The average downtime is 44ms in both cases, indicating that the additional work required to distribute checkpoints to simulation nodes does not impact the performance of the checkpointing process significantly.

We recorded how long the processing running in parallel to the VM's execution took for each checkpoint. This determines whether our implementation is capable of handling the volume of data generated by checkpointing with sufficient speed (see Chapter 5.2.1).

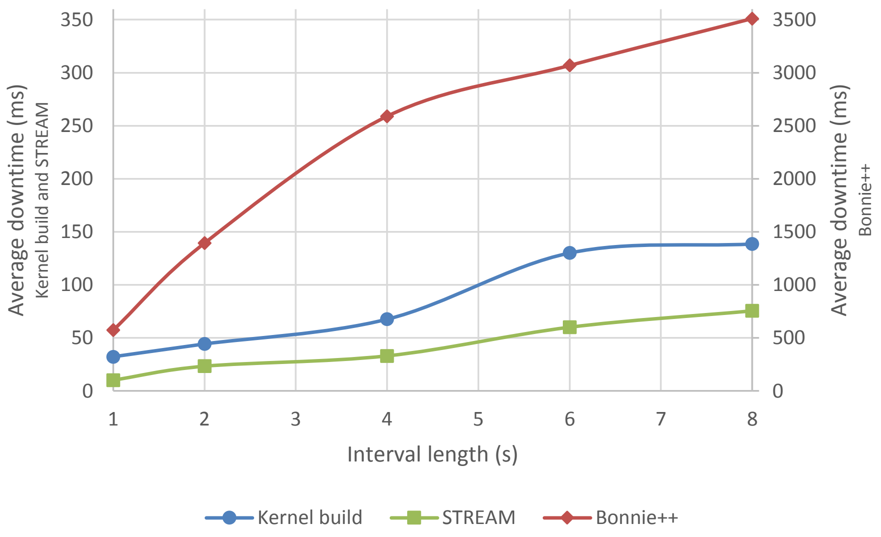

Figure 6.10: Average VM downtime per checkpoint for varying workloads and interval lengths, split scales

Asynchronous processing: Figure 6.11 plots the average asynchronous checkpoint processing time for varying workloads and interval lengths. Increased interval lengths also lead to slightly longer asynchronous processing times. However this growth rate is considerably lower than with VM downtime (see Figure A.5). While the downtime is affected solely by the number of dirty RAM pages and disk sectors that have accumulated during an interval the asynchronous processing can benefit from more deduplication potential. The asynchronous processing design presented in Chapter 4.1.1 can compensate for individual checkpoints taking longer to process than the desired checkpointing interval length by waiting longer to create the next checkpoint. However, the average processing time must remain below the interval length to avoid forming a backlog. In our measurements all average processing times for the different checkpointing interval met this goal.

Figure A.7 breaks down the asynchronous checkpoint processing time for the *kernel build* workload. The process is dominated by hashing the dirty RAM pages and disk sectors for deduplication, which accounts for roughly 70%. 20% of the time is spent on hash table lookups (see Chapter 5.2.3) and writing the deduplicated data to the disk (buffered by the operating system's file system cache). The remaining 10% are writing the state maps (see Chapter 4.1.2) to the disk (also buffered).

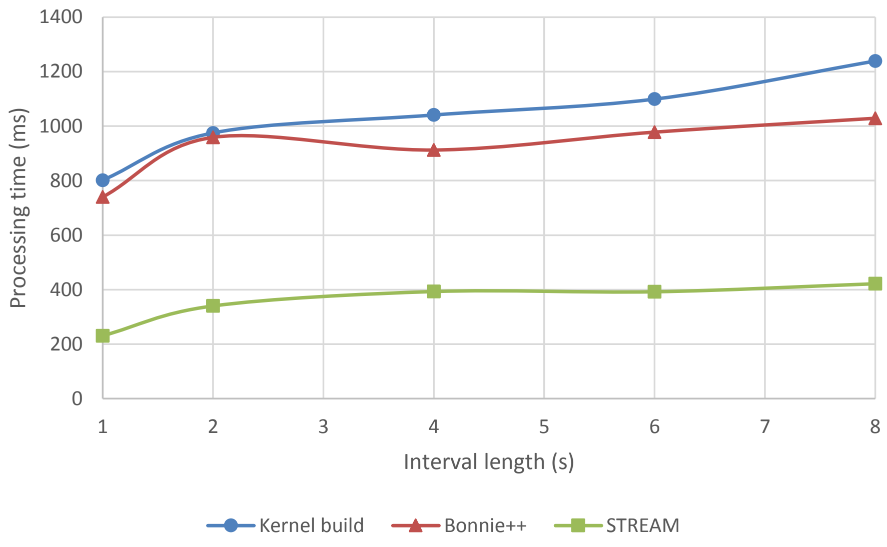

Figure 6.11: Average asynchronous checkpoint processing time for varying workloads and interval lengths

6.3.3 Distribution performance

We recorded the time required for the storage server to load all relevant data and assemble it into a complete VM snapshot. This indicates how the append-only flat file scheme we introduced in Chapter 5.2.4 in order to improve write speed affects read speed.

Figure 6.12 compares the measurements for the two use-cases introduced in Chapter 4.2 using a logarithmic scale:

- For *immediate parallelization* most checkpoints could be loaded within approximately 100ms indicating that these requests required only a moderate amount of disk IO with most data presumably provided from a still hot file system cache. Some checkpoints could be loaded in less that 10ms indicating that they were still available completely in-cache.
- For *deferred parallelization* loading most checkpoints took longer than 100ms (ranging up to 10s in the extreme) indicating that these requests required heavy disk IO. The file system cache started off cold and only slowly became effective as cross-checkpoint deduplication caused data to be read multiple times.

We recorded the time required to transmit the data for entire checkpoints from the server to the nodes. This shows whether the network connection is a bottleneck in our design.

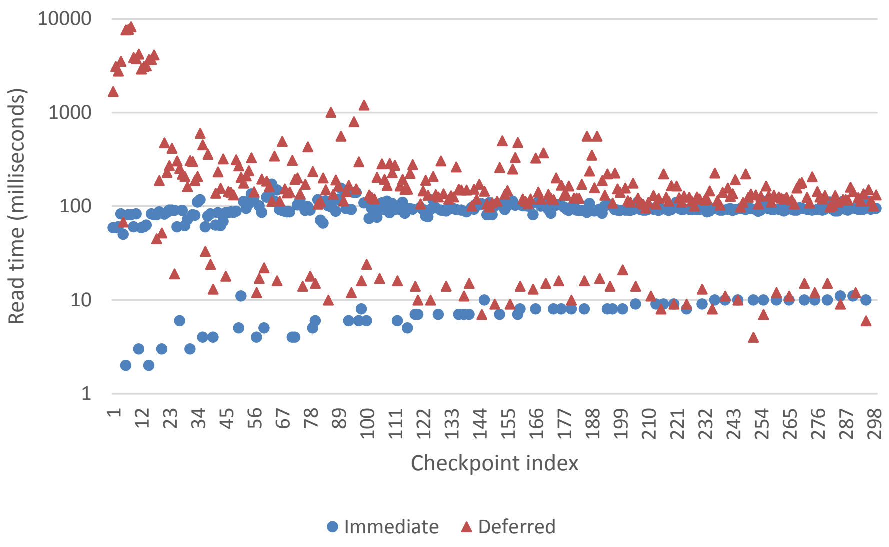

Figure 6.12: Checkpoint data from-disk read time comparison between immediate and deferred simulation for workload *kernel build* with interval length 2s, logarithmic scale

Figure 6.13 plots the network transfer time for the *kernel build* workload. The minimum transfer time of 2.5s is achieved when there is no network contention, i.e., only a single checkpoint is being transferred over the 10GBit link. Overlapping distribution of checkpoints to different nodes over the same network interface cause the slightly longer average transfer time of 3.5s. The maximum transfer time of 7s is reached early in the simulation process when multiple checkpoints are scheduled nearly simultaneously.

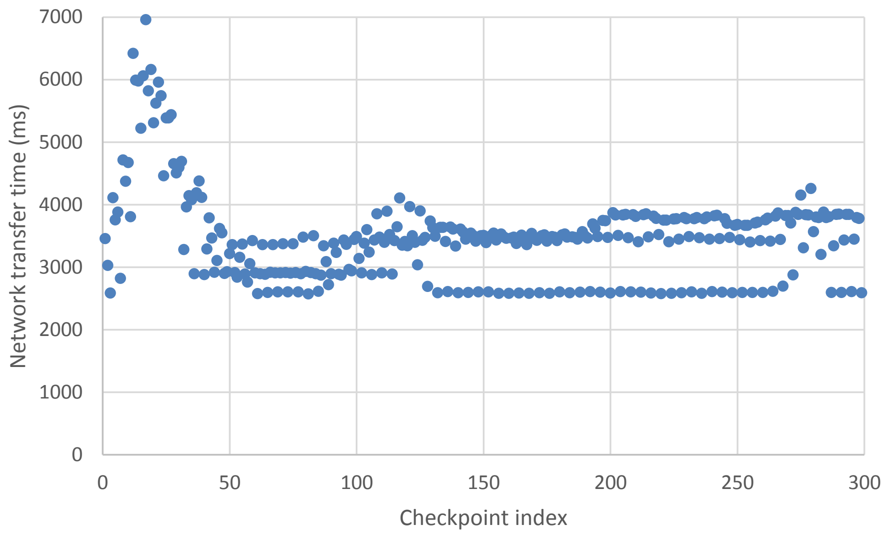

Figure 6.13: Checkpoint data network transfer time for workload *kernel build* with interval length 2s

6.3.4 Total runtime

As the final step in evaluating our implementation we looked at the total runtimes of workloads including the virtualization and the parallelized simulation phases. By comparing these results with the expected runtime of a conventional, non-parallel simulation we were able to determine the achievable speedup, the primary indicator of *SimuBoost*'s viability.

We also compared our measurements with the predictions made by the formal model presented in Chapter 3.2 using Equation 3.5. We chose $N = 4$ to match our evaluation setup of 4 simulation nodes (see Chapter 6.2.1), $T_{vm} = 600$ for the 10 minute runtime of our workloads (see Chapter 6.2.2), $s_{sim} = 31$ to match our fixed simulation slowdown assumption (see Chapter 6.1) and $s_{log} = 1$ since our implementation does not yet perform any non-deterministic event logging. We based the simulation initialization time $t_i = 3.7$ and the checkpointing slowdown factor $s_{cp} = 1.024$ on our existing measurements.

We measured the total runtime starting with the virtualization and ending with the last running simulation node. Figure 6.14 plots these results against the predictions made by the formal model. Figure 6.15 transforms these results to the achieved speedup using Equation 3.6. The choice of workload has no noticeable impact on the speedup. Increasing the interval length increases the speedup with diminishing returns. Our implementation's performance tracks the predictions made by the formal model very closely.

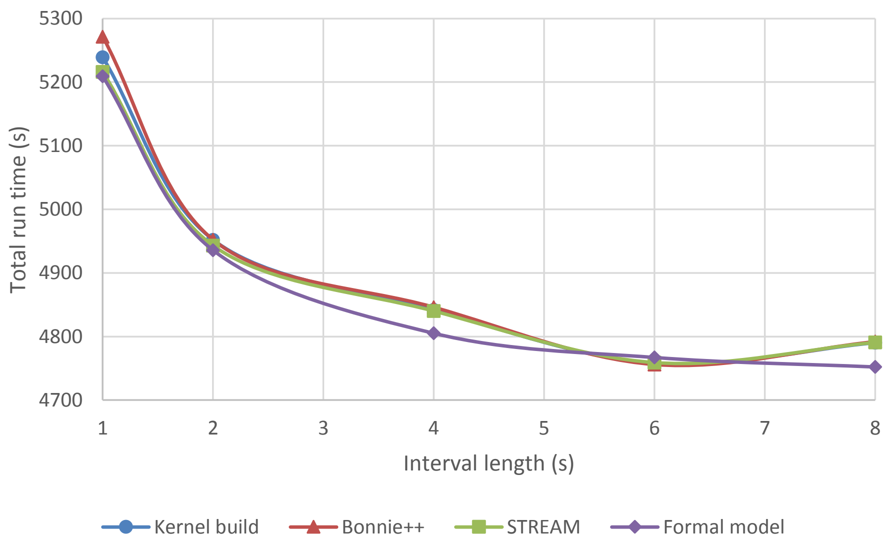

Figure 6.14: Total parallelized simulation runtime for varying workloads and interval lengths, compared with formal model predictions for $N = 4; s_{sim} = 31; s_{log} = 1; s_{cp} = 1.024; T_{vm} = 600; t_i = 3.7$

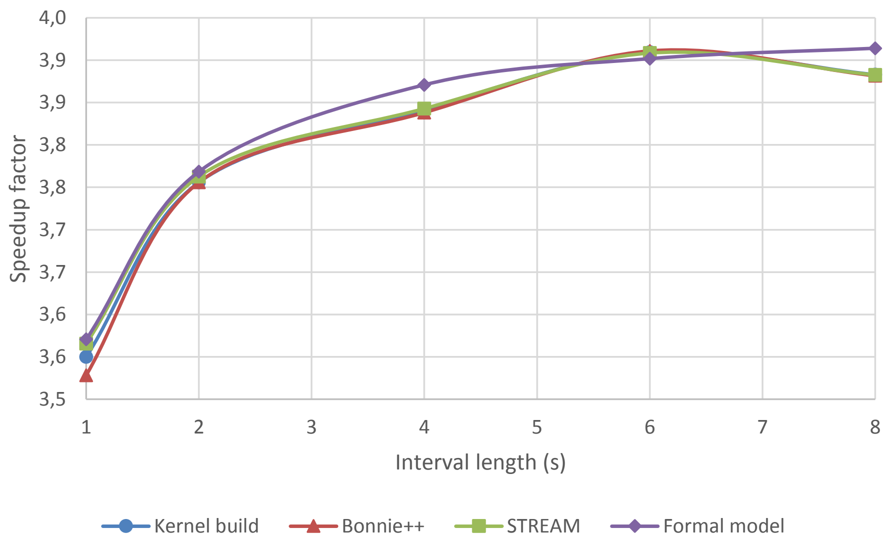

Figure 6.15: Speedup achieved for varying workloads and interval lengths, compared with formal model predictions for $N = 4; s_{sim} = 31; s_{log} = 1; s_{cp} = 1.024; T_{vm} = 600; t_i = 3.7$

6.4 Discussion

The results presented in Chapter 6.3 show that the choice of workload has a significant impact on the number of dirty RAM pages and disk sectors that accumulate per checkpoint. This directly affects the length of the VM downtime during checkpointing. Workloads that accumulate more data than provisioned for lead to significantly longer multi-second downtimes (see Chapter 5.2.2).

Our measurements using a constrained number of available simulation nodes show that the downtime has a negligible impact on the total speedup achieved, even for extreme cases with multi-second downtimes. This stands in contrast to the predictions made by the *SimuBoost* paper [5] for an unlimited number of available simulation nodes. In this scenario the downtime is one of the main factors determining the speedup.

Achieving a low downtime remains critical since it enables workloads to stay interactive and avoids network connections being dropped. Our checkpointing implementation is able to satisfy the requirement of downtimes lower that 100ms layed out in Chapter 3.1.

The asynchronous processing of checkpoints as described in Chapter 4.1.1 always finished before the next interval started in our test cases. This satisfies our secondary goal of ensuring uniform interval lengths. Our measurements show that a significant amount of time is spent on calculating hashes of RAM pages for deduplication. This indicates that additional parallelization of the hash calculations could help further increase the margin between asynchronous processing time and interval length.

Increasing the interval length usually increases the number of dirty RAM pages that accumulate per checkpoint. This leads to longer downtimes since more data needs to be hashed for deduplication and potentially stored. On the other hand longer intervals lower the total number of checkpoints recorded and thereby reduce the amount of on-disk storage required. For setups with a limited number of simulation nodes longer intervals also improve the total speedup, as predicted by the formal model introduced in Chapter 3.2.

We measured almost identical speedups for the different workloads. The measurements closely tracked the predictions made by the formal model. This indicates that the formal model correctly captures the performance characteristics of the components of the *SimuBoost* process implemented in our prototype.

The average RAM deduplication rate of 15% for the *kernel build* workload indicates reasonable effectiveness for real-world workloads. The 85% deduplication rate for the benchmarks indicates very high effectiveness for synthetic workloads. The very high rate of deduplication of disk sectors against RAM pages effectively reduces the checkpoint storage problem to capturing dirty RAM pages.

The number of CPU instructions executed per interval vary considerably for some workloads. This means our assumption of a constant simulation slowdown made in Chapter 6.1 may lead to predictions that diverge to a certain degree from later measurements made with a yet unimplemented complete *SimuBoost* solution.

Since checkpointing performance is the same for immediate and deferred parallelization we can infer that distributing checkpoints while they are being created does not harm the write performance. However, loading the checkpoint data later without the benefit of still being cached slows down the read performance noticeably.

The network transfer times for checkpoints are distributed across a wide range rather than all checkpoints being transferred in roughly the same (optimal) time. This indicates that the available network bandwidth is the limiting factor for our current design.

Chapter 7

Conclusion

The main objective of this work was to evaluate the viability of distributed system simulation using the *SimuBoost* concept.

We improved an existing checkpointing and deduplication solution, reducing the VM downtime in order to meet *SimuBoost*'s requirements. We also implemented a system for distributing simulations in a cluster. The resulting prototype enabled us to empirically measure *SimuBoost*'s speedup potential.

We evaluated our prototype by executing a number of workloads while recording and distributing checkpoints. The evaluation showed that our improvements greatly reduced the downtime caused by checkpointing. We were able to demonstrate that a *SimuBoost* implementation can achieve significant speedups over conventional full-system simulation.

7.1 Future work

Checkpoints created by *QEMU* in *KVM* mode cannot be loaded in *QEMU* in *TCG* mode. *V2E* [54] presents an implementation capable of loading checkpoints created by a virtualization system in a simulation system. Adding equivalent functionality to our prototype would make it possible to execute parallel simulations and record traces, albeit with reduced accuracy caused by simulations still diverging from the virtualization.

Additionally implementing the recording and deterministic replay of events proposed by *SimuBoost* (see Chapter 2.4.2) would turn the prototype into a fully functional *SimuBoost* implementation.

In Chapters 5.2.2 and 5.2.3 we describe tuning parameters exposed by the implementation that require a certain level of a priori knowledge for optimal performance. Future improvements of the implementation could aim to auto-tune these parameters.

In Chapter 4.1.1 we decided to avoid queuing of checkpoint processing to prevent backlogs from growing uncontrollably. An auto-tuning implementation might be able to effectively counter this effect and could thus benefit from queuing to ensure uniform interval lengths.

In Chapter 6.2 we describe our evaluation setup consisting of a limited number of simulation nodes. Additional insight could be gained by performing similar measurements using a set of nodes large enough to satisfy the "no queuing" assumption made in the *SimuBoost* paper [5].

Chapter 6.3 showed that the performance of our design is limited by the cluster network bandwidth. In Chapter 4.2.1 we decided to transmit complete snapshots of the VM from the server to the nodes to minimize the amount of state kept on the nodes. Future work could implement and evaluate an approach were the nodes use local deduplication caches to reduce the amount of network traffic.

Appendix A

Additional graphs

The following graphs provide additional data from the measurements discussed in Chapter 6.3.

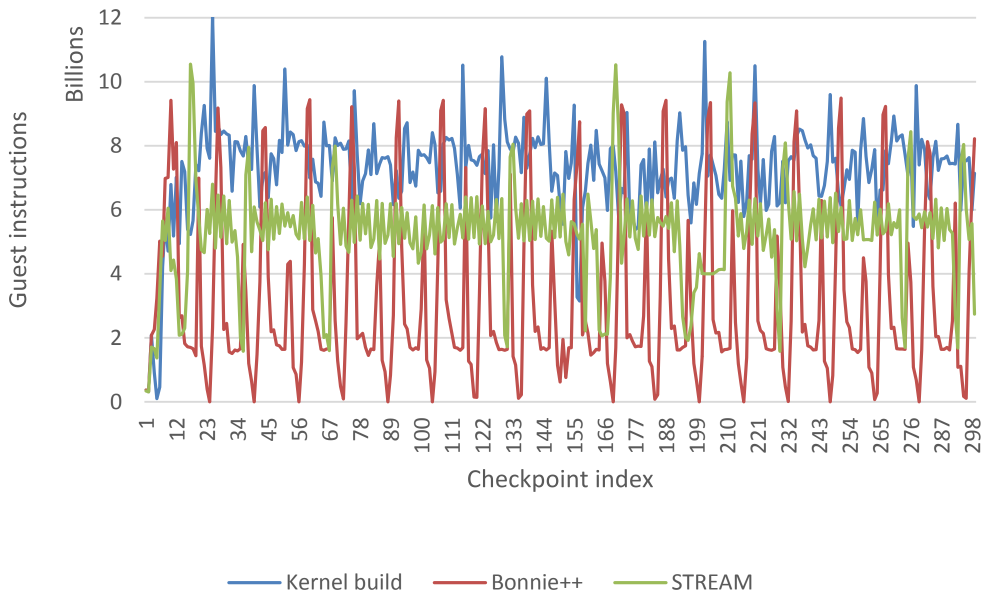

Figure A.1: Number of CPU instructions executed within the VM during a single interval for varying workloads with interval length 2s

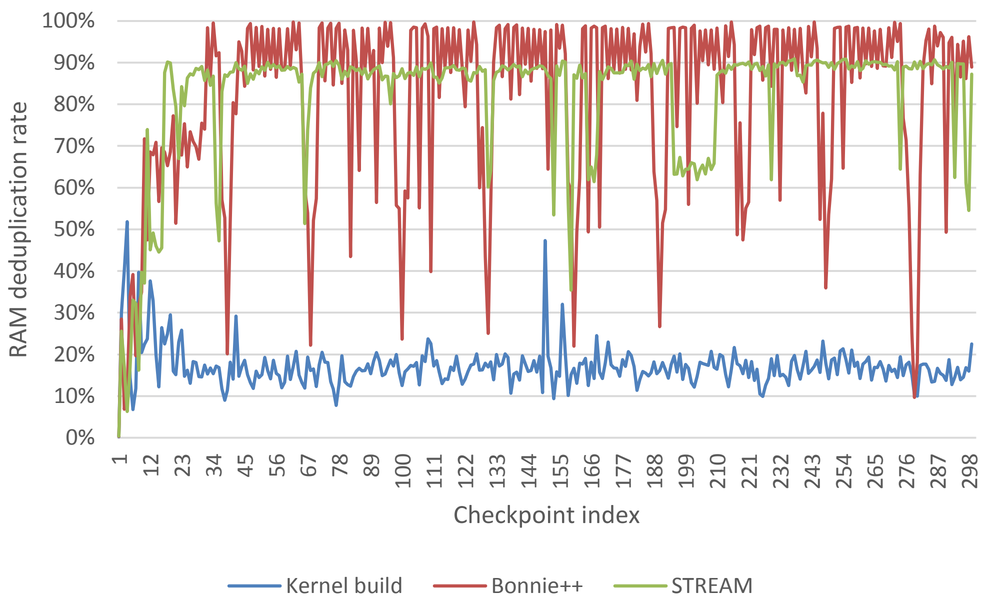

Figure A.2: RAM page deduplication rate per checkpoint for varying workloads with interval length 2s

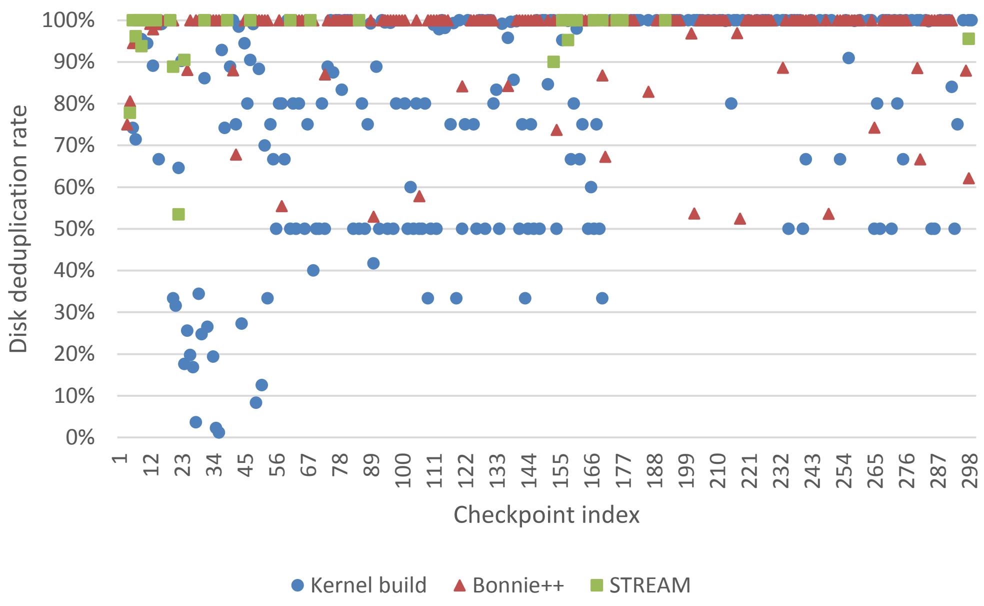

Figure A.3: Disk sector deduplication rate per checkpoint for varying workloads with interval length 2s

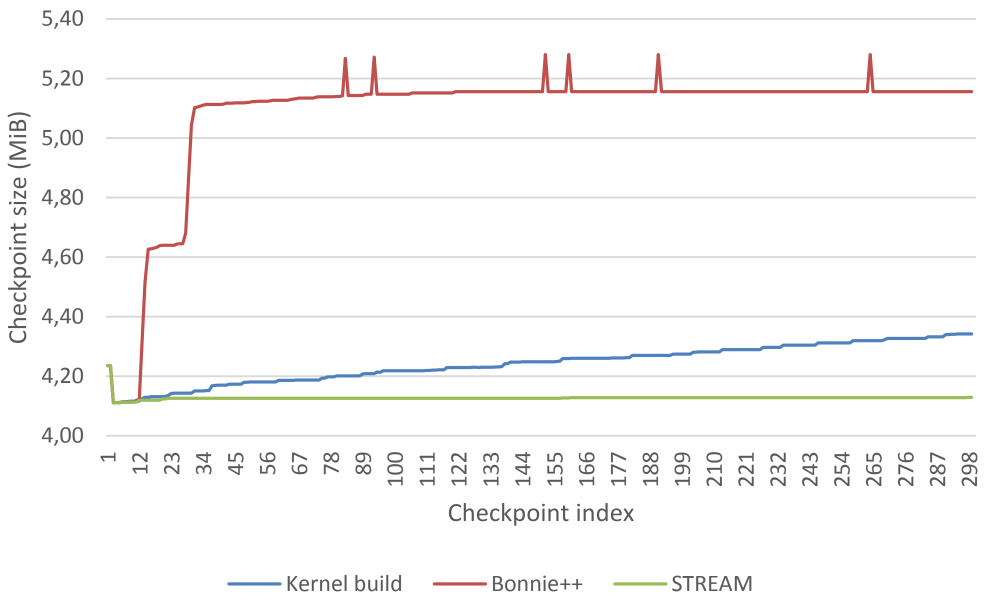

Figure A.4: On-disk size of individual checkpoints' metadata for varying workloads with interval length 2s

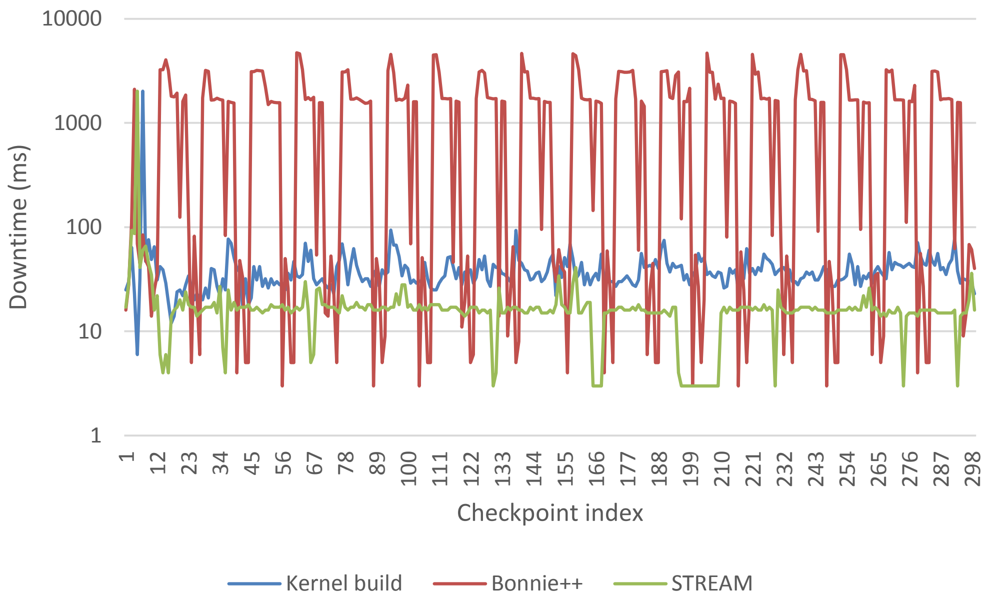

Figure A.5: VM downtime per checkpoint for varying workloads with interval length 2s, logarithmic scale

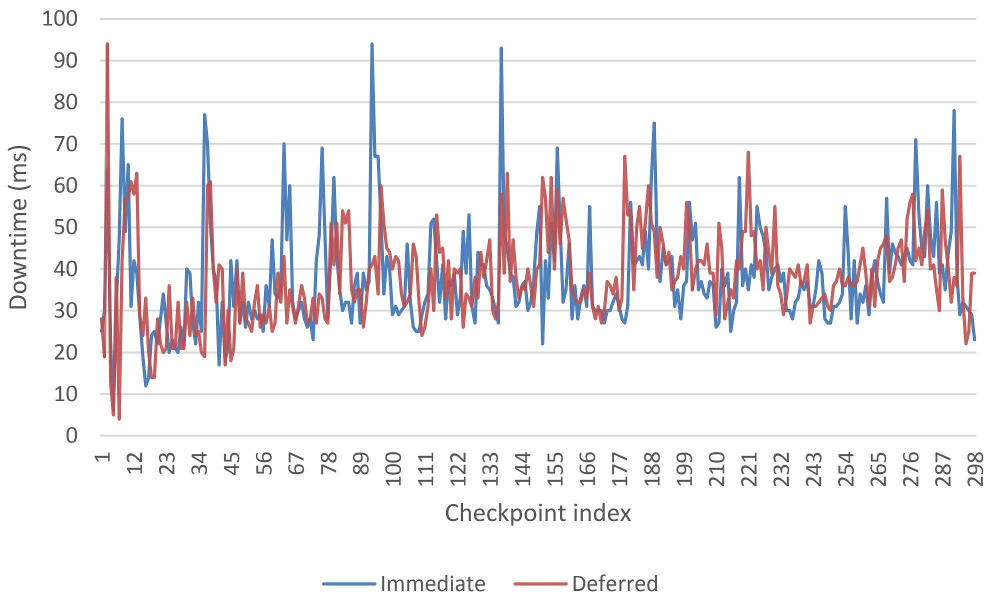

Figure A.6: VM downtime comparison between immediate and deferred simulation for workload *kernel build* with interval length 2s

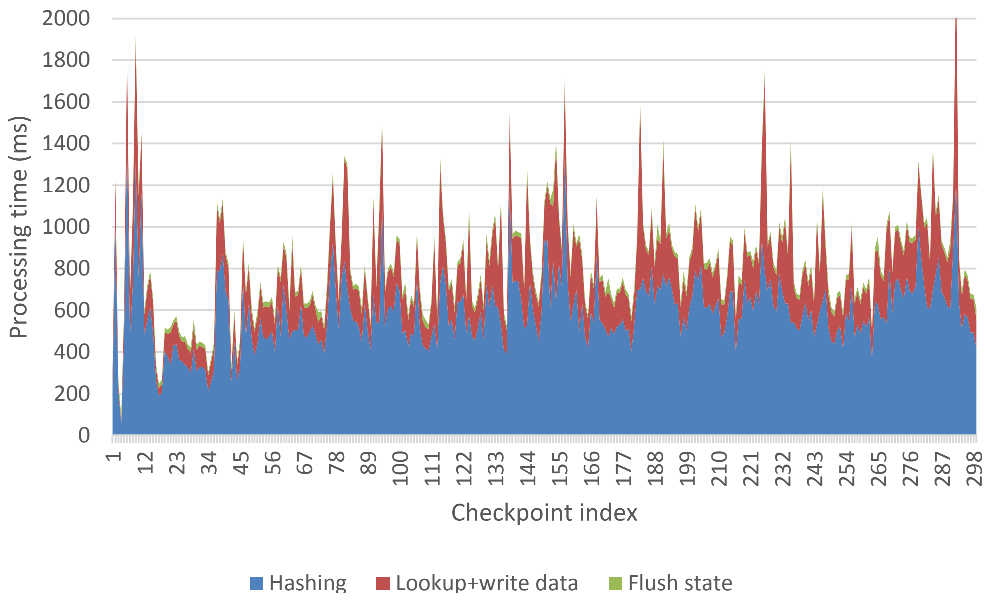

Figure A.7: Breakdown of asynchronous checkpoint processing time for workload *kernel build* with interval length 2s

Bibliography

[1] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 133–143, New York, NY, USA, 2012. ACM.

[2] *QEMU website*. `http://www.qemu.org/`.

[3] Mateusz Jurczyk, Gynvael Coldwind, et al. Identifying and exploiting windows kernel race conditions via memory access patterns. 2013.

[4] *Bochs website*. `http://bochs.sourceforge.net/`.

[5] 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.

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

[7] *MongoDB website*. `http://www.mongodb.org/`.

[8] 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. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, December 2002.

[9] Mark Probst. Dynamic binary translation. In *UKUUG Linux Developer's Conference*, volume 2002. sn, 2002.

[10] *VirtualBox Manual: Chapter 10. Technical background*. `https://www.virtualbox.org/manual/ch10.html`.

[11] *VirtualBox website*. http://www.virtualbox.org/.

[12] *VMware website*. http://www.vmware.com/.

[13] Jim Smith and Ravi Nair. *Virtual machines: versatile platforms for systems and processes*. Elsevier, 2005.

[14] Intel. Intel vanderpool technology for ia-32 processors (vt-x) preliminary specification. 2005.

[15] AMD. *AMD64 Virtualization Codenamed "Pacifica" Technology: Secure Virtual Machine Architecture Reference Manual*, 2005.

[16] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974.

[17] *Hyper-V website*. http://technet.microsoft.com/en-us/windowsserver/dd448604.aspx.

[18] Avadh Patel, Furat Afram, and Kanad Ghose. Marss-x86: A qemu-based micro-architectural and systems simulator for x86 multicore processors. In *1st International Qemu Users' Forum*, pages 29–30, 2011.

[19] Peter S Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[20] Stuart Hacking and Benoît Hudzia. Improving the live migration process of large enterprise applications. In *Proceedings of the 3rd International Workshop on Virtualization Technologies in Distributed Computing*, VTDC '09, pages 51–58, New York, NY, USA, 2009. ACM.

[21] Michael R. Hines, Umesh Deshpande, and Kartik Gopalan. Post-copy live migration of virtual machines. *SIGOPS Oper. Syst. Rev.*, 43(3):14–26, July 2009.

[22] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 169–179, New York, NY, USA, 2007. ACM.

[23] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2Nd Conference on Symposium on*

Networked Systems Design & Implementation - Volume 2, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[24] 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, pages 101–110, New York, NY, USA, 2009. ACM.

[25] Ajay Surie, H. Andrés Lagar-Cavilla, Eyal de Lara, and M. Satyanarayanan. Low-bandwidth vm migration via opportunistic replay. In *Proceedings of the 9th Workshop on Mobile Computing Systems and Applications*, HotMobile '08, pages 74–79, New York, NY, USA, 2008. ACM.

[26] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 161–174, Berkeley, CA, USA, 2008. USENIX Association.

[27] Paula Ta-Shma, Guy Laden, Muli Ben-Yehuda, and Michael Factor. Virtual machine time travel using continuous data protection and checkpointing. *ACM SIGOPS Operating Systems Review*, 42(1):127–134, 2008.

[28] Kurt B Ferreira, Rolf Riesen, Ron Brighwell, Patrick Bridges, and Dorian Arnold. libhashckpt: hash-based incremental checkpointing using gpu's. In *Recent Advances in the Message Passing Interface*, pages 272–281. Springer, 2011.

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

[30] Pierre Riteau, Christine Morin, and Thierry Priol. Shrinker: Improving live migration of virtual clusters over wans with distributed data deduplication and content-based addressing. In *Euro-Par 2011 Parallel Processing*, pages 431–442. Springer, 2011.

[31] Jui-Hao Chiang, Han-Lin Li, and Tzi-cker Chiueh. Introspection-based memory de-duplication and migration. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, pages 51–62, New York, NY, USA, 2013. ACM.

[32] Eunbyung Park, Bernhard Egger, and Jaejin Lee. Fast and space-efficient virtual machine checkpointing. In *Proceedings of the 7th ACM SIG-*

PLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '11, pages 75–86, New York, NY, USA, 2011. ACM.

[33] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[34] *QEMU Wiki: KVM*. `http://wiki.qemu.org/KVM`, retrieved March 9th 2015.

[35] *Wikibooks: QEMU: Images*. `http://en.wikibooks.org/wiki/QEMU/Images`, retrieved March 29th 2015.

[36] *Wikibooks: QEMU: Monitor*. `http://en.wikibooks.org/wiki/QEMU/Monitor`, retrieved March 29th 2015.

[37] 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.

[38] Robert B. Miller. Response time in man-computer conversational transactions. 1968.

[39] *MongoDB website: Document Databases*. `http://www.mongodb.com/document-databases`.

[40] *BSON specification*. `http://bsonspec.org/`.

[41] *Redis website*. `http://redis.io/`.

[42] *Redis documentation: Persistence*. `http://redis.io/topics/persistence`, retrieved March 26th 2015.

[43] *Memcached website*. `http://memcached.org/`.

[44] *LevelDB website*. `https://github.com/google/leveldb`.

[45] *Kyoto Cabinet website*. `http://fallabs.com/kyotocabinet/`.

[46] *Kyoto Tycoon website*. `http://fallabs.com/kyototycoon/`.

[47] *LevelDB documentation: Implementation details*. `http://htmlpreview.github.io/?https://raw.githubusercontent.com/google/leveldb/master/doc/impl.html`, retrieved March 9th 2015.

[48] *Slurm Workload Manager documentation*. `http://www.schedmd.com/slurmdocs/slurm.html`.

[49] *KVM website*. `http://www.linux-kvm.org/`.

[50] *Intel 64 and IA-32 Architectures Software Developer's Manual*.

[51] *CityHash website*. `https://code.google.com/p/cityhash/`.

[52] *FarmHash website*. `https://code.google.com/p/farmhash/`.

[53] *inoticoming Manpage*. `http://manpages.ubuntu.com/manpages/hardy/man1/inoticoming.1.html`.

[54] Lok-Kwong Yan, Manjukumar Jayachandra, Mu Zhang, and Heng Yin. V2e: Combining hardware virtualization and softwareemulation for transparent and extensible malware analysis. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pages 227–238, New York, NY, USA, 2012. ACM.