Karlsruher Institut für Technologie

Deduplicating Virtual Machine Checkpoints for Distributed System Simulation

Bachelorarbeit
von

cand. inform. Nikolai Baudis

an der Fakultät für Informatik

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

Bearbeitungszeit: 02. Juli 2013 – 02. November 2013

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

Karlsruhe, den 2. November 2013

Deutsche Zusammenfassung

Simulationen ganzer Systeme (*Full System Simulation*) haben aufgrund der erhöhten Komplexität bei der Simulation eine viel geringere Ausführungsgeschwindigkeit als hardwarebeschleunigte Virtualisierungen. Dies führt zu einer über tausendfachen Verlangsamung. *SimuBoost* zielt darauf ab, diese Lücke in der Leistung zu schließen, indem mithilfe von parallelisierten und verteilten Simulationen die Simulationsgeschwindigkeit erhöht wird. Gestartet werden diese durch *Checkpoints* von virtualisierten Systemen. Um einen optimalen Grad an Parallelisierung zu erreichen, werden diese Checkpoints in niedrigen Intervallen von 1–2 s gespeichert. Dadurch gelingt es, den durch das Speichern der Checkpoints entstehenden *Overhead* zu minimieren.

In dieser Arbeit wird ein Checkpoint-Mechanismus vorgestellt, der die Anforderungen für SimuBoost – eine minimierte Datenmenge und *Downtime* – erfüllt. Der Checkpoint-Mechanismus speichert inkrementelle Checkpoints des in *QEMU* virtualisierten Systems und kann dadurch diese Anforderungen erfüllen. Zudem dedupliziert er sowohl Daten des Hauptspeichers als auch der Festplatten mithilfe eines hashbasierten Caches. Die Checkpoints werden in einer NoSQL-Datenbank gespeichert, damit sie auf einfache Weise an die Simulationen verteilt werden können.

Die Evaluation zeigt, dass der hashbasierte Cache die Datenmenge erheblich reduziert, vor allem durch Benutzung von in der Vergangenheit schon in Checkpoints gespeicherten Daten: Bis zu 42 % der Speicherseiten können durch die Deduplizierung eingespart werden. Die Evaluation zeigt aber auch, dass die durch Datenbankanfragen entstehenden Kosten für 40–60 % der Gesamtlaufzeit, die für Checkpoints benötigt wird, verantwortlich sind. Daher muss dieses Verfahren noch mit CoW kombiniert werden, um eine bessere Laufzeit für die Checkpoints zu erreichen.

Abstract

Full system simulation suffers – due to its increased complexity – from a much lower execution speed than hardware-accelerated virtualization, resulting in a slowdown of over 1000. To close this performance gap, *SimuBoost* aims at increasing full system simulation's execution speed by performing parallelized distributed simulations, which are started using checkpoints from a hardware-accelerated virtualization. To reach an optimal degree of parallelization, checkpoints have to be taken at an interval of 1–2 s, which minimzes the overhead of the checkpointing.

In this thesis, a checkpointing mechanism that meets the requirements for SimuBoost – minimized data amount and downtime – is proposed. To meet these requirements, the checkpointing mechanism performs incremental checkpointing of virtualized systems in QEMU and deduplicates both memory and block device data via a hash-based cache. The checkpoints are saved in a NoSQL database for a simplified distribution.

The evaluation shows that the hash-based cache greatly decreases the data amount, especially by reusing data that was already checkpointed in the past: Up to 42 % of memory pages can be saved through deduplication. However, the evaluation also shows that the overhead caused by database queries is too high and that it is responsible for 40–60 % of the total downtime. Therefore, this approach needs to be optimized by combining it with CoW to reach a suitable downtime.

Contents

1 Introduction

Full system simulation is a useful tool in operating systems development. By modeling the hardware of a full system, simulators can facilitate hardware-independent development and provide a system that is inspectable and modifiable. For example, breakpoints can be used to suspend the system on certain conditions and the system state can be modified. However, the execution speed of full system simulators is much lower than that of hardware-accelerated, virtualized systems. Therefore, only short workloads and simplified hardware models can be simulated in a reasonable execution time. *SimuBoost* aims at increasing the simulation speed through virtualization and parallel simulation. The workload is executed on a virtualized system, while taking checkpoints in frequent intervals, which contain the full system state. Based on these checkpoints, the workload is simulated on another system for one checkpointing interval. By using multiple simulation nodes, the simulation can be parallelized.

The objective of this thesis is to develop a checkpointing mechanism that is suitable for the requirements of SimuBoost: To achieve a high parallelization speedup, the checkpointing interval for SimuBoost is within a few seconds. To keep the downtime as low as possible, incremental and deduplicated checkpointing is required. In addition to that, the checkpoints need to be suitable for distribution to the simulation nodes, with distribution targets that are unknown in advance. This effect is due to fluctuating simulation speed of single workloads, which results in an unpredictable order of distribution to the simulation nodes.

The checkpointing mechanism developed in this thesis uses incremental checkpointing and hash-based deduplication and stores checkpoints in a database for distribution. The implementation takes checkpoints of virtual machines running in QEMU and KVM and stores them in MongoDB. To obtain all system states and to make use of internal functions, e.g., for memory management, the checkpointing mechanism is integrated into QEMU. Using these internal functions, incremental checkpoints can be saved and loaded. To deduplicate the data during the checkpointing, a hash-based approach is used together with a cache to identify recurring data that was already checkpointed. The checkpoints are stored in a NoSQL database, so

that checkpoints can be loaded even if the order of distribution targets is not known in advance.

As the evaluation shows, this checkpointing mechanism deduplicates up to 42 % of memory and 80 % of block devices, depending on the workload. Also, the effectiveness of the hash-based deduplication cache is shown, which causes a high amount of previously checkpointed data to get deduplicated with the hashing hardly increasing the downtime of the virtual machine, compared to the high costs of database queries that was saved. However, the downtime has to be optimized by combining the presented approach with CoW to reduce the sudden database load during the checkpointing, which is accountable for the majority of the downtime.

In Chapter 2, the fundamentals of full system simulation, checkpointing, and the concepts used in the checkpointing approach developed in this thesis are explained. Chapter 3 further analyzes the use case and requirements for the checkpointing mechanism and also the amount of modified data in a checkpointing interval as well as its deduplication potential. In Chapter 4, the general function of the checkpointing mechanism is described. Also, design decisions for the checkpointing concepts are elaborated. Chapter 5 describes the integration into the virtual machine monitor and how data is collected, deduplicated, and sent to the database. An Evaluation of the checkpointing mechanism is performed in Chapter 6, showing the efficiency of the deduplication approach and the performance regarding deduplication, data amount, and downtime. Chapter 7 concludes the thesis and gives an outlook of future work.

2 Background

This chapter explains the fundamentals of full system simulation and checkpointing and connections between concepts that are used in this thesis. Also, already existing checkpointing mechanisms will be explained and classified.

2.1 Full System Simulation

In operating systems (OS) development, tasks like debugging or memory investigation can be difficult on real hardware or even impossible, if CPU or device development is not yet fully completed [20]. Full system simulators offer the advantage of development that is independent to physical hardware. Another point is the increased inspectability, i.e., the complete state of the system can be investigated [10], since full system simulation includes the whole system. Therefore, the CPU, memory, and devices are modeled. One type of such simulation, functional simulation, focuses on functional correctness and emulates the exact behavior of the target system [2].

Simics is such a full system simulator that runs an unmodified OS and emulates different devices and CPU architectures [20]. Several models exist to simulate different processors like x86, MIPS, ARM, etc., and to simulate devices. Additionally, networks of simulated systems can be run and analyzed with Simics.

The *MARSSx86* simulator is based on QEMU and runs an unmodified OS as well [24]. It was designed – as the name would suggest – for x86-based architectures and is able to perform cycle-accurate simulations of both single-core and multi-core configurations.

Full system simulators can run a virtual machine (VM) by emulating a full system and using an unmodified OS, similar to virtual machine monitors (VMMs). The main difference to VMMs is in the advanced analyzation tools full system simulators provide: Full system simulators can use breakpoints and hooks, single step through the execution, record user input, modify the systems, etc. [20]. However, this increased feature set comes with additional complexity, resulting in a more resource-intensive and time-consuming execution. For example, the slowdown of a Linux

kernel build increases by a factor of over 1000 [26] when using Simics compared to a hardware-accelerated virtualization, which uses processor extensions like Intel VT-X or AMD-V to increase virtualization performance.

2.1.1 SimuBoost

To speed up functional full system simulation, *SimuBoost* [26] uses hardware accelerated virtualization. While the virtualization node executes the workload, it takes checkpoints in fixed intervals. These checkpoints are later distributed to simulation nodes, each of which simulates only one checkpointing interval, i.e., the workload that accumulates between two consecutive checkpoints. By using checkpoints as a starting point, the simulation can be parallelized and therefore accelerated. Figure 2.1 shows the virtualization node and three simulation nodes. The workloads *i[1]*, …, *i[k]*, …, *i[n]* are executed using hardware virtualization at first. Then, using checkpoints to distribute the system state, the workloads are simulated as soon as a checkpoint was taken on the virtualization node. Although workloads are based on previous system states, this approach enables parallel simulation.

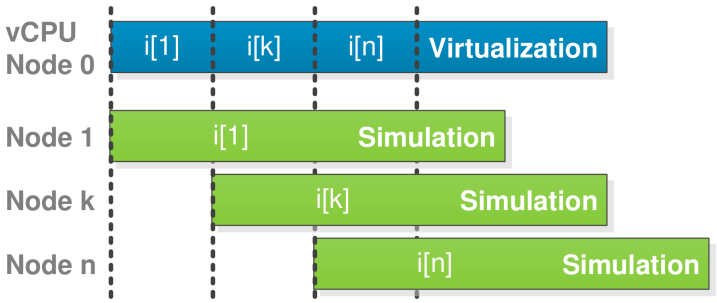


Figure 2.1: Virtualization and simulation of a workload using SimuBoost [26]. The system state needed for the parallel simulations is aquired using checkpoints of the virtualized system.

Because of the large difference in execution speed between hardware-accelerated virtualization and full system simulation, a large speedup can be achieved through parallelization. At the same time, SimuBoost needs to frequently create checkpoints in order to generate independent intervals for simulation. Taking these checkpoints results in an overhead for the virtualization node, which reduces the overall reachable speedup by parallelization, making it a crucial factor for performance. Another important aspect is the interval length, which affects the temporal relationship between the checkpointing overhead and an actual simulated interval, i.e., effectively the speedup. This speedup decreases if the checkpointing interval is too low or too high; an optimal speedup was found using an interval in the range of 1–2 seconds.

Additionally, taking checkpoints at a high frequency results in a high data amount, which has to be distributed to the simulation nodes. Thus, SimuBoost depends on both efficient hardware-accelerated virtualization and and an efficient checkpointing mechanism.

2.2 QEMU

QEMU is a generic and open source VMM that can run a guest operating system in a VM by emulating different CPU architectures [4]. It supports full system emulation, therefore no changes to the guest OS are required. Devices such as block devices, displays, or network devices can be emulated to provide the guest OS with an entire virtual system.

QEMU is often used together with Kernel-Based Virtual Machine (KVM), a virtualization solution as a Linux kernel module [17]. KVM can substantially improve virtualization performance by using the hardware extensions present in modern AMD and Intel CPUs for processor state virtualization [13], if the same architecture is used for the guest and host.

For memory management, KVM allocates the guest's memory and uses shadow page tables [17], which are hidden from the guest and maintained by the host. It translates guest virtual addresses to the corresponding host physical addresses. Additionally, Intel's *Nested Page Table* (NPT) and AMD's *Extended Page Table* (EPT) structures are used to quickly translate guest physical addresses to host physical addresses without lookups to the host page tables [18].

Block devices such as hard disk drives and CD-ROM drives are emulated by QEMU. They are typically managed as a single file inside the host systems file system. While multiple block device formats are supported, the *qcow2* format [22] offers the most interesting features such as snapshot support and copy-on-write (CoW) overlay images, which represent changes to the base image.

One advantage of virtualization is an encapsulated system that is easily accessible through the VMM, e.g., to save the system state. This can be achieved with a checkpointing mechanism.

2.3 Virtual Machine Checkpoints

The terms snapshot and checkpoint are often used interchangeably in the context of virtual machines. In some cases – like the just mentioned qcow2 format – a snapshot, however, refers to saving only the state and data of block devices, whereas a checkpoint means that the entire system state is being saved. In this thesis, a VM checkpoint is considered as containing a *complete system state*, which can later be

used to restore a virtual machine to the exact state in which the checkpoint was saved. This includes CPU and device states, memory, and block devices.

In practice, checkpointing mechanisms are not only employed to restore a system to a previous state, but also as key technology in VM replication and migration. Over the years, previous work has proposed a variety of checkpointing mechanisms for different applications. The *Remus* VM replication mechanism [7] targets high availability via replication. This is achieved by frequent VM checkpoints that are transferred to a backup host. The mechanism is optimized for high-frequency checkpointing using a live migration approach, i.e., disk and memory changes are stored in a buffer while the VM is running until it finally gets suspended to complete the checkpoint. It is then sent directly to the backup host, where it is stored in memory.

When taking a checkpoint, a consistent state needs to be saved, i.e., all data in a checkpoint needs to be saved at the exact same point in time. Therefore, the VM has to be suspended for a moment and started again later, which causes a *downtime*. Sun and Blough describe a more general approach to a checkpointing mechanism based on copy-on-write (CoW) [28], which writes a full checkpoint to a file in every checkpoint interval. By using CoW-buffers to copy memory while the VM is running, the downtime is minimized. However, they found that duplicate data constitutes the majority of the checkpointing costs, since the checkpoints are not deduplicated.

Another general approach, *libhashcpkt* [11], performs incremental checkpointing, i.e., it only considers data that has actually changed for a checkpoint. Looking at memory, this is called a page-based incremental approach. Additionally, the data is hashed and compared to previously checkpointed hashes, making it hash-based instead of page-based. To speed up checkpointing, the hashing is outsourced to the GPU.

To conclude the overview of checkpointing mechanisms, a short classification of important applications and concepts follows. VMMs usually provide a mechanism to save the VM state. However, they only save the state, typically to a file, without considering performance. More sophisticated checkpointing mechanisms can be classified by their intended application:

- General mechanisms such as default VMM checkpointing mechanisms or the CoW-approach mentioned above merely save a checkpoint to a file in order to load it afterwards.

- Migration or replication oriented mechanisms such as Remus transfer a checkpoint to another host if an event or user command occurs in order to clone a VM (migration), or keep a cloned VM up-to-date at a certain interval (replication).

- Other checkpointing mechanisms, such as the one developed in this thesis, also require for a distribution of checkpoints or a checkpoint storage, from which hosts can obtain checkpoints.

Another possible classification can be made by the performance aspects that were optimized by the mechanism:

Downtime High checkpointing frequencies, as required by SimuBoost, can be achieved by decreasing the time required to save or transfer the checkpoint. This can be achieved by decreasing the data amount or by using a CoW approach.

Data amount When a high number of checkpoints is taken, the amount of data to be saved, stored, and loaded becomes more important. To reduce the data amount, two popular strategies are:

1. Incremental checkpointing: By only checkpointing data that was changed since the last checkpoint, downtime and data amount can be greatly decreased.

2. Deduplication: The amount of data to be collected, transferred, and stored can be minimized, especially by deduplicating it prior to the checkpointing. This is accomplished by eliminiating duplicate data, i.e., memory pages or block device sectors that exist more than once.

2.4 Hash Functions

Since the checkpointing approach presented in this thesis uses hash-based deduplication, this section gives a short introduction to hash functions and collision probabilities.

In general, a hash function is a compression function $H : \{0,1\}^* \rightarrow \{0,1\}^n$ [16], which relates the input of arbitrary length to a fixed-length (n) output. A good hash function is considered having as few collisions as possible, i.e., only "few" collisions $H(x) = H(x')$ exist for two distinct input values $x \neq x'$. Also, as collisions obviously cannot be avoided completely, the hash function should distribute the output well over all possible values. Similar to hash functions in cryptography, collision-resistance is crucial to a hash function that is used for a hash-based checkpointing approach: Since data is addressed by hash, a collision would result in at least one page or sector that cannot be restored properly when loading the checkpoint, probably rendering the checkpoint and subsequent checkpoints useless. Therefore, collisions have to be avoided. The problem of finding a collision for k randomly chosen inputs and codomain size $N = 2^n$ is related to the *birthday problem* [16]. The probability

for such a collision, p, can be approximated with $p > 1 - (1 - \frac{k}{2N})^{k-1}$ [27]. Using this approximation, Table 2.1 lists collision probabilities for different conditions, i.e., the probability that – using an n-bit hash function – a collision occurs among k hashes.

$N = 2^n$	k	p
2^{32}	$8 \cdot 10^4$	0.52
2^{32}	10^7	1
2^{64}	$8 \cdot 10^4$	$1.73 \cdot 10^{-10}$
2^{64}	10^7	$2.71 \cdot 10^{-6}$
2^{128}	$8 \cdot 10^4$	$9.40 \cdot 10^{-30}$
2^{128}	10^7	$1.47 \cdot 10^{-25}$

Table 2.1: Collision probability of hash functions for different output sizes and hash count.

2.5 NoSQL Databases

Because the approach presented in this thesis uses NoSQL databases to store the checkpoints, this section gives a short overview on different types of databases and popular examples.

To store large amounts of data, relational or non-relational databases are often used. Non-relational databases can be classified by the way they store data [9]:

- *Key-value stores* use a simple data model, which uses at least pairs of keys and values to address data; some systems provide additional data types. The simpler data model usually results in better performance of queries. However, key-value stores are not sufficient for every use case.
- *Document stores* use an ID to address more complex data (documents), for example in the JSON format. Allowing nested objects and data types like arrays, boolean values, etc., they can be much more complex than key-value stores.
- *Column-family stores*
- *Graph databases*

NoSQL databases are generally simpler, faster, and more scalable than relational databases, since no relational connections need to be maintained. Also, compared

to relational databases, they do not suffer from the overhead resulting from complex querying and management functionality and are much more efficient and resource-inexpensive [8]. To achieve such high efficiency with storing and retrieving operations, key-value stores often keep part of the data, e.g., the keys, in memory; some stores keep the complete data set in memory [5].

Redis is a popular, open-source key-value store, yet offering more powerful data structures such as lists, sets, and hash maps [25]. It supports atomic operations for appending to a list, incrementing values, etc. Redis is an in-memory key-value store, i.e., it keeps the complete data set in memory. Additionally, the data is copied to disk on certain conditions based on time or the number of writes, adding persistence to the otherwise volatile storage. The most important characteristics of Redis are very fast operations, since they take place in memory, but also that the data set should not exceed the memory available to Redis in order to keep the performance high. Redis uses – being a key-value store – relatively simple text-based queries, such as `SET <key> <value>` and `GET <key>`. To speed up multiple queries, Redis offers a pipelining mode, where queries are collected and sent out in one block, i.e., instead of multiple queries and responses, only a single query and response are used. This is particularly useful for inserting or retrieving data in bulk, because it reduces the overhead of the network communication with the database server.

Another popular open-source database is *MongoDB*, which is an on-disk document store [9]. Data is represented using the *BSON* format, which is *JSON*-oriented, but supports binary data. It is then stored in tables that can contain multiple collections. MongoDB also provides capped collections, which are – contrary to regular collections – limited in size and are represented similarly to a ring buffer, thus overwriting old entries if the maximum size is reached. Using such a collection increases the performance of queries, but implies that data either not reaches the maximum size or gets lost. Also, indices are used in order to increase performance. Queries to MongoDB are more complicated compared to Redis queries, because of the more complex data structures used. For example, `db.<collection>.insert( { <key>: <value> } )` inserts data into a specific collection. Similar to Redis, the MongoDB libraries support pipelining of data, either by concatenating multiple BSON objects, or a more complex query including search terms, conditions, etc. Another aspect that affects database performance is the *write concern*: For write operations such as `insert` or `update`, MongoDB provides adjustable levels of acknowledgement using two variables w and j: w causes MongoDB to ignore errors ($w = -1$), perform an unacknowledged write ($w = 0$), i.e., not waiting for the data to be written on disk, or an acknowledged write ($w = 1$), where the library waits for a successful write. Additionally, the journaled write concern $j = 1$ can be used together with $w = 1$ in order to wait until MongoDB has written the data to its journal, which provides additional reliability.

3 Analysis

To speed up functional full system simulation, *SimuBoost* [26] uses a hardware virtualization node and multiple simulation nodes. In a fixed interval, checkpoints of a virtualized system are taken and distributed to the simulation nodes. Each of the nodes then starts a simulation and analysis of the execution for one interval.

3.1 Requirements

This use case leads to the requirements that are described in the following sections: The checkpointing interval affects the data amount, which in turn influences the virtual machine (VM) downtime. Furthermore, requirements to checkpoint distribution are explained.

Checkpointing Interval Because SimuBoost distributes the checkpoints to multiple simulation nodes, the virtualization node needs to provide multiple simulations with checkpoints. The checkpointing interval, i.e., the fixed time between two consecutive checkpoints, directly influences the data amount and therefore the time each simulation interval consumes. For SimuBoost, a lower checkpointing interval in the range of seconds (1–2 s) was found to result in a smaller overhead [26]. At higher intervals, the parallelization becomes sub-optimal; at lower intervals, the VM downtime becomes a greater overhead, since more checkpoints have to be taken.

Data Amount The checkpointing mechanism has to take and store a high amount of checkpoints, especially when the virtualization proceeds slower than the simulations and the checkpoints accumulate. In order to save a large number of checkpoints, the data amount has to be minimized. Additionally, a minimal data amount per checkpoint allows for a faster distribution of single checkpoints.

Downtime While a checkpoint is taken, the VM usually has to be suspended for a moment and started again later in order to save a consistent state of the system, causing a downtime for the VM. For SimuBoost, this downtime should be as short

as possible, because it constitutes a constant overhead in the virtualization stage and reduces the speedup of the parallelized simulation.

Distribution Deduplicated and incremental checkpoints need to be stored temporarily until requested by a simulation node and until they are not longer required to restore subsequent checkpoints. On a request, the complete system state needs to be loaded, i.e., not only the incremental checkpoint but also data from previously saved checkpoints is needed to restore the system state. Thus, all previous checkpoints that contain data used in a later checkpoint must still be available.

To meet the most important requirements regarding checkpointing performance, *data amount* and *downtime*, incremental checkpoints can be taken. Incremental checkpointing is a widely used technique in checkpointing mechanisms. By only saving the modified parts of a system state, e.g., modified memory pages instead of the whole memory, the size of a checkpoint can be reduced.

3.2 Modification Rate

With an incremental checkpointing mechanism, the strongest impact on the data amount is the rate at which memory pages and block device sectors are written per checkpointing interval. Therefore, this section analyzes the modification rate of data for different workloads during an interval.

The modification rate firstly depends on the checkpointing interval, since obviously more data can accumulate in a longer interval. The amount of accumulated data per time, however, highly depends on the workload. Observing the amount of dirty pages and sectors, e.g., during desktop usage, a kernel build, or benchmarks, shows characteristics like memory-intensive or I/O-intensive workloads. The modification rate is a first measure for the data amount of a checkpoint, since it represents the incremental data amount. Figure 3.1 shows the amount of dirty pages and sectors per 2000 ms interval during desktop usage, i.e., starting and using applications such as a file manager, Firefox, LibreOffice, etc. The VM was running Ubuntu 12.04, with 2 GiB RAM and a 10 GiB qcow2 image. It shows 3388–54 269 dirty pages (avg=24 237, median=23 203, sd=13 993) per checkpoint with multiple spikes, especially when applications are starting, and 0–15 104 dirty sectors (avg=2165, median=768, sd=3232), with some spikes, e.g., when files are saved.

In Figure 3.2, the amount of dirty pages and sectors during a kernel build is shown, using the same setup as above. While having a higher average workload, this workload shows less spikes. Also, significantly more block device activity occured, resulting in a higher number of dirty sectors. Dirty pages are at 4600–48 900

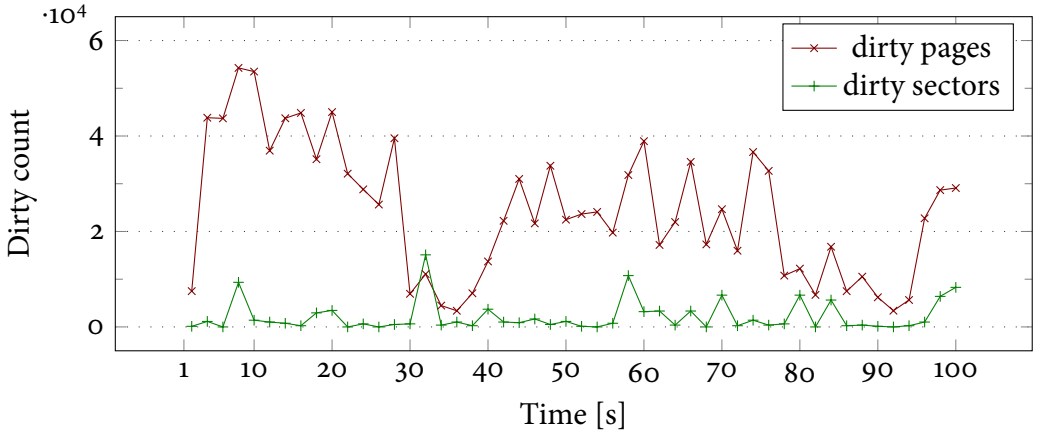


Figure 3.1: Dirty count of pages and sectors during desktop usage, running in a single-core virtualized Ubuntu 12.04 with 2 GiB RAM and a 10 GiB qcow2 image.

(avg=25 352, median=22 777, sd=7878) and dirty sectors at 0–45 100 (avg=4474, median=2112, sd=7536).

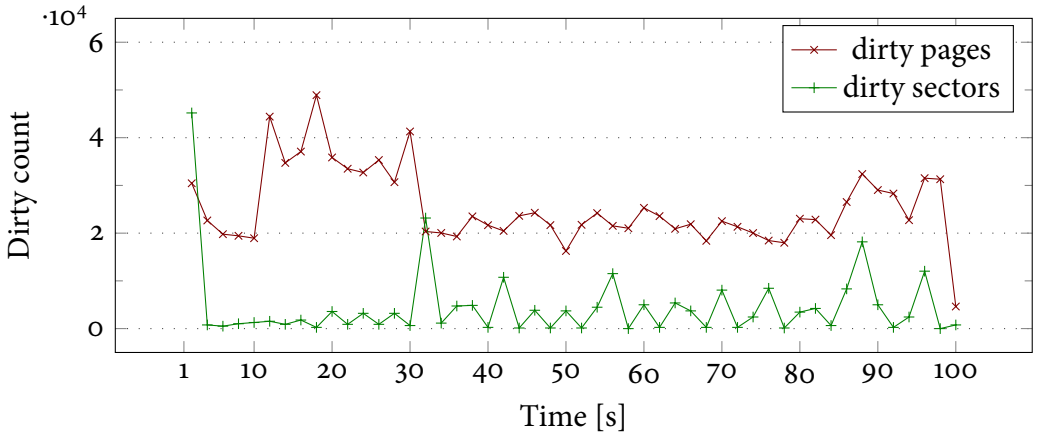
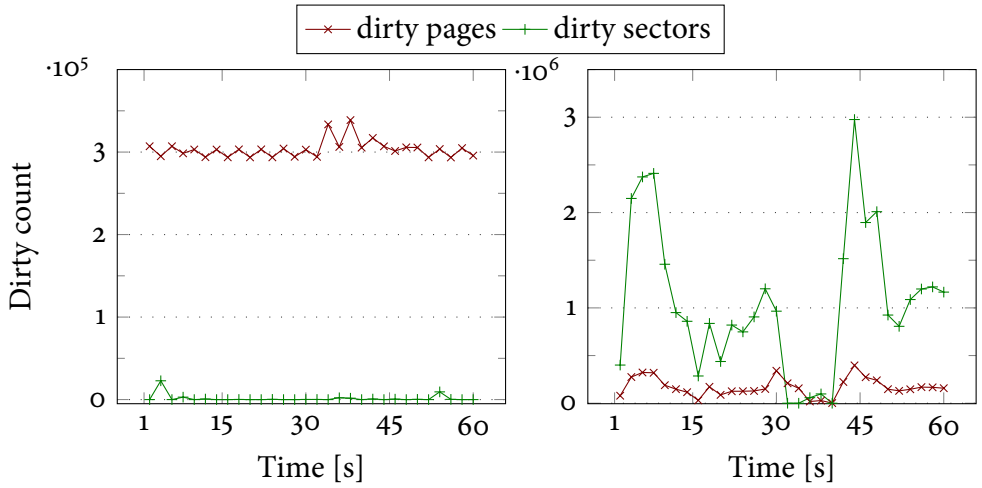


Figure 3.2: Dirty count of pages and sectors during a kernel build, running in a single-core virtualized Ubuntu 12.04 with 2 GiB RAM and a 10 GiB qcow2 image.

These measurements show a great potential of incremental checkpointing to reduce the data amount, because only 5–10 % of the total memory have to be checkpointed. However, the data amount depends on the modification rate, which in turn heavily depends on the workload. Using workloads with a higher memory load such as the *STREAM* benchmark, which is shown in Figure 3.3a, or a much higher I/O load such as the *Bonnie++* benchmark, which is shown in Figure 3.3b, results in a heavily increased dirty count. Such high workloads can cause difficulties when using short checkpointing intervals, because the data amount for every checkpoint –

and therefore also the downtime – is higher. These benchmarks lead to dirty pages of 290 000–338 000 (avg=303 675) and dirty sectors of 0–2 975 000 (avg=1 054 901) in every 2000 ms interval.



(a) STREAM benchmark

(b) Bonnie++ benchmark

Figure 3.3: Dirty count of pages and sectors during STREAM and Bonnie++ benchmarks, running in a single-core virtualized Ubuntu 12.04 with 2 GiB RAM and a 20 GiB qcow2 image.

3.3 Data Duplication

In virtualization, memory often gets deduplicated to increase memory utilization, e.g., to increase a host's capacity for VMs. For example, *Kernel Samepage Merging* (KSM) [1] is often used together with QEMU and KVM to find and share duplicate memory pages in a system. The KSM deduplication was improved by Miller et al. by prioritizing deduplication of pages with typically high redundancy through hints from the I/O layer [23]. Barker et al. found self-sharing opportunities, i.e., pages that can be deduplicated in a single VM, to be on average at 14 %, excluding zero pages [3]. For multiple VMs, Gupta et al. measured that memory deduplication potential between VMs lies between 50 % for heterogenous workloads and 60 % for homogenous workloads [12].

Since VMs generally offer potential for deduplication, this section examines whether parallels exist between duplication among multiple VMs and among multiple incremental checkpoints. This deduplication opportunities could then be used to further reduce the data amount for checkpoints. Among multiple checkpoints, different types of duplicates in memory and block devices exist that are potentially

essential for this checkpointing approach and need to be differentiated. Similar to the naming scheme of Barker et al. [3] for memory sharing opportunities, in this thesis, deduplication opportunities are separated into two categories:

1. *Intra-checkpoint duplication*: Two or more pages or sectors contain equal data at the same point in time, i.e., in the same checkpoint data is saved more than once.

2. *Inter-checkpoint duplication*: At different points in time, two or more pages or sectors contain equal data, i.e., data is repeatedly written throughout multiple checkpoints.

To show the potential of inter-checkpoint deduplication, both intra- and inter-checkpoint duplication is measured along with the total number of modified data, i.e., data of incremental checkpoints. Figure 3.4 shows the percentage of intra- and inter-checkpoint duplication of memory pages during a kernel build. Page quantities were measured in intervals of 2000 ms using 2 GiB RAM. Inter-checkpoint duplicates were measured among the last ten intervals in each interval. Intra-checkpoint duplicates are at under 10 %, while intra-checkpoint duplicates are at 15–55 %.

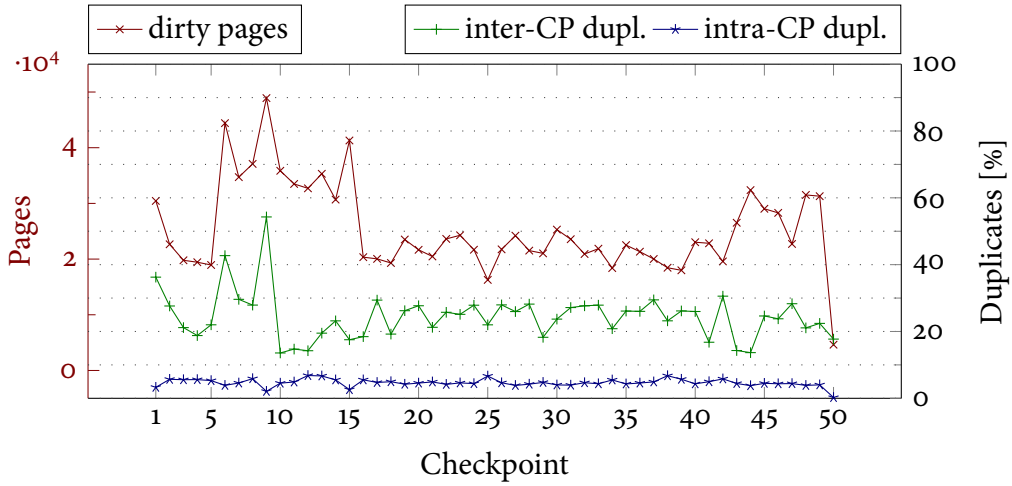


Figure 3.4: Percentage of intra- and inter-checkpoint duplicates in dirty pages during a kernel build, measured at 2000 ms intervals.

Such a large number of deduplication opportunities, especially that of inter-checkpoint duplicates, suggests that deduplication can be used for the checkpointing mechanism in this thesis to reduce the data amount beyond the savings achievable through incremental checkpointing.

3.4 Conclusion

This chapter analyzed the requirements of a checkpointing mechanism for SimuBoost as well as the potential data reduction by examining typical modification rates and duplication. The modification rate showed that incremental checkpointing reduces the relevant data amount to only 5–10 %, depending on the workload. By analyzing duplication in checkpoints, especially throughout multiple checkpoints, a promising percentage of duplicates was found that can possibly be eliminated to further reduce the data amount and downtime.

4 Design

When taking checkpoints at a high frequency, it is vital to focus on optimizing the performance, i.e., to keep data amounts and latencies small. The following design approach includes several concepts to optimize performance, which – besides a general overview of the checkpointing mechanism – are presented in this chapter.

The main focus of the mechanism is in taking frequent checkpoints at an interval in the range of seconds to supply full system simulations with system states, while taking measures to optimize the data amount for distribution, and finally enable to distribute them. The checkpointing mechanism is intended for a producer-consumer use case: One virtualized system creates checkpoints, which are stored temporarily in a database and are later retrieved by simulation nodes. The amount of stored checkpoints, however, heavily depends on the performance of consuming nodes, since the speed at which checkpoints are produced and consumed has to be mostly balanced.

4.1 Incremental Checkpointing

As shown in Chapter 3, incremental checkpointing can decrease the data amount of a checkpoint to only 5–10 %. Therefore, the checkpoints taken by the presented approach are taken incrementally, only including modified data in each checkpoint. The incremental checkpointing concept, however, makes loading of checkpoints rather difficult: To load a certain checkpoint, data that has remained unchanged for a long period has to be collected from previous checkpoints, possibly reaching to checkpoints taken at a much earlier interval.

4.2 Checkpoint Metadata

To solve the problem of difficult loading, this design separates single data entitites, i.e., memory pages and block device sectors, from the checkpoint metadata, which describes the data entities belonging to a certain checkpoint. Then, references to the entities instead of the actual data is saved in the checkpoint metadata. This simplifies

both saving and loading: Data structures to keep track of pages and sectors can easily be maintained and saved with the checkpoint; loading a checkpoint requires retrieval of a fixed amount of metadata, independent of the modification time of pages and sectors. This is accomplished by referencing all entities that belong to the checkpoint, i.e., the complete set of pages and sectors in use at the time of checkpointing are referenced. The approach in this thesis collects this metadata in three types of headers:

- *Memory headers*, which reference each page that is present in the system.
- *Block headers*, which reference each modified sector not present in the initial disk image, i.e., all sectors that differ from the disk image.
- *Device headers*, which contain the complete information – besides memory and block devices – needed to restore the system state. This header is not referencing information, since device states are relatively small, i.e., about 80 KiB, depending on the VM configuration.

However, this approach implies particular problems: By including references to recurring data entities repeatedly regardless of modification, checkpoint headers include duplicate references. This overhead can be considered small, since references to the data entities are much smalller than the data itself. For example, a VM with 2 GiB RAM and therefore 524 288 pages (using a page size of 4 KiB) that have to be checkpointed and 10 % of modified pages (52 429) results in 205 MiB of modified data, but – using a 64-bit hash function – in only $52\,429 \cdot 64\,\text{bit} = 410\,\text{KiB}$. Another problem are the references, which have to uniquely identify a data entity.

4.2.1 Hash Functions

Values that uniquely identify any data can be computed by using a hash function. Applied on the data entities, a hash function – if chosen appropriately regarding the collision avoidance parameters as explained in Section 2.4 – returns a unique identifier for this data. After hashing the data entities, they can be referenced by their hash value. This, however, leads to the problem of finding the right hash function to avoid collisions. Since the collision probability of a hash function depends on both the codomain size and the quantity of data entities, the quantity needs to be known in advance to choose an appropriate hash function. The design presented in this thesis does not generally limit the number of checkpoints, although most of the measurements were taken with 50 checkpoints stored at the same time. Using the average rate of modified pages and sectors from Section 3.2 and the average deduplication rate from Section 3.3, the amount of data entities that exists at the same time can be estimated to not exceed $1.31 \cdot 10^6$ ($= 0.75 \cdot 35\,000 \cdot 50$, with 25 % duplicates,

35 000 modified data entities and 50 checkpoints). Using the approximation from Section 2.4, this results in a collision probability of about $4.07 \cdot 10^{-8}$ for a 64-bit hash function. If more checkpoints need to be available at the same time, the choice of a hash function has to be reconsidered.

4.3 Deduplication

Using hashes to identify single data entities brings another advantage: Hashes of already stored data can be saved and used to deduplicate data in subsequent entitites: By comparing a hash with all previously stored values, duplicates can be found. This can be accomplished by using a cache that stores sent hashes and allows search and insert operations. Then, hashes are first searched and, if no duplicate was found, inserted. Using such a cache results in hash-based deduplication of pages and sectors as long as the corresponding hashes stay in the cache. Figure 4.1 shows the basic procedure for deduplicating data. A data entity is first hashed, then its hash is used to search the cache. If an entry with the same hash was found (*cache hit*), its index is updated to denote the time of the cache hit. If no entry was found (*cache miss*), the hash is added to the cache and sent to the database. In both cases, the hash is added to the corresponding header afterwards.

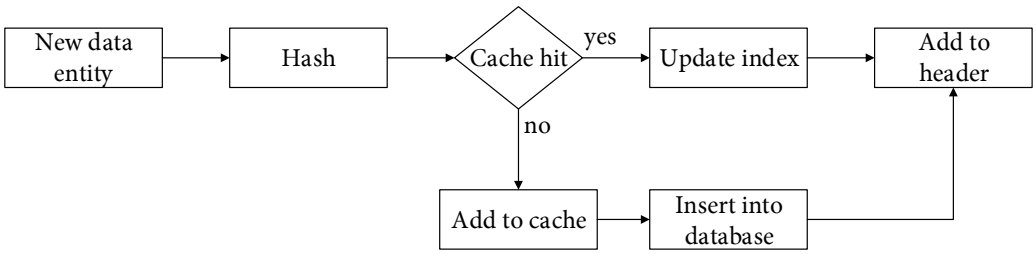


Figure 4.1: Procedure for deduplicating data using a hash-based cache. On a cache hit, its index is updated; on a cache miss, it is sent to the database.

4.4 Checkpoint Storage

After incrementally obtaining data and deduplicating it, the checkpoints have to be distributed. For SimuBoost, it needs to be accessible in a central location, from where simulation nodes can obtain checkpoints. To eliminate the need for prior knowledge about the targets of distribution, the checkpoint mechanism should simply provide checkpoints, and not control where the checkpoints are loaded. This requirements can be met by a shared filesystem. This way, however, data needs to be

searched for by filenames, e.g., to find a data entity with a specific hash. Therefore, a database comes into consideration, because data can be stored and accessed efficiently with all advantages databases have over shared filesystems. Selecting an appropriate database type and storage model requires knowledge of downtime and space consumption requirements of the checkpointing mechanism: Key-value stores usually keep data in memory and therefore offer faster access times, but also limit the data set to the size of available memory. To save many checkpoints both quickly and reliably in a database, an on-disk store can be used.

The strategies described in Sections 4.1 and 4.3 require checkpoint headers and checkpoint data to be saved separately. In order to increase the throughput of queries, the database layout should also separate headers and data, since only a few headers per checkpoint are created. The layout used for this approach is shown in Figure 4.2. Headers can be accessed by a sequentially increased index, while data can be accessed by its hash. The headers itself contain the hash values and in this way reference the data needed for the checkpoint.

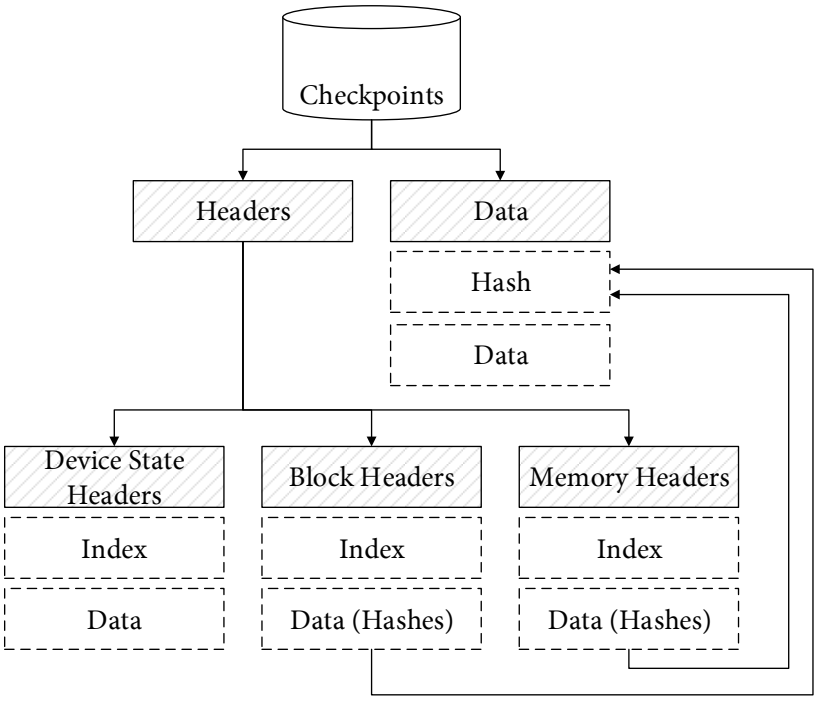


Figure 4.2: The database layout. Headers are accessed by a sequentially increased index and reference data entities, which are accessed by their hash.

To keep the time needed for saving a checkpoint to the database as low as possible, a fast insert mechanism needs to be used. As explained in Section 2.5, such batch or pipelining operations can greatly improve the performance of large

insert queries. Additionally, it is necessary to ensure sufficient disk space for the checkpointing mechanism to work properly.

4.4.1 Retention Policy

After the simulations were started using the checkpoints and they are no longer required, they cannot be easily deleted, since headers and data entities are separeted. This can be solved using a different approach: The amount of checkpoints stored in the database is fixed and a retention policy is used. Only a fixed amount of the most recent checkpoints is saved and data from previous checkpoints is deleted at certain intervals. This requires a retention value to be set as well as a mechanism to search for and delete entries older than the retention value. Data that is used in more recent checkpoints, however, must be retained: Because data is split from the checkpoint headers and deduplicated, multiple headers can reference the same data. Only data that is referenced by no checkpoint more recent than the retention value can be deleted.

4.5 Conclusion

This design includes the following checkpointing concepts:

- Incremental checkpointing
- Hash-based deduplication of memory and block device data
- Checkpoint distribution via a database

The basic architecture is depicted in Figure 4.3. It shows a VMM, in which the checkpointing mechanism resides for convenient access to all system states and modified data. Data is obtained incrementally, memory and block device data is hashed and then compared with previous data for deduplication. Then, headers and deduplicated data are sent to the database and can later be loaded from a checkpointing mechanism that resides in a simulation node.

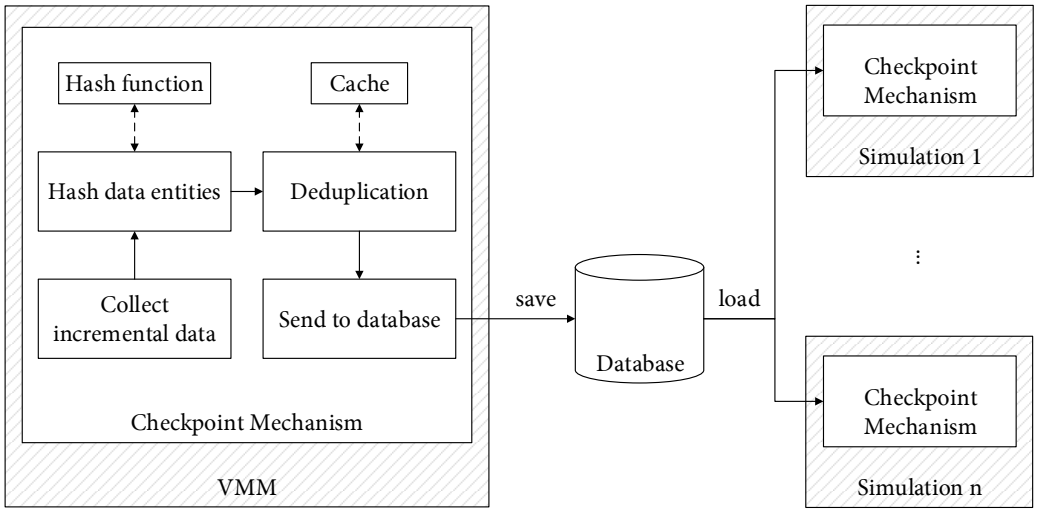


Figure 4.3: Overview of the checkpointing mechanism inside the VMM and the checkpointing operation, and the constellation of the saving virtualization node and the simulation nodes.

5 Implementation

As mentioned in Chapter 4, the checkpointing mechanism should be integrated into the virtual machine monitor (VMM) to make use of internal functions for collecting modified data. The implementation developed in this thesis uses QEMU 1.5.50/KVM (Linux kernel 3.8.0) for virtualization and MongoDB 2.0.6 for storing checkpoints. This chapter describes the implementation, beginning with the integration into QEMU. Further, data collecting as well as the structure of a checkpoint and database-specific details are explained.

5.1 Collecting Data

The first step for the actual checkpoint procedure is to collect all incremental data. Therefore, this section describes how QEMU internally handles and represents memory, block devices, and device states and how the presented checkpointing mechanism collects the data that is needed for a checkpoint.

5.1.1 Memory

Since main memory that is available to the guest is – as explained in Section 2.2 – managed by QEMU, its internal memory model can be used: The memory allocated by the QEMU process is organized in *MemoryRegion* objects, each representing a contiguous region of memory. All memory that is available to the guest is organized in *RAMBlocks*, which, for example, represent the main memory or video memory, and each occupy one MemoryRegion. The RAMBlocks are organized as a linked list, which allows iterating over them. Inside these RAMBlocks, pages can be accessed using the base address of a RAMBlock's MemoryRegion and an offset to the actual page, which is a multiple of `TARGET_PAGE_SIZE` (i.e., 4 KiB for x86 and x86-64 guests). These addresses are mapped inside QEMU's virtual address space. Figure 5.1 depicts QEMU's memory model, as well as the linked list structure and how memory pages can be accessed.

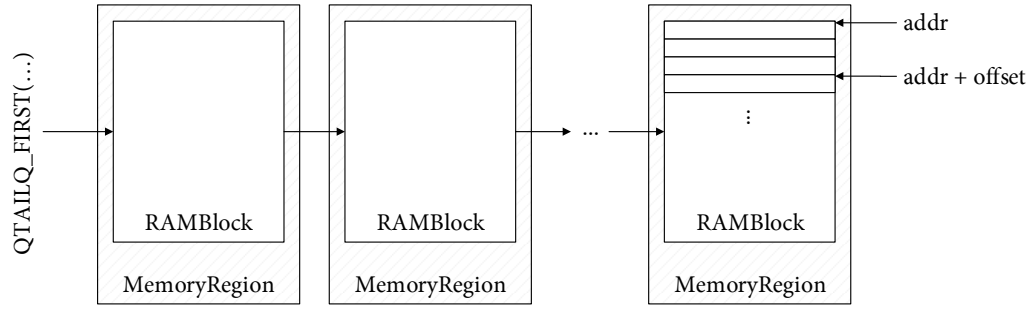


Figure 5.1: QEMU MemoryRegions and RAMBlocks. Each RAMBlock is placed inside a MemoryRegion and can be accessed with an address and offset. RAMBlocks are organized as a linked list.

Similar to the QEMU's built-in checkpointing approach, the implemented checkpointing mechanism uses QEMU's `QTAILQ` macros to access the linked list elements, e.g., `QTAILQ_FIRST(&ram_list.blocks)` returns the first RAMBlock in the list. In order to save an incremental checkpoint of the memory, dirty pages in all RAMBlocks need to be saved, which requires a mechanism to detect modified pages. Therefore, pages are iterated over using the `migration_bitmap`, which is already implemented in QEMU and used during the built-in pre-copy-based migration mechanism for iterating over memory pages. When modifying a page, QEMU sets the corresponding bit in order to mark the page as modified. Using this bitmap, QEMU provides efficient functions, e.g., to find and reset the next dirty bit inside the bitmap. Calling this function on a MemoryRegion returns the offset of the next modified page, which is – besides the base address – all that is neeeded to access it. QEMU provides a function to quickly check wheter a memory page is a zero page, i.e., it contains only zeros. These zero pages can be handled differently. If a zero page is found, its hash is saved and for all subsequent zero pages, the hash does not need to be calculated again. Additionally, zero pages are saved differently: Instead of saving a page of zeros in MongoDB, a boolean property in the memory data documents is set to mark the hash as zero page hash. When loading a checkpoint, zero pages are recognized using this property. The hash of such a zero page is saved locally during the loading process to identify subsequent zero pages, which makes the loading more efficient.

5.1.2 Block Devices

QEMU supports multiple block device formats and options for including them in a virtualized system. For checkpointing of block devices, the implementation makes a few assumptions to simplify the process:

1. An identical initial version of block device images is already present on both saving and loading hosts; a checkpoint includes changes since this base image. Because these initial images can be very large, it would cause an excessive load on MongoDB to store the complete image in the first checkpoint.

2. QEMU is started in snapshot mode by using the `-snapshot` option, which holds all changes to an image in a temporary file and only applies them when the `commit` HMP command is invoked. Using this snapshot mode, dirty sectors can be found much faster, since only the overlaying change image needs to be iterated.

3. Only writeable and currently in-use block devices are checkpointed. Because of the first assumption, an initial image of all block devices is already present when loading a checkpoint and read-only block devices cannot be modified. Disabled block devices are not needed by the loading system and if they are enabled later, they will get checkpointed as well.

With this assumptions, a very lightweight strategy can be used to perform checkpointing of block devices. Figure 5.2 shows how block devices are accessed if QEMU's snapshot mode is used: All write operations are performed using the temporary image file. For a read operation, two cases are possible: If a sector is allocated in the temporary image file, it is read from there; if the sector is not allocated, it is read from the backing file, i.e., from the initial image. When committing changes, the sectors from the temporary image are written to the initial image.

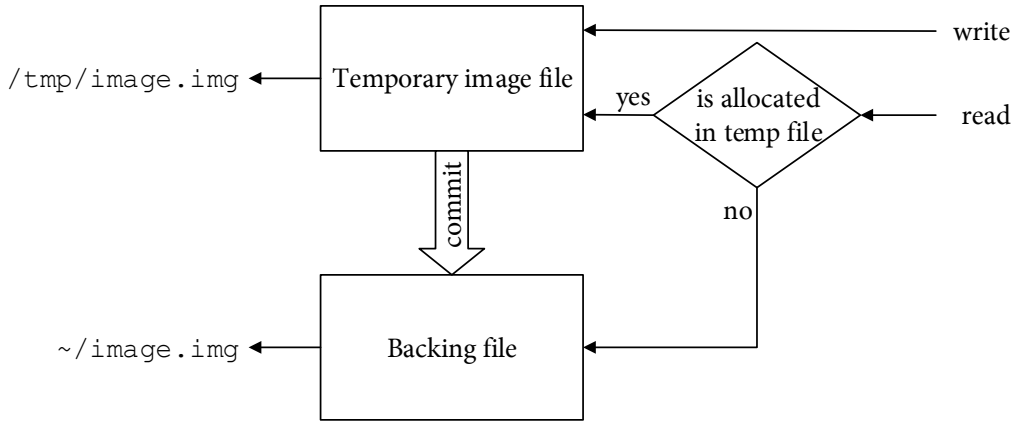


Figure 5.2: Block device read and write operations using QEMU's snapshot mode. Write operations are performed on the temporary image file, while read operations access both image files, reading from the corresponding file.

At the beginning of the checkpointing mechanism, dirty tracking is set up for all eligible block devices. In every checkpoint, all block devices are iterated over, search-

ing for dirty sectors. In the first checkpoint, all allocated sectors of writable block devices are saved, essentially saving the changes made to the initial images. This works because QEMU's snapshot mode is used, which creates `BlockDriverState` objects with a temporary image file for every initial image. If a sector in this temporary image is allocated, it is differing from the initial image. To collect incremental data for consecutive block device checkpoints, only dirty and newly written sectors of these temporary allocated sectors have to be iterated.

After finding such a sector, QEMU's functions for sequential reading and writing are used to read a number of sectors at once, which saves more iterations. This number of sectors, n, is determined by a call to the `bdrv_is_allocated()` function, which checks whether the next sectors are allocated and returns the number of consecutively allocated sectors n. Then, n sectors are read into a buffer using the `bdrv_read()` function.

5.1.3 Device States

The approach mentioned above already covers memory and block devices. For a loadable, complete system state, device states are also required. Since QEMU already provides a checkpointing mechanism and device states constitute a very small data amount (about 80 KiB), this mechanism can be used to checkpoint all device states. This has advantages in terms of minimal required changes to the original QEMU code, while the full range of devices with QEMU support is also supported by the checkpointing mechanism in this thesis.

The default checkpointing mechanism of QEMU iterates over all devices and writes their state to a file by either writing a buffer using `qemu_put_buffer()` or a certain amount of bytes, which leads to calls of the `qemu_put_byte()` function. These calls need to be redirected into a buffer that can later be sent to MongoDB instead of a file. To make use of QEMU's checkpointing mechanism for only the device states, invocations of the checkpoint function, `savevm()`, need to ignore memory and block devices while the checkpointing mechanism is running, since this is covered by the mechanisms explained above. The implementation presented here uses a dynamic array for the device states. This way, the device states are not limited in size and can easily be saved in the database.

To load the checkpoints, QEMU's corresponding function for loading checkpoints, `loadvm()`, can be used. The changes to this function are reading from a buffer instead of a file, which can be accomplished by changing the `qemu_get_buffer()` and `qemu_get_byte()` functions analogous to the corresponding write functions.

5.2 Checkpoint Structure

While the checkpointing mechanism collects data, a checkpoint has to be built. As described in Section 4.2, three types of metadata are collected during the checkpointing, which provide the information required to load a checkpoint, and saved in headers. The device state header consists of only the data described in Section 5.1.3. For memory and block devices, more complex data is required, which is stored in C structs.

5.2.1 Memory Headers

Listing 5.1 contains the C structs that hold the memory metadata. For each RAMBlock, a RAMInfo is created, containing the ID string, which is used to identify RAMBlocks, the number of pages in this block, and an array of hashes. For each exisiting page – whether it was marked dirty at the time the checkpoint was created or not – its hash value is included in the memory header. This prevents searching for data in previous checkpoints when the checkpoint is loaded and simplifies the saving process. This does not decrease efficiency, because only modified pages have to be hashed and written to a copy of the previous RAMInfo objects.

```
1  typedef struct {
2      char idstr[256];
3      uint64_t num_pages;
4      hash_t *hashes;
5  } RAMInfo;
6
7  typedef struct {
8      size_t size;
9      uint64_t num_blocks;
10     uint64_t num_pages;
11     RAMInfo *blocks;
12 } RAMHeader;
```

Listing 5.1: C structs for memory metadata. The RAMHeader holds general information and pointers to the RAMInfo objects, which in turn contain hashes for each page of a RAMBlock.

To send the memory metadata as one contiguous binary entry to the database, it is helpful to allocate a single block of memory for the structs, since their space

consumption can be computed in advance: For each RAMBlock, a RAMInfo is needed and for each page, a hash value is needed. Figure 5.3 shows how the memory metadata is allocated in memory: The `blocks` pointer in the RAMHeader object references the first RAMInfo, which in turn contains a reference, `hashes`, to the first hash value of RAMInfo.

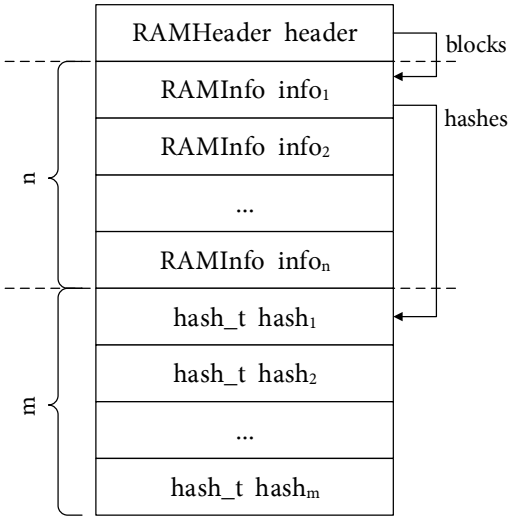


Figure 5.3: RAMHeader and RAMInfo objects in memory. The metadata contains a RAMInfo object for each RAMBlock and a hash value for each memory page.

5.2.2 Block Device Headers

The approach implemented for memory headers cannot be used for block device metadata, because the amount of sectors that need to be included in the checkpoint, i.e., the sectors that differ from the initial image, constantly changes. Also, preallocating a complete block device header would produce an unnecessary overhead. Therefore, a more dynamic header is needed for block device metadata: The `BlockHeader` struct, which contains all information needed to restore a single block device, is shown in Listing 5.2. It contains the block device's name, by which it is identified, and sector addresses as well as hash values for each changed sector. Both addresses and hashes are stored using dynamic arrays, which are increased in size if they reach the maximum index. Therefore, index and size variables are used to point to the first free field in the `sector` and `hash` arrays, and to keep track of the maximum amount of fields allocated for the arrays.

Saving the BlockHeaders in the database relies more on the database layout than the RAMHeader, storing the checkpoint index, device name, sector array, and hash array in a MongoDB document.

```
typedef struct {
    char device_name[32];
    uint64_t index;
    uint64_t size;
    int64_t *sector;
    hash_t *hash;
} BlockHeader;
```

Listing 5.2: C struct for block device metadata. For each device, a BlockHeader contains the sector address and hash value of each sector that differs from the initial image.

5.2.3 Data

The data itself, i.e., actual memory pages and block device sectors, have to be read from the VM's memory and overlay images, respectively. After taking measures to collect the data addresses and therefore gain access to the data itself as described in Sections 5.1.1 and 5.1.2, the next step is hashing the data. The hashing is accomplished using the *CityHash64* function [6], which is an appropriate hash function as discussed in Section 4.2.1. If the hash value leads to a cache miss in the deduplication cache (see Section 5.2.4), and the corresponding data is therefore eligible to be sent to the database, the data is written to a BSON object and added to a temporary buffer. If the size of this buffer reaches the previously defined limit `CMD_LIMIT` or at the end of a checkpoint, it is flushed, sending out all BSON objects to MongoDB. This usage of MongoDB's bulk insert function reduces the number of single requests and speeds up the inserting process.

5.2.4 Cache

In order to deduplicate the data, a hash-based approach is used, i.e., data is compared by its hash value to find duplicates. Because of very fast search and insert operations, trees are an appropriate data structure for this type of cache. For the tree operations, the *GNU libc tsearch* functions [19] are used, which use red/black trees. The elements of the tree are structs containing the hash and an index, which indicates the checkpoint number in which the corresponding data was last sent to the database. The following strategy is used to search and – if necessary – insert hashes into the tree:

1. Run a search over the tree in order to find the relevant hash.

2. If a hash was found (cache hit): Update its index to the current checkpoint number.
   If no hash was found (cache miss): Insert the hash into the tree.

Additionally, a cache cleanup is performed in each checkpointing interval in order to meet the retention policy: If the hash was last touched (inserted, or updated on a cache hit) at a checkpoint older than the retention value, i.e., if its index i satisfies $i < cur - ret$, with cur = current checkpoint index and ret = retention value, it is deleted from the cache. For SimuBoost, ret has to be set dynamically to keep all checkpoints that were not already loaded.

5.3 Database

The data that is collected, deduplicated, and buffered as described in the previous sections, finally has to be sent to the database. To decrease the downtime, all headers are sent after the VM was resumed. The memory and block device data is sent as soon as the buffer gets full, or at the end of a memory or block device checkpoint, respectively. This is done by using the `mongo_insert_batch` function, which is capable of inserting an array of BSON objects into the database. All insert queries are performed using a *write concern* [21] of $w = 0$ and $j = 0$, which results in an unacknowledged query, i.e., the library does not wait for MongoDB to report a successfull insert, but still handles network errors depending on the network configuration of the system.

5.4 QEMU Integration

The checkpointing mechanism is implemented inside QEMU in order to use all internal functions for checkpointing. Figure 5.4 shows an overview of the implementation. To start and stop the mechanism, commands and handlers for the QEMU *Human Monitor Protocol* (HMP) are implemented. The `start-cp` HMP handler initializes data structures and the database connection and then starts a checkpointing thread, which runs for the entire duration of the checkpointing. In this thread, the actual checkpointing takes place in each interval: If the VM is currently running, the checkpoint mechanism stops it. Then, it takes device states, memory and block devices checkpoints. During the memory and block device checkpoints, data is deduplicated and sent to the database. After taking the checkpoint, the VM is resumed, the checkpoint headers are sent to the database as well, and the checkpoint thread waits for the next interval. If the `stop-cp` HMP handler is called, the checkpointing mechanism waits until the end of the current checkpoint and then stops the checkpointing.

5.5 Conclusion

The implementation developed in this thesis relies on strong integration into QEMU. This benefits a simpler collection of incremental data. While collecting the data, headers are built that contain information about which data has to be loaded into a specific page or sector. Memory and block device data is first hashed and then deduplicated using a red/black tree. Memory and block device headers use these hashes to reference the data entities. For device states, the checkpointing mecha-

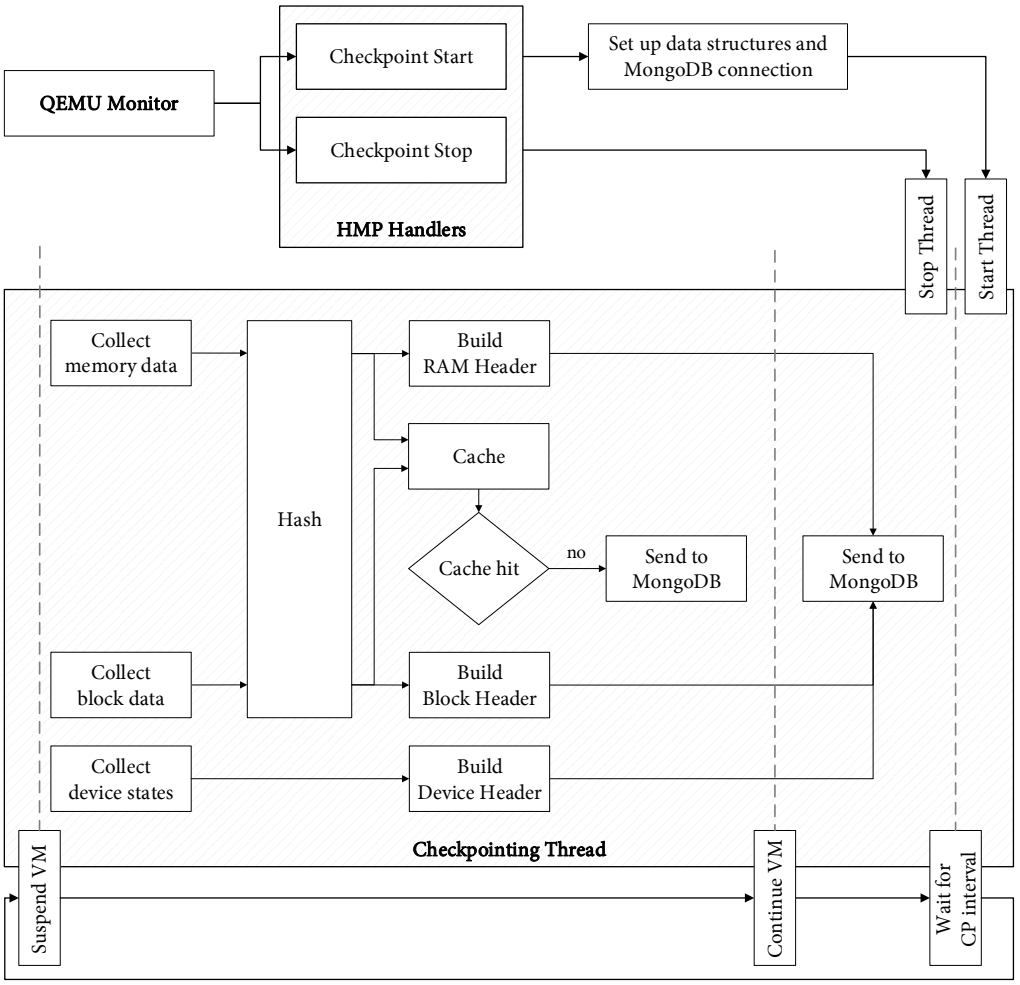


Figure 5.4: Overview of the checkpointing mechanism implementation. Via HMP Handlers, the checkpointing thread is started and stopped. It collects data, separating headers from actual data entities, deduplicates it and sends both headers and data to MongoDB.

nism relies on QEMU's built-in checkpointing function, `savevm()`, to make the checkpointing mechanism work for different device configurations. After collecting and deduplicating the data, checkpoints are saved to MongoDB, where they can be loaded from.

6 Evaluation

The checkpointing mechanism presented in the previous chapters takes incremental and deduplicated checkpoints and stores them in a database. This chapter performs an evaluation of the checkpointing mechanism, first describing the methodology and evaluation setup and later evaluating both correctness and important performance aspects.

6.1 Methodology

To evaluate the checkpointing mechanism, different workloads and benchmarks were executed while taking checkpoints. For each checkpoint, performance data is measured (e.g., dirty pages and sectors, pages and sectors sent, total downtime, sending time, etc.) and written to log files.

The first criterion for the evaluation is the correctness of the presented checkpointing implementation. Since with a checkpoint a system can be restored to the exact state the system had at the time of checkpointing, this state can be compared. Different parts of the system can be examined to assess whether checkpoints restore the system to the exact same state. Section 6.3 performs an evaluation of the checkpointing mechanism's correctness by comparing the system state at different points in time.

The second criterion is the performance of the proposed checkpointing mechanism. Because SimuBoost requires a downtime as small as possible, different strategies were implemented to reduce both data amount by deduplication and downtime of the VM during checkpointing. Using different measurements, Section 6.4 examines and evaluates the checkpointing mechanism's performance.

The benchmarks used for evaluation are a Linux kernel 3.11.6 build and the SPEC CPU2006 401.bzip2 and 471.omnetpp benchmarks. These workloads were chosen because a kernel build presents a workload that stresses memory, CPU and I/O devices, and the two SPEC benchmarks present an easy as well as a difficult case for deduplication: In the bzip2 benchmark, six different files are compressed and decompressed using three differend compression levels [14]. Since compression and

decompression is performed entirely in memory, this is expected to include many duplicates, e.g., because the same file is decompressed three times and therefore the same data is written to memory. The omnetpp benchmark generates network traffic and simulates a large network. This workload is less likely to produce duplicate data and therefore to produce a higher data amount to checkpoint.

As described in Section 5.1.1, zero pages can be treated differently to increase the performance. In the evaluation, zero pages are excluded in the total page count and in the amount of deduplicated pages to only evaluate the deduplication cache approach. Further, the presented numbers of dirty pages and sectors only include incremental data, since this is the data amount that has to be deduplicated. At last, the first checkpoint is excluded in the performance evaluation, because its data amount includes the complete memory and overlay image, which does not yield representative values.

6.2 Evaluation Setup

The workloads used for the evaluation were executed on a test system, which is equipped with an Intel Xeon E3-1230 CPU, 16 GiB 1333 MHz RAM, a 500 GB Seagate Barracuda 7200.12 hard disk and a 128 GB Samsung 830 SSD, running a 64-bit Ubuntu 12.10 system. Both the modified version of QEMU 1.5.50 that includes the checkpointing mechanism implementation and the MongoDB 2.0.6 server are running on this test system. Using the modified QEMU, VMs based on Ubuntu 12.04 are started, which execute the workloads and benchmarks. The MongoDB server is running on the same system as QEMU to achieve a faster transfer and therefore a shorter downtime. For the simulation nodes that load the checkpoint, this increased transfer time while loading is acceptable, since it does not have such a strong impact on the speedup of SimuBoost as the checkpointing downtime [26].

6.3 Correctness

To evaluate the correctness of the checkpointing mechanism, the system state at the time of checkpointing and at the time of loading is compared. The proposed checkpointing mechanism is considered as working correctly, if it restores the system to its exact state when loading a checkpoint. Since device states are checkpointed using QEMU's built-in checkpointing function, it is assumed that they are loaded correctly. To compare memory and block device state, QEMU's HMP commands `pmemsave` and `drive_backup` are used, which copy the complete memory visible to the guest and the overlay image for a block device to a file. First, a workload is executed to reach a certain system state that differs from a freshly booted system. The VM is

then stopped to preserve this state, while taking a system image with `pmemsave` and `drive_backup`. Then, a checkpoint is taken and the VM is resumed, executing another workload to reach a second system state. After loading the checkpoint, a second system image is taken. The checkpoint mechanism is considered as correct within this test, if the memory and block device image files of the two system states are identical.

Besides regular loading after different workloads, the evaluation approach described above is successfully performed. However, there is still a probability for a hash collision as explained in Section 2.4 with $N = 2^{64}$ and k depending on the workload, which could possibly render the complete set of checkpoints useless, if it occurs in the first checkpoint for data that is rewritten in every checkpoint. A collision-free functioning of the checkpointing mechanism cannot be guaranteed, but a hash function can be chosen appropriately to keep the probability very small.

6.4 Performance

A very important aspect of the checkpointing mechanism proposed in this thesis is its performance. In Section 2.3, the two performance aspects *data amount* and *downtime* were specified, which are evaluated in this section. Additionally, an evaluation of the efficiency of the hash-based cache used for deduplication follows.

6.4.1 Deduplication

This section evaluates the deduplication strategy implemented in this thesis. The hash-based cache approach reduces both send time and data amount, therefore its performance, i.e., the reduction of the data amount, is an important aspect. To evaluate the deduplication performance, checkpoints are taken while executing different workloads and measuring the cache hit rate of both memory pages and block device sectors. The cache hit rate effectively reflects the amount of deduplicated data, since every cache hit eliminates the sending of one page or sector. For every checkpoint, the deduplication rate of pages and sectors is measured.

Figure 6.1 shows dirty memory pages and the deduplication rate for 50 checkpoints during multiple runs of a kernel build with 2 GB of memory. The deduplication rate in this runs is 11–52 % (avg=17.50, median=16.38, sd=6.04).

In Figure 6.2, the deduplication rate for block device sectors during the same benchmark is shown. Both dirty sectors and the deduplication rate have strong fluctuations. The deduplication rate in this run is within 7–71 % (avg=26.10, median=16.95, sd=19.80).

Since the proposed hash-based cache approach for deduplication that can make use of recurring data from previous checkpoints, the deduplication rate is signif-

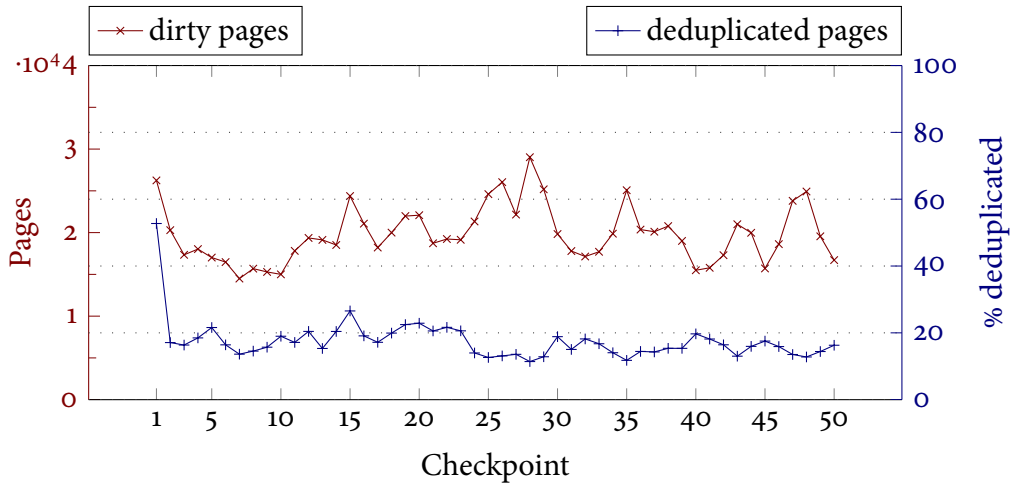


Figure 6.1: Dirty count of pages and percentage of memory deduplication during a kernel build with a checkpointing interval of 2000 ms and 2 GB RAM.

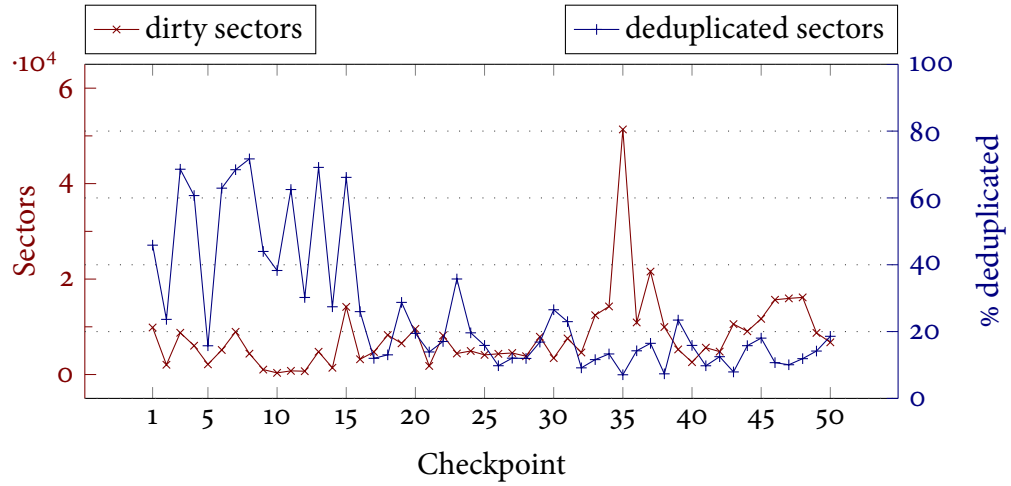


Figure 6.2: Dirty count of sectors and percentage of block device deduplication during a kernel build with a checkpointing interval of 2000 ms.

icantly increased. Figure 6.3 shows the percentage of intra- and inter-checkpoint deduplication of 50 checkpoints during a kernel build with 2 GiB RAM. The amount of inter-checkpoint deduplication is at 10–46 % (avg=14.55, median=13.00, sd=5.94), while the intra-checkpoint deduplication is only at 1–14 % (avg=1.71, median=1.00, sd=2.28).

To further evaluate the deduplication, the distance of inter-checkpoint duplicates is measured, i.e., the difference of the current checkpoint index c_{curr} and the index of the corresponding cache hit c_{hit}. The measurement can be accomplished by

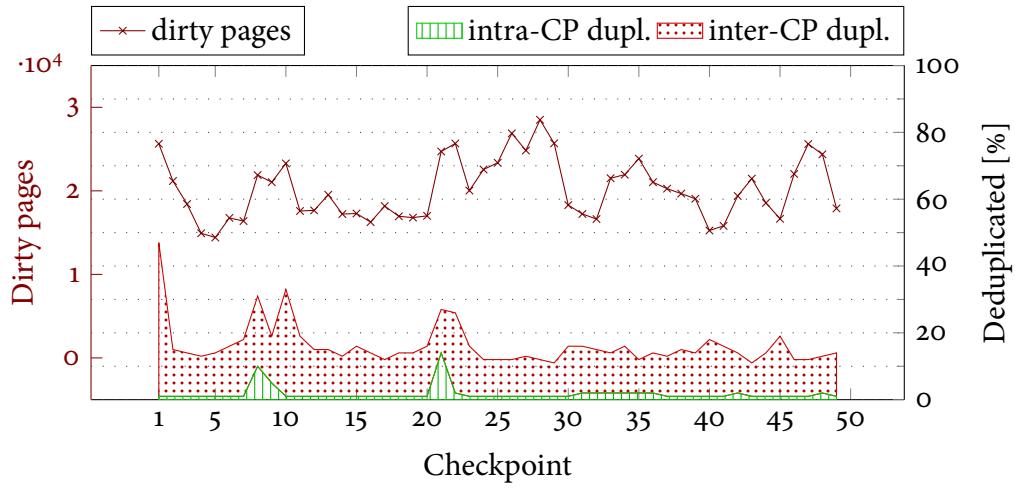


Figure 6.3: Dirty pages and percentage of intra- and inter-checkpoint deduplication of memory pages during a kernel build with 2 GB RAM.

using a $m \times n$ matrix for cache size m and amount of checkpoints n with entries $m_{ij} = c_{curr} - c_{hit}$, which gives detailed information about the duplicate distance. Figure 6.4 shows the amount of cache hits and their distance for nine checkpoints with indices 5, 10, …, 45 during a kernel build. Most of the cache hits are within a 5 checkpoint distance, but hits at the maximum distance occur, i.e., the oldest checkpoint available in the cache, which is indicated by the dashed line. The area below the dashed line contains checkpoints that are not yet in the cache, therefore it is only relevant for the starting phase where the number of cached checkpoints is lower than the maximum number.

This reflects the functioning and efficiency of the hash-based cache for deduplication. However, it does not accurately reflect the actual distance of inter-checkpoint duplicates. This is because the cache updates the index c_{hit} on a cache hit, because the element is used in the checkpoint at the current index: $c_{hit} = c_{curr}$. This update causes multiple accesses to cached elements to be only counted as an inter-checkpoint duplicate once per checkpoint (and counted as an intra-duplicate on further accesses) and frequently accessed elements to constantly move backwards in the distance (i.e., they move towards the xz plane in the plot shown in Figure 6.4). To find out the actual distance of inter-checkpoint duplicates, i.e., the time the duplicate data has first been inserted into the cache, updating of indices can be disabled so that the matrix mentioned above provides the actual values. Figure 6.5 shows the actual distance of the duplicates and therefore the viability of inter-checkpoint deduplication during a kernel build. The deduplication rate for the maximum distance is at 700–1000 pages per checkpoint, which is the majority of the duplicate pages.

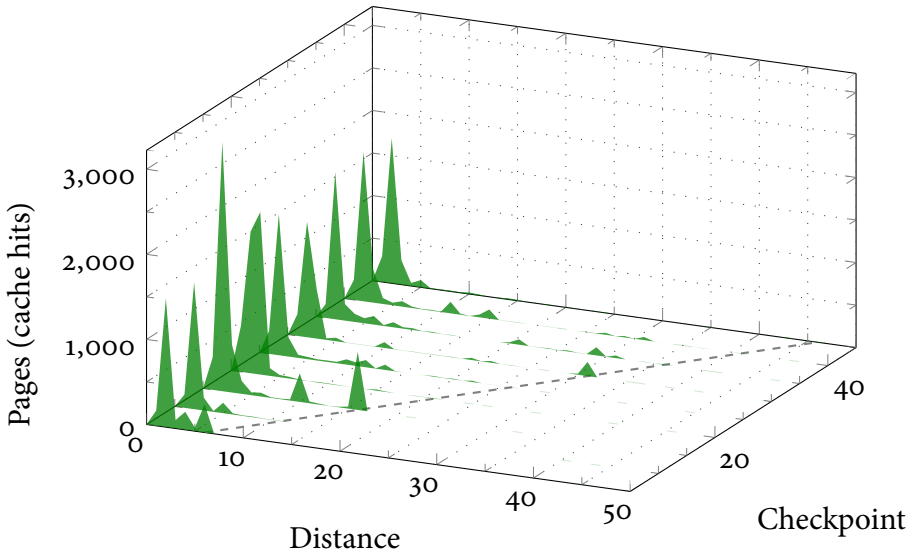


Figure 6.4: Memory page cache hits and their distance for nine checkpoints (indices 5, 10, ..., 45) during a kernel build. The green areas show the amount of cache hits at a certain distance for each checkpoint.

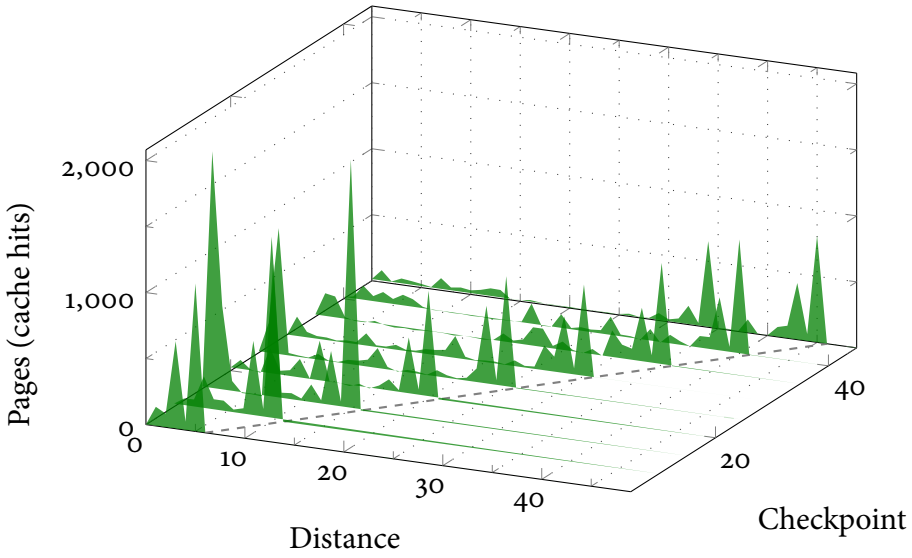


Figure 6.5: Memory page cache hits and their distance for nine checkpoints (indices 5, 10, ..., 45) during a kernel build, without index updating. The green areas show the amount of cache hits at a certain distance for each checkpoint.

The overall amount of deduplicated pages and sectors is important for reducing the data amount. This amount heavily depends on the workload: If checkpointed

memory or block device data is reused later, it can be fully deduplicated. If, however, only novel data is written, the deduplication rate is accordingly low. Table 6.1 shows the average percentage of deduplicated pages and sectors in 50 checkpoints during various benchmarks. Depending on the workload, the memory deduplication is at 8.26–42.16 % and block device deduplication is at 34.79–80.64 %. The two SPEC benchmarks were chosen because of the percentage of memory write operations, which amounts to 13 % for the bzip2 benchmark and to 25 % for the omnetpp benchmark [15]. Additionally, the bzip2 benchmark compresses and decompresses six different files using three different compression levels for each [14]. The compression and decompression happens in memory, so recurring data leads to a large amount of inter-checkpoint duplicates. In the omnetpp benchmark, network traffic is generated [14], which leads to fewer inter-checkpoint duplicates.

Benchmark	Memory		Block dev.	
	Avg. count	Avg. %	Avg. count	Avg. %
Kernel build	3484	16.90 %	1728	34.79 %
SPEC bzip2	6766	42.16 %	162	67.07 %
SPEC omnetpp	2565	8.26 %	155	80.64 %

Table 6.1: Average deduplication of memory pages and block device sectors, and average percentage of deduplicated data for various benchmarks. 50 checkpoints were taken in an interval of 2000 ms.

6.4.2 Data Amount

Another important aspect is the amount of data that has to be transferred and stored in the database. The data amount heavily depends on the number of checkpoints to be stored and the workload of the VM during the checkpoints. Since the implementation in this thesis uses QEMU's built-in checkpoint mechanism to checkpoint device states, which saves all state variables without deduplication, this part of the checkpoint is almost constant at on average 80 KiB per checkpoint, which is negligible in this evaluation. Memory and block device checkpoints, however, vary in size, depending on the workload. Another factor is the deduplication rate, which can greatly decrease the amount of data, and also depends on the workload.

To evaluate the data amount used by the checkpointing mechanism, multiple runs per configuration are executed using equally configured VMs with 2 GiB RAM, while taking 50 checkpoints at an interval of 2000 ms. Then, the disk space used by the MongoDB tables is measured and the deduplication amount is calculated. The average consumed disk space for different workloads and the estimated amount of

deduplicated data is shown in Table 6.2. The estimated reduction is calculated using the deduplication percentage from Table 6.1. Depending on the workload, the disk space consumed by the checkpoints can be greatly reduced.

Benchmark	Disk space used		Reduced by dedupl.	
	Memory	Block dev.	Memory	Block dev.
Kernel build	3.91 GiB	168.67 MiB	795.25 MiB	89.99 MiB
SPEC bzip2	1.59 GiB	122.61 MiB	1158.05 MiB	249.72 MiB
SPEC omnetpp	5.92 GiB	6.30 MiB	532.90 MiB	26.26 MiB

Table 6.2: Average total disk space consumed by the database after 50 checkpoints and amount of disk space reduced through deduplication for various benchmarks.

6.4.3 Downtime

As explained in Section 3.1, it is necessary to suspend the VM for a moment in order to save a consistent system state, leading to a downtime. For SimuBoost, the downtime needs to be as small as possible. Therefore, this section performs an evaluation of the downtime, measuring the detailed execution time inside of QEMU.

The downtime heavily depends – besides on the workload – on the database configuration and the hardware of the database server. As explained in Section 2.5, MongoDB can be used with different write concerns. Another important factor is the disk that MongoDB uses: SSDs offer much higher write performance than HDDs. To evaluate the performance of the checkpointing meachnism for different hardware and write concern configurations, multiple runs were evaluated during a kernel build. Figure 6.6 shows the downtime over 50 checkpoints for a MongoDB table on a HDD using write concerns $w = 1$ and $w = 0$, and a table on a SSD with write concern $w = 0$. The HDD checkpoints with $w = 1$ lead to the highest downtimes of 1200–5700 ms (avg=2588, median=2364, sd=961) with wide fluctuations. Using a write concern of $w = 0$ results in generally lower downtimes of 950–6300 ms (avg=2105, median=1886, sd=1122), still with wide fluctuations. The lowest and most stable downtimes are achieved by using a SSD with the $w = 0$ write concern, resulting in downtimes of 700–2600 ms (avg=1151, median=1001, sd=376).

To further examine the downtime, more detailed measurements are performed for various parts of the checkpointing mechanism for each checkpoint:

- *Total time*: The time needed to save a complete checkpoint, including all data obtainment, hashing, and sending to the database. This is essentially the downtime of the VM.

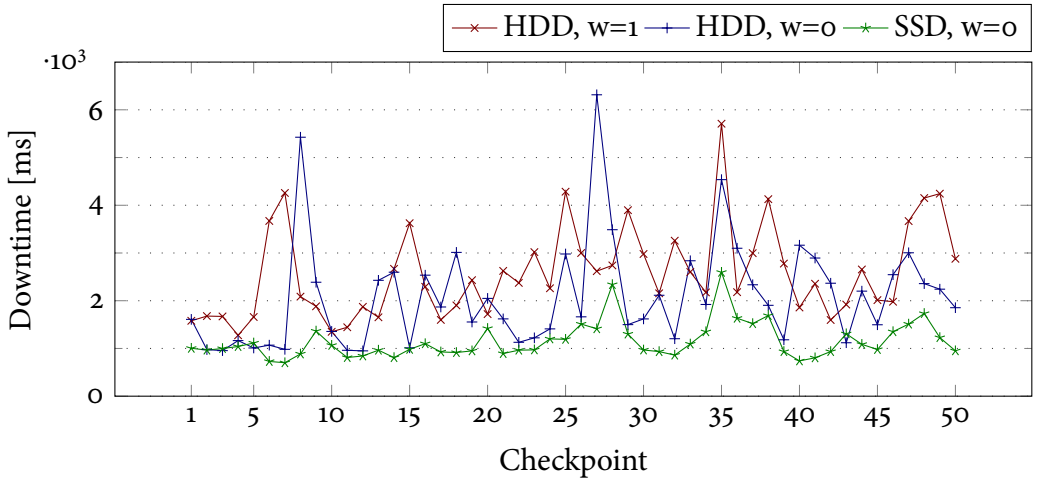


Figure 6.6: Downtime of the checkpointing mechanism with different disks used for the database and write concerns. The checkpoints were taken during a kernel build using a checkpointing interval of 2000 ms.

- *Hash time*: The sum of the time needed for calculation of the hash values for pages and sectors. This depends on the performance of the hash function and the number of dirty pages and sectors.

- *Cache time*: The additional time needed for searching and inserting in the cache tree. The cache time depends on the cache size itself and on the amount of insert operations, which are executed after a cache miss and take additional time.

- *Send time*: Waiting for the database query using a blocking request causes a waiting time for the response, or at least for a successful sending of the data. The send time heavily depends on the database, which can cause delays regardless of the checkpointing mechanism.

With these measurements, the actual downtime (i.e., the total time) can be broken down into the time needed for certain parts of the checkpointing. Thus, time-intensive parts can be identified. Figure 6.7 shows the percentage of the different checkpointing components, broken down into the single components: Hash time, cache lookup time, and database send time. The cache produces a hardly detectable increase in downtime. Hash function invocation adds under 20 % per checkpoint, although this time heavily depends on the hash function. The majority of the downtime can be attributed to sending data to MongoDB, which takes about 40–60 % of the time per checkpoint. The remaining part, which is not displayed in

the figure, can be attributed to the data collection, copying of data to buffers, and other components of the checkpointing mechanism.

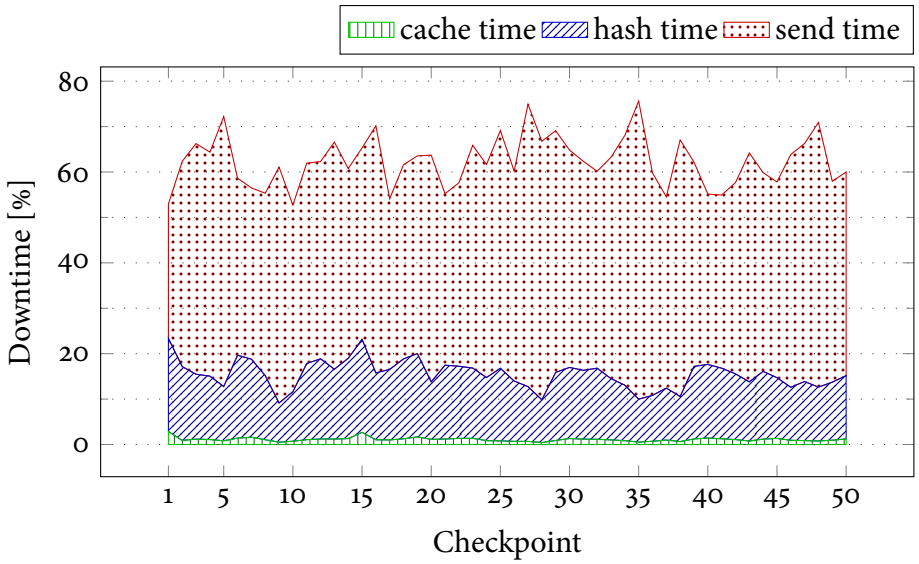


Figure 6.7: Time for sending, hashing and caching of memory and block device data as percentage of the total downtime.

The large part of downtime caused by sending the data can be explained by looking at the `mongostat` program while checkpoints are saved: The database spends a long time in a locked state and with a write queue, which is caused by write operations and causes other write operations to accumulate. This also explains the lower downtime when using a SSD, because data can be written at a higher rate and less write operations accumulate. The problem of accumulating write operations in MongoDB can be solved by better distributing the insert operations for data: This can be accomplished by either using CoW to insert data while the system is running and therefore distributing the insert operations time-wise, or by using a *sharding* deployment of MongoDB. By sharding, the insert operations can be divided on multiple servers, resulting in horizontal scaling and reducing the load on a single server.

6.5 Conclusion

The evaluation performed in this chapter showed that the checkpointing mechanim is working correctly within the boundaries of hash collision probabilites. Further, the performance aspects were examined, which showed that the hash-based memory and block device deduplication, especially the inter-checkpoint deduplication,

greatly decreases the data amount of checkpoints. The evaluation of the downtime during checkpoints showed that most of the downtime stems from sending data to the database and hashing the data. This can be improved by using CoW or sharding of MongoDB servers. The average downtime is still under 2 s and therefore meets the requirements for SimuBoost.

7 Conclusion

The execution time of modern functional full system simulation is much higher than virtualization due to more complex system modeling and extensive analyzation tools. Therefore, SimuBoost aims at decreasing the execution time of simulation through virtualization and parallelized simulation. In frequent intervals, checkpoints of the virtualized system are taken, which provide the simulated systems with a complete system state. After that, the simulated systems execute the workload for one checkpointing interval.

In this thesis, a checkpointing mechanism capable of saving and loading deduplicated incremental checkpoints was developed. The requirements also included easily distributable checkpoints, i.e., the difficulties with loading incremental checkpoints needed to be avoided: Instead of searching previous checkpoints for data until the checkpoint is loaded completely, the checkpoints in this approach were separated into data and headers, which additionally benefitted the deduplication. After storing the checkpoint data and headers in a database, systems can obtain the header and after that the checkpoint data regardless of previous checkpoints.

The applicability of this checkpointing mechanism has been shown by evaluating its performance:

By using a hash-based cache for deduplication, previously sent data was saved from being sent again, which – depending on the workload – drastically reduced the data amount that needed to be sent and later stored in the database. The downtime of the VMs was evaluated, showing that the downtime heavily depends on fast hash algorithms and strategies speed up database inserts.

7.1 Future Work

The approach presented in this thesis included a checkpointing mechanism capable of saving and loading checkpoints suitable for SimuBoost's requirements. The loading mechanism, however, was only performed in QEMU and has to be ported to a full system simulator in future research. Also, as mentioned in Section 3.1, the downtime of the VM caused by the checkpointing should be as short as possible. Sec-

tion 6.4.3 showed that the majority of the downtime was consumed by the database insert operations. To address this issue, *sharding* can be used with MongoDB, i.e., multiple MongoDB servers share a collection and queries are divided between these MongoDB instances. To further reduce the downtime, a CoW approach can be used to collect data while the system is running.

Bibliography

[1] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using ksm. In *Proceedings of the linux symposium*, pages 19–28, 2009.

[2] Eduardo Argollo, Ayose Falcón, Paolo Faraboschi, Matteo Monchiero, and Daniel Ortega. Cotson: Infrastructure for full system simulation.

[3] Sean Barker, Timothy Wood, Prashant Shenoy, and Ramesh Sitaraman. An empirical study of memory sharing in virtual machines. In *Usenix ATC*, 2012.

[4] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[5] Mateusz Berezecki, Eitan Frachtenberg, Mike Paleczny, and Kenneth Steele. Many-core key-value store. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1–8. IEEE, 2011.

[6] Cityhash. Cityhash. https://code.google.com/p/cityhash.

[7] 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.

[8] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, volume 7, pages 205–220, 2007.

[9] Stefan Edlich, editor. *NoSQL : Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken*. Hanser, München, 2010.

[10] Jakob Engblom. Full-system simulation. *European Summer School on e mbedded Systems.(ESSES'03)*, 2003.

[11] 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.

[12] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C Snoeren, George Varghese, Geoffrey M Voelker, and Amin Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. *Communications of the ACM*, 53(10):85–93, 2010.

[13] Irfan Habib. Virtualization with kvm. *Linux Journal*, 2008(166):8, 2008.

[14] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[15] Aamer Jaleel. Memory characterization of workloads using instrumentation-driven simulation. 2010.

[16] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC Press, 2008.

[17] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[18] KVM. Memory - kvm. http://www.linux-kvm.org/page/Memory.

[19] GNU libc manual. Tree search function. http://www.gnu.org/software/libc/manual/html_node/Tree-Search-Function.html.

[20] 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.

[21] MongoDB Manual. Write concern. http://docs.mongodb.org/manual/core/write-concern/.

[22] Mark McLoughlin. The qcow2 image format, 2008. https://people.gnome.org/~markmc/qcow-image-format.html.

[23] Konrad Miller, Fabian Franz, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. Xlh: More effective memory deduplication scanners through cross-layer hints. In *Proceedings of the 2013 USENIX Annual Technical Conference*. USENIX Association, 2013.

[24] 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.

[25] Redis. Documentation. http://redis.io/documentation.

[26] Marc Rittinghaus, Konrad Miller, Marius Hillenbrand, and Frank Bellosa. Simuboost: Scalable parallelization of functional system simulation. *Simulation*, 1:1, 2013.

[27] Mahmoud Sayrafiezadeh. The birthday problem revisited. *Mathematics Magazine*, 67(3):220–223, 1994.

[28] Michael H. Sun and Douglas M. Blough. Fast, lightweight virtual machine checkpointing. CERCS Technical Reports GIT-CERCS-10-05, Georgia Institute of Technology, 2010.