Karlsruher Institut für Technologie

Checkpoint Distribution for SimuBoost

Master Thesis
of

Andreas Pusch

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

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

Created during: May 10, 2017 – October 26, 2017

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

Karlsruhe, October 26, 2017

Abstract

Full system simulation provides means for analyzing systems by allowing reproduction of physical hardware events as well as providing access to analytical data and analysis tools. Benefits of these simulations are the support for malware analysis, memory studies, and high availability testing as well as operating system development and debugging. One major downside though is the slowdown of the simulation of a factor of 31 up to 810 in comparison to hardware-assisted virtualization. This slowdown limits the applicability of full system simulation to short running workloads. SimuBoost aims to solve this issue by running the workload in a hardware-assisted virtual machine and splitting it into multiple simulation intervals. At each interval, SimuBoost creates an incremental checkpoint which is used to bootstrap the simulation of the interval. These checkpoints are distributed by SimuBoost across a simulation cluster which runs the simulation intervals in parallel.

The current distribution mechanism leads to high network load which can result in a network bottleneck and limits the scalability of SimuBoost. This distribution leads to longer checkpoint loading times and a slowdown of the interval simulations as well as the overall achievable speedup. We are evaluating distributed storage and multicast as possible solutions to achieve our goals of reduced checkpoint loading times, increased scalability and a reduced network load to make SimuBoost a viable option for Gigabit Ethernet networks using commodity hardware and therefore increase its applicability.

We have implemented a multicast solution that has satisfied all of our goals. Our solution does not exhaust a Gigabit Ethernet network using the build-linux-kernel and SPECjbb benchmarks and results in stable checkpoint loading times. Our solution scales well and achieves a speedup when doubling the number of parallel simulations from 12 to 24.

Deutsche Zusammenfassung

Durch Full-System Simulation können Systeme analysiert werden um beispielsweise Malware Analysen, Hochverfügbarkeitstests und Speicherstudien durchzuführen. Des Weiteren kann Full-System Simulation auch zur Entwicklung und Fehlerbeseitigung von Betriebssystemen verwendet werden. Die Benutzbarkeit wird jedoch durch eine Verlangsamung der Ausführung um Faktoren von 31 bis zu 810, im Vergleich zu hardwarebeschleunigter Virtualisierung, beeinträchtigt. SimuBoost ist eine Simulationslösung die versucht Full-System Simulation durch das Aufteilen der Simulation in Simulationsintervalle, sowie der parallelen Ausführung dieser Intervalle, zu beschleunigen. Hierzu müssen inkrementelle Checkpoints der zu simulierenden virtuellen Maschine erstellt und in einem Simulationscluster verteilt werden. Die erreichbare Beschleunigung ist dabei maßgeblich von der Anzahl der parallelen Simulationen und der Intervallänge abhängig.

Das aktuelle Verteilungsverfahren führt jedoch zu einer hohen Netzauslastung und einem Flaschenhals bei der Übertragung der Checkpoint Daten vom Virtualisierungshost zu den Simulationsknoten. Eine längere Übertragungsdauer der Checkpoints durch hohe Netzauslastung führt zu längeren Checkpoint Ladezeiten, verringert die erreichbare Beschleunigung der Simulation sowie die Skalierbarkeit von SimuBoost. Wir evaluieren Distributed Storage und Multicast als mögliche Lösungen für die Verteilung der Checkpoints. Unsere Ziele sind verringerte Checkpoint Ladezeiten, besser Skalierbarkeit und eine reduzierte Netzwerkauslastung, sodass eine Anwendung von SimuBoost in Gigabit Ethernet Netzwerken ohne Spezialhardware möglich ist.

Wir haven ein neues Verteilungsverfahren auf Basis von Multicast implementiert, dass unsere gestellten Ziele erreicht. Unsere Lösung lastet ein Gigabit Ethernet Netzwerk bei Verwendung der Build-Linux-Kernel und SPECjbb Benchmarks nicht aus und erreicht stabile Checkpoint Ladezeiten. Unsere Lösung skaliert gut und erreicht eine Beschleunigung bei Verdopplung der parallelen Simulationen von 12 auf 24.

Contents

Chapter 1

Introduction

Full system simulation is an important tool in areas such as operating systems debugging and malware analysis. Full system simulators allow reproduction of physical hardware in combination with additional analysis tools to enable system analysis on the level of individual instructions. However, common full system simulators suffer from high slowdowns. Rittinghaus et al. [1] describe an average slowdown of factor 31 for functional simulation with QEMU [2] as well as a factor of 810 with Simics [3]. These slowdowns restrict the applicability of full system simulators for dynamic analysis to short-running workloads. Furthermore, the slowdowns reduce the interactivity of the simulation due to long delays for responses to user input.

SimuBoost [1] is a project that utilizes parallelization to achieve fast full system simulation. For this goal, SimuBoost runs the workload of interest in a hardware-assisted virtual machine (VM) and periodically creates incremental checkpoints of the VM. The SimuBoost server distributes these checkpoints across a cluster of simulation nodes to bootstrap parallel simulations of the intervals. The execution speed difference between hardware-assisted virtualization and functional simulation drives a parallelization of the simulation. The speedup that can be achieved by SimuBoost is dependent on the right checkpoint interval length and the number of parallel simulations. The checkpoint distribution is an important factor as it delays the actual start of the parallel simulations. This delay essentially means it takes longer for the simulation of the individual checkpoint to finish in relation to the moment the producer has created the checkpoint. As a result, the complete simulation takes longer, and the achievable speedup is reduced. Therefore, the checkpoint distribution needs to be reliable and as fast as possible to reduce any overhead to a minimum. In a direct distribution approach by Eicher [4], the simulation nodes pull all necessary data for the current simulation interval from SimuBoost. Eicher has shown that using a small number of

consumers and a Gigabit Ethernet network connection saturates the network and leads to a network bottleneck.

The goal of this thesis is to provide a distribution solution for the checkpoints that scales well and is as fast as possible to enable SimuBoost to achieve an optimal speedup of the simulation. This distribution solution should be able to work on commodity systems using a Gigabit network connection with a theoretical maximum transmission rate of 125 MB/s. This restriction makes using SimuBoost easier because no expensive hardware is required.

To achieve our goals, we utilize SimuBoost's checkpoint compression to reduce the amount of data that needs to be transmitted. Furthermore, we utilize a pushing mechanism to distribute data across the cluster as a means to distribute the network load more even across the entire cluster. This approach aims to reduce the bottleneck on the connection between the virtualization host and the simulation nodes.

The checkpoint distribution mechanism we have introduced achieves a reduction of checkpoint loading times using a build-linux-kernel scenario with six nodes and four parallel jobs per node from an average of about 382 seconds to about 12 seconds compared to the direct distribution approach. As a result, our solution reduces the runtime of the complete simulation from about 156 minutes to about 69 minutes.

In Chapter 2 we talk about backgrounds such as virtualization and simulation, SimuBoost, distributed storage, and multicast. Next in Chapter 3 we analyze the current status of the checkpoint distribution and propose possible solutions to resolve the distribution issues. In Chapter 4 we elaborate the designs of our proposed solutions and afterwards in Chapter 5 we show implementation details of the multicast approach. Finally in Chapter 6 we evaluate our solutions and give a conclusion and outlook in Chapter 7.

Chapter 2

Background

This chapter provides the necessary background regarding virtual machines, full system simulation and checkpointing. Furthermore, we will explain the basics of SimuBoost as well as networking basics for network attached storage, distributed storage, and multicast. This information is essential to understand our challenges and the approaches we are taking.

2.1 Virtual Machines

Virtual machines (VMs) [5] provide new capabilities such as software isolation, and compatibility layers for computer systems. To explain virtual machines, we will first look at the different layers of abstraction in a regular desktop computer.

The instruction set architecture (ISA) divides the hardware from the software. It hides the hardware details from components such as the operating system, drivers, the memory manager, and the scheduler. The instruction set architecture is dependent on the hardware, examples are the x86 instruction set, which is the most common for personal computers, and the ARM Cortex instruction set architecture, which is mainly used for embedded devices such as smartphones.

The application binary interface (ABI) provides abstractions from the operating system as well as user instructions. The GCC compiler [6], for example, uses the Itanium C++ ABI [7, 8] to act as an interface between user C++ code and the operating system and operating system libraries. Responsibilities of the C++ ABI include memory layouts for C++ data objects as well as function calling interfaces, exception handling interfaces, global naming, and various object code conventions.

Finally, there is the application programming interface (API) which provides an interface for applications and application libraries. The Windows API [9], for example, provides standardized access for applications and libraries to basic

resources like file systems, devices, and processes as well as other services like user interfaces, network services, and the Windows registry. Figure 2.1 illustrates these three layers of abstraction.

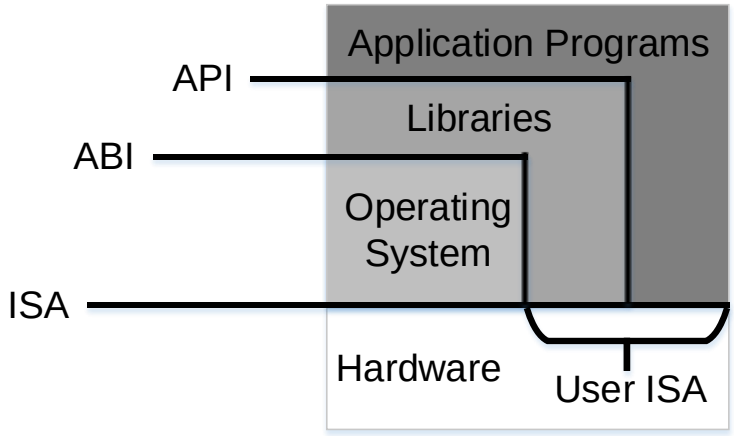

Figure 2.1: An overview of machine abstraction layers. [5]

A virtual machine works on top of these abstractions and maps virtual resources or state to the real resources of the host system and uses the instructions and system calls of the host system to carry out the virtual instructions. There are two main types of virtual machines based on the abstractions they work on.

Process virtual machines translate application binaries for the same or different API, ABI or user ISA than the host platform. Examples for process virtual machines are Wine [10, 11] and Cygwin [12, 13]. These virtual machines provide an operating system compatibility layer to execute binaries which require a different operating system API than the one of the host operating system. Another type of process virtual machine is a high-level language runtime such as the Java VM [14, 15] and the Common Language Infrastructure (CLI) [16, 17]. These solutions provide platform independence for applications by abstracting the operating system in the form of libraries.

System virtual machines, on the other hand, reside on the ISA level. A Virtual Machine Manager (VMM) [18] is responsible for virtual instruction handling, resource allocation management, and resource access. Depending on the type of system virtualization, the VMM runs either in privileged mode, as a user application, or in dual mode where some parts are privileged, and some are not. Examples for VMM which run in privileged mode (native system VMs) are Xen [19] and Hyper-V [20, 21]. Oracle VirtualBox [22] and QEMU [2, 23] are examples

for VMs with non-privileged VMM in which the VMM runs as a user application (hosted system VMs). Additionally, hybrid solutions such as QEMU with KVM [23, 24] operate in both modes (dual-mode/hybrid system VMs).

System virtual machines have several benefits such as compatibility for foreign or deprecated platforms, security by isolation of software between virtualized systems (guests), and easy replication, and migration of virtual machines between hosts. The most important benefit for this thesis though is system analysis. A user can inspect applications and operating system operations at the ISA level and use this ability for debugging, software analysis and research.

2.1.1 Checkpointing

A key part of the benefits of system virtual machines is the ability to create checkpoints [5]. A system virtual machine checkpoint is a snapshot of the complete state of the machine. It consists of the contents of the RAM, the disk, the CPU state, and registers, as well as the device data provided by the virtual machine manager. Virtual machine checkpointing provides benefits in several areas such as virtual machine migration [25, 26], fault tolerance [27, 28] and debugging [29, 30]. An important metric of checkpoints is the downtime of the virtual machine, meaning how long the machine has to be stopped to create the checkpoint. Another important metric is the size of the checkpoint. Especially in cases like virtual machine migration, where checkpoints have to be transmitted via a network, checkpoints have to be as small as possible to reduce transmission times and network load.

Incremental Checkpoints

Incremental checkpoints [31–33, 36] are an approach to reduce the size of checkpoints by only storing differences since the previous checkpoint. The major challenge of incremental checkpointing is the detection of content modifications. Some approaches such as proposed by Mehnert-Spahn et al. [33] utilize existing page table entry bits, namely the dirty bit or the write bit, to detect page modifications. Argawal et al. [32] propose an approach that uses a secure hash function to uniquely identify changed blocks in memory.

Copy-On-Write

Copy-on-write (CoW) [34] is a technology which can be used to detect modifications in memory pages and to decrease the checkpointing downtime. Furthermore, operating systems such as Mach [35] use this technology as a kernel performance optimization. CoW memory is shared read-only in a protected mode. As soon

as a write operation is performed on the protected memory, a copy of the affected memory page is created. Sun et al. [31] describe the implementation of a CoW-based checkpointing solution for the Xen VMM. When using CoW for checkpointing, the VM is suspended to put its memory in a protected read-only state and is started again afterwards. Modifications to parts of the memory create copies which can be used while the VM is running. This approach provides a consistent view of the VM without having to copy any state while the VM is suspended and therefore without prolonging the downtime of the VM.

2.1.2 Emulation

Emulation [5] allows a virtual machine to implement interface and functionality of a different system. The VM translates instructions meant for one instruction set architecture to another emulating the desired hardware. Privileged instruction handling is done using a technique called *Trap'n'Emulate* [18]. When a guest system attempts to execute a privileged instruction, the virtual CPU traps into the VMM, which in turn emulates the operation on the guest state. Emulation is the key technology for most virtual machines in both process and system virtualization.

Interpretation and Binary Translation

Interpretation and binary translation [5] are important methods used in system emulation. In interpretation, the emulator fetches source instructions, analyses them and performs the required operation on the host system. Interpretation is simpler to implement than binary translation but has high execution costs. In binary translation, the emulator fetches a block of source instructions, generates a block of associated target instructions and stores these instructions for repeated use. This approach is more complex and has higher initial costs due to the pre-translation overhead for entire code blocks. However, binary translation also reduces the execution costs. In practice, it is best to use a staged operation between both solutions to achieve optimal results.

2.1.3 Hardware-Assisted Virtualization

As we have seen in the previous subsection, emulation requires translations which are complex and lead to a slowdown of the execution speed of the virtual machine compared to a native system. Hardware-assisted virtualization [37, 38] has been developed to improve the speed of virtual machines. This method is faster than emulation as the required instructions are directly executed on the host system's

hardware. For this approach to work, the guest system must have the same instruction set architecture as the host system, and the host must allow operations such as CPU context switches between virtual machines. Hardware-assisted virtualization is less flexible than emulation due to the requirement of the guest to have the same instruction set architecture as the host. We look at x86 architecture extensions for hardware-assisted virtualization as an example. Technologies such as AMD's SVM [37, 38] and Intel's VT [38, 39] export new primitives and provide the capability to store the state of a guest virtual CPU in an in-memory data structure. Furthermore, the hardware supports an additional execution mode called *guest mode*, which enables direct execution of guest code, including privileged code. A new instruction, *vmrun*, allows the transfer from guest to host mode and vice versa.

2.2 Full System Simulation

Full system simulation [3, 40] aims to provide means for analyzing a system by allowing reproduction of physical hardware events as well as providing additional analysis tools. This functionality is achieved by using system virtual machines and providing easy access to analytical data, e.g., by registering hooks for memory accesses. Full system simulators use mechanisms such as deterministic replay [41, 42] to record and replay the execution of a guest system. The entire VM execution, including the operating system and applications, is recorded and replayed. The recording includes non-deterministic events for example in the form of data and timing of device inputs such as virtual disks and virtual network interface cards.

Examples for full system simulators are Simics [3], a commercial full system simulator that is able to simulate memory access delays and caching effects, and MARSSx86 [43], an open-source full system simulator for x86 systems based on QEMU.

One of the benefits of full system simulation is the support for analysis, development, and debugging of operating systems. Additionally, applications such as malware analysis, memory studies, and high-availability testing benefit as well from full system simulation.

The major downside of full system simulation is the slowdown of the execution due to overhead resulting from analysis requirements. The main reason for this slowdown is the emulation of the workload. Emulation is necessary because hardware-assisted virtualization does not grant the required level of control for system analysis. Additionally, memory hooks and analysis tools lead to a further slowdown of the simulation. Rittinghaus et al. [1] describe an average slowdown of factor 31 for functional simulation with QEMU as well as a factor of 810 with Simics.

There are two common kinds of simulators, functional and micro-architectural. Micro-architectural simulators such as gem5 [44] and MARSSx86 [43] simulate the internal implementation details of hardware components. These simulators can be used to model the design and behavior of components such as a CPU's ALU, cache memory, and control unit. Functional simulators such as Simics [3] and Parallel Embra [40] on the other hand do not focus on hardware implementation details. Instead, they aim to provide the ability to perform analysis on the level of individual instructions.

2.3 SimuBoost

SimuBoost [1] is a project that utilizes parallelization to achieve fast, functional full system simulation. Traditional simulators like QEMU [2, 23] and Simics [3] suffer from slowdowns (see Section 2.2), which limits their applicability to short running workloads. SimuBoost aims to solve this issue by splitting the simulation along the time axis into a number of intervals as illustrated by Figure 2.2.

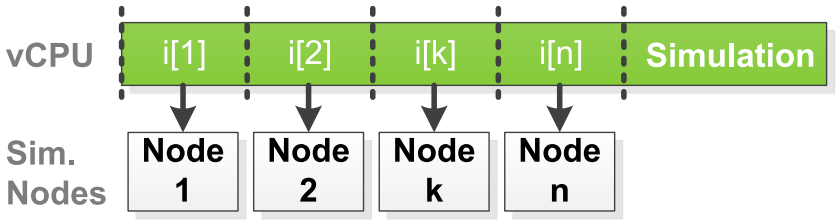

Figure 2.2: The simulation is split into n intervals depending on the required degree of parallelism. [1]

At each interval, SimuBoost creates an incremental checkpoint which it distributes across a simulation cluster. These checkpoints serve as starting points for parallel simulations. The difference in execution speed between the hardware-assisted virtualization of the workload on the virtualization node and the emulation on the simulation nodes drives a parallelization of the simulation. (see Figure 2.3).

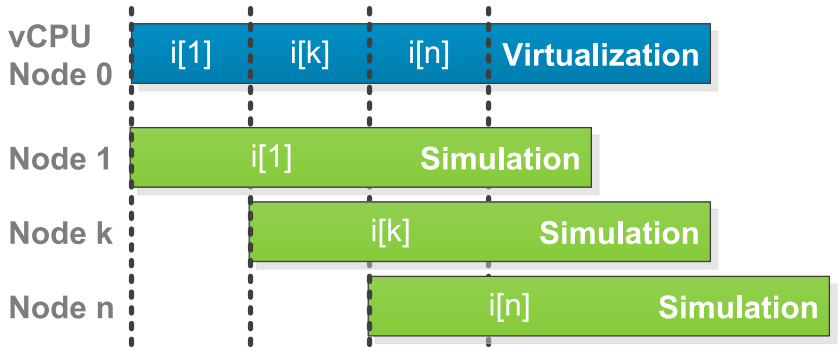

Figure 2.3: Parallel simulation using checkpoints at interval boundaries as starting points. [1]

The overhead of the checkpoint distribution directly impacts the time until a checkpoint simulation finishes and the next simulation can be started. The longer the checkpoint loading process takes, the longer the simulation will take which reduces the achievable speedup of SimuBoost.

QEMU and KVM SimuBoost uses a modified QEMU [2, 23], a generic and open source machine emulator. This version of QEMU contains tracing hooks for the tracing framework Simutrace [45] and support for SimuBoot's checkpointing functionality. Additionally, SimuBoost uses Kernel-based Virtual Machine (KVM) [23, 24], a component of QEMU and a set of Linux kernel modules for hardware-assisted virtualization. SimuBoost requires a modified Linux kernel with KVM for the checkpointing.

2.3.1 Speedup and Scalability

Rittinghaus et al. [1] have provided a formula to calculate the possible speedup of the simulation. Let n := the number of intervals, t_c := the constant VM downtime for a checkpoint and t_i := a simulation's initialization time[1]. Further, let s_{log} := the slowdown in the virtualization stage incurred by the logging of non-deterministic events, T_{vm} := the workload's run-time with conventional virtualization, T_{sim} := the workload's run-time with conventional functional simulation

[1]At this time checkpoints can be migrated to a different node and loaded to initialize the simulated machine's state on the basis of the checkpointed information.

and s_{sim} := the effective slowdown between virtualization and functional simulation. The speedup S(L) of the SimuBoost approach compared to serial functional simulation is described as:

$$S(L) = \frac{s_{log} T_{vm} * s_{sim} L}{s_{log}^2 T_{vm}(t_c+L)+s_{log} t_i L+s_{sim} L^2}$$

Figure 2.4 illustrates the importance of checkpoint creation and distribution on the parallelization of the simulation. As new intervals are submitted, additional nodes are allocated until the first simulation finishes. After that, simulations can be scheduled onto previously allocated nodes because of the assumption that simulations complete with approximately the same rate than new intervals are submitted.

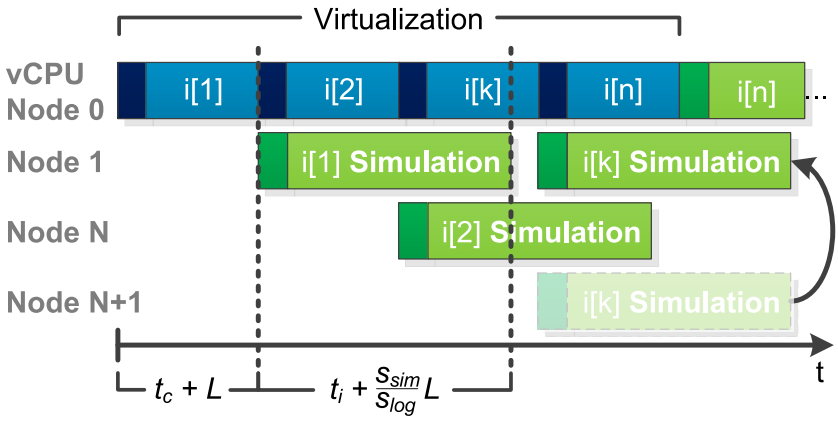

Figure 2.4: Checkpoint creation and distribution delay the checkpoint loading. Subsequent intervals can be scheduled onto previously allocated nodes. [1]

2.3.2 SimuBoost Checkpointing

In this section, we will explain the SimuBoost checkpointing process [4, 46]. A checkpoint captures the entire state of the related system including the contents of the system memory, the disk, the internal state and registers of the CPU, and the state of the attached devices. SimuBoost creates these checkpoints incrementally, which avoids having to copy the entire state at every checkpoint. Instead, SimuBoost only stores differences to the last checkpoint which reduces both, the duration of the checkpointing process, as well as the required storage space. Furthermore, SimuBoost utilizes copy-on-write [31, 34] to reduce the checkpointing downtime and data deduplication [47], and delta compression [26] to reduce the amount of data that has to be stored on disk. In data deduplication, two identical memory pages are combined. Instead of having two copies of a page, both

pages point to the same data. SimuBoost utilizes checkpoint deduplication [4, 48] of RAM pages and disk sectors both within a single checkpoint as well as across multiple checkpoints. Delta compression as introduced in REMUS [26] on the other hand does not require identical memory pages to reduce checkpoint data. Instead, it utilizes similar data with a higher granularity than the memory page size to compress the required amount of data. When a checkpoint is created, Remus checks the cache of previously transmitted pages. If the cache contains the same page, only the differences (delta) of the page data will be sent. REMUS uses a hybrid approach in which by default, XOR is used to detect the differences which are then run-length encoded. However, REMUS falls back to gzip [49] compression of the memory page if the XOR compression rate falls below 5:1 or the previous page is not present in the cache. SimuBoost utilizes a similar approach but only uses delta compression once on a memory page and then starts over with a new copy of the page for the next checkpoint. This approach reduces dependencies over several successive checkpoints.

SimuBoost stores references to memory pages and disk sectors together with the user data in checkpoint files. The actual memory page and disk sector data are located in an append-only flat file. SimuBoost uses data compression algorithms such as lz4 [50–52] and a similar custom compression to reduce the size of the checkpoint data. Figure 2.5 illustrates the checkpoint processing.

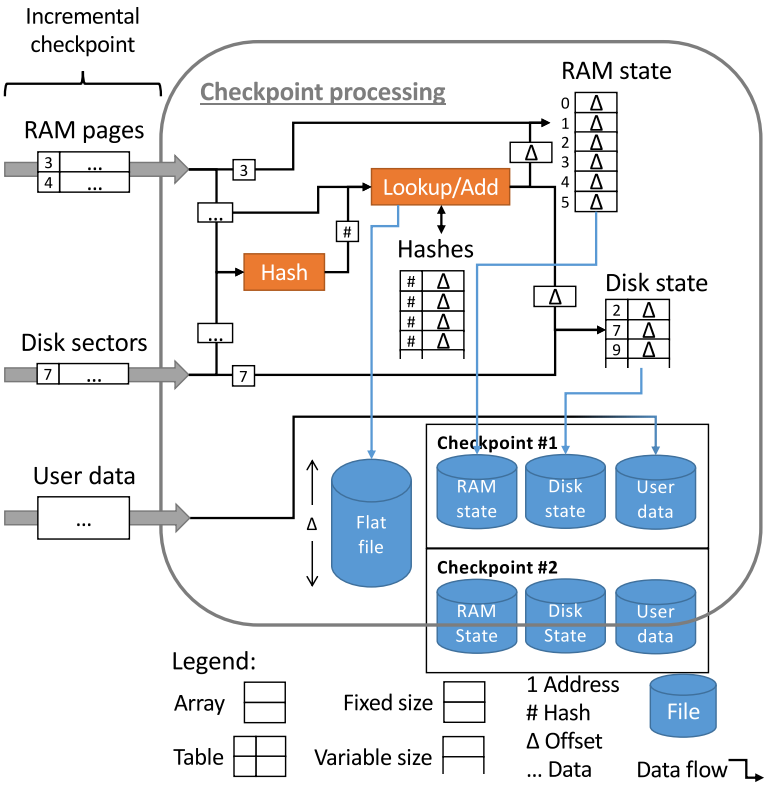

Figure 2.5: An overview of the SimuBoost checkpoint processing [4].

2.4 Ethernet Data Transmission

As explained in Section 2.3, the creation, and distribution of the checkpoints plays a critical role in the achievable simulation speedup of SimuBoost. Eicher [4] has already provided a method for fast and efficient checkpoint creation. Therefore, this thesis focuses on the distribution of the checkpoint data over an Ethernet network. In the following subsections, we will explain the data transmission options we have considered for our approach.

2.4.1 Network Attached Storage

Network attached storage (NAS) [53] is a file level storage system that provides access to storage via a standard Ethernet connection. In this system, a server provides a file service by accepting network messages from clients to access and

modify the data. The server processes client requests such as open, close, read, and write and performs the required action and transmits the requested data back to the client.

Samba

Samba [54] is an Open Source software suite that implements the Server Message Block(SMB) protocol, more specifically the Common Internet File System (CIFS) [55] dialect of the protocol [56]. The CIFS protocol is used for accessing file and print services from server systems over a network. A server running Samba can share local data with multiple nodes via an Ethernet network.

2.4.2 Distributed Storage

In distributed storage [57–59], data is distributed across several machines via a network connection. Instead of a single server which shares the data like a Samba NAS, a distributed storage solution consists of multiple servers which provide parts of the data. Benefits of this kind of storage solution are a distribution of network load across several nodes as well as the possibility to provide redundancies which are completely transparent to the user accessing the data.

Ceph

Ceph [59] is a popular distributed storage system that provides a broad spectrum of storage solutions based on object storage [60] as a foundation. Object storage manages data as flexible-sized data containers (objects) rather than blocks of data on disk. Ceph offers different storage options which are all based on the distributed object store RADOS. Additionally, there is LIBRADOS, a native interface to the underlying RADOS object storage. Figure 2.6 illustrates the overall architecture of Ceph and its storage options.

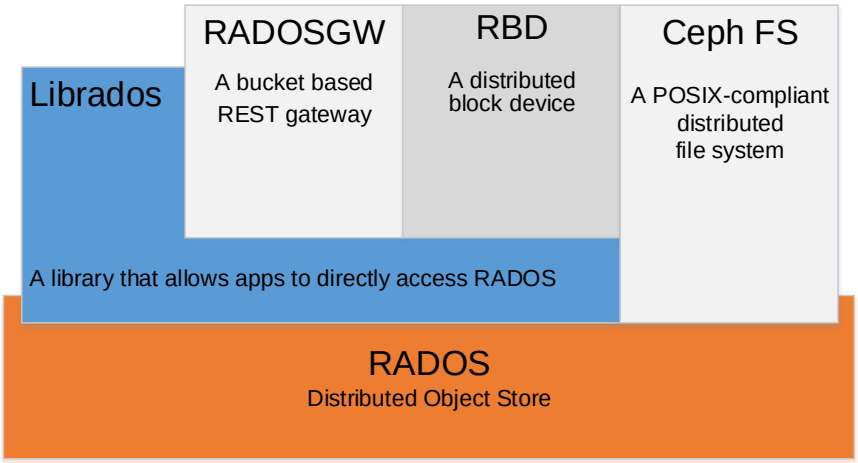

Figure 2.6: An overview of the Ceph architecture [61] showing the four storage options Ceph provides. All of these options are based on RADOS distributed object storage.

Ceph FS [62] is a POSIX-compliant distributed file system. Ceph FS uses a basic RADOS storage cluster to store its data. Figure 2.7 shows all of the components required by Ceph FS.

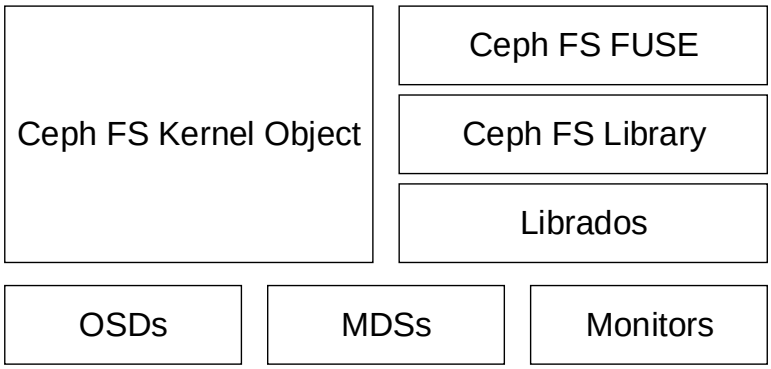

Figure 2.7: An overview of the Ceph FS components. [62]

Ceph OSD daemons (Ceph OSDs) [61] interact with the underlying physical or logical storage units (OSDs). The minimal requirement for a Ceph Storage Cluster which is configured to make two copies of its data are two Ceph OSD

daemons. Besides interacting with the OSDs to store data, handle data replication, recovery, backfilling and rebalancing, the Ceph OSD daemons also provide monitoring information to the Ceph Monitors [63]. Ceph Monitors are another required component of Ceph FS. They provide health and status information of the cluster to the users. The Ceph Metadata Server (MDS) [61] is only required for Ceph FS. Ceph FS utilizes the MDS to store its metadata. The MDS allows for POSIX file system users to execute basic commands such as *ls, find and cd*.

RADOS Block Device (RBD) [64] is a block storage solution. The data is stored striped over multiple OSDs in a Ceph cluster. Figure 2.8 shows the components of an RBD. The kernel module and librbd library are used to interact with the actual OSDs.

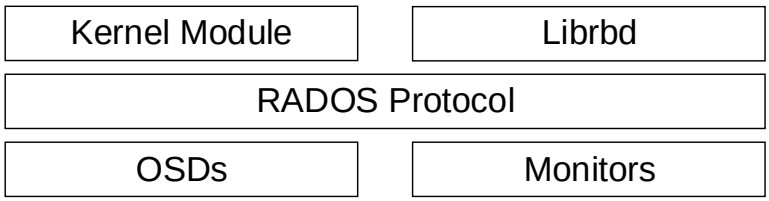

Figure 2.8: An overview of the RADOS Block Device components. [64]

The third storage option is RADOSGW [65], a bucket-based REST gateway. It uses a FastCGI module for interacting with a Ceph Storage Cluster. Furthermore, it provides interfaces compatible with OpenStack Swift and Amazon S3. Figure 2.9

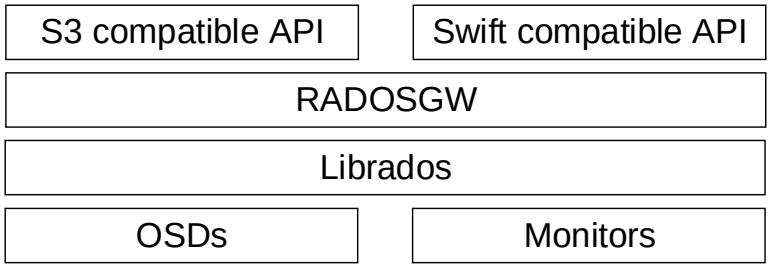

Figure 2.9: An overview of the RADOSGW components. [65]

Now that we have seen the basic components, we can look at how Ceph stores the data and how its distribution mechanism works. Ceph is based on object storage, so data is treated as a set of objects. Ceph stores the data objects in so-called

placement groups (PGs) [66], which in turn are assigned to OSDs in a storage pool [66]. The CRUSH algorithm [67, 68] determines how data is stored and retrieved. This algorithm maps each object to a placement group and then maps each placement group to one or more Ceph OSD daemons. The CRUSH map [67,68] is a map of the Ceph cluster which regulates a uniform distribution of data across the cluster. This map contains a list of OSDs, a list of *buckets* for aggregating the devices into physical locations, and a list of data replication rules. Ceph clients can communicate with OSDs directly using the CRUSH algorithm and the CRUSH map without requiring a centralized server or broker. The clients calculate the placement of the individual objects in the placement groups by hashing the object ID and applying an operation based on the number of PGs in a storage pool and the ID of the pool [61]. Figure 2.10 illustrates this data placement.

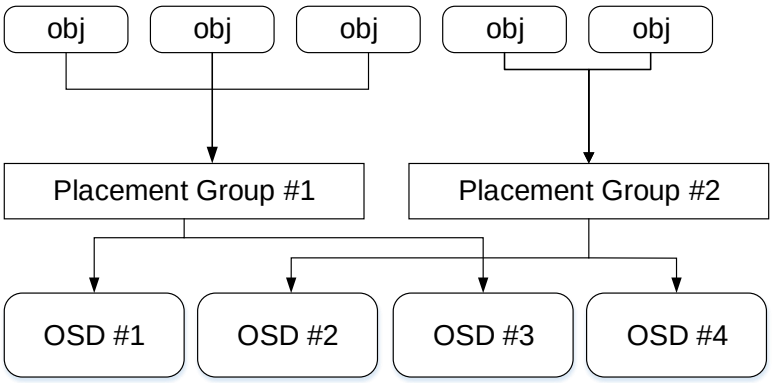

Figure 2.10: An overview of the object to placement group to OSD placement. [61] The placement is calcuated using the hash of the object ID and operations based on the number of placement groups.

GlusterFS

GlusterFS [69, 70] is free and open source distributed file system based on file storage. GlusterFS supports distributed volumes, replicated volumes, striped volumes as well as combinations of the three. Figure 2.11 illustrates a distributed GlusterFS volume which is the default setting for GlusterFS.

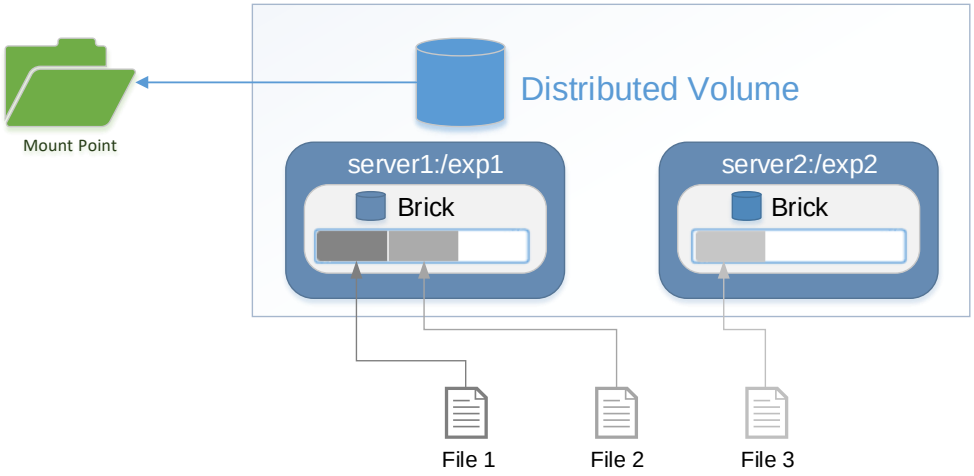

Figure 2.11: A distributed GlusterFS volume. The files are distributed as a whole between the bricks in the distributed volume. [71]

In a distributed volume, GlusterFS distributes the files across bricks in the volume. A brick is the basic unit of storage, which is represented by an export directory on a server. This server has to be part of a trusted storage pool, meaning a trusted network of storage servers. GlutserFS uses a distributed hash table (DHT) translator to select the placement of each file. In the default behavior, each brick is assigned a range within a 32-bit hash space. The file is also assigned a value in this space range by hashing its name and is then placed in the related brick. Furthermore, after the DHT translation, an automatic file replication (AFR) translator is used in cases where replication is required. This translator is responsible for maintaining replication consistency, as well as providing a way of recovering data and to serve fresh data. Another replication feature of GluterFS is Geo-Replication. Unlike AFR, Geo-Replication provides asynchronous replication across geographically distinct locations. It uses a master-slave model to replicate the data to a remote location.

An alternative to the distributed volume is the striped volume. In a striped volume, GlusterFS divides files into chunks and distributes these chunks across the bricks. Figure 2.12 illustrates a striped GlusterFS volume.

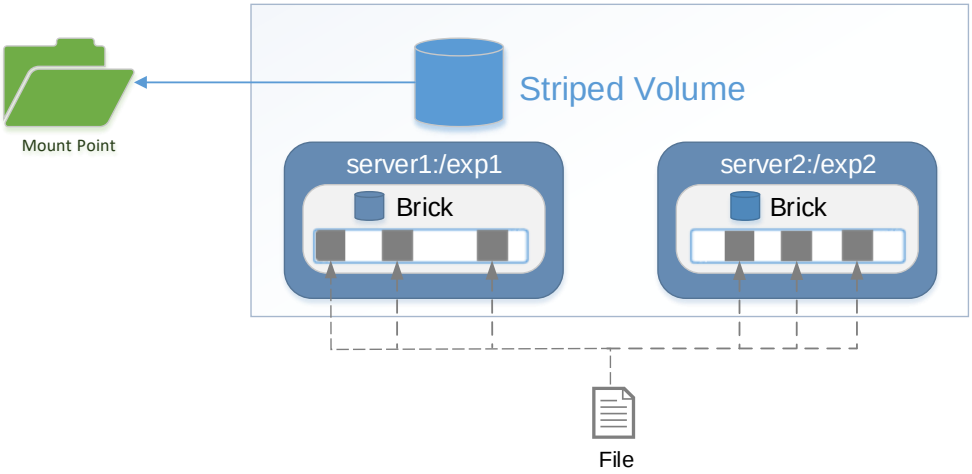

Figure 2.12: A striped GlusterFS volume. The files are split up into chunks which are then distributed between the bricks in a striped volume. [71]

A striped volume is especially useful when working with large files that would otherwise land on a single brick and effectively lead to an uneven distribution in relation to the file size. A parameter called *stripe size* dictates over how many bricks GlusterFS distributed the file chunks. The chunk size is specified by the *cluster.stripe-block-size* parameter which defaults to 128 KiB. The file system distributes these chunks in order across the bricks. The first chunk is stored on the first brick, the second chunk on the second brick and so on. For example, if we store an 800 KiB file in a striped volume with six bricks and a stripe size of six, GlusterFS splits the file into six 128 KiB and one 32 KiB chunk. The first and the seventh chunk will be stored on the first brick while the other chunks will be stored on the other five bricks in order.

Every server node must run a Gluster management daemon (glusterd). This daemon serves as the volume manager and oversees GlusterFS processes and coordinates dynamic volume operations such as adding and removing volumes non-disruptively. These daemons need to form a trusted storage pool before a GlusterFS volume can be configured. The *peer probe* command can be used to add additional storage servers to the trusted storage pool which initially only consists of the local server.

GlusterFS only runs in userspace and utilizes FUSE [72]. FUSE is a kernel module that supports interaction between the kernel non-privileged user applications. Figure 2.13 shows the process of handling a file operation in a GlusterFS volume.

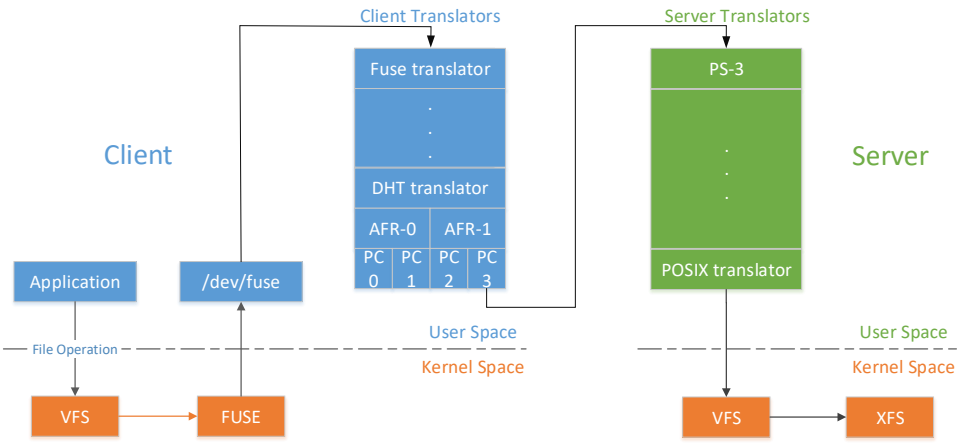

Figure 2.13: An illustration of a file operation with a GlusterFS setup. The file operation passes through several client translators, is transmitted via TCP/IP or Infiniband, then again translated by the server translators, and finally handed to the underlying file system. [70]

At the beginning of the file operation issued by the client, GlusterFS interacts with the kernel using the FUSE kernel module. This module calls the FUSE translator, and further client translations are applied as explained above. The protocol client translator (PC) at the end directly communicates with the Gluster management daemon on each brick. This translator is divided into multiple threads, one for each brick in the GlusterFS volume. The target server containing the brick translates the request using several translators starting with the protocol server (PS) translator and ending with a POSIX translator to perform the requested operation on the file system.

2.4.3 Multicast

Internet Protocol (IP) multicast [73] is a networking technology that enables a sender to send data to multiple recipients, a *host group*, at once. For this thesis, we will focus on Any-Source Multicast (ASM) for IP [74]. This multicast transmission works by using a dedicated IP address from a reserved range of IP addresses. The reserved addresses for IPv4 multicast are within the range of 224.0.0.0 to 239.255.255.255. Each IP address of this range specifies a host group. Receivers may join and leave hosts groups at any time. A single host can be a member of multiple groups. The sender, however, does not need to be part of a host group. A generic switch receiving multicast packets will forward them to every port just like a broadcast packet, flooding all available ports. Routers require support for the

Internet Group Management Protocol (IGMP) to route multicast packets. Hosts report their host group memberships to multicast routers using host membership reports. The multicast routers, on the other hand, can send Host Membership Query messages to discover which host groups have members on their attached local networks. Furthermore, switches that support IGMP snooping listen to the IGMP traffic. These switches learn to which ports hosts of a host group are connected and in turn only forward related messages to the respective ports. As a result, these switches do not flood all ports when forwarding multicast packets.

User Datagram Protocol

The User Datagram Protocol (UDP) [75, 76] is a minimal, unreliable, best-effort-message-passing transport protocol which does not guarantee delivery of data. However, it also does not require a backchannel which makes it an ideal candidate for data transfers that have bandwidth limitations and do not require mechanisms for reliable delivery of data. UDP supports multicast transmissions using IP multicast. The UDP header consists of four fields. The source port field is optional and indicates the port of the sending process. The destination port specifies the target port and is required to transmit the data. The length field describes the length of the user datagram including both the header and the data length. The fourth header field is the checksum. This checksum is the 16-bit's complement of the one's complement sum of information from the IP header, the UDP header, and the data.

Chapter 3

Analysis

Full system simulation provides functional emulation of physical hardware in combination with additional analysis tools to enable system analysis on the level of individual instructions. It is an important tool in areas such as operating systems debugging and malware analysis. The biggest issue though is the slowdown of the simulation. The main reason for this slowdown is the emulation of the workload using binary translation. Emulation is necessary because hardware-assisted virtualization does not grant the required level of control for system analysis (see Chapter 2). Additionally, memory hooks and analysis tools lead to further slowdown. Rittinghaus et al. [1] describe an average slowdown of factor 31 for functional simulation with QEMU [2] as well as a factor of 810 with Simics [3]. These slowdowns restrict the applicability of full system simulators for dynamic analysis to short-running workloads. Furthermore, they reduce the interactivity of the simulation due to long delays for responses to user input. SimuBoost [1] aims to solve this issue by distributing the simulation across a cluster of simulation nodes (consumers). The simulation host (producer) creates periodic checkpoints, for example every 5 seconds, and distributes these checkpoints across the simulation cluster to bootstrap the simulation of individual intervals. The producer utilizes hardware-assisted virtualization while the consumers use emulation which allows for better analysis. The difference in execution speed between the hardware-assisted virtualization and emulation drives a parallelization of the simulation. The speedup that can be achieved by SimuBoost is dependent on the right interval length. The checkpoint distribution delays the actual start of the simulation. This delay essentially means it takes longer for the simulation of the individual checkpoint to finish in relation to the moment the producer has created the checkpoint. As a result, the complete simulation takes longer, and the achievable speedup is reduced. Therefore, the Checkpoint distribution needs to be reliable and as fast as possible to reduce any overhead to a minimum. The goal of this thesis is to provide a distribution solution for the checkpoints that is as fast as possible to

enable SimuBoost to achieve an optimal speedup of the simulation. This distribution solution should be able to work on commodity systems using a Gigabit network connection with a theoretical maximum transmission rate of 120 MiB/s. This restriction makes using SimuBoost easier because no expensive hardware is required.

3.1 Direct Checkpoint Distribution

Eicher [4] has improved on existing checkpointing and deduplication solutions for SimuBoost, to reduce the VM downtime during checkpoint creation. Furthermore, he has developed a system for distributing simulations in a cluster. Eicher uses a modified version of the storage server backend of Simutrace [45] version 3.1.4. In his *direct distribution* approach, Simustore generates a VM image and distributes the complete VM RAM image, HDD modifications since the beginning of the runtime and device states uncompressed. Eicher has shown that using a small amount of consumers and a Gigabit network connection already saturates the network and leads to a network bottleneck. We have decided to reproduce this experiment using a bigger cluster and SimuBoost version 3.4.0. In this first direct distribution approach, we request all of the required data for each checkpoint from the producer using TCP sockets. When loading a checkpoint, the consumer creates a TCP socket connection to Simustore running on the producer. The consumer then *pulls* the required data for the checkpoint from the producer. We have tested this approach with the Phoronix *Timed Linux Kernel Compilation* (build-linux-kernel) [77] workload on a cluster with one producer and six consumer nodes. Furthermore, we are running two simultaneous jobs per consumer node, for a maximum of twelve parallel interval simulations. The optimal checkpointing interval for this scenario according to the model by Eicher [4] is around 5 seconds[1].

Figure 3.1 shows that loading a checkpoint using the direct method can take up to about 248 seconds. The mean loading time for a checkpoint in this setup is about 120 seconds. Assuming a slowdown of the simulation of factor 31 and the 5-second intervals, a simulation would take 155 seconds. Checkpoint loading times of an average of 120 seconds would increase the runtime of a simulation interval by about 77 percent. As a result, an optimal speedup cannot be achieved using this direct distribution approach. Furthermore, Figure 3.2 shows that the direct distribution approach completely exhausts the network connection of the producer when loading checkpoints. We can see a pattern which shows that the network load accumulates at certain points in time and is not distributed evenly

[1] $T_{vm} = 960s$, $s_{sim} = 31$, $s_{log} = 1$, $s_{cp} = 1\,024$, $N = 12$, $T_i = 3.7$

across the entire runtime. The reason for this behavior is the *pulling* mechanism used in this distribution. Instead of transmitting the checkpoints as soon as the producer has created them, the consumers pull these checkpoints at loading time. As a result, the network experiences no load when the consumers are performing a simulation. This distribution leads to an accumulation of data transfers to the times when the simulations finish, and new simulations start and therefore results in high and long transmission bursts which exhaust the network.

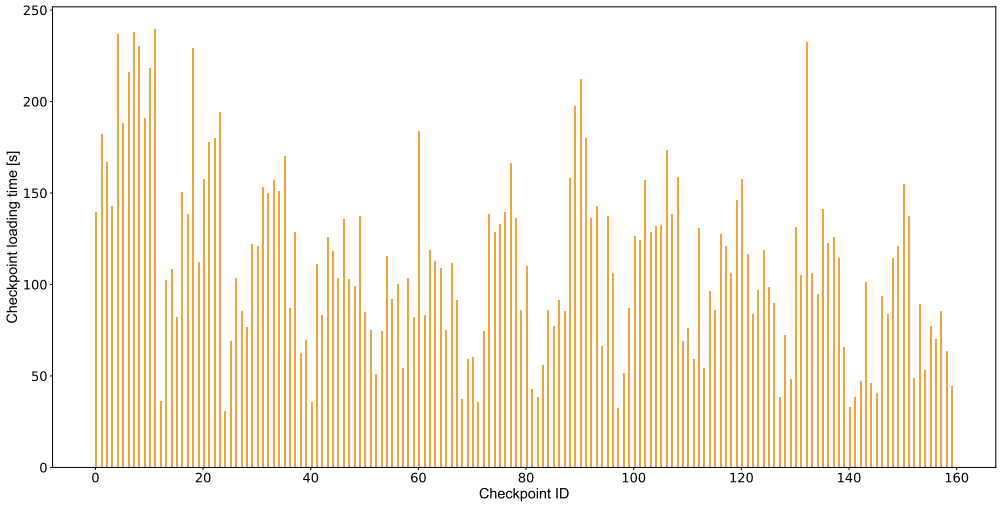

Figure 3.1: Load times for the direct distribution approach using build-linux-kernel scenario and about 5s intervals. The graph shows mean loading times of 120 seconds which is almost as high as the simulation time of an individual interval assuming a slowdown of factor 31.

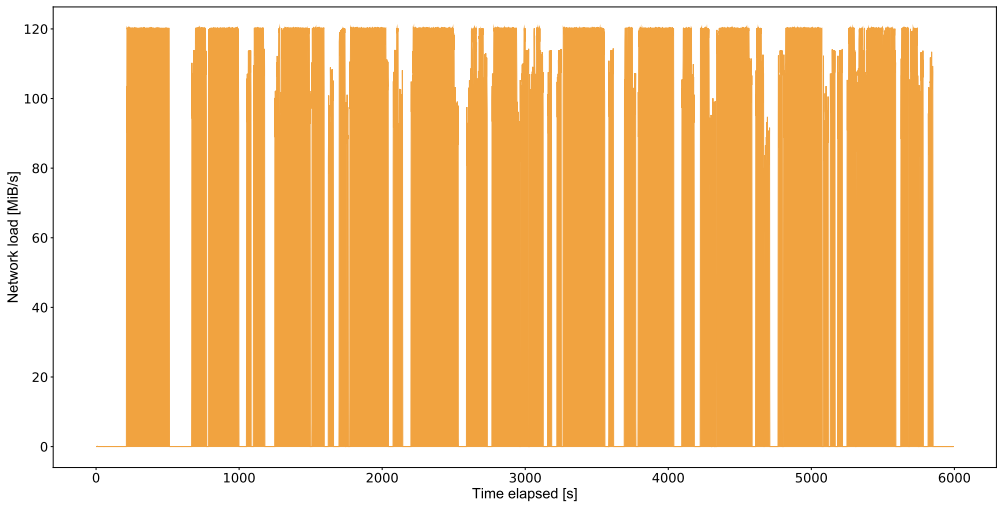

Figure 3.2: Send rate for the direct distribution approach using build-linux-kernel scenario. The network is completely exhausted during transmission bursts, reaching its maximum transmission capability of 120 MiB/s.

3.2 Bandwidth Requirements

We want to know what the minimal amount of checkpoint data is that has to be transmitted. This way we can analyze different solutions and compare them to a theoretical optimum. Not only the maximum transmission rates are important, but also the transmission rates during different times of the simulation. The ultimate goal is to achieve a solution that does not exhaust the connection between the producer and the consumers and thus allows the simulation to scale out better by avoiding a network bottleneck.

SimuBoost utilizes data deduplication and compression to reduce the amount of data it has to store on the disk. The SimuBoost checkpoints consist of *.ckpt* files and an append-only flat file (see Chapter 2). SimuBoost holds state maps for the virtual machine RAM pages and disk sectors. These state maps only contain the offset of the related data. The state maps are stored in the *.ckpt* files along with additional data that the VMM may require to be able to completely restore the virtual machine state. Examples of this kind of data are various device states such as CPU register content and metadata such as RAM size. SimuBoost stores the actual RAM and disk data in the append-only flat file. An in-memory hash table is used for data deduplication before SimuBoost appends new data to the flat file. Additionally, SimuBoost compresses the data in the flat file using lz4 [50–52] compression as well as using custom compression methods.

We have added the SPECjbb©2015 benchmark(SPECjbb) [78] to our tests to provide a broader range of use cases. SPECjbb is a very memory intensive workload that increases the number of modified memory pages and leads to an overall increased size of checkpoint data as well as an increased simulation duration. In the build-linux-kernel scenario, the compression reduces the checkpoint data from 22 GiB to 4 GiB and achieves a deduplication rate of 23.83%. In the SPECjbb benchmark, SimuBoost achieves a compression from 84 GiB to 17 GiB and a deduplication rate of 16.37%. The size of the individual *.ckpt* files starts between 250 and 300 KiB and increases up to 1.9 MiB in the build-linux kernel benchmark and up to about 800 KiB in the SPECjbb benchmark. We have monitored the disk write rate on the producer during the benchmarks to get the exact amount of data that is written at each point in time. Figure 3.3 shows the amount of disk writes during the linux-kernel-build scenario as well as a SPECjbb scenario.

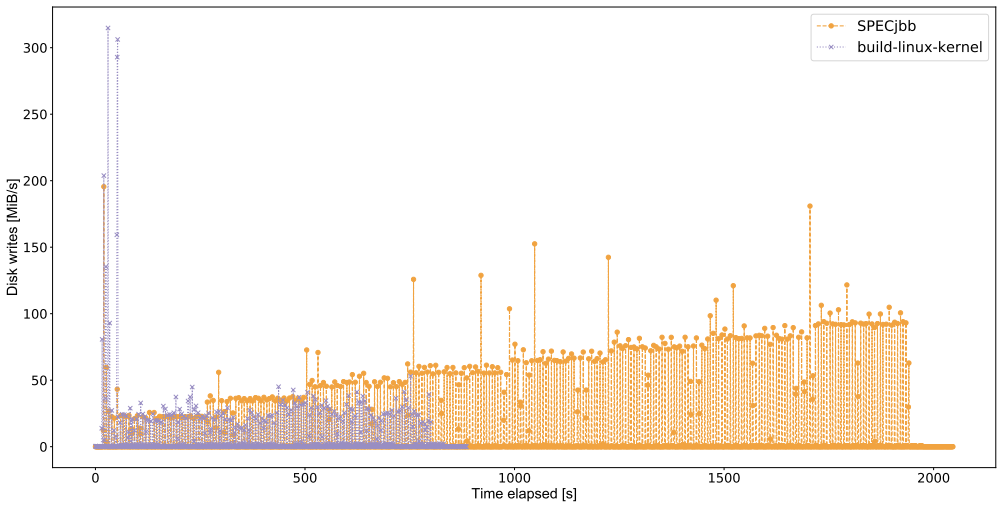

Figure 3.3: Disk write rate for build-linux-kernel and SPECjbb scenarios. These rates are low enough to allow transmission using a Gigabit Ethernet connection.

Figure 3.3 depicts a disk write spike at the beginning for both, the SPECjbb and the build-linux-kernel scenarios, due to the initial RAM image being written. After this initial increase, the write rate consistently stays below 55 MiB/s for the build-linux-kernel scenario as well as below 100 MiB/s for the SPECjbb benchmark with the exception of 15 higher peaks of which six are above 125 MiB/s. We can neglect these peaks in the SPECjbb benchmark as they are not consecutive and only last a second each. These results show that a Gigabit network connection is conceptually fast enough for our scenarios. Our goal, therefore, is to get the checkpoint data transmission rates as low and as close to the disk write rates

as possible, so they do not significantly affect the load times of the checkpoints. We also want to achieve a scalable solution that reduces the relation of network load to cluster size. The direct distribution approach has shown to be impractical as it transmits more checkpoint data because it does not utilize the checkpoint compression. Furthermore, the network load increases with the number of consumers as each consumer needs to receive at least the complete VM RAM image and other modifications as mentioned above. As a result, the direct distribution approach exhausts the network and leads to high checkpoint load times.

3.3 Pulling vs. Pushing

By using a network attached storage (NAS) solution such as Samba, we can benefit from the compression. Simutrace compresses the checkpoint data and stores the compressed data on disk. Using a NAS, we can *pull* the already compressed checkpoints from the producer to the consumers instead of loading the uncompressed data from Simutrace as we do in the direct distribution approach. This new approach reduces the network load and as a result, decreases the checkpoint loading times because there is less data we need to transmit. Local Simutrace instances on every consumer node handle the decompression and reconstruction of the VM state. Figure 3.4 shows the resulting checkpoint loading times compared to the direct distribution approach.

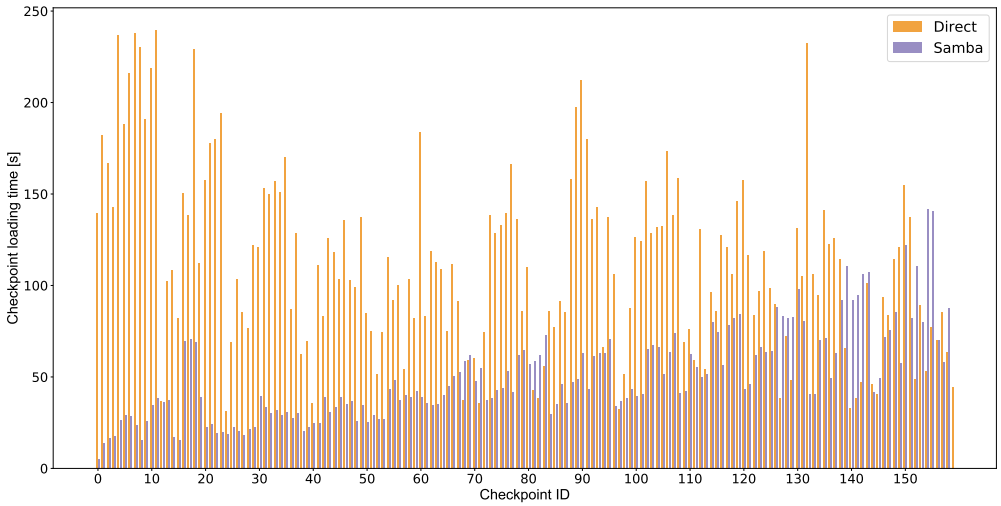

Figure 3.4: Comparison of load times between the direct distribution and Samba approach using build-linux-kernel scenario. Samba profits from the data compression at first while getting closer to the direct distribution approach towards the end of the simulation.

Especially the first checkpoints profit from the compression as depicted by the shorter loading times. We can also observe that the loading times get close to the direct distribution approach. The reason for this behavior is the distribution of the network load as can be seen in Figure 3.5. At first, we have a low network load due to the high compression of the checkpoints. After a while, with the compression becoming less efficient, the bursts get longer and similar to the direct distribution approach, which explains the almost equal loading times at the end. The reason for the worsening compression is the fact that at first, the VM RAM image consists mainly of zeros, which are easy to deduplicate and compress. Over time though, the guest system overwrites these zeros with data which makes deduplication and compression less effective. The highest loading time in the Samba approach is about 142 seconds and the mean loading time for a checkpoint is about 51 seconds. These times are a significant improvement over the direct distribution approach reducing the mean checkpoint loading time by more than a half. However, this is still about 10 times higher than the optimal interval length. Additionally, the mean checkpoint loading time will get worse for longer running workloads because the compression gets less effective the longer the workload is running.

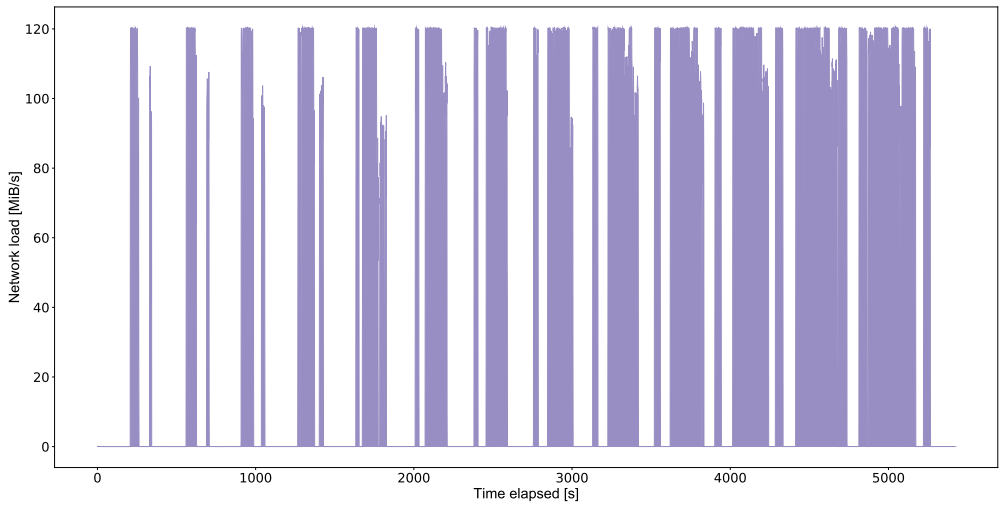

Figure 3.5: Send rate for the Samba approach using build-linux-kernel scenario. Samba needs to transmit fewer data in the beginning due to a high data compression but gets close to the direct distribution approach towards the end of the simulation.

The NAS approach still *pulls* the checkpoint data from the producer leaving the connection between the producer and the consumer nodes a bottleneck due to the cumulated transmission bursts over a single connection. *Pushing* the incremental checkpoint data to the cluster nodes could reduce the load on one single

connection at every checkpoint load. The amount of data pushed would equal the disk write rate of SimuBoost. Additionally, pushing could lead to a more evenly distribution of the data. By avoiding a cumulation of network transmissions, the network load could decrease which would result in better checkpoint loading times overall. Furthermore, this kind of distribution approach would avoid the increasing network overhead at the last third of the checkpoints as seen in Figure 3.5. The slowdown seen in the last third of the Samba approach comes from a worse compression and not from an increased amount of data. We can conclude this behavior from Figure 3.3 which shows that the write rate stays constant for the build-linux-kernel scenario.

3.4 Checkpoint Distribution

We are considering two possible solutions for *pushing* checkpoints through the network. We can push the compressed incremental checkpoint data to a single consumer at a time while changing the target consumer every time to achieve an even distribution. As an alternative, we can also push the compressed incremental checkpoint data to multiple consumers at once. There are two possible approaches we can use.

Distributed File System One solution is a distributed file system [79, 80]. A distributed file system regulates the simultaneous access to data files from multiple clients as well as the data distribution between multiple storage hosts. From the view of a single client, the distributed file system is not distinguishable to a local file system, however the data the clients are accessing may be distributed across multiple storage hosts. In a distributed file system, the producer sends the data of the incremental checkpoints only once. The file system then distributes the checkpoint data to the consumers. This distribution reduces the network load on a single connection, in particular between the producer and the consumers, and distributes the checkpoint data across all of the consumers. However, this distribution may lead to a lot of redundant data transfers based on the key for the distribution and the amount of data that is required by a consumer. In the worst case scenario, a consumer needs to retrieve all of the checkpoint data from other consumers in the network and possibly even all of the data at a certain point in time from a single consumer. This case would be similar to the pulling scenario from the NAS approach in terms of network load distribution. Though instead of pulling all of the checkpoint data from the producer, the consumer would pull all data from another consumer node in the simulation cluster. This is still a slightly better distribution than in the NAS approach because not all of the checkpoint data lies on a single system.

Multicast Another possible solution which avoids the distribution problem of a distributed file system and possibly also some of its overhead is multicast. Schmidt et al. [81] have shown that multicast offers the best performance for efficient distribution of virtual machine images in a cloud computing environment. While it is not the same scenario, it is similar regarding the amount of data that is transmitted and the distribution using a single sender and multiple receivers. In a multicast approach, the producer sends all of the checkpoint data once. Every node receives all checkpoints, and ideally, there is no additional transmission necessary. This distribution would make it an optimal solution with the only downside of an increased storage capacity requirement for every consumer node. In practice, however, multicast connections suffer from issues such as out of order delivery and packet loss which may impact the approaches efficiency and requires a mechanism to handle these issues.

3.5 Conclusion

As we have seen in our first benchmarks, the direct distribution approach not only exhausts the network connection but as a result leads to very long checkpoint loading times, which makes the goal of an optimal simulation speedup unachievable. Utilizing checkpoint compression, for example by using a network attached storage solution such as Samba, reduces the amount of checkpoint data that has to be transmitted. However, because of the *pulling* mechanism, this distribution still leads to a bottleneck at the connection between the producer and the consumer nodes due to cumulated network transmission bursts. Using *pushing* could help distributing the network load more evenly across the simulation time. A distributed file system could help with reducing the amount of checkpoint data that has to be transmitted at every checkpoint load by initially pushing the checkpoint data evenly across all of the consumers. The consumers then exchange the required data between each other thus distributing the network load across the simulation cluster. Multicast can decrease the amount of transmitted checkpoint data even more to the point where, in the best case scenario, every checkpoint is only transmitted once. In this case, the producer pushes out the checkpoints once Simustore has written them to the disk and the consumers receive the data of the created checkpoints all at once.

Chapter 4

Design

We have seen in our analysis in Chapter 3 that we need to reduce the network load on the connection between the producer and the consumer nodes. Figure 4.1 illustrates the bottleneck between the producer and the consumer nodes in our setup. We use this scheme as a baseline for our data distribution solutions. The producer runs the virtualized system and creates the incremental checkpoints. The consumers are the simulation nodes which pull the checkpoints from the producer. Because the consumers pull all of the data from the producer, the connection between the producer and the consumers becomes a bottleneck.

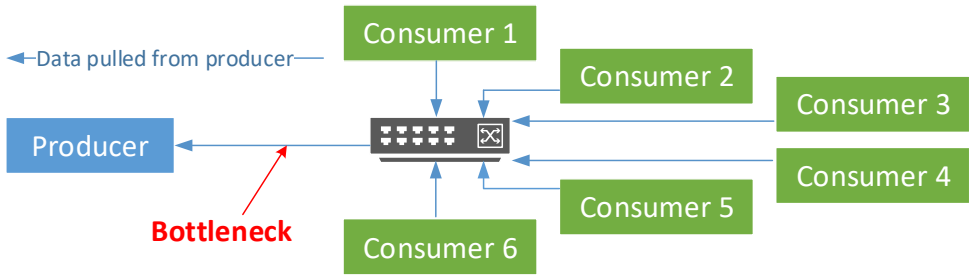

Figure 4.1: Illustration of the data distribution using the direct distribution and Samba approaches. All data gets pulled from the producer leading to a bottleneck.

Our analysis has shown that pushing the checkpoint data to the consumer nodes is more desirable than pulling the data from the producer. The pushing mechanism allows for a more evenly distribution of the network load instead of the accumulated bursts we have observed when pulling the data using the direct distribution and Samba approaches. Our ultimate goal is to reduce the transmission overhead of the parallel simulations to a minimum, so the checkpoint loading

time does not limit the achievable speedup of SimuBoost. Utilizing SimuBoost's compression does significantly reduce the amount of data that has to be transmitted and therefore needs to be included in our solution. The compression reduces the amount of data to allow transmission via a Gigabit Ethernet network, which allows us to use commodity hardware and therefore increase SimuBoost's applicability. Additionally, we want to reduce the relation of the bandwidth requirements to the cluster size to increase SimuBoost's scalability. In the direct distribution and Samba approaches the network load increases with the number of consumers as each consumer needs to receive at least the complete VM RAM image and other modifications, which limits the scalability of SimuBoost. Based on our analysis, we have decided to evaluate distributed storage as well as multicast transmission as solutions to the checkpoint distribution problem.

4.1 Distributed Storage

In the distributed storage approach, the producer pushes the data of a checkpoint across all consumer nodes as soon as the checkpoint is written to disk. Each node then reconstructs the checkpoints by accessing the local storage and pulling all missing data from the other consumers. This approach is similar to the Samba solution shown in Chapter 3. However, the network load is distributed between the consumer nodes and instead of pulling all data from the producer, we pull some of the data from the other consumers. As a result, we try to reduce the bottleneck on the connection between the producer and the consumers and achieve a more even distribution of transmissions across all consumer nodes. The actual distribution however is based on the algorithm for the data distribution used by the storage solution and the data accesses required for the checkpoints whose uniformity is based on the level of data deduplication. A further benefit over the direct distribution solution is the utilization of the file system cache, which should reduce the amount of data that has to be pulled from other consumer nodes. Figure 4.2 illustrates the data distribution concept using a distributed file system. It shows that initially, the producer pushes the checkpoint data to the cluster. When the consumers load checkpoints, they pull any data that is not already available locally from the other consumers.

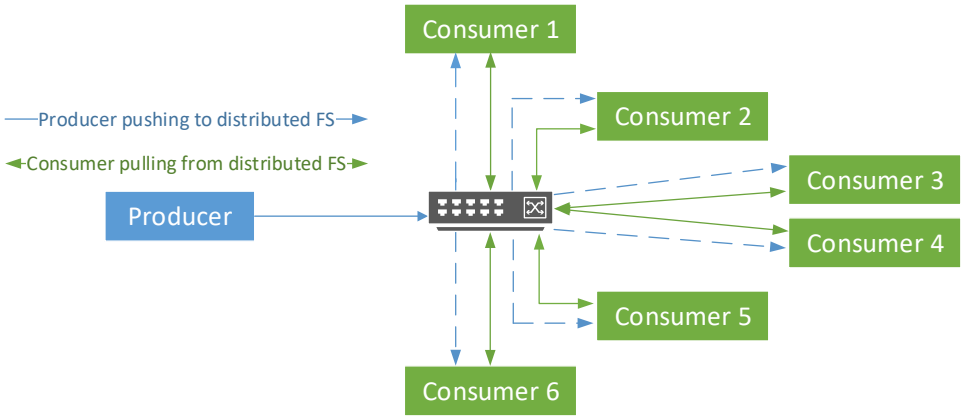

Figure 4.2: The producer distributes the checkpoint data across the distributed FS. A file system algorithm selects one consumer for every chunk of data and sends the data to the respective consumer node. The consumers exchange required data when necessary.

Replication is an important factor in distributed file systems. With an increasing amount of physical storage hardware, the chance of a failure increases as well. For our use case, however, replication is not required and would lead to an increased amount of network load due to redundant data being transmitted. The loss of data, and therefore the loss of an entire consumer node, would break a simulation interval. Even though the other simulation intervals could continue to run in case replication is enabled, we have still decided to disable replication for our use case. The reason behind this decision is that the main use case of SimuBoost is research and development. We argue that an experiment can be repeated in case of hardware failure and want to focus on a fast data distribution with a reduced likelihood of a network bottleneck. Replication can still be enabled if required, for example, because of experiments which are difficult to repeat or production load that needs to be simulated. These cases, however, are out of the scope of this thesis.

Giacinto Donvito et al. [82] have compared HDFs [83], GluterFS [69], and Ceph [59]. They have concluded that GlusterFS is the fastest of the three solutions. Furthermore, they have concluded that Ceph seems promising though it has stability and performance issues. We have decided to evaluate Ceph and GlusterFS for our solution. We are considering Ceph in the hope that the stability and performance issues have since then been resolved.

Ceph Ceph is a popular distributed storage system which offers three different storage options. These storage options are all based on the distributed object store RADOS. Object storage manages data as flexible-sized data containers (objects) rather than blocks of data on disk. Ceph FS [62], a distributed file system, is one of the storage options Ceph provides and the easiest to use in our approach. Ceph FS, like the other storage options, uses a basic RADOS storage cluster to store its data. However, it also potentially has the highest overhead. For example, it requires at least one Ceph metadata server to provide basic POSIX functionality such as the `ls`, `find`, and `cd` file system commands. RADOS Block Device (RBD) [64] is a block storage solution. RBD is supposed to be faster than Ceph FS, but accessing it from different nodes is more challenging due to the requirement to have a storage solution that allows simultaneous data accesses from different consumers. If multiple clients mount a RBD using a regular file system instead of a distributed file system on top, several problems can occur. One of these problems is the lack of notification about file system changes to other consumers. If consumer A writes new data to the RBD, consumer B does not know about these changes. Without a notification mechanism, consumer B would have to actively check the file system for changes repeatedly. Another problem comes from simultaneous reads and writes. Without a proper file locking mechanism, simultaneous reads and writes from multiple consumers to the same file could lead to corrupt data. Consumer A might be modifying a file while consumer B reads the file and therefore receives intermediate data that does not equal the file before or after the modification. Similarly, two consumers writing to the same file might lead to corrupt or lost data. The third option is RADOSGW [65], a bucket-based REST gateway. It uses a FastCGI module for interacting with a Ceph storage cluster. Furthermore, RADOSGW provides interfaces compatible with OpenStack Swift and Amazon S3. For use with SimuBoost though, it would be necessary to integrate RADOSGW manually. SimuBoost stores data on a regular file system and expects the interface to be designed that way. For example, SimuBoost utilizes memory mapped files and benefits from the file cache, both of which would not be applicable to a REST gateway. As a result, the REST gateway is the most complex solution for our use case. Therefore, we have selected Ceph FS for our approach.

GlusterFS Other than Ceph, which is based on object storage, GlusterFS is based on file storage and is, therefore, simpler to set up and configure for our use case. It utilizes the file system already set up on the storage cluster nodes to create a distributed volume. A GlusterFS distributed volume consists of bricks. A brick is an exported folder on a storage cluster node. GlusterFS distributes the data across the bricks based on the selected distribution mechanism. GlusterFS supports a file-based distribution, which is the default, as well as striping [84].

Both the file-based distribution, as well as the striping distribution, can be combined with replication. The default file-based distribution uses a distributed hash table to select the placement of each file within the distributed volume. The striping distribution splits the files into chunks and distributes them across the cluster. Because the complete simulation data consists of a single large flat file and multiple small *.ckpt* files, the file-based distribution does not fit our scenario well as it would put the large flat file into a single brick. As a result the connection between the consumer containing this brick and the other consumers would become a bottleneck. Therefore, we test the striping distribution for our approach.

4.2 Multicast

Multicast is another possible solution to solve the checkpoint distribution problem. In this approach, the producer pushes the data of the incremental checkpoints to every node as soon as the data has been written to disk using a single multicast transmission for each checkpoint. As a result, there is a fixed amount of network traffic that does not depend on the number of consumer nodes, but only on the disk writes of SimuBoost, which in turn is determined by the modification rate of the workload and the compressibility of the data. Furthermore, in contrast to the approach using a distributed file system, there is no additional communication between the consumers. The main downside of the multicast solution is the increased storage capacity that is required on every node. However, there are a few further challenges that have to be taken into consideration regarding the data integrity and reliability of multicast.

4.2.1 Data Integrity

To guarantee accurate simulation intervals, we have to make sure that the data we receive is valid and not changed in any way. Through the use of UDP and the UDP checksum, we can assume that all of the packets we receive are valid. The UDP checksum is 16 bits long which means that there is a 1 out of 65536 chance that a corrupt packet is not detected as such by the checksum. We deem this sufficiently low to justify not providing any further checks. Furthermore, due to the compression of the checkpoint data, there is also a chance that any corruption not detected by the UDP checksum will lead to a failure during decompression of the data and thus not remain undetected.

4.2.2 Reliability

IP multicast as we use for our setup is unreliable. In IP multicast, we send our data packets to a specific IP address from a range of addresses reserved for multicast. In case of IPv4, this range is from 224.0.0.0 to 239.255.255.255. The multicast receivers listen for packets sent to the selected IP address from the multicast range. Unlike TCP, there is no backchannel to the sender. As a result, the sender does not know whether the receiver can keep up with the transmission, packets got lost, or packets arrive out of order.

Many reliable multicast protocols, which work on top of IP multicast, have been proposed to deal with these issues. ACK-based protocols [85, 86] introduce a TCP-like backchannel and require the sender to maintain the state of all receivers. These protocols use unicast ACKs or non-acknowledgments (NAKs) to inform the sender about their state. NAK-based protocols such as LBRM [87], RAMP [88] and NORM [89] only send NAKs to the sender when a retransmission is required. Due to the missing ACKs, the sender has no state information and cannot determine what data has been received and can be safely released from memory. Therefore, additional mechanisms like polling need to be implemented in these protocols. Ring-based protocols [90, 91] combine the throughput advantage of NAK-based protocols with the reliability of ACK-based protocols with the addition of tokens and timers. Tree-based protocols [86, 92] aim to increase the scalability of reliable multicast by processing only aggregated acknowledgments. These protocols are characterized by dividing the receivers into groups forming a tree structure. Ryan G. Lane et al. [93] have evaluated several of these protocols in the context of grid computing.

However, we have decided to implement our own, specialized solution. One of the reasons for this decision is that we want to reduce any overhead as much as possible and we want to keep the complexity of any additions to SimuBoost as low as possible. A major benefit of our specialized solution over a generic reliable UDP protocol is the fact that we do not need to retransmit every lost packet on every consumer node. While we do send all checkpoint data to every consumer, each consumer only requires the data necessary for the checkpoints it is assigned. Therefore, it is unnecessary to retransmit every lost packet, which means that we can reduce the retransmissions to a minimum. Another benefit of our design is the distribution of the retransmissions along the time axis. Most data loss occurs during times of high network load and therefore during the runtime of the benchmark. A generic reliable multicast solution would retransmit data as soon as the loss is detected and therefore further increase the network load during, or shortly after the workload runtime. However, due to the slowdown of the simulation intervals, the overall simulation runtime is longer than the benchmark time, and the lost data may be required after the workload has finished. Figure 4.3 illustrates

this behavior. Our approach retransmits the data at the time it is required by the consumer. As a result, the retransmissions are spread more evenly across the overall simulation runtime. In the following subsections, we will look in more detail into the specific variables regarding multicast reliability.

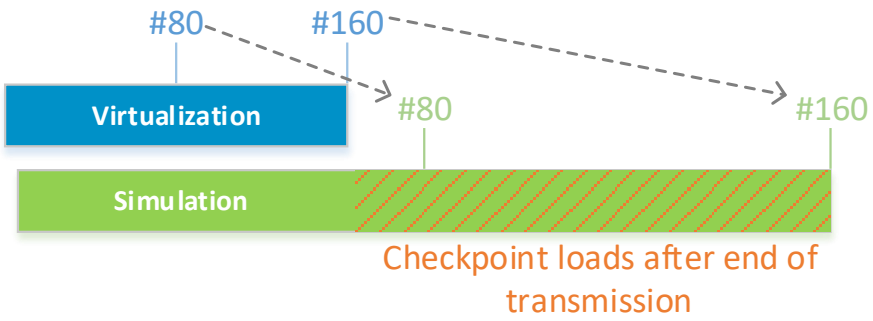

Figure 4.3: Most checkpoints are loaded after the benchmark has finished and therefore after the initial transmission of the checkpoint data.

Out-of-Order Delivery

Due to buffers on the sender side, the switch, and the consumer side, packets can change their order. Furthermore, different paths individual packets take through the network, as well as middleware like firewalls that inspect packets, can also lead to a changed order of packets at the receiver. Protocols such as TCP handle this issue by re-ordering packets on the receiver side or requesting retransmissions. We do not fix ordering in our approach. Instead, we deal with this problem by using sparse files and by transmitting the data offset with every packet. As a result, we can write every packet of data to its designated offset without having to care about the order these packets arrive in.

Packet Loss

Packet loss can happen on different stages of the transmission and for several reasons. One of the reasons are buffer overflows. When the data arrives at the receiver it is first loaded into the buffer of the network interface, from there, the network stack loads the data into an operating system buffer (receive buffer). Both buffers are limited and can overflow which results in packet loss. The overflows of the receive buffer happen when the receiving application cannot keep up with the rate at which the data arrives. Another reason for packet loss are errors on

the transmission path due to physical problems like broken cables or switches that modify data and produce malformed packets. The receiving network interface drops all of these malformed packets.

Packet Loss Reduction To avoid excessive packet loss, we increase the receive buffer size of the consumer nodes. We can reduce the impact of short transmission bursts using the increased buffer. Furthermore, we implement a rate limiting algorithm on the producer to control the average transmission rate as well as burst rates for the data transfer. Using rate limiting, we pause the sender if a certain threshold of transmitted bytes per second is reached. We try to achieve a reasonable rate limit to reduce the amount of data that reaches the consumer nodes at a time. This rate limit should allow the consumer to catch up, while not slowing down the transmission to the point where we reduce the achievable speedup of SimuBoost. The rate limiting does rely on gaps in between the transmissions in order to distribute the network load using a lower transmission rate across these gaps. If there are insufficient transmission gaps, the overall transmission time will increase and therefore lead to delays which might impact checkpoint loading times.

Packet Loss Handling In a typical reliable environment, sender and receiver communicate with each other and the sender re-transmits any lost packets to the receiver. This retransmission works because the receiver reports back to the sender. This backchannel, however, significantly increases the network load and overall processing overhead for both the sender and the receiver. Therefore, we do not set up a backchannel for our multicast approach. Instead, we detect lost packets on the receiver by counting the amount of data received and comparing it to the overall file size which we receive as part of the information sent with each data packet. Should all data packets of a file get lost, the receiver would not have a local copy of the file which also indicates packet loss. In both cases, we switch to a fallback solution in which the receiver retrieves the missing data through a reliable Samba NAS. This way we can still profit from the checkpoint data compression. As a result, we have two possible data flows in the multicast approach: the multicast transmission for which the producer pushes the data to the consumers and the Samba repair transmissions for which the consumers pull the data from the producer. Figure 4.4 illustrates these data flows.

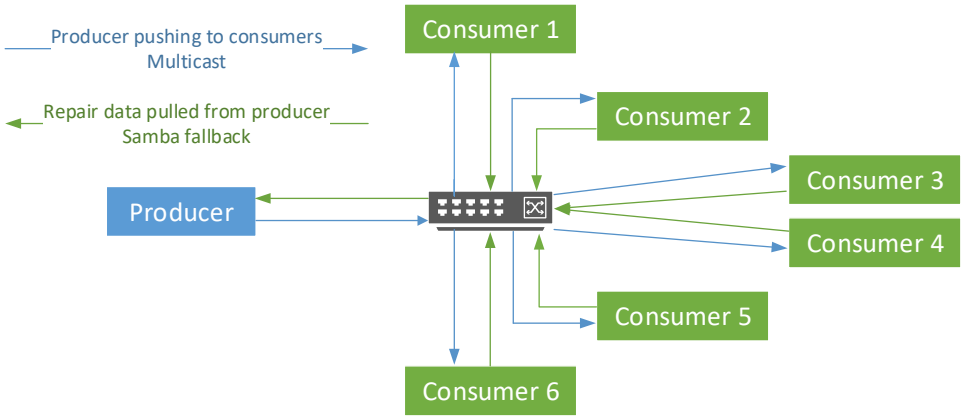

Figure 4.4: The producer pushes the data via multicast to the consumers. The consumers pull lost data from the producer using Samba.

4.3 Conclusion

Both, the distributed storage approach, as well as the multicast approach, are promising solutions which are posing different challenges. The distributed storage solution poses the risk of too much additional overhead due to metadata handling, data distribution, and simultaneous file system access from different nodes. While the producer pushes the data to the distributed storage, the individual consumer nodes still need to pull at least some parts of the checkpoint data from the other nodes. We cannot predict what data a consumer requires before it loads its assigned checkpoints. Therefore we aim to achieve an even data distribution in the distributed storage cluster. In the worst case scenario, we pull all data for multiple simulation intervals from a single node, which is still better than pulling all data for *all* simulation intervals from the producer. We have selected the object-storage-based Ceph FS and the file-based GlusterFS for our distributed storage design. We will disable any data replication if possible to avoid duplicate data transmissions. We do not need to be able to handle the loss of a consumer node. Multicast, on the other hand, makes sure that every node has all of the data, so conceptually, there are no additional data transfers necessary. In practice, however, multicast poses risks and overhead regarding packet reliability. Packets can get lost, and as a result, we have to repair the checkpoints by pulling the lost data from the producer. While we cannot completely prevent packet loss, we can reduce it by modifying receive buffers as well as by limiting transmission rates.

Chapter 5

Implementation

In this chapter, we explain the implementation of our multicast solution. In our multicast implementation, we create a UDP socket to transmit all of the data that is created during the checkpointing as soon as the related checkpoint has been written to disk. As a result, every node receives the complete checkpoint data from the producer without requiring to request it when loading the actual checkpoint. However, reliability is an important factor considering that IP multicast is by default unreliable. Therefore, we have to deal with the out-of-order delivery of packets as well as packet loss. As mentioned in Chapter 4, we have designed a custom reliable multicast solution using increased buffer sizes and rate limiting for packet loss reduction. Furthermore, we use a reliable Samba NAS as a fallback solution to retransmit lost data.

5.1 Packet Loss Reduction

The pushing mechanism we use results in a high network load at the beginning of the simulation because of the initial VM RAM image and device data we transmit. As a result, we push more data out than the consumers can read, which leads to packet loss due to overflowing receive buffers. We have implemented two methods to reduce packet loss. We have increased the socket buffers on the consumers, and we have implemented a rate limiting algorithm on the producer to reduce the amount of data the producer sends at once. It is important to reduce the amount of packet loss because every set of data that has to be repaired requires a retransmission which leads to more network load.

Receive Buffer We have decided to increase the socket receive buffer on the consumer nodes to reduce the impact of short, high transmission bursts as we have observed during the beginning of the workload runtime in Chapter 3. We

have expanded SimuBoost with a parameter to set the buffer size of the receive socket. Additionally, it may be necessary to increase the system-wide maximum receive buffer size of the operating system. On Ubuntu 16.10 for example, we use the command *sysctl -w net.core.rmem_max=209715200* to set the maximum receive buffer size to 200 MiB.

Rate Limiting We have implemented a token bucket [94] algorithm to control the rate at which data is sent by the producer. The algorithm helps us to reduce the network load across the entire workload runtime. Given a target transmission rate r, a refresh rate [1] S and a burst rate [2] b we fill the bucket of size b with $(r * S)/1000$ tokens every S milliseconds. If the bucket is full, we discard any additional tokens. A thread increments this counter after S milliseconds. Every time the producer tries to send new data, it checks if the size of the data to be transmitted is less than or equal to the counter size and if so it reduces the counter by this value and transmits the data. SimuBoost will try to send data until there are not enough tokens available. The checkpointing process is blocked until there are enough tokens for the current transmission. We have decided to implement this behavior instead of putting the checkpoints in a queue and continuing to create further checkpoints. When queuing data for transmission, we would expect the bandwidth to free up soon so the queued data can be transmitted and does not built up any more. However, we have decided to instead throttle the producer so data does not pile up in case the network bottleneck lasts longer. The sending rate influences the optimal value of the socket receive buffer on the consumers. If data is sent at a too high rate over a prolonged time, it can lead to receive buffer overflows as more data reaches the target. A data loss and therefore required repair is more expensive due to additional transmissions and therefore should be avoided as much as possible.

We have expanded SimuBoost with parameters for the token bucket algorithm. These parameters allow users to configure the rate limiting based on their requirements.

5.2 Packet Loss Handling

While an increased receive buffer size and rate limiting help in reducing the number of lost packets, they do not completely eradicate these losses. Therefore, we have implemented several ways of packet loss detection and repair. To detect the

[1] The rate at which we fill the bucket in milliseconds.

[2] The maximum amount of data that can be sent in addition to the transmission rate.

loss of *.ckpt* file data, we keep track of every *.ckpt* file we have received by storing it in a hash table with the checkpoint id as the key. Every *.ckpt* file packet contains the checkpoint ID, the offset of the data to handle out-of-order delivery, and the size of the complete *.ckpt* file. We count the amount of data that has been received for every checkpoint and store it in the *.ckpt* hash table. When loading a checkpoint, we compare the amount of data that has been received with the total *.ckpt* file size. In case the received amount of data is less than the *.ckpt* file size or the file is not present at all, we copy the data from a reliable Samba NAS as a fallback solution. However, the Simustore needs to run for the entire simulation runtime, else we loose the data in the hash table and load every checkpoint from the fallback.

When SimuBoost tries to load data from the flat file, we first check the length and validity of the content we try to read. Each memory segment contains information about the size of the data in this segment. It is stored in a structure called *SimuBoost1DbEntryHead*. If the size we read from this structure is zero, we first copy the complete entry head from the Samba fallback. Next, we check if the new size is valid and then copy the remaining content of the memory segment size from the fallback as well. SimuBoost also checks whether the requested offset is within the range of the flat file. If it is not, SimuBoost updates the internal representation of the database file once to check if new data is available. If the offset is still not within the current range of the file, we copy the the *SimuBoost1DbEntryHead* at the requested offset from the Samba fallback. Next we read the *uncompressedSize* from the head structure and copy this amount of data as well. Afterwards, SimuBoost updates the internal representation again and throws an exception if the requested offset is still not within the range. However, other errors in the *SimuBoost1DbEntryHead*, the content after the head, as well as in pages referenced by delta pages can still slip through this check. Fortunately, the decompression method is very likely to fail if the data is still invalid and throws an exception int hat case. SimuBoost uses two kinds of compression, lz4 [50–52] as well as a custom compression method. Both compression methods utilize inter-block compression. Instead of compressing blocks of data independently, the algorithms compress each block based on information of the previous blocks. As a result, an invalid block affects other blocks as well. We catch the exception thrown by the decompression method once and copy the data at the requested offset and of the provided buffer size from the Samba fallback. Additionally, we check if the content at the offset contains a delta page and if so, we repair the page referenced by this delta page as well.

Chapter 6

Evaluation

We use the same methods already introduced in our analysis (see Chapter 3) to evaluate our solutions. We look at the checkpoint loading time as well as the network load during the simulation. We will compare our results with the values of the direct distribution approach.

6.1 Job distribution

This section explains how we distribute the simulation jobs in our simulation cluster. The job distribution is an important factor. Even though it is not the purpose of this thesis to evaluate job distribution, it is a vital part of the checkpoint distribution. Using the same checkpoint distribution mechanism with a different job distribution mechanism might lead to different results.

iWatch iWatch [95] is a realtime filesystem monitoring program based on the Linux kernel subsystem inode notify (inotify) [96]. We use iWatch to monitor the Simustore working path to detect newly created checkpoints.

SLURM Simple Linux Utility for Resource Management (SLURM) [97] is an open source cluster workload manager. SLURM allocates resources, provides a framework for job management and manages a queue of pending work to arbitrate conflicting requests. We use SLURM to distribute the simulation jobs in our simulation cluster. We use a SLURM configuration with the *serial* select type, the *CR_CPU* select parameter and a *backfill* scheduling. This means that all available resources on a simulation node will be filled first before the next node receives new jobs.

Job Creation and Distribution Approach iWatch launches a SLURM job for each newly created checkpoint. SLURM then assigns this job to a simulation node in the cluster which in turn executes a checkpoint loading script. This script launches a QEMU instance which connects to the already running Simustore. The script then loads the checkpoint in that QEMU instance and then sleeps for a pre-calculated amount of time, based on the model by Eicher [4]. We use the pre-calculated value to estimate the runtime of the interval, because our SimuBoost prototype does not support deterministic replay (see Section 2.2), yet. Figure 6.1 illustrates this approach. After the simulation has finished, the script shuts down the QEMU instance and exits. The SLURM daemon on the consumer notifies the SLURM control daemon (SLURMCTLD) about the exit status of the script, and the SLURMCTLD removes the job from the job queue.

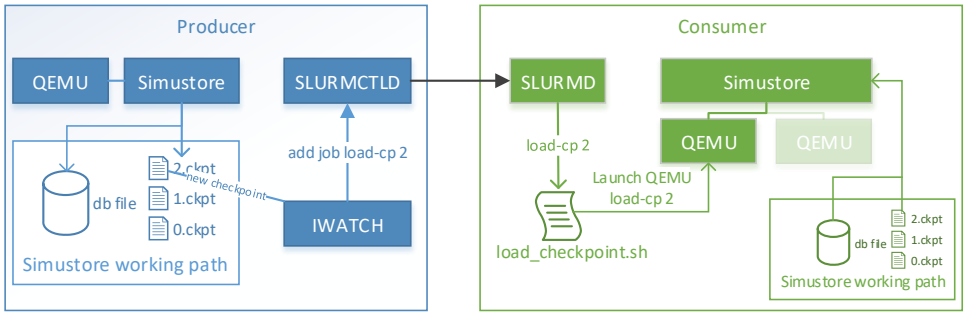

Figure 6.1: Job creation and distribution. iWatch detects new checkpoints and creates slurm jobs. Each job executes a checkpoint loading script.

System Monitoring We are running an instance of Nigel's performance Monitor for Linux (nmon) [98] on every node to collect data such as network load and disk write rates. Furthermore, we use *netstat* to measure the amounts of packets sent as well as packet loss and buffer errors.

6.2 Evaluation Setup

Our setup consists of one producer and six consumer nodes. Table 6.1 lists the specifications of the individual nodes. The producer and consumer nodes are connected via a consumer Gigabit Ethernet switch that does not support IGMP snooping.

	CPU	RAM	Disk
Producer	Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz	64 GiB	256 GB SSD, 1024 GB SSD
VM Guest	Virtual single-core CPU	2 GiB	10 GB virtual HDD
Consumer 1	Intel(R) Xeon(R) CPU E31220 @ 3.10GHz	16 GiB	120 GB SSD, 1024 GB SSD
Consumer 2	Intel(R) Xeon(R) CPU E31220 @ 3.10GHz	16 GiB	120 GB SSD, 1024 GB SSD
Consumer 3	Intel(R) Xeon(R) CPU E31220 @ 3.10GHz	16 GiB	120 GB SSD, 1024 GB SSD
Consumer 4	Intel(R) Xeon(R) CPU E31220 @ 3.10GHz	16 GiB	120 GB SSD, 1024 GB SSD
Consumer 5	Intel(R) Xeon(R) CPU E31220 @ 3.10GHz	16 GiB	120 GB SSD, 1024 GB SSD
Consumer 6	Intel(R) Xeon(R) CPU E31220 @ 3.10GHz	16 GiB	120 GB SSD, 1024 GB SSD

Table 6.1: The server is connected to the consumers via a 1 GBit Ethernet connection.

We use Ubuntu 16.10 x64 as an operating system on the producer and consumer nodes with a custom Linux kernel of version 4.3.0. The custom Linux kernel is required by SimuBoost because of a modified KVM version which is required for the checkpointing mechanism.

6.2.1 Benchmarks

We use the the Phoronix *Timed Linux Kernel Compilation* (build-linux-kernel) [77] and SPECjbb©2015 (SPECjbb) [78] benchmarks. SPECjbb is a very memory intensive workload that increases the number of modified memory pages and leads to an overall increased size of checkpoint data as well as an increased simulation duration.

6.3 Distributed File System

Distributed storage is the first solution we are testing that uses a pushing mechanism. In this approach, the producer pushes the data across the simulation cluster. Every data transmission is sent to a single node which is selected based on file system criteria. We have selected Ceph FS as well as GlusterFS for our evaluation. When loading a checkpoint, a simulation node pulls any missing data from the other nodes in the cluster.

6.3.1 Ceph FS

Ceph FS is a popular distributed file system based on object storage as a foundation. In our Ceph FS approach, we have set up the metadata server and the monitor server on the producer node. The Ceph monitor provides health and status information for the cluster while the metadata server allows the execution of

basic POSIX file system commands such as ls, find, and cd. The Ceph OSD daemons are running on the consumers. These daemons interact with the underlying physical or logical storage units. We use a dedicated partition for every OSD daemon and one partition per consumer node. Our Ceph FS setup uses a configuration without authentication to reduce any unnecessary overhead. Figure 6.2 illustrates the setup.

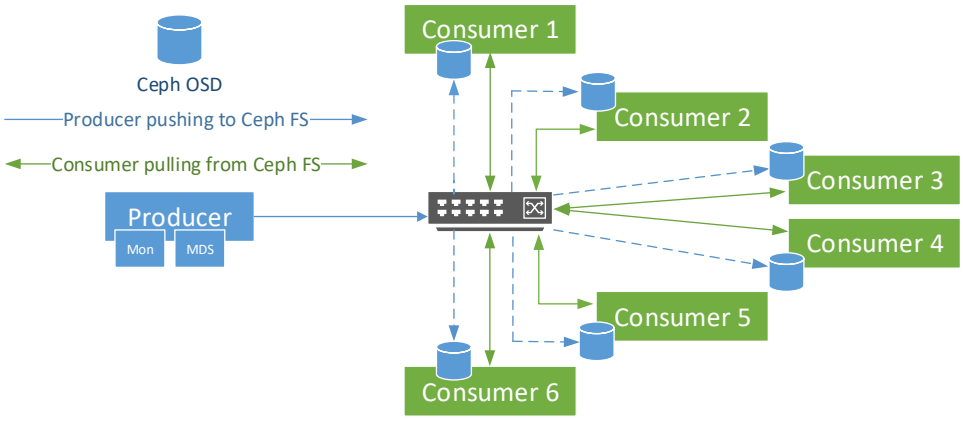

Figure 6.2: Ceph FS cluster setup. The producer distributes the checkpoint data between the Ceph OSDs. The consumers exchange the required data between each other when necessary.

We have configured a replication size of one, which means there will be no additional copy of the data [99], to reduce the amount of data that is transmitted. This is acceptable for us as we do not need to be able to compensate for the loss of consumer nodes as elaborated in Chapter 4. Furthermore, we have configured a Ceph FS for the actual checkpoint data with 512 placement groups and a Ceph FS for the metadata with 8 placement groups. These placement groups contain the actual checkpoint data in the form of data objects. We have calculated the optimal placement group values using the *Ceph PGs per Pool Calculator* [100] tool.

Additionally, we have modified the Ceph CRUSH map [101] for our OSD daemons to use a different algorithm for the data distribution. The CRUSH map determines how Ceph stores and retrieves data in the storage cluster. The default algorithm used for the distribution is the *Straw* algorithm. We have selected the *List* algorithm for our use case as it is the best solution for storage that does not shrink which fits our use case very well. Its major downside over the *Straw* algorithm is a suboptimal reorganization behavior in the case of data removal or re-weighting of data.

Tests

Unfortunately, our Ceph FS distributed storage solution has shown to be unfit for our scenario. Ceph FS suffers from high latency for checkpoint writes by the producer when we run the build-linux-kernel benchmark scenario. As a result, we occasionally have a high checkpoint writing latency on the producer of more than a minute for individual checkpoints. We have tried to use different distribution algorithms as well as increasing the replication size and placement group numbers without success. Figure 6.3 shows the checkpoint writing delays running the build-linux-kernel scenario [1] with two parallel simulations per consumer node and a checkpointing interval of 4 908 ms. In addition to the *List* algorithm, the figure also shows results using the default *Straw* algorithm to rule out any problems specific to the selected algorithm.

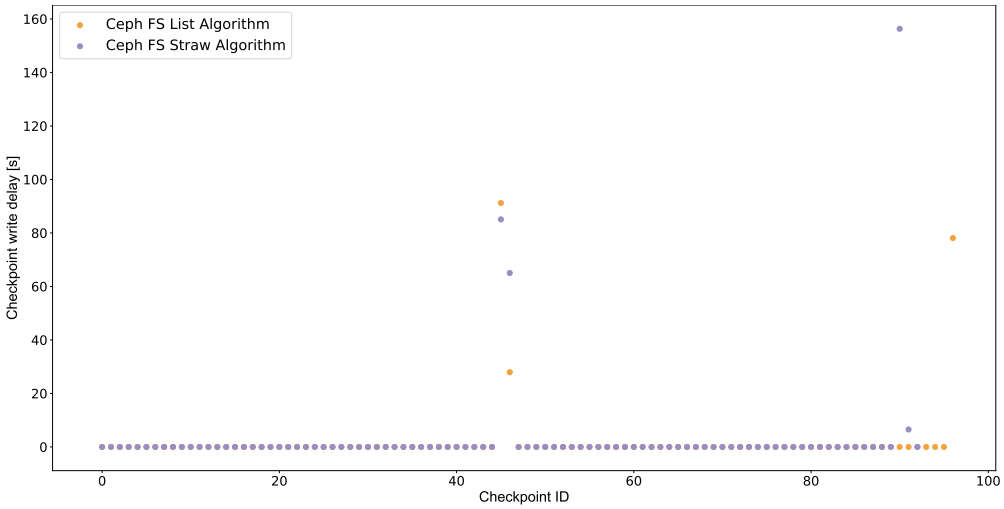

Figure 6.3: Checkpoint write delays using Ceph FS and the build-linux-kernel scenario. We can see three checkpoint writing delays of up to about 90 seconds.

We can see delays of up to about 90 seconds for a single checkpoint using the *List* algorithm as well as up to about 160 seconds using the *Straw* algorithm. These delays are not a general writing problem as we do not experience them when we are only writing checkpoints at the beginning of the simulation. The problem seems to come from simultaneous reads by the consumers as well as cluster re-balancing operations due to the data exchange between the consumers. Because of these delays, the producer only manages to create 97 checkpoints using *List* and 93 checkpoints using *Straw* instead of the about 160 checkpoints the direct distribution solution creates in the same time frame. The reduced amount

[1] $T_{vm} = 960s$, $s_{sim} = 31$, $s_{log} = 1$, $s_{cp} = 1\,024$, $N = 12$, $T_i = 3.7$

of checkpoints leads to less parallelization and therefore reduces the achievable speedup. The average checkpoint delay across all checkpoints is about 2 seconds.

6.3.2 GlusterFS

GlusterFS is another distributed file system. It works similar to Ceph FS with the main difference that it uses file storage instead of block storage. A GlusterFS volume consists of bricks which are the individual storage folders on the storage nodes (see Section 2.4.2). Instead of a single monitor and metadata server on the producer like in the Ceph FS setup, GlusterFS requires a GlusterFS daemon (GlusterD) on every node except the producer. The producer in this scenario only requires a GlusterFS client to write to the distributed volume. The GlustesrFS daemons are connected via peering to exchange volume information. We have configured GlusterFS to use a striping distribution. The striping distribution splits the files into chunks and distributes them across the cluster. This distribution is especially useful when working with large files. The default file-based distribution of GlusterFS would otherwise put each file on a single node. We have a flat file which contains about 90% of the overall data. This data would be allocated to a single node and therefore limit the data distribution and lead to a new network bottleneck. We have set the *stripe size*, which dictates over how many bricks the chunks of a file are distributed, to six. The reason we have chosen six is that we want to distribute the database file across all nodes because every node requires data from the database. We have tested a default chunk size of 128 KiB, as well as a chunk size of 2048 KiB. All of the *.ckpt* files in the build-linux-kernnel scenario we use to test the distribution are below 2048 KiB size. This means that using the chunk size of 2048 KiB, GlusterFS does not split up these files but only the database flat file which is about 3.7 GiB large. However, this also means that all *.ckpt* files land on the first consumer node while the flat file is distributed across all six consumer nodes. An optimal solution would probably use the file-based distribution for the *.ckpt* files and a striping-based distribution for the database file. Figure 6.4 illustrates the GlusterFS distribution setup.

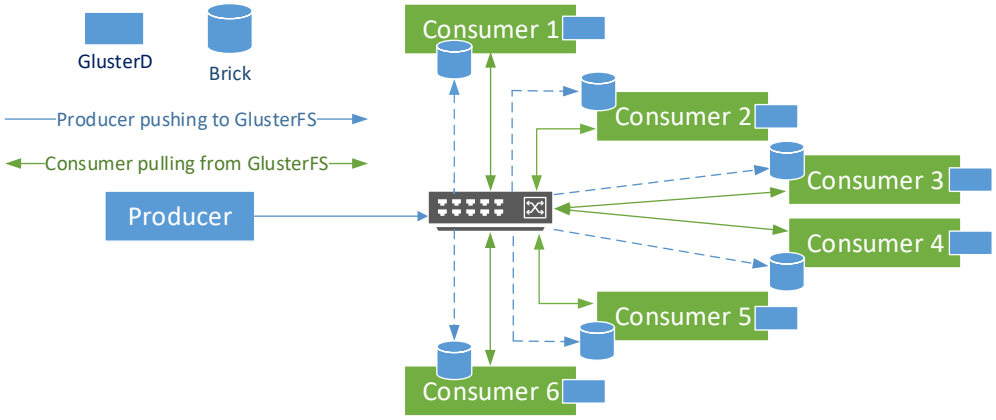

Figure 6.4: GlusterFS cluster setup. The producer distributes the checkpoint data across the GlusterFS volume which consists of individual bricks. The GlusterFS daemons handle the configuration and data distribution. The consumers exchange required data when necessary.

Tests

While we have not seen as high writing delay peaks as with Ceph FS, we had consistently delayed checkpoint writes of up to about 12 seconds as shown in Figure 6.5. Other than Ceph FS, the checkpoint writing delays of GlusterFS seem to be distributed across the entire runtime of the workload. We can see that overall, GlusterFS using 128 KiB chunks seems to perform worse, resulting in 70 created checkpoints instead of 97 like in the Ceph FS approach. The configuration using 2048 KiB chunks on the other hand seems to be slightly better with 113 created checkpoints. The average checkpoint delay across all checkpoints is about 2 seconds using 128 KiB chunks and 1.9 seconds using 2048 KiB chunks. The reason why GlusterFS using 128 KiB chunks has about the same average delay as Ceph FS, but a significantly lower number of checkpoint intervals is because the Ceph FS distribution has a high delay at the last checkpoint, while the GlusterFS delays are distributed evenly across the entire workload runtime.

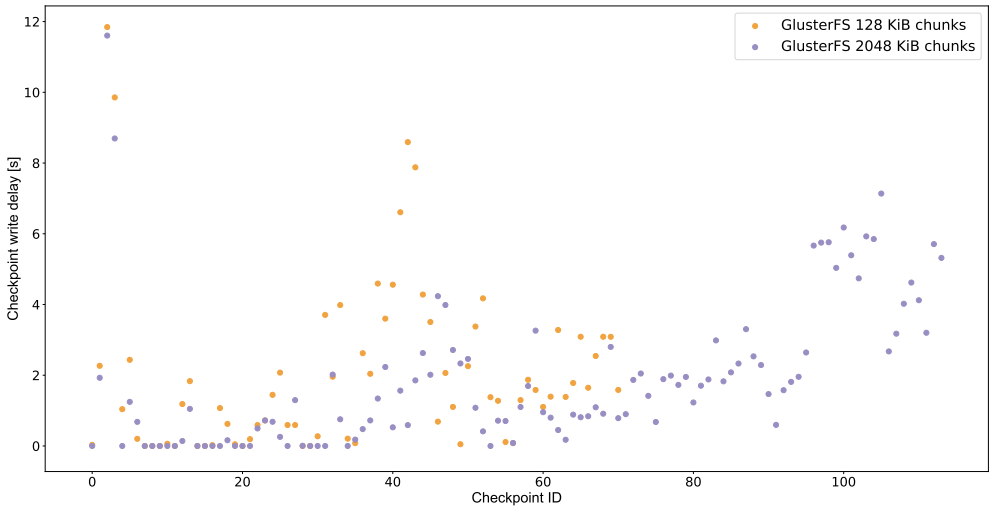

Figure 6.5: Checkpoint write delays using GlusterFS and the build-linux-kernel scenario. We can see three checkpoint writing delays of up to about 12 seconds with both 128 KiB chunks as well as 2048 KiB chunks. The 2048 KiB chunks do better in terms of overall delay and the amount of created checkpoints.

6.3.3 Conclusion

While we have not tried every possible configuration for Ceph FS as well as GlusterFS and don't want to rule out distributed storage in general, we have decided to not pursue this approach any further for this thesis and to focus on the multicast solution. The delays using both systems hint that this seems to be an issue related to the way distributed file systems work in general. It might be possible to increase GlusterFS performance using different cache and chunk sizes as well as setting up two separate volumes; one file-based distribution volume for the *.ckpt* files and one striping-based distribution volume for the database flat file. The applicability of this solution is very much dependent on the interval length of the checkpoints. Ceph FS also has several parameters which can be tweaked such as placement groups, cache sizes, and synchronization intervals. Furthermore, Ceph supports a distribution mechanism called erasure code which is similar to striping but requires an additional cache tier and is more complex to setup. With bigger intervals, write delay may have less impact and become negligible. However, we cannot consider this a general solution for SimuBoost.

6.4 Multicast

In this section, we will evaluate the results of the multicast approach and compare it to the Samba and direct distribution approaches. First, we test our methods for packet loss reduction as shown in Chapter 5.

6.4.1 Packet Loss Reduction Tests

We have tested multiple receive buffer values and rate limits using multicast transmission and the build-linux-kernel and SPECjbb scenarios. We have selected the values RX-Errors, RX-Drop, and Receive Buffer Errors from the netstat tool for our comparison. RX-Errors are errors that have happened during the transmission of data, these packets have been rejected by the network interface because they are malformed due to physical problems like broken cables, or intermediate devices that modify packets. RX-Drops are packets that have been dropped by the network interface because of buffer overflows. We do not change the interface buffers to not interfere with other applications on the system. Furthermore, we do not expect the interface buffers to play a big role in packet loss. Receive Buffer Errors are errors related to the size of the socket buffer. If more packets arrive than our multicast receive endpoint in the SimuBoost client can handle, packets start getting lost due to buffer overflows. In the build-linux-kernel scenario, all of our consumer nodes together receive about 21 million packets.

At first, we look at the packet loss using different receive buffer sizes and no token bucket rate limitation. Figure 6.6 shows a large amount of packet loss for both, the default 208 KiB buffer size as well as 2 MiB buffer size. The packet loss depicted is the sum of all consumer nodes.

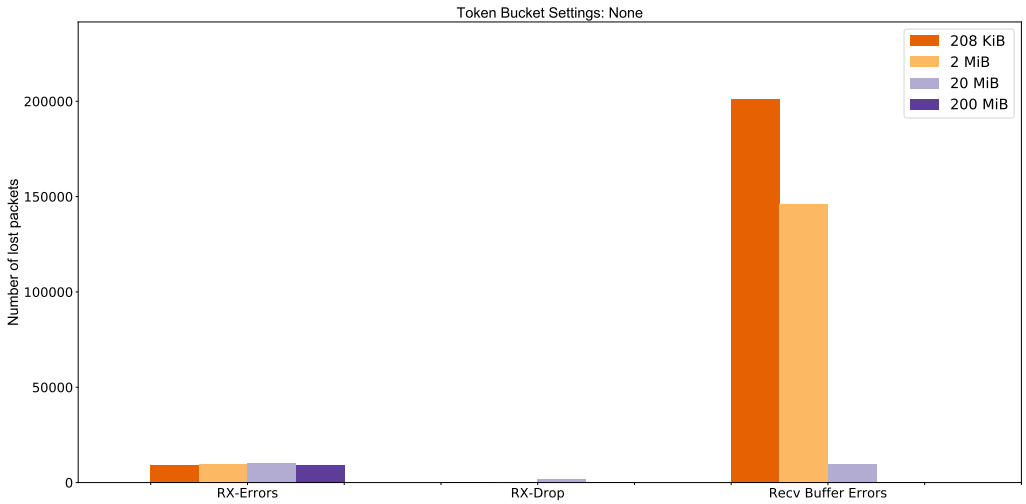

Figure 6.6: Receive errors using different receive buffer sizes without rate limiting. Using multicast transmission and the build-linux-kernel scenario.

As expected, the RX-Errors are not related to the receive buffer size because these errors occur on the transmission path. We can see that the packet loss using the default 208 KiB as well as using 2 MiB receive buffers are well above 100 000 packets. 100 000 lost packets results in about 133 MiB lost data assuming a data payload of 1400 bytes per packet. Increasing the receive buffer to 20 MiB has significantly reduced packet loss to less than 25 000, however, we can see a slight peak in RX-Drops. This result means that the buffer at the network interface experiences overflows. However, the number of dropped packets seems insignificant with about 1500 lost packets. Increasing the receive buffers further up to 200 MiB results in zero receive buffer errors and also zero RX-Drops.

Next, we look at the results using the token bucket rate limiting algorithm. Figure 6.7 shows the results of different receive buffer sizes using a rate limit of 125 MiB/s and 5 MiB bursts. 125 MiB/s is the maximum rate Gigabit Ethernet can handle, though the actual rate we achieve is lower at about 110 MiB/s due to overhead in our implementation. Note that the total number of packets is the same on all runs.

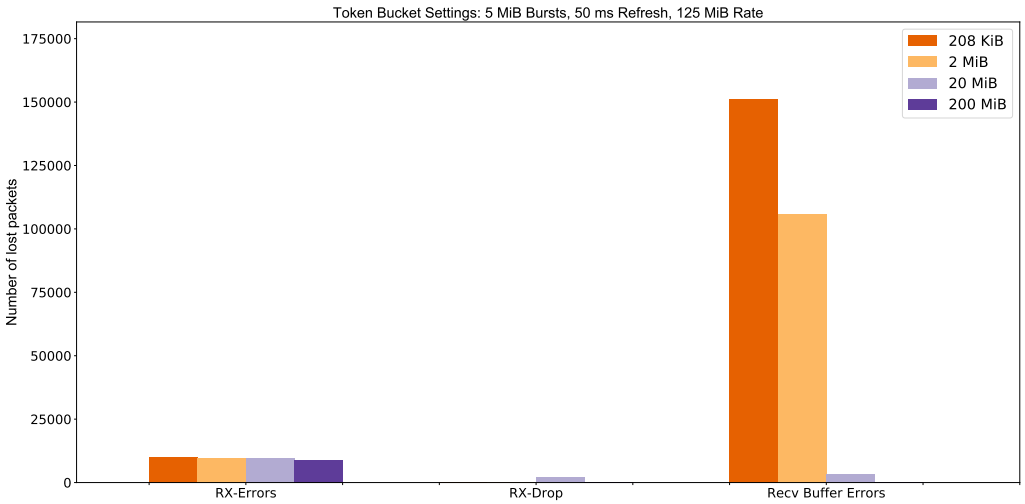

Figure 6.7: Receive errors using different receive buffer sizes and a transmission rate of 125 MiB/s and 5 MiB bursts. Using multicast transmission and the build-linux-kernel scenario.

As we can see, the rate-limiting has significantly reduced the packet loss using receive buffer sizes 208 KiB, 2 MiB and 20 MiB. The latter now experiences only about 3 000 losses due to receive buffer overflows. Furthermore, the receive buffer errors using the default buffer size of 208 KiB have been reduced from about 201 000 to about 150 000. The errors using a 2 MiB buffer have also been significantly reduced from about 145 000 to about 106 000 lost packets.

Figure 6.8 shows the results of different receive buffer sizes using a rate limit of 90 MiB/s and 5 MiB bursts. We have not tried to reduce the rate any lower to avoid blocking the producer. As we have seen in our analysis, the checkpoint write rate of the producer using the more write intensive SPECjbb benchmark stays below 100 MiB/s for the most part.

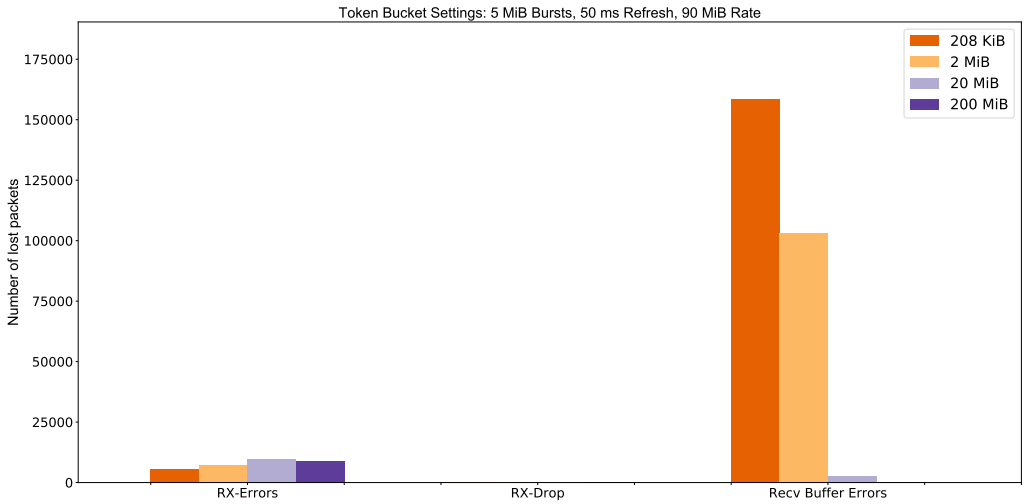

Figure 6.8: Receive errors using different receive buffer sizes and a transmission rate of 90 MiB/s and 5 MiB bursts. Using multicast transmission and the build-linux-kernel scenario.

As we can see in the figure, the build-linux-kernel scenario does not benefit from the lower data transmission rate. This is expected because the disk write rate using this scenario consistently stays below 55 MiB/s as shown in our analysis (see Chapter 3). The initial high transmission peaks of up to about 300 MiB/s are already reduced by the 125 MiB rate limit and the receive buffer size of 20 MiB seems to be enough the reduce the impact of the lower bursts.

Further on, we will evaluate the different rate limits using the 20 MiB receive buffer and the SPECjbb scenario, because our analysis has shown that the disk write rate of SPECjbb reaches up to 100 MiB/s with few bursts even higher than that. In the SPECjbb scenario, all of our consumer nodes together receive about 31 million packets. Due to a defect in the switch we have used for the previous tests, we had to install a new device. Therefore, a comparison between the amount of lost packets between these measurements is not valid. We do not require a comparison because the goal of the following tests is to find out whether the impact of the different rate limits differs for the SPECjbb benchmark.

Figure 6.9 shows a comparison between different rate limits and a receive buffer size of 20 MiB using the SPECjbb scenario. The figure shows that unlike the build-linux-kernel scenario, the SPECjbb scenario profits from the 90 MiB/s rate limit. This result reinforces our assumption that the rate limit benefits workloads with a higher average disk write rate while the receive buffer seems to work well for short bursts of high network load. The reasons for the significantly lower packet loss rates in general using this scenario are the new switch we have used,

as well as the about 100 MiB/s lower initial disk write burst of the SPECjbb scenario in the beginning of the workload, and the following two higher bursts of the build-linux-kernel scenario which are all at about 300 MiB/s (see Chapter 3).

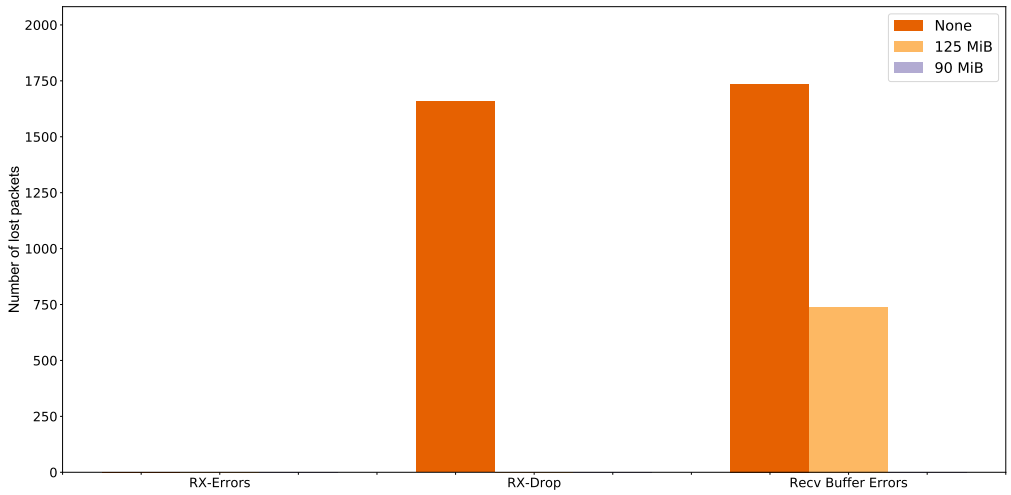

Figure 6.9: Comparison of receive errors using the SPECjbb scenario, a 20 MiB receive buffer, and different rate limits. The comparison shows that the SPECjbb scenario, unlike the build-linux-kernel scenario, benefits from these rate limits due to a higher average data rate.

6.4.2 Simulation Tests

Based on our packet loss reduction tests, we have set the receive buffers on the consumer nodes to 20 MiB and the rate limiting to 90 MiB/s with 5 MiB bursts. We will start with the build-linux-kernel scenario [2] we have already used for our analysis with two parallel simulations per consumer node and a checkpointing interval of 4 908 ms. As shown in Figure 6.10, the multicast solution is significantly faster than the direct distribution approach and very consistent. The average loading time of the direct distribution solution is about 120 seconds. The checkpoint loading using multicast on the other hand only requires about 5 seconds on average. This average loading time is about 24 times faster than the direct distribution solution. Assuming a simulation slowdown of 31 and 5 second intervals, the simulation of an interval would take 155 seconds. The average checkpoint loading time of 5 seconds using the multicast solution only increases the runtime of the interval by about 8 percent instead of the 77 percent increase using the direct distribution.

[2] $T_{vm} = 960s$, $s_{sim} = 31$, $s_{log} = 1$, $s_{cp} = 1\,024$, $N = 12$, $T_i = 3.7$

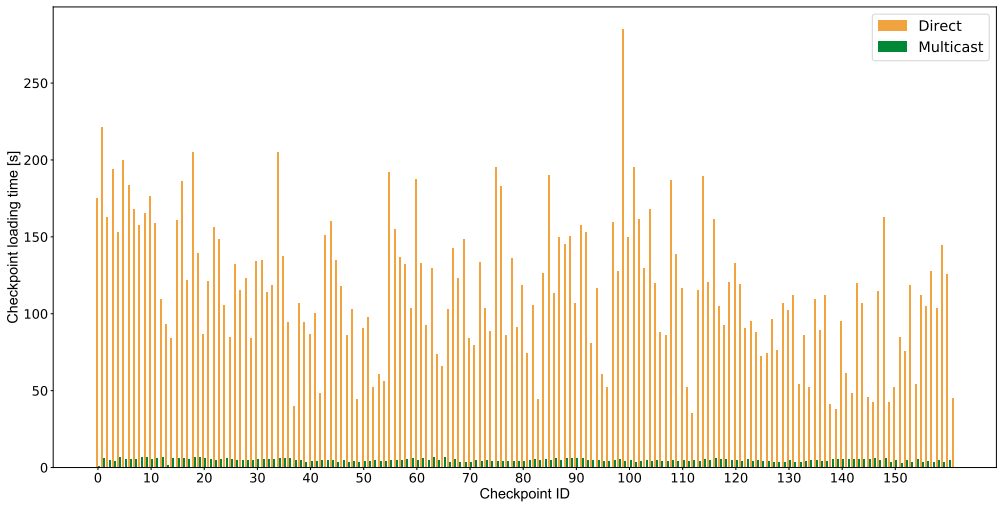

Figure 6.10: Comparison between direct distribution and multicast approaches. Build-linux-kernel scenario with six nodes, two jobs per node and a simulation interval of 4 908 ms.

To test shorter checkpointing intervals and an increased number of checkpoints, we have increased the number of parallel simulation jobs per consumer node from two to four and adjusted the model to get the optimal interval. Figure 6.11 shows the results of the build-linux-kernel scenario [3] using a checkpointing interval of 2 454 ms. The direct distribution approach experiences average checkpoint loading times of about 382 seconds. Multicast on the other hand still achieves stable checkpoint loading times of about 12 seconds. Reducing the checkpointing interval in half has roughly doubled the checkpoint loading times using the multicast approach. The reason for the overall increased checkpoint loading times is the increased parallelism on the consumer nodes, which leads to a decrease in available memory for the file system cache by 4 GiB. We can see that the direct distribution approach has created about five more checkpoints on average. This difference is not due to writing delays as we have experienced in the distributed file system approach. Therefore, we have not further examined them. We consider this deviation a result of our whole evaluation setup.

[3] $T_{vm} = 960s$, $s_{sim} = 31$, $s_{log} = 1$, $s_{cp} = 1\,024$, $N = 24$, $T_i = 3.7$

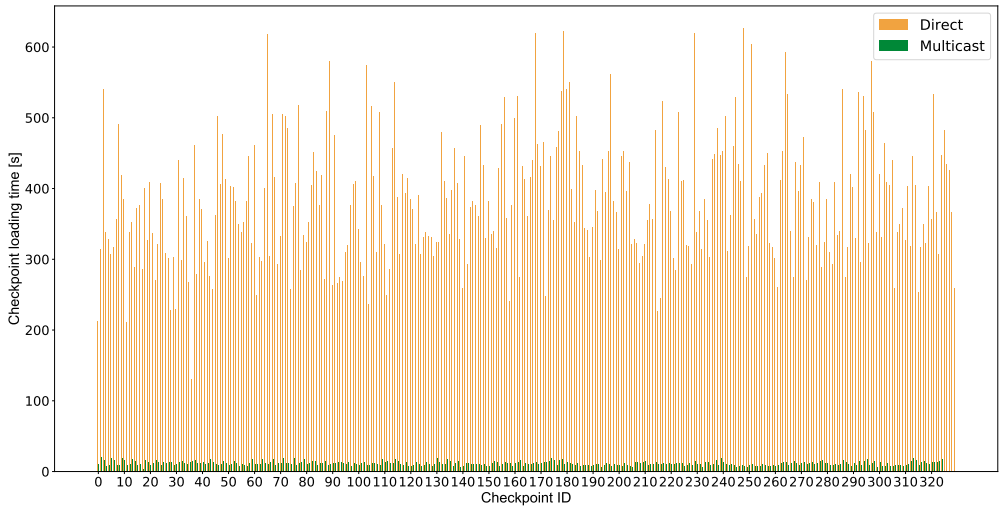

Figure 6.11: Comparison between direct distribution and multicast approaches. Build-linux-kernel scenario with six nodes and four jobs per node.

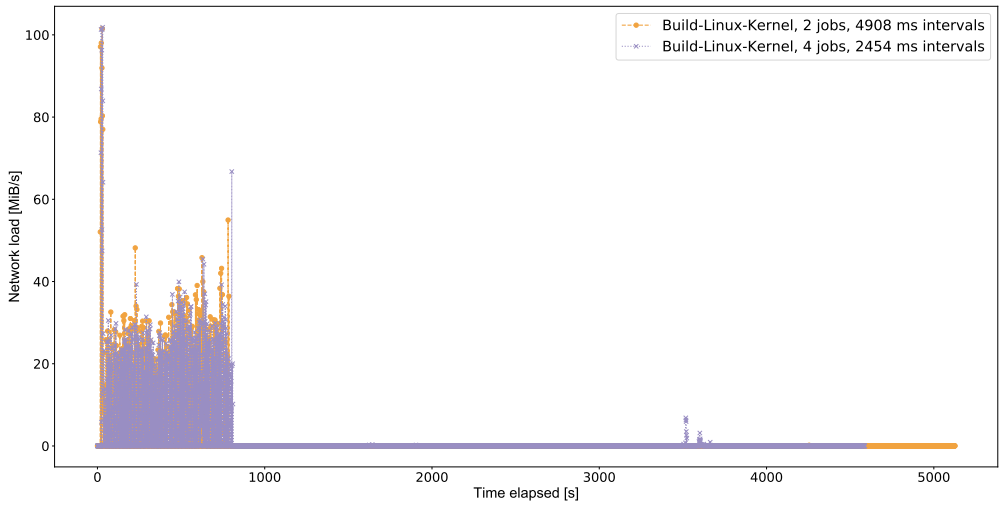

Figure 6.12: Network load of both build-linux-kernel scenarios. High initial peak due to initial vRAM image and data. Small peaks due to checkpoint repairs.

Figure 6.12 shows the network load of both build-linux-kernel scenarios. We can see an initial transmission peak due to the initial vRAM image and data being transmitted. Because of the pushing mechanism, we can continue to send the checkpoints until all checkpoint data has been transmitted. Furthermore, the figure shows data repair transmissions later during the simulation. This result reinforces our decision to use a custom multicast repair approach that only repairs data when

it is required. This way we utilize the idle network instead of retransmitting data while the network load is already increased. Additionally, we can see that the scenario with more parallel jobs and shorter intervals achieves a better speedup as it finishes faster. While it looks like both scenarios transmit a relatively equal amount of data, the scenario with four jobs per node transmits significantly more data with about 8.6 GiB instead of about 5.3 GiB with two jobs per node.

Next we look at our results using the SPECjbb benchmark scenario [4] with a checkpointing interval of 6 720 ms. SPECjbb is a Java benchmark that takes about twice as long as the build-linux-kernel to finish. It causes more changes to memory pages which results in a significantly higher data size. As shown in 3, the compressed overall checkpoint size of the SPECjbb is about 17 GiB while the combined checkpoints of build-linux-kernel only reach about 4 GiB when compressed. Figure 6.13 shows a similar result to the build-linux-kernel scenario.

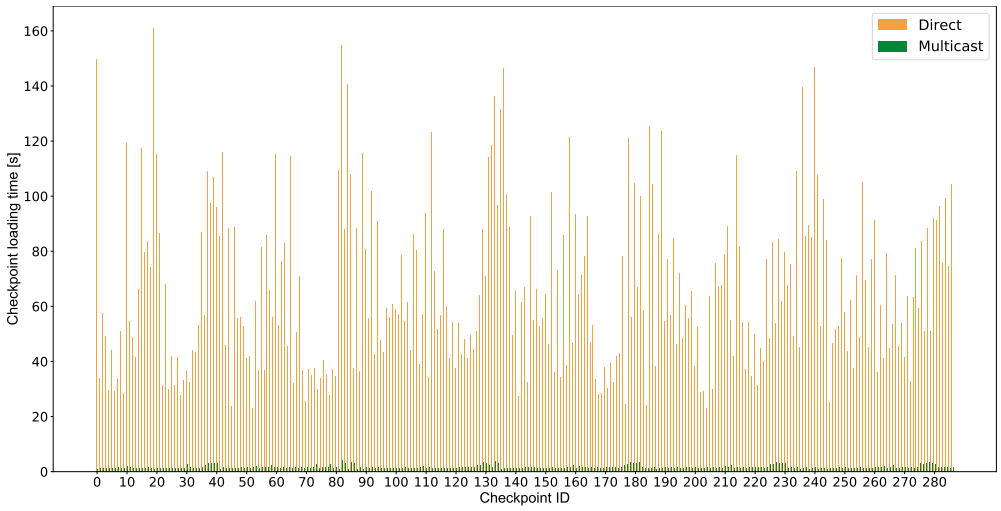

Figure 6.13: Comparison between direct distribution and multicast approaches using the SPECjbb scenario with six nodes and two jobs per node. This figure shows the impact of packet loss on both the direct distribution as well as the multicast approaches.

The average checkpoint loading time of the direct distribution is about 65 seconds. Like in the build-linux-kernel scenario, multicast performs much better with an average checkpoint loading time of 1.5 seconds which, assuming an interval length of about 7 seconds and a slowdown of 31, increases the interval runtime by less than 1 percent while the direct distribution has an impact of about

[4] $T_{vm} = 1800s$, $s_{sim} = 31$, $s_{log} = 1$, $s_{cp} = 1\,024$, $N = 12$, $T_i = 3.7$

30 percent. This result is even better than the results from the build-linux-kernel scenario. Furthermore, similar to the build-linux-kernel scenario, the checkpoint loading times remain consistent.

Figure 6.14 shows the SPECjbb scenario [5] using a checkpointing interval of 3 360 ms. We can see that multicast remains consistent with average checkpoint loading times of about 4 seconds. The average of the direct distribution approach reaches about 180 seconds. As with the build-linux-kernel tests, using four jobs per simulation node and half of the simulation interval has increased both the direct distribution as well as multicast checkpoint loading times.

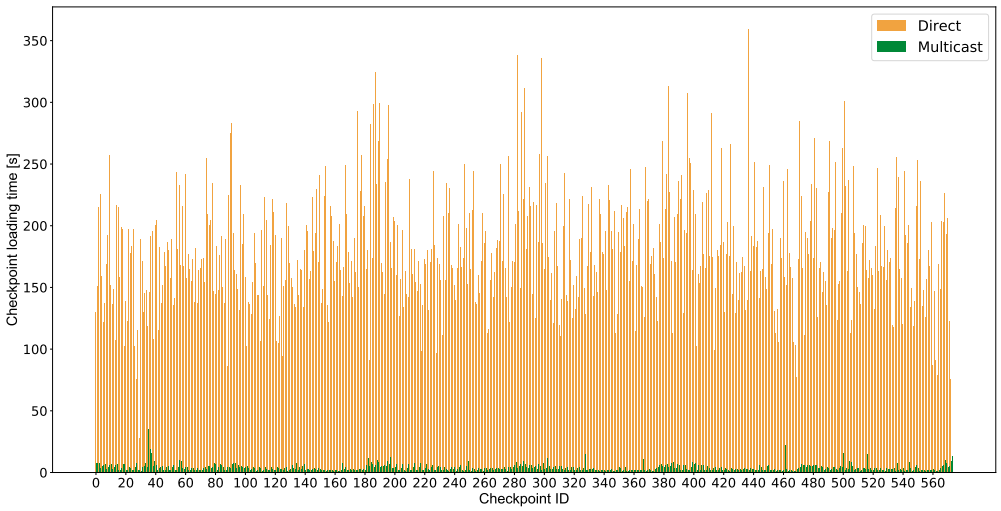

Figure 6.14: Comparison between direct distribution and multicast approaches. SPECjbb scenario with six nodes and four jobs per node. This figure shows stable loading with the exception of few peaks due to checkpoint repairs.

Looking at the network load using the SPECjbb benchmark in Figure 6.15 we can see more checkpoint repair transmissions compared to the build-linux-kernel scenario. Similarly to the build-linux-kernel though, the transmissions are distributed across the simulation runtime. As with the build-linux-kernel scenario, increasing the number of parallel jobs and decreasing the intervals has resulted in a better speedup which is shown by the shorter overall runtime. There is a significant difference in the amount data transmitted between the two scenarios. While the scenario with two jobs per node results in about 19.1 GiB of data, the scenario with four jobs per node increases the amount of data to about 31.8 GiB.

[5] $T_{vm} = 1800s$, $s_{sim} = 31$, $s_{log} = 1$, $s_{cp} = 1\,024$, $N = 24$, $T_i = 3.7$

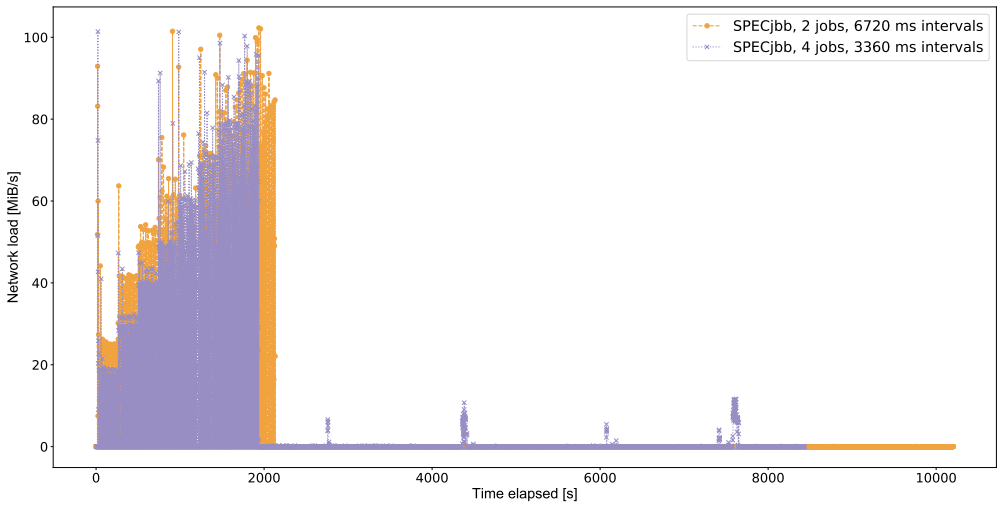

Figure 6.15: Network load of both SPECjbb scenarios. High initial peak due to initial vRAM image and data. Small peaks due to checkpoint repairs distributed evenly across the simulation runtime.

Overall simulation runtime Table 6.2 compares the overall simulation runtimes of the native execution without parallelization, as well as the direct distribution and multicast approaches. To estimate the native simulation time, we have multiplied the workload runtime T_{vm} with the slowdown factor between virtualization and functional simulation s_{sim}. We have used $T_{vm} = 16min$ for the build-linux-kernel scenario as well as $T_{vm} = 30min$ for the SPECjbb scenario. The slowdown factor in our calculations is $s_{sim} = 31$. These are the same values we have used to calculate the interval length for our scenarios.

	build-linux-kernel 2 jobs	build-linux-kernel 4 jobs	SPECjbb 2 jobs	SPECjbb 4 jobs
Native	496 min	496 min	930 min	930 min
Direct	103 min	156 min	185 min	198 min
Multicast	79 min	69 min	162 min	135 min

Table 6.2: Comparison of overall simulation runtimes using the direct distribution and multicast approaches. Multicast scales well which leads to a reduction of simulation time with an increased parallelism. The direct distribution approach does not scale very well and leads to an increased simulation runtime when performing more parallel simulations.

We can see that the simulation using SimuBoost is significantly faster than the estimated native execution by a factor of at least 6 for multicast and 3 for direct

distribution. The SPECjbb scenario profits more from the parallelization of the simulation than the build-linux-kernel scenario. Furthermore, this figure shows that the direct distribution approach does not scale well. In both the build-linux-kernel and the SPECjbb benchmarks, increasing the number of parallel jobs in the direct distribution increases the overall simulation runtime. Using multicast, on the other hand, an increase in the number of parallel jobs reduces the overall simulation runtime. This result means that the direct distribution leads to a slowdown while the multicast solution profits from more jobs and increases the speedup.

6.5 Discussion

We have discarded the distributed storage solution due to checkpoint writing delays on the producer. There are several factors which might have an impact on this delay such as file distribution calculations, data rebalancing, and file locks. We have only tested basic configurations, and both Ceph FS and GlusterFS provide many options which can be tweaked to achieve different results. However, we have decided to focus on multicast as it seems to be a more promising solution. For example, the network load using multicast is more predictable with the exception of repair transmissions.

The multicast approach has achieved stable checkpoint loading times. Unlike the Samba solution shown in Chapter 3, multicast has shown no significant increases in loading times over time. Furthermore, we have experienced significantly lower average checkpoint loading times of a factor of up to about 43 compared to the direct distribution approach. Using an increased receive buffer and a rate limiting algorithm, we were able to reduce the amount of packet loss. The few losses we have experienced did not significantly impact the average checkpoint loading times. However, individual checkpoints such as shown in Figure 6.14 have experienced increased loading times of a factor of about 13. The packet loss is dependent on several factors such as the network hardware, operating system buffers, transmission rates, overall network load and the efficiency of the multicast implementation. We provide two ways of regulating packet loss by providing configuration options for receive buffer sizes and rate limiting. Due to the many factors involved, the limited amount of hardware we had, and a limited amount of workloads we were able to run, we can only provide limited recommendations. For our tests, a rate limit of about 95 MiB/s and a 20 MiB receive buffer on the consumer nodes have provided good results in terms of packet loss. The overall network load is close to the disk write rate and does not exhaust a Gigabit Ethernet connection. By using a custom packet loss handling mechanism, we were able to distribute retransmissions over the course of the entire simulation runtime. A good example for this distribution is Figure 6.15, which shows checkpoint repair

transmissions distributed across the runtime of the entire simulation and especially after the higher initial network load during which the data loss occurred.

Finally, we have compared the overall simulation runtimes using an estimate for a native, non-parallel simulation as well as measured runtimes of the direct distribution and multicast approaches. It is important to note that we do not use deterministic replay (see Section 2.2) for our tests, instead we sleep a pre-calculated amount of time (see Section 6.1), which means the values for the direct distribution and multicast also provide only an estimate. We have shown that the direct distribution approach does not scale well and leads to a slowdown of the simulation when increasing the number of parallel jobs and respectively shortening the checkpointing intervals. The multicast approach on the other hand has achieved a higher speedup using the increased parallelism. This shows that we have reached the goal of increasing the scalability of the checkpoint distribution.

6.6 Conclusion

As we have shown in this chapter, the distributed storage solution suffers from checkpoint writing delays on the producer node. Both, the Ceph FS, and the GlusterFS suffer from these delays although in different manifestations. We leave a further optimization of these approaches for future work. The multicast solution provides fast checkpoint loading times which are close to or even below the checkpoint interval. We achieve these loading times by utilizing the checkpoint compression, reducing the network, and sending the data continuously. The key factor of this distribution is sending the data of each incremental checkpoint only once as soon as it has been created using multicast transmissions. The repair transmissions we have to make because of the packet loss are rare, small, and distributed across the runtime of the whole simulation. Therefore they do not have a significant impact on the average checkpoint loading times. We have shown that we can meet the goal of providing a fast checkpoint distribution over a Gigabit Ethernet connection without a network bottleneck. Furthermore, we have shown that our solution scales well with an increased amount of parallelism and a reduced checkpoint interval.

Chapter 7

Conclusion

SimuBoost [1] utilizes parallelization to achieve fast full system simulation. For this goal, SimuBoost runs the workload of interest in a hardware-assisted virtual machine (VM) and periodically creates incremental checkpoints of the VM. The SimuBoost server distributes these checkpoints across a cluster of simulation nodes to bootstrap parallel simulations of the intervals. The execution speed difference between hardware-assisted virtualization and functional simulation drives a parallelization of the simulation. The speedup that can be achieved by SimuBoost is dependent on the number of parallel simulations and the checkpoint interval length. The checkpoint distribution is an important factor as it delays the actual start of the parallel simulations. As a result, the complete simulation takes longer, and the achievable speedup is reduced. Therefore, the checkpoint distribution needs to be reliable and as fast as possible to reduce any overhead to a minimum. In a direct distribution approach by Eicher [4], the simulation nodes pull all necessary data for the current simulation interval from SimuBoost. Eicher has shown that using a small number of consumers and a Gigabit Ethernet network connection saturates the network and leads to a network bottleneck.

We have reproduced the results of Eicher, showing that the direct distribution approach exhausts a Gigabit Ethernet network and results in slow loading times and reduced speedup and scalability. By using a network attached storage, we were able to utilize the data compression of SimuBoost. This approach allowed us to achieve significantly better results. However, due to the pulling mechanism, this approach still causes high network load as well as an uneven distribution of network load. Additionally, the connection between the virtualization host and the simulation nodes remains a bottleneck. Therefore, we have evaluated solutions of pushing data to the simulation nodes to achieve a more even distribution of network load. Our first solution, distributed file systems, has resulted in high write delays on the virtualization host and therefore was not a viable option. We were able to achieve good results using multicast transmission. By pushing all

available data to every node using single multicast transmissions as well as a custom method to detect and repair packet loss, we were able to reduce the impact of the checkpoint distribution on the runtime of the individual interval simulations. Our solution increases the individual checkpoint simulation runtime by less than 10 percent, while the direct distribution increases the runtime by up to about 77 percent.

7.1 Future Work

We have evaluated basic configurations for the distributed file systems Ceph FS and GlusterFS. Our tests have shown checkpoint writing delays on the virtualization host which are unacceptable for our solution. Optimizations of these distributed file systems might yield better results.

Future work may also investigate the use of jumbo frames to enhance the network throughput. Jumbo frames increase the available maximum transmission unit of Ethernet networks from 1500 bytes up to 9000 bytes. Jin et al. [102] have shown that the throughput in Myrinet, a gigabit-per-second local area network, increases significantly with a larger MTU size. Bencivenni et al. [103] have achieved throughput improvements using UDP and jumbo frames in a 10 Gigabit Ethernet network. SimuBoost, however, only benefits from higher throughputs when disabling the rate limiting algorithm we have implemented. Further investigation is required to find out whether SimuBoost would be able to benefit from higher throughputs.

Bibliography

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

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

[3] 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.

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

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

[6] Inc. Free Software Foundation. Gcc, the gnu compiler collection. `https://gcc.gnu.org/`. Accessed: 27.09.2017.

[7] Inc. Free Software Foundation. Abi policy and guidelines. `https://gcc.gnu.org/onlinedocs/libstdc++/manual/abi.html`. Accessed: 27.09.2017.

[8] Linux Foundation. Itanium c++ abi (revision: 1.83). `http://refspecs.linuxbase.org/cxxabi-1.83.html`. Accessed: 27.09.2017.

[9] Microsoft. Windows api index. `https://msdn.microsoft.com/en-us/library/windows/desktop/ff818516(v=vs.85).aspx`. Accessed: 20.10.2017.

[10] Bob Amstadt and Michael K Johnson. Wine. *Linux Journal*, 1994(4es):3, 1994.

[11] WineHQ. Winehq wiki. `https://wiki.winehq.org/`. Accessed: 20.10.2017.

[12] Jeffrey Racine. The cygwin tools: a gnu toolkit for windows, 2000.

[13] Cygwin authors. Cygwin documentation. `https://cygwin.com/docs.html`. Accessed: 20.10.2017.

[14] Bill Venners. *The Java Virtual Machine*. McGraw-Hill, New York, 1998.

[15] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java virtual machine specification*. Pearson Education, 2014.

[16] Erik Meijer and John Gough. A technical overview of the common language infrastructure, 2001.

[17] James S Miller and Susann Ragsdale. *The common language infrastructure annotated standard*. Addison-Wesley Professional, 2004.

[18] Mendel Rosenblum and Tal Garfinkel. Virtual machine monitors: Current technology and future trends. *Computer*, 38(5):39–47, 2005.

[19] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003.

[20] Anthony Velte and Toby Velte. *Microsoft virtualization with Hyper-V*. McGraw-Hill, Inc., 2009.

[21] John Kelbley, Mike Sterling, and Allen Stewart. *Windows Server 2008 Hyper-V: Insiders Guide to Microsoft's Hypervisor*. John Wiley & Sons, 2011.

[22] Oracle. Virtualbox documentation. `https://www.virtualbox.org/wiki/Documentation`. Accessed: 20.10.2017.

[23] Robert Warnke Thomas Ritzau. Qemu, kernel-based virtual machine (kvm), xen + libvirt. `http://qemu-buch.de/de/index.php?title=QEMU-KVM-Book`. Accessed: 20.10.2017.

[24] Yasunori Goto. Kernel-based virtual machine technology. *Fujitsu Scientific and Technical Journal*, 47(3):362–368, 2011.

[25] Haikun Liu, Hai Jin, Xiaofei Liao, Liting Hu, and Chen Yu. Live migration of virtual machine based on full system trace and replay. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 101–110. ACM, 2009.

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

[27] Daniel J Scales, Min Xu, Matthew D Ginzton, et al. Low overhead fault tolerance through hybrid checkpointing and replay, July 30 2013. US Patent 8,499,297.

[28] Arun Babu Nagarajan, Frank Mueller, Christian Engelmann, and Stephen L Scott. Proactive fault tolerance for hpc with xen virtualization. In *Proceedings of the 21st annual international conference on Supercomputing*, pages 23–32. ACM, 2007.

[29] Samuel T King, George W Dunlap, and Peter M Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 1–1, 2005.

[30] Wim De Pauw and Donald P Pazel. Method and apparatus for non-deterministic incremental program replay using checkpoints and syndrome tracking, August 21 2006. US Patent App. 11/507,166.

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

[32] Saurabh Agarwal, Rahul Garg, Meeta S Gupta, and Jose E Moreira. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 277–286. ACM, 2004.

[33] John Mehnert-Spahn, Eugen Feller, and Michael Schoettner. Incremental checkpointing for grids. In *Linux Symposium*, volume 120, 2009.

[34] Francisco Javier Thayer Fábrega, Francisco Javier, and Joshua D Guttman. Copy on write, 1995.

[35] Keith Loepere. Osf mach final draft kernel principles. *Open Software Foundation and Carnegie Mellon University*, 1993.

[36] Elmootazbellah Nabil Elnozahy, David B Johnson, and Willy Zwaenepoel. The performance of consistent checkpointing. In *Reliable Distributed Systems, 1992. Proceedings., 11th Symposium on*, pages 39–47. IEEE, 1992.

[37] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *ACM SIGOPS Operating Systems Review*, 40(5):2–13, 2006.

[38] Leendert Van Doorn. Hardware virtualization trends. In *ACM/Usenix International Conference On Virtual Execution Environments: Proceedings of the 2 nd international conference on Virtual execution environments*, volume 14, pages 45–45, 2006.

[39] R Hiremane. Intel virtualization technology for directed i/o (intel vt-d). *Technology@ Intel Magazine*, 4(10), 2007.

[40] R Lantz. Fast functional simulation with parallel embra. In *Proceedings of the 4th Annual Workshop on Modeling, Benchmarking and Simulation*, 2008.

[41] Min Xu, Vyacheslav Malyugin, Jeffrey Sheldon, Ganesh Venkitachalam, Boris Weissman, et al. Retrace: Collecting execution trace with virtual machine deterministic replay. In *In Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation, MoBS*. Citeseer, 2007.

[42] Lok-Kwong Yan, Manjukumar Jayachandra, Mu Zhang, and Heng Yin. V2e: combining hardware virtualization and softwareemulation for transparent and extensible malware analysis. *ACM Sigplan Notices*, 47(7):227–238, 2012.

[43] 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 UsersâĂŹ Forum*, pages 29–30, 2011.

[44] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[45] Marc Rittinghaus, Thorsten Groeninger, and Frank Bellosa. Simutrace: A toolkit for full system memory tracing. White paper, Karlsruhe Institute of Technology (KIT), Operating Systems Group, May 2015.

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

[47] Nagapramod Mandagere, Pin Zhou, Mark A Smith, and Sandeep Uttamchandani. Demystifying data deduplication. In *Proceedings of the ACM/IFIP/USENIX Middleware'08 Conference Companion*, pages 12–17. ACM, 2008.

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

[49] L Peter Deutsch. Gzip file format specification version 4.3. 1996.

[50] Yann Collet. Lz4 explained. `https://fastcompression.blogspot.de/2011/05/lz4-explained.html`. Accessed: 10.03.2017.

[51] Yann Collet. Lz4 frame format : Final specifications. `https://fastcompression.blogspot.de/2013/04/lz4-streaming-format-final.html`. Accessed: 10.03.2017.

[52] Yann Collet. Inter-block compression. `https://fastcompression.blogspot.de/2013/08/inter-block-compression.html`. Accessed: 10.03.2017.

[53] Randy H Katz. Network-attached storage systems. In *Scalable High Performance Computing Conference, 1992. SHPCC-92, Proceedings.*, pages 68–75. IEEE, 1992.

[54] Jay Ts, Robert Eckstein, and David Collier-Brown. *Samba*. " O'Reilly Media, Inc.", 2003.

[55] Paul J Leach and Dilip Naik. A common internet file system (cifs/1.0) protocol. Technical report, Internet-Draft, IETF, 1997.

[56] Christopher R Hertel. *Implementing CIFS: The Common Internet File System*. Prentice Hall Professional, 2004.

[57] Yaniv Pessach. *Distributed Storage: Concepts, Algorithms, and Implementations*. 2013.

[58] TC Jepson. The basics of reliable distributed storage networks. *IT professional*, 6(3):18–24, 2004.

[59] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[60] Michael Factor, Kalman Meth, Dalit Naor, Ohad Rodeh, and Julian Satran. Object storage: The future building block for storage systems. In *Local to Global Data Interoperability-Challenges and Technologies, 2005*, pages 119–123. IEEE, 2005.

[61] Inc. Inktank Storage and contributors. Ceph documentation - ceph architecture. `http://docs.ceph.com/docs/jewel/architecture/`. Accessed: 24.04.2017.

[62] Inc. Inktank Storage and contributors. Ceph documentation - ceph filesystem. `http://docs.ceph.com/docs/jewel/cephfs/`. Accessed: 24.04.2017.

[63] Inc. Inktank Storage and contributors. Ceph documentation - monitor config reference. `http://docs.ceph.com/docs/master/rados/configuration/mon-config-ref/`. Accessed: 24.04.2017.

[64] Inc. Inktank Storage and contributors. Ceph documentation - ceph block device. `http://docs.ceph.com/docs/jewel/rbd/rbd/`. Accessed: 24.04.2017.

[65] Inc. Inktank Storage and contributors. Ceph documentation - ceph object gateway. `http://docs.ceph.com/docs/jewel/radosgw/`. Accessed: 24.04.2017.

[66] Inc. Inktank Storage and contributors. Ceph documentation - placement groups. `http://docs.ceph.com/docs/master/rados/operations/placement-groups/`. Accessed: 24.04.2017.

[67] Sage A Weil, Scott A Brandt, Ethan L Miller, and Carlos Maltzahn. Crush: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 122. ACM, 2006.

[68] Inc. Inktank Storage and contributors. Crush maps. `http://docs.ceph.com/docs/master/rados/operations/crush-map/`. Accessed: 26.04.2017.

[69] Eric B Boyer, Matthew C Broomfield, and Terrell A Perrotti. Glusterfs one storage server to rule them all. Technical report, Los Alamos National Laboratory (LANL), 2012.

[70] Inc Red Hat. Glusterfs documentation. `https://gluster.readthedocs.io/en/latest/`. Accessed: 26.04.2017.

[71] Inc Red Hat. Glusterfs documentation: Architecture. `https://gluster.readthedocs.io/en/latest/Quick-Start-Guide/Architecture/`. Accessed: 26.04.2017.

[72] Nikolaus Rath. Linux fuse (filesystem in userspace). `https://github.com/libfuse/libfuse`. Accessed: 20.10.2017.

[73] Ariel J Frank, Larry D Wittie, and Arthur J Bernstein. Multicast communication on network computers. *IEEE software*, 2(3):49, 1985.

[74] S Deering. Host extensions for ip multicasting. *RFC1112*, 1997.

[75] Jon Postel. User datagram protocol. *RFC768*, 1980.

[76] Shepherd, Greg and Fairhurst, Gorry and Eggert, Lars. UDP Usage Guidelines. RFC 8085, RFC Editor, March 2017.

[77] Phoronix Media. Timed linux kernel compilation [pts/build-linux-kernel]. `https://openbenchmarking.org/test/pts/build-linux-kernel`. Accessed: 29.06.2017.

[78] Standard Performance Evaluation Corporation. Standard performance evaluation corporation - jbb2015. `https://www.spec.org/jbb2015/`. Accessed: 29.06.2017.

[79] John H Howard, Michael L Kazar, Sherri G Menees, David A Nichols, Mahadev Satyanarayanan, Robert N Sidebotham, and Michael J West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):51–81, 1988.

[80] Mary G Baker, John H Hartman, Michael D Kupfer, Ken W Shirriff, and John K Ousterhout. Measurements of a distributed file system. In *ACM SIGOPS Operating Systems Review*, volume 25, pages 198–212. ACM, 1991.

[81] Matthias Schmidt, Niels Fallenbeck, Matthew Smith, and Bernd Freisleben. Efficient distribution of virtual machines for cloud computing. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 567–574. IEEE, 2010.

[82] Giacinto Donvito, Giovanni Marzulli, and Domenico Diacono. Testing of several distributed file-systems (hdfs, ceph and glusterfs) for supporting the hep experiments analysis. In *Journal of Physics: Conference Series*, volume 513, page 042014. IOP Publishing, 2014.

[83] Dhruba Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 11(2007):21, 2007.

[84] Alex Davies and Alessandro Orsaria. Scale out with glusterfs. *Linux J.*, 2013(235), November 2013.

[85] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking (TON)*, 5(6):784–803, 1997.

[86] Sanjoy Paul, Krishan K. Sabnani, JC-H Lin, and Supratik Bhattacharyya. Reliable multicast transport protocol (rmtp). *IEEE Journal on Selected Areas in Communications*, 15(3):407–421, 1997.

[87] Hugh W Holbrook, Sandeep K Singhal, and David R Cheriton. Log-based receiver-reliable multicast for distributed interactive simulation. *ACM SIGCOMM Computer Communication Review*, 25(4):328–341, 1995.

[88] Alex Koifman and Stephen Zabele. Ramp: A reliable adaptive multicast protocol. In *INFOCOM'96. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation. Proceedings IEEE*, volume 3, pages 1442–1451. IEEE, 1996.

[89] B Adamson, C Bormann, M Handley, and J Macker. Negative-acknowledgment (nack)-oriented reliable multicast (norm) protocol. Technical report, 2004.

[90] Jo-Mei Chang and Nicholas F. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems (TOCS)*, 2(3):251–273, 1984.

[91] Brian Whetten, Todd Montgomery, and Simon Kaplan. A high performance totally ordered multicast protocol. *Theory and Practice in Distributed Systems*, pages 33–57, 1995.

[92] Rajendra Yavatkar, James Griffoen, and Madhu Sudan. A reliable dissemination protocol for interactive collaborative applications. In *Proceedings of the third ACM international conference on Multimedia*, pages 333–344. ACM, 1995.

[93] Ryan G Lane, Scott Daniels, and Xin Yuan. An empirical study of reliable multicast protocols over ethernet-connected networks. *Performance Evaluation*, 64(3):210–228, 2007.

[94] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks*. Prentice Hall Press, Upper Saddle River, NJ, USA, 5th edition, 2010.

[95] Joao Eriberto Mota Filho Michael Prokop. Ubuntu manuals - iwatch. `http://manpages.ubuntu.com/manpages/zesty/man1/iwatch.1.html`. Accessed: 29.06.2017.

[96] Heinrich Schuchardt Michael Kerrisk. Linux programmer's manual: Inotify. `http://man7.org/linux/man-pages/man7/inotify.7.html`. Accessed: 20.10.2017.

[97] Andy Yoo, Morris Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Job scheduling strategies for parallel processing*, pages 44–60. Springer, 2003.

[98] Nigel Griffiths. Nmon for linux. `http://nmon.sourceforge.net/`. Accessed: 20.10.2017.

[99] Inc. Inktank Storage and contributors. Ceph documentation - pools. `http://docs.ceph.com/docs/jewel/rados/operations/pools/`. Accessed: 20.10.2017.

[100] Inc. Inktank Storage and contributors. Ceph pgs per pool calculator. `http://ceph.com/pgcalc/`. Accessed: 26.04.2017.

[101] Inc. Inktank Storage and contributors. Manually editing a crush map. `http://docs.ceph.com/docs/master/rados/operations/crush-map-edits/`. Accessed: 26.04.2017.

[102] Hyun-Wook Jin, Chuck Yoo, and Sung-Kyun Park. Stepwise optimizations of udp/ip on a gigabit network. In *European Conference on Parallel Processing*, pages 745–748. Springer, 2002.

[103] Marco Bencivenni, Angelo Carbone, Armando Fella, Domenico Galli, Umberto Marconi, Gianluca Peco, Stefano Perazzini, Vincenzo Vagnoni, and Stefano Zani. High rate packet transmission on 10 gbit/s ethernet lan using commodity hardware. In *Real Time Conference, 2009. RT'09. 16th IEEE-NPSS*, pages 167–182. IEEE, 2009.