Karlsruher Institut für Technologie

SimuBoost: Scalable Parallelization of Functional System Simulation

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Dipl.-Inform. Marc Rittinghaus

aus Iserlohn

Tag der mündlichen Prüfung: 19.07.2019

Hauptreferent:	Prof. Dr. Frank Bellosa Karlsruher Institut für Technologie
Korreferent:	Prof. Dr. Hans P. Reiser Universität Passau

Abstract

Gathering detailed run-time information such as memory access traces in operating system and security research often involves functional full system simulation (FFSS). The simulator runs the workload of interest in a virtual machine (VM), gradually interpreting or translating instructions so that they operate on the state of the VM and allow for comprehensive instrumentation.

While functional full system simulation is a powerful tool, a severe limitation is its immense slowdown. For QEMU, we have measured average slowdowns of 30x and 60x for plain simulation and tracing of memory accesses, respectively. Simulators offering more advanced instrumentation capabilities can even be an order of magnitude slower. This quickly renders functional simulation impractical for long-running, networked, or interactive workloads. Furthermore, the slowdown creates unrealistic timing behavior whenever activities external to the virtual machine (e.g., I/O) are involved.

In this thesis, we present **SimuBoost**, a method for drastically accelerating functional full system simulation. SimuBoost runs the workload in a fast and interactive hardware-assisted virtual machine while periodically taking checkpoints. These checkpoints then serve as starting points for simulations, enabling to simulate and analyze each interval simultaneously in one job per interval. Heterogeneous deterministic replay guarantees that the simulations repeat the exact same execution as in the hardware-assisted run, including interactions and recorded timing.

Our prototype is able to significantly reduce the run time of functional full system simulation while providing full interactivity. Simulating an entire kernel build completes in just 16% more time than needed to run the same workload in a regular hardware-assisted virtual machine. SimuBoost is able to maintain this performance even with full instrumentation for memory tracing.

This thesis represents the first project to apply the concept of partitioning and parallelization of execution time to interactive full system virtualization in a manner that allows for immediate parallel functional simulation. We complement

the practical implementation with a performance model to formally describe the properties of the acceleration method and predict speedups. In contrast to previous work, SimuBoost places a strong focus on scalability beyond the limits of a single physical machine. It therefore makes heavy use of virtual machine checkpointing technology. In this course, we present two novel methods for efficiently and effectively reducing the size of periodic checkpoints.

Contents

Chapter 1

Introduction

A common approach to gathering detailed run-time information about applications is to run the software of interest in a functional, that is instruction-level, simulator such as Valgrind [190] or Pin [162]. In contrast to regular execution, where instructions run natively on the CPU, a functional simulator executes applications in a virtual machine (VM). It gradually interprets or translates instructions so that they operate on the state of the VM instead of the physical machine. This way, the execution becomes completely transparent and can be freely instrumented.

Whenever functional simulation is involved, operating-system-centric research usually depends on functional *full system* simulation (FFSS), which includes system libraries, services, drivers, and privileged kernel-mode components. FFSS has been demonstrated to be very effective for OS debugging [137], security analyses [287], and collecting detailed execution traces [286]. Google engineers identified over 20 kernel security vulnerabilities in Windows 8 by analyzing the memory access patterns at the system call interface [133]. In a similar analysis, we revealed a critical code execution vulnerability in the Xen hypervisor [279]. However, full system simulation is not only an important tool in operating system research. It has been shown that even for application-level studies simulating the application without the underlying operating system can be highly inaccurate [53, 135, 169, 292].

While functional full system simulation has proven to be very powerful, a well-known limitation is its immense slowdown. Depending on the required level of detail and the degree of instrumentation, running a workload with FFSS is up to multiple orders of magnitude slower compared to native execution. We have measured average slowdowns of 30x and 60x with QEMU [38] for plain simulation and tracing of memory accesses, respectively. While a kernel build takes less than 13 mins natively, running the exact same execution takes around 5 hours with QEMU; and 10 hours when tracing memory accesses. A single tracing run of povray even takes almost 2 days, whereas a non-instrumented execution

completes in less than 19 mins natively. With Simics [163], a full system simulator offering more advanced instrumentation capabilities, we measured a slowdown of up to 1000x when hooks for memory tracing are installed [219]. Similar slowdowns for functional simulation have been reported by other researchers [58, 135, 164, 200, 286].

In practice, this slowdown creates severe obstacles for comprehensive use of functional full system simulation:

Interactivity Scenarios that should capture interactivity with a human user or an external network device are not feasible. A single keystroke can quickly take from multiple seconds up to minutes until being fully processed, making human-user interactions cumbersome and unnatural. Network protocols such as TCP, in turn, react to the slowdown with throttling and timeouts.

Accuracy of Results Since the simulation considerably slows down the simulated applications and operating system, activities dependent on events external to the virtual machine such as I/O operations appear to complete faster – a phenomenon called *time dilation* [169]. This distorts measurements and produces unrealistic execution behavior.

Coverage Evaluating a test scenario in full length can take considerable time, forcing researchers to reduce coverage. The authors of the Google study summarize that the slowdown was the primary restrictive factor, which limited the coverage of their analysis to the system boot phase and short desktop usage [133] – potentially missing further vulnerabilities.

A common method to cope with the long run time of full system simulations is to adapt or shorten the workload so that the overall simulation time remains in reasonable bounds [140]. However, developing reduced but representative workloads is a complex and time-consuming task and depending on the use case (e.g., security analysis) the method is not applicable. It is thus desirable to keep the original workload intact and to accelerate the simulation instead.

The speed of a functional simulation is mostly determined by the ratio of executed physical instructions per simulated instruction. To improve this ratio simulators have been vigorously optimized [67, 95, 119, 120, 281]. A common denominator of all these techniques, however, is that improvements at the level of the generated code and execution flow do not provide the required leap and generally stay below the 5x speedup mark.

A prevailing practice to accelerate simulation is thus to limit the simulation to samples [229, 237, 284]. The gathered information can then be extrapolated to draw conclusions for the whole workload. At the same time, this represents a decisive disadvantage because sampling can only be used to estimate metrics that can be extrapolated from samples (e.g., instructions per cycle). However, it is less suited to observe the actual system execution as required in security research,

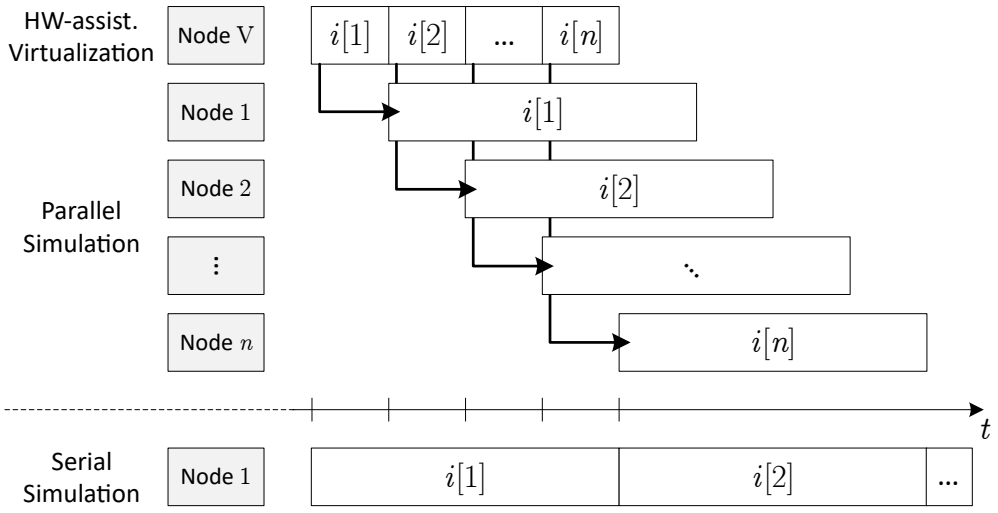


Figure 1.1: The workload is executed with fast hardware-assisted virtualization. Checkpoints at the interval boundaries serve as starting points for parallel simulations. The parallel simulation completes much faster than a serial simulation.

malware analysis, or debugging. It does not permit the tracking of individual events such as pairs of memory allocations and deallocations [52] or specific memory access patterns [133]. Instead, these applications require continuous simulation.

A promising approach is the partitioning and parallelization of simulation time [111]. The idea is to split the simulation into disjoint time intervals that can be processed simultaneously. The method proved to possess high scalability with speedups near the number of employed CPUs [101, 192, 211, 229, 267]. However, current solutions based on this concept are not applicable to continuous functional simulation, are restricted to single user-mode processes, or lack interactivity. Moreover, the dependency on process forking for spawning parallel simulations restricts most implementations to the degree of parallelism that can be supplied by a single host[1], depriving the concept of its primary strength.

In this thesis, we present **SimuBoost** [219], an acceleration approach based on partitioning and parallelization of simulation time that drastically reduces the slowdown of functional full system simulation (see Figure 1.1). The core idea is to run the workload in a virtual machine using fast hardware-assisted virtualization (e.g., Intel VT-x [125]). This virtual machine offers full interactivity. At regular intervals[2], the hypervisor takes snapshots of the VM (i.e., memory contents, device states, etc.). The checkpoints then serve as starting points for

[1]For instance, pFSA [229] forks a hardware-assisted virtual machine and switches to simulation in the newly created child. Performing a subsequent process migration would in turn require *checkpointing* the process, which is equivalent to checkpointing the VM.

[2]Between hundreds of milliseconds and multiple seconds.

simulations, enabling to simulate and analyze each interval simultaneously in one job per interval. By transferring checkpoints over the network and using multiple nodes (i.e., CPU cores and hosts) a heavily parallelized simulation of the target workload can be achieved.

Functional full system simulation can be built to always produce identical runs. Hardware-assisted virtualization, in contrast, is subject to non-deterministic input such as erratic I/O completion timing. SimuBoost records this non-determinism and uses heterogeneous deterministic replay [87, 287] to accurately reproduce the execution in the simulations, including realistic timing behavior, as well as user and network input in interactive workloads.

Both checkpointing and recording of non-determinism need to be geared toward low run-time overhead to (1) retain the execution speed difference that drives the parallelization, (2) keep perturbations on the examined workload as small as possible, and (3) preserve seamless interactivity. The simulation nodes, in turn, need to quickly receive and load the checkpoints. Furthermore, the resource consumption (e.g., memory) of simulations should be kept low so as to permit a maximum degree of parallelism on each host. In the course of this thesis, we present and evaluate techniques to cope with these challenges.

Our prototype for QEMU/KVM [38, 139] is able to drastically reduce the run time of functional full system simulation while maintaining full interactivity. Simulating an entire kernel build using SimuBoost completes in about 15 mins (in contrast to 5 hours with serial simulation), which is only 16% more time compared to a native run with hardware-assisted virtualization (without SimuBoost). SimuBoost is able to deliver this performance even with memory tracing, completing the instrumented simulation in only 19% more time compared to the same baseline.

1.1 Contributions

This thesis makes the following main contributions:

- SimuBoost is the first project to apply the concept of partitioning and parallelization of execution time to interactive full system virtualization in a manner that allows for immediate parallel functional simulation. Our evaluation examines the applicability and effectiveness of the approach to diverse types of workloads and determines its scalability with regard to increasing simulation slowdown from instrumentation.
- This thesis presents the first formalization of the partitioning and parallelization process with a performance model to predict parallel simulation times and estimate optimal interval lengths for given workloads and simulation cluster sizes.

- With continuous checkpointing constituting a central technology of SimuBoost, this thesis includes a thorough quantitative comparison of widely employed checkpointing and page dirty logging techniques. With *pre-scan*, we devised a novel method to track page modifications in virtual machines that combines the low downtime of conventional page-protection-based dirty logging with the low overhead of page table scanning.
- In contrast to previous work, SimuBoost places a strong focus on scalability beyond the limits of a single physical machine. This thesis therefore gives a detailed analysis of checkpoint compression methods to allow for live distribution of checkpoints over commodity network infrastructure such as Gigabit Ethernet. In this course, we present a new method for compression of state maps called *SimuBoost device state compression (SDS)* that achieves 61% higher compression than LZ4 in 21% less time.
- As part of this thesis, we developed the first publicly available full system heterogeneous deterministic replay engine for the x86 platform[3]. Moreover, this is the first work in the research literature to describe *MMU-induced non-determinism*, the challenges of its detection, and possible solutions.

1.2 Scope of This Thesis

This thesis serves to evaluate the general applicability of partitioning and parallelization to interactive continuous functional full system simulation with a focus on scalability beyond single machines. That means we concentrate on the conceptional properties of this acceleration method (e.g., its speedup characteristics) and the technical challenges it entails in the fields of checkpoint creation and distribution as well as deterministic replay.

We limit the discussion to single-core virtual machines, thereby building a foundation for future research on accelerating multicore VMs. As we are well aware of the fact that deterministic replay comes with increased overhead in such environments, we provide a detailed overview on the current state-of-the-art in multiprocessor replay in § 2.4.3.

Although SimuBoost is perfectly suitable for malware analysis and we consider this an important use case, we also do not dive into the challenges of preventing malware from breaking replays, for example, by exploiting inaccuracies or bugs in the simulator. This is merely a question of accurate engineering and a deliberate definition of the replay boundary rather than general feasibility as demonstrated by Yan et al. [287].

[3] https://github.com/simutrace/

1.3 Underlying Publications and Theses

Besides others, we advised the following students when they wrote their study, diploma, bachelor's, or master's theses at the Operating Systems Group. Their evaluations provided valuable insight and directed the final design of SimuBoost. They have contributed to this thesis, in chronological order:

- **Nikolai Baudis** implemented the first prototype for incremental checkpointing as part of his bachelor's thesis [37]. He also investigated data duplication in incremental checkpoints and explored MongoDB [10] as a checkpoint storage solution.
- **Bastian Eicher** examined alternatives to MongoDB in his master's thesis [89] in order to reduce the downtime for stop-and-copy checkpoints. This led to the use of plain files. This design eventually made its way into our final prototype – although entirely revised in implementation and format. Bastian Eicher also developed a first version of the performance model for constrained hardware setups and expedited checkpoint distribution by using a pulling approach that uses Simutrace's built-in network support.
- **Jan Ruh** expanded upon the analysis of recurring memory pages and disk sectors in incremental checkpoints in his bachelor's thesis [224] by identifying the semantic background of duplicates.
- **Nico Böhr** supplied a first prototype of SimuBoost's copy-on-write checkpointing in his bachelor's thesis [42] so as to further reduce the downtime of checkpoints and preserve interactivity.
- **Janis Schötterl-Glausch** investigated in this bachelor's thesis [234] the suitability of Intel Page Modification Logging (PML) [125] for alleviating the run-time overhead incurred by the detection of modified pages in incremental checkpointing.
- **Simon Veith** implemented a heterogeneous deterministic replay engine for the ARMv7 architecture in his master's thesis [262], thereby exploring the applicability of SimuBoost to other architectures than x86.
- **Thomas Schmidt** supported this thesis with his bachelor's thesis [233] by exploring alternatives to functional full system simulation for memory tracing. He thereby reinforced the motivation for simulation as the method of choice for detailed run-time information retrieval.
- **Andreas Pusch** worked in his master's thesis [209] on the efficient distribution of checkpoints in a simulation cluster. He tested distributed file systems and developed the multicast-based checkpoint transfer that we envision for SimuBoost. His work has yet to be integrated into our prototype.

- **Michael Zangl** evaluated in his master's thesis [293] the applicability of chunk-based recording [215] to heterogeneous deterministic replay for multiprocessor virtual machines. His work shows a possible path for future multiprocessor support in SimuBoost.
- **Johannes Werner** has made an assessment of virtual machine working sets in his bachelor's thesis [277] to supply a quantitative basis for the development of sparse checkpoints.
- **Jan Ruh** subsequently developed a working implementation of sparse checkpointing in his master's thesis [225]. This technology presents a decisive factor in the scalability of SimuBoost as is allows for a much higher density of simulations per physical host.
- **Benedikt Morbach** studied the origin of MMU-induced non-determinism in his master's thesis [183]. He discusses different ways to cope with this generally overlooked source of non-determinism and demonstrates a working recording and replay system for it.
- **Janis Schötterl-Glausch** extended his work on efficient page modification tracking in his master's thesis [235]. He evaluated the potential of page table scanning as an alternative to setting page protections. He, moreover, implemented the pre-scan method. A majority of his code made its way into our current prototype.

The following paper (identical in name) has been published before the submission of this thesis, introducing the core idea behind SimuBoost:

- M. Rittinghaus, K. Miller, M. Hillenbrand, and F. Bellosa. SimuBoost: Scalable parallelization of functional system simulation. In *Proceedings of the 11th International Workshop on Dynamic Analysis*, WODA'13, Houston, Texas, USA, Mar. 2013.

In addition, we successively presented preliminary results at the GI Fachgruppentreffen Betriebssysteme in 2013, 2014 (Simutrace [218]), 2016, and 2018.

1.4 Organization

The remainder of this thesis is organized as follows:

Chapter 2 – Background provides a detailed overview of current virtualization technologies and compares their advantages and disadvantages with respect to dynamic analysis in the field of operating system research. The chapter also introduces important research related to checkpointing and deterministic replay.

Chapter 3 – Functional Full System Simulation then sheds a light on the possibilities that functional full system simulation offers in the retrieval of detailed run-time information on programs and the operating system. It demonstrates how the slowdown of this technology poses a major obstacle for its comprehensive use. Afterward, the chapter gives a thorough introduction to existing methods for acceleration and explains their shortcomings for operating system research.

Chapter 4 – SimuBoost formulates a set of goals for a suitable acceleration technique, elaborates on the idea behind SimuBoost, and finally highlights differences to existing work based on partitioning and parallelization of simulation time.

Chapter 5 – Performance Model describes a formalization of the parallelization process and a deduces of a set of equations for predicting parallel simulation times and optimal interval lengths.

Chapter 6 – Continuous Checkpointing contrasts techniques for efficient continuous checkpoint creation, storage, and loading and presents detailed quantitative results.

Chapter 7 – Checkpoint Distribution deals with the distribution of checkpoints in a simulation network. With the goal of being able to use SimuBoost with commodity network infrastructure, this chapter puts a focus on strong checkpoint compression.

Chapter 8 – Heterogeneous Deterministic Replay concludes the design overview with a discussion of the heterogeneous deterministic replay component. The chapter especially elaborates on the refinements we have made to the binary translator in QEMU to faithfully reproduce actual hardware behavior.

Chapter 9 – Evaluation puts all building blocks together and gives a thorough evaluation of the overall SimuBoost concept; examining its speedup and scalability characteristics. In this chapter, we also review the applicability of the performance model using measurements of actual parallel simulations.

Chapter 10 – Conclusion summarizes the key concepts and findings of this thesis and closes with an outlook on future research directions.

Chapter 2

Background

This thesis presents a novel approach to full system analysis that equips researchers and developers with a tool to inspect arbitrary workloads down to instruction-level detail while maintaining interactivity and fidelity. A two-stage analysis workflow leverages a combination of (1) fast hardware-assisted virtualization for realistic workload execution and (2) heterogeneous deterministic replay in a full system simulation for detailed analysis. In contrast to previous work, our approach is streamlined for functional system analysis, rather than microarchitectural analysis. Therefore, it lends itself to holistic operating system and software-level security research. Whereas current tools in this area suffer from the immense slowdown induced by functional system simulation, we present a lightweight checkpointing-based method to efficiently parallelize the simulation, thereby drastically increasing simulation speed.

In this chapter, we introduce terms, techniques, and principles that this thesis is based on. Sections 2.1 and 2.2 supply an introduction to the concept of virtualization and virtual machines, which are a key technology to our approach. A focus is placed on virtual machines capable of running full operating systems with the help of dynamic binary translation and hardware-assisted virtualization. Sections 2.3 and 2.4 expand on the topic of virtualization by covering relevant technologies and literature in the area of virtual machine checkpointing and deterministic execution replay.

2.1 Virtualization

Virtualization describes the process of representing some form of resource through a virtual one. This introduces a level of indirection which can have diverse benefits such as increased control, flexibility, or isolation. The underlying resource can be a physical one or, in the case of nested virtualization, itself be already virtual.

A prominent example is *virtual memory*, which was first available in the early 1960's [94] and today is the de facto standard in desktop computers, servers, and even mobile devices [32, 239]. Prior to the adoption of virtual memory, physical memory was directly exposed to programs running on the system. As a process can potentially access arbitrary memory in this configuration, it may inadvertently or maliciously address memory owned by another process or even the operating system (OS). This severely complicates memory management, especially with multiprogramming, and at the same time puts the system's stability at risk.

In contrast, virtual memory systems add an additional indirection for memory addresses. Subsequently, all addresses used in a process are interpreted as virtual addresses and are translated to physical ones at run time by an address mapping device – the *memory management unit (MMU)*. It is the responsibility of the operating system to establish the mappings between virtual and physical addresses. Typically, each process gets its own set of mappings, thereby creating distinct virtual address spaces. Contrary to direct physical memory access, this makes providing and sharing physical memory an explicit decision by the operating system and greatly improves stability and security. In addition, memory management is simplified because virtually contiguous regions do not need to be contiguous in physical space – a feature called *artificial contiguity* [214]. Partitioning the RAM for use by multiple processes thus becomes much easier.

Virtualization can also be an opportunity to incorporate new features and extend the interface provided by the raw underlying resource. For instance, the MMU invokes the operating system with a *fault*, if it is unable to translate a virtual address [254, p. 234f]. The OS can leverage this mechanism for implementing memory overcommitment. This creates the illusion for a process to own more physical memory than actually available in the system. To this end, the operating system dynamically swaps data between secondary storage (e.g., a hard disk) and RAM, adjusting mappings as needed. If a process accesses virtual memory that is currently not backed by RAM, the fault gives the operating system the opportunity to trigger swapping. Since execution can generally continue afterward, these operations are transparent to a process.

More formally, the principle of virtualization can be expressed by defining a set of virtual resources $V_t^p = \{v_0, v_1, \ldots, v_n\}$ at the time t that are mapped to the physical resources $R_t = \{r_0, r_1, \ldots, r_m\}$, where $p \in \{p_0, p_1, \ldots, p_q\}$ denotes a particular instance of V [77, 102]. For each moment of time t and each instance p of V, there exists a function [102]:

$$f : V_t^p \rightarrow R_t \cup \{\phi\}$$

such that if $v \in V_t^p$ and $r \in R_t$, then

$$f(v) = \begin{cases} r & \text{if } r \text{ is the physical resource for v} \\ \phi & \text{if } v \text{ does not map to a physical resource} \end{cases}$$

The value $f(v) = \phi$ causes a fault that invokes a handling procedure in some form of privileged execution context which controls f. In a virtual memory system, V^p holds all addresses of the virtual address space (i.e., process) p, R comprises the set of available physical addresses, and $f(v) = \phi$ denotes an exception, where the CPU transfers control to the fault handler in the OS. For virtual memory, typically $n \cdot q \gg m$ applies, that is, the number of virtual pages n over all processes q is much larger than the number of available physical pages m.

With nested virtualization, V and R can be interpreted as two adjacent levels of virtual resources where the physical resources are positioned in layer 0 and f maps virtual resources from layer $k+1$ to layer k [102].

When considering two consecutive time steps (see Figure 2.1), we can characterize virtualization as the construction of the isomorphism $h \circ f(V_t) = f \circ g(V_t)$, where for a sequence of operations g that modifies V there is a corresponding sequence of operations h that performs an equivalent modification to R [206,240]. In practice, this means we must implement $h \circ f(V_t)$ to create the illusion of $f \circ g(V_t)$.

In a virtual memory system, possible operations g on V are reads and writes. So in order to implement virtual memory, we must be able to perform equivalent reads and writes on the corresponding physical addresses, thereby constructing h. Today's CPUs fully integrate this functionality. For each virtual memory access, the CPU first inquires the physical address. The MMU handles the address translation, thus supplying f. The CPU then realizes h by simply performing the requested operations on the physical memory.

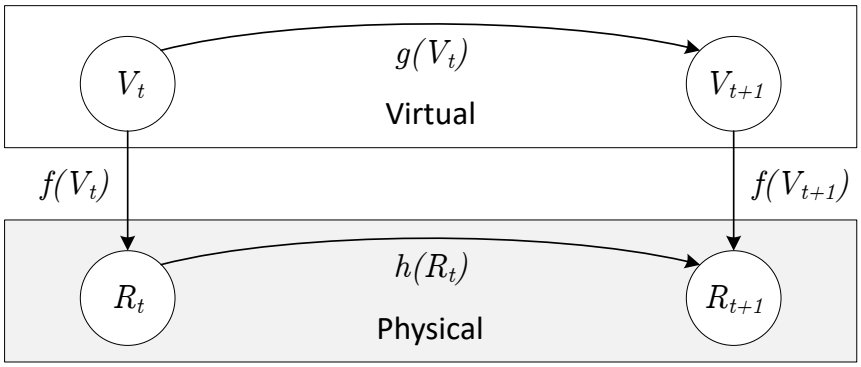


Figure 2.1: For each state modification g in V, there must be an equivalent modification h in R [240, p. 4].

2.1.1 Virtual Machines

We can generalize V and R to contain not only one particular type of resource such as memory but, for instance, all resources comprising a modern computer system. This includes processors, secondary storage, and connected devices (e.g.,

network adapters). V then defines a *virtual machine (VM)* and R represents the physical machine. We refer to V as the *guest*, whereas R is the *host*. The situation $f(v) = \phi$ is called a *VM fault*. The privileged code that implements f and h, and which is responsible for catching faults is the *virtual machine monitor (VMM)*.

Compared to a virtual memory system, a VM imposes considerably higher complexity. However, the same formal definition holds. For building a faithful reproduction of a physical machine the VMM must provide the following three major components:

Virtual State Description (V) The set V of virtual resources must describe a complete model of the guest's state. This may include the register contents of virtual CPUs, status information of virtual interrupt controllers, a screen buffer for a virtual display adapter, and much more. The model can be restricted to the purely functional level, but may also include microarchitectural features, depending on the virtualization purpose.

Resource Mapping (f) Virtual resources must always be backed by physical resources at some point. To that end, the VMM can partition spatial resources as in the case of main memory and secondary storage, or use multiplexing via time-sharing for resources such as CPU time. Sometimes virtual resources cannot be simply mapped 1:1 to host resources if the architectures of V and R differ. In these cases, the VMM has to explicitly model the virtual resource in R. This additionally has to be done in a way that is compatible with R. For instance, a 64-bit guest CPU register might need to be split into two 32-bit words on a 32-bit host.

Host State Transfer Function (h) While V and f are sufficient to describe a virtual machine at a fixed point in time, execution requires a mechanism that modifies the host in a way that is equivalent to what is dictated by the operations in the guest. For a CPU, this task can be accomplished for example by some form of emulation. The VMM interprets each processor instruction in the guest and modifies the host's representation of the guest's CPU accordingly. In addition, the VMM has to advance the virtual instruction pointer as well as virtual timers and reflect actions on virtual I/O devices.

Tightly connected to the types of virtual resources comprising a virtual machine is the architectural layer at which the virtual machine is constructed. Considering a typical computer system architecture (see Figure 2.2), we can identify three primary layers [240, p. 6ff]:

The *application programming interface (API)* specifies an expected behavior and methods of communication with a piece of software that implements this behavior. Usually, a developer is confronted with an API as a set of source-code level functions accompanied by a set of well-defined parameters and return values. A prominent example in the field of operating systems is POSIX.1 [124].

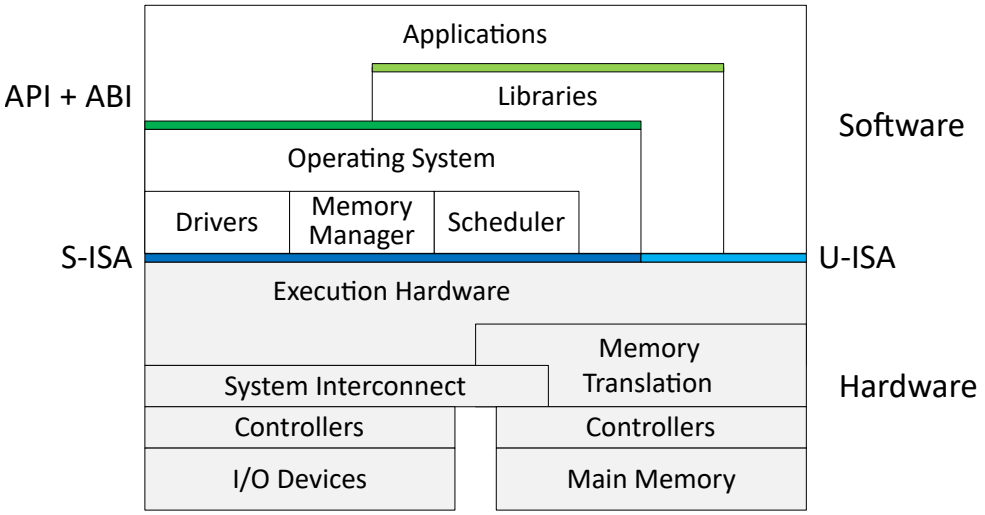


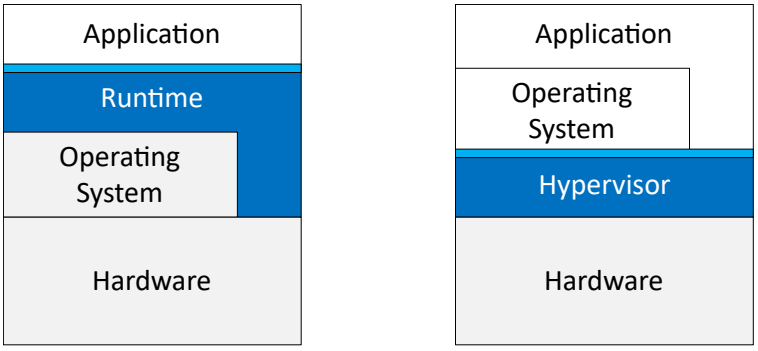
Figure 2.2: Applications use a well-defined API and ABI to interface with libraries and the operating system. They may only use unprivileged instructions defined by the user ISA, while the OS also has access to privileged operations specified by the system ISA [240, p. 7].

The *application binary interface (ABI)*, in contrast, specifies the exact machine-level format of data structures (e.g., field alignment) and a calling convention. The latter determines into which registers and stack locations the caller and callee should place parameters and return values, respectively. The ABI thus depends on the characteristics of the hardware on which the software should run.

The *instruction set architecture (ISA)* builds the boundary between software and hardware. It defines the basic interface of the hardware, including available machine registers, processor instructions, the I/O model as well as memory alignment and consistency constraints. The ISA can be split into an unprivileged interface – the *user ISA* – available to all processes running on a system, and a privileged one – the *system ISA* – eligible for operating system access only.

Process Virtual Machines

A VM virtualizing the API, ABI, and user ISA is called a *process virtual machine* (see Figure 2.3a) [240, p. 13ff]. Characteristic of this type of VM is its restriction to a single user-level process. The VMM is thus often referred to simply as *runtime*. A runtime does not virtualize the whole physical machine, but merely provides programs an execution environment which can deviate from the one defined by the underlying operating system. The simplest manifestation of a process virtual machine is an unchanged process context, effectively obliterating the runtime. The deviation from this native context, however, may be of any arbitrary extent.



(a) Process Virtual Machine (b) System Virtual Machine

Figure 2.3: A VM constructed at the level of the API, ABI, and user ISA forms a process virtual machine. It virtualizes a single application only and runs alongside native processes on the hosting OS. A VM implemented at the ISA-level (user+system) virtualizes an entire computer system and is called a system VM. It is capable of running operating systems.

With DIGITAL FX!32 [61], Hookway presented a process virtual machine which enabled Windows NT 4.0 systems designed around the Alpha processor to transparently execute x86 NT applications. The API (i.e., the Windows API) thus remained the same, while the (user) ISA and ABI changed and needed to be translated by the runtime.

A more recent example is Linux Subsystem for Windows (WSL) [291], which runs unmodified x86 ELF64 Linux binaries on Windows. It leaves the instruction set architecture fixed and translates the API and ABI from Windows to Linux. With Wine [31], a process VM operating in the reverse direction is available as well.

Process virtual machines also serve as a runtime environment for high-level languages such as Java. The Java virtual machine (JVM) [155] abstracts the operating system in a set of standard libraries (Java API) and comes with a virtual ISA as part of the binary specification. The JVM thereby performs a translation on all three layers (i.e., API, ABI, and user ISA), making Java applications generally platform independent.

While each of these examples performs excessive virtualization, leaving all three layers untouched is equally reasonable. For example, applications typically come as a collection of dynamically linked libraries, preventing optimization across static program binaries. Dynamo [34] addresses this problem by employing a process VM to transparently optimize programs at runtime. Compared to a static compiler, Dynamo can interpret applications as a trace of instructions rather than a set of independent binaries. Since these traces are a product of the input given to the program, Dynamo can tailor the code to the usage profile at hand.

A major drawback of process virtual machines, however, is their inability to run operating systems. This limits their applicability in the field of operating system

research, where a holistic view on the interaction of applications, the OS, and the hardware is usually required. Even for application-level studies it has been found that leaving out the effects of operating system invocations (e.g., on CPU caches) can considerably corrupt measurements [53, 135, 292].

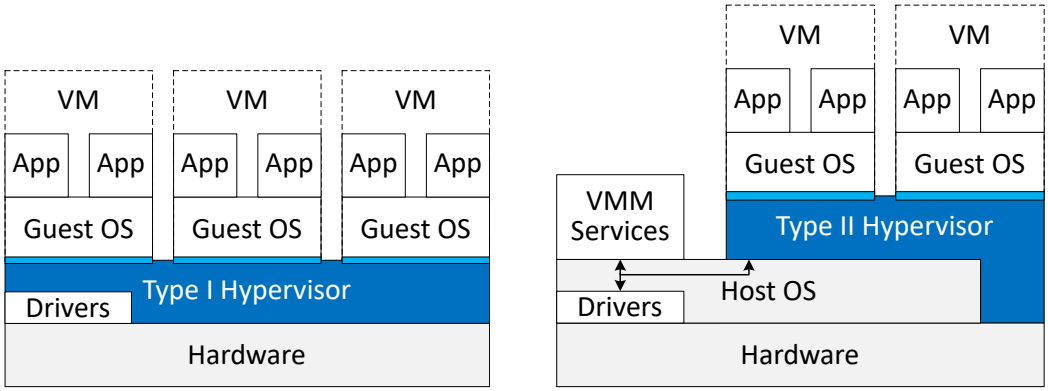
System Virtual Machines

A virtual machine that is constructed with the system ISA in mind is called a *system virtual machine* and the VMM is often referred to as *hypervisor* (see Figure 2.3b) [240, p. 17ff]. Supporting this level of virtualization comes with the necessity to emulate system events like exceptions and interrupts and to considerably extend the set of virtual resources included in the virtual state description V. Whereas a process VM can generally limit V to CPU registers and the memory of a single user-mode address space, and h to executing unprivileged instructions, a system VM has to virtualize all devices present in a computer system, including a full-fledged MMU. As with process VMs, the ISA may or may not match between the host and the guest. While all this raises the complexity of the VMM, it also puts system VMs into the position to run operating systems. This has made system VMs a very powerful and widely employed concept.

System virtual machines were first used in the 1960s and 1970s to provide software compatibility, aid in modification and testing of operating systems, and run diagnostic programs [103]. IBM VM/370 [106] is the best-known hypervisor of that time supporting full system virtualization. Over the years, many additional use cases have emerged. Examples are migrating operating systems to new platforms [48, 260], providing strong isolation to safely run untrusted applications and services [221, 278], flexible resource management via migration within and across data centers [24], fault tolerance [45], software distribution [222], and intrusion detection [87, 97] – to name only a few. Chen et al. [57] even argue that operating systems and applications should generally be relocated to system virtual machines for transparently adding new services below the OS level. In fact, many IaaS and PaaS offerings in the cloud today are based on system virtual machines, the ecosystem around Amazon EC2 being only one prominent example.

The architecture of existing hypervisors can be roughly classified into two general designs, which are illustrated in Figure 2.4:

A *type I hypervisor* runs directly on bare hardware without the need for an operating system (see Figure 2.4a). This has the advantage that the extent of the trusted computing base (TCB), that is, all software that security depends on [147], is kept small. However, missing the infrastructure provided by an operating system forces a type I hypervisor to reimplement essential system components such as physical memory management and (vCPU) scheduling. Furthermore, a type I



(a) Type I Hypervisor (b) Type II Hypervisor

Figure 2.4: A type I hypervisor runs on bare hardware. It thus has to come with own drivers. A type II hypervisor is installed on an existing host operating system and builds upon the abstractions provided by the OS to provision resources to virtual machines. Hybrid approaches may additionally resort to direct hardware access (e.g., through a kernel module) for controlling hardware-assisted virtualization features.

hypervisor cannot resort to existing drivers, effectively narrowing the choice of systems that are supported. A popular type I hypervisor is VMware ESXi [266].

A *type II hypervisor* counters these disadvantages by utilizing a commodity operating system as its interface to the hardware, for the cost of a much larger TCB (see Figure 2.4b). In this setup, the hypervisor runs on a par with other processes on the system [2, 38, 46]. Research has also been done to optimize the hosting OS for this scenario [136]. If the VMM makes use of privileged hardware virtualization features such as Intel VT-x [125] or ARM TrustZone with Hyp mode [32], the VMM is typically accompanied by a kernel-mode component. Popular representatives for this are QEMU/KVM [74, 139] and VirtualBox [195].

In practice, many hybrids of these designs have emerged. Microsoft Hyper-V [173] and Xen [35], for instance, run on the bare machine as in type I but always create a specific privileged virtual machine – called Domain 0 (Dom0) in Xen – which hosts an operating system for hardware control as in type II. VMware Workstation [49], on the other hand, dynamically switches between the host OS and the hypervisor.

2.1.2 Conclusion and Terms

Virtual machines lift the concept of virtualization to a higher level by incorporating multiple resource types into the set of virtual resources. While process virtual machines are limited to a single user-level application, system virtual machines create a whole virtual computer system, capable of running operating systems. At the same time, the level of indirection introduced by virtualization can translate

between different APIs, ABIs, and ISAs in the guest and host or provide an anchor for instrumentation, thereby allowing researchers to collect detailed run-time information. This makes system virtual machines particularly interesting for operating system and system security research. In addition, they can improve the accuracy of application-level studies by including OS invocations, thereby resembling a more realistic execution environment.

> In the remainder of this thesis, we therefore concentrate on running operating systems and applications in system virtual machines for the purpose of detailed, holistic inspection with a focus on research and development. We use the terms hypervisor and virtual machine monitor (VMM) interchangeably and refer to system virtual machines when simply using the term virtual machine (VM). Based on our implementation, we concentrate on type II hypervisor technology, although the proposed concepts are equally applicable to type I hypervisors.

2.2 Virtualization Techniques

The power of virtual machines comes at a high price regarding the complexity of the virtualization layer. The virtual state description V, the resource mapping f, and the host state transfer function h reflect this by having to specifically be designed for each type of resource.

In the following, we divide the virtualization of a computer system into three major resource types (see Figure 2.5): (a) the processor, (b) the memory hierarchy, and (c) miscellaneous I/O devices. We take a brief look at the virtualization techniques commonly used in each area to give an overview of the strengths and weaknesses of the individual approaches and their resulting applications. In this thesis, we focus on *functional* virtualization which is only concerned with faithful reproduction of the ISA, completely omitting microarchitectural details.

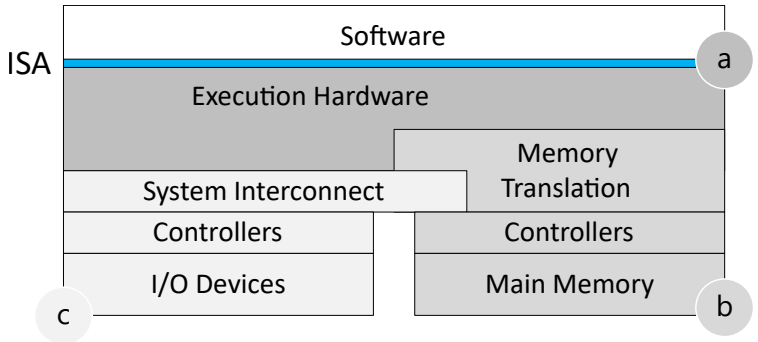


Figure 2.5: We can divide a computer architecture into three major areas: (a) the processor, (b) the memory hierarchy, and (c) I/O devices.

The following explanations on virtualization techniques are laid out accordingly. The section closes with a case study on QEMU/KVM, the virtualization software used as a basis for our implementation. The technical background helps to comprehend the design decisions and implementation details in this thesis. While the descriptions focus on Intel x86, most concepts are equally applicable to other modern architectures.

2.2.1 Processor Virtualization

The central processing unit (CPU) is the heart of a computer system and as such, the processor virtualization is an integral component in each virtual machine monitor. The task of the VMM is to construct a virtual representation of a physical processor *in software* – a virtual CPU (vCPU). The vCPU's architecture may or may not match the host processor's architecture.

Virtual State Description (V) The virtual state description has to comprise all state that the guest processor holds. This heavily depends on the level of detail that a CPU model should have. For functional virtualization, this is foremost represented by the processor's externally accessible registers. For instance, in the case of an x86-64 processor V has to maintain, besides others, the general-purpose registers `RAX` to `R15` as well as the instruction pointer `RIP`, the flags register `RFLAGS`, and the control registers `CRx`. In addition, V includes CPU internal information that is necessary to faithfully reproduce the externally visible behavior – e.g., the current privilege level (CPL).

Resource Mapping (f) The resource provided by a processor is its computation time. Backing computation time of a guest's vCPU with physical host resources thus equates to donating host computation time to this particular vCPU. Irrespective of the actual method of implementing this, the net effect is always the execution of guest instructions by one of the host's processors. This is comparable to an operating system provisioning computation time to a certain thread by dispatching it. In fact, some hypervisors map each vCPU to a dedicated thread [139], thereby delegating vCPU scheduling to the host OS.

Host State Transfer Function (h) Transferring the state of the guest processor from one time step to the next is equivalent to modifying V in accordance to the instructions at the vCPUs instruction pointer (IP). The method that a particular VMM uses to accomplish this task is the major criterion by which the use cases for this VMM are defined. This is because the processor virtualization greatly determines the execution speed of the VM and the degree of instrumentation that can be applied. The choice of the virtualization method also inherently decides if the VMM is capable of translating between different guest and host ISAs.

In the following, we recapitulate the four methods for processor virtualization proposed in the literature. Although the descriptions remain limited to single-core virtualization, the techniques can equally be applied to multicore virtualization, for example, by duplication onto multiple host cores [139] or by multiplexing [260].

Interpretation

To construct a virtual processor the easiest approach is to take a look at how physical CPUs process instructions and develop a design based on that concept. To improve the throughput for a given clock rate via instruction-level parallelism, CPUs divide each instruction into a series of steps, thereby forming a pipeline. Although in its simplicity not representative of modern CPUs, the classic 5-stage RISC pipeline illustrated in Figure 2.6 provides a good overview of what phases each instruction must pass until it retires [112, p. A.5f]:

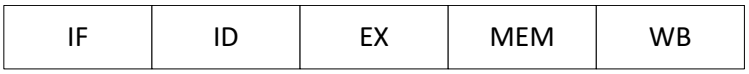


Figure 2.6: The classic 5-stage RISC processor pipeline: instruction fetch (IF), instruction decode (ID), execute (EX), memory access (MEM), and write back (WB) [112, p. A.7].

In the first phase, *instruction fetch (IF)*, the processor reads the current instruction from the main memory using the address in the IP. To save space and memory bandwidth, each instruction is encoded in a dense machine specific format as laid out in the instruction set architecture (ISA). The next stage in the pipeline is therefore *instruction decode (ID)*, which extracts the operation and its operands from the encoded representation and reads the CPU's register file as needed. The *execute (EX)* cycle is responsible for performing the actual computation using an arithmetic logic unit (ALU) for integer and boolean register operations. For branch instructions, this stage can be used to determine the branch target[1]. For memory referencing instructions, the execution stage computes the effective memory address, for example, by adding a base register and an offset. The actual memory access is done in the *memory access (MEM)* stage, whereas the result of register-register and register-immediate operations is simply forwarded to the next phase. The processing of the instruction is finalized in *write back (WB)*, where results are written to the register file.

An interpreter is the simplest software implementation of this procedure (see Figure 2.7). Its core is the decode-and-dispatch loop which processes one instruction at a time until a halt condition such as a VM shutdown is met or a virtual interrupt arrives [240, p. 29ff]. The operations performed in the loop mirror the stages in a

[1] With instruction-level parallelism the branch target is often computed early in the decode stage to reduce wasted cycles where the pipeline is fed with wrong instructions.

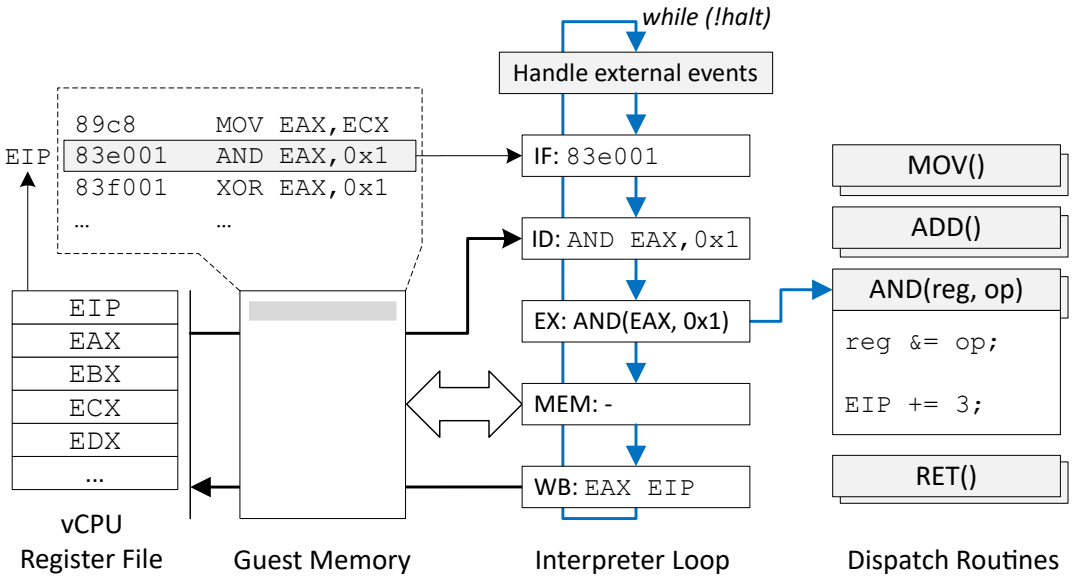


Figure 2.7: The core of an interpreter is the decode-and-dispatch loop which processes one instruction at a time. The loop calls dispatch routines specific to each instruction to perform computations on the guest's memory and register file. External events such as interrupts are handled on instruction boundaries.

basic processor pipeline. An important characteristic of interpreters is the use of dispatch routines to implement the execution cycle. The interpreter assigns each instruction an individual routine which carries out the necessary computations on the virtual CPU registers. The appropriate routine is then called from the loop.

To interpret CISC architectures a simple RISC-like design is not sufficient. On x86, for example, a logical `AND` instruction is not limited to operate on registers but can also directly work with a memory operand and an immediate value [125]. The interpreter therefore must be able to read, modify, and write memory in one instruction. To that end, the fixed structure of the main loop can be relaxed and the dispatch routines are empowered to load and store guest memory, effectively merging the MEM and WB cycles into the dispatch routines for greater flexibility. Another solution is to generate a set of RISC-like micro-operations in a decode front end just like physical x86 CPUs do [93, 125]. Tröger et al. [258] demonstrated this approach in Bochs [2], a popular interpreter for the x86 platform.

An advantage of interpreters is their simple approach to virtualization, which facilitates implementation. With the addition of dispatch routines, an interpreter can be extended to support new instructions in the guest ISA without difficulty. Since these routines are usually written in a high-level programming language, the interpreter can be built for any host architecture for which a corresponding compiler exists. The translation from the guest ISA to a potentially different host ISA is thus implicitly done by the compiler, making interpretation a portable virtu-

alization technique. Interpretation also lends itself to research and development because the dispatch routines can easily be instrumented to, for example, capture instruction or memory access traces. In contrast to static binary rewriting [150], instrumentation can be dynamically turned on or off without prior preparation of the executed code.

The major disadvantage of interpretation is, however, the slowdown compared to native execution which quickly exceeds two orders of magnitude [174]. This stems from the fact that an interpreter has to execute a multitude of host instructions for each guest instruction. Milhocka et al. [174] demonstrated that some of the techniques used in silicon such as branch prediction, prefetching, and decoded instruction caches, are also effective in software, almost doubling the interpreter performance. They also found that the main loop is responsible for around 50% of total execution time. In the literature, various methods have been proposed to reduce the overhead of the main loop, most notably threaded interpretation [240, p. 37f]. However, even with this technique, overall performance remains low compared to native execution [174, 238]. For example, interpreting a register arithmetic instruction such as `ADD` in Bochs 2.3.7 takes 25 cycles on a Core 2 Duo, while the same operation takes 0.33 cycles[2] when executed natively [92]. This is even worse for more complex operations. For instance, a floating point multiplication takes 175 times the number of cycles.

Dynamic Binary Translation (DBT)

With the aim to reduce the number of host cycles spent on each guest instruction, *dynamic binary translation (DBT)* takes a slightly different approach to processor virtualization than interpretation. Instead of using dispatch routines to work on the guest state, a binary translator generates for each guest instruction a set of host instructions that perform an equivalent operation on the guest state, but which can execute directly on the host processor (see Figure 2.8).

By recompiling multiple guest instructions en bloc, a translator can reduce the virtualization overhead compared to interpretation. This way, Mimic [168], a full system simulator for System/370 based on binary translation, requires on average only 4 host instructions per guest instruction. This is because the host does not branch to dispatch routines for every guest instruction, but instead executes a sequential code sequence. This also opens the door for optimizations. If, for example, consecutive guest instructions work on the same register, the binary translator can enclose the translation with only two accesses to the vCPU's register file. It is also possible to directly map guest registers to host registers [281]. We therefore can think of the translation result as an optimized version of multiple concatenated dispatch routines.

[2] Average number of clock cycles per instruction for a series of independent `ADD`s in the same thread. The instruction latency is 1 clock cycle [92].

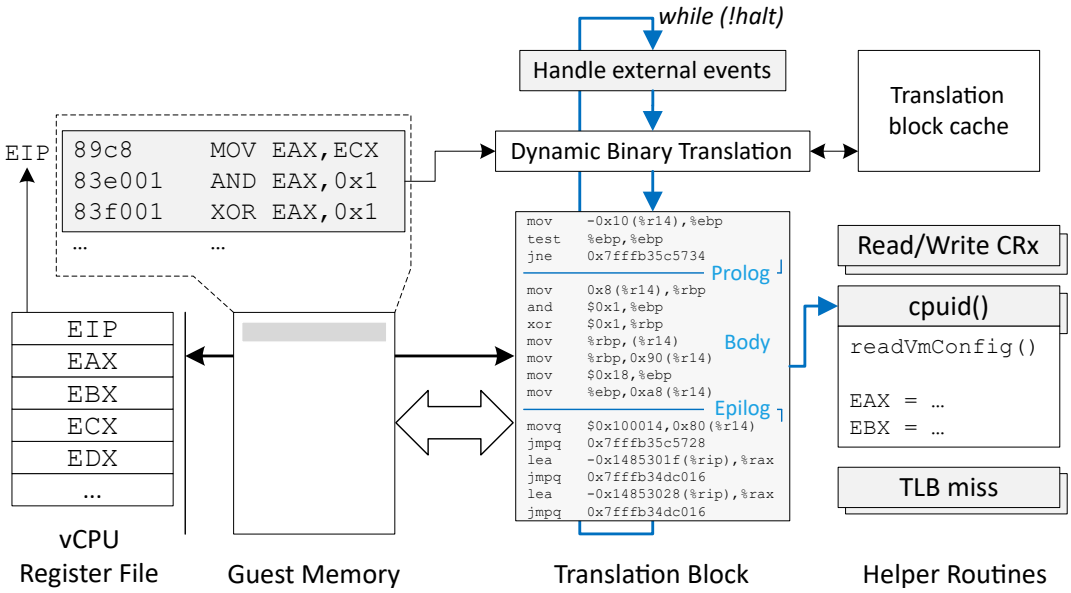


Figure 2.8: A binary translator reduces virtualization overhead by translating consecutive guest instructions to a set of natively executable host instructions which directly operate on the guest state. Helper routines facilitate the virtualization of complex instructions. Translated blocks are held in a cache to reduce overall translation overhead.

Nevertheless, to keep the complexity of the translator in reasonable bounds and preserve maintainability, seldom used but hard to translate guest instructions are usually still realized with support from *helper routines* [49, 168, 260]. An example is the `cpuid` [125] instruction on x86 which returns processor identification and feature information and requires reading the configuration of the virtual machine. Since the generated code resides in the same address space as the VMM, the translator can invoke a helper routine by simply generating a call to the routine's address. The helper routine then naturally jumps back to the translated code on return.

For a system virtual machine, which runs a full operating system and starts programs dynamically, possibly even modifying or generating code on-the-fly (e.g., self-modifying code or just-in-time compilation), a static binary translation ahead of time is not feasible. Aggravating this is the code discovery problem [240, p. 52f], which describes the difficulty inherent to von Neumann architectures to dissect code from data [65, 116]. Therefore, binary translation for system virtual machines is dynamically done at run time by the hypervisor.

Compared to interpretation, having to perform translation at run time before being able to execute guest code increases latency and virtualization overhead. Translations are thus cached for reuse. Since most programs exhibit strong locality of reference [254, p. 216] and even long-term phase behavior [34, 78, 145, 237], a translation cache can amortize translation costs. To maintain translation cache

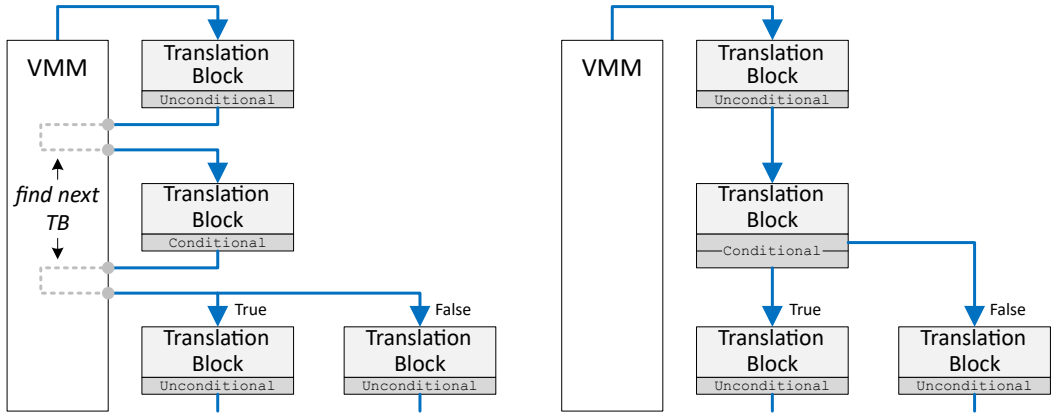
coherency under the presence of potential guest code modification, the translator can either rely on a `flush` instruction as on SPARC [242], which guest code has to use to indicate modifications [67], or it has to check memory accesses for writes to translated code [38, 75, 281]. This way, the VMM is able to detect stale translations and evict them from the cache.

The unit of translation is a *translation block (TB)*. A translation block comprises a set of recompiled guest instructions, the *body*, and is encapsulated by a *prolog* and an *epilog* that are responsible for the transition between the hypervisor and the translated guest code, and vice versa (e.g., updating the virtual instruction pointer on TB exit). The VMM sizes translation blocks based on the concept of *basic blocks* as used in compiler construction. A basic block denotes a straight-line code sequence with no branches in except to the entry and no branches out except at the exit [112, p. 67]. A translation block deviates from this concept in that it is determined by the actual execution flow of the vCPU [240, p. 57]: it begins at the instruction executed immediately after a branch or a jump and ends with the next branch or jump. This is because the prolog requires that execution always starts at the beginning of a TB. Furthermore, at the end of a TB conditional branches represent a natural barrier at which the evaluation of the condition at run time is required to decide which path the translation should follow. The same is true for unconditional branches whose branch target is computed at run time (i.e., a register-relative jump). Although translations can conceptually follow unconditional jumps with a known fixed target, it is also reasonable to avoid this. A branch target may be used by multiple code locations and in combination with a TB's single entry rule, this can impair translation block reuse.

While the idea of a translation block strictly limits control flow to a single entry and exit, in practice, a TB can also be exited out of order as part of guest exception emulation – e.g., when the guest divides by zero or on a fault in guest virtual memory. In addition, the prolog may introduce further exit paths to abort execution if necessary. Translation blocks therefore usually possess a single entry, but multiple exits.

There are also binary translators that build much larger translation blocks by taking unconditional jumps and defaulting to a predicted branch target for conditional and register-relative branches, exiting the TB on false prediction [34, 61]. While this is beneficial to reduce costly transitions between the hypervisor and guest code, it requires online profiling techniques to be effective. This complicates the translation mechanism and introduces additional overhead.

A simpler approach is *translation block chaining* [67]. The idea is illustrated in Figure 2.9. If a translation for a branch target exists, the VMM can connect the two TBs, thereby directly transferring control from one TB to the next. Conditional branches generate two exits that connect to different blocks. This saves costly passes through the VMM just like large translation blocks do. However, chaining



(a) Without Chaining (b) With Chaining

Figure 2.9: Chaining directly connects translation blocks, saving passes through the virtual machine monitor.

does not remove the overhead of each block's prolog and epilog and it does not allow optimizations across basic blocks. Nevertheless, since it brings noticeable performance improvements, is easy to implement, and can also be applied if large translation blocks are used, chaining is commonly found in binary translators [20, 34, 38, 49, 67, 99, 260, 281].

A drawback of chaining is that it makes replacement in the TB cache more difficult because the links create dependencies between blocks. A solution is to leverage back pointers [240, p. 135f] or to employ a more aggressive cache replacement strategy that flushes entire cache partitions while at the same time the cache prohibits interdependence between partitions [240, p. 138f].

A noteworthy property of binary translation is that the handling of external interrupts gets delayed. Compared to a physical machine, where interrupts can occur at arbitrary locations in the execution flow, a binary translator checks for pending events only in the main loop (see Figure 2.8). While this is also true for interpreters, a binary translator works at the granularity of translation blocks – i.e., multiple guest instructions. This delay can actually be measured in the guest and has already been used by malware to protect from inspection by security analysts [287]. Chaining even amplifies the problem. To avoid this, the TB prolog can check for pending interrupts and abort execution, or chaining can be actively broken when asynchronous events are encountered [38]. However, the delay caused by a single translation block remains, being inherent to binary translation.

To sum up, we can state that binary translation is a powerful alternative to interpretation. It is equally capable of executing system VMs with a different ISA than the host. Like with interpretation, the guest code can easily be instrumented, which makes this technique ideal for research and development. Binary translation generally provides higher performance than interpretation by reducing the expansion factor of guest to host instructions and allowing optimizations within

translation blocks. Nevertheless, binary translation still has considerable overhead compared to native execution. Rosenblum et al. [223] report a slowdown of 5x to 10x for their MIPS R4000-based SGI processor translation in SimOS. Similar results have been published for the MIPS R4000 emulation in Embra [281]. For a CISC architecture like x86, the translation overhead is even higher. For example, running a Linux kernel build with binary translation in QEMU 2.6.5 [38] incurs a slowdown of 20x compared to native execution[3].

Direct Execution

To improve processor virtualization performance, we can (1) generate more efficient translation results and (2) further reduce the overhead that the translation and the VMM impose. With the restriction that the guest ISA must match the host ISA, *direct execution* aims at running a dominant subset of the virtual processor's instructions unmodified on the host processor. This minimizes translation costs and at the same time reduces the expansion factor from guest to host instructions.

In direct execution mode, SimOS incurs a slowdown of 2x [223] compared to native execution. Adams and Agesen [20] even report an average slowdown of only 4% for computation-heavy user-mode code in VMware Workstation [49], while the VM achieves 67% of the native speed when building the Linux kernel.

The key challenge with this technique is to identify which guest instructions can be executed directly and upon which instructions the VMM must enforce emulation. In this course, Popek and Goldberg [206] studied the requirements for a platform to be virtualizable using direct execution. They identified two important instruction groups:

1. **Control sensitive instructions** affect the configuration of resources in the machine such as changing the CPU's privilege level and altering the memory configuration, for instance, switching to a different virtual address space (i.e., writing `CR3` on x86).
2. **Behavior sensitive instructions** do not attempt to change the configuration of resources but their behavior or result depends on it. An example is the `POPF` instruction on x86 which overwrites the flags register with a word from the stack. In user mode the instruction elides the interrupt enable flag (`IF`), whereas in kernel mode the flag is applied. `POPF`'s behavior thus depends on the CPU's current privilege level.

A VMM can virtualize an ISA with direct execution if sensitive instructions are privileged – i.e., they trap when executed in user mode [206], causing a VM fault. A hypervisor can leverage this fact by running guest code physically always in

[3] Phoronix Test Suite 5.2.1 [11], Intel Xeon E5-2630v3 @ 2.40GHz, single-core VM, 4 GiB RAM. Native compilation restricted with `NUM_CPU_JOBS=1` and pinning the benchmark to core 0.

user mode – a technique called *de-privileging* [20]. Under the assumption that the VMM implements proper memory virtualization (§ 2.2.2) the hypervisor gains control whenever the guest attempts to access the hardware configuration or performs operations that would otherwise break virtualization by revealing the true state of virtualized hardware resources. The VMM tracks the guest's mode of operation (i.e., user or kernel) with a shadow bit in the vCPU structure. Similar to the dispatch routines employed in interpreters, the VM fault leads to the call of a handler which emulates the specific instruction, respecting the shadowed mode bit. In addition, the hypervisor registers exception handlers to capture all exceptions that the direct execution of guest instructions may raise. The concept is known as classic virtualization or more figuratively *trap-and-emulate* [20, 51, 103].

A popular VMM based on trap-and-emulate was CP-67/CMS [172] which implemented a time-sharing system on the IBM System/360 Model 67 for transparent multi-user access. Similarly, Disco [48] used VMs based on trap-and-emulate to port commodity operating systems designed for uniprocessors to large-scale shared-memory multiprocessors by running multiple copies of the OS in parallel.

Both, Model 67 and the MIPS R10000 RISC processor required by Disco, are predestined to direct execution because all sensitive instructions are privileged and trap in the hypervisor. Unfortunately, this is not the case for all architectures. A comprehensive study of the x86 instruction set by Robin and Irvine [126] revealed 17 *critical instructions* – i.e., sensitive instructions that do not trap – one being the `POPF` instruction already mentioned. This makes x86 not classically virtualizable. The same is true for ARM [203].

With direct execution, the VM always executes in physical user mode. Therefore, running user-mode guest code is not a problem since the processor runs at the expected privilege level and instructions behave identically to native execution [22]. This is not the case if the vCPU enters kernel mode because the CPU remains in physical user mode. In consequence, critical instructions may behave unexpectedly. Kernel-mode code must thus be virtualized as follows:

(a) **Identical Binary Translation** A possible solution is to dynamically switch between direct execution and binary translation based on the current state of the virtual CPU [49]. As a direct execution engine is not capable of cross-ISA virtualization, DBT can be optimized for this scenario, primarily generating identical translations [20, 49]. However, this requires a rather complicated use of segment registers to enable execution from within the code cache while at the same time allowing unmodified memory accesses [49].

(b) **Paravirtualization** A simpler solution is to statically replace occurrences of critical instructions in the source code of the guest operating system with explicit invocations of the hypervisor [35,46,82,278]. An obvious drawback to this technique is that the guest operating system needs to be modified, which in turn requires access to the source code; an obstacle for commercial operating systems.

Since trapping is costly, taking 2030 cycles[4] for the `rdtsc` instruction on a Pentium 4 [20], it is also an option to selectively use binary translation to speed up operations that can architecturally be trapped. Adams and Agesen report 1254 cycles for DBT with helper routines and 216 cycles for in-TB emulation [20]. The technique has therefore been adopted by various VMMs [49, 75, 223]. Likewise, paravirtualization is not just limited to virtualizing critical instructions. The Denali Isolation Kernel [278] leverages this approach to extend the ISA with purely virtual registers and instructions, making certain operations such as idling in the guest OS more efficient. Disco [48] uses stores and loads on special predefined addresses to optimize (de-)activation of interrupts and access to privileged registers. Paravirtualization is also commonly used to reduce I/O virtualization overhead (§ 2.2.3).

To sum up, we can conclude that direct execution significantly improves virtualization speed, making it considerably faster than dynamic binary translation; even for platforms that are not classically virtualizable. On the flip side, direct execution limits virtualization to the host architecture. Furthermore, it is inherently less suited for research than interpretation or DBT because it does not allow easy instrumentation. It thus provides little information about the workload's performance or behavior [223]. Using direct execution for x86 requires the combination with binary translation or paravirtualization, which makes it rather complicated or requires extensive modification of the guest operating system [46].

Hardware-Assisted Virtualization (HAV)

With the addition of special virtualization features to the instruction set architecture, it is possible to efficiently virtualize architectures which are not classically virtualizable. Hardware-assisted virtualization (HAV) thus improves upon direct execution in that it does not require binary translation or paravirtualization for kernel-mode code. But even for architectures that are virtualizable with pure trap-and-emulate, HAV can simplify the VMM design by providing an explicit interface for virtualization and taking away some of the burdens from VMM developers such as manually shadowing processor data structures. Prominent examples of HAV extensions are Intel VT-x [125] and AMD SVM [21] for the x86 platform, and TrustZone with Hyp mode for ARM [32].

[4] `rdtsc` takes approximately 80 cylces when executed natively on a Pentium 4 [92].

> Since this thesis is centered on the Intel x86 platform, we will restrict the following description to the processor virtualization in VT-x and refer to this particular technology if not stated otherwise. In accordance with the Intel Software Developer's Manual [125], we will further use the term *virtual machine extensions (VMX)* for the processor and memory virtualization capabilities. Despite the focus on VMX, most of the concepts laid out are equally applicable to other hardware virtualization extensions.

With direct execution the guest always runs in user mode, thereby enabling the VMM to trap privileged operations by the guest operating system. HAV considerably facilitates virtualization by letting the guest OS run at its intended privilege level. To that end, VMX introduces a new mode of operation which is more privileged than kernel mode: *VMX root mode*. This mode enables the hypervisor to enforce restrictions upon on the guest operating system running in kernel mode, just like an OS controls processes in user mode.

Figure 2.10 illustrates the basic execution cycle. The hypervisor runs in VMX root mode. By calling the `VMLAUNCH` instruction the CPU switches to VMX non-root mode, thereby starting to execute guest code. If the CPU encounters an exit condition, it jumps back into the hypervisor. VMX defines 65 exit reasons in total [125], covering vectored events like interrupts and exceptions as well as the attempted execution of a privileged instruction in the guest, for instance, reading or writing a model-specific register (MSR). Unlike with interrupt handlers,

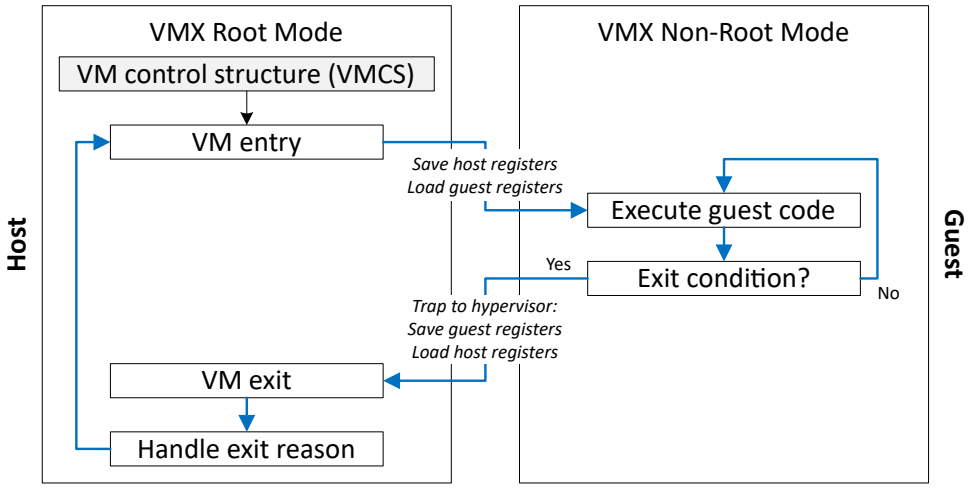


Figure 2.10: The VMM runs in VMX root mode. A VM entry brings the processor into VMX non-root mode, thereby switching to the guest. If the CPU encounters an exit condition (e.g., a privileged instruction), the CPU leaves the VMX non-root mode, giving the hypervisor the chance to handle the exit reason. The VM control structure (VMCS) stores the guest and host CPU states and controls VM execution (figure based on [262]).

a VM exit does not transfer control to a specialized dispatch routine, but instead always continues after the `VMLAUNCH` instruction. Entering and leaving a VM is thus comparable to a classic context switch between different processes in an operating system. After gaining control, the VMM can retrieve information about the nature of the exit and perform appropriate steps to handle it. The cycle starts anew with an invocation of the `VMRESUME` instruction.

Eminent in this scheme is the virtual machine control structure (VMCS). The VMCS is organized into four logical groups [125]:

- **Guest State Area** The processor's state is saved into the guest state area on VM exists and loaded from there on VM entries. The VMM can freely modify this data to, for example, emulate the effects of a privileged instruction.

- **Host State Area** The processor's state is saved into the host state area on VM entry and loaded from there on VM exit.

- **VMX Control Fields** These fields control the processor's behavior in VMX non-root mode. The fields can be divided into execution, entry, and exit control. This way, the hypervisor can, for instance, configure what conditions cause a VM exit.

- **VM Exit Information** To handle a VM exit, the VMM must be able to discern the precise reason for the transition. The exit information comprises a specifically encoded integer value and an additional exit qualification whose format depends on the cause of the exit.

The performance of hardware-assisted virtualization heavily depends on the cost and the frequency of transitions between the hypervisor and the guest. An initial comparison between software virtualization and HAV in 2006 showed mixed results [20]. The main focus of research since then has therefore been to minimize transition frequency and round-trip cost. While the Prescott CPU still required 4000 cycles per round-trip, the number has been considerably reduced to 540 cycles on the more recent Haswell processor [23, 233]. The use of tagged TLBs and extensive address translation caching can further dampen the overhead of the address space switch on VM exits [171, 288]. At the same time, various techniques such as second level address translation have been developed which successfully reduce the exit rate [23, 259]. Today, hardware-assisted virtualization thus achieves near native performance which makes it the prevalent processor virtualization technology in widespread productive use.

Since the fundamental concept with HAV is still trap-and-emulate, the same restrictions in research and development concerning the inability of easy instrumentation apply.

2.2.2 Memory Virtualization

Memory management is one of the fundamental tasks of an operating system. This includes managing the physical memory, maintaining address translation structures, configuring mappings, and installing the correct memory configuration on a context switch. The operating system performs all these actions under the assumption to have full control over the machines physical memory and configuration. In a virtual machine, this is a false assumption as the hypervisor claims ownership of physical resources to be able to distribute or share them between multiple VMs. In addition, it has to reserve some of the resources for its own use, effectively hiding them from the virtual machines. Just as with processors, the VMM thus has to virtualize the memory in the host so as to give each guest the impression of full control.

In order to comprehend the approaches to memory virtualization relevant for this thesis, we first briefly recapitulate memory in a non-virtualized environment.

The predominant way of providing access to physical memory in computers today is by means of virtual memory [94]. When a process uses a memory address the CPU interprets it as a virtual address and the *memory management unit (MMU)* transparently translates the address to the corresponding physical address at run time – if possible, faulting into the operating system otherwise. Every process has its own set of virtual addresses. In their entirety, they form the *virtual address space* of the process. Depending on the architectural support, various methods exist to implement address spaces. Examples are base and limit registers, segmentation, and paging [254, p. 181ff], with paging being the prevalent technique.

In a page-based architecture, the MMU translates addresses in the granularity of *pages*, with 4 KiB being a common page size. Each address space possesses a distinct *page table* which maps virtual to physical pages [254, p. 195ff]. Each *page table entry (PTE)* describes the mapping of a single virtual page. Typical information in a PTE include [125]:

- **Physical Frame Number (PFN)** This field provides the index of the target physical page.
- **Present (P)** This bit determines if the mapping is valid. If not set, the MMU raises a page fault when the CPU accesses the corresponding virtual page.
- **Protection (RWX)** These bits specify access permissions such as read (R), write (W), and execute (X). A prohibited access causes a page fault.
- **Accessed (A) and Dirty (D)** The A/D-bits indicate page usage. The MMU sets the accessed bit when it uses the PTE for translation. If the memory access is a write, the MMU also sets the dirty bit.

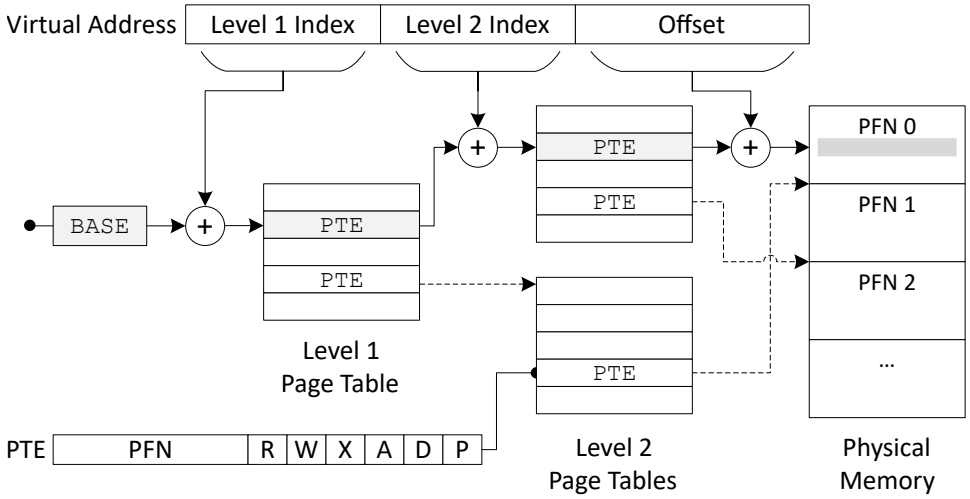


Figure 2.11: The virtual address is split into indices into a two-level page table and an offset into the physical page. PTEs point to the PFN of the next page table or to the final physical page, depending on the level.

The virtual address space of a process is usually populated sparsely. In consequence, it is a waste of memory to organize a page table as a linear list of PTEs. Most entries will not be used as they do not map to physical memory anyway – i.e., the present bit is not set. Many architectures such as x86, ARM, and MIPS rather structure a page table as a multi-level hierarchy of page tables as exemplarily depicted in Figure 2.11. The MMU splits the virtual address into a set of indices into page tables (e.g., two on 32-bit x86) and an offset into the target physical page. PTEs in higher levels point to the physical page that contains the next page table, whereas PTEs on the last level point to the target physical page. Since entries in higher levels can be invalid not all page tables need to be allocated, efficiently reflecting the sparse population of the virtual address space.

As with regular virtual memory, the core idea to virtualize a computer's memory architecture is to introduce a level of address indirection, in this case between the guest and the host. This enables the VMM to freely organize the memory of VMs in the same way as virtual memory gives operating systems the ability to freely assign physical memory to processes. The result is that both the process and the VM get the illusion of having full control of the system's entire memory. We can describe V, f, and h as follows:

Virtual State Description (V) V holds the state of each of the pages in the guest's physical address space.

Resource Mapping (f) f describes the mapping of guest physical pages to host memory and (virtual) devices.

Host State Transfer Function (h) h translates reads and writes on guest physical pages to equivalent operations on the respective host memory.

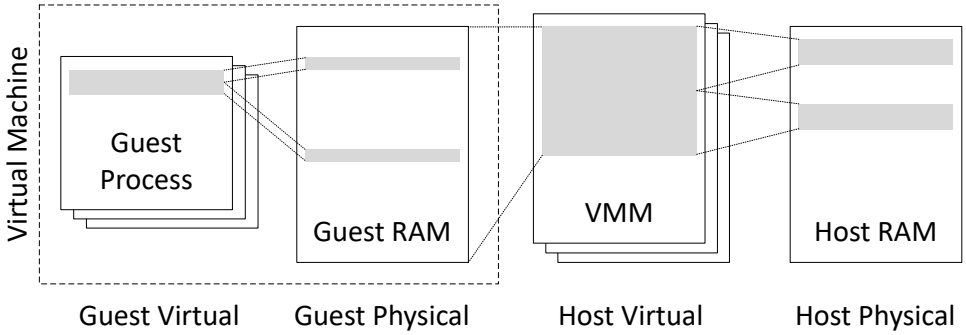


Figure 2.12: Virtualizing a virtual memory architecture creates four levels of memory addresses, requiring three stages of translation.

If the guest supports virtual memory, the VMM effectively has to cope with nested memory virtualization. This gives us a total of four address levels (see Figure 2.12): (1) *guest virtual address (GVA)*, (2) *guest physical address (GPA)*, (3) *host virtual address (HVA)*, (4) *host physical address (HPA)*.

Goldberg [102] explicitly formalized this additional step in order to stress the difference between the fault condition $f(v) = \phi$ in the translation from GVAs to GPAs and GPAs to HVAs. A fault on a GVA is a *guest page fault* and should eventually lead to a call of the page fault handler in the guest OS. In contrast, a fault on a GPA is a *VMM page fault* and should transfer control to the hypervisor.

Depending on the design of the VMM, the HVA and HPA can collapse into a single address level. A type I hypervisor, for instance, may choose to directly map guest physical memory to host physical memory, whereas a type II hypervisor may represent the guest physical memory as an area of regular anonymous virtual memory in an address space supplied by the host OS. In the latter case, the translation from a GPA to an HVA can be as simple as adding an offset. At the same time the VMM delegates the assignment of guest physical memory to host physical memory to the host OS.

The actual method for implementing memory virtualization is tightly connected to the technique used for processor virtualization. Considering the frequency of memory accesses, efficient memory virtualization is at the same time crucial to the overall performance [20, 55, 75]. In the following, we describe three standard approaches to memory virtualization that are relevant to this thesis.

> We restrain the description to page-based virtual memory because this model is ubiquitous in modern architectures. Every approach thus translates (1) from GVAs to GPAs and (2) from GPAs to *host addresses (HAs)* – i.e., either HVAs or HPAs. We further assume a forward multi-level page table hierarchy as shown in Figure 2.11 for GVA to GPA translation and ignore additional levels of, for example, segmentation like in x86.

Software MMU

If processor virtualization is done via interpretation or binary translation, a natural and very flexible way to achieve memory virtualization is to not only emulate the CPU in software but also the memory management unit – i.e., construct a software MMU [38, 260, 281]. The interpreter or DBT engine explicitly calls the software MMU for every memory access in the guest. The software MMU then translates the guest virtual address to a host address and performs the actual memory access on host memory.

In order to translate a GVA to a GPA, the software MMU must walk the page table hierarchy of the guest. Considering that every instruction fetch is a memory access and on average every third instruction performs a data memory access [55, 121, 243, 281], performing a page table walk in each of these cases would be very expensive. Remember that every page table walk itself requires multiple accesses to guest physical memory, triggering (possibly multi-stage) translations from GPAs to HPAs. It is thus vital for the performance to avoid page table walks where possible by caching address translations. In a physical machine, the translation lookaside buffer (TLB) is responsible for this task, reaching a cache hit rate of over 99% [201, p. 439], depending on the workload. Software MMUs consequently copy this design by integrating a software TLB [165, 260].

Embra [281] implements a software TLB with a fixed-size 4 MiB table that maps every possible page in a 32-bit guest virtual address space. On a memory access, Embra retrieves TLB information from the table using the virtual page number of the GVA and performs permission checks afterward. If the entry is not valid or permissions are insufficient, the software MMU calls a helper routine to emulate the guest page fault exception. Otherwise, Embra combines the HVA from the TLB entry with the page offset bits of the memory address and initiates the load or store on the respective location in host memory.

While the approach conceptually offers a 100% hit rate and very fast access through 1:1-mapping, it is not suited for modeling 64-bit virtual address spaces due to the resulting table size. However, while in hardware a limited-size but fully-associative or n-way cache can be installed, comparing the tags of multiple cache lines in software is expensive [256]. A software TLB is thus often a limited-size direct-mapped table which uses the page number modulo the TLB size for line selection [38, 260] – a tradeoff between speed, size, and hit rate[5].

To optimize for the common case, binary translators inline the whole TLB hit path, exiting the TB only on a TLB miss (see Figure 2.13). The TLB miss handler is then a helper routine which performs the walk of the guest page table and updates the respective TLB entry. If successful, the helper routine returns to the translation

[5] Average software TLB miss rate in QEMU ranges from 3% to 8% [256].

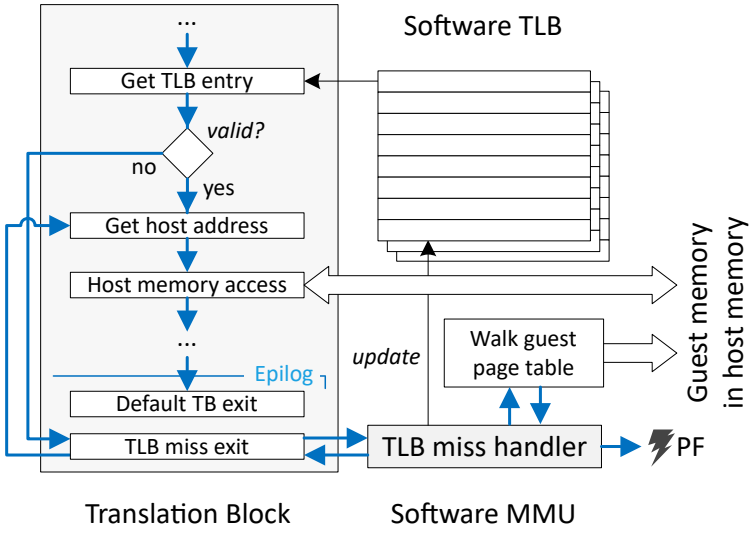


Figure 2.13: The software TLB is a direct-mapped cache for fast address translation. A valid entry contains the translated host address for a GVA. While the VMM inlines the common case of a TLB hit, it implements the TLB miss path with a helper routine. Permission checks in the TLB hit path are avoided by creating separate TLBs for reads, writes, and instructions fetches.

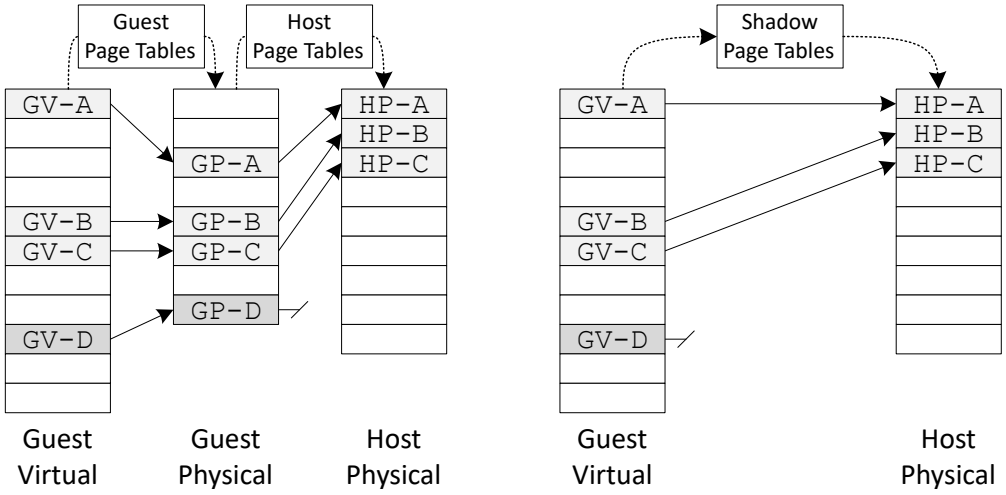
block afterward and the CPU can access the desired memory location via the host address. Otherwise, the handler exits the TB with a page fault (PF) exception.

An additional optimization is to create separate TLBs for reads, writes, and instruction fetches, each for kernel and user mode [165, 260]. This obviates the need for an explicit permission check in the TLB hit path. Instead, finding a valid entry in the corresponding TLB equates to also having the required access permission.

Nevertheless, the flexibility and ease of instrumentation offered by a software MMU still come at a high cost. Between 23% and 43% of total execution time in a binary translator are attributed to operations in the software MMU [55, 256].

Shadow Page Tables (SPT)

Whenever the processor virtualization is designed for direct execution, memory accesses must also be directly executable by the CPU. A software MMU is thus not an option because it requires explicit invocation. Instead, the CPU has to be configured to natively translate guest virtual addresses. However, activating the current guest page table would break memory virtualization because it does not reproduce the translation from GPAs to HAs and it could give the guest access to arbitrary host memory.



(a) Actual Mapping (b) Shadow Page Tables

Figure 2.14: A shadow page table (SPT) merges a guest page table with the per-VM host page table and can be directly used by the hardware MMU. Changes must be carefully tracked to keep the SPT up-to-date.

Shadow page tables (SPT) [20] solve this conflict by merging guest page tables with translations from guest physical memory to host physical memory. The concept is illustrated in Figure 2.14. Without shadow page tables, the actual mapping from GVAs to HPAs is accomplished with two page tables[6]:

1. The **guest page table** translates from guest virtual to guest physical memory. It is managed by the guest operating system. There is one such page table per guest virtual address space (i.e., usually per guest process).

2. The **host page table** translates from guest physical to host physical memory. It is managed by the hypervisor. There is one such page table per virtual machine.

Whenever the guest attempts to install a page table – i.e., set the vCPU's CR3 register on x86 – the VMM intercepts the configuration and creates a shadow page table. To that end, the VMM reads the guest page table and creates for each PTE a shadow page table entry (SPTE) which directly maps to host physical memory. This essentially eliminates the additional level of indirection. The SPTE is set to reflect the intersection of the permissions in the source page tables. The VMM can then install the shadow page table instead of the guest page table and allow the CPU direct execution. On a page fault, the VMM has to trace back the cause to either the guest or the host page table and invoke the appropriate page fault handler.

[6]For simplicity reasons, we assume a direct mapping from guest to host physical memory. If the VMM first translates GPAs to HVAs, the shadow page table must also integrate this step.

To maintain shadow page table coherency, the VMM is forced to track all changes to the source page tables and propagate these to the SPT. Since the host page tables are managed by the hypervisor, modifications to these can be easily forwarded to the SPT. The guest page tables, however, can be modified by the guest OS at any time without the VMM taking notice. In practice, three approaches for detecting changes to guest page tables have emerged [18, 35, 49, 240]:

(a) **Trap TLB Invalidations** For a PTE change to become apparent to the hardware, an operating system has to make sure that the TLB does not contain a stale copy of the PTE by invalidating the TLB entry or flushing the whole TLB. A VMM can trap these operations and synchronize the SPT with the guest page table.

(b) **Write Protection** The VMM write-protects all guest physical pages in the host page table that contain guest page tables. This way, the write protection is mirrored in the shadow page table and attempts by the guest to modify the guest page table lead to a VMM page fault. The hypervisor can then emulate the modification and propagate it to the SPT. In contrast to the first approach, this method allows to also track changes to guest page tables that are currently not active and for which the guest OS will not invalidate the TLB. Consequently, the hypervisor can cache and reuse SPTs.

(c) **Paravirtualization** The guest OS explicitly informs the hypervisor of modifications with the help of hypercalls. This approach is much simpler and faster but requires the guest to be adapted for memory virtualization.

An additional challenge with SPTs concerns the coherency of the accessed and dirty bits in the guest page table. When the MMU uses a PTE for address translation it sets the A/D-bits as appropriate. With shadow page tables, the MMU updates the A/D-bits in the SPT instead of the guest page table. The VMM thus has to propagate the A/D-bits back to the guest page table so that page usage information is available to the guest OS.

Although shadow page tables have originally been developed for the use in same-ISA virtualization, there are also research projects that have successfully applied the technique to cross-ISA virtualization, which typically suffers from the slow performance of a software MMU [55, 270]. However, since the approach is based on the premise that the host virtual address space can accommodate the guest and provide extra space for the DBT engine and data, a working prototype has only been demonstrated for 32-bit guests on 64-bit hosts.

While shadow page tables allow direct execution of guest instructions, they are still a major performance bottleneck. Agesen et al. state that for many workloads SPTs are responsible for as much as 90% of the guest exits with HAV [23]. This is because page faults always trap into the hypervisor even if they can and have to be handled by the guest OS. In an early study, Adams and Agesen even found

that for workloads that perform much I/O, create processes, or switch contexts rapidly, pure software virtualization with DBT and software MMU outperforms HAV with SPTs [20]. The authors attribute this to the high exit rate and PTE synchronization costs of SPTs.

Second Level Address Translation (SLAT)

The primary challenge with memory virtualization is that the hardware does not natively support the additional level of address indirection between the guest and the host. Software MMUs solve this by leaving the hardware MMU out of the virtualization process, performing the translation of guest virtual addresses entirely in software. Shadow page tables move the address translation from GPAs to HPAs to the construction time of the page tables, allowing final address translation in hardware. However, the reduction step induces high overhead due to page faults and the need for constant synchronization.

The focus of CPU manufactures has therefore been to improve memory virtualization performance for hardware-assisted virtualization by supporting two levels of address translation natively [21, 32, 125]. Figure 2.15 exemplarily illustrates the concept. In addition to the guest page table, the VMM configures a host page table which translates from GPAs to HPAs. Intel refers to this page table as *extended page table (EPT)* [125]. To make the EPT visible to the hardware the hypervisor

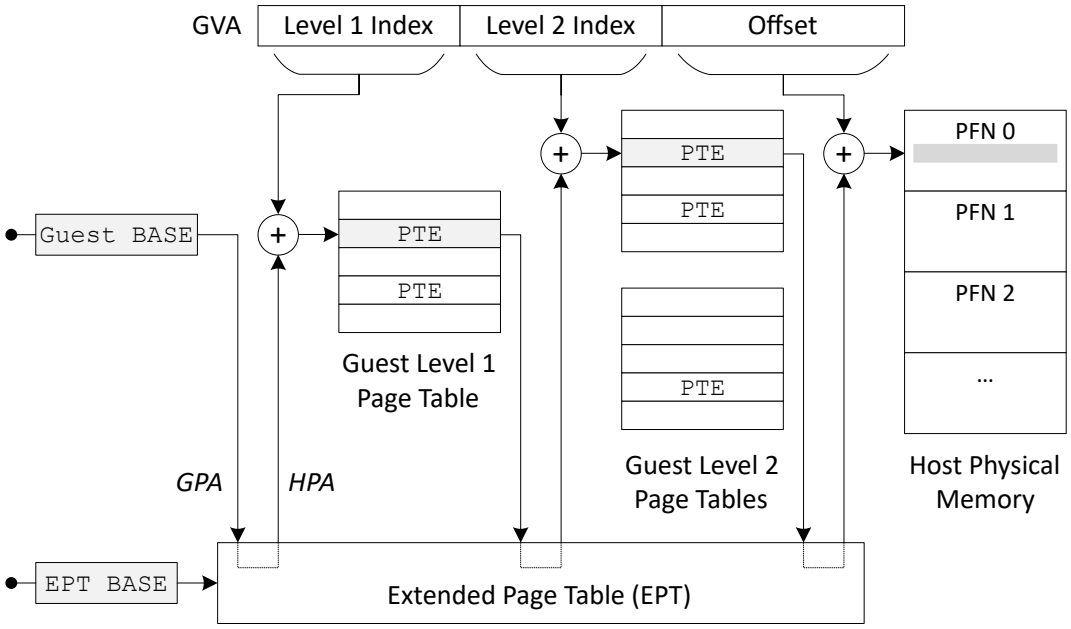


Figure 2.15: The hardware MMU natively supports two levels of address translation. For every guest physical address, the MMU performs a walk of the extended page table (EPT), retrieving the host physical address. A single guest page table walk thus requires multiple EPT walks.

configures the EPT base pointer in the virtual machine's VMCS, while the guest OS is free to install the guest page table using the conventional control register (e.g., CR3). Whenever the guest then uses a GVA, the hardware MMU translates the address to an HPA without intervention from the VMM. Since each of the levels in the guest page table hierarchy uses guest physical addresses, the hardware MMU has to walk the EPT multiple times per guest page table walk[7]. Modern processors therefore incorporate a plethora of page table structure caches to successfully mitigate the cost of TLB misses in most cases [171].

A major improvement of EPTs over SPTs is that the hardware MMU is capable of automatically vectoring page faults to the correct page fault handler, depending on whether the fault is caused by a violation in the guest page table or the extended page table – i.e., an *EPT violation*. This drastically reduces the VM exit rate [23], effectively eliminating VMM interposition from the management of guest page tables. At the same time, the EPT allows the hypervisor to configure protections on guest physical pages to, for example, transparently track dirty pages [72] in a guest-agnostic way.

Due to the high degree of performance, ease of use, and the flexibility of using the EPT to transparently implement advanced VMM features, second level address translation has become the standard method for memory virtualization in productive use. Just like hardware-assisted virtualization, the tight hardware integration, however, makes SLAT less suited for research and development as memory accesses cannot be easily instrumented [233].

2.2.3 I/O Virtualization

Besides the virtualization of processors and main memory, the virtualization of I/O devices is the third major pillar of every virtual machine. The decoupling of the logical from the physical device enables time and space multiplexing as well as flexible mapping of I/O resources. The hypervisor has to handle the full range of devices making up or connecting to a computer such as disks, network adapters, and human interface devices but also interconnects like PCI. This is where a great amount of the complexity in system VMs originates from compared to process virtual machines. The complexity is rooted in the diversity of device types and the ways these devices are interfaced. At the same time, some devices such as graphics adapters or high-speed network interfaces have high performance requirements, requiring a low virtualization overhead (e.g., latency) to be effectively used in a virtual machine. As a result, the type of device eventually determines the method of how it can be practically virtualized and presented to a virtual machine (i.e., V, f, and h).

[7] Contrary to what is shown in Figure 2.15, EPTs are only supported for 64-bit address spaces and consequently have four instead of just two levels.

To get a better grasp on the spectrum of I/O devices in a VM, we can classify devices into the following three categories [240, p. 404f]:

- **Dedicated Devices** This class comprises devices that are solely dedicated to a certain virtual machine. The VMM does not need to employ a mechanism for sharing such a device with other virtual machines or the host. A dedicated device may be entirely virtual (e.g., a network adapter for a virtual network between VMs) or it may correspond to a particular physical device. In the latter case, I/O requests can bypass the hypervisor, allowing performance comparable to native operation.

- **Partitioned Devices** For a partitioned device, the VMM dedicates a certain amount of a device's resources to a particular virtual machine. The hypervisor may, for instance, split the storage space of a disk into several partitions and assign them to VMs. In contrast to a fully dedicated device, the VMM has to intercept I/O requests so that they target the right partition.

- **Shared Devices** Some devices can conveniently process requests from multiple independent sources. Network adapters are an example for this. While each virtual machine gets its own virtual network adapter, the physical network adapter in the host is shared on a per-request basis, with the hypervisor routing packets between VMs and the physical device. A subclass of shared devices are spooled devices such as printers.

With the hypervisor being in the interposition between the hardware on the one side and the virtual machine on the other side, the hypervisor has to (1) control the physical devices (back end), (2) construct a virtual replicate of each device and present it to the VM (front end), and (3) translate requests between these two endpoints.

A type II VMM relieves itself from the necessity to actually drive the physical devices in the back end as it employs a host operating system for this task. This way, it can build on the abstractions exported by the OS. An intuitive example is the use of the host OS's file system. The hypervisor represents the virtual disk as a regular file instead of using whole disks or disk partitions which it has to manage on its own [49]. While this makes type II hypervisors more flexible and increases the range of supported hardware, I/O requests have to traverse additional layers of abstraction. Consequently, I/O performance can be worse than in type I hypervisors, which interface physical devices directly [222,247,265]. The control of physical devices for entirely virtual devices is obviously not required. However, this does not obviate the need for a back end component; a virtual network adapter, for instance, still has to transmit packets across virtual machine boundaries to other adapters.

The virtual representation of a device in the front end can be constructed so that it complies with the hardware interface specification – i.e., it faithfully mirrors

all device registers and device memory. The guest operating system then uses the same mechanisms to communicate with the device as on a physical machine, namely *memory-mapped I/O (MMIO)* and *port-mapped I/O (PMIO)*:

The physical address space in a real computer is not solely determined by main memory, but merely represents a composition of RAM and various devices' memory and registers (e.g., graphics memory, registers of controllers, etc.). Thus, in addition to what is generally perceived as virtual memory, another level of memory virtualization can be found at the layer of the physical address. Communicating with memory-mapped devices then boils down to ordinary reads and writes in MMIO regions. PMIO, in contrast, often uses specific instructions for performing I/O (e.g., the `IN` and `OUT` instructions on x86 [125]) and targets a different address space than the general physical one. In any case, the device may report events to the operating system by setting a specific result register on which the CPU has to poll, or it fires an interrupt.

A virtual machine replicating the hardware interface thus has to support MMIO and PMIO as well as interrupts, including the implementation of a virtual interrupt controller. Whenever the vCPU accesses an MMIO or PMIO address the VM has to trap into the hypervisor. For a hardware-assisted virtual machine, this can be accomplished by invalidating the pages of MMIO regions in the EPT so that accesses cause a VMM fault. In addition, the VMCS is configured to trap PMIO. A binary translator, on the other hand, can inspect the physical address at run time and route the access appropriately. PMIO instructions are translated to invocations of corresponding helper routines. This way, the request eventually reaches the back end driver, where it interacts with the respective physical device or the host OS (e.g., to perform a read in a virtual disk file). The VMM signals the completion of the I/O request to the VM by changing the value of a device register in the front end and by potentially injecting a virtual interrupt into the VM.

While this approach makes the virtual device compatible with native drivers in an unmodified guest OS, a hardware-compliant device model is rather complex to implement. Furthermore, because the hardware interface has generally not been designed to support efficient virtualization, there is considerable overhead in I/O operations. The legacy IDE interface, for example, uses 8-bit port I/O to communicate with the disk controller, requiring repeated traps into the VMM for transmitting the sector number, buffer address, and length [265].

An alternative is to implement a paravirtualized device model where the front end uses a hypercall-based shared-memory interface, optimized for efficient I/O virtualization [35]. Special guest drivers add the support for such an interface without the need for adaptation of the guest operating system [136]. Nevertheless, a generic hardware interface is usually still available to support the installation of guest operating systems and facilitate the guest OS boot phase, where paravirtualized drivers are not loaded yet.

A drawback of a paravirtualized interface is that it is specific to a certain hypervisor. In response to this, a standardized interface between device front ends and back ends has been published under the name Virtio [226].

Besides the selection of the device model, numerous other techniques have been developed to improve virtual machine I/O performance. Intel released a CPU feature called APICv [125, 193], which virtualizes the interrupt-related states and APIC registers in the VMCS. This removes the need to emulate the APIC in software when employing hardware-assisted processor virtualization. Similar solutions have also been introduced by other hardware vendors such as AMD [122] and ARM [73]. With single-root I/O virtualization (SR-IOV) [202] physical devices natively support virtualization by exporting virtual instances. The hypervisor can treat these as dedicated devices, thereby removing VM exists for I/O reads and writes. Direct Interrupt Delivery (DID) [259] combines SR-IOV and APIC virtualization to directly deliver interrupts to the guest, effectively eliminating VM exists due to I/O operations.

2.2.4 Conclusion and Terms

The architecture of a system virtual machine is defined by the techniques employed for the virtualization of the processor, the main memory, and I/O devices. While different virtualization approaches can be used in each field, they are strongly interwoven in practice, forming three general types of system virtual machines:

	Processor Virtualization	Memory Virtualization	I/O Virtualization	Examples
Software	Interpretation, DBT	Software MMU	Software Virtual Devices	Bochs [2], QEMU [38]
Direct Execution	Direct Execution, DBT	Shadow Page Tables	Software Virtual Devices	VMware Workstation [49]
Hardware	Hardware-assisted	Extended Page Tables	Software Virtual Devices, SR-IOV, APICv	Hyper-V [173], KVM [139]

Table 2.1: Overview of System Virtual Machines

Software techniques for system virtualization provide the best support for cross-ISA virtualization and transparent instrumentation. They therefore lend themselves to research and development, where a detailed insight into the execution of a system is required. However, the flexibility provided by software solutions comes at a high performance cost, degrading execution speed by up to multiple orders of magnitude compared to a native installation. Furthermore, although software techniques are designed to mimic hardware behavior, they always remain approximations of the true hardware implementations. Besides the change in timing, a study by Martignoni et al. [167] revealed thousands of deviations in

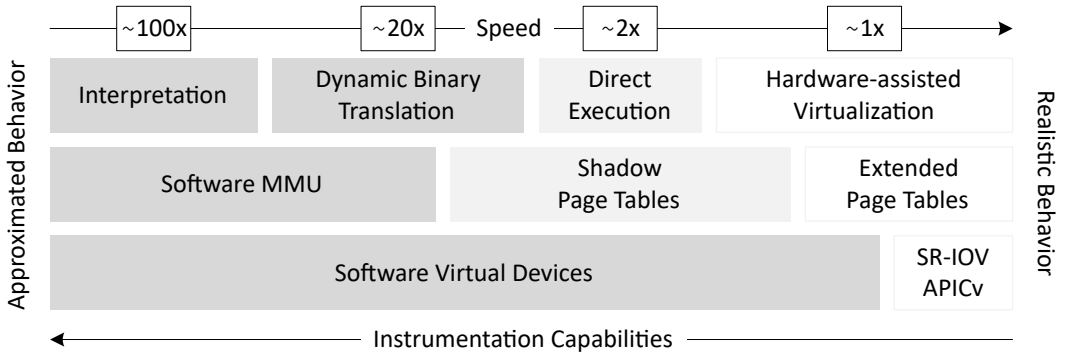


Figure 2.16: Software techniques for system virtualization provide the best support for instrumentation, but suffer from slow virtualization performance and approximated system behavior. Hardware-assisted virtualization offers the best performance and realistic behavior, but little instrumentation capabilities.

the behavior and support of instructions, exceptions, and the computation of CPU flags in popular software virtual machines. These deviations are visible to the guest and can thwart the inspection of applications in the VM [212, 287] and ultimately question the accuracy of results obtained from a software virtual machine.

Conversely, hardware-assisted virtual machines on the other end of the spectrum offer near native performance as well as realistic timing and execution behavior. However, they achieve this by removing software interposition where possible, which in turn deprives hardware virtual machines of the capability of detailed and flexible instrumentation, creating a dilemma for researchers and developers.

> In the remainder of this work, we will use the term *emulation* to denote the execution within a software virtual machine (i.e., interpretation, DBT, software MMU, etc.) and differentiate it from the execution within a hardware-assisted virtual machine (i.e., HAV, EPTs, etc.).

2.2.5 Case Study: QEMU/KVM

QEMU [38] is a popular open source type II hypervisor with frequent use in research and development. QEMU can be configured to emulate a process virtual machine or a system virtual machine with support for a wide range of guest and host architectures, including Alpha, ARM, x86, PowerPC, MIPS, and others. The system virtual machine comes with a broad set of emulated virtual devices, allowing to set up a rich execution environment. With Kernel-based Virtual Machine (KVM) [139], a Linux kernel module allowing privileged access to hardware virtualization extensions, QEMU can be seamlessly extended to run hardware-assisted system virtual machines. Selecting between the integrated

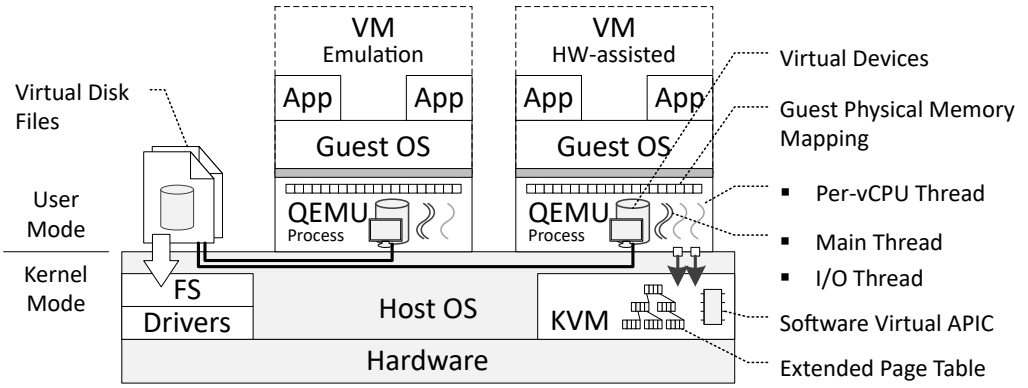


Figure 2.17: Each virtual machine receives its own independent QEMU process running on a host OS. While emulation (left) runs entirely in user mode, hardware-assisted virtualization (right) uses Kernel-based Virtual Machine (KVM). In any case, QEMU organizes and maps the guest physical memory and manages virtual devices.

emulation engine and KVM is as simple as adding the ––`enable–kvm` command line argument, which instructs QEMU to hand over a great part of processor and memory virtualization to KVM. This versatility makes QEMU particularly interesting in the scope of this thesis, where we aim to combine hardware-assisted virtualization and emulation to a fast but flexible full system analysis platform.

Being a type II hypervisor, QEMU runs as a regular user-mode process on top of a host operating system. Every instance of QEMU runs a single independent virtual machine. The lifetime of a virtual machine thus corresponds to the lifetime of the respective QEMU process. There is no central control service such as the configuration store in Xen [35], which simplifies the overall design.

QEMU processes that have hardware-assisted virtualization activated share the KVM kernel module but receive distinct instances of the kernel-level data structures representing a virtual machine. QEMU communicates with KVM through I/O controls[8]. It accesses the `/dev/kvm` device file to instantiate a new virtual machine [139]. KVM then creates additional device files to handle requests directed to a particular VM or vCPU [138].

To control the execution of the virtual machine, each QEMU process offers a command line interface called *QEMU monitor*. It can receive instructions from the user at run time, for example, to suspend or resume the VM. The monitor is driven by the main thread.

[8] An *I/O control (ioctl)* is a system call performed on special files in the operating system to trigger device-specific actions beyond the semantic of read and write [254, p. 771]. An ioctl comprises a control code, identifying the requested operation, and a set of operation-specific arguments. The kernel vectors the ioctl to the handler function of the module that created the device file.

Processor Virtualization

Latest versions of QEMU create one host thread for every virtual processor core, irrespective of whether emulation or hardware-assisted virtualization is used[9]. The virtual CPUs are thus subject to the scheduling in the host operating system, however, preemption is disabled as long as the vCPU thread is in VMX non-root mode.

The emulation mode in QEMU uses a dynamic binary translator, called *Tiny Code Generator (TCG)*. Translation blocks are based on basic blocks only. So TCG does not perform translations across jumps or branches. However, TB chaining is employed to improve emulation performance. Since QEMU supports a wide range of guest and host architectures, implementing an $n : m$ translation engine would be cumbersome. Instead, QEMU implements a two-stage pipeline which translates guest code into a RISC-like intermediate language first. In the second phase, the intermediate language is then translated to the host architecture. This effectively decouples the guest and host architectures and, in addition, creates an ISA-agnostic environment to apply optimizations and instrumentation.

QEMU is geared toward high execution performance. To achieve this, the translator makes use of various optimizations. For example, while a physical CPU computes flags (e.g., overflow, carry, zero, etc.) instantaneously, QEMU postpones flag computation to the time they are needed – for instance, in a conditional jump or when the vCPU's flags register is pushed onto the stack [38]. Another optimization is the lazy update of the instruction pointer. Instead of writing the IP after every instruction, only a single update is made at the end of a TB. As with most translators, interrupt delivery occurs at the boundary of TBs only [287].

While these and other optimizations equip QEMU with one of the fastest execution engines, this does come at a cost. QEMU lacks support for an accurate instruction counter, there is no instruction or memory tracing facility (not even an easy to use hooking facility as provided by Bochs [2]), the emulation cannot be extended with a timing model such as in Simics [163] or gem5 [41], and according to Martignoni et al. [167] QEMU suffers from numerous deviations in the execution behavior compared to a physical CPU.

When QEMU runs with KVM enabled, the entire emulation engine lies fallow and processor virtualization is handed over to KVM. For this purpose, a vCPU thread enters the KVM kernel module with an ioctl on the corresponding vCPU's device file. In the kernel, the general execution flow is comparable to what is depicted in Figure 2.10. However, if an exit from the guest cannot be handled in KVM (e.g., an MMIO access), the vCPU thread returns from the ioctl with a respective error code and leaves solving the situation to the user-mode QEMU process.

[9] The version used in this thesis employs only one host thread in emulation mode and switches between virtual CPUs based on a round-robin scheme.

Memory Virtualization

QEMU organizes the mapping of MMIO regions and RAM in the guest physical memory as a directed acyclic graph of *memory regions* (see Figure 2.18). A memory region associates a range of guest physical addresses to RAM, ROM, or MMIO. Areas that hold actual memory rather than device registers are backed by *RAM blocks*. A RAM block is an allocation of host virtual memory in the address space of the QEMU process. The `PC.RAM` block represents the guest's physical memory, allowing guest reads and writes directly from within the QEMU process with regular memory operations. This also reduces the translation from GPAs to HVAs to the addition of an offset. Further RAM blocks are allocated for the BIOS, device ROMs, and video memory. Accesses to MMIO regions are routed to handlers in the virtual devices.

The layout of the guest physical address space is managed by QEMU even if KVM is enabled. However, KVM replicates the mapping in the extended page table – or the shadow page table in legacy systems – to make the memory accessible in hardware-assisted execution. To that end, KVM mirrors each required memory region from QEMU in a data structure called *memory slot*. Since the virtual address space of QEMU and thus also the memory of the virtual machine is subject to the memory management of the host operating system, KVM provides callbacks for notifications of paging operations governed by the host OS. This way the extended page table remains coherent.

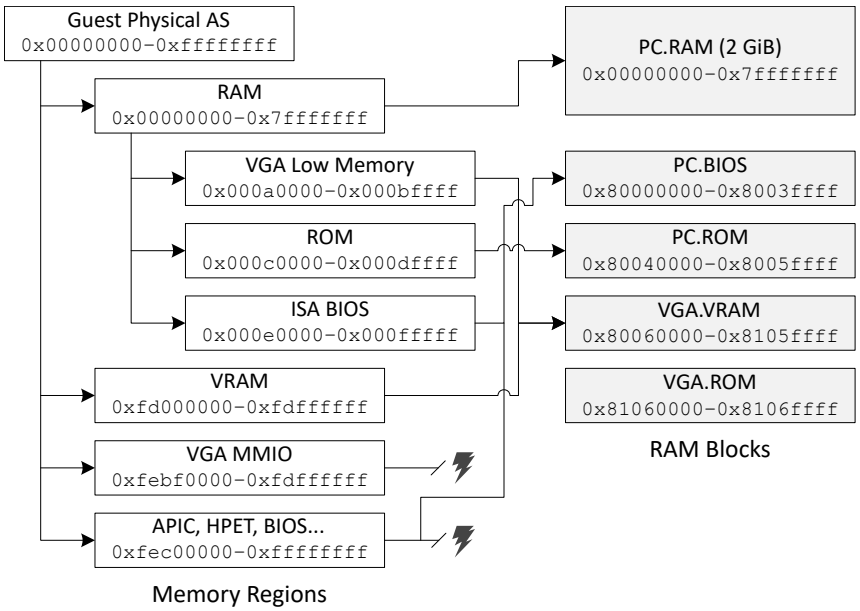


Figure 2.18: The guest physical address space is a composition of RAM, ROM, and MMIO regions. While RAM and ROM areas are backed by actual host memory through RAM blocks, MMIO regions cause a VM exit and the hypervisor routes the access to the respective virtual device.

I/O Virtualization

To asynchronously perform I/O operations, for example in the context of direct memory access (DMA), QEMU maintains a dedicated I/O thread. When a vCPU thread accesses a device register to issue an I/O operation, the access causes a VMM fault (EPT violation). The vCPU thread then exits the KVM kernel module because it is unable to handle the exit reason. In user mode, QEMU directs the execution to the virtual device which creates an asynchronous I/O request. This is, for instance, a read of the virtual disk that the device implementation translates to a corresponding file system read (see Figure 2.17). Meanwhile, the vCPU thread can continue execution of guest instructions. When the I/O operation completes, the I/O thread kicks the vCPU thread out of the guest to inject a virtual interrupt.

Since a transition from VMX root mode to user mode in VMX non-root mode is a long way, KVM provides an own implementation of some devices which are accessed very frequently – in particular the APIC. This concept has already been successfully used in other hypervisors [247].

In research and development, it is often necessary to maintain a read-only virtual disk image so that repeated experiments always start with exactly the same system configuration. QEMU supports this mode with the `-snapshot` command line argument, which redirects disk writes to a temporary memory overlay that QEMU dismisses when the VM terminates.

2.3 Checkpointing

An advantage of virtual machines is the decoupling of the system state from the physical hardware. This allows the VMM not only to monitor but also to save the live state of a virtual machine in a *checkpoint* [57, 222]. Based on this checkpoint, the VMM can restore the virtual machine at a later time, potentially even on a different host. Virtual machine checkpoints are therefore the foundation for many technologies that make virtual machines so useful in the first place. Examples are VM migration [44, 66, 189, 230, 282] to optimize the load in and across data centers or to balance the availability and cost in the IaaS spot market [236], VM replication for fault tolerance [72, 253, 261], debugging and forensics [137, 252], and fast spawning of virtual machines [146, 264]. Checkpointing is also a key technology in our method for acceleration of functional full system simulation.

To successfully restore a virtual machine a checkpoint must comprise all volatile state. This can be broken down into (1) the state of all virtual devices (e.g., CPU registers), (2) the complete contents of the guest physical memory, and (3) the contents of storage media. If the virtual machine uses a read-only disk image, the latter may be reduced to the modified sectors. This usually makes the guest physical memory the largest item [189].

The difficulty in taking a checkpoint is that the VMM has to capture all this data in a way that creates a consistent image of the virtual machine at a defined point in time. The simplest approach is to stop the VM for the duration of the checkpoint so that no modifications can occur. Afterward, the VM can resume execution. This method is known as *stop-and-copy (SnC)* [66]. An obvious drawback is the long suspension time immanent to the concept in which the virtual machine may not provide services. Taking a checkpoint with stop-and-copy can disrupt a virtual machine up to multiple seconds [66]. Kangarlou et al. [134], for example, measured 8.5 s for a small guest with only 600 MiB of RAM.

This, however, is only one metric to characterize a checkpointing solution. A more comprehensive evaluation comprises the following parameters (based on [296]):

- **Downtime** The duration in which the virtual machine is suspended to retrieve a consistent image. Services are unavailable during this phase. A downtime below 100 ms is generally not perceived by humans [178] and does not break a virtual machine's network connectivity [72].
- **Performance Degradation** The slowdown of the virtual machine caused by any asynchronous operations or maintenance of metadata for the purpose of checkpointing. This metric may be measured as a change in, for example, the run time (kernel build) or the number of transactions per second (database). This may also be more generally referred to as *probe effect* – the divergence in execution with and without checkpointing.
- **Checkpoint Time** The total time to create and save or transmit a checkpoint.
- **Checkpoint Size** The total amount of data saved or transmitted.
- **Host Overhead** The additional CPU and memory costs on the host.

Over the years, extensive research has been invested to improve the performance of checkpointing solutions (e.g., reduce the downtime) and adapt the technology to various applications. Different optimization techniques can be applied to tailor the characteristics of the checkpointing mechanism to the scenario at hand.

> In the following, we take a brief look at commonly used approaches. Although we focus on the guest physical memory, the same techniques apply to secondary block storage such as disks. For simplicity, we will refer to guest physical pages simply as pages.

2.3.1 Pre- and Post-Copy

One of the most frequent scenarios where checkpointing technology comes to action is the migration of virtual machines between different physical hosts. Since the migration should usually be transparent to users of the virtual machine, the most important metric in this scenario is the downtime.

The prevalent migration technique in hypervisors today is *pre-copy* [255]. Originally designed to migrate single processes, it has found its way into virtual machine migration [66, 189]. The core idea is to split the migration into two major phases: In the first phase, the VMM asynchronously copies the guest physical memory to the destination host. During this stage the virtual machine is still running and able to modify memory. On completion, pre-copy repeats the operation, but only saves the guest physical pages that have been modified during the last iteration[10]. The process loops for several rounds. While in the first round all pages are saved, the number of pages per iteration decreases and eventually converges to the *write working set (WWS)* [66] of the virtual machine. This is usually considerably smaller than the entire guest physical memory. The algorithm enters the second phase after a certain number of rounds has been reached or the number of modified pages falls below a threshold. The virtual machine is then suspended and the state of devices (e.g., CPUs) as well as the residual pages are synchronized. The execution can afterward be resumed on the destination machine. By migrating to a fake target on the same host, the technique can also be used to take local checkpoints of a virtual machine [54, 134, 252].

Pre-copy is effective in reducing downtime compared to stop-and-copy. However, the downtime is highly dependent on the *page modification rate* of the workload because the rate determines the number of pages in the last round. Clark et al. [66], for instance, report 60 ms for a popular game server, 210 ms for SPECweb [15], but 3.5 s for a custom memory stress tool. Nathan et al. [188] measured 28 ms for a file server, 35 ms for RUBiS [13], but 380 ms for a Linux kernel compile. Especially memory intensive applications such as HPC workloads, where memory pages are faster dirtied as they can be transferred over the network, have been reported to suffer from downtimes of multiple seconds [123]. This is even the case for VMs as small as 156 MiB of RAM [158].

Pre-copy trades a higher total migration time as well as a higher total amount of transmitted data for a lower downtime. Since it asynchronously copies and transmits memory, it can also noticeably degrade the performance of the virtual machine through memory and network contention [123]. In consequence, various pre-copy approaches apply dynamic rate-limiting [44, 66, 188] or adjust the termination criteria based on the page modification rate to initiate the stop-and-copy phase when no further reduction in downtime can be expected [123, 282]. In this process, performance models of pre-copy migration aid in predicting the non-trivial dynamics [181, 186].

Another optimization targets the order in which dirty pages are transferred. Due to spatial locality, it is likely that around dirty pages a cluster forms that will also be modified in the near future. Clark et al. [66] therefore copy pages in

[10] Dirty pages are detected by write-protecting guest physical memory. An attempt to modify a page triggers a VMM page fault which marks the page as dirty (e.g., in a dirty bitmap) and grants write permission again. See Chapter 6 for a discussion of dirty logging methods.

pseudo-random order to reduce page re-transmits. Svärd et al. [251] reorder pages by assigning page priorities, where less frequently modified pages receive higher priority and are copied first. Moghaddam et al. [181] employ a memory change probability density function. An effective alternative to reordering is to skip saving pages altogether for the iterations in which they are hot [187, 188].

Post-copy [113] immediately stops the source VM and activates a copy of the virtual machine on the destination host without having transferred the memory state first. Instead, the copy operation runs concurrently with the resumed execution in the target VM. If in this course the destination host accesses a missing page, the target VM experiences a VMM page fault and the hypervisor retrieves the respective page from the source VM.

In contrast to pre-copy, post-copy transfers every page only once, reducing total migration size. Since post-copy immediately moves the execution to the destination host, the technique is well-suited to quickly relocate VMs on sudden load peaks [114]. This, however, comes at the cost of degraded performance due to demand paging. To reduce page faults, Hines et al. propose an adaptive pre-paging of frequently accessed data [113]. A similar technique is used in VM-FlockMS [27] to facilitate an early application resume. Jettison [40] can tolerate the cost of demand paging because it is specifically designed to consolidate idle virtual machines, which intrinsically generate only a few page faults.

Since post-copy is a pure migration technique, it cannot be used to create local checkpoints.

2.3.2 Data Exclusion

Pre- and post-copy perform a major part of the checkpointing work concurrently with the execution of the virtual machine. Both approaches thus improve the downtime because they considerably reduce the amount of data that needs to be saved while the VM is suspended. This method can be extended by identifying pages in guest physical memory that do not need to be saved for a successful restore of the virtual machine in the first place. These pages can then be excluded from the checkpoint entirely to reduce the checkpoint size and thereby the downtime. To what extent data exclusion comes with an increased total checkpoint time or induces run-time overhead depends on the method to detect and track eligible pages for exclusion.

Natural targets for memory exclusion are free pages – i.e., guest physical memory that is not assigned to processes or the kernel – and the file system cache, together taking up on average between 50% and 80% of the physical memory [70, 198]. Therefore, excluding these pages from checkpoints has been extensively researched [26, 62, 70, 113, 131, 143, 180, 198].

Hines et al. [113] employ a ballooning driver in the guest operating system, which, when invoked by the checkpointing mechanism, allocates as much memory as possible, forcing the guest OS to use up all free pages and eventually evict pages from the file cache. Since these pages are allocated to the ballooning driver and hence are not used by any other component in the guest, the checkpointing mechanism can safely ignore them. A drawback of this and other paravirtualized methods [26, 117, 198] is that they require a specifically prepared guest. Geiger [132] infers the semantic of guest physical pages without guest intervention by tracking disk accesses, page faults, and page table updates[11]. Park et al. [198] detect cache pages by intercepting I/O in the hypervisor but require paravirtualization to identify free pages. A similar technique is utilized by Jo et al. [131] to leverage shared storage in live migration. The use of debugging information has also proven effective in closing the semantic gap and has been expedited by multiple research projects [62, 70, 143]. Koto et al. [143] used this method to additionally exclude empty slab caches in the Linux kernel.

Whenever the use case demands periodic checkpointing, for example, to allow rollback recovery on system failure, the checkpointing mechanism can leverage the fact that unchanged data has already been saved by a previous checkpoint. This data can consequently be excluded from the current checkpoint. The concept is known as *incremental checkpointing* [204] because each checkpoint contains only the modifications since the last one. This makes the performance of incremental checkpointing highly dependent on the workload (i.e., the modification rate) and the interval length. The technique is very similar to the iterative copy rounds in pre-copy but differs in that each pre-copy round does not represent a consistent delta of the guest's physical memory image.

King et al. [137] create an incremental checkpoint every 25 s to enable time-traveling and reverse debugging. Remus [72] and Paratus [86] utilize incremental checkpointing to periodically synchronize a backup host for high availability. The checkpointing interval in Remus is in the range of 25 ms to 100 ms, which is why it comes with a noticeable performance degradation of up to over 100%. Slightly better results have been reported for Kemari [253], which creates incremental checkpoints whenever the VM is about to send an event to an external device instead of using a fixed frequency. VM-μCheckpoint [269] keeps two incremental checkpoints in memory to protect VMs against transient failures. The authors also integrated a prediction of dirty pages to reduce the number of page faults. This combination has also been applied in VPC [161] for consistently checkpointing a cluster of virtual machines. Reported downtimes for a single VM with 512 MiB RAM range from 55 ms for an idle system up to 187 ms for Apache [1]. Unfortunately, the authors provide no information on the checkpointing frequency used

[11] While tracking page faults and page table updates does not incur any considerable cost with shadow page tables, extended page tables make this approach unattractive because guest page faults do not trap in the hypervisor.

in their experiments. Incremental checkpointing on a sub-page level has been demonstrated by Lu et al. [160], showing a reduction in checkpoint size between 2.7x and 4.5x when moving from a block size of 4 KiB to 64 bytes.

2.3.3 Data Deduplication

The concepts presented so far achieve improved checkpointing performance by asynchronous processing, reordering, and exclusion of guest physical memory. However, they remain on the structural level of the guest memory and do not leverage the potential that lies within the contents itself. Numerous studies have revealed the high amount of duplicated memory within and across virtual machines and proposed methods to deduplicate redundant pages [36, 107, 177, 179, 266, 283]. This potential can also be leveraged in checkpointing to improve the performance in the same way as the exclusion of memory does – by reducing the number of pages to be saved.

CloudNet [282] uses memory deduplication to speed up pre-copy live migration. It identifies redundant memory by hashing fixed-sized blocks[12] with SuperFastHash [16] and maintaining a synchronized FIFO cache at the source and destination hosts. If CloudNet encounters a hash match, it verifies equality with a regular memory compare (`memcmp()`) and sends a 32-bit index into the cache only instead of the actual data.

The authors of CloudNet report a duplication rate between 13% and 60%[13] for a Linux kernel compile and SPECjbb [14], respectively. Evaluation of memory deduplication in pre-copy live migration by Nathan et al. [187] has shown an average reduction of transmitted data by 17% for a number of different benchmarks. Both studies confirm only slight improvements for sub-page (e.g., 1 KiB) deduplication granularity. Nathan et al. also note the high computational overhead of 11x incurred by hashing. However, they use SHA-1 which is a 160-bit cryptographic hash function, in contrast to the simple 32-bit hash used in CloudNet[14].

Memory deduplication is most effective when multiple VMs with homogeneous configurations are considered for deduplication [27, 266]. The technique is therefore particularly popular with solutions that checkpoint or migrate virtual machine clusters [27, 79, 118, 216], showing a data reduction of 30% to 80% for certain workloads (e.g., Sysbench [17]) [79]. As with single VM deduplication, increasing the granularity delivers only moderate improvements.

[12] The authors also experimented with Rabin Fingerprints over a sliding window, but found it to be much slower without substantially improving redundancy detection rate [282]. This has also independently been confirmed by Hou et al. [118].

[13] The majority of the 60% redundant data is from all-zero pages.

[14] Hash collisions potentially decrease deduplication rate but do not lead to data corruption due to the subsequent memory comparison.

Chiang et al. [62] use virtual machine introspection to deduplicate free guest physical pages to an all-zero page, irrespective of the actual contents of the free pages. A comparable approach has been adopted by Sapuntzakis et al. [230] who actively zero free memory prior to deduplication.

A high degree of content similarity can also be found on the disk [142, 177, 179]. Deduplication is thus equally well-suited when checkpointing disk data.

2.3.4 Data Compression

The most popular approach to reducing the size of a checkpoint based on its contents is the application of a data compression algorithm such as gzip [4]. As data compression is conceptually orthogonal, it can be freely combined with other techniques and is a good baseline measure. Since memory pages generally possess a high number of zero bytes [90] as well as a high degree of similarity in non-zero words [130], the compression ratio is generally good, with gzip reducing the size by more than 60% in most migration scenarios [27, 118, 144].

As with incremental checkpointing, data compression can benefit from a checkpoint history. *Delta compression* exploits the fact that most memory pages experience only small modifications between periodic checkpoints (or copy rounds in pre-copy). Consequently, it can be more efficient to store the difference – the delta D – between the current C and last contents – the reference R – of a page. The delta then contains mostly zero bytes which can be compressed with run-length encoding (RLE). The difference is usually computed with an XOR ($\oplus$) operation:

$$R \oplus C = \begin{bmatrix} 92 & AA & 2C & 62 \\ F0 & EA & 56 & 78 \\ 35 & 5C & D6 & 3D \\ D2 & 40 & 33 & 92 \end{bmatrix} \oplus \begin{bmatrix} 92 & AA & 2C & 62 \\ F0 & \mathbf{B6} & \mathbf{03} & 78 \\ 35 & 5C & D6 & 3D \\ D2 & 40 & 33 & 92 \end{bmatrix} = \begin{bmatrix} 00 & 00 & 00 & 00 \\ 00 & \mathbf{5C} & \mathbf{55} & 00 \\ 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 \end{bmatrix} = D$$

A possible RLE of D in this example could be **05 02** 5C 55 **09** (almost 70% reduction), with the bytes in bold alternatingly encoding the number of successive zero and non-zero bytes. Decompression first unpacks the run-length encoding and then computes $C = R \oplus D$. The combination of XOR and RLE is also known as XOR*ed Binary Run-Length Encoding (XBRLE)* [108].

Delta compression found widespread use in migration and checkpointing projects [27, 100, 108, 180, 187, 213, 250, 282, 296]. Although not included in Remus [72], the authors propose to use delta compression for reducing the size of incremental checkpoints. A first evaluation employed an address-indexed LRU cache of 8192 previously saved pages and revealed an average 70% reduction in size. In a hybrid compression mode, the prototype switches to gzip if the compression ratio with delta compression falls below a certain threshold. The authors report a saving of 90%, compared to 80% for gzip alone. This suggests that delta compression

is not effective for all pages (i.e., heavily modified pages) but, nevertheless, can significantly reduce checkpoint size and notably improve compression when combined with a general-purpose compression algorithm.

Hou et al. [118] measured similar compression ratios for delta compression and gzip – delta compression even surpassing gzip with 92% versus 80% for RUBiS [13]. However, they also examined different checkpoint frequencies (from 25 ms to 5 s) and found that delta compression considerably loses effectiveness with increasing interval length, for example, falling from 60% to 20% in FFmpeg [3]. This observation is conclusive as pages accumulate more changes with larger interval lengths, thereby creating noisy deltas. Likewise, the size of the reference page cache must be chosen appropriately.

Delta compression can also be combined with data deduplication. MDD [296] uses the fingerprinting function from Difference Engine [107] to find pages with high similarity for which MDD then only saves a compressed delta. As before, identical pages can be specified with an index only. MDD hence extends pure delta compression because the reference page is addressed by content similarity instead of the GPA. Deshpande et al. [79] describe the same technique for live gang migration of virtual machines. Gerofi et al. [100] diverge from the method by utilizing a density-based hash function for similarity detection.

Other compression algorithms have also found their way into checkpointing. VMFlockMS [27] uses zlib [96]. Hacking and Hudzia [108] describe PDelta, a modified version of zdelta [257], itself being a delta compression technique based on zlib. Jin et al. [130] propose an adaptive algorithm which either compresses pages with RLE, WKdm[15] [280], or LZO [7], depending on the contents' characteristics.

2.3.5 Other Techniques

Although pre-copy can be used for taking *local* virtual machine checkpoints, in this scenario, shorter downtimes are achievable with *copy-on-write (CoW)* [254, p. 229], without the cost of increased total checkpoint time and repeated memory copies.

Instead of actually copying the guest physical memory in the downtime, the VMM only write-protects the memory in the SPT/EPT. The VM can then resume execution. Concurrently, the VMM writes the guest physical memory into the checkpoint and releases write-protection for saved pages. If the VM attempts to modify a page that has not yet been copied, the MMU triggers a VM page fault, thereby giving the hypervisor a chance to save the respective page prior

[15] WKdm is specifically designed for quickly compressing memory pages with strong word-level similarity and many zero bytes; originally developed for compressed caching of virtual memory.

modification. In contrast to pre-copy, pages are asynchronously saved after the point in time that is represented by the checkpoint. This requires only a single copy operation per page to create a consistent image. A notable drawback is the overhead caused by page faults, especially for short intervals.

CoW has been suggested in Remus [72] and included in several other research projects for memory [137, 157, 248, 269] and disk [44, 230] checkpointing. Gerofi et al. [100] combine CoW with incremental checkpointing, thus using CoW only for the pages that have been modified since the last checkpoint. However, since they use page protections to determine dirty pages, this does not reduce the number of page faults. Wang et al. [269] use the information about dirty pages from the last checkpoint to predict the dirty pages for the next interval and copy these in the downtime. While this effectively reduces page faults, it increases the downtime (not evaluated) as well as the checkpoint size (due to misprediction). Unfortunately, only little quantitative information on the performance characteristics of CoW checkpointing are present in the literature or provided measurements are sparse and seem implausible[16].

Checkpointing comprises many operations that, from a conceptual perspective, can trivially be parallelized such as copying memory or performing hashing and compression. Song et al. [241] demonstrated that multithreading can considerably improve performance for pre-copy live migration, with a reduction of total migration time of up to 10x and cut in the downtime of at least 2x. MECOM [130] successfully applies multithreading in the compression stage.

An alternative to software solutions is to develop custom hardware. Stevens et al. [246] present the design of a modified memory controller which can checkpoint the memory state directly to an SSD without relying on system software support. Dong et al. [84] propose the use of phase-change RAM as a persistent storage target for checkpoints to alleviate checkpointing overhead in future exascale systems.

2.3.6 Conclusion

Checkpoints capture a consistent state of a virtual machine, including its memory, persistent storage, and devices (e.g., CPU, video adapter, etc.). Checkpoints are the key technology to virtual machine live migration, high availability systems, and drive advanced debugging solutions. Depending on the application, different performance metrics for checkpointing such as the downtime, the checkpoint size, or the performance degradation of the virtual machine play a particularly important role. Numerous techniques from data exclusion over data compression to asynchronous processing exist, allowing to tailor the properties of a checkpointing

[16] The downtimes measured by Sun and Blough [248], for example, exceed our own results by one to two orders of magnitude.

solution to the scenario at hand. The common denominator in all approaches is to reduce the amount of data needed to be included in the checkpoint and to interleave operations with the execution of the virtual machine. In that course, some methods leverage the data-modification characteristics of the workload (e.g., incremental checkpointing), and others benefit from characteristics in the contents of the data itself (e.g., data compression). The performance of each approach is thus heavily dependent on the workload and the requirements that arise from the particular checkpointing application such as the used interval length.

2.4 Deterministic Replay

Tracking down bugs with traditional debugging techniques has become a challenging task with the advent of increasingly complex and parallel software. It is not always possible to faithfully reproduce the execution that triggers erroneous behavior. Furthermore, the simple act of debugging may mask issues which do not surface while the system is being observed – so-called Heisenbugs [104]. Security researchers are facing a similar situation with the spread of malware that stays dormant when it detects the presence of analysis tools such as a full system simulator. *Deterministic replay* aids in these situations by providing the ability to reproduce a previous execution of a supervised program or system. The replay run outputs identical results and is indistinguishable from the original run, but allows transparently altering the execution environment if needed – for example, to enable sophisticated debugging and analysis techniques. Over the years, many applications for deterministic replay have been published, including debugging [185, 220, 228], security and malware analyses [63, 64, 88, 287], live migration [157], fault tolerance [45, 231], and remote desktopping [128].

Based on the observation that program execution is per se deterministic and might only be altered due to external events, a common approach to deterministic replay is recording and replaying events that change execution in a non-deterministic way. These events can be roughly classified into three categories [215]:

- **Synchronous events** are operations that are always invoked at fixed locations in the instruction flow but might return non-deterministic results. Examples are reading I/O memory, the timestamp counter (`RDTSC`), or random numbers (`RDRAND`). During replay, logged values need to be returned.

- **Asynchronous events** are triggered by external devices and usually surface as interrupts. While they have a deterministic effect on the system (i.e., the execution of a specific interrupt handler), they appear at arbitrary positions in the instruction stream. It is hence essential to record accurate timing information and to precisely inject the events during replay.

- **Compound events** describe operations that are non-deterministic in both their timing and their effect on the system. The most prominent example is DMA. The completion time of a DMA operation, as well as the data written by the operation, is neither known in advance nor fixed between multiple repetitive runs. Incoming network traffic also falls into this category.

To accurately reproduce asynchronous and compound events, the replay system must store some form of *landmark*, which precisely (i.e., instruction-level granularity) identifies a specific point in the instruction flow at which the event occurred. This may, for example, be an instruction counter or a snapshot of the vCPU's state.

Recording non-deterministic events can introduce notable overhead and degrade the performance of the virtual machine. This is especially the case with hardware-assisted virtual machines because their high performance is achieved by avoiding software interposition where possible. However, for capturing non-deterministic events it is indispensable that the hypervisor is involved in processing these events. For this purpose, deterministic replay systems may have to deactivate hardware virtualization features such as direct interrupt delivery and force additional VM exits, for instance, to trap the execution of certain instructions.

Additional overhead can incur during the replay phase. As asynchronous events must be injected at an exact position in the instruction flow, the hypervisor must diligently observe the execution of the virtual machine and then, at the right point in time, interrupt the execution for replay. On the other side, the run time of a replay can also be shorter than the original run because the VMM can skip phases of idling and the result of I/O operations can potentially be immediately read from the recording log without having to invoke the respective virtual device.

Over the decades, many deterministic replay solutions have been developed. The following are important metrics that guide research in this area (based on [60]):

- **Probe Effect** The effect on the observed system such as the introduced slowdown. A large recording overhead greatly perturbs the timing behavior of the supervised system and in some cases can contradict the original motivation for deterministic replay (e.g., to record accurate timing information for later analysis). Many published solutions have therefore been geared toward minimizing the recording overhead and by this limiting the probe effect.

- **Log Size** The total amount of data required for the replay. Thousands of non-deterministic events shape the execution of a virtual machine every second. Compound events such as DMA reads may contain considerable amounts of data such as disk sectors or network packets. A dense encoding, possibly including compression, can therefore be beneficial, especially if the log should be transferred over a network or is persistently stored for a longer period of time.

- **Replay Slowdown** The increase in run time for the replay compared to the original run or a non-replayed execution of the inspected workload. However, since the relative timing of events in the virtual machine remains faithful during replay irrespective of its slowdown, the overhead may not be critical in itself, depending on the purpose of the deterministic replay (e.g., in an offline analysis scenario).
- **Replay Accuracy** The level of detail at which the replay is able to reproduce a faithful execution. Although it seems counter-intuitive at first, it is perfectly viable for a replay to not be a hundred percent accurate. Improper computation of CPU flags or divergent memory contents, for example, technically only become problematic when they influence the execution and thereby eventually break the replay.

Deterministic replay solutions can be classified roughly according to the abstraction level (e.g., processes vs. VMs) and virtualization technology (i.e., emulation, paravirtualization, or HAV) they are targeted at, as well as by their support for multithreading and multiprocessors [60].

> In the following, we use this classification to give an overview of deterministic replay approaches present in the literature. Since we deal with multiprocessor support in § 2.4.3 separately, the evaluation results quoted in § 2.4.1 and § 2.4.2 exclusively relate to uniprocessor solutions.

2.4.1 Homogeneous Replay

A *homogeneous replay* keeps the same execution environment between the recording and replay phase. A recording for a virtual machine executing with fast hardware-assisted virtualization is thus intended to also replay in exactly this configuration. This makes homogeneous replay very efficient because the recording can rely on the fact that the system will behave identically during replay. The CPU will, for instance, return the same identification string and feature set without extra efforts from the replay engine. This eases implementation and to some degree reduces the overall complexity.

Homogeneous replay is an attractive technology for simple debugging scenarios and has been realized in several user-space process debugging tools [127, 228]. These capture and replay at the level of system calls and signals instead of device I/O and interrupts[17]. While this method can also be used to replay an emulated

[17] A replay will always behave deterministically only at the abstraction levels *above* the recording. A process replay is thus not capable of replaying operating system internals but merely captures the input from the OS to the process. Contrary to intuition, recording at the virtual machine level – i.e., replaying a full system – incurs less overhead than process-level recording [63].

full system virtual machine by simply recording and replaying the emulator process [98], it is more efficient to embed the technology into the hypervisor itself.

Bressoud and Schneider [45] first described a deterministic replay that natively targets system virtual machines to implement fault tolerance. Their approach uses the number of executed instructions as a landmark. This allows the replay to leverage the recovery register of HP's PA-RISC processor for precise event injection because the CPU interrupts execution after the number of instructions configured in the register has been retired.

ReVirt [87] is technically very similar. It records and replays x86 system virtual machines based on UML for the purpose of intrusion analysis. In contrast to the replay system by Bressoud and Schneider, ReVirt must maintain a full log and cannot discard events that happened before a synchronization point. The authors of ReVirt therefore reduced log size by excluding data read from the hard disk as it can deterministically be re-read during replay. In addition, logs are compressed with gzip. The log growth rate in all benchmarks is below 1.5 GiB per day, a kernel build producing only 80 MiB per day. The authors report a run-time overhead of 8% for the recording and 2% for the replay (compared to the recording). ReVirt uses the CPU's instruction pointer accompanied with a branch count and the current value of the `ECX` register[18] as a landmark. ReVirt has been very popular within the research community and also has become the foundation for other replay projects [137].

ReVirt is restricted to Linux virtual machines that run with UMLinux [46], a specifically modified Linux kernel which allows virtualization with direct execution (§ 2.2.1). ReVirt is therefore not well-suited if a particular operating system kernel is required or if an arbitrary operating system should be inspected. A step toward a more generic platform is done by VMRS [71] and XenTT [50], which perform deterministic replay of unmodified but paravirtualized Linux guests in Xen.

Of particular interest for research are deterministic replay solutions that use emulation because the replay supports powerful analyses. ExecRecorder [76] uses the Bochs interpreter for malware attack analysis. Since ExecRecorder does not depend on paravirtualization, even commercial operating systems such as Windows can be faithfully replayed. Using an interpreter also simplifies the replay itself because non-determinism can easily be captured in a fully software-emulated virtual machine. In addition, the emulation can be extended to provide reliable counters for the landmark. The log growth rate is considerably higher than in ReVirt, reaching 10 GiB per day for a Linux web server. Since ExecRecorder does not compress logs, this suggests that replay logs are generally well compressible. Measurements in [50] and [286] confirm this observation. However, as with ReVirt, measurements with ExecRecorder also reveal that the log growth rate

[18] Allows to differentiate iterations for instructions with the repeat (`REP`) prefix.

greatly depends on the workload. Similarly, with less than 4%, the average run-time overhead during recording is almost negligible.

Other homogeneous replay projects using emulators are [59, 83, 85, 244], all employing binary translation with QEMU due to its comparably fast binary translator. Nevertheless, the slow overall execution speed of software-based virtual machines remains a major drawback of such systems, limiting their value in practice.

2.4.2 Heterogeneous Replay

Combining the fast virtualization speed of hardware-assisted virtual machines and the powerful analysis capabilities of emulators has been the goal for various research projects [63, 64, 286, 287]. The general idea is to record all nondeterministic events with hardware-assisted virtualization and replay the events in the emulator, thereby presenting exactly the same execution to analysis tools. Although this approach does not reduce the run time of the emulation, it recreates the realistic timing and potential user and network interaction of the original hardware-assisted run in the emulation. This makes the emulation representative of a productive run, irrespective of the slowdown induced by the emulation and analysis. Since the execution environment between recording and replay changes, we refer to this method as *heterogeneous replay*.

Compared to homogeneous replay, heterogeneous replay introduces a number of new challenges. Since the CPU implemented in the emulator does not match the host's CPU, instructions such as `CPUID` on x86 will usually return different values. The recording thus has to capture more information to allow a faithful replay. A larger obstacle is the plethora of inaccuracies hidden in most emulators [167] that can lead to divergence in the replay (see Chapter 8 for a thorough discussion). The emulation thus has to be meticulously adjusted to match the behavior of the physical CPU. While this makes heterogeneous replay less portable and requires much more development efforts, the result is a minimally invasive analysis tool that even allows the evaluation of interactive workloads for which traditional emulation is often too slow. However, only a few research projects have targeted full system heterogeneous replay and publications are from VMware mostly.

With ReTrace [286], VMware published a trace collection tool for x86 based on heterogeneous replay that even found its way as an experimental feature into VMware Workstation 6.0. ReTrace first records the workload of interest with hardware-assisted virtualization. In the second phase, ReTrace replays the execution with interpretation, allowing to transparently install tracing hooks. Just as reported for projects in the field of homogeneous replay, the run-time recording overhead is on average 5% for CPU intensive workloads but climbs to 2.6x for OS and I/O intensive workloads. A major drawback of ReTrace is that it does not cope with the immense emulation slowdown bound to the tracing phase. The

authors only state that the tracing for a 30s benchmark was stopped after 2 hours, without giving more detailed information on the exact full emulation run time[19].

VMware continued their efforts in the area of heterogeneous replay with Aftersight [63], a framework for integrating complex analyses into the replay. In contrast to ReTrace, Aftersight also supports replay in QEMU because its binary translator is better suited for instrumentation compared to the heavily optimized and specialized one in VMware Workstation. Due to the incompatible device models between VMware and QEMU, the replay cannot just feed non-deterministic events to the emulation, but must also fully replay the output of devices. To generate a device model agnostic log, Aftersight must first perform an additional replay in VMware Workstation and capture all device output. In consequence, the virtual devices in QEMU are not longer needed and the authors could strip out everything except the components that deal with instruction execution and memory access.

The second installment of ReTrace has been published by VMware with Crosscut [64]. In contrast to previous replay systems, Crosscut allows users to slice replay logs along time and abstraction boundaries so that a replay only includes the processes or components that are of interest to the research question at hand. Retargeting a log, for example, to focus on a certain process, requires a dedicated replay run that will deliver a new refined log. This concept is a direct evolution of the process used in Aftersight to generate a device model agnostic log.

Restricting the replay to a certain component is also a key feature of V2E [287], a heterogeneous replay engine for malware analysis. The user can define a recording realm (e.g., a certain process or kernel module), which is the only part that is replayed. V2E creates two mutually exclusive guest physical memory spaces, where each individual guest physical page can only be present in the EPT of the recording or the main realm. Whenever the CPU is running in the recording realm and accesses a page present in the main realm, its content is captured in the replay log and the page's ownership changes. This allows V2E to always present the right memory contents to the recording realm in the replay, despite the absence of the main realm. Just like Aftersight, V2E uses QEMU for analysis (but KVM for recording). Due to the malicious context in malware analysis, V2E strongly depends on a precise replay to prevent the malware from crashing the emulator, as this would torpedo the analysis. In consequence, V2E also records exceptions. Although deterministic by nature, they are difficult to emulate accurately. This extended recording, as well as the additional overhead of capturing realm boundary crossings, comes at the cost of a high recording overhead between 5x and 17x. This potentially perturbs and distorts the original workload execution, making the design less suitable than Crosscut for analyzing time-sensitive behavior.

[19] If the workload would have finished after 2 hours, the slowdown would have been around 240x. The authors report a general slowdown of 2 orders of magnitude *at minimum*.

2.4.3 Multiprocessor Replay

Albeit supporting multithreading, multiple processes, or even a whole virtual machine, the approaches presented so far only cope with uniprocessor deterministic replay. To faithfully replay a multithreaded application on a uniprocessor, it is, for example, sufficient to accurately replay the scheduling decisions [127, 227]. Compared to a multiprocessor replay, such a system does not have to handle the non-determinism emerging from the true parallelism of multiple concurrently executing CPUs. In addition to the techniques employed in the uniprocessor case, a multiprocessor replay therefore has to also include information to precisely reconstruct the timing between parallel cores whenever an inter-processor event takes place. These may be easily observable in the operating system or hypervisor like in the case of inter-processor interrupts (IPI), but may also happen without explicit intervention as in the case of shared memory accesses. The outcome of a race for a shared mutex, for instance, is not known in advance and depends on the many non-deterministic factors inherent to the hardware such as the memory access ordering, the relative timing between the involved CPUs, and their current cache states. The challenge in multiprocessor replay is thus to make originally transparent inter-processor interaction such as via shared memory visible to the recording component without compromising efficiency.

Most of the literature in the field of multiprocessor deterministic replay examines the replay of single applications comprising one or more processes [30, 152, 153, 191, 197, 200, 220, 245].

Instant Replay [152] implements a concurrent-read-exclusive-write (CREW) protocol with a set of specifically crafted read and write locks which must be used in the application to access shared objects. The CREW protocol assigns ownership exclusively to one processor but permits multiple concurrent readers. To replay the order of operations on a shared object, Instant Replay assigns each shared object a version number which is incremented on modifications. During replay, the locks delay access until the object's version number matches the one during recording, thereby enforcing the same access order[20].

A similar approach is used in RecPlay [220], which records the order in which threads acquire synchronization objects. While the concept is appealing from a performance perspective, it is not capable of faithfully replaying across race conditions – which do not employ correct synchronization by nature. RecPlay thus utilizes an on-the-fly data race detection during the replay phase which aborts replay on the first race condition.

Respec [153] has been designed with online replay (e.g., to parallelize security checks) in mind, that is, the recorded and replayed processes execute concurrently. Respec splits the execution of a program into epochs and relaxes the accuracy

[20] Instant Replay does not reproduce the order of reads of the same value (i.e., same version).

of the replay within the epoch. Instead of forcing exactly the same execution on an instruction-by-instruction basis, Respec only guarantees that the program generates the same output (i.e., invocation of system calls and final program state on epoch end). As Instant Replay and RecPlay, Respec only logs the order in which synchronization primitives are acquired. If, however, the replay fails to reproduce the same output (e.g., due to a race condition), Respec rolls back the original execution in the recording stage to the beginning of the current epoch. It then retries execution and replay of the current interval in the hope that the race condition will not be triggered again[21]. Respec thus tries to avoid race conditions in the recording.

Recap [197] and Flashback [245] record the values read when accessing shared memory and in consequence can faithfully replay race conditions – in contrast to the aforementioned methods. While Recap generates specific memory access code in the compiler and thus requires a recompilation for replay, Flashback uses page protections to trigger a page fault on every shared memory access. Although the authors of Flashback did not explicitly evaluate this solution, a high performance degradation for frequent shared memory accesses can be expected.

Multiprocessor replay for system virtual machines has been first demonstrated with Flight Data Recorder by Xu et al. [285]. FDR is designed as a hardware processor extension for a sequentially consistent memory system, which records the non-determinism in a sliding window of 1 billion CPU cycles before a trigger (e.g., a fatal crash). FDR resolves shared memory accesses by capturing cache coherence messages. An evaluation using a timing simulation revealed an estimated run-time overhead of less than 2%.

A software version of the cache coherency protocol recording in FDR has been adopted in PinPlay [200], a framework for deterministic replay of parallel programs. Since PinPlay uses dynamic binary translation to allow program instrumentation, the logger possesses a high slowdown of 80x (single-threaded program) to 146x (multithreaded program) compared to native execution.

With SMP-ReVirt [88], Dunlap et al. published a multiprocessor extension for ReVirt [87]. Similar to Instant Replay, SMP-ReVirt employs a CREW protocol, but it applies the protocol to the guest physical memory of a Xen virtual machine, thereby enabling full system multiprocessor replay. The ownership of guest physical pages is controlled by adapting page protections in per-processor EPTs and switching on page faults. While single-core recording run-time overhead is around 15% for a Linux kernel build, the overhead considerably increases with the number of CPUs, reaching 2x for two processors, and 9x for four processors.

Although this suggests that the CREW protocol is not scalable, ReEmu [59] implements an optimized version in a parallel variant of the QEMU full system emulator

[21] Respec manipulates the original thread scheduling if the same epochs must be rolled back twice.

with promising results. The authors report an average slowdown between 60% and 77% for PARSEC benchmarks with 1 to 16 cores. However, compared to native execution, the run-time overhead reaches up to 33x due to the overhead of DBT.

Besides the cache coherence protocol recording and the CREW protocol, a third approach is to split the instruction flow of each processor into a stream of atomic chunks. The processor either commits the chunk and makes its effects (e.g., changes in memory) publicly visible or, in case there is data dependence across two concurrently executing chunks, performs a rollback. The approach thus controls the original execution so that data races can be resolved. To replay a multiprocessor execution, the system only has to record the order of chunks, which incurs less overhead (run time and log size) than working at the granularity of individual memory accesses. The concept has been first presented with DeLorean [182] as a simulated hardware solution for full system multiprocessor replay. Samsara [215], in turn, implements the concept in software for hardware-assisted virtual machines using CoW and CPU register snapshots to allow rollback. Although generally performing better than SMP-ReVirt with a slowdown of 6x for a kernel build with four processors, the run-time overhead for recording remains high.

Intel therefore invested into a prototypical x86 architecture extension for multiprocessor replay, called QuickRec [205]. QuickRec also uses a chunk-based recording approach, but as a processor feature can rely on address snooping to detect conflicts at the moment they occur. This allows the CPU to immediately terminate the chunk, obviating rollbacks. The authors evaluated the performance overhead with an FPGA implementation and found it to be negligible. However, since QuickRec only captures shared memory interleaving, it requires software to capture all remaining non-determinism. This incurs a run-time overhead of 13%. Although the authors demonstrated a prototype for program replay, there is no design capable of full system replay.

2.4.4 Conclusion

By capturing non-deterministic events such as interrupts and the output of I/O devices, deterministic replay is able to precisely reproduce the execution of a program or whole system. It is thus a powerful technique that aids in tracking down bugs and analyze malware, and can even improve virtual machine live migration and fault tolerance systems. Whereas a homogeneous replay executes in the same environment (e.g., hardware-assisted VM) as the recording, heterogeneous replay switches environments. This makes heterogeneous replay particularly interesting for research and development as it allows recording with hardware-assisted virtualization, providing full interactivity, realistic timing, and transparency to malware, and replaying with emulation for pervasive instrumentation and anal-

ysis. However, heterogeneous replay does not reduce the immense slowdown incurred by the (instrumented) emulation and is thus limited in its applicability. While the size of the resulting recording logs, as well as the run-time overhead during recording, is not a problem for uniprocessor platforms, multiprocessor deterministic replay is generally considered inefficient in software. Unfortunately, complete hardware/software co-designs for commodity architectures such as x86 capable of full system multiprocessor replay have not been presented yet.

Chapter 3

Functional Full System Simulation

With *full system simulation*, we describe the process of running a system virtual machine for the purpose of development and research, for example, to collect detailed execution traces. In contrast to a generic binary translator, a full system simulator typically implements additional instrumentation features, performs instruction or cycle counting, and possesses a sophisticated tracing facility. To this end, the full system simulator has to create a software model of the targeted architecture. As with every model, we are – to some degree – free to choose the level of detail we want to integrate into the model. For a virtual processor, it generally will not be reasonable to approximate the result of instructions or reduce the precision of registers, as this will most probably break compatibility with the targeted ISA and prevent software in the guest to run properly. Since the ISA is the developer-visible interface to the hardware, conforming to the ISA is a line which a model should not fall below. A simulation at this level is called a *functional full system simulation (FFSS)* because it is only concerned about the correctness of the output of instructions rather than the emulation of the inner-workings of system components.

Depending on the use case of the virtualization, it might, however, be desirable to also model microarchitectural details such as the design of the instruction processing pipeline and precise timing. This is especially the case if effects of changes in the microarchitectural implementation of a processor should be evaluated – traditionally the field of computer hardware architects. However, the publication of the Meltdown [156] and Spectre [141] attacks unveiled a whole class of security vulnerabilities in modern processors that also forced extensive changes in operating systems. Since these attacks are based on the out-of-order and speculative execution within the processor, a model restraining to the ISA level is not sufficient to discover such problems. Microarchitectural models therefore can conceptually be useful not only in hardware design but also in OS and security research. However, detailed information on the inner-workings of modern general-purpose CPUs is proprietary knowledge and as such not available to the

research community. Researchers are therefore forced to reverse engineer such CPUs and carefully tune microarchitectural models experimentally [80, 292], or simply make educated guesses [91]. Still, this does not catch all subtleties of the actual hardware and models of existing commercial CPUs remain an approximation[1]. The teams behind Meltdown and Spectre consequently had to resort to trial and error experiments with real hardware instead of being able to deduce the attacks straight from the microarchitectural design. A primary focus for operating and security research therefore remains the virtualization at the functional level, which has been demonstrated to be very effective for debugging [137], security analyzes [133, 287], collecting detailed execution traces [286], and many more.

In contrast to productive use of virtual machines, where virtualization speed and management features such as live migration and fault tolerance are paramount, using VMs for research shifts the focus to sophisticated instrumentation and tracing capabilities. This way, researchers are able to gather valuable information on the run-time behavior of applications and the operating system. While production VMs thus bet on the performance of hardware-assisted virtualization, full system simulations require the flexibility of emulation techniques, that is, interpretation or dynamic binary translation.

For instance, SimOS [223] implements three execution modes. Due to the limited availability of hardware-assisted virtualization back in the 90s, the fastest mode uses direct execution and can be utilized to set up the virtual machine and fast-forward over the OS boot phase during benchmarks. However, the authors highlight that direct execution inherently neither supports instrumentation nor does it allow to model timing aspects. It further requires compatibility between the guest and host architectures. The authors therefore state that direct execution is generally inappropriate for research studies. As hardware-assisted virtualization gives away even more software control, the same applies to modern virtualization. The second operation mode in SimOS uses dynamic binary translation. It can be used to perform arbitrary instrumentation, collect execution traces, and gather run-time statistics such as (simulated) cache hits and misses. In the third mode, SimOS falls back to interpretation, which allows it to apply comprehensive processor and memory timing models[2].

We can conclude that functional simulation is the predominant technique for simulation-based OS and security research that requires detailed run-time information. Variants of dynamic binary translation or interpretation, in turn, are

[1] Average error rates in metrics such as execution time or instructions per cycle (IPC) for microarchitectural models compared to real hardware are around 5% to 15% [25, 91, 159, 292], going up to 40% [25] for some benchmarks. Simple microarchitectural models, albeit used by hundreds of publications, show errors up to 77% [80].

[2] Although technically possible, using binary translation in this mode provides little benefit due to the many model-related helper calls. At the same time, it adds significant complexity because the advanced simulation features must be reflected in dynamically generated code.

the virtualization technique of choice for all major full system simulators such as Simics [163], QEMU [38], Bochs [2], and gem5 [41], and they are the driving power in binary instrumentation tools such as DynInst [47], Pin [162], and Valgrind [190].

Since the main contribution of this thesis is a novel approach to the acceleration of functional full system simulation, the next section provides an assessment of the execution speed of current full system simulators. Building on these results, Section 3.2 takes a look at existing acceleration methods for full system simulation and discusses their applicability in operating system and security research. The chapter concludes by summarizing the limitations of the state-of-the-art techniques in Section 3.3.

3.1 Assessment of Simulation Speed

Despite being a versatile technique for detailed system inspection, a major drawback of functional full system simulation is the immense slowdown incurred by the emulation. Figure 3.1 depicts the slowdown for various workloads running with QEMU's binary translator compared to the execution in a hardware-assisted virtual machine[3]. Whereas a kernel build completes in less than 13 mins with HAV, a functional simulation takes about 5 hours. The figure also illustrates that the slowdown strongly depends on the instruction mix of the workload. The

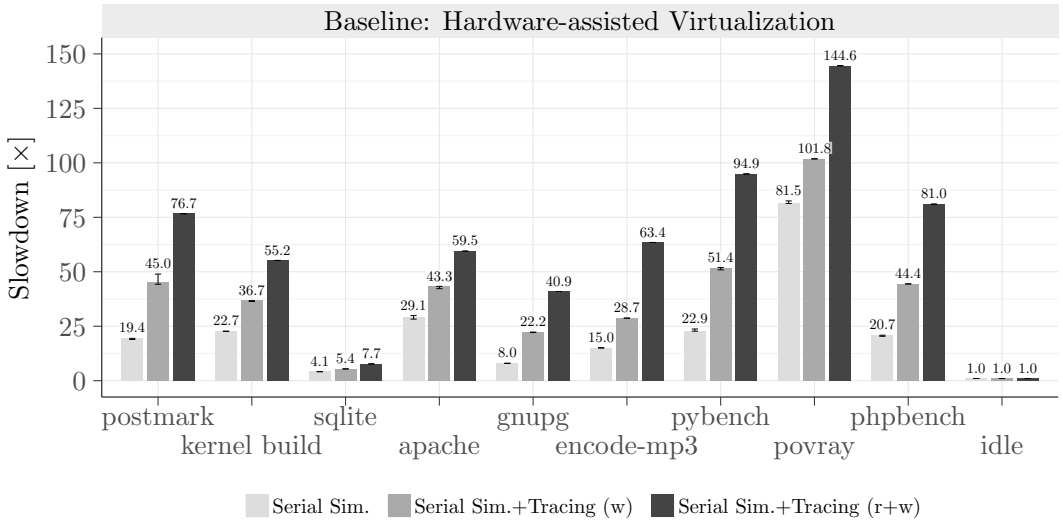


Figure 3.1: Running a serial simulation with QEMU is significantly slower than running the same workload in a fast hardware-assisted virtual machine. Installing hooks for tracing memory accesses slows down the simulation even more.

[3] See § 9.1 on page 185 for information on the benchmark environment and the test scenarios.

FPU-heavy povray benchmark runs for over one day instead of 19 mins. Installing hooks for tracing memory accesses prolongs a single execution to almost two days, making measurements exceedingly time-consuming. With Simics [163], we measured a slowdown of up to 1000x for SPECint2006 when hooks for tracing (reads and writes) are installed [219]. Similar slowdowns for functional simulation have also been reported by other researchers [58, 135, 164, 200, 286].

In practice, this slowdown creates severe obstacles for comprehensive use of functional full system simulation:

Interactivity Scenarios that should capture interactivity with a human user or an external network device are impractical. Prominent examples are evaluations that include desktop usage or benchmarks such as SPECweb [15] which require a second virtual system to generate requests. With a slowdown of one to two orders of magnitude, applications are barely usable. A single keystroke can quickly take from multiple seconds up to minutes until being fully processed. Network protocols such as TCP, in turn, react with throttling and timeouts[4].

Accuracy of Results Since the simulation considerably slows down the guest, activities dependent on external events appear to complete faster. For instance, in the same amount of wall-clock time for an I/O operation, the simulated CPU retires drastically fewer instructions than a real CPU, which will in relation appear as if the I/O device operates much faster – a phenomenon called *time dilation* by Chen and Bershad [56]. Similarly, timer interrupts occur more frequently, which effectively shortens time slices and increases the number of context switches[5].

Coverage With contemporary functional full system simulation, evaluating modern benchmarks in their full length takes considerable time. As apparent from Figure 3.1, the run time is in addition very sensitive to further overhead incurred by instrumentation, quickly rendering simulation impractical for long-running workloads or heavyweight analyses.

The key to productive and comprehensive use of functional full system simulation in operating systems and security research is therefore the reduction of the slowdown and the ability to run full-length, interactive workloads with realistic timing behavior.

[4] Simics can also simulate network peers in order to produce a realistic communication pattern. This, however, creates more complex simulations with even higher slowdowns and the concept is inherently not applicable to human interaction.

[5] This can be to some degree alleviated with a virtual clock based on retired instructions. However, without a sophisticated timing model, this still misses the variations in processing speed for different instruction mixes.

3.2 Acceleration Techniques

A common method to cope with the limited computational resources in full system simulations is to adapt or shorten the workload so that the overall simulation time remains in reasonable bounds. However, the reduced workload may not retain the original characteristics and thus produces results of questionable value. Developing reduced but representative workloads is thus a complex and time-consuming task. With MinneSPEC [140], efforts have been made to generate reduced input sets for the popular SPECCPU2000 benchmark suite. A scaled-down commercial workload suite has also been published [28]. Nonetheless, the principle is not generally applicable and may inherently cut the value of a study by reducing coverage (e.g., evaluating only a small set of test cases [133]).

Instead of adapting the workload it is conceptually also plausible to run the workload in a process virtual machine such as Pin [162], thereby avoiding the overhead of simulating the guest operating system. However, process virtual machines are unable to track dynamic behavior across a full system including privileged system components and multiprogramming activity. They are thus not well-suited for operating system research and are of no use if kernel internals should be analyzed (e.g., as in [133]). The restriction to the scope of a single application also raises questions concerning the accuracy of results obtained through such narrow inspection. While it comes naturally that I/O-heavy workloads such as databases and web servers are clearly influenced by OS activity [135, 169], Cain et al. found that ignoring the effects of the operating system can lead to severe measuring errors even for CPU-bound benchmarks such as SPECINT2000 [53].

It is thus desirable to keep the original workload intact – including the guest operating system – and to accelerate functional full system simulation instead.

3.2.1 Optimizing the Execution Engine

The speed of a functional simulator is mostly determined by the ratio of executed host instructions per simulated guest instruction. Improving the simulation speed thus equates to improving this very ratio. This can be achieved by reducing the time that the CPU spends on executing simulator code (e.g., to control the execution or jump between dispatch routines), and by generating more efficient code in the case of a DBT engine.

Shade [67] and Embra [281], for example, avoid loads and stores to vCPU registers when successive instructions use the same virtual registers. QEMU updates various vCPU state such as CPU flags and the instruction pointer only when it is actually needed [38]. Fu et al. [95] extend the intermediate language in QEMU to support vector instructions, thereby improving the speed of selected SIMD benchmarks. Other research [115, 119, 210] uses compiler back ends such as LLVM [149] to

leverage sophisticated code optimization features when generating TBs from the intermediate language. In contrast, Zhang et al. [295] apply optimizations on the final compilation result.

Reducing the time spent in the simulator has been done, for instance, in SPIRE [129]. The project resolves guest instruction pointers to host instruction pointers at indirect branches in the guest code by installing a lightweight trampoline at the target guest IP, instead of using address hashing and a lookup table. Generating larger translation blocks with more efficient traces (i.e., sequences of hot basic blocks) is another approach [120].

A common denominator of all these techniques is that improvements at the level of the generated code and execution flow do not provide the required leap and generally stay below the 5x speedup mark (in the face of slowdowns of one to two orders of magnitude). In fact, the authors of SoftSDV [260] considered to move various complex operations from helper calls directly into the generated translation blocks but found the resultant loss in flexibility and maintainability to not justify the optimization.

3.2.2 Reducing the Observation Space

A prevailing practice to make analysis via simulation applicable to long-running workloads is to trade accuracy for speed by limiting the simulation to short time frames – so-called samples. To this end, many simulators support dynamically switching between different detailed simulation modes at run time, enabling the user to fast-forward between samples with some kind of accelerated execution. In Simics [163], the user is able to choose whether timing models are applied or not. SimOS [223] and MARSSx86 [199] allow switching between functional and microarchitectural simulation. PTLSim/X [292] and FSA [229], in turn, provide hardware-assisted virtualization as an alternative mode of execution.

A fundamental challenge with sampling is to find the right samples so the selected subset reflects the overall workload characteristics. The gathered information can then be extrapolated to draw conclusions for the whole workload. In research, three primary variants emerged [289]:

In *truncated execution*, the workload is simulated for only a short duration with the presumption that the abbreviated execution phase is representative for the whole program. Most applications exhibit an initialization phase, where internal data structures are set up and input data is loaded into memory before the application actually starts performing its task. The latter phase then usually dominates the program behavior. A common variant in truncated execution is thus to fast-forward over the initialization phase and start detailed simulation or analysis for a limited duration afterward. Depending on the level of simulation and type of analysis, a warm-up phase is prepended before taking measurements

to line-up any additional state (e.g., a cache model). Because the policy is easy to implement, it has been widely adopted in the literature. According to a study by Yi et al., over 50% of publications[6] on HPCA, ISCA, and MICRO base their results on this technique [289]. However, the same study also revealed that truncated execution is highly inaccurate. This finding has also been confirmed by other groups [101, 273]. The inaccuracy is caused by the fact that the approach does not account for changing program behavior and at the same time depends on manually and arbitrarily chosen parameters such as the time span to simulate.

SimPoint [109, 237] leverages sampling to reduce simulation time and increases accuracy by selecting multiple time frames to sample from. It is thus able to incorporate changing program behavior. Furthermore, the time frames are selected algorithmically through detecting phase behavior in the simulated workload. SimPoint thereby focuses the simulation and analysis on windows with representative characteristics. To gather initial information on the program behavior and to fast-forward between samples, SimPoint requires functional simulation. It is thus only suitable to accelerate more detailed execution modes such as microarchitectural simulation, which face even higher slowdowns than emulation. However, it does not present a solution to accelerate functional simulation, which is the goal of our work[7]. At the same time, SimPoint's strength to build on phase behavior becomes its weakness when no sufficient phase behavior exists. SimPoint has been developed with a single process in mind. However, even for such a scenario, it has been shown that representative intervals may not be clearly identifiable due to too complex program behavior (e.g., gcc [273]). For operating system research, where a mix of processes run alongside OS kernel and driver threads, observing phase behavior becomes even more difficult.

SMARTS [284] evades this problem by collecting samples periodically with high frequency, ignoring program behavior. The number of samples taken in SMARTS is thus higher, while each sample is considerably smaller (1000 instructions versus 100 M in SimPoint). SMARTS employs sampling theory to choose a minimal sampling frequency and to achieve a quantifiable accuracy and precision in its measurements. As with SimPoint, SMARTS targets the acceleration of microarchitectural simulations and requires functional simulation to fast-forward between sampling points and to warm up microarchitectural state. The functional simulation consequently occupies more than 99% of simulation run time. DirectSMARTS [58] demonstrates how simulations using SMARTS can benefit from acceleration methods for functional simulation, in this case, using emulation with dynamic binary translation instead of interpretation in the process-level RSIM [196] simulator. To improve the run time for subsequent experiments with the same benchmark, Wenisch et al. checkpoint the warmed-up state in so-called LivePoints [276]. While subsequent runs can then start off the checkpoints, the

[6] Over a ten years period, ending in 2005.

[7] In fact, an accelerated functional simulation could be used to generate SimPoints much faster.

concept still requires a complete functional simulation beforehand. FSA [229], a recent publication targeting SMARTS, skips most of the functional simulation by using hardware-assisted virtualization instead. FSA then dynamically switches to functional simulation before each sample to warm up microarchitectural state and finally switches to detailed simulation for the sample itself.

A major drawback of all sampling-based methods is that they are directed toward the estimation of metrics that can be extrapolated from samples (e.g., instructions per cycle, etc.). Sampling is less suited to observe the actual system execution as required in security research, malware analysis, or debugging. Moreover, limiting the observation window to discrete samples may not be an option because it does not permit the tracking of individual events. For example, identification of memory <allocation, deallocation>-pairs cannot be done this way, as used in Undangle [52] to detect invalid pointers in use-after-free and double-free vulnerabilities. The same applies to the memory access pattern analysis in Bochspwn [133] and the evaluation of sharing opportunities for memory deduplication [105, 176, 217]. Instead, such applications demand an acceleration method that offers continuous simulation.

3.2.3 Parallelizing Multicore Simulations

An alternative to code optimization and sampling is to parallelize the simulation of cores in multicore simulations. In a simple design, emulating multicore systems can be done by switching between the vCPUs in a round-robin fashion. This reduces the complexity of the simulator because it serializes the simulation and thus requires less synchronization and does not introduce non-determinism from inter-processor interference. The time-sharing, however, further reduces the overall execution speed proportionally to the number of vCPUs. Parallel multicore simulation mitigates this additional slowdown by emulating each vCPU in parallel on a dedicated hardware thread. Many existing simulators support this mode of operation today officially or through unofficial patches [81, 148, 268, 271, 294]. Graphite [175] is even capable of distributing the simulation of vCPUs across multiple machines to enable many-core (i.e., thousands of cores) simulations. Portero et al. [207] expanded on these capabilities with a simulator that also delivers timing and functional models for on-chip inter-connection systems.

While achieving good speedups compared to serial multicore simulations (e.g., 3.8x for a quad-core ARM simulation [81]), the approach of parallel multicore simulation does not accelerate the execution of the vCPUs themselves. Its scalability is therefore inherently limited to the degree of simulated parallelism. Single-core simulations do not benefit. However, to make FFSS a prevalent solution for system inspection, we need to drastically reduce the slowdown for the emulation of a *single core* because this slowdown presents the entry obstacle for functional full system simulation.

3.2.4 Parallelizing the Simulation Time

The parallelization of simulation time is another method based on parallelization, which has been first proposed for microarchitectural simulations. Lauterbach [151] suggested to periodically create instruction traces of short samples and to simulate the samples in parallel on multiple processors. A benefit of this method compared to parallel multicore simulation is that it almost linearly scales with the number of processors available for simulation instead of the simulated degree of parallelism. However, the solution by Lauterbach requires a time-consuming (days to weeks [151]) setup phase in which representative samples are detected and instruction traces are generated.

A similar, but much faster variant of sampling with SMARTS has been presented under the name pFSA [229]. In contrast to FSA, which serially switches back and forth between hardware-assisted virtualization and simulation, pFSA forks the VM whenever it reaches a sampling point. The forked VM then performs the functional warming and detailed simulation in parallel to the execution of the hardware-assisted parent and other concurrently running simulations. As with the trace-based approach by Lauterbach, the speedup is almost linear to the number of cores used for simulation. However, since pFSA uses forking instead of checkpoints, scaling the parallelization over multiple hosts is not easily possible. Furthermore, every iteration in the analysis requires a full re-run of the workload. Since pFSA uses hardware-assisted virtualization, the execution is non-deterministic. Scarce events such as race conditions cannot be faithfully reproduced this way. In addition, pFSA is not able to simulate interactive workloads because external input is not fed into the simulations.

LiveSim [110] also periodically forks the simulator process during a setup phase in a functional simulation so as to preserve in-memory snapshots of the VM. This is comparable to the approach in pFSA, however, LiveSim forks the children again before actually starting simulations in order to keep the original saved state for repeated experiments and to allow reproducing identical simulations – given that the simulations are fully deterministic.

Forking has also been utilized in Shadow Profiling [184] to perform parallel profiling of a user-mode process with Pin [162]. In contrast to pFSA and LiveSim, which use periodic sampling, the degree of parallelism in Shadow Profiling is determined by a user-defined load factor that limits the number of concurrent simulations. Whenever the number of simulations falls below the load factor, Shadow Profiling forks off a new simulation at the current instruction pointer. Each simulation then runs for a configured number of instructions.

Despite that the authors of LivePoints [276] do not present any implementation, they mention the idea of starting parallel simulations of samples based on checkpoints. These can be more easily transferred over a network than live processes.

While this would help with the limited scalability of pFSA and LiveSim, LivePoints still depend on functional simulation and share the other shortcomings with pFSA and LiveSim such as the inability to faithfully simulate interactive workloads. Considering that these projects are designed for simulating very short samples (1000 instructions) in microarchitectural simulations, this is not surprising. Instead, it underlines that, while these approaches present innovative concepts, they are not suited for general OS and security research, which requires repeatable continuous simulation of potentially interactive workloads.

Nevertheless, the partitioning and parallelization of simulation time is a promising approach and conceptually not limited to samples, but can also be applied to whole time intervals. This has been successfully shown by Heidelberger and Stone [111], as well as Nguyen et al. [192], who, similarly to Lauterbach, suggest to use trace-driven simulation. However, they do not use samples but split the trace into disjoint time intervals that – in a second stage – can be processed in more detail simultaneously. Nguyen et al. measured a speedup of 14x when using sixteen processors. Although this allows continuous full-length simulation of the original workload, it still requires a time-consuming setup with functional simulation to collect the (potentially very large) trace beforehand. In addition, any changes to the information contained in the trace necessitate a full re-run.

To warm up architectural state (e.g., caches) at the interval beginnings, Nguyen et al. propose to overlap intervals. DiST [101] further enhances this idea by introducing automatic warm-up phase scaling. The authors report speedups between 20x and 39x for forty processors. DiST directly parallelizes microarchitectural simulations by spawning parallel processes that use functional simulation to fast-forward to the respective interval. Since both the functional simulations and the microarchitectural simulations are deterministic in nature, it is guaranteed that the executions do not diverge. Nevertheless, DiST is not capable of handling interactive workloads and cannot be applied to accelerate functional full system simulation.

A solution specifically for single-threaded user-mode processes has been demonstrated with SuperPin [267] – a parallelized version of Pin. In contrast to Shadow Profiling, SuperPin splits the workload execution into slices, either on a system call or after a fixed timeout. At interval boundaries, SuperPin forks the process and the child switches to simulation. The child stops when it detects a CPU state signature that SuperPin has recorded at the end of the respective interval. Simulations are therefore delayed until the next interval. Whereas SuperPin reproduces the return values of system calls, the authors did not provide any information about whether SuperPin is able to faithfully replay asynchronous signals. The speedup over serial simulation ranges from 3x to 7x for eight processors.

Parallelization through checkpointing and deterministic replay for multi-threaded processes, including asynchronous signals, has also been found to be very effective

in JetStream [211] – an approach to parallelize dynamic information flow tracking (DIFT) in Pin. The speedup for DIFT queries ranges from 12x to 48x for 128 cores. The authors stress the performance benefits that checkpointing brings compared to fast-forwarding by plain replay as, for instance, used in DiST.

3.3 Conclusion: Limitations of the State of the Art

Virtual machines have proven as a powerful environment to conduct operating systems and security research. However, the range of tools applicable heavily depends on the technology utilized for virtualization. While interpretation and dynamic binary translation provide researchers with a maximum of flexibility and allow for detailed instrumentation and tracing, in practice, the slowdown for functional full system simulation is a major obstacle. Our measurements confirm the observations published in the literature that the slowdown for functional full system simulation typically ranges from one to two orders of magnitude, with a high sensitivity to additional overhead from analysis code. This makes analyzing long-running workloads infeasible, prohibits interactivity, and at the same time perturbs results by distorting the timing within the simulation. Although incorporating detailed architecture models can produce a more faithful timing, this does not cope with the lack of interactivity and introduces considerable further slowdown. In addition, such models require extensive development efforts to exhibit realistic timing. It has been found that simple timing models, albeit used by hundreds of publications, possess considerable errors.

On the other side of the spectrum, hardware-assisted virtualization brings near-native execution speed, including full interactivity. However, the removal of software interposition deprives researchers of the possibility to leverage instrumentation and tracing tools. This pushes hardware-assisted virtualization toward productive use and makes it less attractive in research and development.

While heterogeneous deterministic replay is able to inject recorded interactions into a functional simulation and reproduce the realistic timing of interrupts from the original hardware-assisted run, it does not accelerate the simulation itself. Current solutions to speed up simulations, in turn, are inadequate for operating systems and security research as they are either not applicable to continuous functional simulation, restricted to single user-mode processes, limited in scalability, or prohibit interactivity.

Partitioning and parallelizing the simulation has been shown to be a powerful concept with promising speedup characteristics. However, none of the approaches present today is able to apply the technique to continuous functional full system simulation to allow a comprehensive analysis of long-running and interactive workloads.

Chapter 4

SimuBoost

While functional full system simulation (FFSS) is a powerful technique, its immense slowdown in the range of one to two orders of magnitude (1) deprives simulations of interactivity with human users and non-simulated network peers, (2) limits the coverage of experiments that can be achieved in reasonable time, and (3) cuts the accuracy of measurements through time dilation. As we concluded in Chapter 3, existing methods of acceleration fall short of providing a scalable solution for realistic and repeatable continuous functional simulation of interactive virtual machines. This presents a major obstacle for comprehensive use of FFSS in operating system and security research. It is therefore desirable to develop an acceleration method that provides a drastic speedup over conventional FFSS, thereby mitigating the limitations that developers and researchers are facing today when utilizing such simulation techniques.

In this chapter, we introduce **SimuBoost**, our acceleration method for functional full system simulation. Based on the shortcomings of existing techniques, we begin in Section 4.1 by inferring a set of goals to which a solution should adhere. In Section 4.2, we elaborate on the core idea behind SimuBoost and explain how SimuBoost is able to satisfy these goals. To highlight in which points our work differs from previous publications in this area, we briefly recapitulate the relevant related work and perform a direct comparison with SimuBoost in Section 4.3. A summary in Section 4.4 concludes the chapter. The following chapters then describe and evaluate the key building blocks of our prototypical SimuBoost implementation.

> As outlined in § 1.2, we focus on single-core simulations only to demonstrate the general feasibility of our approach. We discuss to what extent SimuBoost is applicable to multicore simulations and what challenges remain to be solved on this route in the course of Chapter 10.

4.1 Goals

An acceleration method for functional simulation and tailored to operating system research should fulfill the following goals:

Temporal Completeness Sampling [229,237,284] generates speedup from simulating short windows only. This method does not permit the tracking of individual events (e.g., memory (de-)allocations [52]) and is not well suited to discover operational sequences (e.g., memory access patterns [133]). Truncation, on the other side, has been shown to suffer from high inaccuracy by ignoring program phase behavior [289]. An acceleration method should therefore provide full-length continuous simulation.

Spatial Completeness Completeness does not only relate to the temporal dimension, but also includes the spatial domain – i.e., the code that can be observed. A functional simulation for operating system research must include privileged system components and drivers. Accordingly, a suitable acceleration method must be applicable to *full system* simulation.

Scalability Multicore and multiprocessor systems are prevalent today and even clusters are readily available as cloud service. A technique to speed up FFSS should make use of this computational capacity and scale with the available physical parallelism. This should be independent of the simulated degree of parallelism so that also single-core simulations benefit.

Interactivity The inability to cover workloads that interact with a human user or a non-simulated network peer is a major downside of conventional functional simulation. The acceleration method should increase the simulation speed to such an extent that full interactivity is possible and is maintained even under the stress of added instrumentation.

Accuracy Inaccurate simulations inevitably lead to inaccurate measurements. Depending on the information to be obtained from the experiments, a lack of accuracy can render simulation as a methodology useless. This is especially a concern if the additional cost (i.e., development time and run time) for a realistic hardware model are not affordable or the know-how is proprietary. In the absence of such a model, the acceleration method should at least mitigate inaccuracies due to time dilation [56] while minimizing the probe effect [60] – i.e., the divergence from a non-simulated execution with the same speed.

Portability The field of operating system research naturally deals with a diverse set of hardware architectures. For this reason, it is desirable for the acceleration method to be universally applicable and not depend on implementation details of a particular hardware or simulation platform. Instead, it should be usable on widely available commodity hardware.

4.2 Approach

To achieve a maximum speedup for a given hardware setup, the scalability of an acceleration method is of utmost importance. In § 3.2, we highlighted that the *parallelization of simulation time* has proven to provide very good scalability with almost linear speedups. In contrast to methods that parallelize the execution of simulated CPUs, the parallelization of simulation time is also applicable to single-core simulations. At the same time, it is not limited to sampling, but equally allows continuous full-length observation (i.e., temporal completeness). Another advantage is that the concept is very portable because it does not merely gain speedup from optimizing implementation details – which exhibits only limited scalability anyway.

Due to its effectiveness, we have chosen to employ the parallelization of simulation time as the core idea in *SimuBoost* [219]. The run time of the serial simulation is split into disjoint time intervals (see Figure 4.1). The intervals are then distributed over a set of nodes for parallel simulation, where the nodes are dedicated physical CPU cores on one or more hosts. In this model, the overall run time equals the simulation time of the longest interval.

The simulation for interval $i[k]$ with $k \in [2, n]$ and $n \in \mathbb{N}$ requires the simulated machine's state at the beginning of this very interval. The machine state, in turn, results from the execution of the intervals $i[1]$ to $i[k-1]$. This creates a dependency chain which forbids the early start of parallel simulations. Another drawback of this method is that it neither improves accuracy nor allows interactivity because each interval is still executed with conventional slow functional simulation.

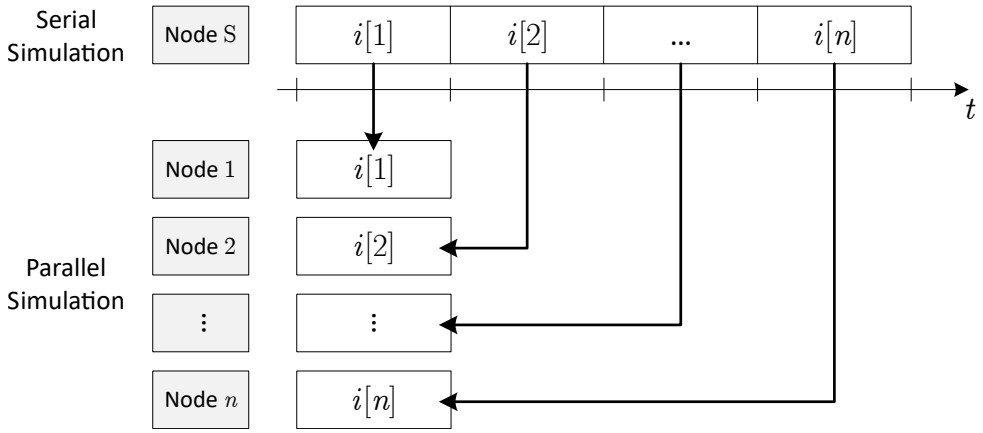


Figure 4.1: The serial simulation is split into disjoint time intervals and distributed over simulation nodes (e.g., CPU cores or hosts) for parallel simulation.

A setup phase with an initial full-length serial simulation as in trace-driven approaches would allow gathering the machine state at the interval boundaries. Subsequent runs could then be executed with parallelization. However, the lack of accuracy and interactivity remains. Also, requiring a time-consuming setup phase is unfavorable and considerably increases turnaround time.

To quickly retrieve the machine state at sampling points, pFSA [229] leverages hardware-assisted virtualization (HAV). Instead of running the workload in a serial functional simulation to fast-forward between samples, pFSA uses a regular fast virtual machine and forks off simulations as appropriate. SimuBoost adopts this general principle (see Figure 4.2) and applies it to continuous simulation: The workload executes in a hardware-assisted virtual machine (HVM) with near-native speed. Periodically, SimuBoost takes a checkpoint to capture the state of the VM, thereby marking interval boundaries. The checkpoint contains a consistent image of the virtual machine's memory, device states, and persistent storage at the time of taking. SimuBoost uses the checkpoint to bootstrap the simulation of the respective interval on a different node. In contrast to forking, checkpoints can more easily be transferred over a network, improving scalability. They can also be saved to disk for repeated runs. Although the simulations do not collectively start right at the beginning, the slowdown between hardware-assisted virtualization and functional simulation (exemplarily depicted as 4x) drives the parallelization.

Using a hardware-assisted virtual machine as input to the simulation has the additional benefit of recovering interactivity. Assuming checkpoints can be taken with low overhead (see Chapter 6), the HVM is fast enough to be actively controlled by a user just like a regular virtual machine in productive use. Similarly, the virtual machine is capable of communicating with non-simulated remote peers. Since the simulations execute in parallel to the HVM, the performance of the virtual

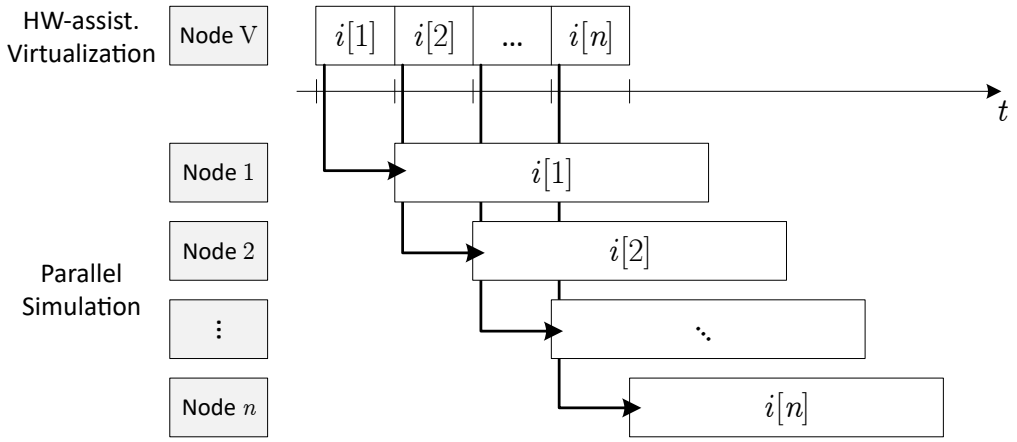


Figure 4.2: The workload runs in a hardware-assisted virtual machine with near native speed. At interval boundaries, SimuBoost takes checkpoints, which it uses to bootstrap parallel simulations. The slowdown between HAV and functional simulation (here 4x) drives the parallelization.

machine is decoupled from the execution speed of the simulations. This effectively maintains interactivity even in the face of additional instrumentation overhead.

The portability of the approach is only limited in that the target architecture must support some form of fast hardware-assisted virtualization. However, this feature is generally present today (e.g., x86, ARM, MIPS, and PowerPC) or planned for future releases (e.g., RISC-V).

4.2.1 State Deviation

Whereas simulations can be built to always emit identical deterministic runs, for example, according to a specified timing model, hardware-assisted virtualization is subject to non-deterministic input such as erratic I/O completion timing. In consequence, the parallel simulations in SimuBoost experience different timing behavior than the hardware-assisted execution of the same interval. Furthermore, the checkpoints only capture the result of past interaction, but they do not contain information on interactions in the current interval. Hence, any external input such as user commands or network packets received by the HVM is not reproduced in the simulations. The same applies to non-deterministic instruction results (e.g., readings of a timestamp counter). Consequently, the executions in the HVM and the FFSS of each interval diverge. Besides effectively losing interactivity, this is problematic in two ways:

1. Since simulations start off checkpoints that originate from the hardware-assisted virtual machine, the divergence breaks the *functional continuity* between interval boundaries in the simulation stage. That is, the machine state at the end of the simulation of interval $i[k]$ does not match the state at the beginning (i.e., the checkpoint) of interval $i[k+1]$. For example, recording a coherent instruction trace under such circumstances is infeasible.

2. For researchers to take measurements and retrieve data from a simulation which behaves differently from what they can see (i.e., the HVM) is at least confusing. In the worst case, the simulation is of little value if it does not reproduce the desired behavior as triggered in the HVM.

A possible solution is to take checkpoints not periodically but at each non-deterministic event. This immanently captures the point in time as well as the state modification caused by the event. This is similar to what has been done in Kemari [253], where a backup VM is synchronized via a checkpoint whenever the master VM sends network packets or writes to persistent storage. However, Kemari possesses a 2x performance overhead. Furthermore, our experiments show that on average 7400 (max: 230K) non-deterministic events per second occur during a Linux kernel build. This suggests that we would have to take a checkpoint every 135 µs on average and every 4 µs at peak times, which is clearly not feasible without severe performance degradation.

To counter state deviation, SimuBoost therefore leverages *heterogeneous deterministic replay*. Non-deterministic input such as the timing of interrupts, the payload of I/O operations, and the results of non-deterministic instructions are recorded during the hardware-assisted virtualization as discrete events and are precisely replayed in the simulations. This restores functional continuity and reproduces user as well as network input, which is indispensable to simulate interactive workloads. Since non-deterministic input needs to be fully captured prior simulation, SimuBoost delays the parallelization by one interval (compare Figure 4.2).

An advantage of deterministic replay is that it injects realistic timing into the simulation. This frees researchers from having to install a sophisticated timing model. In fact, the authors of PTLsim/X [292] advocate the use of deterministic replay even for microarchitectural simulations in order to create realistic timing. Furthermore, it has been shown that small changes in timing can have a great effect on simulation results [29]. By replaying different runs of the same scenario this can easily be taken into account with SimuBoost. This stands in stark contrast to deterministic simulations like, for instance, in Simics [163], which always produce exactly the same execution and thus miss to capture variations. On the other side, if a repeated, exactly identical execution is needed with SimuBoost, the same recording can be replayed any number of times.

On the flip side, the combination of checkpointing and deterministic replay closely ties the simulation to the hardware-assisted execution. Although Viennot et al. demonstrated that a replay can tolerate modified executable code to some extent [263], experiments generally have to remain passive observations. SimuBoost is thus not suited to, for instance, evaluate the effects of novel memory architectures because this would require a feedback loop into the execution to apply new timing information. However, forcing the simulation to deviate from the recorded execution path breaks functional continuity and prevents a correct replay of interactions and other non-deterministic events. Nevertheless, SimuBoost is perfectly suited if a detailed insight into an existing realistic execution is desired – for example when debugging or collecting traces. These traces, in turn, can be fed into new architectural models.

4.3 Comparison with Related Work

In 2002, the authors of ReVirt [87], one of the early works on homogeneous full system deterministic replay, already mention the idea to create intermediate checkpoints during native execution and then start replay off checkpoints, in this case to fast-forward to points of interest. In contrast to SimuBoost, the authors did not discuss parallelization. In fact, being homogeneous in nature, the replay in ReVirt remains to be a native execution and thus does not involve

functional simulation. Moreover, the authors conceived checkpointing to be a rare event (once every few days), which, according to the authors, does not justify special optimization. The actual ReVirt prototype misses support for any form of checkpointing and is only able to start replay from a powered-off virtual machine.

Three years later, King et al. developed a debugging tool for operating systems based on ReVirt, called TTVM [137]. A distinctive feature is its ability to perform reverse debugging for which TTVM has to periodically create checkpoints. Hence, King et al. eventually extended ReVirt with support for checkpointing, although not to realize parallelization. Nevertheless, with intervals of 10 s to 25 s the checkpointing frequency is rather low (too low for online parallelization as in SimuBoost). Since ReVirt supplies the replay functionality, TTVM also inherits its restriction to native execution.

In 2007, Xu et al. [286] proposed an acceleration method close to SimuBoost in a side note of their work on ReTrace. They suggested parallelizing extensive full system execution tracing by combining hardware-assisted virtualization with checkpointing, heterogeneous deterministic replay, and functional full system simulation. However, their implementation in ReTrace does not include any parallelization (or checkpointing) but solely demonstrates tracing in a serial heterogeneous replay. The authors also did not discuss requirements or challenges bound to the parallelization (e.g., in the area of checkpoint creation and distribution), nor did they provide a perspective on the possible gain from this acceleration method. This is in contrast to our work, which provides this and presents a working prototype.

A combination of native execution with checkpointing and deterministic replay has actually been implemented only in JetStream [211], a recent tool to parallelize dynamic information flow tracking. A major limitation of JetStream is its restriction to single user-mode processes. SimuBoost, on the other hand, targets the full system and thus supports holistic system analysis, including groups of processes as well as privileged kernel-mode components – which is indispensable for operating system research. Moreover, contrary to SimuBoost, JetStream splits the execution into intervals offline. That is, it first records an execution, then replays it to create checkpoints, producing one interval for each CPU, and finally reruns the whole execution in parallel using the simulator. While the authors found the approach to be very effective, the offline method considerably delays the start of simulations. SimuBoost creates checkpoints on-the-fly during the original run to allow immediate simulation and reduce turnaround time. To be effective, SimuBoost must thus create checkpoints continuously while at the same time it has to keep the overhead at a minimum so as to preserve interactivity. Covering the entire system and starting parallel simulations right away as in SimuBoost thus raises the bar for efficient (subsecond) interval checkpoint creation and distribution, where at the same time less semantic information is available and the data volume is higher compared to application-level checkpointing only.

The restriction to user-mode processes also applies to Shadow Profiling [184] and SuperPin [267]. These projects do use application-level deterministic replay, but besides missing full system support, they are limited in scalability by resorting to forking the simulator process instead of using checkpoints. The latter is also the case with pFSA [229], which is geared toward parallelizing the microarchitectural simulation of discrete samples[1]. Shadow Profiling only simulates individual samples and the evaluation did not exceed two parallel instances. SuperPin and pFSA are evaluated to the maximum degree of parallelism provided by the available test platforms. Including hyperthreads, the benchmarks, however, did not exceed 16 and 32 parallel instances, respectively. In most cases, hyperthreads also brought no performance improvement. Conversely, the checkpointing in SimuBoost allows distributing the simulation over multiple physical hosts, thereby boosting the available hardware parallelism and enhancing scalability.

Another project related to SimuBoost has been presented by Girbal et al., called DiST [101]. It implements parallelization of microarchitectural simulations across multiple physical hosts but *without* using checkpoints. Instead, each node fast-forwards to the respective interval by executing the workload with functional simulation. Hence, DiST leverages the speed difference between microarchitectural and functional simulation, similar to the way SimuBoost leverages the speed difference between functional simulation and hardware-assisted virtualization. Yet, an important difference between both approaches is that the checkpoints relieve SimuBoost from having to fast-forward. The authors of DiST report this to be a major scalability issue and consequently advocate the use of checkpoints instead. Due to its dependency on pure functional simulation, DiST also cannot faithfully handle interactive workloads.

To conclude, we can state that splitting the simulation time into parallelizable epochs has proven in the past to provide considerable acceleration potential. However, none of the existing solutions based on this principle applies to continuous functional full system simulation. To the best of our knowledge, SimuBoost is the first approach to enable parallelization of interactive and long-running functional full system simulations.

In addition, this work contributes valuable insights in the area of efficient contiguous checkpointing, relevant also to related research areas (e.g., fault tolerance systems like Remus [72]). Furthermore, only marginal research has been done on how to perform heterogeneous deterministic replay at the machine level; with no solution, including ReVirt, being available to the research community[2]. SimuBoost, on the other hand, is freely accessible at `https://github.com/simutrace/`.

[1] pFSA does not provide any means to perform replay.

[2] The only projects covering full system heterogeneous deterministic replay have been undertaken in closed source by VMware for their proprietary VMware Workstation product [63, 64, 286] and by Yan et al. for V2E [287]. Unfortunately, the authors did not publish their source code and multiple attempts to receive a copy remained unsuccessful.

4.4 Conclusion

Despite the fact that the parallelization of simulation time has been successfully leveraged to substantially accelerate simulation in general, no previous work to date has applied this method to functional full system simulation. SimuBoost strives to represent the first workable prototype. The core idea is to run the workload of interest in an interactive hardware-assisted virtual machine. SimuBoost periodically takes checkpoints with high frequency, creating disjoint intervals. Then, the checkpoints are distributed in a simulation cluster to start parallel simulations of the respective intervals. Heterogeneous deterministic replay guarantees that simulations precisely reproduce the execution observed in the virtual machine and that functional continuity is maintained.

While the principle behind SimuBoost is easy to describe, the actual implementation comes with many intricate technical challenges. We need a performance model to select the optimal interval length for a certain configuration. Checkpoints have to be created with high frequency, while at the same time SimuBoost must keep the downtime and run-time overhead low in order to maintain interactivity and representativeness. To start simulations with low latency, SimuBoost has to quickly transfer these frequent checkpoints over the network and spawn hundreds to thousands of individual simulations over the course of a single workload. To fulfill the portability goal it is desirable that this process can still be handled by commodity network infrastructure such as Gigabit Ethernet. And finally, with only little experience in heterogeneous full system replay among the research community and no prior project to build on, implementing such a mechanism from scratch comes with its own set of challenges. With Chapters 6 to 8, we subsequently dedicate one chapter to each of these four areas, before presenting a thorough evaluation of SimuBoost's acceleration performance in Chapter 9.

> We have implemented our prototype in QEMU/KVM 2.6.5 (§ 2.2.5) on Linux 4.3. If not stated otherwise, all benchmarks presented in the following four chapters have been run using our main evaluation setup (System V/1) as summarized in § 9.1.1 on page 189 and the following. Results are the median of five runs and error bars indicate the empirical 0.025 and 0.975 quantiles – i.e., they cover 95% of samples. For box plots, we indicate the empirical 0.25 and 0.75 quantiles (50% of samples) with the box and the 0.025 and 0.975 quantiles with the whiskers. A detailed description of the benchmarks can be found in § 9.1.2.

Chapter 5

Performance Model

An open question in our description of SimuBoost so far is the determination of the right interval length. Similarly, we want to estimate the speedup that is attainable with a certain configuration. The performance model for SimuBoost aims to fill this gap.

In offline approaches such as trace-driven simulation, where all required information is available prior to simulation, the run time T_{vm} of the workload in the hardware-assisted virtual machine can simply be split into N equally-sized chunks, with N being the number of nodes. The interval length L is then $L = \frac{T_{vm}}{N}$. Assuming that on average the simulation time for every interval varies only marginally, the simulation load is uniformly distributed and the parallel simulation time roughly corresponds to the simulation time of a single interval (compare Figure 4.1).

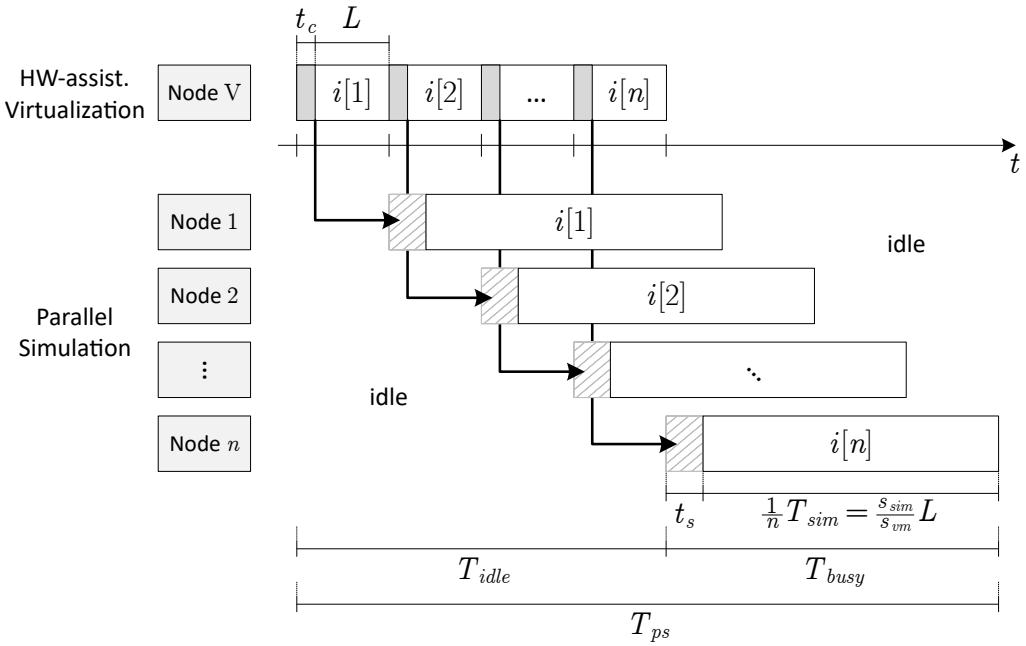
However, this model is not applicable to SimuBoost because SimuBoost creates checkpoints on-the-fly – i.e., online. This much more resembles a producer-consumer problem, where the interval length determines the production rate, and the number of nodes and the simulation time per interval limit the consumption rate. In practice, additional factors such as the downtime during checkpointing and the checkpoint loading time come into play.

To make predictions on the run-time characteristics that can be expected from SimuBoost, we have developed a corresponding mathematical model [89, 219, 225]. It provides information about the interval length that should be chosen for a given workload and hardware configuration to achieve the best possible speedup. It also allows us to estimate the parallel simulation time and identify important metrics that affect the performance.

In Section 5.1, we first describe the optimal case, in which a sufficient number of nodes is available to reach the maximum speedup for a given scenario. In Section 5.2, we then extend this model to be applicable in cases where only a limited amount of nodes is at the researcher's disposal.

5.1 Optimal Setup

SimuBoost takes checkpoints periodically throughout the run time of the experiment. Parallel simulations may start when a checkpoint becomes available and the non-deterministic events of the respective interval have been collected – i.e., at the end of the interval in the virtualization stage. To reach a maximum degree of parallelization and thus acceleration, intervals need to be simulated as soon as they become available. The optimal model therefore assumes that there is always a free node which can immediately start a simulation.



T_{vm}	:= Total time for hardware-assisted execution (without SimuBoost)
T_{sim}	:= Total time for serial simulation (without SimuBoost)
T_{ps}	:= Total time for parallel simulation
T_{idle}	:= Idle time of node n
T_{busy}	:= Busy time of node n
L	:= Interval length
s_{vm}	:= Slowdown of virtualization due to SimuBoost
s_{sim}	:= Slowdown of simulation compared to virtualization (T_{vm})
t_c	:= Checkpointing downtime
t_s	:= Simulation start-up time

* All times and lengths are in seconds [s], and $s_{vm}, s_{sim} \geq 1$.

Figure 5.1: The parallelization hides the execution time for all but the last simulation. The overall parallelized run time is thus the sum of the idle and busy times of the nth node.

Let $N \in \mathbb{N}$ be the number of nodes, and let $n \in \mathbb{N}$ be the number of intervals of length L. Then a simple way to comply with this assumption is:

$$N = n \tag{5.1}$$

That means for every interval, we have exactly one dedicated simulation node as illustrated in Figure 5.1. Using more than n nodes (i.e., $N > n$) does not provide any benefit because these nodes would not receive any work.

5.1.1 Parallel Simulation Time and Speedup

The first metric we can determine using Assumption 5.1 is T_{ps}, which is the total run time of a parallel simulation with SimuBoost, starting with the virtualization stage. T_{ps} thus represents the time a user has to wait from the beginning of an experiment until the simulation of the last interval finishes.

Since today it is common to execute workloads in virtual machines even in productive use, we take the run time T_{vm} with regular hardware-assisted virtualization as the baseline. However, when executing the workload with SimuBoost, the checkpointing and recording of deterministic events incurs overhead which slows down the hardware-assisted execution, mainly for two reasons. First, additional VM exits occur while the VM is running, for example, for event recording. We denote this overhead by the slowdown factor s_{vm}. Second, the checkpointing suspends the virtual machine once per interval to take a consistent snapshot of the machine's state. This further prolongs the effective run time of the workload in the virtualization stage. We refer to the downtime induced by a single checkpoint with t_c. Therefore, when SimuBoost takes a checkpoint every $(t_c + L)$ seconds in wall-clock time, the VM has performed the same amount of work as in $\frac{L}{s_{vm}}$ seconds without SimuBoost.

The first component that determines T_{ps} is the idle time $T_{idle}(n)$, which is the time the last node has to wait until the last interval in the workload has been executed. Only then, the simulation of the last interval can start (see Figure 5.1):

$$T_{idle}(n) \;:=\; n(t_c + L) \tag{5.2}$$

The second component that determines T_{ps} is the performance of the simulation itself. If we assume that the effort to simulate the workload (i.e., instructions, I/O, etc.) is evenly distributed over the entire run time[1], the simulation of every interval takes roughly the same time. Let s_{sim} be the slowdown factor from hardware-assisted virtualization (T_{vm}) to conventional full system simulation,

[1] In practice, this may not hold if the workload exhibits clear phase behavior. Nevertheless, for reasons of simplicity, we ignore this in the model and assume uniform load.

including the overhead for analysis. Then $\frac{L}{s_{vm}} s_{sim} = \frac{s_{sim}}{s_{vm}} L$ is the net simulation time of a single interval and $T_{sim}(n)$ is the overall time for serial simulation:

$$T_{sim}(n) \;:=\; n \cdot \frac{s_{sim}}{s_{vm}} L \tag{5.3}$$

Since each simulation starts off a checkpoint, we have to account for the time that is required to transfer the checkpoint to the target node, initialize the simulator, and load the checkpoint. We denote this start-up time by t_s. This gives us the total simulation time per interval:

$$T_{busy}(n) \;:=\; t_s + \frac{1}{n} T_{sim}(n) \tag{5.4}$$

Considering a 1:1 mapping between nodes and intervals – which is the premise in the optimal setup – there is always a free node available to simulate a newly arriving interval. Together with the uniform interval simulation time, this effectively hides the parallel simulation of all intervals, except the last one (i.e., n), behind $T_{idle}(n)$. We can therefore define the parallel simulation time $T_{ps}(n)$ as:

$$T_{ps}(n) \;:=\; T_{idle}(n) + T_{busy}(n) \tag{5.5}$$

In practice, the number of intervals is not directly controlled by the user. Instead, the user configures an (optimal) interval length L, and the number of intervals results from the given run time T_{vm} of the workload. It is thus more useful to define T_{ps} in terms of L. Since with SimuBoost the effective run time extends to $s_{vm} T_{vm}$, we can calculate the number of intervals with:

$$n(L) \;:=\; \left\lfloor \frac{s_{vm} T_{vm}}{L} \right\rfloor \tag{5.6}$$

The use of the floor function is required because the model assumes that every interval has the length L. If $s_{vm} T_{vm}$ is not a multiple of L, that is, $s_{vm} T_{vm} = nL + r$ with $r > 0$, we thus cut off the remaining run time r and do not include it in the parallel simulation as the user can always run the VM for one interval longer, if necessary[2]. For T_{ps}, we receive:

$$T_{ps}(n(L)) \;:=\; T_{idle}(n(L)) + T_{busy}(n(L)) \tag{5.7}$$

$$=\; \left\lfloor \frac{s_{vm} T_{vm}}{L} \right\rfloor (t_c + L) + t_s + \frac{s_{sim}}{s_{vm}} L \tag{5.8}$$

The speedup with SimuBoost over conventional serial simulation is then:

$$S(n(L)) \;:=\; \frac{T_{sim}(n(L))}{T_{ps}(n(L))} \tag{5.9}$$

[2] In practice, T_{ps} is determined by the completion time of the last interval simulation, irrespective of the index of the interval. The model assumes this to always be interval $i[n]$. If we would simulate r, we would allocate a new node like for every other interval and $i[n+1] \triangleq r$ would be the last interval. However, depending on the length of $i[n+1]$, the simulation time might be hidden by $i[n]$ – i.e., $i[n+1]$ completes before $i[n]$. The model would need to take this into account by finding the maximum completion time between $i[n+1]$ and $i[n]$. For simplicity, we instead cut off r and require the user to prolong the overall recording phase, if necessary.

5.1.2 Optimal Interval Length

If we would have an unlimited number of nodes to our disposal, we intuitively could just decrease L (i.e., increase n) to improve speedup. This would raise the degree of parallelism and improve efficiency by shortening the compulsory idle phases at the beginning and the end (i.e., the "stairway" in Figure 5.1 becomes taller and steeper). At the same time, low interval lengths would reduce the run time of the last interval, which greatly effects T_{ps}. However, with shorter intervals the overall overhead for checkpoint creation, distribution, and loading increases, which impairs the resulting speedup and eventually becomes a limiting factor. Conversely, choosing L too long leads to poor parallelism, which also hurts the speedup. Figure 5.2a exemplarily illustrates the model-predicted relationship between speedup and interval length for a Linux kernel build[3].

To determine the optimal interval length L_{opt} for a given scenario, we would need to solve $\frac{\partial}{\partial L} S(n(L)) = 0$ for L. However, due to the floor function in $n(L)$, $S(n(L))$ is not differentiable (see Figure 5.2b). At each discontinuity d, the number of intervals decreases by one if we increase L (i.e., $\lim_{L \to d^-} r = 0$), or decreases by

[3] Parameters for Linux kernel build: $T_{vm} = 435\,\text{s}$, $t_s = 0.44\,\text{s}$, $t_c = 0.012\,\text{s}$, $s_{sim} = 53$, $s_{vm} = 1.107$. Values have been determined experimentally with a prototype of SimuBoost for $L = 1\,\text{s}$.

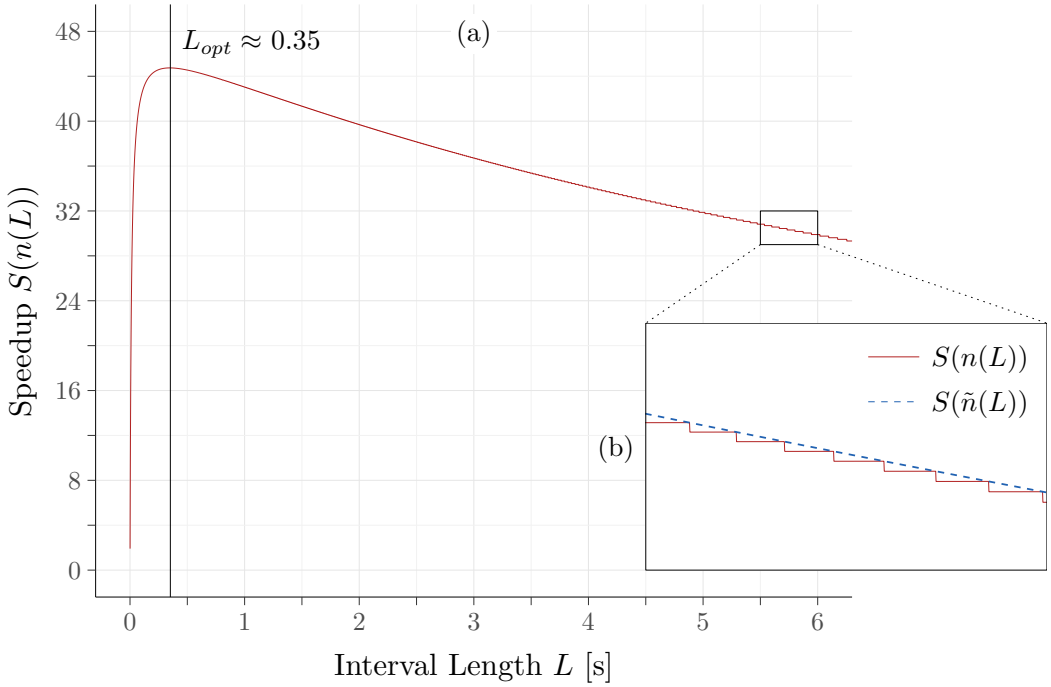


Figure 5.2: Choosing the right interval length is crucial. Speedup becomes limited by overhead for too short values of L; too long values lead to poor parallelism. The optimal interval length is marked with L_{opt}.

one if we increase L (i.e., $\lim_{L \to d^+} r = L$), respectively. This affects the degree of parallelism ($N = n$) and creates the (small) steps in the speedup function.

Besides computing the optimal interval length numerically, we can use the differentiable approximation $S(\tilde{n}(L))$, with:

$$\tilde{n}(L) \quad := \quad \frac{s_{vm} T_{vm}}{L} \tag{5.10}$$

$S(\tilde{n}(L))$ equals $S(n(L))$ for every L where $r = 0$ and interpolates in between. As shown in Figure 5.2b, $S(\tilde{n}(L))$ overestimates the speedup between steps compared to the predicted non-differential solution because the increase in saved simulation time due to parallelization is larger than the increase in simulated workload time. We can determine the relative error between $S(\tilde{n}(L))$ and $S(n(L))$ with:

$$\Delta S(L) \quad := \quad \frac{S(\tilde{n}(L))}{S(n(L))} \tag{5.11}$$

Figure 5.3 shows the relative error around the corresponding numerically determined optimal interval lengths. For all benchmarks, the divergence stays below 0.2%, but rises for ascending interval lengths. However, this is not a problem because the optimal setup prefers a high degree of parallelism, which in turn requires a large number of intervals. L_{opt} can thus be expected to lie in the range

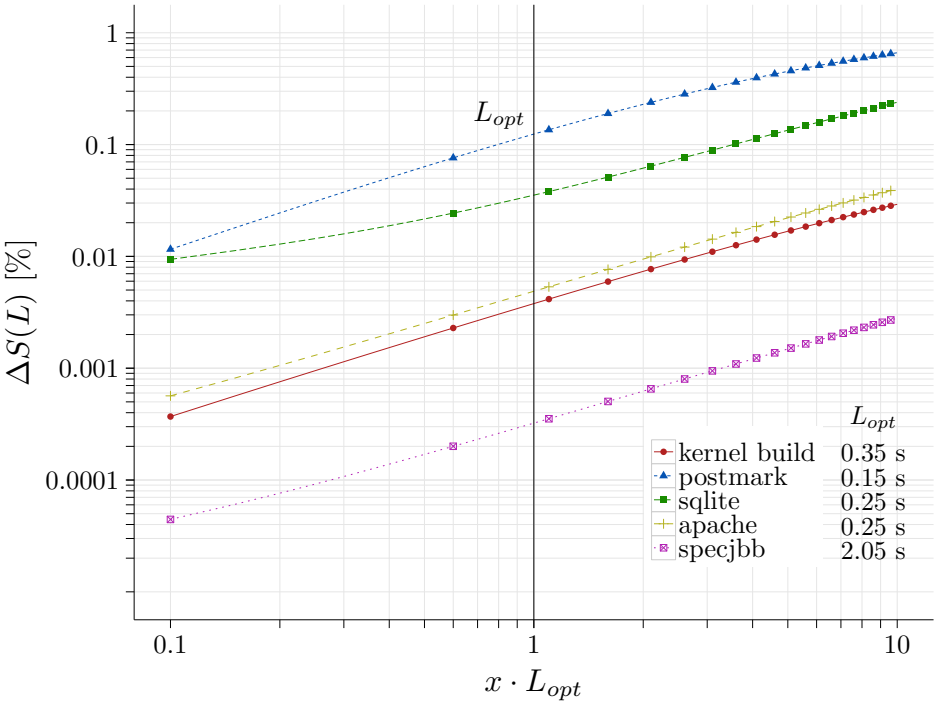


Figure 5.3: The error of $S(\tilde{n}(L))$ compared to $S(n(L))$ around the numerically determined L_{opt} is below 0.2% for every benchmark.

of short interval lengths, where the step width and therefore the error is small. Subsequently, we conclude that $S(\tilde{n}(L))$ is sufficiently accurate for calculating L_{opt} and solve $\frac{\partial}{\partial L}S(\tilde{n}(L)) = 0$ for L. Since we can safely assume that all parameters are positive, we omit the negative result and get:

$$L_{opt} := \sqrt{\frac{s_{vm}^2 T_{vm} t_c}{s_{sim}}} \tag{5.12}$$

$$S_{opt} := S(L_{opt}) \tag{5.13}$$

5.1.3 Optimal Number of Nodes and Efficiency

The speedup S_{opt} requires that SimuBoost can allocate a sufficient number of simulation nodes. Otherwise, the interval production rate exceeds the consumption rate, which invalidates the basic premise of the optimal case. In a naive implementation, every interval is simulated on a different node, that is, $N = n$. Utilizing more than n nodes does not deliver any additional speedup as these nodes will not receive any work. However, even choosing $N = n$ wastes computational resources because when nodes complete their interval, they idle for the rest of T_{ps}. Instead, it is more efficient to reuse nodes whenever they run idle.

Assuming that intervals are created periodically and that the simulation of every interval takes roughly the same time, intervals complete at the same rate they are created – producing uniform steps on both sides of the "stairway". In this case, new nodes are required only until the first simulation finishes, which is $T_{ps}(1)$. Subsequent intervals can be scheduled on previously allocated nodes that ran out

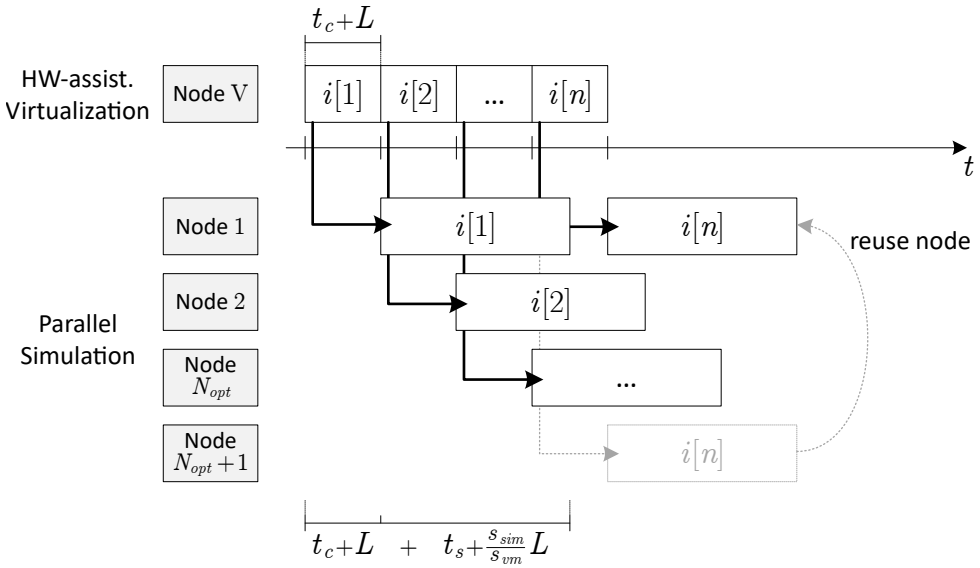


Figure 5.4: New nodes are required until the first simulation finishes. Subsequent intervals can be scheduled on previously allocated nodes.

of work (see Figure 5.4). The necessary number of nodes N is hence the number of new intervals created until $T_{ps}(1)$. Since we produce one interval every $(t_c + L)$ seconds, we can compute $N(L)$ as follows:

$$\begin{aligned} N(L) &:= \left\lfloor \frac{T_{ps}(1)}{t_c + L} \right\rfloor = \left\lfloor \frac{t_c + L + t_s + \frac{s_{sim}}{s_{vm}} L}{t_c + L} \right\rfloor \\ &= \left\lfloor \frac{t_s + \frac{s_{sim}}{s_{vm}} L}{t_c + L} + 1 \right\rfloor \end{aligned} \tag{5.14}$$

$$N_{opt} := N(L_{opt}) \tag{5.15}$$

We can then calculate the efficiency E of our approach by looking at the speedup that can be achieved for the number of nodes used:

$$E(n(L)) := \frac{S(n(L))}{N(L)} \tag{5.16}$$

$$E_{opt} := E(L_{opt}) \tag{5.17}$$

As an example, we apply the model to a Linux kernel build and choose the same parameters as in Figure 5.2. With a slowdown factor $s_{sim} \approx 53$, the workload would run 6 h and 25 min with conventional serial simulation. For $L_{opt} \approx 0.35$ s, our model predicts a total parallel simulation time $T_{ps}(L_{opt})$ of 8 min and 35 s. This would be a speedup of about $S_{opt} \approx 44$. To achieve this, we would require $N_{opt} = 48$ nodes, which would give us a parallelization efficiency of $E_{opt} \approx 93\%$.

5.2 Constrained Setup

The current model assumes that for every submitted checkpoint there is a free node that can immediately start a simulation. However, in practice, this is often not the case because the hardware setup is too small to provide a sufficient number of nodes. That is:

$$N < N_{opt} \tag{5.18}$$

The current model is thus a convenient way to estimate the maximum benefit one can expect from SimuBoost, but to be practical, the model requires an extension that allows predicting the benefit (and optimal interval length) for smaller hardware configurations.

If the number of nodes is not sufficient to keep up with the production rate of checkpoints, at some point all nodes are busy and new intervals need to be queued for later simulation (see Figure 5.5). As soon as a simulation finishes, the respective node can then fetch a checkpoint from the queue and continue simulation.

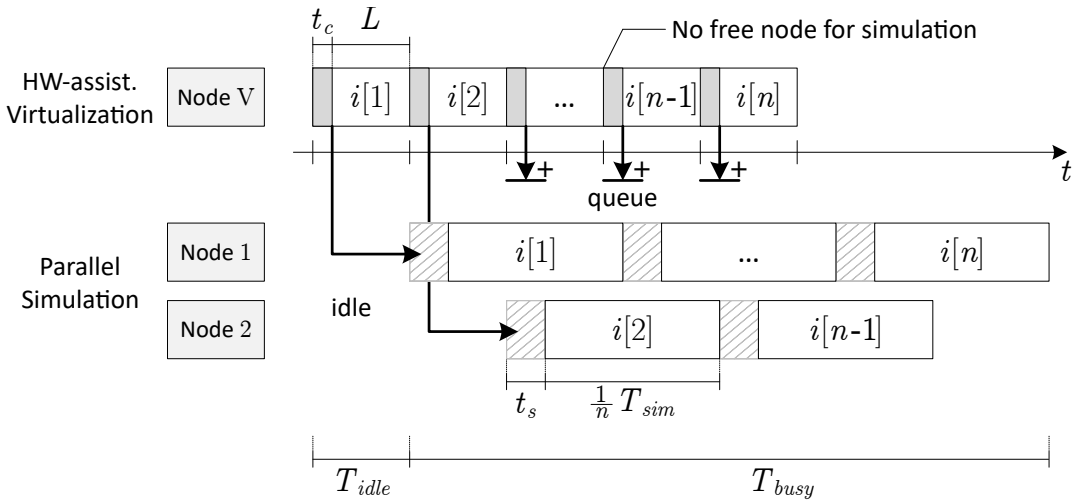


Figure 5.5: If too few simulation nodes exist, the production rate exceeds the consumption rate and checkpoints need to be queued.

5.2.1 Parallel Simulation Time and Speedup

If we assume that (1) all simulations take roughly the same amount of time and (2) simulations are distributed uniformly among the available nodes, we can infer that the node which simulates the last interval determines the overall parallel simulation time (compare Figure 5.5). In the optimal setup, where $N = n$, this is always the last node[4]. In a constrained setup, where simulations need to be queued, this is not the case. However, the general idea to calculate T_{ps} by adding the idle and busy times of the node that simulates the last interval is still applicable. So for a constrained setup we have to determine T_{idle} and T_{busy} as before, but for the node that simulates the last interval.

With uniform simulations, we can achieve optimal assignment of nodes to intervals by utilizing simple round-robin scheduling. We can thus find the index i of the node that simulates the last interval with $((n-1) \bmod N) + 1$, where $i \in [1, n]$. This lets us define:

$$T_{idle}(N,n) \;:=\; (((n-1) \bmod N) + 1)(t_c + L) \tag{5.19}$$

Node i has to simulate $\lceil \frac{n}{N} \rceil$ intervals. The busy time of the node can therefore be computed with:

$$T_{busy}(N,n) \;:=\; \left\lceil \frac{n}{N} \right\rceil (t_s + \frac{1}{n} T_{sim}) \tag{5.20}$$

This results in a parallel simulation time of:

$$T_{ps}(N,n) \;:=\; T_{idle}(N,n) + T_{busy}(N,n) \tag{5.21}$$

$$=\; (((n-1) \bmod N) + 1)(t_c + L) + \left\lceil \frac{n}{N} \right\rceil (t_s + \frac{1}{n} T_{sim}(n)) \tag{5.22}$$

[4] If only N_{opt} nodes are deployed, the index of the last node changes, but as no queuing is involved, this does not affect T_{ps} and is mathematically equivalent.

Verification

To verify that $T_{ps}(n)$ is just a special case of the more general $T_{ps}(N,n)$, we derive Equation 5.5 from Equation 5.21 by choosing $N = n$:

$$\begin{aligned} T_{ps}(n,n) &= (((n-1) \bmod n)+1)(t_c+L) + \left\lceil \frac{n}{n} \right\rceil (t_s + \frac{1}{n} T_{sim}(n)) \\ &= ((n-1)+1)(t_c+L) + \lceil 1 \rceil (t_s + \frac{1}{n} T_{sim}(n)) \\ &= n(t_c+L) + (t_s + \frac{1}{n} T_{sim}(n)) \\ &= T_{idle}(n) + T_{busy}(n) \\ &= T_{ps}(n) \end{aligned}$$

■

However, $T_{ps}(N_{opt},n) \neq T_{ps}(n)$ because of the floor function in Equation 5.15 and due to the fact that $T_{ps}(N,n)$ does not incorporate the idle gaps that may happen between simulations (see Figure 5.4, between $i[1]$ and $i[n]$). The model should thus slightly underestimate the simulation time for N_{opt}.

With $n(L) := \left\lfloor \frac{s_{vm} T_{vm}}{L} \right\rfloor$ (Equation 5.6), we can express T_{idle}, T_{busy}, and T_{ps} in terms of the interval length L as before:

$$T_{idle}(N,n(L)) := (((\left\lfloor \frac{s_{vm} T_{vm}}{L} \right\rfloor - 1) \bmod N)+1)(t_c+L) \tag{5.23}$$

$$T_{busy}(N,n(L)) := \left\lceil \frac{1}{N} \left\lfloor \frac{s_{vm} T_{vm}}{L} \right\rfloor \right\rceil (t_s + \frac{s_{sim}}{s_{vm}} L) \tag{5.24}$$

$$T_{ps}(N,n(L)) := T_{idle}(N,n(L)) + T_{busy}(N,n(L)) \tag{5.25}$$

Equivalent to the optimal setup, the speedup S is:

$$S(N,n(L)) := \frac{T_{sim}(n(L))}{T_{ps}(N,n(L))} \tag{5.26}$$

Figure 5.6a depicts the estimated speedup for a Linux kernel build given various numbers of simulation nodes and interval lengths. The top-most line $S(48,n(L))$ closely follows the predicted optimal scenario (compare Figure 5.2), with N_{opt} being 48 for the kernel build. However, since $S(N_{opt},n) \neq S(n)$, L_{opt} is slightly different from the calculation with $S(n)$.

A noticeable difference to the optimal case is the (inverted) sawtooth-like shape, whose strength varies with the number of nodes and the interval length. In addition to this distinctive feature, the function still exhibits the small non-differential steps that we can also observe in the optimal case.

Figure 5.7 illustrates how these steps emerge. When the interval length increases the number of intervals gradually diminishes according to $n(L) := \left\lfloor \frac{s_{vm} T_{vm}}{L} \right\rfloor$ (Equation 5.6). A large step occurs each time the number of intervals falls down to a

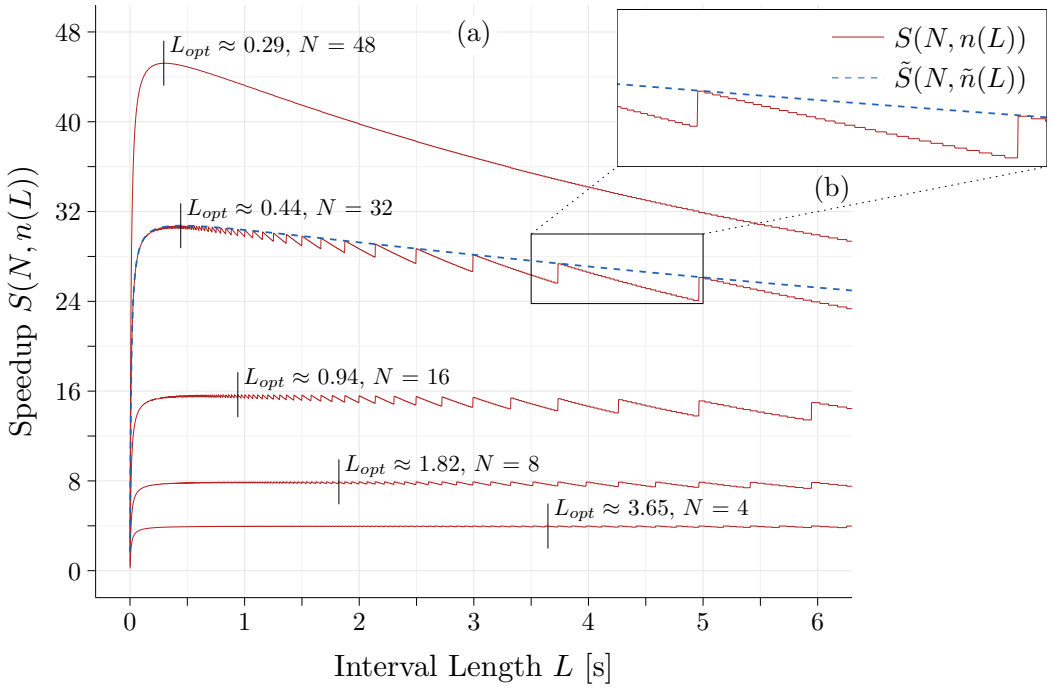


Figure 5.6: In addition to the non-differential steps known from the optimal case, we can observe an overlay with a sawtooth-like function, whose strength depends on the number of nodes and the interval length. The approximation (in blue) does not reproduce these features.

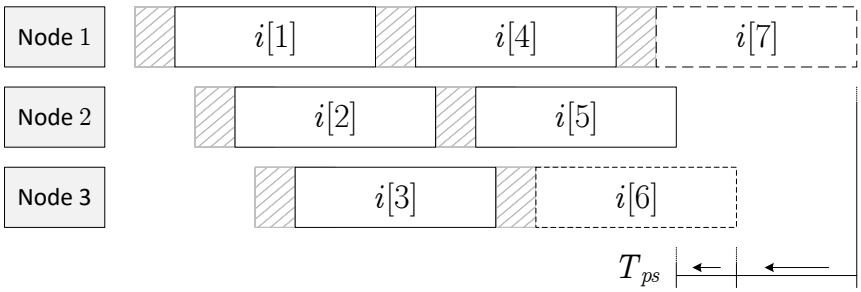


Figure 5.7: Large jumps occur when the number of intervals passes a multiple of the number of nodes. Effects on the start time and length of simulations have been omitted for simplicity.

multiple of the number of nodes. In the figure, this is the case when n drops from 7 to 6. The larger the interval length, the larger the jump because at the edge of each large step the first node simulates one interval more than any other node – i.e., an increase in L affects the total simulation time of the first node more than the others. The speedup, in turn, jumps up because T_{ps} experiences a noticeable reduction. Small steps in the speedup result from intermediate drops of n, in the figure, for example, from 6 to 5.

For a small number of nodes, both the small and large steps are less accentuated because each node has to simulate a large number of intervals. The effect of a drop by one interval, even at the edge of a large step, is thus relatively small. On the other hand, if N is (near) N_{opt}, the idle phase of the last node roughly corresponds to the simulation time of one interval. In consequence, the additional offset between the completion times of the first and last node (creating the large step) disappears.

It is also notable that, according to the model, the speedup becomes considerably less sensitive to the chosen interval length for a decreasing number of simulation nodes.

5.2.2 Optimal Interval Length

Just like in the optimal case, $S(N, n(L))$ is not differentiable. This prevents us from calculating L_{opt} for a given N directly. Therefore, we again approximate the speedup equation: we assume that simulations are uniformly distributed among the available nodes and that each node is busy for roughly the same amount of time. We consequently ignore some of the effects we have discussed earlier in order to reduce the complexity of the model. Based on this simplified premise, it is enough to look at the completion time of the last node (just like in the optimal case). We then receive the following set of simpler equations:

$$\tilde{T}_{idle}(N,n) := N(t_c + L) \tag{5.27}$$

$$\tilde{T}_{busy}(N,n) := \frac{n}{N}(t_s + \frac{1}{n}T_{sim}(n)) \tag{5.28}$$

$$\tilde{T}_{ps}(N,n) := \tilde{T}_{idle}(N,n) + \tilde{T}_{busy}(N,n) \tag{5.29}$$

$$\tilde{S}(N,n) := \frac{T_{sim}(n)}{\tilde{T}_{ps}(N,n)} \tag{5.30}$$

Figure 5.6 depicts the approximation for the exemplary scenario of $N = 32$ in blue. $\tilde{S}(N, \tilde{n}(L))$ matches $S(N, n(L))$ in points where $n = kN$ with $k \in \mathbb{N}$ (i.e., at large steps), and overestimates the speedup in between. However, just like in the optimal setup, L_{opt} can be expected to lie in the area of interval lengths where the error is negligible[5] (compare Figure 5.6a). It is thus sufficient to work with the estimated solution, especially considering the fading sensitivity to suboptimal interval lengths with a decreasing number of nodes. Furthermore, compared to the true physical run-time behavior, the model already makes various assumptions (e.g., equal interval simulation times) that are likely to affect the accuracy much more noticeably.

[5] If a more precise solution is desired, the approximated L_{opt} can be used as a starting point to numerically find the (predicted) true L_{opt}. In this course, it is sufficient to look at the interval lengths around the approximated L_{opt} where $n = kN$ as these mark local maxima.

We can therefore approximate L_{opt} by solving $\frac{\partial}{\partial L}\tilde{S}(N, \tilde{n}(L))) = 0$ for L:

$$L_{opt} := \frac{\sqrt{t_s s_{vm} T_{vm}}}{N} \tag{5.31}$$

This enables us to calculate the optimal interval length for a given scenario with a limited number of simulation nodes. While the performance model predicts a parallel simulation time of 8 min and 35 s for a Linux kernel build in the optimal case (i.e., 48 nodes), the model estimates T_{ps} with 16 min and 30 s to almost double when using half the number of nodes and $L_{opt} \approx 0.6$ s; this corresponds to near linear scaling. Accordingly, the speedup falls down to 23x. The efficiency can be calculated analogously to the optimal case, reaching around 97%. We compare model predictions with actual measurements in Chapter 9.

5.3 Conclusion

According to exemplary calculations, SimuBoost should be able to provide considerable acceleration with a high degree of scalability; at the same time preserving interactivity and delivering full temporal and spatial completeness.

Given a certain interval length, the performance model aims at estimating the parallel simulation time for both, (a) the optimal setup, where a sufficient number of simulation nodes is present to achieve the highest possible speedup, and (b) constrained configurations, which supply only a suboptimal number of nodes. In both cases, we use the differentiation of the speedup function to find the optimal interval length.

The model requires a set of input parameters of which some describe the particular workload at hand (e.g., its run time), whereas others characterize the performance of the mechanisms employed by SimuBoost with respect to the used hardware. The parameters can be acquired with a limited test run of the envisioned scenario.

Compared to the actual behavior on physical hardware, the model is subject to a set of simplifications. The most prominent one is the assumption that all intervals possess the same constant simulation time, irrespective of workload phase and selection of simulation node. The model also does not incorporate factors such as a potential rise in simulation slowdown due to shared CPU caches and limited memory bandwidth when locating multiple simulations on the same physical host (e.g., on SMP systems). Similarly, short interval lengths may reduce the efficiency of DBT code caches in simulators. Nevertheless, to keep the model's complexity as well as the number of input parameters in reasonable bounds, we have decided to abstract from these factors and rather design the model to be appropriate for grasping the general performance characteristics of SimuBoost and guide in the selection of the right interval length. The final evaluation in Chapter 9 clarifies to what extent the model can stand up to the task.

Chapter 6

Continuous Checkpointing

Checkpointing describes the process of capturing a consistent image of an entities' state at the time of taking. Checkpoints may optionally be persisted on a storage device or transferred over a network to a different system and subsequently be used to either revert an existing entity to the saved state or initialize a new entity. In this work, checkpointing refers to the state of a hardware-assisted virtual machine, i.e., a full system with its guest physical memory, secondary storage, and (virtual) devices such as the VM's vCPU.

We use the term *continuous checkpointing* to denote periodic checkpointing over the course of an experiment. Figure 6.1 illustrates the process in SimuBoost. While the workload of interest is running in a hardware-assisted virtual machine, SimuBoost snapshots the VM's state at regular intervals and queues the checkpoints for parallel simulation. This chapter deals with the design and implementation of this continuous checkpointing mechanism, including fast loading.

According to the performance model we developed in the previous chapter, SimuBoost requires a comparably high frequency of one checkpoint every few seconds

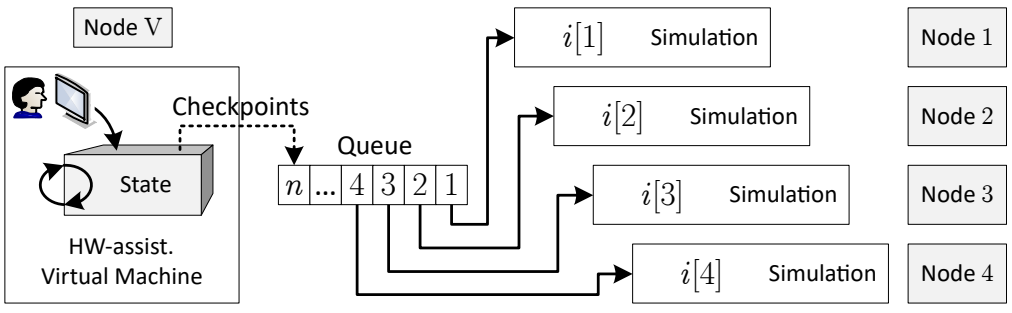


Figure 6.1: The workload runs in a hardware-assisted virtual machine. SimuBoost periodically checkpoints the VM and eventually uses these checkpoints to bootstrap parallel simulations.

up to subsecond intervals to be effective. The demands on the continuous checkpointing mechanism are therefore high since any costs are paid repeatedly in short succession. In § 2.3, we already elaborated on the metrics that can be consulted to measure checkpointing costs. Of particular interest to the checkpoint creation in SimuBoost are the *downtime* and the *probe effect* – i.e., the divergence of the checkpointed execution compared to the same run without SimuBoost, for example, as a result of additional page faults.

The frequent downtimes are problematic in two ways. First, injecting these downtimes changes the conditions under which a workload runs and in turn adds to the probe effect. Second, the periodic suspension becomes a critical factor if the downtime crosses the boundary from which on interactivity gets impaired. This is especially a concern in SimuBoost as restoring interactivity is a primary goal of our work. To avoid the latter, we set a minimum target of 100 ms for the downtime, as this value is generally not perceived by humans [178] and does not break a virtual machine's network connectivity [72]. However, to minimize the probe effect, we strive to further reduce the downtime – if this is possible and proportionate to design complexity.

The probe effect is of great importance to our work because we exactly reproduce the hardware-assisted execution – which is to some degree affected by checkpointing and event recording – via deterministic replay in the simulations. Any conclusions drawn from the simulations may thus be directly influenced by the induced overhead. Consequently, a high probe effect has the potential to compromise the representativeness of experiments and we must take care to keep it low. Defining a tolerable limit, however, is not generally possible as it depends for one on the metrics used to quantify the probe effect, and second on the question that should be answered by an experiment and what metrics are relevant in that course. Running a workload with SimuBoost might, for example, influence the effective quantum length and scheduling order of a compute-bound process (e.g., due to downtimes). This might, however, be irrelevant if an isolated instruction flow of this process and its kernel interaction should be extracted from the simulation.

Another non-trivial question is how to properly quantify the probe effect in the first place. Measuring it in its most general form, that is, as a divergence in the instruction stream and timing, requires some form of monitoring to be able to detect differences. This, in turn, cannot be done on contemporary computing platforms without inducing a probe effect itself. Alternatively, leveraging deterministic simulation as in Simics seems feasible. We could simulate executing the workload in a virtual machine with and without SimuBoost. However, even with a complex timing model, the significance of results from such examinations (for real hardware) is questionable; especially considering that even small differences in timing can greatly affect the outcome [29]. Nor does this answer the question of the results' relevance for a specific workload.

Nevertheless, to somehow quantify the probe effect, we choose to determine the application-specific performance degradation for the examined workload. Since most of our benchmarks process a fixed input set, we can measure this as an increase in run time. For SPECjbb, we take the average of the reported points over all warehouses as the basis. To illustrate the effect of checkpointing on the network, we additionally run the iperf3 benchmark. While these metrics do not provide any information on the true divergence in the execution flow, they do a good job in giving a general sense of the probe effect and they can be determined with negligible overhead. Furthermore, optimizing the checkpoint mechanism to exhibit a low performance degradation is also likely to reduce the divergence altogether.

In the following sections, we describe the design and implementation of the checkpointing mechanism in SimuBoost, using the downtime and the probe effect as our primary criteria for assessment. Since the deterministic replay relieves us from having to explicitly checkpoint the state of secondary storage (see Chapter 8), we focus on methods to save the guest physical memory – usually being the largest item anyway [189]. Nevertheless, the presented concepts are also applicable to block storage[1].

Section 6.1 elaborates on the benefits of incremental checkpoints and discusses techniques to detect modified guest data – so-called *dirty logging*. We also demonstrate how the downtime can be very effectively reduced by applying copy-on-write. Section 6.2 illustrates how SimuBoost organizes and asynchronously stores checkpoints on disk for later transfer to simulation nodes. This storage backend is also used for all experiments in Section 6.1. A method to perform fast checkpoint loading in the presence of deterministic execution is presented in Section 6.3. We conclude in Section 6.4.

6.1 Checkpoint Creation

The simplest method to capture a consistent image of the guest system is to suspend the virtual machine, synchronously copy all volatile state, and finally resume the execution – a method called *stop-and-copy (SnC)* [66]. Figure 6.2 depicts the resulting median downtime for various guest memory sizes, interval lengths, and benchmarks using this method. In each case, saving the state of the virtual devices (e.g., the vCPU) takes around 3 ms ($\triangleq$ 120 KiB). The remaining downtime originates from saving the guest physical memory. Even for small VMs with 256 MiB of RAM, stop-and-copy touches the limit of 100 ms. Larger VMs exceed the limit, with 8 GiB of RAM inducing a downtime of around 2 s, irrespective of the interval length and workload.

[1] To be able to test checkpointing independently from deterministic replay, we implemented lightweight virtual disk checkpointing in our prototype but disabled it in the following.

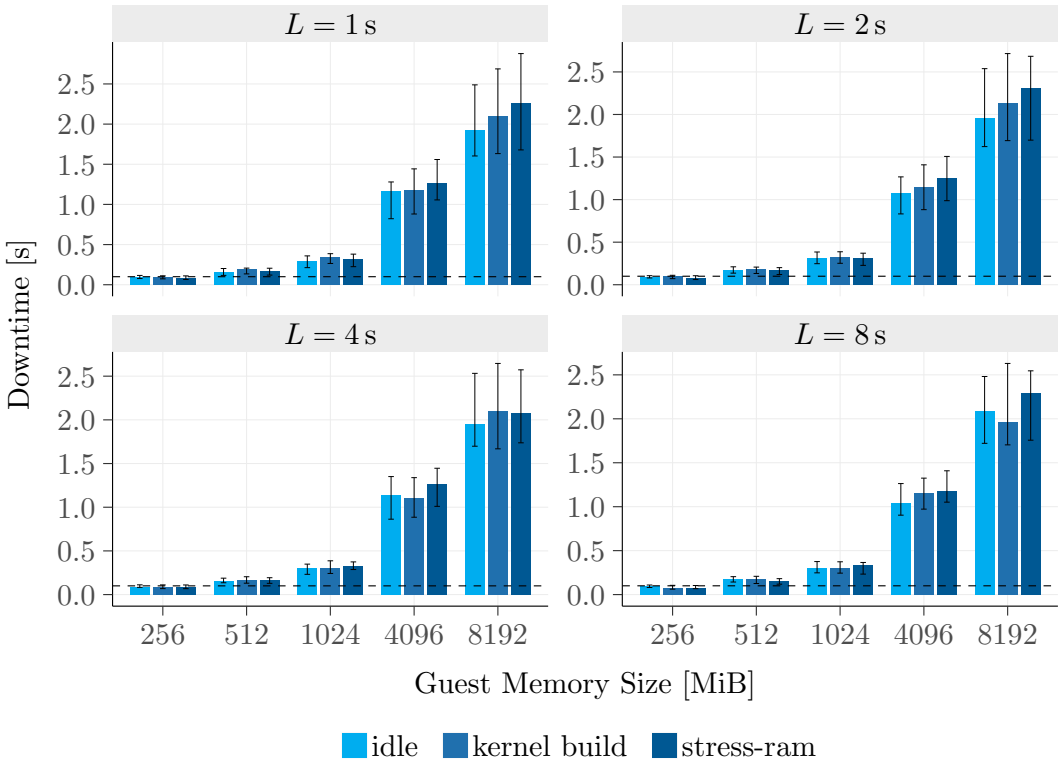


Figure 6.2: The downtime for stop-and-copy checkpointing touches the limit of 100 ms even for small VMs and quickly reaches multiple seconds for common configurations. Despite being highly sensitive to the VM size, the downtime is largely unaffected by the interval length L and type of workload; differences are within the variance.

As expected, the VM experiences a noticeable performance loss with stop-and-copy checkpointing (see Figure 6.3a), where the checkpointing frequency is a decisive factor for the overall run time. This is because the interval length determines the number of checkpoints and thereby the number of downtimes included. A sole increase in guest physical memory size from 256 MiB to 8 GiB with $L = 8\,\mathrm{s}$ only leads to a moderate 28% slowdown for a kernel build. However, increasing the checkpoint frequency to one checkpoint per second – which is still three times lower compared to the 300 ms intervals suggested by our model – almost quadruples the run time. As any rise in the downtime proportionally weights more when the number of checkpoints is higher, the run time becomes more sensitive to the guest physical memory size with increasing checkpointing frequency.

Running the iperf3 benchmark alongside a kernel build also reveals that the excessive downtimes clearly manifest in the network connection of the guest, pushing the mean bandwidth down to a third (see Figure 6.3b).

We can thus conclude that stop-and-copy checkpointing is not well suited for continuous checkpointing in general and for SimuBoost in particular. It impairs

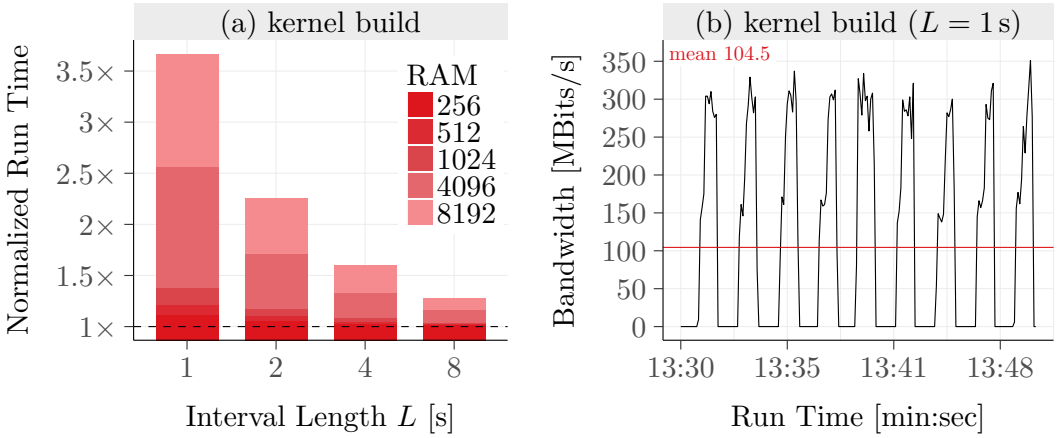


Figure 6.3: (a) With stop-and-copy the kernel build time almost quadruples in the most demanding configuration. (b) The downtimes are clearly visible in the network bandwidth.

interactivity (user and network) and inhibits a considerable run-time overhead. This is especially the case when the checkpointing frequency and guest physical memory size are high.

6.1.1 Incremental Checkpointing

Our first goal in improving the checkpointing performance is to bring the downtime below the limit of 100 ms so that we preserve interactivity. As a side-effect, this will in turn also lead to a reduction in run-time overhead. Since saving the state of virtual devices such as the vCPU has proven to be comparably inexpensive, we focus on the guest physical memory only. Nevertheless, all downtimes still include device copy time further on.

Application-level checkpointing, as for instance in JetStream [211], benefits from information about the address space layout and can thus distinguish between indispensable and recoverable data and between used and free memory. This allows saving downtime simply by reducing the amount of data that must be checkpointed. Possible ways to apply this technique to VMs and close the semantic gap between host and guest is to leverage paravirtualization or virtual machine introspection (VMI).

With paravirtualization, the guest actively propagates information to the hypervisor, for example, on recoverable file cache pages as well as free memory pages. Paravirtualization thus presumes error-free and reliable cooperation from the guest, otherwise checkpoints may get corrupted. For SimuBoost, corrupted checkpoints mean that the simulation would prematurely fail, preventing any analysis. This is especially unfavorable if malicious clients should be inspected.

With VMI, the hypervisor uses debug symbols to identify and parse the respective guest kernel data structures without active assistance from the guest. Although this makes VMI more robust against erroneous (and malicious) guests, VMI requires explicit support for a specific guest (e.g., a certain Linux version). However, always having to adapt the checkpointing mechanism – a central component of our research tool – to the guest OS is cumbersome. In addition, excluding pages from checkpoints based on VMI information alone can be problematic. For instance, the guest may erroneously access free pages due to an overflow or use-after-free bug. Without preserving these pages in the original execution, any effects of such bugs cannot be faithfully reproduced in the simulation.

A popular way to accomplish an effective but guest-agnostic reduction in downtime is to use *pre-copy* (see § 2.3.1); the prevailing technique for live VM migration. However, a drawback of pre-copy is that the reduction in downtime comes at the cost of repeated page transfers. From the perspective of checkpointing (rather than migration), this means we buy a reduced downtime by increasing the overall amount of memory checkpointed. This is also unfavorable if being avoidable.

In contrast to migration, which is a singular event, we can benefit from the periodicity in continuous checkpointing. Similar to pre-copy, we use the observation that a workload modifies only a limited set of pages within a certain time frame. A checkpoint can thus be reduced to these pages because all other pages can be received from previous checkpoints – a method called *incremental checkpointing*.

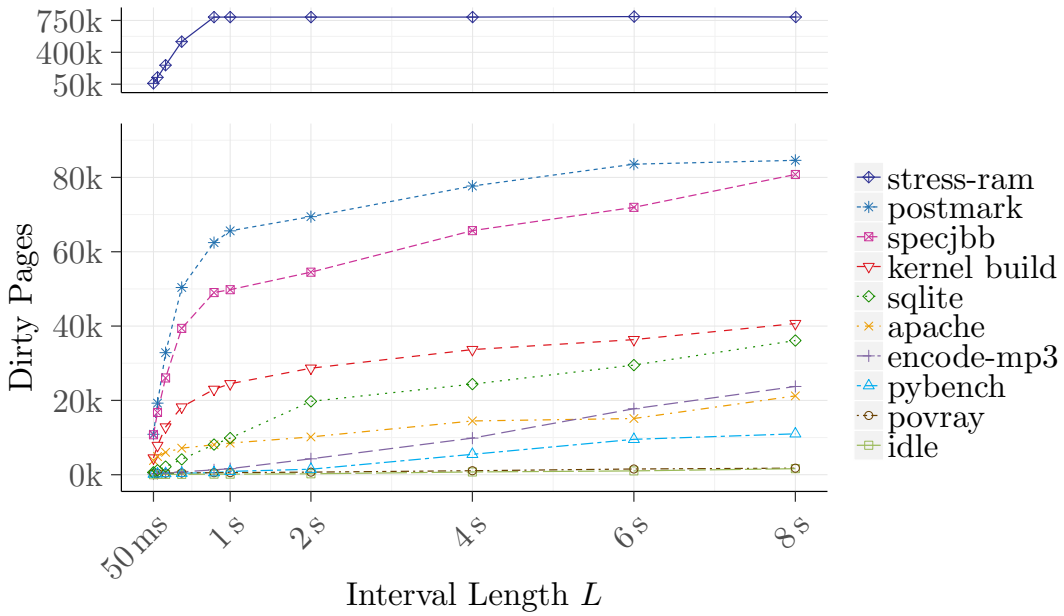


Figure 6.4: The number of dirty pages increases with the interval length, but remains within 85k pages (≈ 330 MiB) for most workloads and $L \leq 8$ s. Only stress-ram, the worst-case benchmark, greatly exceeds this rate. The kernel build proves to be a good indicator for the average case.

Figure 6.4 illustrates the median number of dirty pages per interval over various interval lengths and workloads[2]. The measurements illustrate three central advantages of incremental checkpointing compared to stop-and-copy:

1. The number of dirty pages per interval is generally considerably smaller than the total number of guest physical pages. Except for stress-ram, which dirties memory with maximum speed, all tested benchmarks remain below 70k pages per second ($\approx$ 270 MiB). This is also a greater saving than can be expected from paravirtualization or VMI.

2. Stop-and-copy has shown a strong dependency between run-time overhead and checkpointing frequency. With incremental checkpointing the number of dirty pages inherently decreases for shorter intervals, proportionally lowering the overhead for high checkpoint frequencies.

3. Each workload exhibits an individual page modification rate, depending on the operations carried out. Checkpointing compute-bound workloads (e.g., povray) or idle phases is thus generally very lightweight.

However, when looking at the four most demanding workloads in Figure 6.5 (i.e., a page modification rate over 20k pages/s), we can see that incremental checkpointing is not able to reliably push the downtime below the 100 ms mark. Whereas only two checkpoints during the initial expansion phase of the kernel build exceed the limit, almost 9% of checkpoints for postmark, and 17% of checkpoints for SPECjbb take too long. For the stress-ram benchmark, none of the checkpoints can be created within the permissible time frame. With 800 ms to 1200 ms the downtime is far too high for a fluent interactive experience.

Figure 6.5 also reveals another drawback of incremental checkpointing. Although it is generally advantageous that the downtime is proportional to the page modification rate because it makes the checkpointing of less demanding workloads more lightweight, this very dependence is also a disadvantage. Over the course of a workload's execution, the downtime may strongly fluctuate, which makes it difficult to predict and use as a constant factor in our performance model. It is thus desirable to (1) further decrease the downtime so that it is within bounds even for demanding workloads and (2) to make the downtime less volatile.

To accomplish the first goal we need to either further reduce the overall work for checkpointing or shift the existing work from the downtime to the run time of the workload – i.e., make the work asynchronous. For the second goal, it is necessary to break the dependency between downtime and workload, ideally, without losing the benefits we get from incremental checkpointing.

[2] In the following, we use *write protection* to determine dirty pages. For a comparison of dirty logging techniques, see § 6.1.2. We further look at the 4 GiB guest memory configuration only because differences with incremental checkpointing are (inherently) negligible.

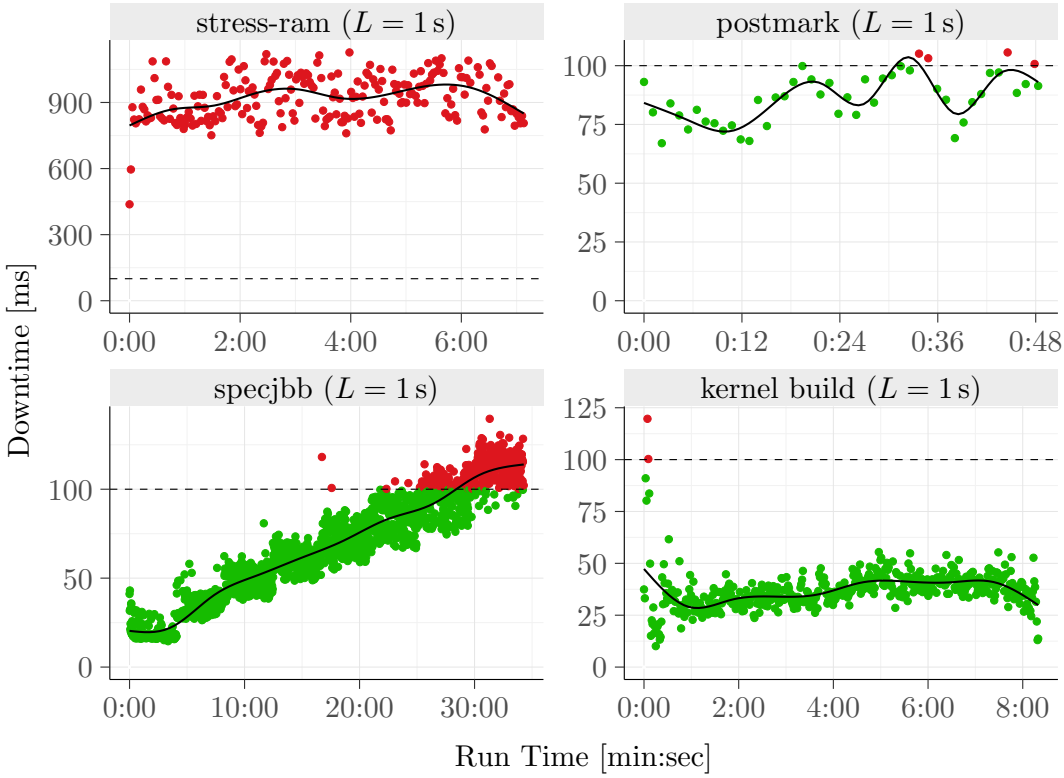


Figure 6.5: Incremental checkpointing cannot reliably keep the downtime below 100 ms for demanding workloads with more than 20k dirty pages/s. Since the downtime is proportional to the page modification rate, it strongly fluctuates. The regression line depicts the mean downtime.

The first can be achieved by employing *copy-on-write*. In this course, the hypervisor uses the downtime solely to apply write protections in the EPT, instead of actually copying data. Write protection is set for all pages that need to be included in the incremental checkpoint – i.e. all pages modified during the previous interval. This allows the hypervisor to take a consistent snapshot of the respective pages because it prevents the guest from making any further changes without first triggering the hypervisor via a page fault. Write access to all other guest pages as well as general read access remains permitted. After setting the write protection, the VM resumes execution. The checkpointing mechanism then asynchronously saves the guest pages and releases the write protection for every page copied. If the VM attempts to write to a protected page, the page fault handler in the hypervisor can copy the respective page out of line and restore write permissions so that the guest can continue its operation.

Since the copy operation accounts for most of the work associated with checkpointing, adding copy-on-write considerably reduces the downtime (see Figure 6.6). Although the downtime remains relatively high in the case of stress-ram, for all other benchmarks the downtime is in the range of 2.7 ms to 9.8 ms for all checkpoints and interval lengths. Using copy-on-write thereby also helps with fulfilling

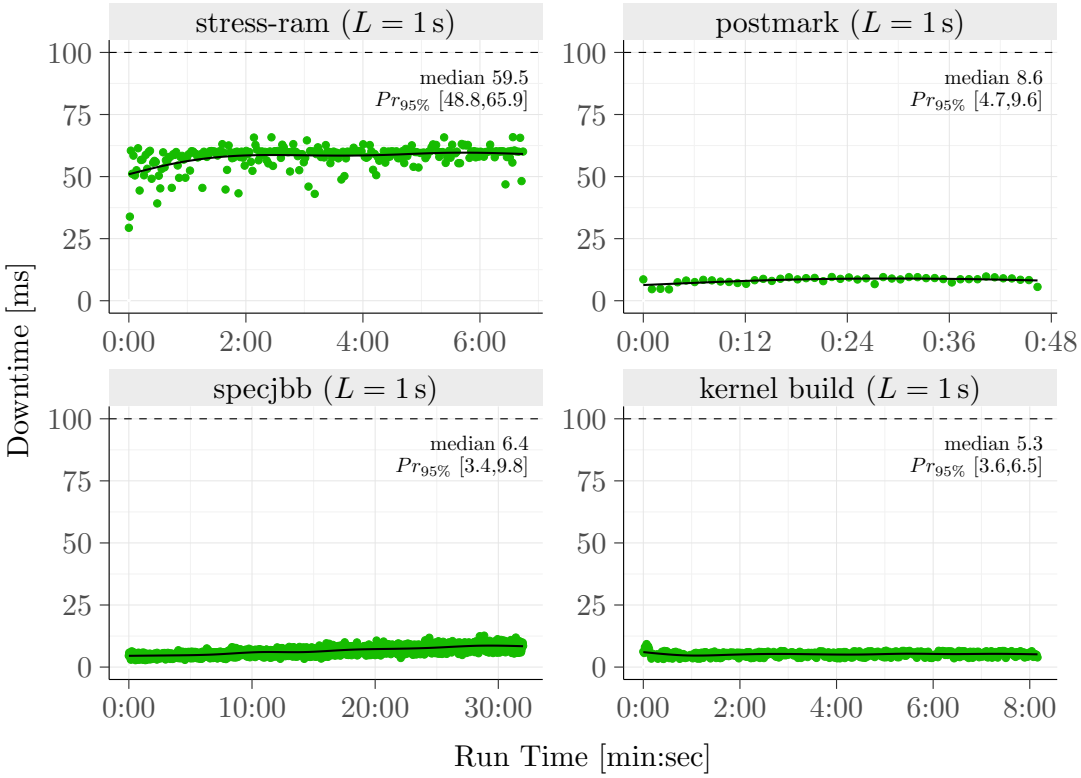


Figure 6.6: Incremental checkpointing with copy-on-write reliably pushes the downtime below 100 ms, even for stress-ram, with 66 ms being the highest value. For all setups, CoW reduces absolute variance within and across workloads. The regression line depicts the mean downtime. $Pr_{95\%}\,[a, b]$ denotes the empirical 0.025 and 0.975 quantiles.

the second goal, that is, making the downtime less volatile. Although the downtime still fluctuates over the course of a workload's execution[3], the reduced overall downtime leads to a smaller absolute variance. Hence, the individual workload phases – which can be clearly seen in Figure 6.5 – are much less pronounced.

Checkpointing Frequency

Since the duration of each interval is defined by the configured interval length L, any asynchronous operations such as copying pages must complete within this bound. Otherwise, the checkpointing solution is not able to sustain the checkpointing frequency and successive checkpoints must be delayed[4]. This defines a practical lower limit for the interval length.

[3] With copy-on-write, the downtime depends on the number of write protections to set, which in turn depends on the interval length and the current page modification rate of the workload.

[4] Alternatively, the VM could be suspended. However, this would prolong the downtime, which negatively affects interactivity and the run-time overhead. Delaying the next checkpoint instead acts as a autoregulation.

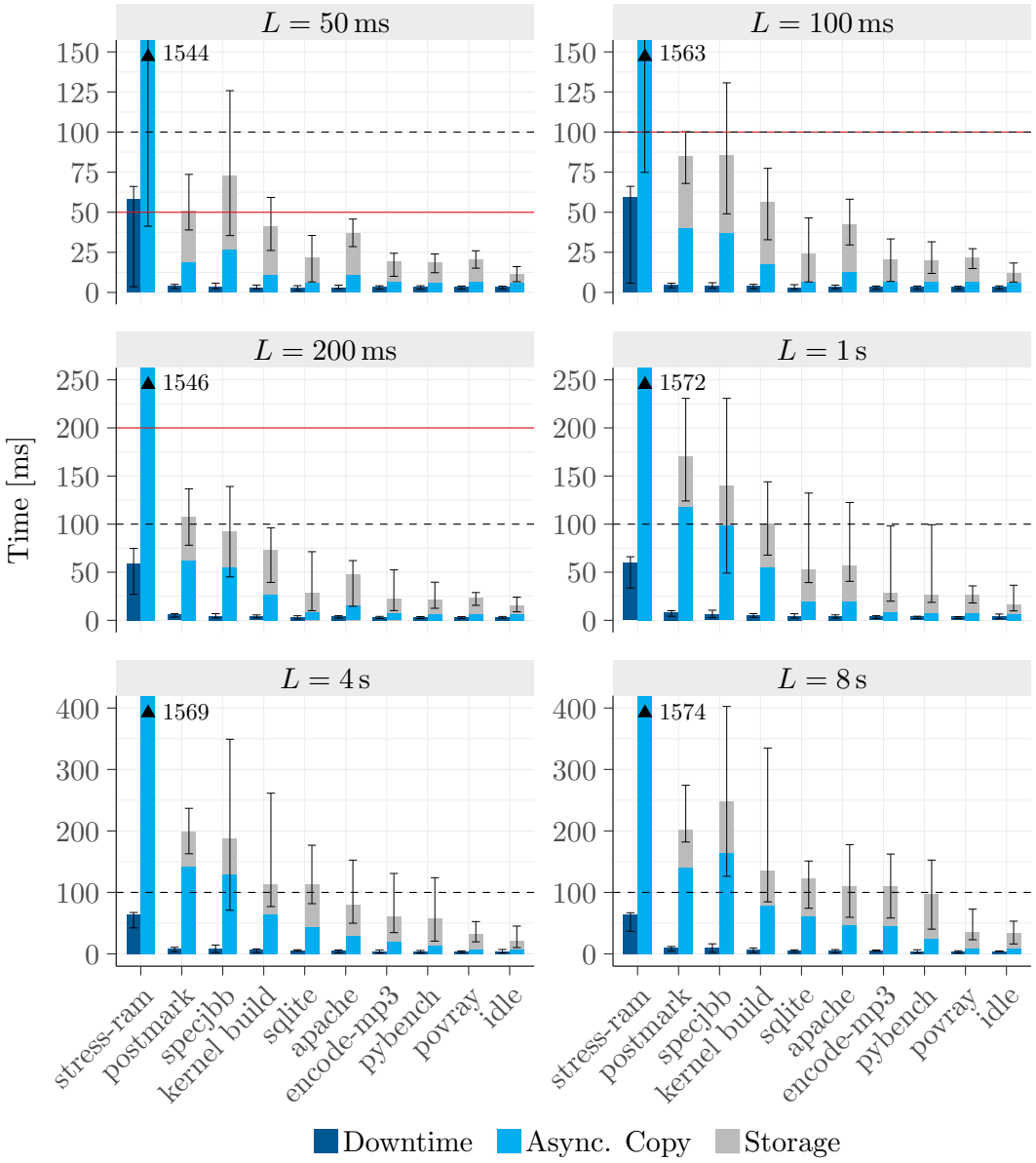


Figure 6.7: The downtime with incremental copy-on-write checkpointing is well below the 100 ms limit for all interval lengths and benchmarks. The solid red line marks the interval length as a hard limit for any asynchronous operations – i.e., the sum of asynchronous copy and data storage (see § 6.2 and § 7.2). Depending on the workload, the asynchronous operations can take too long for $L < 130\,\text{ms}$ (stress-ram: $L < 1.8\,\text{s}$), but otherwise complete timely. All three metrics are proportional to the page modification rate of the workload and the interval length. An exception to this is stress-ram. Since it touches memory so fast, changing the interval length has only little effect on the measurements, but merely decreases variance.

Figure 6.7 summarizes the required total time for asynchronously copying dirty pages and storing them as a checkpoint on disk using our storage backend[5]. The asynchronous processing time is typically much lower than the interval length. However, with $L = 50$ ms checkpoints may not complete in a timely fashion for workloads with a page modification rate over 20k pages/s. Only for $L \geq 130$ ms the processing time consistently remains below the interval length. An exception to this is stress-ram, for which the checkpointing frequency can be sustained only for $L \geq 1.8$ s. However, this is to be expected because stress-ram modifies memory so quickly that the asynchronous copy time alone must inevitably converge toward the time it takes to capture a full (i.e., not incremental) image of the guest memory. In fact, due to the overhead induced by (1) performing the copy asynchronously (e.g., shared memory bandwidth) and (2) superfluously using the data structures for dirty page identification, the pure copy time is even higher. It exceeds the stop-and-copy downtime for the 4 GiB guest memory configuration ($\approx$ 1260 ms, see Figure 6.2) by around 19%, although stress-ram touches only a 3 GiB buffer.

stress-ram	postmark	specjbb	kernel build	sqlite	apache
1800 ms	75 ms	130 ms	60 ms	40 ms	50 ms

gnupg	encode-mp3	pybench	povray	phpbench	idle
50 ms	25 ms	25 ms	30 ms	25 ms	20 ms

Table 6.1: Estimated shortest interval length per workload[6].

We can conclude that from the perspective of downtime and asynchronous processing time, incremental copy-on-write checkpointing is generally well-suited for fast continuous checkpointing and only fails to sustain the configured checkpointing frequency when demanding workloads meet very short intervals. An exception to this is stress-ram. However, we consider this scenario to be primarily of synthetic nature, as our results from the real-world benchmarks confirm.

Run-Time Overhead

To get a complete impression of the performance, we additionally have to account for the run-time overhead imposed on the workload (see Figure 6.8). With stop-and-copy checkpointing, the downtime is the only component affecting the run-time overhead. Furthermore, always copying the entire guest memory makes the run-time overhead insensitive to the properties of the workload executing in the VM – i.e., all workloads experience the same run-time overhead. This is not

[5]The storage backend performs full data reduction as necessary for network transfer over Gigabit Ethernet. See § 6.2 for a general introduction and § 7.2 for details on compression.

[6]The data does not account for the decreased number of pages to copy for values below 50 ms. Actual interval lengths may thus be a little shorter.

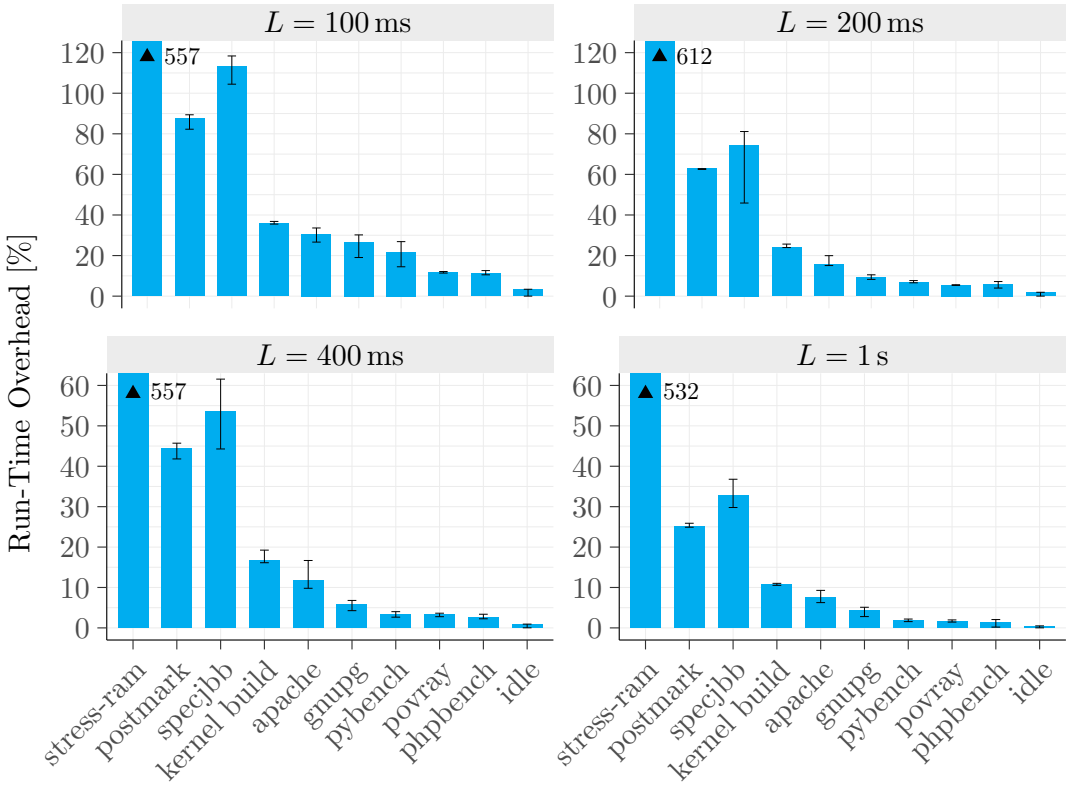


Figure 6.8: For most workloads, the run-time overhead with incremental copy-on-write checkpointing is much lower than with stop-and-copy. However, stress-ram exhibits almost three times the overhead.

the case with incremental copy-on-write checkpointing. Just like the downtime, the run-time overhead depends on the page modification rate of the workload as well as on the interval length. Using incremental copy-on-write thus considerably decreases the run-time overhead for most workloads. Whereas we observe a 2.5x overhead for the kernel build with stop-and-copy and $L = 1\,\text{s}$ (see Figure 6.3), the overhead falls down to 10% for the same configuration with incremental copy-on-write. Nevertheless, for demanding workloads such as postmark and SPECjbb, the run-time overhead remains on a high level. For stress-ram, it even increases significantly from 2.5x to 6x - 7x.

As we take the run-time overhead as a metric to estimate the probe effect, it is desirable to look into ways to further reduce it. With incremental copy-on-write, the run-time overhead can primarily originate from two sources (besides the downtime):

- The guest experiences additional page faults whenever the workload accesses a page that is queued for checkpointing but has not been copied yet (and consequently is still write protected).
- To perform incremental checkpointing in the first place, we have to track page modifications – so-called *dirty logging* (see § 6.1.2).

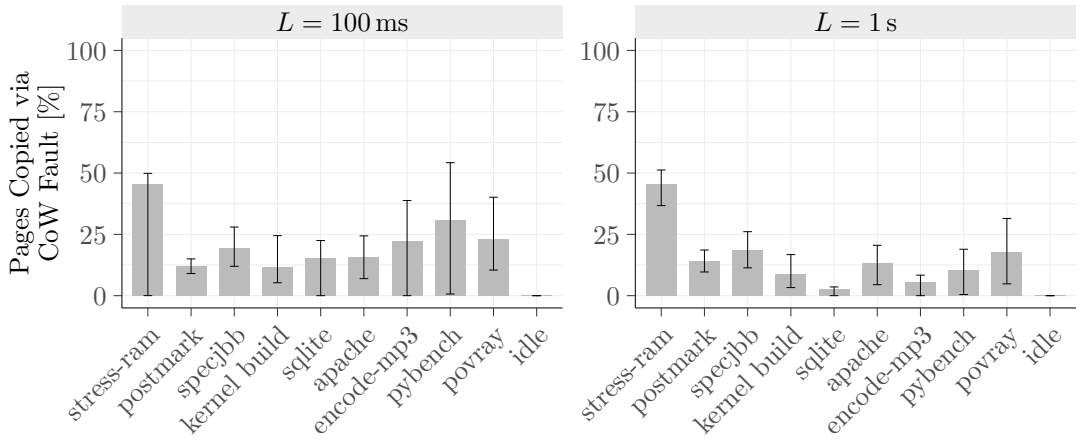


Figure 6.9: The majority of pages is saved concurrently to the guest without causing a copy-on-write page fault. For most workloads, the relative share of CoW cases slightly declines with increasing interval lengths[7].

Figure 6.9 illustrates the median share of pages that are copied in the course of copy-on-write page faults compared to the total number of pages saved. In addition to the page modification rate, the most decisive factor for a high share of CoW cases is the rate at which the workload writes to previously modified pages – i.e., the locality of write accesses. If the page modification rate or the locality is particularly high, the workload quickly accesses the same pages right after the VM has been resumed from the downtime. This is where the majority of CoW faults happen [235]. The asynchronous copy is not fast enough to prevent these CoW faults. This can be seen especially well in the case of povray. Although povray has the smallest page modification rate (besides idle) with only 550 pages/s, its high write locality results in a share of CoW faults comparable to SPECjbb, which modifies 100x more pages in the same time. The stress-ram benchmark, on the other hand, possesses a page modification rate high enough to revisit the same pages in short succession, despite having modified its whole 3 GiB buffer in the meantime.

A potential way to save run-time overhead could be to perform the asynchronous copy with multiple threads in parallel so as to reduce the chance that a write leads to a copy-on-write page fault. However, a preliminary estimation [235] of the maximum benefit that can be expected with this method revealed only a small margin of 6% improvement in run time for SPECjbb and $L = 100\,\mathrm{ms}$[8]. For larger intervals, the potential gain is negligible. Consequently, although further research into this direction is necessary to properly quantify the share of copy-

[7] For workloads with high write locality, the majority of CoW faults happen very quickly after resuming the VM. Whereas the number of CoW faults remains similar between interval lengths, the total number of pages increases (see Figure 6.4 on page 106). The relative share of pages saved with CoW thus decreases.

[8] The measurement used dirty logging via EPT scanning (see § 6.1.2) and omitted the write protection for pages to copy, thereby preventing CoW faults.

on-write faults in the total overhead, the results suggest only moderate room for improvement. This, in turn, indicates that using copy-on-write is a cost-effective way to implement asynchronous checkpointing.

In the following, we therefore concentrate on the dirty logging technique to describe methods for run-time overhead reduction.

6.1.2 Dirty Logging Techniques

For solutions based on incremental checkpointing, a fundamental task is to track which pages are modified in a certain time frame. For SimuBoost, tracking happens always in the unit of individual checkpointing intervals. Since checkpointing is a periodic process in SimuBoost, dirty logging is constantly activated. At interval boundaries – i.e., when actually taking a checkpoint – the gathered information is used to select pages for saving. Afterward, the hypervisor marks all pages clean again, resumes the VM, and tracks page modifications in the new interval.

In the following, we take a look at three different approaches that we have evaluated for SimuBoost:

Write Protection A popular method to detect write accesses to guest pages is to remove the write permissions in the EPT. Subsequent attempts to modify a guest page then trigger the hypervisor, where the page fault handler can mark the page as (to be) modified. Afterward, the write permission to the page can be restored and the VM continues execution. This is the same approach as in copy-on-write, except that the respective page is not copied in the page fault handler. This technique can therefore easily be combined with CoW by just applying write protection to all pages, instead of only the ones to be copied. The page fault handler then decides if the page should be saved. In any case, the page must be marked as dirty.

A worthwhile optimization is to set write protection only for those pages that have actually been written in the last interval. For all other pages, the write protection is still intact. Of course, this optimization is not applicable to the first checkpoint.

We used write protection for all results presented so far that involve page modification data or incremental checkpointing. Accordingly, we can conclude that write protection offers a very low downtime, but may noticeably increase run-time overhead for demanding workloads. This is because for each interval, every first write to a guest page results in a page fault. This produces a page fault rate equivalent to the page modification rate (compare Figures 6.4 and 6.10a).

Scan An alternative to write protection is to leverage the A/D-bits in the EPT of modern CPUs. Whenever the CPU modifies a guest page, it sets the access and dirty bits in the corresponding hierarchy of PTEs[9]. Instead of trapping (first) write accesses during the execution of the guest, we can scan the EPT for dirty bits. However, we have to do this synchronously in the downtime to get a consistent dirty log. Scanning therefore trades an increased downtime for a reduction in page fault-induced run-time overhead. To get accurate information for the next interval, all dirty bits must be reset before resuming the VM.

A very important optimization is to use the access bits on higher levels in the EPT to prune the scanning on memory areas that have not been touched. Consequently, only a fraction of the EPT is actually evaluated for each checkpoint (see Figure 6.10b). Just like the dirty bits, the access bits on higher levels (not on the last level) need to be reset to allow accurate pruning the next time.

Pre-Scan As an extension to scanning, we have devised an asynchronous pre-scan which performs a first traverse of the EPT shortly before the downtime while the guest is still running [235]. Similar to a synchronous scan, the pre-scan collects and resets dirty bits as well as access bits on higher levels. During and after the pre-scan, the VM continues to access and modify pages. Therefore, the final synchronous scan in the downtime is still required to complete the dirty log. However, this final scan is typically considerably faster than without pre-scan because most of the dirty bits have already been collected (see Figure 6.10b). Performing multiple rounds has proven ineffective and does not provide any additional savings [235]. Instead, pre-scan would likely benefit from dirty bits in higher levels of the EPT in order to avoid needless scans of accessed-only PTEs.

Pre-scan aims at reducing the downtime compared to pure scanning at the cost of additional asynchronous work. In contrast to write protection, the asynchronous operations do not directly affect the execution of the guest (e.g., by page faults). Nevertheless, it potentially creates a run-time overhead by taking away from the available memory bandwidth, polluting shared caches, and requiring TLB evictions.

Figure 6.11 summarizes the measured run-time overhead for all dirty logging configurations. Scan is able to noticeably reduce the overhead compared to write protection by avoiding the majority of page faults (CoW faults remain). The improvement is thus proportional to the page fault rate and particularly large for stress-ram; but also the real-world benchmarks benefit up to 30% for $L = 100\,\text{ms}$, and still up to 24% for $L = 1\,\text{s}$. As expected, scan considerably prolongs the downtime (Figure 6.12), almost doubling it for postmark. Whereas the downtime remains uncritical for all real-world benchmarks and interval lengths, it exceeds the limit of 100 ms for stress-ram in every case.

[9] On x86, only the last level possesses a dirty bit. A/D-bits are only set on TLB misses.

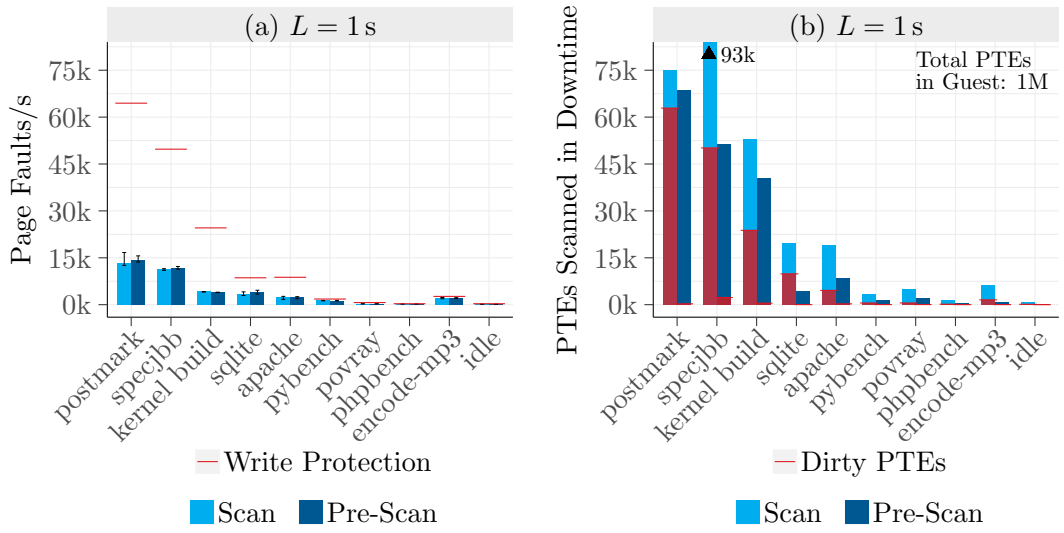


Figure 6.10: (a) Dirty logging via write protection is responsible for the majority of page faults. Scan avoids these page faults. (b) After pre-scan only a fraction of dirty PTEs has to be collected in the downtime. In total, at most 7% of the guest's PTEs in a 4 GiB VM need to be evaluated in the downtime (scan: 9%).

Fortunately, pre-scan is able to push the downtime below the critical mark, with only occasional peaks up to 112 ms. For the other benchmarks, pre-scan reaches the level of write protection with a maximum deviation of 1 ms in the most demanding scenario. At the same time, our evaluation shows on average no significant increase in run-time overhead for pre-scan compared to scan. Although in some cases (e.g., the kernel build) a slight increase in run time can be observed. Nevertheless, in most cases, pre-scan is able to combine the reduced run-time overhead of scan with the low downtime of write protection.

Despite the direct dependence between page table size and guest physical memory size, both scan and pre-scan are insensitive to different RAM configurations (see Figure 6.13a). This is due to the early pruning of untouched memory areas.

Compared to stop-and-copy, incremental checkpointing with copy-on-write and pre-scan considerably improves network throughput (see Figure 6.13b). Nevertheless, the downtimes are still clearly distinguishable – which cannot be completely avoided. However, the median (and mean) bandwidth is the same as for a run without checkpointing (*native*). This is because right after each downtime the QEMU I/O thread runs in a short burst in order to process all pending packets. This explains the increased peak performance with checkpointing enabled.

We also briefly evaluated Intel Page-Modification Logging (PML) [125] as part of a bachelor's thesis [234]. With PML the hypervisor configures a buffer of 4 KiB size, which the CPU autonomously fills with guest-physical addresses to which the VM writes. When the buffer is exhausted the CPU triggers a VM exit, which the

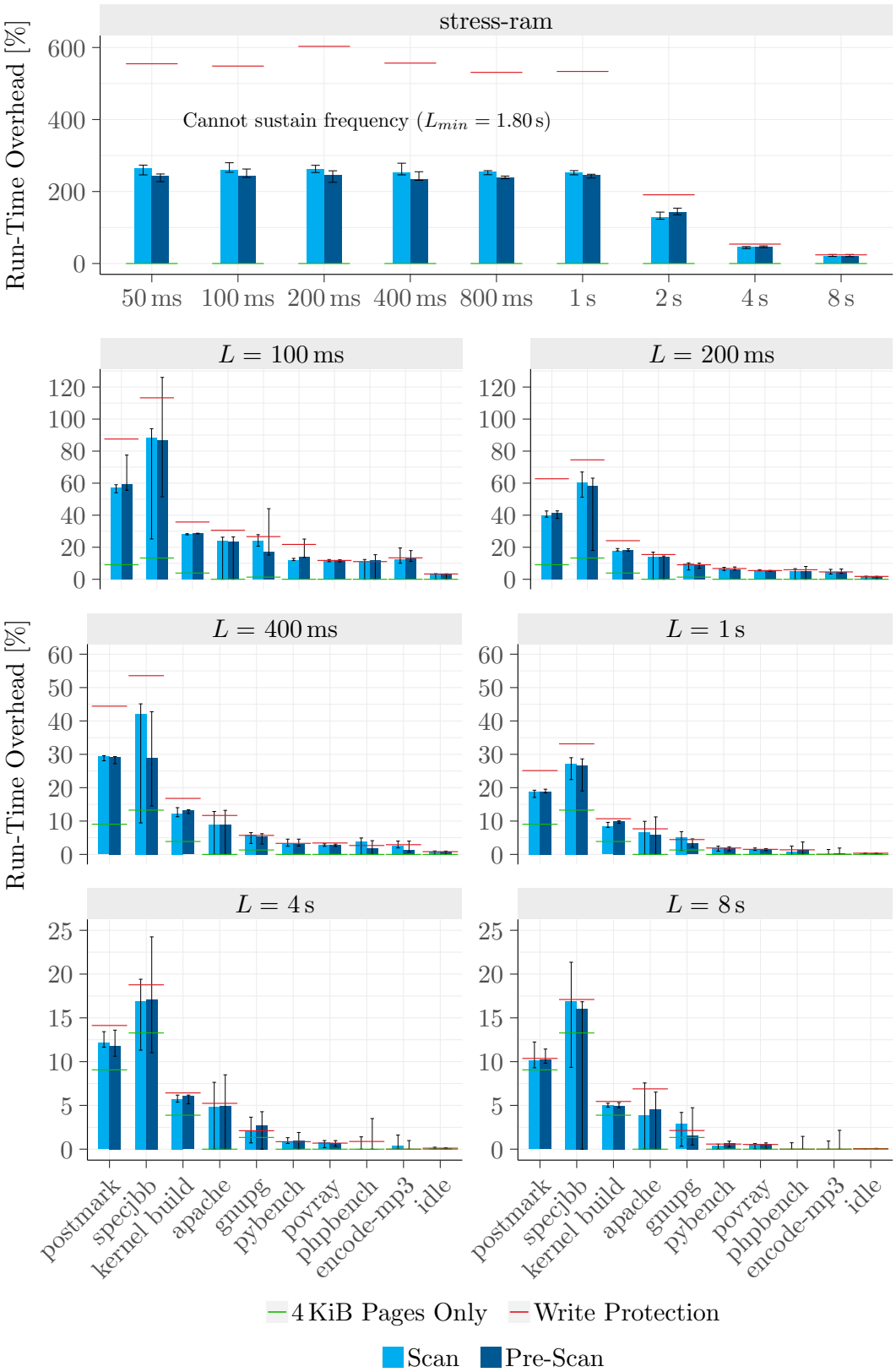


Figure 6.11: Scan reduces the runtime-overhead in every case. Pre-scan induces no significant change in overhead compared to scanning. The green line marks the overhead with no dirty logging but only 4 KiB pages to back VM memory.

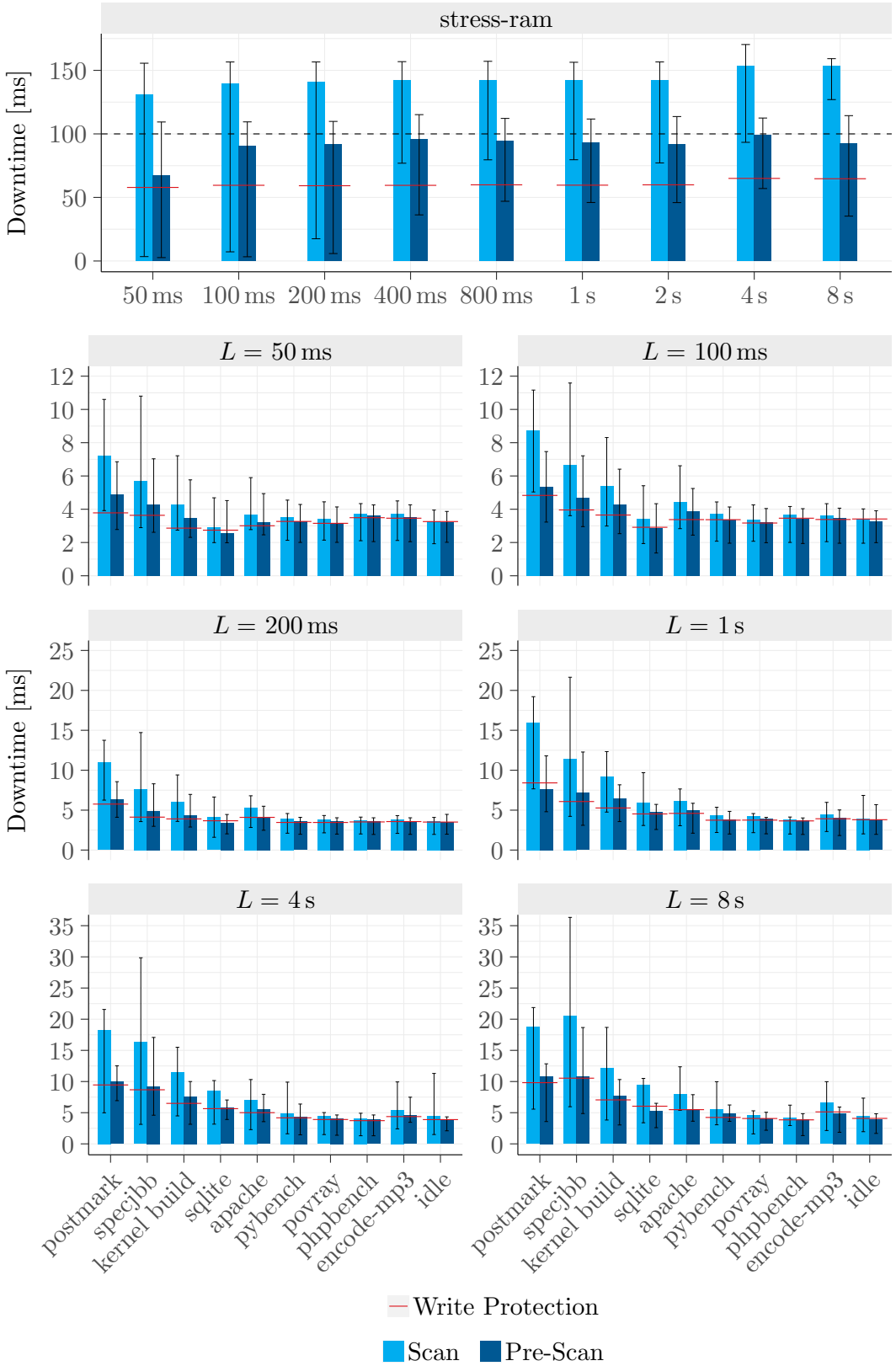


Figure 6.12: Scan noticeably increases the downtime for demanding workloads and crosses the limit of 100 ms for stress-ram. For most workloads, pre-scan pushes the downtime close to the level of write protection.

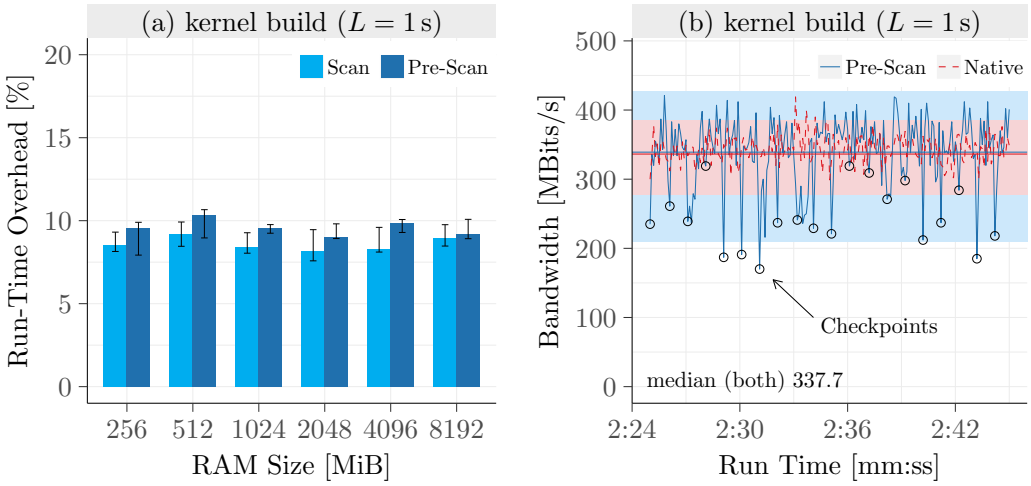


Figure 6.13: (a) Scan and pre-scan are insensitive to the guest physical memory size. (b) Downtimes with pre-scan are much less accentuated in the network bandwidth than with stop-and-copy, but still clearly visible.

hypervisor can use to extract a (partial) dirty log and clear the buffer. To avoid creating one entry for each write access, the CPU only considers writes to pages whose dirty bit is not yet set. Subsequent accesses to the same page are thus ignored until the corresponding dirty bit is reset in the downtime. PML shows an improvement in the run-time overhead of 9% points over write protection for 100 ms intervals during a kernel build. The improvement falls below 0.5% points for 1 s intervals. The downtime is at the level of write protection.

6.1.3 Dirty Logging Granularity

A cost factor in dirty logging not discussed so far is the size of the memory pages used to back the guest physical memory. With no dirty logging active, the transparent huge page support in Linux automatically merges consecutive small pages (4 KiB) to large pages (2 MiB). However, when dirty logging is active, KVM breaks all large pages into small pages to get a finer granular dirty log. Switching to small pages means more and longer traversal of the EPT and less efficient use of the TLB. In Figure 6.11, we can see that for most workloads the change in page size alone is responsible for a large part of the run-time overhead (green line). This becomes especially apparent for $L = 8\,\mathrm{s}$, where the overhead of actual checkpointing diminishes.

Not all benchmarks suffer from small pages though. In fact, for sqlite the reduction in run-time overhead even exceeds the additional overhead from checkpointing, providing a net improvement in run time between 5% ($L = 200\,\mathrm{ms}$) and 25% ($L = 8\,\mathrm{s}$). However, comparing the bare native run time with fixed 4 KiB and 2 MiB page sizes, respectively, reveals that the reduction results from deactivating transparent huge page support rather than choosing a specific page size (see

	apache	phpbench	sqlite	stress-ram
4 KiB	66%	80%	73%	89%
2 MiB	62%	80%	71%	100%

Table 6.2: Run time with fixed page size compared to transparent huge pages.

Table 6.2). This is also the case for apache and phpbench. The only exception is stress-ram, for which 4 KiB pages surprisingly seem to be better suited.

All other benchmarks show similar performance with fixed large pages and transparent huge page support but suffer from 4 KiB pages. We can thus conclude that using fixed 2 MiB pages generally gives the best performance (except for stress-ram). It would consequently be beneficial to preserve large pages even with dirty logging enabled.

A direct consequence, however, is that information on which pages have been modified is only present at the coarse granularity of large pages. As expected, stress-ram is only marginally affected by this because the workload touches a large contiguous buffer. For a kernel build, on the other hand, we have to save up to 6x the amount of memory per checkpoint (see Figure 6.14). This is even worse for most of the workloads we have evaluated (see Table 6.3), with phpbench exhibiting the highest excess of two orders of magnitude. Nevertheless, the workloads showing particularly high excess rates generally possess write working sets of only a few MiBs. In these cases, the absolute amount of memory which must be copied with large pages is therefore still tolerable.

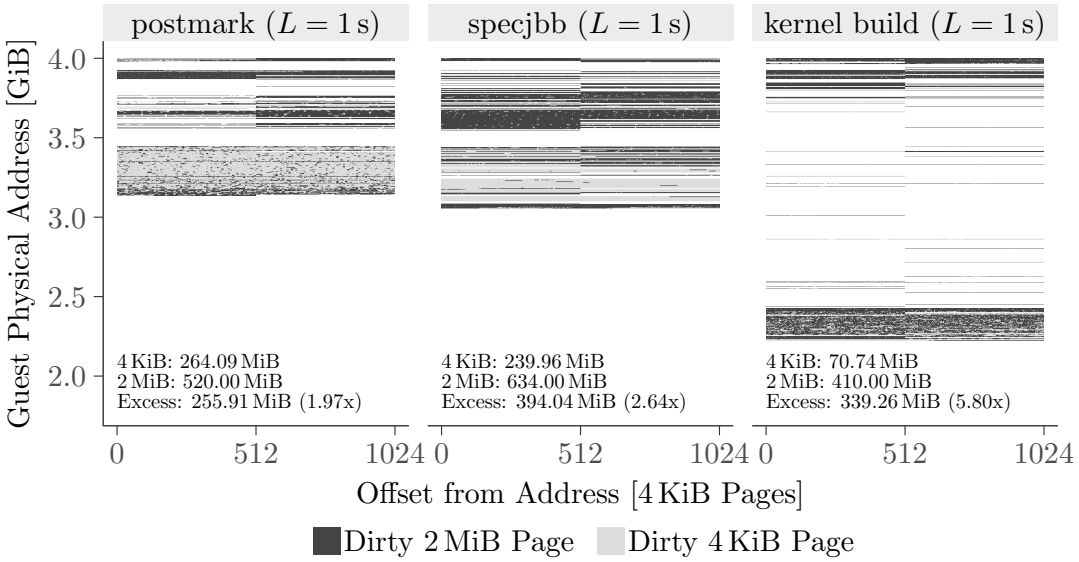


Figure 6.14: Upper 2 GiB of VM memory. Using large pages for tracking page modifications requires up to 6x more memory to be copied. The images show the checkpoints representing the median excess.

	stress-ram	sqlite	gnupg	apache	encode-mp3
4 KiB	3072	38	17	18	6
2 MiB	3346	194	184	260	118
Excess	280 (1.09x)	152 (5x)	85 (11x)	244 (14x)	113 (19x)

	pybench	phpbench	povray	idle
4 KiB	2.47	0.59	2.16	0.45
2 MiB	174	58	150	38
Excess	174 (75x)	57 (106x)	151 (67x)	38 (85x)

Table 6.3: Large pages can increase the amount of memory to be copied by two orders of magnitude. Values have been obtained for $L = 1$ s and are the respective median given in MiB[10].

For the more demanding workloads such as a kernel build or postmark, this can become a problem though. We can observe that the excess ratio increases for shorter intervals, where the working set is smaller [235]. For postmark, for example, the excess increases from 2x at $L = 1$ s to 6.6x at $L = 100$ ms. In consequence, the minimal interval length rises. We can estimate this for postmark to be around 225 ms compared to 75 ms with 4 KiB pages. This exceeds the optimal interval length of 150 ms calculated by our model. Using large pages can thus potentially impair the speedup.

Another effect of the higher data volume is an increase in run-time overhead. This works against the reduction in run-time overhead we gain from switching to large pages. For SPECjbb, we found that only at intervals of 2 s and longer, we start to benefit with a 4% higher benchmark score [235].

An alternative to using large pages for the entire guest physical memory could be to map them only in areas outside the write working set[11] or otherwise demand a high degree of dirtiness in the respective 2 MiB region so as to reduce the copy excess. Attaining detailed access information in areas backed by large pages would, however, not be possible from the dirty logging itself. A solution could be to feed information from the storage backend back into the hypervisor. As we perform data deduplication (see § 7.2) at the granularity of small pages there, we could detect which parts of a large page have been actually modified.

However, such more complex solutions are out of the scope of this work. We therefore keep using small pages (or transparent huge pages) for the remainder of this thesis and leave efficient integration of large pages as a future research topic.

[10] Note that we explicitly calculated the median for the excess – i.e., the measurements presented are not simply derived from the 4 KiB and 2 MiB values by subtraction.

[11] Excluding the write working set would require a weak overlap between the read and write working sets, which is strongly dependent on the workload [277]. Otherwise, we would not touch the large pages, which is a prerequisite for benefiting from them.

6.1.4 Design and Implementation

We implemented our checkpointing solution in QEMU/KVM (§ 2.2.5). In this process, we added a new thread which periodically creates checkpoints by repeatedly performing the following sequence of operations:

0. Initialize checkpointing

1. Suspend VM
2. Save device states (e.g., CPU registers)
3. Collect dirty bits
4. Write protect dirty pages
5. Resume VM

Synchronous (Downtime)

6. Save and unprotect dirty pages
7. Sleep until next checkpoint
8. Perform Pre-Scan (optional)

Asynchronous

9. Finalize checkpointing

To control the checkpointing thread we extended the QEMU monitor with commands to start (`start—cp`) and stop (`stop—cp`) checkpointing, where the start command accepts various arguments to parametrize the process (e.g., frequency, dirty logging technique, etc.).

In the **initialization step (0)** the checkpointing thread establishes a connection to the storage backend (§ 6.2) and requests buffer space in memory for the first checkpoint. Afterward, the selected dirty logging technique starts. This includes the allocation of dirty bitmaps (one per memory region) in user and kernel space to track page modifications[12]. The thread then enters the checkpointing loop.

The next operation is to **suspend (1)** the virtual machine and wait for the vCPU threads to be kicked out of guest mode (if necessary via inter-processor interrupt) and block in user mode. Suspending the VM also completes all outstanding I/O operations so that the virtual machine is in a consistent state. This is when the downtime begins.

To **save the device states (2)** we use the existing migration code in QEMU. It serializes the states in a form that can easily be saved, transferred, and restored. However, we also found it to be rather slow, taking on average 3 ms to collect the 120 KiB of device data per checkpoint. Considering a median downtime of less than 10 ms, this reveals optimization potential for future versions.

Before actually being able to save modified memory pages, the checkpointing thread has to **collect dirty logging information (3)** – i.e., the data on which

[12] Dirty bitmaps in user space primarily track modifications from virtual devices (e.g., DMA).

pages have been modified in the last interval. For write protection, this includes the respective dirty bitmaps in user and kernel space. For scan and pre-scan, the thread additionally scans the EPT for dirty bits[13]. In every case, gathering dirty information also includes resetting the corresponding bits so as to allow tracking page modifications in the next interval.

We use a dedicated bitmap-like data structure – the *copy map* – to merge dirty bits from the various sources and control the following asynchronous copy operation. The copy map reserves one byte for every guest memory page (i.e., 1 MiB for 4 GiB of VM RAM). This allows us to store three states per page: (a) clean (do not copy), (b) dirty, (c) currently copying. The latter state synchronizes the asynchronous copy with a concurrent CoW fault to the same page. Every element in the copy map thus also functions as a spinlock. Since CoW faults may be caused by the vCPU threads in kernel space or QEMU's I/O threads in user space, the copy map is accessible from both modes as shared memory.

Using the guest physical address space as the basis for the copy map is unpractical because just like in a real system this address space is a sparse composition of various MMIO, ROM-, and RAM-backed regions. The copy map therefore covers RAM blocks (see Figure 2.18 in § 2.2.5), which form a contiguous address space of all potentially accessible memory. This includes device memory such as VGA RAM but excludes the memory regions just mapping device registers. The content of these areas is preserved by saving the device states in the previous step.

To perform incremental checkpointing, the first checkpoint has to capture the entire system state so that subsequent checkpoints only have to store deltas. For the first checkpoint, the copy map is thus initialized to all ones, indicating modification of all pages.

After identifying which pages need to be copied for the current checkpoint, the checkpointing thread **write protects all these pages (4)** in the EPT. Since this does not prevent the user-mode QEMU process from writing to its own guest physical memory mapping, we further instrumented the respective write methods in QEMU. This allows us to vector accesses to the CoW fault handler if needed.

It is now safe to **resume the VM (5)**. This ends the downtime. All operations following this point happen asynchronously to the VM execution.

The checkpointing thread continues by iterating through the copy map to find pages that need to be copied. When **copying (6)**, the corresponding entry in the copy map is set accordingly to indicate this operation. If the concurrently executing VM attempts writing to the same page at this very time, the CoW fault spins on the copy map entry until the checkpointing thread has finished saving

[13] Note that KVM writes to the dirty bitmaps even when using scan-based dirty logging whenever a write triggers a page fault (e.g., CoW fault) or the instruction needs to be emulated.

the page[14]. The CoW fault then finds this page to be clean and permits the write attempt without any further interruption. The same applies to pages that have already been copied but whose write protection is still intact. If, however, the VM accesses a dirty page before the checkpointing thread, the CoW fault handler takes care of saving the page and marking it clean in the copy map. The checkpointing thread then simply skips the page. For fast access to the EPT, the checkpointing thread performs all copy operations in kernel space.

Our storage backend (§ 6.2) supplies the buffer space to which all checkpointed data is saved. This happens in segments of 64 MiB. When the buffer is exhausted the checkpointing thread submits the filled segment and requests an empty one. This briefly interrupts the asynchronous copy and the checkpointing thread returns to user-mode for communicating with the storage backend. After receiving new buffer space the checkpointing thread continues where it left off.

When all dirty pages have been saved and unprotected, the checkpoint is complete and the checkpointing thread **sleeps (7)** until the next checkpoint – i.e., until the current interval ends. The sleep time is computed by taking the configured interval length and subtracting the total time for all asynchronous operations[15]. In case checkpointing takes longer than permitted by the interval length, the sleep time becomes negative and we immediately take the next checkpoint.

If **pre-scan (8)** is active, the sleep time is shortened by the estimated pre-scan time. We use an exponential moving average for this purpose. The true interval length thus fluctuates by a few hundred microseconds around the set value.

The (asynchronous) pre-scan basically works the same as the synchronous EPT scan during the downtime. However, due to fact that the VM is concurrently running and setting A/D-bits in the EPT, collecting and resetting these bits need to be done more carefully in order to avoid inconsistencies [235]: To be effective in shortening the scan in the downtime we have to reset the access bits in higher EPT levels. At the same time, we must not reset the access bit without *atomically* flushing the TLB. Otherwise, writes to clean pages in the corresponding page table may not be reflected in a newly set access bit and go unnoticed. In order to mitigate this race, we first collect and reset all access bits in higher EPT levels. We then flush the TLB and use the collected information to direct the actual scan for dirty pages.

When the user enters the `stop—cp` command, the checkpointing thread **finalizes checkpointing (9)** the next time it wakes up. This disables dirty logging, frees respective data structures, and closes the connection to the storage backend. Eventually, the checkpointing thread terminates.

[14] Spinning also ends after reaching a threshold, but this causes the checkpoint to fail.

[15] We do not include the downtime because according to our performance model the interval length should determine how long the guest may execute between checkpoints. This guarantees that the guest makes progress irrespective of the length of the downtime.

6.2 Checkpoint Storage

SimuBoost uses the checkpoints to bootstrap parallel simulations. Since, depending on the simulation speed and the number of available nodes, it can take some time until a certain interval is actually simulated, we have to persist each checkpoint on disk in order to free up main memory for the next one[16]. This also allows the researcher to repeat the same recorded run with a different analysis, archive the run, or share it with colleagues.

An important factor in the design of the storage solution constitutes the fact that we employ incremental checkpointing. This is because each checkpoint depends on the data of previous checkpoints:

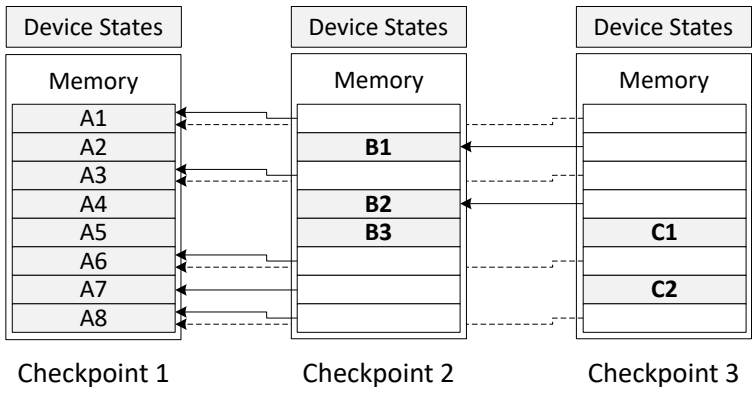


Figure 6.15: Incremental checkpoints are not self-contained, but depend on previous checkpoints, creating a dependency chain.

In order to load checkpoint 3, we have to restore A, B, and C pages. As in practice some pages (e.g., kernel text segments) live throughout the entire run time, this can create long dependency chains and make checkpoint loading cumbersome and slow. Furthermore, this complicates the distribution of individual checkpoints to simulation nodes because we also have to identify and send the dependent checkpoints.

We therefore break the dependency chain by separating the data from the checkpoints and instead just retain metadata within each checkpoint to locate all required data (see Figure 6.16). As the device states are self-contained, this is only necessary for the memory pages (and disk sectors, if included). The data is concentrated in a global key-value database, with each page or sector having its own unique identifier. The checkpoints only comprise a data structure – called *state map* – per block-based device (i.e., RAM and disks) that links every address in the device's address space to its data in the database via the corresponding identifier. For the guest memory snapshot, this is done at the granularity of 4 KiB

[16]Taking checkpoints of SPECjbb with $L = 1$ s would require around 370 GiB of main memory.

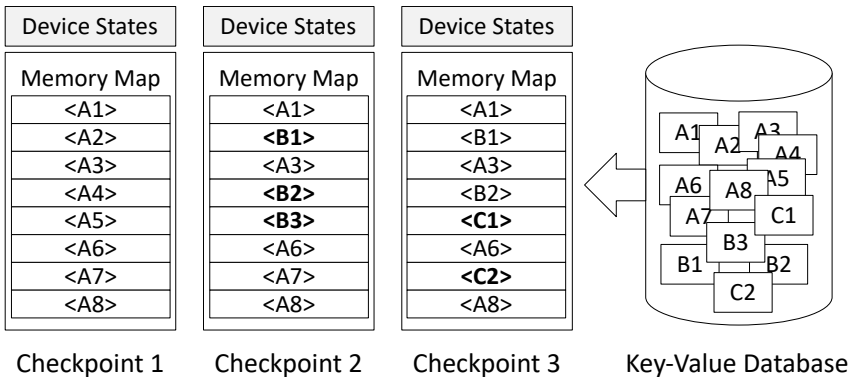


Figure 6.16: Separating the checkpoints from the data breaks the dependency chain. Pages reside in a central database and checkpoints contain references only.

pages and besides the physical memory also covers all other QEMU RAM blocks such as the video buffer. Whenever SimuBoost creates a new incremental checkpoint, it copies the states maps over to the new checkpoint and updates them according to the captured delta. This gives a complete self-contained image of referenced pages and makes accessing previous checkpoints superfluous. We store the state maps together with the device states on disk as one file per checkpoint.

To load a particular checkpoint a simulation node now conceptually only has to read the respective checkpoint file from a shared network file system, establish a network connection to the database, and retrieve the referenced data blocks. In our first prototype, we attempted to use various existing key-value stores. However, we did not find a suitable candidate. While we did not evaluate Redis [12] due to its missing support for swapping between memory and disk, LevelDB [8], MongoDB [10], as well as Kyoto Cabinet [6], proved to be too slow for our purposes [37]. Instead, we found that using an ordinary flat file as "database" gives best performance. In this design, a simple 64-bit file offset functions as a unique identifier and new data is simply appended to the end of the file. Swapping between disk and memory, caching, and read-ahead is taken over by the file system cache.

> Although a flat file hardly fulfills the requirements of a database, we keep the term to discern it from the individual checkpoint files.

6.2.1 SimuBoost Extension for Simutrace

We integrated the entire checkpoint storage logic in Simutrace [218]. Simutrace uses a client-server architecture, where the server is responsible for processing and storing the data. Designed as a fast append-only database for detailed traces (e.g., memory traces), Simutrace offers a stream-based interface. A client creates a data store and can then register streams for this store by supplying a description

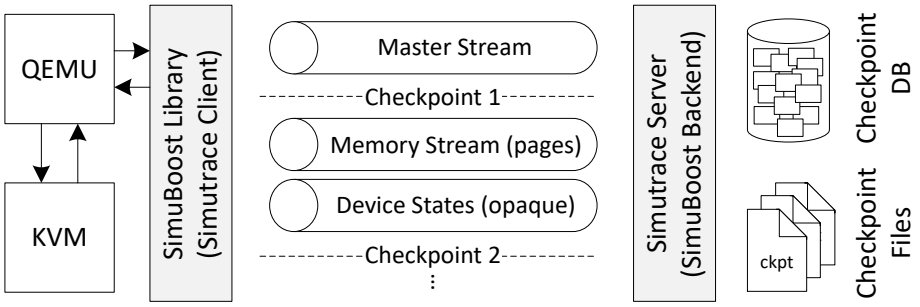


Figure 6.17: We use a specialized storage backend and a thin wrapper library to utilize Simutrace for fast asynchronous processing and storage of checkpoints.

of the data type that the stream is to receive. Although Simutrace supports variable-sized entries, elements in each stream are generally expected to be of fixed size. Accordingly, a type description comprises the size of each record together with a unique type identifier. In the server, each stream is associated with a type-specific en-/decoder using this identifier. If the designated type is unknown, Simutrace selects a default en-/decoder. When the client writes data to a stream, the corresponding encoder in the server receives it, asynchronously processes it (e.g., compression), and (optionally) persists the data on disk. Transmission to the server is done in segments of 64 MiB – for local connections using shared memory. To make Simutrace more flexible, the actual storage backend defining the en-/decoders as well as the store's on-disk format can be exchanged. The client therefore selects the desired storage backend when creating a new store.

For SimuBoost, we have extended Simutrace with a checkpointing-specific storage backend and a small wrapper library exporting a custom client interface (see Figure 6.17). This way, we can make use of the memory management, asynchronous processing capabilities (see § 7.2), and network connectivity already built into Simutrace. The library offers simple functions such as for creating or opening a checkpoint as well as writing or reading pages. Internally, our extensions create per checkpoint (1) one stream per block-based device (e.g., memory) with the appropriate data type (e.g., 4 KiB page), and (2) a stream with a variable-sized opaque type for the device states. To later identify which stream belongs to which checkpoint, the first stream in the store – the *master stream* – receives one entry per checkpoint with index data. Since Simutrace allows a client to seek to a particular element in a stream, we can quickly read the right entry using the checkpoint number as the offset.

When QEMU creates a new incremental checkpoint, it writes only the modified pages to the memory stream. It prepends a small header to each page, which contains its address so that the storage backend is able to discern what pages it received. As transmission between client and server happens in the unit of 64 MiB, the server receives around 16K pages with each segment. Due to shared memory, this does not incur any additional copy. The server then immediately

responds with a free 64 MiB segment in the existing shared memory buffer and asynchronously processes the 16K pages while QEMU can continue saving pages. When QEMU later reads the memory stream to open the respective checkpoint, the decoder does not only return the incrementally saved pages, but instead provides a complete memory image of the checkpointed VM.

6.3 Checkpoint Loading

We have extended the monitor in QEMU with a load command (`load-cp`), which accepts a number to designate the desired checkpoint. QEMU then establishes a connection to the Simutrace server[17], reads the respective element in the master stream, and eventually opens the referenced streams for the device states and block-based devices. As the SimuBoost storage backend returns complete images, QEMU only has to apply the retrieved elements (e.g., pages) at the address provided by the respective element's header. Afterward, the virtual machine can be resumed and continues execution at the checkpointed location.

In order to run a checkpoint taken in KVM with the binary translation engine (i.e., simulation), we had to make slight modifications to TCG. Otherwise, the vCPU does not correctly transition from kernel to user mode, entering an endless loop of guest kernel page fault handler invocations. Furthermore, the guest kernel must not call any KVM paravirtualization features (e.g., to indicate spinlocks) because these are not implemented in TCG and consequently freeze the simulation.

Although technically working, restoring full VM images for the interval simulations is suboptimal. This is because only a fraction of the guest's memory will actually be accessed during the execution of the intervals (see Table 6.4). Restoring all memory thus wastes precious resources:

- With a cold file system cache, loading a full image of a 4 GiB VM can take over 10 s for a Linux kernel build. Even with the entire checkpoint database in the cache, the process still requires on average 2.7 s. For comparison, simulating a 100 ms interval with a 60x slowdown takes 6 s. In this case, we spend at least 30% of the overall time on loading.
- When starting a new VM, QEMU reserves space in its virtual address space to hold the VM's various RAM blocks (e.g., guest physical memory, video buffer, etc.). When we restore a checkpoint, we write to every virtual page comprising these areas in QEMU. This, in turn, forces the OS to allocate host physical memory to back all virtual pages, irrespective if the VM actually accesses the page later[18].

[17] All our commands also take a connection string for the Simutrace server and a checkpoint store name, allowing to load a checkpoint from a remote server.

[18] Allocated but untouched pages map to a shared zero-page with CoW, not consuming host RAM.

	stress-ram	postmark	specjbb	kernel build	sqlite
Pages	788525	72441	87572	32903	23507
Memory	3080 (75%)	283 (7%)	342 (8%)	129 (3%)	92 (2%)

	apache	encode-mp3	pybench	povray	idle
Pages	14058	2480	1111	2070	385
Memory	55 (1.3%)	10 (<1%)	4 (<1%)	8 (<1%)	2 (<1%)

Table 6.4: Working set sizes for 1 s intervals. Memory is given in MiB. Percentage values relate to the full 4 GiB RAM of the test VM.

Fortunately, the parallelization of simulations hides most of the loading time spent on unneeded pages when it comes to the impact on the overall parallel simulation time [225]. Still, this unnecessarily delays progress and wastes computing time, which is better invested in simulations. The increased memory consumption, on the other hand, limits the simulation density, that is, the number of parallel simulations per physical host. A simulation for a VM with 4 GiB RAM, for example, requires a little bit over 4 GiB of host memory – excluding any memory for analysis. Cutting the memory consumption down to the true working set allows the researcher to employ a smaller cluster or even run the parallel simulation on a single workstation with acceptable performance. This reduces complexity and costs, which are important factors regarding the adoption of a new technology.

In this section, we therefore set out to confine the restored data set to what is actually needed in the interval so as to shorten the loading time and reduce the memory consumption.

6.3.1 Sparse Checkpoints

The basis for being able to perform a reduction is the use of deterministic replay in the simulations. This guarantees that the simulations access only the exact same set of pages. Since all other pages are not touched, they can be considered irrelevant for the interval and be omitted. However, this assumes that the analysis does not access these pages either. Otherwise, missing pages need to be explicitly included or loaded on demand.

A challenge with loading only the pages actually necessary for the simulation is to identify these pages in the first place. Intuitively, this is the working set of the virtual machine over the span of the respective interval. In fact, however, only the read working set is needed. This includes pages that the guest reads, or reads and writes, but excludes pages only written, as these do not affect the instruction flow. In practice, though, it is difficult to exclusively measure the read working set when running a VM with hardware-assisted virtualization. This is because [125]:

(a) EPTs do not allow permitting write access without read access.
(b) The MMU sets both the access bit and the dirty bit for write accesses.

This prevents us from discerning if a page has only been written or also read. The deduced working set must therefore include written-only pages to being measurable with current virtualization technology. Nevertheless, using simulation, we found that for most workloads, the share of exclusively written pages is below 4% of the determined working set [277]. An exception is sqlite, for which the share reaches 22%; but even in such cases, the working set is still small compared to the guest memory size (see Table 6.4). We can thus conclude that, although not delivering the optimal result, measuring the full working set instead of the read working set using method (a), i.e., setting page protections, or (b), i.e., collecting A/D-bits, is sufficient for our intended goal of significantly reducing the loading time and memory consumption.

Tracking the working set using page protections is a viable solution. However, as we already have a mechanism for scanning A/D-bits in place, we decided to use method (b). This way, we could simply extend the (pre-)scan in our incremental checkpointing to also collect access bits. These bits are then stored in a separate bitmap (1 bit per page) so that we are still able to name the modified pages for checkpointing. Just like the copy map, the access bitmap is shared between kernel space and user space, allowing us to quickly send it to Simutrace for storage. We append the access bitmap to the data stream for device states. The storage backend eventually extracts the access information and saves it in a separate file together with the checkpoint. When loading a checkpoint, we test for the existence of an access bitmap and, if available, send only the pages marked in the bitmap. We call such reduced images *sparse checkpoints* because only the relevant areas in the guest memory will be populated.

Since we have to measure the working set for a checkpoint over the span of the corresponding interval, the access bitmap is only available at the end of the interval (see Figure 6.18). However, this does not constitute a problem because the dependence on the log of non-deterministic events delays the simulations for one interval anyways.

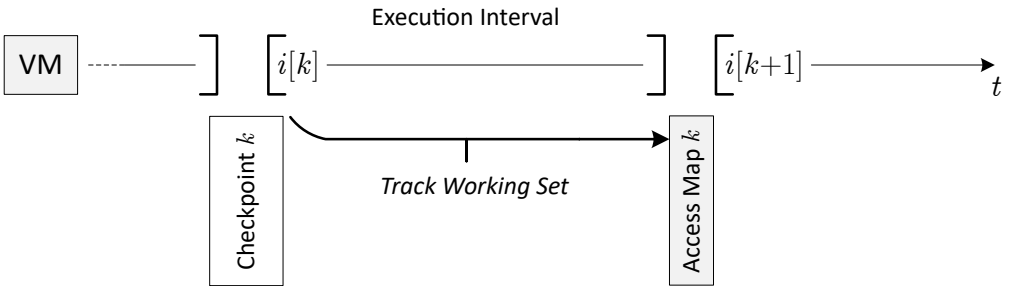


Figure 6.18: Access information is collected over the span of the interval and added to the checkpoint afterward.

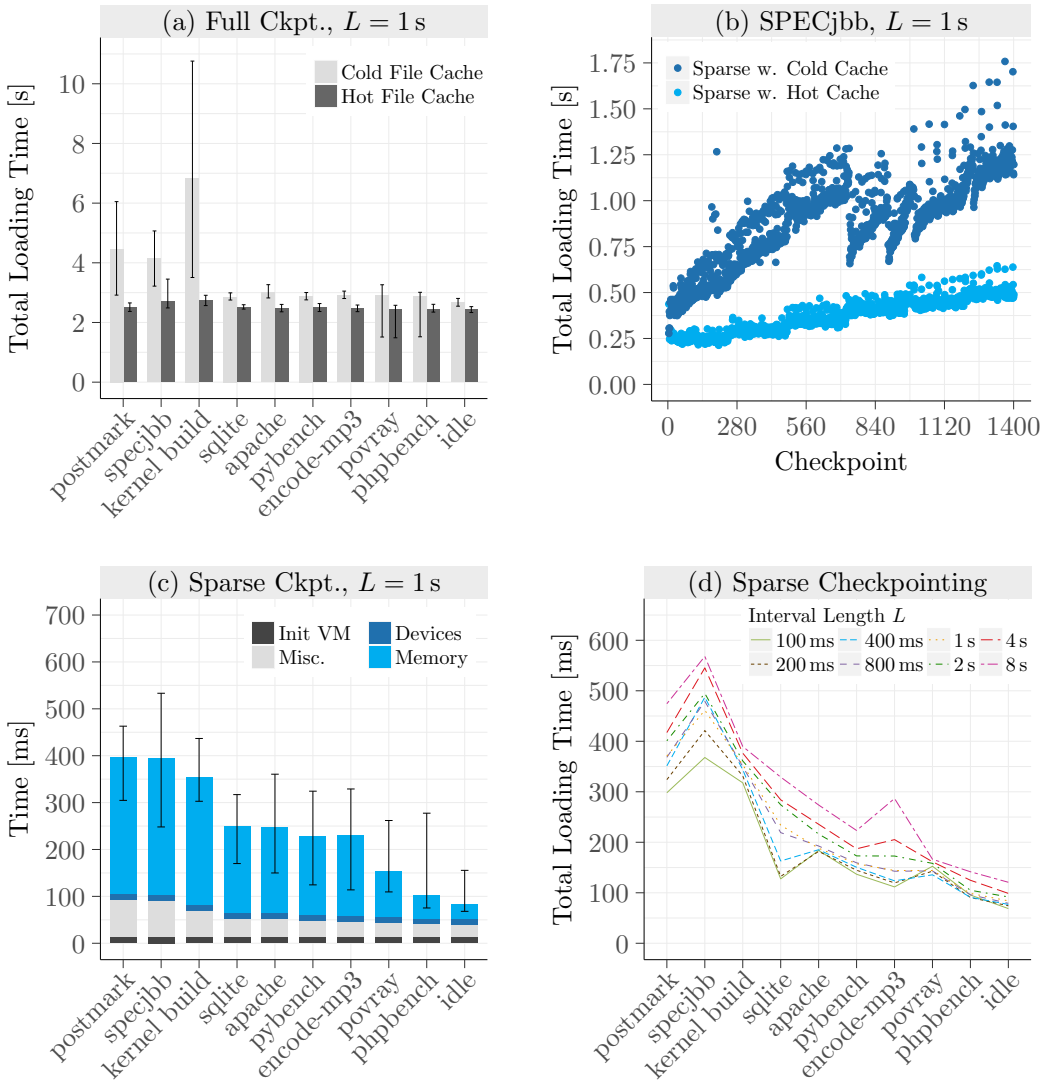


Figure 6.19: (a) Loading time of full checkpoints easily reaches multiple seconds per checkpoint. (b) Loading time for SPECjbb with sparse checkpoints ranges from 250 ms to 1.75 s, depending on program phase (clearly visible) and file cache state. (c) Loading time with sparse checkpointing is consistently shorter than for full checkpoints, with a high dependency on the workload. *Misc.* summarizes various operations such as opening and closing the checkpoint store. (d) Although loading time increases with growing interval length, dependence on workload and program phase is stronger.

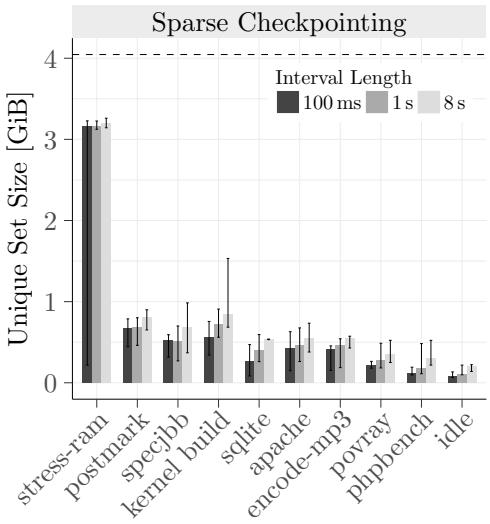


Figure 6.20: Sparse checkpoints significantly reduce the memory consumption of the simulations. The unique set size (USS) comprises all pages private to the started instance of QEMU. The dashed line denotes the memory consumption when loading full checkpoints.

In Figure 6.19a, we can see that loading full checkpoints takes at least 2 seconds, even when the file cache already holds substantial parts of the checkpoint database. Although we load the same number of pages for the three most demanding workloads, they exhibit a much larger cold cache loading time. This is because these benchmarks consume large amounts of guest memory, either by explicitly allocating it (e.g., SPECjbb) or by accessing a lot of files (e.g., kernel build), which fills the guest file cache. In both cases, checkpoints contain many pages with unique contents. As we use deduplication (see § 7.2), these checkpoints access more different (uncached) locations in the checkpoint database[19]. This also explains the large variance for the kernel build because the guest file cache holds less unique pages at the beginning of the compilation than at the end. The loading time consequently rises over the run time.

With sparse checkpoints, the loading time does no longer depend on the total number of (unique) pages in the guest RAM at the time of checkpoint creation, but on the number of pages accessed in the respective interval. This has three implications:

First, the workload phase is reflected in the loading time. In Figure 6.19b, we can clearly identify the increase in warehouses over the duration of the SPECjbb benchmark. Second, the loading time is significantly lower as much fewer pages need to be restored (see Figure 6.19c). Even for the three most demanding workloads, we save around 85% of loading time ($L = 1$ s). For idle, the saving is 95%. For stress-ram, with its 3 GiB buffer, the saving is still 30%. Looking at

[19] An exception to this is stress-ram, which fills its 3 GiB with identical pages. Its loading time is thus around 3 s.

	$L = 100\,\text{ms}$	$L = 1\,\text{s}$	$L = 8\,\text{s}$
stress-ram	-	-	48% (↗ 140%)
postmark	111% (↗ 87%)	30% (↗ 58%)	12% (↗ 21%)
kernel build	47% (↗ 68%)	13% (↗ 34%)	6% (↗ 22%)
pybench	15% (↗ 10%)	-	-

Table 6.5: The savings in loading time and memory consumption come at the cost of a higher run-time overhead (first value). The second value provides the percentage increase compared to regular checkpointing with pre-scan only. For stress-ram, we omit interval lengths for which we cannot sustain the checkpointing frequency, for pybench, we omit values below 1%.

Figure 6.19d reveals that this trend can be observed over all interval lengths, with the workload having a stronger influence on the loading time than the interval length[20]. The third implication is the reduction in memory consumption (see Figure 6.20). Except for stress-ram, every workload can be simulated with less than one-third of the memory required with full checkpoints. The less demanding workloads even fit in around 512 MiB, which is only 13% of the original demand.

A noticeable downside of sparse checkpointing is that it comes at the cost of a higher run-time overhead (see Table 6.5). This is due to the additional time required for collecting the access bits in the EPT (i.e., transfer information to bitmap, reset bits also on lowest EPT level), which involves a lot of atomic instructions. Due to pre-scan, the effect remains negligible on the downtime.

6.4 Conclusion

Naive checkpointing using stop-and-copy incurs significant costs in terms of downtime and run-time overhead and is thus unsuitable for SimuBoost. We have shown that combining incremental checkpointing with copy-on-write and efficient dirty logging achieves superior performance and is fast enough to be leveraged in SimuBoost.

We found that scanning the EPT for dirty bits is faster than setting write protections, although scanning noticeably increases the downtime. We presented pre-scan, a novel approach to dirty logging, which moves most of the scanning time out of the downtime by performing a first scan asynchronously to the execution of the workload. This way, pre-scan successfully combines the low downtime of write-protection-based dirty logging with the reduced run-time overhead of

[20] The spread between the shortest and the longest interval lengths mirrors the access locality of the workload. If the access locality is low, increasing the interval length has a strong effect on the working set size (e.g., sqlite, compare Figure 6.9 on page 113).

EPT scanning. With less than 10 ms for all real-world benchmarks and $L \leq 8$ s, the downtime is significantly below the limit of 100 ms. This enables us to provide perceptually fluent interaction even during checkpointing. Although the downtimes are still visible in network connections (which cannot be completely avoided), the mean bandwidth is not negatively affected ($L = 1$ s). The run-time overhead introduced by checkpointing, however, can become comparably large. We measured up to 90% overhead for SPECjbb, 60% for postmark, and 30% for a kernel build with $L = 100$ ms. With $L = 1$ s the run-time overhead drops to one third (i.e., 10% for a kernel build).

Finally, we described sparse checkpoints as a possible way to reduce the loading time of checkpoints from seconds to a couple of hundred milliseconds as well as the memory consumption of simulations from gigabytes to less than 512 MiB. This way, SimuBoost can be run more efficiently on smaller simulation clusters or even on a single workstation only. However, our evaluation has revealed that this comes at the cost of increased run-time overhead. Employing sparse checkpointing thus remains a tradeoff between hardware requirements and probe effect.

Chapter 7

Checkpoint Distribution

When a checkpoint has been created, the next step in SimuBoost is to schedule the simulation of the respective interval based on the available pool of simulation nodes. This can be done in two ways: fixed or dynamic.

With *fixed scheduling*, each node is a priori assigned a defined set of intervals (i.e., checkpoints). The assumption behind this policy is that we can benefit from this knowledge in such a way that the total simulation time is reduced. Since we cannot accelerate the execution of the simulations itself with this method, vectors for optimization emerge primarily in the initialization phase. The initialization comprises (a) receiving a checkpoint, (b) starting the simulator, and (c) loading the checkpoint. In addition, code caches used in the simulation require some time to heat up, which can be considered part of the initialization phase. A scheduling policy providing benefits must therefore save time in at least one of these areas.

If we assign each node a consecutive set of intervals, we get a seemingly optimal case. When we reach the end of an interval, we can simply proceed simulation without even loading a checkpoint. Due to the deterministic replay, the end state of the current interval is equal to the initial state of the next interval. Furthermore, we do not have to start a new instance of the simulator and the code caches are already hot. In fact, however, scheduling a consecutive set of intervals is equivalent to choosing a longer interval length (which is only a matter of configuration). As we can expect to start from an optimal interval length, this will hurt the speedup which in turn results in higher overall simulation time. We can thus conclude that a fixed scheduling policy should not assign consecutive intervals to the same node.

A viable alternative is to assign each node intervals according to a round-robin policy, that is, node N_k simulates the intervals $i[k + jN]$ with $j \in [0...\frac{n}{N})$, n being the number of intervals, and N being the number of simulation nodes. This complies with our performance model and minimizes the idle phase at the beginning of the parallel simulation. As each node only simulates every Nth

interval, we cannot avoid loading the corresponding checkpoints and the code caches are likely less hot. In the case that checkpoints must be queued because the simulation cluster is too small (i.e., $N < N_{opt}$), we can benefit from the fixed assignment by transferring checkpoints to the right nodes before they are actually needed. In the optimal setup (i.e., $N \geq N_{opt}$), on the other hand, the target node always completes its last interval when a new interval becomes ready. In consequence, we cannot save time by transferring checkpoints ahead of time.

With *dynamic scheduling*, we do not a priori assign intervals to nodes, but instead dynamically select the next free node whenever an interval can be simulated. If we apply the assumption from the performance model that every simulation takes the same amount of time, this is, in fact, equivalent to the fixed round-robin policy. If, however, the simulation time varies, for example, because of different phases in the workload (idle = fast, FPU = slow) or due to a heterogeneous simulation cluster, dynamic scheduling is more flexible. It does not have to wait for a certain node to become free, but can select another one and thereby keep all nodes busy. As a downside, transferring checkpoints beforehand is not easily possible.

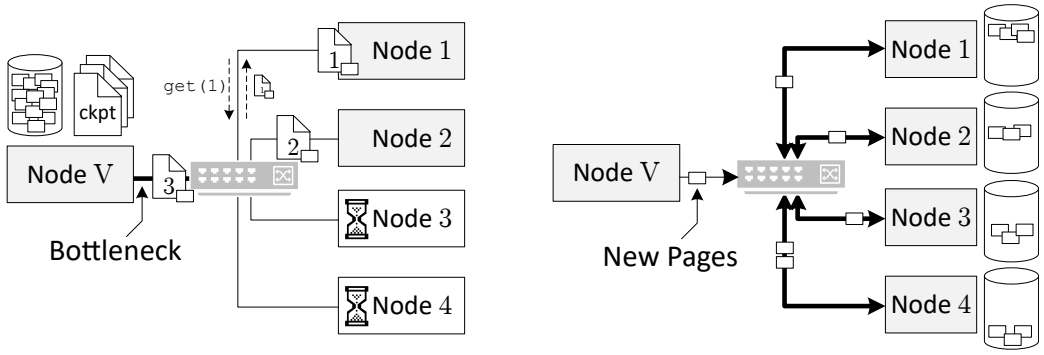
Dynamic scheduling can be further divided into undirected and directed scheduling. Whereas *undirected scheduling* picks a free node randomly, *directed scheduling* attempts to select a free node for which the costs to simulate a particular interval are smaller than for the other free nodes. The scheduler could, for example, incorporate knowledge about redundant data across checkpoints and choose a node which has already received most of the data from previous checkpoints.

Due to the increased flexibility, we have decided to use dynamic scheduling in SimuBoost, but leave it undirected to keep the design simple. In addition, the distribution design as described in this chapter decouples the selection of a simulation node from the actual checkpoint transfer. This is likely to further reduce any potential benefits that can be gained from directed scheduling.

Since there are sophisticated solutions for dynamic (undirected) job scheduling already available (e.g., SLURM [290]), we do not dive any further into scheduling, but concentrate on how to efficiently get the checkpoint data to the corresponding nodes. In Section 7.1, we contrast two fundamental approaches to transfer checkpoint data. In the following Section 7.2, we present how different compression techniques can be leveraged to enable fast transfer also with commodity network infrastructure such as Gigabit Ethernet. Building on this capability, we describe the actual distribution concept for SimuBoost in Section 7.3.

> Just like in Chapter 6, we ignore disk checkpointing in the following as it is not needed in SimuBoost with deterministic replay in place. Nevertheless, the discussed techniques are also applicable to disk sectors. We further omit the replay logs as they are generally small in comparison. See Chapter 8 for details on the log growth rate.

7.1 Pulling versus Pushing



(a) Pulling (b) Pushing

Figure 7.1: (a) The simplest method is to pull checkpoints from the virtualization host when needed. In this case, the link on the virtualization host becomes a bottleneck. (b) An alternative is to let the virtualization host push new data (here exemplary memory pages) into the network where they are distributed among the simulation nodes.

When a node has been selected to simulate a certain interval, it has to somehow receive the corresponding checkpoint. An intuitive approach is to connect to the virtualization host (i.e., the system creating the checkpoints) and load the checkpoint over the network. Our storage backend supports TCP remote connections as part of Simutrace (see § 6.2.1). That means we can just open the checkpoint store remotely and read the respective streams over the network. We thereby *pull* the (full) checkpoint from the virtualization host to the simulation node.

A major weak point of this solution is, however, its limited scalability. The larger the simulation cluster, the more checkpoints need to be sent over the single link of the virtualization host. This quickly exhausts a Gigabit Ethernet adapter and as a result, loading a checkpoint can take up to multiple minutes in a commodity network [209]. This is even the case for small setups and amplified by the fact that checkpoint loading may occur in bursts if the simulation time is mostly uniform (see Figure 7.2a). Although sparse checkpoints or extensive compression[1] can to some degree free up network bandwidth, the link to the virtualization host remains a conceptual bottleneck.

For checkpoint creation, we take advantage of the fact that only a fraction of the guest physical memory is actually modified between checkpoints. This drastically reduces the amount of data to be saved. We can pass this saving on to the checkpoint distribution by constantly *pushing* only the new data into the network, instead of sending entire checkpoints by request (see Figure 7.1b). This way the load on the link to the virtualization host only depends on the workload's page

[1] The compressibility of complete checkpoints declines with the run time as former zero pages start to contain less compressible data. Loading thus slows down over time [209].

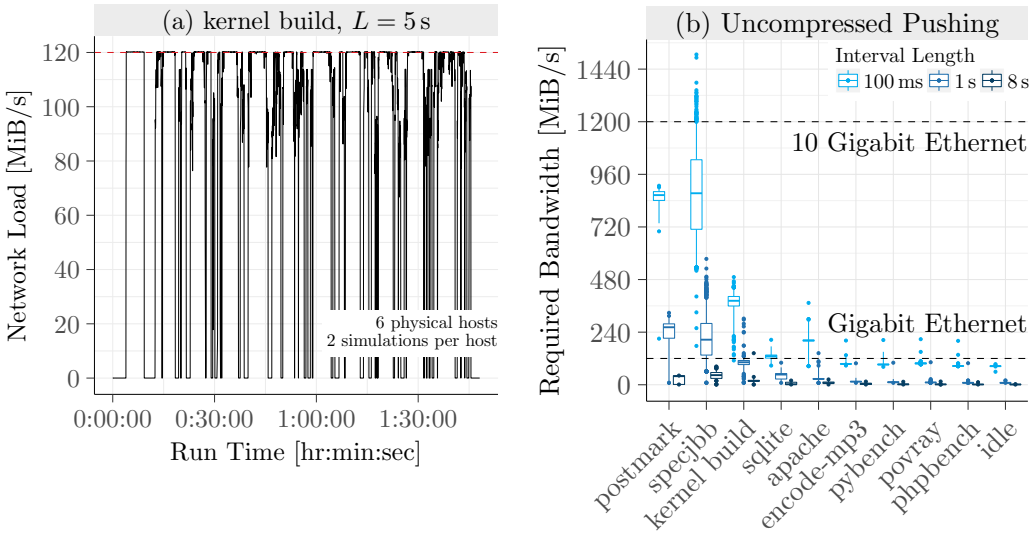


Figure 7.2: (a) Network load with pulling at the virtualization host. Even a small setup quickly exhausts Gigabit Ethernet. In addition, with uniform simulation times, the load follows a sequence of bursts (data from [209]). (b) Gigabit Ethernet neither provides enough bandwidth to allow pushing. In fact, demanding workloads can even exhaust 10 Gigabit Ethernet. The box illustrates the first and third quartiles with the median in the middle. The whiskers represent the 2.5 and 97.5 percentiles, respectively. Outliers are plotted as dots[3].

modification rate and the interval length, but not on the size of the simulation cluster[2]. Although this does not fully remove the bottleneck, it increases scalability. This is because we can now distribute the data among the simulation nodes so that the load is evenly spread. This can, for example, be achieved with a distributed network file system. When a simulation node then loads a checkpoint, the file system transparently accesses the storage on other simulation nodes, if necessary.

A fundamental prerequisite for this approach is that the amount of data created per interval does not exceed the bandwidth of the link that connects the virtualization host with the simulation cluster. For Gigabit Ethernet, that means if SimuBoost creates one checkpoint per second, each checkpoint must not be larger than 120 MiB. As we can see in Figure 7.2b, this is not the case, especially for demanding workloads such as postmark or SPECjbb. Even the kernel build, being a moderate benchmark, occasionally creates exceedingly large checkpoints with $L = 1\,\mathrm{s}$. For 100 ms intervals, half the benchmarks overload Gigabit Ethernet, with SPECjbb peaking at over 12 gigabit/s.

[2]Having more simulation nodes generally means using a shorter interval length – i.e., more checkpoints – in order to increase parallelism. However, the number of modified pages decreases exponentially with shorter intervals (see Figure 6.4 on page 106).

[3]The repetitive pattern of two high outliers for the more lightweight workloads originates from loading the test framework which starts the benchmarks.

As we intend to run SimuBoost on commodity network infrastructure, we therefore must first reduce the size of the incremental checkpoint data. Only then we can proceed with distributing it in the simulation cluster using pushing.

7.2 Checkpoint Data Reduction

Every checkpoint consists of (a) opaque device states (e.g., CPU registers), (b) the state map describing the relationship between guest physical memory addresses and data locations in the checkpoint database, and (c) the incremental data (i.e., pages) itself, which SimuBoost persists in the database.

This primarily gives us two vectors for reducing the data sent over the network:

1. We can deduplicate identical pages (e.g., zero or copied) so as to avoid sending redundant data into the simulation network.
2. For all non-redundant data, we can apply specialized and/or generic lossless compression techniques.

We have implemented a corresponding data reduction pipeline in our checkpoint storage backend for Simutrace (see Figure 7.3):

In **step 1**, the incoming memory pages are distributed among a set of worker threads which parallelize the upcoming processing. However, we do not create jobs with less than 500 pages to limit the overhead for small checkpoints.

Each thread then starts with the data deduplication stage (see § 7.2.1) in **step 2**. In this phase, the storage backend filters pages that are already present in the checkpoint database. In addition, redundant pages within the newly arriving ones are deduplicated (this works across threads). Whenever the backend recognizes

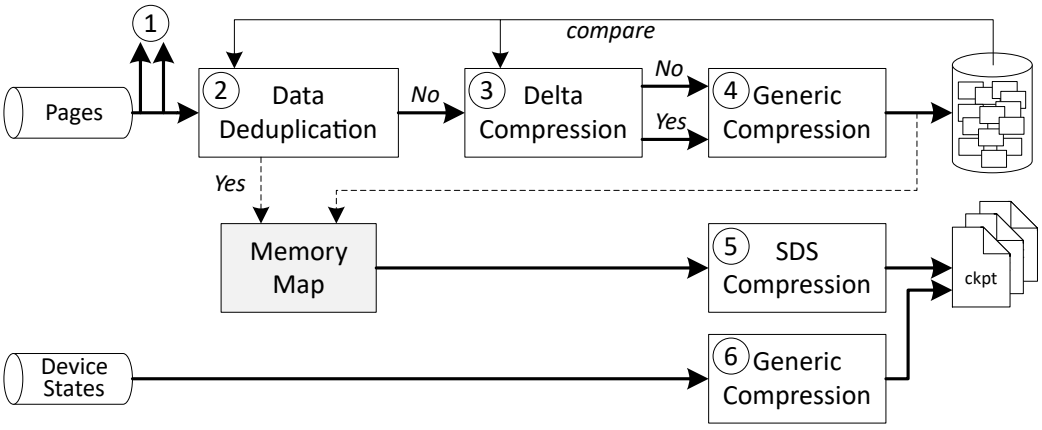


Figure 7.3: Overview of the data reduction pipeline in SimuBoost.

a known page, the offset of the existing entry in the database is written to the memory state map and the new (identical) page is dropped.

The pipeline forwards all non-redundant data to the delta compression (see § 7.2.2) in **step 3**. As described in § 2.3.4, memory pages typically experience only minor changes within each checkpointing interval. In these cases, the compression ratio can notably be increased by compressing the delta instead of the original data. However, this effect generally reverses when the changes are too broad. We thus use a simple heuristic based on the Hamming distance to decide if a page should be replaced with its delta.

In any case, the pages are passed on to **step 4**, the generic compression. Afterward, the storage backend appends the compressed pages to the checkpoint database and updates the offsets in the memory state map accordingly. We employ LZ4 [68] as a generic compressor. LZ4 is a popular derivative of the LZ77 algorithm with a focus on fast lossless operation. Although the amount of data accumulated over a checkpointing session can quickly reach multiple hundred gigabytes, we favor speed over compression ratio in this phase to compensate for the overhead we introduce with the previous stages. Nevertheless, we found this combination to achieve better compression ratios than simply using a more heavyweight generic compression such as LZMA [9]. This is conclusive as the stages 1 and 2 reduce redundancy in both the spatial and the temporal dimensions, and also respect the special structure of memory pages, whereas a generic compression only works on the current input data without incorporating any further knowledge. Furthermore, the incremental nature of the checkpoints requires that pages can be decompressed individually. Compression must thus be applied separately to each page. This deprives generic compressors of the ability to deduplicate identical pages.

When all pages have been processed, the storage backend can write the state map into the corresponding checkpoint file. Since the map for a VM with 4 GiB of guest physical memory is around 8 MiB in size, the state map is compressed in **step 5** beforehand. We use a custom encoder, called *SimuBoost Device State Compressor (SDS)*, as we found it to provide superior performance compared to LZ4 for this particular data structure (see § 7.2.3).

In **step 6**, the pipeline sends the opaque device states ($\triangleq$ 120 KiB) through the LZ4 compression and finally appends the data to the checkpoint file. The device states usually compress down to less than 10 KiB, irrespective of the interval length and workload. This completes the processing and the next checkpoint can be created.

Compression Ratio

To get a better impression of the individual workload characteristics, we omit the first checkpoint in the evaluation, which captures the base image of the VM before the workload starts. Since we perform a fresh boot each time, the image

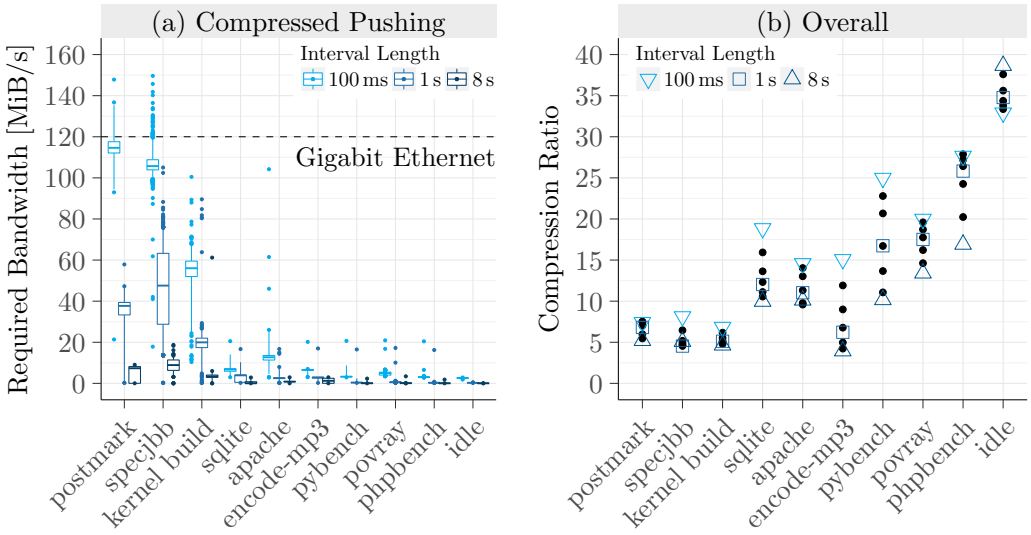


Figure 7.4: (a) With compression, Gigabit Ethernet provides enough bandwidth for most checkpoints (compare Figure 7.2b). (b) Compression ratio over all checkpoints (except the first) is between 4:1 and 39:1, with lightweight workloads and short intervals yielding better results. The black points denote results for $L \in \{200, 400, 800, 2000, 4000\}$ ms.

is identical for all configurations. For our 4 GiB test VM, the compression ratio is 17.3:1, resulting in a final checkpoint size of around 250 MiB. In practice, we therefore can expect a one-time delay of around 2 s, depending on the interval length, before the actual parallel simulation begins.

As illustrated in Figure 7.4a, the presented pipeline sufficiently reduces the required network bandwidth to generally allow pushing new checkpoint data into a simulation network based on Gigabit Ethernet[4]. With compression enabled, only for postmark and SPECjbb with $L = 100$ ms, 13 s of 60 s (22%) and 38 s of 1741 s (2%) of workload run time exceed the limit of 120 MiB/s by at most 30 MiB (i.e. 250 ms transfer time). For postmark, these are primarily the first seconds of run time, whereas for SPECjbb over 60% of the overload situations occur in the last third of the benchmark where the page modification rate is the highest. We can thus expect slight delays in the parallel simulation for benchmarks with comparable load and a short optimal interval length. Nonetheless, according to our performance model, both benchmarks possess a higher optimal interval length than 100 ms (postmark: 150 ms, SPECjbb: 2 s). This means that at least for the given workloads, this should not be a problem in practice.

The compression ratio (i.e., $\frac{uncompressed}{compressed}$) inherently depends on the kind of data generated by the workloads. However, we can observe two general trends (see Figure 7.4b): (1) lightweight workloads in terms of the page modification rate are

[4]We excluded stress-ram because its synthetic nature (all identical pages) leads to compression ratios beyond 11k:1. This reduces the original data volume of over 3 GiB/s to less than 3 MiB/s.

	postmark	specjbb	kernel build	sqlite
$L = 100\,\text{ms}$	50 ↘ 5.5	881 ↘ 134	201 ↘ 29	2.64 ↘ 0.14
$L = 1\,\text{s}$	11 ↘ 1.5	381 ↘ 82	49 ↘ 9.4	0.87 ↘ 0.07
$L = 8\,\text{s}$	1.6 ↘ 0.31	82 ↘ 16	9.5 ↘ 2.0	0.32 ↘ 0.03

	apache	encode-mp3	povray	idle
$L = 100\,\text{ms}$	27 ↘ 1.8	4.2 ↘ 0.28	63 ↘ 3.2	4.9 ↘ 0.15
$L = 1\,\text{s}$	3.8 ↘ 0.34	0.69 ↘ 0.11	6.6 ↘ 0.38	0.59 ↘ 0.02
$L = 8\,\text{s}$	1.3 ↘ 0.13	0.33 ↘ 0.08	1.2 ↘ 0.09	0.13 ↘ 0.003

Table 7.1: Total data per workload before and after compression in GiB.

likely to exhibit a higher overall compression ratio, and (2) the interval length can be used as an indicator for compressibility, where data from short intervals usually compresses better. The latter is probably rooted in the fact that long intervals can accumulate more changes and thereby increase entropy.

In addition to immediate simulation, checkpoints can be stored on the virtualization host for later use[5]. This is also required for small setups, where no additional network nodes are involved, and helpful for repeated simulations because it allows to free up the cluster in the meantime. Other reasons for permanent storage may include archiving regulations or the desire to share a recording with other researchers. The total required disk space per session is thus an equally important metric. This is also the case for pushing if the data is not distributed among the simulation nodes, but stored in full on every node (see § 7.3). Table 7.1 provides a size comparison of raw and compressed data. Our pipeline significantly reduces the required disk space, even for the more lightweight workloads.

The additional data volume for the access bitmap in sparse checkpoints is negligible because the compression ratio is generally above 50:1 using LZ4, with the bitmap encompassing 1 MiB for a 4 GiB VM – i.e., 1 bit per physical page – and shrinking down to less than 20 KiB. In the sum of all checkpoints, this is only 0.1% of total compressed data for SPECjbb and 0.3% for a kernel build ($L = 100\,\text{ms}$).

When looking at the effectiveness of the individual compression methods in Figure 7.5, we can see that it strongly varies between the different workloads. Whereas postmark benefits in particular from data deduplication, all other methods are only marginally able to reduce the checkpoint size, accounting for less than 10% of the total compression. This emphasizes the importance of a good mix of schemes to achieve sufficient overall reduction. Nevertheless, data deduplication presents itself as a powerful compression technique and for most workloads delivers an additional data reduction of more than 48% over generic compression (i.e., a cut in half).

[5]The current prototype requires (additional) local storage to allow for data deduplication and delta compression as no cache for the referenced data exists.

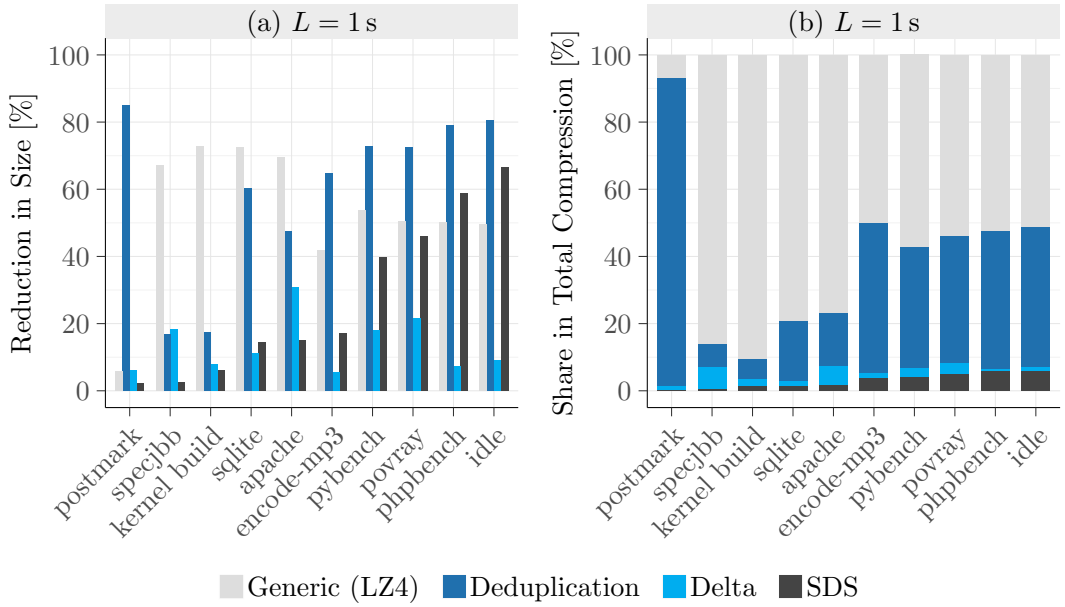


Figure 7.5: (a) The plot shows the additional reduction in size (over generic compression) for each of the methods if all methods to the left are also applied. Generic compression is relative to the uncompressed image. For instance, in the case of idle, SDS can save an additional 67% of total data volume compared to generic compression, if all other methods are in effect. (b) The plot shows each method's share in the overall compression.

Since SDS is specific to the state maps, its share increases with the number of stored maps, i.e., checkpoints, which in turn depends on the workload run time and the interval length (compare Figure B.1). On average we can see the trend that shorter intervals benefit the specialized compression schemes, whereas longer intervals improve the effectiveness of the generic compression.

> Note that we had to redirect the final (compressed) output to `/dev/null` for the benchmarks only encompassing generic compression because the SSD (380 MB/s) in the test system was unable to write out the data fast enough (especially postmark, 758 MB/s); at peaks even for $L = 1\,\mathrm{s}$.

CPU Usage

Introducing the various checkpointing and data reduction steps does come at the cost of CPU time. Figure 7.6 depicts the number of additional CPU cores required to power checkpointing with compression enabled. On average up to three additional cores running in parallel to the one executing the guest VM are needed for $L = 100\,\mathrm{ms}$. This includes overhead for (1) asynchronously checkpointing the VM, (2) compressing the captured data, and (3) writing it onto persistent

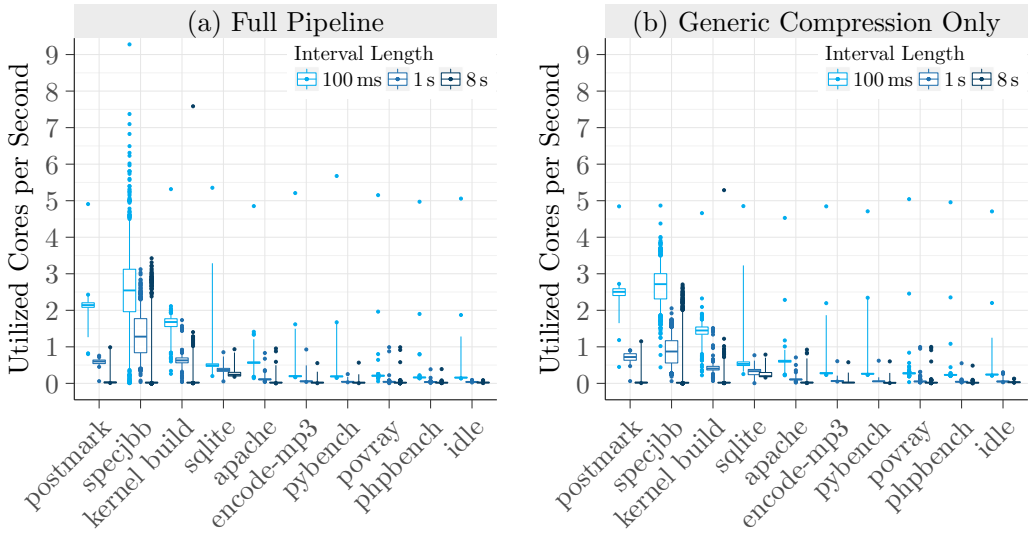


Figure 7.6: Powering the presented data reduction pipeline does not consume considerable more CPU time than applying fast LZ4 compression only, despite the significant increase in compression ratio[6]. The band of peaks around 5 cores represents the checkpoints where the benchmark framework in the VM starts.

storage. The overhead falls notably with longer interval lengths (although not linearly). For 1 s intervals, checkpointing can mostly be handled on one additional core, except for SPECjbb. Comparing the overhead of the presented pipeline with employing solely generic compression reveals that the pipeline incurs comparable costs, despite the significant increase in compression ratio.

7.2.1 Data Deduplication

The deduplication stage removes duplicates (1) between new pages and the ones already stored in the database as well as (2) within the stream of incoming new pages itself. The latter is important in order to recognize redundant content as soon as it first appears and not only in subsequent checkpoints; for example, zero pages, which usually make up the majority of the first checkpoint.

To detect identical pages we compute a hash for all incoming pages. When we were looking for a suitable hash function, we wanted to balance uniform distribution to avoid collisions (i.e., false detection of duplicates) and fast computation. We did not consider cryptographic hash functions such as the SHA family because they generally incur high overhead [187]. Instead, we chose the 64-bit

[6] In fact, (b) does not include the overhead of writing the compressed data onto the SSD due to the exceedingly large data rate for postmark and SPECjbb. However, the prototype does not allow disabling actual storage in (a) because data deduplication and delta compression are involved and pages need to be available for reference. Including the overhead where possible (not shown) in (b) reveals that our pipeline even requires *less* CPU power.

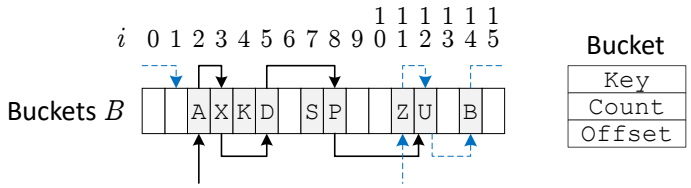


Figure 7.7: For each input key, the fixed-size hash table probes a chain of buckets (here incremental probing) until (a) the key is found, (b) an empty bucket is found, or (c) the maximum number of probes is reached, in which case LFU replacement is used. The buckets map each key to an offset in the page database.

version of Google's non-cryptographic FarmHash [5] as it delivers good distribution characteristics (we did not observe a single collision yet) and very high throughput [208].

Hashing and thereby also the detection of redundant pages is done at the granularity of 4 KiB pages. Whereas previous work has shown slight improvements in deduplication for sub-page (e.g., 1 KiB) granularity (see § 2.3.3), we decided against it to keep the design simple and the number of hash lookups low.

After computing the page hashes, the backend performs a lookup for each hash in a table of previously seen hashes. In an early prototype, we utilized an unbounded hash table (i.e., C++ `std::unordered_map`) for this purpose, using the page hash as the key. However, we quickly found it to cause frequent delays of multiple seconds during checkpointing. The delays resulted from the data structure performing a key re-hashing when growing to accommodate the increasing number of entries. We therefore decided to use a **fixed-size hash table** which caches only a limited set of past hashes (see Figure 7.7):

Let $B[i]$ be a bucket in the fixed-size hash table and $i \in [0, b)$ with b being the number of buckets. We then perform a lookup for the key k as follows with $j \in [0, l)$ and l denoting the length of the probing chain for conflict resolution:

$$i_j := \begin{cases} h(k) \bmod b & \text{if } j = 0 \\ (i_{j-1} + p(j)) \bmod b & \text{if } j > 0 \end{cases}$$

The function h is the key hash function. Since we use the page hash as the key, there is no need to apply any further hashing and we select the identity function (i.e., $h(k) = k$). If the bucket $B[i_0]$ is not free and does not contain k, we continue probing according to the rule implemented by p. We tested the following variants:

$$p(j) := \begin{cases} 1 & \text{(\textbf{L})inear Probing} \\ j & \text{(\textbf{I})ncremental Probing} \\ j^2 & \text{(\textbf{Q})uadratic Probing} \end{cases}$$

If we reach an empty cell, we insert the hash at this position and consider the page to contain previously unseen contents. If we have a hit (i.e., a key match), we

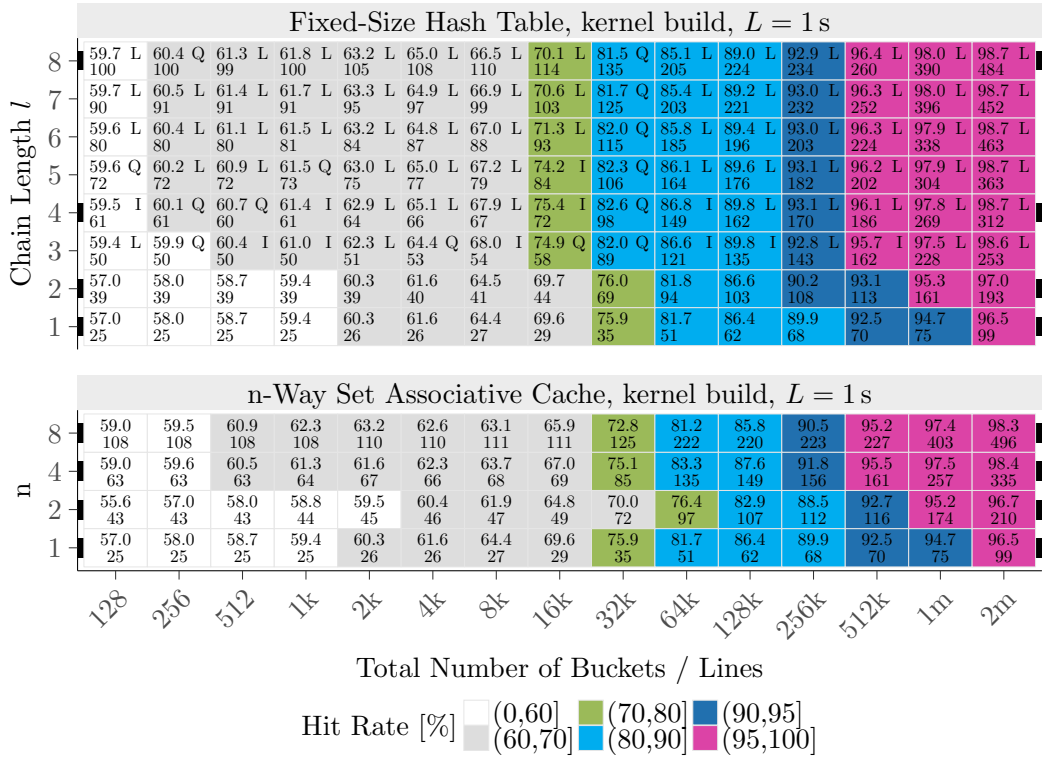


Figure 7.8: The fixed-sized hash table delivers higher hit rates than an n-way associative cache. However, a large direct mapped cache ($n = l = 1$) performs best overall. The plots give the hit rate in percent (top) compared to an unbounded hash table, the time in milliseconds (bottom) to process the checkpointed pages of a kernel build ($L = 1\,\mathrm{s}$), and the probing rule with the best hit-rate-to-time ratio (right).

found a potential duplicate of the new page in the checkpoint database. To rule out a hash collision, we do a 1:1 data comparison with the existing entry. Finally, if all probes remain unsuccessful, we evict one of the visited entries. We employ a least frequently used (LFU) replacement policy by selecting the entry with the lowest number of hits in the chain. While this generally favors old entries, we did not see any notable improvement in hit rates with least recently used (LRU).

With this data structure, we can also model a **direct mapped cache** and an **n-way set associative cache**. These work analogously to the described hash table, except that for the direct mapped cache, l is always 1, and for the n-way set associative cache, h computes the index of the first entry in the respective set, l equates to the set size n, and we always use linear probing.

In Figure 7.8, we compare the hit rates of the various configurations of the fixed-size data structure to an unbounded hash table for a kernel build. We can see that the configuration as n-way set associative cache performs worse than the equivalent configurations as hash table (marked in black). The hash table, in turn, does not provide any relevant improvements in the hit rate for $l > 3$.

In addition, linear probing (L) usually offers the best hit-rate-to-time ratio. If memory consumption is not a concern, however, the configuration as (larger) direct mapped cache delivers the best performance.

For a kernel build (12.5M pages at $L = 1\,\mathrm{s}$) and a hash table with 256k buckets and above, we approximately reach over 93% of the optimal hit rate. The same applies to postmark (4M pages) with a table of 64k-128k entries and SPECjbb (100M pages) with a 2M table[7]. In all cases, the data structure is fully occupied. This suggests the general rule of thumb that for a 93% hit rate, the data structure should comprise about 1 bucket per 8 checkpointed pages. This makes it difficult to choose a universally optimal solution upfront. Nevertheless, the configuration as a 1M direct mapped cache proved to be a good tradeoff between memory consumption, hit rate, and processing time in our tests[8]. Hence, we set this as default in SimuBoost and use it in all of our benchmarks.

Deduplication Ratio

With the data structure detailed in the last section, we have a strong filter at the entrance of the reduction pipeline. Measuring the median deduplication ratio per checkpoint – i.e., the percentage of pages removed from the input of incrementally captured new pages – shows the highly redundant nature of modified memory pages in periodic checkpoints (see Figure 7.9a). This confirms prior research in the field of memory deduplication (see § 2.3.3); although we determined much lower deduplication for SPECjbb 2005 than the authors of CloudNet [282] (up to 20% versus 60%).

Just like with the evaluation of the overall compression performance, we left out the first checkpoint which would otherwise heavily distort the results for the more lightweight workloads. Since the first checkpoint contains many zero pages, its deduplication ratio is with 87% exceptionally high. This reduces the 4 GiB guest RAM image to around 530 MiB. However, considering the effectiveness of deduplication over generic compression[9], we can infer that a substantial share of identical pages in the subsequent checkpoints are not zero pages, as these would also be very effectively compressed generically. In fact, zero pages account for only 0.03% of the total redundancy for postmark and for 41% and 6% in the case of the kernel build and SPECjbb, respectively. Instead, for these three benchmarks we observe between 150k and 1.4M different page hashes, which mostly (40% – 77%) occur only twice during the execution of the workload (see Figure 7.9b).

[7] See Figure B.2 on page 214.

[8] While adaptive caching policies such as ARC [170], which dynamically balances between frequency and recency, could potentially improve performance, we see > 90% of the optimal hit rate as sufficient for our purposes.

[9] See Figure 7.5a on page 143.

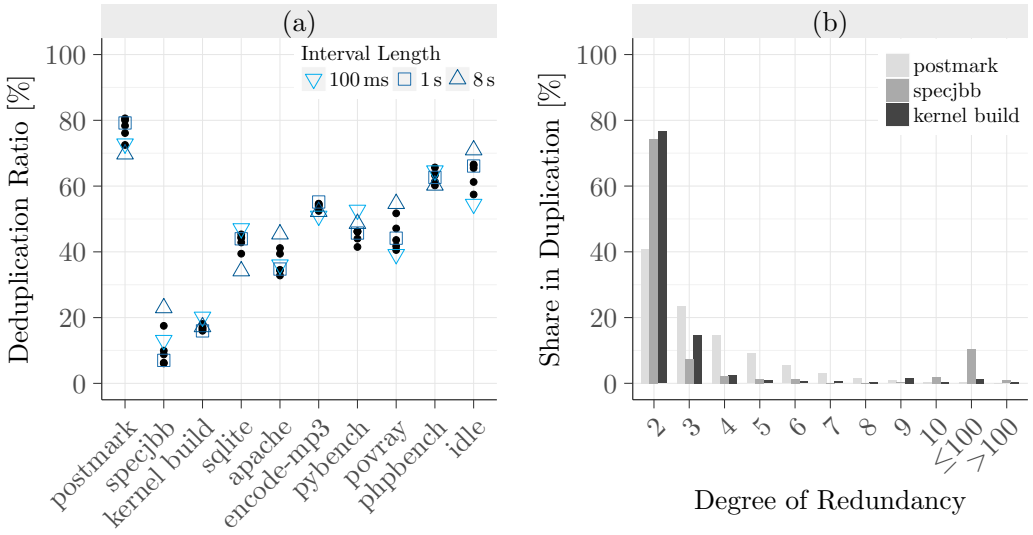


Figure 7.9: (a) The deduplication ratio over all checkpoints (except the first) is highly workload specific with the interval length being no clear indicator. The black points denote results for $L \in \{200, 400, 800, 2000, 4000\}$ ms. (b) The degree of redundancy describes how often certain page content can be observed. Most contents show up only twice.

In a study exploring the semantic origin of incrementally captured (i.e., modified) pages in a kernel build [224], we found that most duplicates stem from anonymous and free memory, but only 8% are located in the guest page cache.

7.2.2 Delta Compression

Whereas data deduplication targets *identity*, delta compression leverages *similarity* to achieve data reduction. The goal is to generate a patch with zeros on most positions – using `XOR` ($\oplus$) – so it can be encoded more efficiently than the original data. On this route we have to tackle three primary design questions:

Reference Selection Delta compression always depends on a reference against which the delta is computed. This reference (1) must be available during decompression, (2) should produce as many zeros as possible, that is, its degree of similarity should be high, and (3) the process of selecting it should be fast.

For the first requirement, we must ensure that the simulation node that eventually loads a checkpoint with a delta compressed page is also in possession of the corresponding reference page. Employing delta compression in a network is thus always challenging as it must only be applied within the set of pages that are local to a certain simulation node. This requires a directed simulation scheduling with a scheduling-aware reference page selection. Otherwise, decompression necessitates supplementary network accesses to retrieve missing reference pages,

which is counterproductive for reducing network traffic[10]. Nevertheless, for local operation as well as for the multicast network distribution proposed in § 7.3, delta compression is applicable and beneficial without having to consider the physical storage location of pages. In the following, we hence assume that all pages are available on all nodes, that is, all pages are potential candidates for reference.

The requirements (2) and (3) are tightly intertwined as finding the optimal reference page from the set of all previously seen pages is a computationally non-trivial task. Difference Engine [107] as well as Gerofi et al. [100] utilize hash tables and specifically designed hash functions which provoke hash collisions for similar page contents. We initially envisioned a similar approach, but eventually dismissed it in favor of a simpler and faster design: Under the assumption that on average pages are only marginally modified between successive checkpoints, we can generally expect the previous version of the same page to be a good candidate. This entirely eliminates the search for a reference page at the cost of a potentially lower compression ratio. Considering that we employ further data reduction mechanisms besides delta compression, we see this an appropriate tradeoff.

Decision Function Delta compression is only effective if the delta can be encoded more efficiently than the original data. However, if changes are too extensive, the encoding efficiency can quickly shift to the detriment of delta compression. If, for example, a randomly filled page R gets cleared by the guest (i.e., all zeros), the delta will contain the original page contents, as $\Delta = R \oplus 0 = R$. Encoding the new (zero) page is thus more efficient than encoding the delta. In consequence, we need a decision function which determines if delta compression should be applied. The *optimal decision function* computes both the fully compressed version (i.e., LZ4) of the delta and the original page and compares the resulting sizes. While this gives the best compression quality it also consumes the most time. We therefore opted for a heuristic based on the Hamming distance between the page's current and previous contents as a measure of similarity.

We calculate the Hamming distance by deriving the delta and counting the number of non-zero 64-bit words[11]. If the number exceeds a fixed threshold, we dismiss the delta and keep the original data. Compared to the optimal decision function, we save the costly compression of the delta or the original data, respectively. Figure 7.10 illustrates that selecting the right threshold is crucial for the quality of the heuristic. With a threshold of at most 25% non-zero words (i.e., 75% similarity) we achieve the best performance and reach between 79% and 100% conformity with the optimal decision function.

[10] Note that this is not the case for data deduplication because no additional data such as the delta must travel the network.

[11] Although current x86 CPUs implement fast bit counting with the SSE 4.2 `popcnt` instruction [125], we found word-level counting to be faster and sufficiently precise.

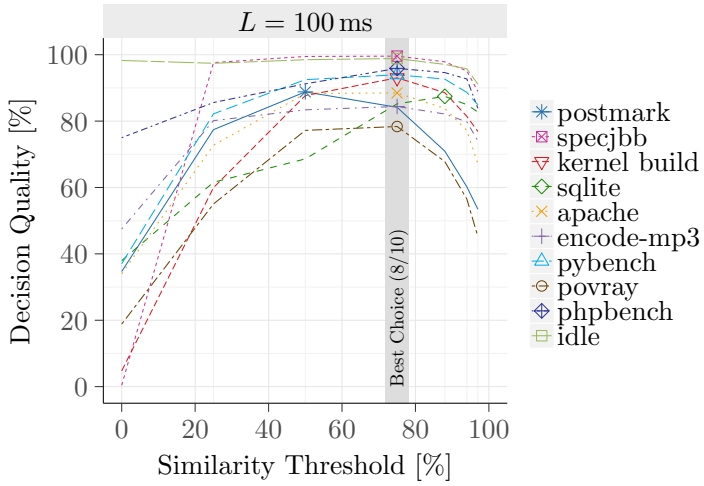


Figure 7.10: Choosing a similarity threshold of 75% – i.e., 25% of the delta are non-zero – gives the best average performance, where decisions match the optimal decision function in between 79% and 100% of cases. The shapes denote the maximum of each curve.

Chain Length Conceptually, there is nothing preventing a reference page to be itself delta-encoded. However, each time SimuBoost delta-encodes a page, it also has to load the corresponding reference page during decompression. This creates additional overhead not only for checkpoint loading, but also for deduplication and delta compression, where SimuBoost accesses (compressed) pages in the database. Although a page cache as in Remus [72, 180] and others [118, 187] could generally relax the situation, it is favorable to avoid long dependency chains in the first place. We therefore limit delta compression to non-reference pages.

As detailed above, we only consider the previous contents of the same page as a reference to avoid costly reference identification. In consequence, the limitation of delta compression to non-reference pages leads to a constant alteration between delta and non-delta encoding[12], wasting potential for compression. We therefore lock the reference until the decision function rejects the delta compression:

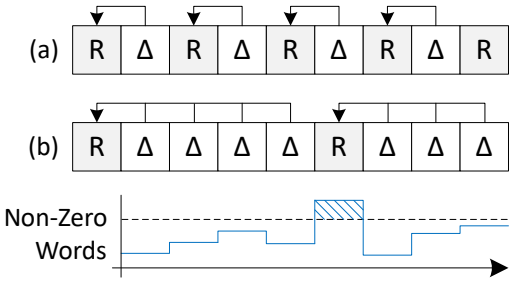


Figure 7.11: (a) Alternating between delta and non-delta encoding for a certain PFN. (b) The reference remains fixed until the decision function rejects delta encoding using this reference.

[12] Assuming the particular page is eligible for delta compression according to the decision function.

Delta Compression Ratio

Evaluating the compression ratio of all pages that passed data deduplication (i.e., they contain new contents), reveals that delta compression can be applied on average in between 20% and 80% of cases (see Figure 7.12a). The benchmarks again skip the first checkpoint to better reflect workload characteristics. As expected, short intervals are better suited for delta compression because they accumulate fewer changes.

In addition, Figure 7.12b shows that delta compression significantly reduces the storage size of the corresponding pages. This also confirms the effectiveness of our selection heuristic. Somewhat counterintuitively, the compression ratio for delta pages increases with the interval length. However, this is conclusive because for longer intervals only those pages with seldom and little modification remain, whereas the pages with more extensive modifications and thus less compressible deltas are already filtered out.

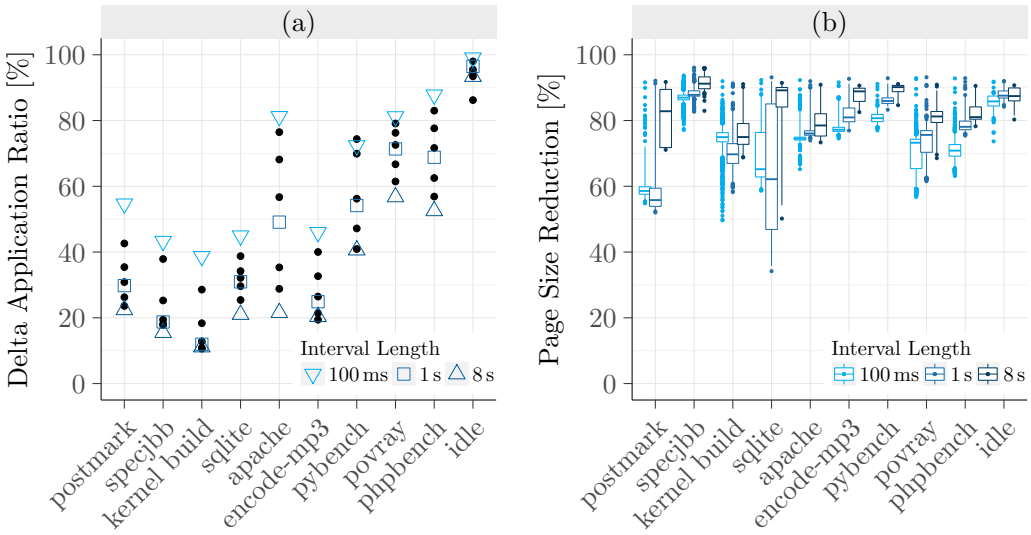


Figure 7.12: (a) As expected, short intervals and lightweight workloads provide higher potential for delta compression. The values relate to the remaining pages after deduplication. The black points denote results for $L \in \{200, 400, 800, 2000, 4000\}$ ms. (b) Delta compressed pages are typically at least 50% smaller than before.

7.2.3 Device State Compression

In the deduplication and delta compression stages, the incoming data volume is, except from the first checkpoint, determined by the workload's page modification rate, the interval length, and the actual page contents. It is thus difficult to predict the overall amount of data before actually executing the workload. Looking solely

at the RAM state map, it is much simpler. Although this does not include the miscellaneous other device states such as the CPU registers, even for a small 256 MiB VM the state map consumes over 4x the space of the other devices states typically stored by QEMU. This makes the state map the next largest position beside the actual guest memory contents.

> We therefore focus on the compression of the RAM state map and resort to generic compression for all other device states as illustrated in the overview of our data reduction pipeline in Figure 7.3 on page 139. However, due to the fact that we also had large sparse devices such as disks[13] in mind when designing the in-memory representation of state maps as well as their compression, some details go beyond the requirements of a RAM map.

Let P be the number of guest physical pages, then we can estimate that we generate $\frac{8 \cdot P}{L}$ bytes per second for storing one 64-bit offset into the checkpoint database per guest physical page per checkpoint. Considering our default VM with 4 GiB of RAM[14], we generate over 80 MiB/s with $L = 100$ ms. This accumulates to 47 GiB for povray (75% of its total data) and over 150 GiB for SPECjbb (17%). With $L = 1$ s, on the other hand, the data rate is only at 8 MiB/s, which is 7% of a Gigabit Ethernet link. As very large VM configurations with tens of gigabytes of RAM are less likely, we can conclude that in practice an efficient device state compression is primarily important for short checkpointing intervals.

From an encoding perspective, the state map possesses a number of interesting properties:

1. **Equal Offsets** Due to data deduplication, pages with the same contents map to the same offset in the checkpoint database. In addition, there are often contiguous ranges of pages with the same contents (e.g., zero pages) that all receive the same offset.

2. **Small Offset Deltas** Pages that cannot be deduplicated are compressed and appended to the checkpoint database. The delta between the offsets of two consecutive pages in the state map is thus in most cases (i.e., no deduplication) very small; smaller than 4 KiB (+ metadata)[15].

3. **Data Alignment** Entries in the checkpoint database are aligned to 16 bytes. The low 4 bits of each offset are thus always 0; except for sparse regions (e.g., in disk maps) that hold the invalid offset `0xFFFFFFFF FFFFFFFF`, in the following simply referred to as `INV_OFF`.

[13] For disks, checkpoints store only modified sectors in reference to a base image.

[14] That is, 1052704 pages including device memory regions such as the video buffer.

[15] This depends on the processing order in the storage backend, which in turn is determined by the page submission – i.e., the order in which the VMM sends pages – and the non-deterministic multiprocessing. Nonetheless, we always submit pages in ascending order of their guest physical address. Furthermore, the backend processes at minimum 500 pages per job, among which the order is not affected by multiprocessing.

We devised a custom compression scheme called *SimuBoost Device State Compression (SDS)* which leverages these properties for dense encoding of state maps.

To save space for sparsely populated state maps, the actual in-memory data structure is a hierarchical two-level table, similar to a page table in virtual memory systems. In the default configuration, the directory table holds 2^{12} second-level tables each covering 2^{16} 64-bit offsets into the checkpoint database for a maximum device size of 1 TiB with 4 KiB pages. Just like with conventional page tables, each entry's address in the address space of the corresponding device (e.g., the guest physical memory) can be inferred from its location in the table hierarchy.

To simplify the design, the (de-)compression works at the granularity of the second-level tables. That means we seek a space-efficient encoding for an array of 2^{16} 64-bit offsets, leveraging the domain-specific knowledge presented in the properties (1) to (3). The compressed representations of the individual tables are then simply concatenated to create the final output.

> For the sake of brevity, we only cover the process of compression. Interested readers may consult the source code for details on decompression.
>
> `/simutrace/storageserver/simuboost/SimuBoost1DeviceState.cpp`

Compression

A simple way to benefit from properties (1) and (2) is to use delta encoding. Instead of storing an array of absolute offsets, we only save the first offset in each table in absolute form. All following $2^{16} - 1$ offsets are translated so that they are relative to their respective predecessor in the table. The resulting deltas are in most cases much smaller values, which can be represented with less than 64 bits. An exception to this are deduplicated pages, where the relative offset may still span several tens of gigabytes. The zero page, for example, will probably be part of the first checkpoint and in consequence, it will be located at the beginning of

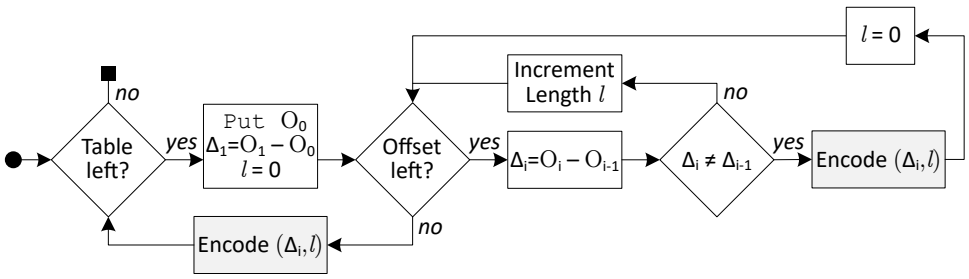


Figure 7.13: Overview of the SDS compression loop. O_i denotes the absolute offset in the current table at position i. Δ_i is the corresponding relative representation. l holds the length of a consecutive run of identical deltas. `Put` writes a value to the final output.

the checkpoint database file. If it is referenced by a state map of a much later checkpoint (i.e., many more pages added to the database), the delta for this entry will be a large negative number. The resulting array of small delta values is thus typically interrupted by sporadic large values jumping back and forth between deduplicated and new pages (and `INV_OFF`). Conversely, a contiguous range of identical offsets leads to a sequence of deltas with value 0. To efficiently encode such areas, we incorporate run-length encoding (RLE). Figure 7.13 summarizes the central compression loop.

In the next step, SDS tries to find a compact encoding for each delta. If LZ4 cannot further compress a page, the page's final size in the database including metadata is 4144 bytes. We thus observe that most deltas are below this value. To further compress the representation of these deltas, we employ a dictionary that stores previously seen deltas. We use the fixed-size hash table described in § 7.2.1 for this purpose. A configuration with 64 buckets, incremental probing, and a chain length of 4 proved to be effective. Hits – called *matches* – can then be encoded by their position in the dictionary using only 6 bits ($2^6 = 64$). Misses, on the other hand, are directly written to the final output. The same applies to overly large deltas. We refer to both as *literals*.

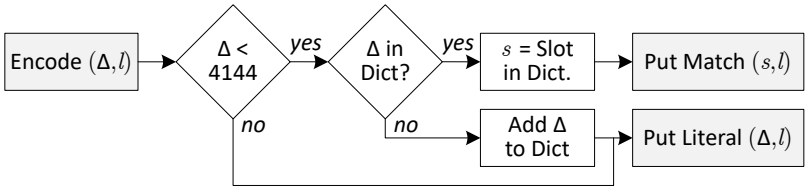


Figure 7.14: Overview of the SDS encode function. Small delta values are matched against a dictionary of previously seen delta values. Hits are encoded as their position in the dictionary, misses and large deltas are written as literals.

The last step is to write the matches and literals to the final output. In this process, SDS discards the first 4 bits which are always 0; property (3). Since the majority of literals are usually smaller than 2^{16}, we provide separate encodings for short and long literals, consuming 2 bytes and 4 bytes, respectively. Matches are encoded using 1 byte only. In each case, a run-length extension of 2 bytes can be added to express up to 2^{16} repetitions. See Figures B.3 and B.4 for more details on the encoding format.

SDS Compression Ratio and Time

To get an impression of SDS's performance, we contrast its compression ratio and time with that achieved by generic LZ4. Since a key element of SDS is its delta encoding, we further perform a generic LZ4 compression of the delta (*LZ4+Delta*). This allows us to better discern which aspect of SDS is responsible for performance

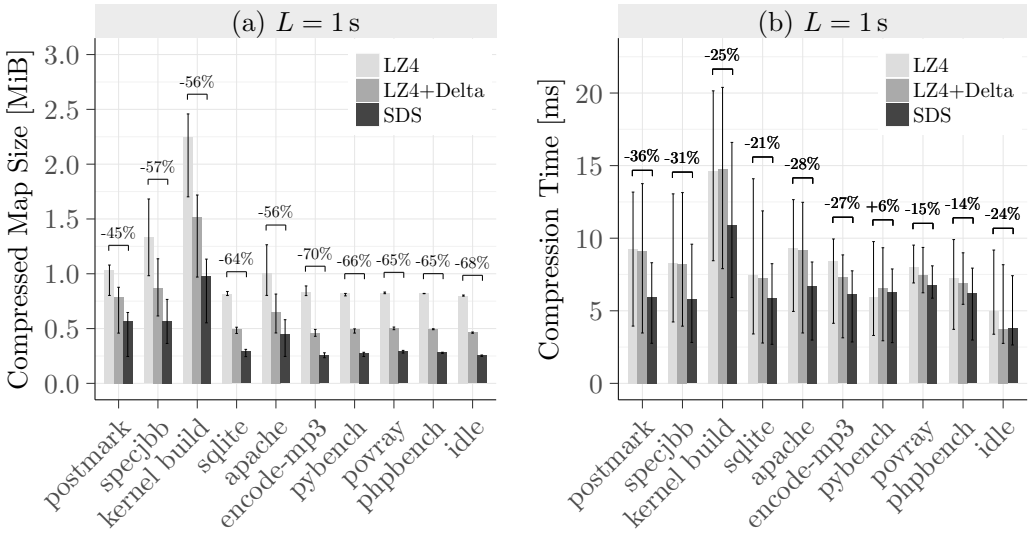


Figure 7.15: (a) SDS achieves on average 61% higher reduction in size than LZ4 and still 39% higher than LZ4+Delta. The 8 MiB state maps are compressed down to between 256 KiB and 1 MiB, depending on the workload, which is a compression ratio of 32:1 and 8:1. (b) At the same time, SDS is on average 21% faster than LZ4. Improvements are especially high for heavyweight workloads with complex state maps.

differences. This time, we include the first checkpoint in the benchmarks because checkpoints inherit the state maps from previous checkpoints anyway.

The results show that SDS delivers substantial higher compression than LZ4 in less time (see Figure 7.15). Only for pybench, the compression with SDS is on average slightly slower. The comparison with LZ4+Delta reveals that around half of the improvement in size reduction can be attributed to the delta encoding. The other half stems from the dense encoding. As LZ4 also supports run-length encoding, we do not assume RLE to be a major factor. The results for the compression time demonstrate that the delta encoding does not inherently translate to a faster generic encoding. We can thus conclude that the custom encoding scheme in SDS is responsible for the advantage in compression time.

7.3 Multicast Checkpoint Distribution

With effective data compression in place, we are able to push the incremental checkpoint data into the simulation network. Pushing with undirected scheduling and data distribution implies that the data must be stored on the simulation nodes in some form that allows all nodes access to non-local data. A possible solution is to employ a conventional distributed file system such as CephFS [275]. The file system is mounted into each node's virtual file system (VFS) tree and

abstracts the actual storage location from applications. CephFS, for example, transparently stripes files across the participating nodes and optionally replicates frequently used contents to improve throughput and load balancing within the network. A program accessing data on the distributed file system is not aware of any network transfers happening in the background. For SimuBoost, a distributed file system is thus a convenient gateway to the network. We can simply switch the target directory for the checkpoint database and files from a local device to the distributed file system. On the simulation nodes, the files can then be opened like if they were stored locally.

We tested two distributed file systems, namely CephFS [275] and GlusterFS [43]. However, we encountered two major performance issues [209]:

First, when appending the checkpoint database during checkpointing, we observed sporadic delays of up to 90 s with CephFS and frequent delays of multiple seconds with GlusterFS. We attribute these delays to possible re-balancing operations and general load from simultaneous reads by the simulation nodes, which were concurrently starting simulations. However, further experiments are needed to confirm this presumption.

Second, loading checkpoints even from a fully distributed checkpoint database could take minutes with both file systems. This was especially the case with later checkpoints toward the end of a benchmark. The low performance is rooted in the fact that the structure of the append-only database in conjunction with

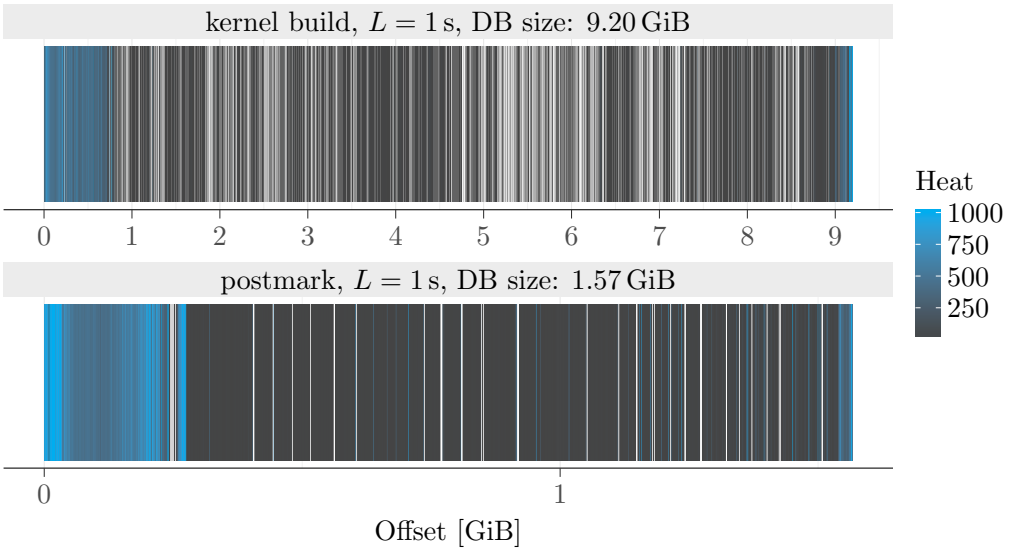


Figure 7.16: Heatmaps of the references on the respective checkpoint database when loading the last checkpoint. Accesses are heavily scattered throughout the entire file. Resolution is 1 MiB with counts capped to 1000, limiting otherwise dominating areas such as the one containing the zero page. White areas denote ranges with no accesses.

deduplication and delta compression leads to thousands of small I/O requests scattered throughout the entire database (see Figure 7.16). Since SimuBoost uses memory mapping to access the database, reads are triggered by page faults at the granularity of 4 KiB. Benchmarks for GlusterFS suggest that this access pattern is particularly slow, irrespective of the stripping or replication scheme, or the number of nodes[16].

Although these are preliminary results only and both file systems offer vast options for tuning, we decided against using a distributed file system. We expect the access pattern to be generally unsuited for network access. Looking at previous work that deals with the distribution of checkpoints (i.e., virtual machines) [232], we instead opted for distribution based on multicast (point-to-multipoint). This technique has also been successfully utilized in SnowFlock [146] and VMScatter [69] to quickly spawn or migrate VMs targeting multiple destination hosts.

In contrast to unicast, each multicast message is sent to all systems configured for the same recipient group in a *single* transmission, regardless of the number of nodes. In the case of SimuBoost, we can employ multicast so that all simulation nodes receive all checkpoint data and are subsequently able to load every checkpoint without further network access. In addition, writing the received data to disk preheats each node's file system cache, which notably improves loading times for demanding workloads[17]. On the downside, all nodes need to be equipped with sufficient disk capacity to store the entire checkpoint database as well as all checkpoint files and supplemental information such as replay logs and bitmaps for sparse loading.

IP natively supports multicast with a designated address range to specify (private) multicast groups (`239.0.0.0` to `239.255.255.255`). We can integrate 1-to-*N* multicast distribution into SimuBoost by additionally routing writes of compressed checkpoint data on the virtualization host to a multicast socket. On the other end, a listener application, which runs on all simulation nodes, receives the packets and reconstructs the original files. To discern for which file the received data is intended, we add a small header to each packet that includes the file name.

Since multicast is inherently not connection-oriented, User Datagram Protocol (UDP) is often employed at the transport layer. This has important implications:

Packet Ordering Whereas protocols such as TCP guarantee that packets reach the client application in the same order they were sent, UDP packets may be received in arbitrary order. In consequence, the listener application cannot just append new data to the end of the specified file.

We can solve this problem by utilizing sparse files at the destination and adding a file offset to the packet header.

[16] 8 parallel reads using the `dd` command on 512 KiB files peak at no more than 10 MB/s [43].
[17] See Figure 6.19 on page 131.

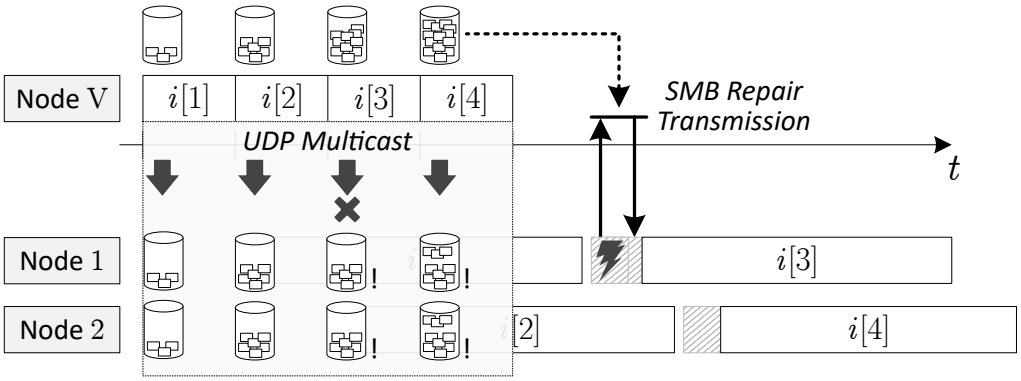


Figure 7.17: With multicast, all simulation nodes receive all checkpoint data and store it locally. If packets get lost, loading an affected checkpoint fails and a lazy repair is done by reading the missing parts from the healthy copy on the virtualization host using a reliable SMB connection.

Reliability UDP does not establish a back channel for informing the sender about successful packet delivery. Packet loss therefore goes undetected. This may happen when the receiver cannot keep up with the data rate and buffers in switches, network adapters, or the operating system overflow. Likewise, physical defects such as broken cables or loose connections can lead to packet loss.

Dealing with unreliability in multicast has a long history in research and many specialized protocols have been presented [33]. A simple technique is to inform the sender of successful transmission using acknowledgments (ACKs) or to signal packet loss with negative acknowledgments (NACKs). While using such a protocol is a viable solution, we found that for SimuBoost strong reliability in multicast is not needed. As each node only loads a limited set of (different) checkpoints we can tolerate data loss as long as it does not affect the checkpoints of interest. Whereas a reliable multicast protocol would immediately initiate retransmission, it is sufficient for SimuBoost to lazily detect "holes" in the created sparse files when actually attempting access[18]. The corresponding node can then fetch the missing parts from the original copy on the virtualization host using a reliable file sharing protocol such as Server Message Block (SMB). As we do not have to expect high packet loss in a restricted research network, repairs should be a rare occurrence. Furthermore, due to the simulation slowdown, this might be after the original execution on the virtualization host ended and multicast traffic completed (see Figure 7.17).

Despite packet loss, data corruption is a potential concern. However, UDP includes a 16-bit checksum, where corrupted packets are automatically dropped. In the improbable event of a checksum collision (in a stable research network), we can in turn expect checkpoint loading to fail because most data is compressed. In both cases, a repair can be attempted using the SMB file share.

[18]This can be done using `lseek` with `SEEK_HOLE`.

We implemented the described multicast distribution approach in an early prototype and preliminary results confirm its viability [209]. To maintain a tolerable packet loss (less than 1.5% for a kernel build) we had to increase the listener's socket receive buffer from 208 KiB to 20 MiB[19]. The experiments further showed that lazy repair over a secondary SMB connection is an efficient solution that creates only negligible extra traffic. As a result, the loading times with multicast are comparable to the ones when loading checkpoints with a hot cache locally on the virtualization host.

> Since we did not integrate multicast distribution in the current SimuBoost version yet, we do not present any further results, but instead refer the reader to [209]. We use an alternative method based on pre-distribution in Chapter 9 to nonetheless evaluate SimuBoost.

7.4 Conclusion

In this chapter, we presented how SimuBoost can distribute checkpoints and accompanied data in a cluster of simulation nodes using contemporary network technology such as Gigabit Ethernet. Whereas pulling entire checkpoints from the virtualization host creates a bottleneck and greatly limits scalability, incrementally pushing new data into the network is a viable solution. A key component in this course is the demonstrated data reduction pipeline, implementing a combination of data deduplication, delta, and generic compression as well as custom device state compression. It reduces the required network bandwidth by a factor of up to 39, enabling pushing for intervals as short as 100 ms, even for demanding workloads. Compared to pure generic compression, the specialized pipeline achieves significantly higher compression at comparable computational overhead.

Although we do not generally rule out pushing based on distributed file systems, first experiments remained disappointing with long delays during checkpoint storage and loading. We therefore opted for a distribution built around IP multicast and preliminary results confirm previous work that recommends this technique for point-to-multipoint transfer of virtual machines. With multicast, all simulation nodes receive all checkpoint data and are thus able to load checkpoints without having to retrieve substantial amounts of data from other nodes first. We discussed various technical challenges bound to the unreliability of multicast transmissions and proposed a lazy repair mechanism based on secondary (reliable) SMB connections. This solution already proved to be effective and efficient in first experiments.

[19] The simulation nodes ran a single-threaded multicast listener with synchronous I/O. As the built-in SSDs provided a write rate above Gigabit Ethernet, it is likely that the single thread created a receiver-side bottleneck.

Chapter 8

Heterogeneous Deterministic Replay

As detailed in § 4.2.1, running simulations based on continuous checkpoints is not sufficient to exactly reproduce the execution from the hardware-assisted virtual machine in the (parallel) simulation. This is because the checkpoints miss all non-determinism that affects the instruction flow in between two consecutive intervals. This includes input from users or remote devices, but also less notable factors such as the precise timing of interrupts and even indirect effects from instructions executed outside the virtual machine.

The common approach to capture this non-determinism is deterministic replay, where the hypervisor records enough information during the original run so that it is able to accurately replay the execution. Accurate replay is difficult when the execution environment changes, which is the case when SimuBoost moves from recording in a hardware-assisted virtual machine to a replay in a purely software-based functional simulation. This change considerably increases the level of complexity because it implies that for all *deterministic* operations the software-based simulation behaves functionally equivalent to the previously used hardware platform. Such a heterogeneous replay system thus has to master two primary challenges: (1) accurately inject recorded non-determinism and (2) guarantee identical outcome for all otherwise deterministic operations. The latter requires us to refine the simulation and in some circumstances extend the recording logs over what is usually sufficient for homogeneous replay systems.

For SimuBoost, we intend to record the guest system at the machine level, thereby covering the full system including all privileged operating system components. Although this first seems to be a waste of computational resources when only a particular context or kernel module is of interest, the evaluation of V2E [287] showed that discerning between recording and non-recording realms comes at a significant run-time overhead (5x to 17x slowdown) during recording. Even the

log growth rate in V2E is comparably high because realm switches often trigger the copy of entire memory pages. When the operating system should be included (as in our case), it is thus not only more flexible but also more lightweight to record the full system.

In research, but also in the commercial world, heterogeneous full system solutions are rare. In fact, we are aware of only four projects, of which three share the same (closed-source) codebase and have been developed by VMWare for their series of proprietary virtualization products (see § 2.4.2). This leaves us with V2E as the only publication with a research focus. Unfortunately, the source code is not publicly available either. In this chapter, we therefore provide an insight into some of the technical challenges we have faced when building a heterogeneous full system deterministic replay.

We start with an overview of the general architecture of our replay system in Section 8.1. This includes an evaluation of the system's run-time overhead, the log growth rate, and log compressibility. In Section 8.2, we present critical points at which we had to adapt the simulation to precisely mimic the behavior of the hardware platform used for virtualization. We conclude in Section 8.3.

> We integrated SimuBoost into QEMU/KVM (§ 2.2.5) with a focus on the x86 architecture - i.e., we record the hardware-assisted execution in KVM and replay the events with the TCG binary translator in QEMU. The following explanations are consequently specific to this platform. Conceptually, SimuBoost can be ported to other architectures, virtualization products, and simulators as long as a fast hardware-assisted VM is available. Whereas checkpointing and data distribution are mostly architecture-independent, the recording and replay are tightly linked to low-level (functional) hardware behavior. To explore the potential for other platforms, we have developed a heterogeneous replay system for the ARMv7 architecture [262]. We also include some findings from this work.

8.1 General Architecture

To control recording in KVM and replay in TCG we added commands to the QEMU monitor that start (`start−rr`) and stop (`stop−rr`) the deterministic replay facility, where the start command accepts the mode of operation – i.e., `record` or `replay`. Deterministic replay and checkpointing must be synchronized so that on a checkpoint the log is pushed into the simulation network. We therefore added an explicit flush of the event log when SimuBoost creates a checkpoint[1]. Moreover, each checkpoint remembers the current offset in the log so that after

[1] Otherwise, Simutrace receives data only in chunks of 64 MiB, which is the default size of one segment in the shared-memory buffer between client and server.

loading a checkpoint in the simulation the replay starts at the right position. In order to recognize an interval end, each checkpoint further adds a special marker event to the log.

According to § 2.4, we have to deal with three types of events: synchronous events, asynchronous events, and compound events. In practice, we can generalize this into having *synchronous* and *asynchronous* events that may or may not carry additional payload. On top of this, asynchronous events must always possess a landmark which specifies their precise location in the instruction stream. Nonetheless, adding a landmark also to synchronous events helps to detect divergences in the replay. We thus use the following basic structure for all events, where `Type` specifies the exact event type such as *Read APIC* and `Length` provides the size of the payload:

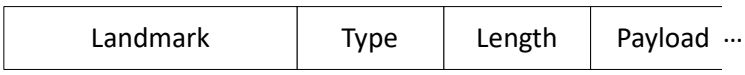


Figure 8.1: Basic event structure.

Despite their common structure, storing synchronous and asynchronous events in the same log is cumbersome. Consider the scenario in Figure 8.2, where we want to replay the execution of a waiting loop. In the recording phase, we captured numerous reads of the CPU's time stamp counter (`RDTSC`), one at the beginning and one for each loop iteration. During the second iteration, an interrupt (`INT`) occurred. For brevity, we assume that the binary translator in the simulation generates a single translation block (TB) for the loop body, including `getTime()` and the condition evaluation. Then the second iteration must run a different version of the TB in order to inject the interrupt. In consequence, we have to know when the next asynchronous event needs to be replayed *before* generating a translation block and *before* having replayed an arbitrarily long sequence of preceding synchronous events. To avoid repetitive, time-consuming discovery of asynchronous events during replay (which would be a linear search), we therefore

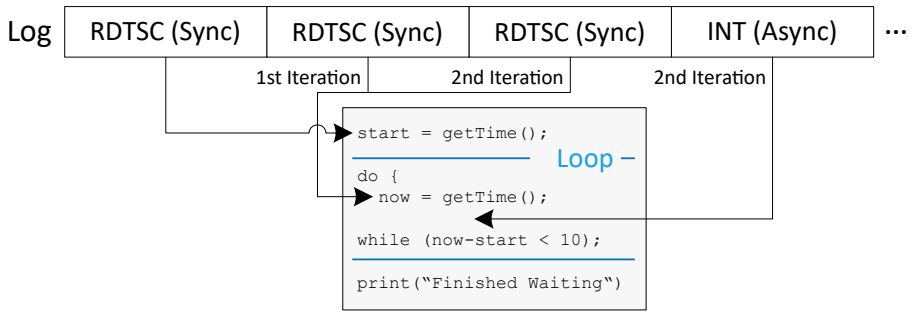


Figure 8.2: The occurrence of asynchronous events must be known before generating a translation block (TB), otherwise the block will not include the replay. Here, the 2nd loop iteration requires a different translation.

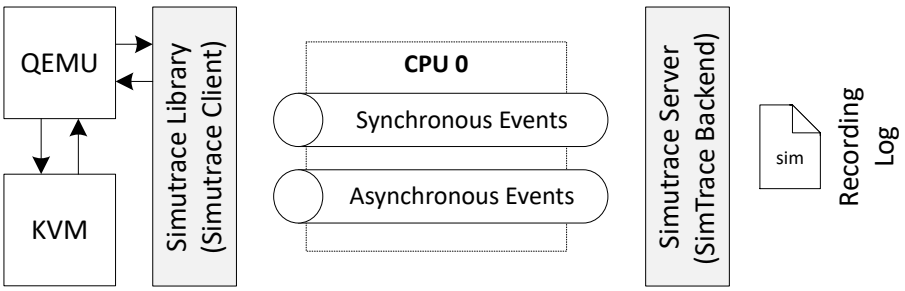


Figure 8.3: We use the general-purpose tracing backend of Simutrace for fast asynchronous compression and storage of non-deterministic events.

put synchronous and asynchronous events into separate logs. This way we always have direct access.

Just like with the checkpoints, we use Simutrace [218] for storage, this time using the regular tracing backend and general-purpose LZMA [9] compression. The separate event logs are implemented as two streams of the same store, creating a single file on disk. Network distribution (§ 7.3) then works in the same way as for any other file; for example, the checkpoint database.

Over the course of a recording session, SimuBoost captures various types of events (see Tables B.1 and B.2 for a complete list). As we are running a hardware-assisted virtual machine with KVM during this phase, the majority of events are recorded in kernel mode. To simplify the design, we do not directly write the events into the user-mode buffer supplied by Simutrace, but instead use a combination of slab and generic memory allocation to temporarily store the events in kernel memory[2]. Whenever the vCPU thread leaves the kernel, we collect all events using the same technique as for iteratively copying pages from kernel into user mode (§ 6.1.4).

8.1.1 Landmark

Existing deterministic replay systems use different landmark designs depending on the target architecture and the intended purpose for replay. With a focus on restricted memory consumption, Sundmark and Thane describe a checksum of CPU registers and stack contents for embedded systems [249]. Similarly, V2E [287] saves a majority of the CPU state (including the instruction pointer, registers, and flags) and simply compares the state during replay – *after each instruction* – with the saved one of the next asynchronous event. Although the authors do not explicitly quantify the overhead of this approach, it can be expected to have a severe impact on the simulation speed, considering how sensitive DBT engines react to bloat in translation blocks (see § 3.1). Besides performance concerns, this

[2]Otherwise, we would have to potentially request new buffer space from user mode in the middle of dispatching a VM exit. We use the slab cache for the common event structure and the generic memory allocation for the optional event payload.

solution also entails the risk of false positives as the CPU state alone is not unique; the easiest example being an endless loop of form `0x00: jmp 0x00`.

At the other end of the spectrum, the combination of the retired instructions counter and the RCX register[3] creates a minimalistic landmark [63, 194]. It is superior to the aforementioned design because the landmark is unique and the instruction counter allows the DBT engine to pinpoint the exact location for event injection. On the downside, this landmark requires hardware support for instruction counting during recording. Although x86 does come with a hardware instruction counter (implemented as a performance counter), it is known to be unreliable [194, 272].

A popular variant therefore drops the instruction counter and incorporates a combination of branch counter and instruction *pointer* instead [50, 87, 128]. The resulting landmark is still unique but tends to be more reliable.

However, when designing a landmark, one always has to bear in mind that the underlying counters have to be replicated in the simulation. We therefore chose a landmark based on the retired instructions counter. This is because we see detailed tracing as a primary use case for SimuBoost. In this context, we regard an instruction counter as an important metric to represent the sequence of operations and to locate events in the instruction flow – more so than a branch counter. In addition, research has shown that there are still variations and bugs in counting for the branch counter depending on the specific processor version [274]. Compensating for unreliable counters thus has to be done (on x86) anyway[4].

We consequently turned to using a *fuzzy landmark*. This landmark combines the retired instructions counter, the instruction pointer (RIP), and the RFLAGS register with a snapshot of the 16 general-purpose x86 CPU registers[5]. For asynchronous events, we position a short window around the instruction count given in the landmark and single-step in this range, performing a 1:1 register compare after each instruction. Only when all registers match we inject the asynchronous event and continue normal execution. This allows us to compensate for little mismatches in the instruction counting between simulation and hardware (see Figure 8.4). In order to prevent errors from accumulating and needing ever larger windows, we remember the offset and correct subsequent landmarks accordingly. By adapting the landmark values instead of the simulated instruction counter, the counter does not experience jumps, which would be incomprehensible in traces. We apply the same corrections for synchronous events but omit the single-stepping window.

[3] The register is necessary because x86 counts REP-prefixed instructions as one, irrespective of the number of repetitions [272]. RCX in turn stores the current iteration.

[4] The instruction counter on our ARM platform turned out to be perfectly reliable [262].

[5] RAX – RDX, RSI, RDI, RSP, RBP, and R8 – R15. We could restrict the set to RIP and RCX, but a comprehensive landmark reveals deviating replays more quickly and helps with debugging.

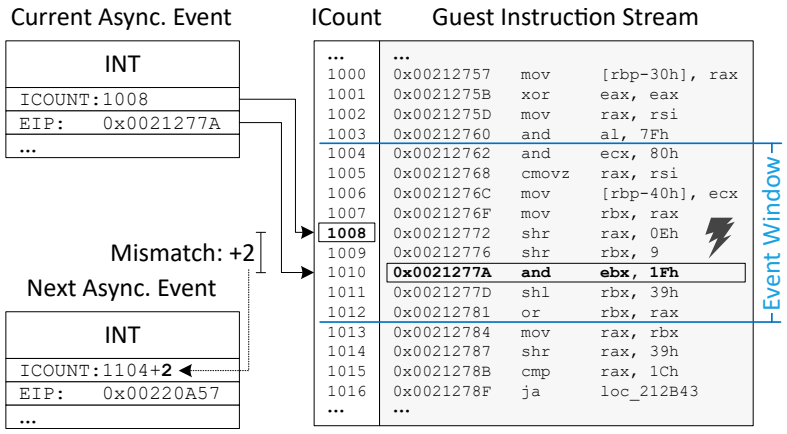


Figure 8.4: We position a window around the landmark's instruction count and single-step in this range, doing a 1:1 compare of the register state after each instruction. Following landmarks are automatically corrected.

Implementation

To count retired instructions on x86 we use the `IA32_PERF_FIXED_CTR0` performance counter, which is fixed to `INST_RETIRED.ANY`. As we are only interested in instructions executed by the guest and not by the hypervisor, a naive way is to manually toggle the performance counter at VM enter and exit. However, this also includes host interrupt handlers that are immediately executed after a VM exit. We therefore use a VMX feature that allows automatic switching of MSR configurations when transitioning between host and guest[6]. In addition, we instrument various operations in KVM to manually count instructions; for example, when KVM traps and emulates an instruction like in the case of port I/O (i.e., the `IN` and `OUT` instructions). Note that KVM virtualizes the performance monitoring unit (PMU) for the guest. The instruction counter is thus still available.

As expected, we also found the `INST_RETIRED.ANY` counter to be unreliable, confirming previous work. In our case, it sporadically undercounts by one. We first suspected the special counting rules discovered by Weaver and McKee [272]; for example, that hardware interrupts and page faults count as an additional instruction. However, these are not responsible. Since we have seen only undercounts so far, we configure the window so that we compare registers for up to eight instructions following a landmark (i.e., in Figure 8.4, from 1008 to 1015).

To integrate asynchronous event injection into QEMU, we have to adapt the generation of translation blocks. The code excerpt given in Figure 8.4 translates to a single TB, with the jump instruction at `0x0021278F` terminating the block. If the vCPU previously visited the code, a translation block for the whole excerpt may still be in the code cache. When the vCPU then reaches the code again, the

[6]Toggling the counter using `VM_ENTRY_(LOAD|SAVE)_IA32_PERF_GLOBAL_CTRL` does not work due to an unfixed CPU bug. Instead, we have to use the MSR-load area in the VMCS.

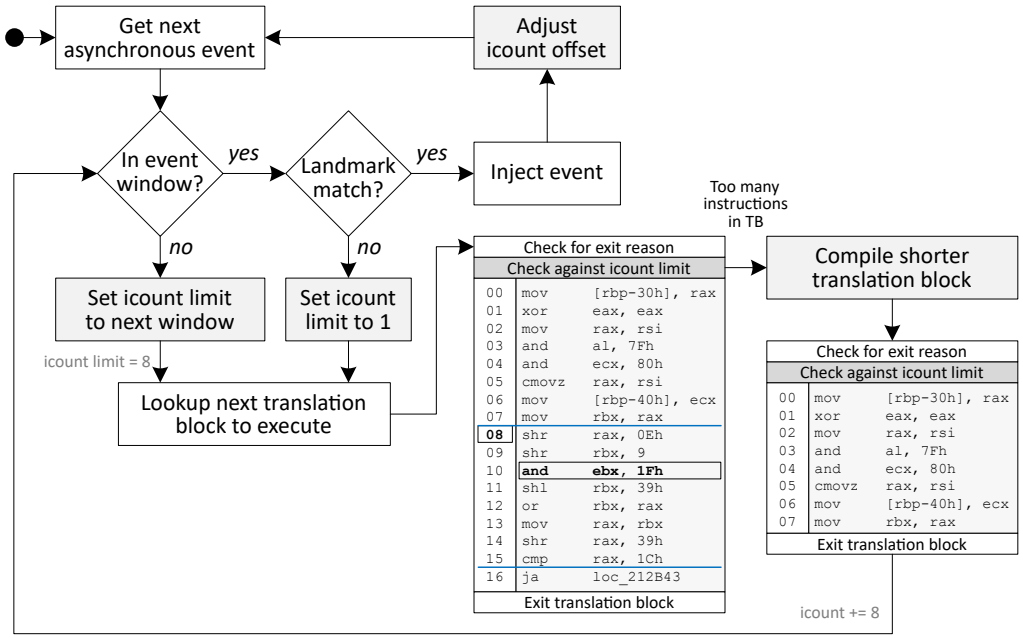


Figure 8.5: Simplified overview of CPU loop during replay. A check in the TB prolog prevents violating single-stepping windows. If necessary a shorter TB is generated (figure based on [262]).

control logic takes the existing TB, which blindly runs to end without injecting the event or even implementing the single-stepping window. Extending the control logic with appropriate checks before starting a translation block is no option due to TB chaining. While flushing the code cache after each asynchronous event is a possible solution, it is a rather expensive one. We therefore test in the TB prolog how many instructions are left until the beginning of the next single-stepping window. If the length of the TB exceeds this limit, the execution returns to the control logic which in turn requests a block of the remaining permissible length. To perform single-stepping, we allow only one instruction to be executed until the vCPU has to return to the control logic for landmark comparison.

Special care needs to be taken for instructions with the `REP` prefix. From the binary translator's point of view, each iteration constitutes one instruction because the conditional jump inherent to the `REP` prefix ends the underlying translation block. During normal operation, the translator achieves high simulation speed by chaining the generated TB to itself. The prefix is often used together with the `MOVS` and `STOS` instructions to implement `memcpy()` or `memset()`, respectively. According to the retired instructions counter in Intel CPUs, it thereby hides possibly hundreds of repetitions of the same instruction behind what is counted as only one instruction. This can severely prolong the actual single-stepping phase over our intended maximum length of 8 instructions and notably harm simulation speed. We therefore inline the test for the `RCX` register in this particular TB and only leave the block for a complete landmark check on a match. Otherwise, execution of the `REP`-prefixed instruction continues as normal, including TB chaining.

Similar performance considerations have to be made for the implementation of the actual simulated instruction counter. Incrementing the counter after every instruction is slow, albeit easy to do. To reduce the frequency of counter manipulations we only update its value when leaving the translation block; either by reaching the block's end or by invoking helper routines. We further added a new mnemonic to QEMU's intermediate language that translates to an `ADDM` instruction on x86 and allows us to increment the counter by a 32-bit immediate value directly in memory.

8.1.2 Replay Boundary

When replaying a full system, we can to some degree decide where to exactly place the *replay boundary* in relation to the virtual devices connected to the system: The most restrictive approach interprets everything outside the vCPU as part of the replay log. In this configuration, the result of every I/O operation is meticulously recorded so that the replay does not rely on any implementation of virtual devices. Instead, the replay is solely fed by the log and all I/O directed toward virtual devices can simply be dropped [285]. This gives a maximum degree of flexibility when it comes to the choice of the simulator because it only has to provide a corresponding CPU model. Such a restrictive variant is also easier to develop and validate. On the flip side, saving all I/O results produces the largest log.

A less comprehensive logging approach actively involves (some of) the virtual devices in the replay. For example, accesses to secondary storage devices can be allowed to pass so that the log does not have to embed the data read from disk [76, 87], only the interrupts. In this case, a matching base image from the start of the recording phase is needed. This concept, however, cannot generally be applied if the data is not recoverable from its original source during replay, such as in the case of a virtual network adapter.

For SimuBoost, both approaches are viable. The first one is especially appealing due to its simplicity, its flexibility, and because it does not require the checkpoints to include a disk image. Nevertheless, we do not drop all virtual devices in the simulation. This has the following two reasons:

First, when developing the replay, we wanted to be able to monitor the output of the system by displaying the screen contents and connecting to the serial port (read-only). To enable the replaying system to control these devices, we allow the vCPU to write to certain I/O addresses[7]. The initial state of the devices (e.g., the screen buffer) is restored from the checkpoints. In order to maintain the deterministic execution, we do not accept input from any device. That means we always redirect reads of I/O addresses to the replay log and drop newly generated interrupts.

[7] For a complete list of permitted addresses, see Table B.3.

Second, we checkpoint the device states with the interface that comes with QEMU. This interface does not explicitly save the configuration of the physical address space (e.g., the mapping of MMIO regions, ROMs, etc.), but merely reconfigures the underlying virtual chipset by restoring its configuration registers. Remember that we checkpoint the memory based on QEMU's RAM blocks (see § 2.2.5), which require adequate mapping when loaded. Therefore, our prototype includes the respective chipset and allows certain I/O commands to pass in the replay, as otherwise these memory regions will not be mapped correctly and connected devices such as the VGA adapter do not work. A partial solution would be to checkpoint the final physical address space instead of the individual RAM blocks. However, this would not allow replaying the BIOS and other early boot code that changes the mapping of the physical address space at run time by reconfiguring the chipset. An alternative would be to explicitly record and replay the mapping and changes thereof as asynchronous event, but we have not experimented with this solution yet.

Direct Memory Access (DMA)

A majority of I/O in a modern computer system is performed using direct memory access (DMA), that is a device (e.g., the disk controller) is able to read from or write to main memory without assistance from the CPU. This significantly increases I/O bandwidth and reduces CPU load. Just as with other operations, we are only interested in data that enters the replay domain, which in this case are DMA writes to memory. However, DMA's parallel operation to the CPU is problematic in this context. From the perspective of the deterministic replay, the DMA-performing device can be regarded as an additional processor that accesses a shared memory area (the common physical memory). That means we have to precisely record the order of parallel accesses to this shared area while the DMA transfer is in progress. Since we are replaying a virtual machine with virtual devices only, we are not dealing with true hardware DMA, but merely have a CPU thread in the hypervisor which performs a memory copy. For QEMU, this is one of the I/O threads that first retrieves the requested data (e.g., a particular disk sector) using OS services and then performs a `memcpy()` to its user-mode mapping of the guest physical memory. Recording the access order can be done by implementing a single-ownership protocol on the respective DMA target area using page protections. That means only the vCPU *or* the I/O thread has exclusive access to the memory area and if the I/O thread is interrupted by the vCPU, we record the thread's copy progress and switch ownership. Ownership changes again when the I/O attempts to continue copying[8].

[8] Such a protocol can to some degree also be implemented for hardware (R)DMA using IOMMU page faults, but requires device-specific handling since the actual data is dropped on page faults. This is problematic if the data cannot be retrieved again.

In normal operation, parallel accesses to a DMA destination are atypical and likely indicate a bug. Instead, the CPU generally waits for the device to signal completion. In the prototype for SimuBoost, we therefore simplify the aforementioned approach by omitting the access protections and only record what data has to be written where and when. We thereby regard DMA more like a transaction without supporting correct replay while the operation is still in progress.

Another problem is getting a consistent landmark for the DMA operation. This is because the I/O thread itself, not being part of the replay domain, does not possess a landmark. Instead, we have to express the completion time with a vCPU landmark. The vCPU, however, may concurrently run and constantly update registers. To get a consistent landmark, we therefore force the vCPU out of the guest into user mode and record the DMA operation in its context. We use QEMU's `run_on_cpu()` function for this purpose.

8.1.3 Evaluation

Although determining the accuracy of a replay seems simple at first, it is difficult to provide a definitive quantification. Whereas divergent instruction flows generally surface as mismatching landmarks that break the replay, more subtle differences can quickly be overlooked. This, for example, applies to small variations in memory that do not influence the instruction flow but may affect analysis results later on (e.g., when searching for redundant memory pages). Such differences often show up only for short periods of time, for instance on a thread stack. In order to guarantee absolute parity between recording and replay, it would be necessary to compare the system states at each instruction. This, in turn, imposes such an enormous run-time overhead that it eventually produces an entirely different execution, providing little information about the original run. Statements regarding the accuracy of a replay, especially in case of heterogeneous replay, must therefore be done with care.

During development, we utilized varying frequencies of checkpoints, memory checksums, and debug events to detect anomalies and refine the simulation (see § 8.2). However, in the final evaluation, we only measured the share of intervals that successfully complete, i.e., that do not diverge due to remaining inaccuracies in the simulator. For these intervals, we further compared the final memory image with the checkpoint of the next interval. On average over 97% of intervals run to completion (see Table B.5), while from this set of intervals around 98% of final memory images match the initial state of the next interval. Whereas these rates can be increased by investing more time in the refinement of the simulation, we consider this level sufficiently high to provide reliable results for the evaluation of SimuBoost in Chapter 9.

Even if the replay matches the original execution, recording itself affects the behavior of the workload to dome degree and causes differences to an unsupervised execution. However, just like with continuous checkpointing, it is difficult to measure the actual probe effect. We therefore again take the run-time overhead as one of the primary assessment criteria for evaluation, with the others being the log growth rate and the log compressibility, considering that the log must be included in the multicast data distribution.

> We do not include a worst-case microbenchmark such as stress-ram for checkpointing. Such a benchmark would simply cause every instruction to trap into the hypervisor (e.g., by only executing `RDTSC` instructions) and would exhibit a correspondingly high run-time overhead, but with little relevance in practice. We therefore restrict the evaluation to our set of real-world benchmarks.

Run-Time Overhead

Compared to an unmodified version of QEMU/KVM, we had to additionally trap the `RDTSC` instruction, extend the VM enter and VM exit paths for instruction counting, and install various hooks for event recording. In case of active recording, these hooks allocate memory, read the vCPU registers to create a landmark, and finally store the event together with its data. Otherwise, additional traps, instruction counting, and hooks are still present, but the hooks return early without performing any further operation. That means our modified KVM version incurs a certain overhead even without active recording.

Figure 8.6a summarizes the run-time overhead for recording all non-deterministic events listed in Tables B.1 and B.2 (including optional ones) compared to the execution of the same workload in an unmodified version of QEMU/KVM – i.e., no hooks, no additional traps, and no instruction counting. For 7 out of 10 workloads, recording incurs only negligible overhead, in most cases remaining below 1%. Exceptions to this are phpbench, SPECjbb, and apache, where the latter reacts very sensitive to recording with almost twice the run time.

To determine the source of the overhead, we repeat the comparison in Figure 8.6b, but take our modified version of QEMU/KVM as the baseline. That means we effectively remove any overhead from the observation that stems from the additional traps, the instruction counting, and the (disabled) hooks. This leaves us with the overhead for actual event recording. We can observe that the run-time overhead for the three aforementioned benchmarks is significantly lower. That indicates that the majority of overhead is bound to the added `RDTSC` trap and the instruction counting. Furthermore, as the other benchmarks do not suffer from comparably high run-time overhead, we can deduce that instruction counting is not the primary cost factor of the two. It merely amplifies the overhead for

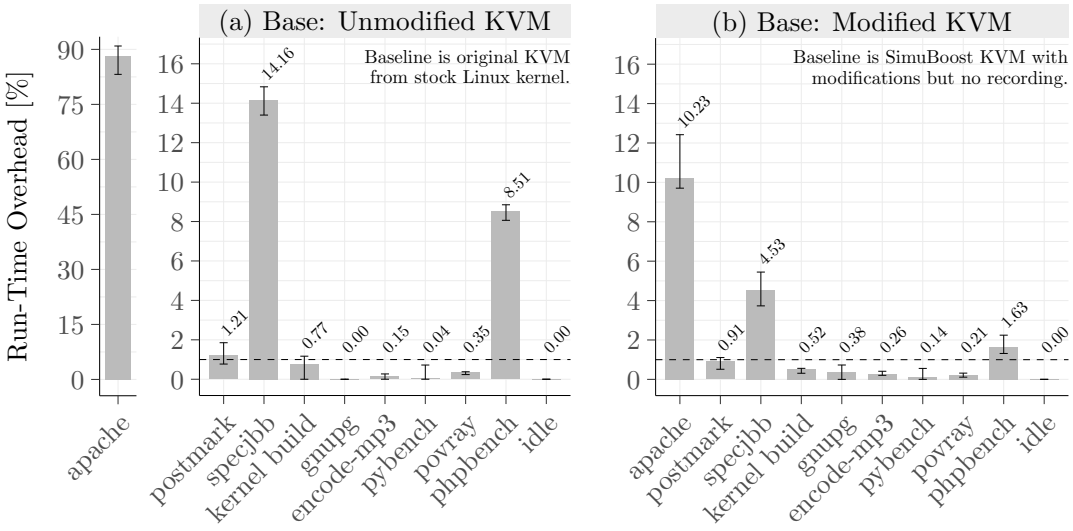


Figure 8.6: (a) Recording is very lightweight for most workloads, with the run-time overhead often remaining below 1%. However, some workloads, especially apache, react very sensitive. The baseline is an unmodified KVM. (b) The overhead decreases notably when comparing to the modified KVM, suggesting that our modifications cause the majority of overhead, not the actual event recording.

the additional switches between the hypervisor and the VM because each switch entails the configuration and reading of the corresponding performance counter.

We can conclude that the majority of the run-time overhead of our deterministic recording solution is caused by the RDTSC trap. This becomes especially visible for phpbench, SPECjbb, and apache, which show above-average rates for synchronous events (see Figure 8.7b) with over 98% of recorded events being timestamp readings (see Figure B.5a).

Log Growth and Log Compressibility

For the distribution with multicast, the log growth and log compressibility are important metrics as they determine the necessary network bandwidth for the recording logs. Figure 8.7b illustrates that for all workloads even uncompressed distribution over Gigabit Ethernet is conceptually viable – also for recording intensive workloads such as apache. However, under the constraint of parallel checkpoint distribution, compression is appropriate.

We employ the default compression method of the trace storage backend in Simutrace, which is a built-in LZMA [9] encoder. In our experiments, all logs show very good compressibility (see Figures 8.7c and 8.7d), reducing the average log growth to less than 100 KiB/s for most workloads. Only apache (3 MiB/s), encode-mp3 (1.3 MiB/s), and SPECjbb (800 KiB/s) exceed this value. Whereas in the case of apache and SPECjbb the sheer number of RDTSC readings is responsible

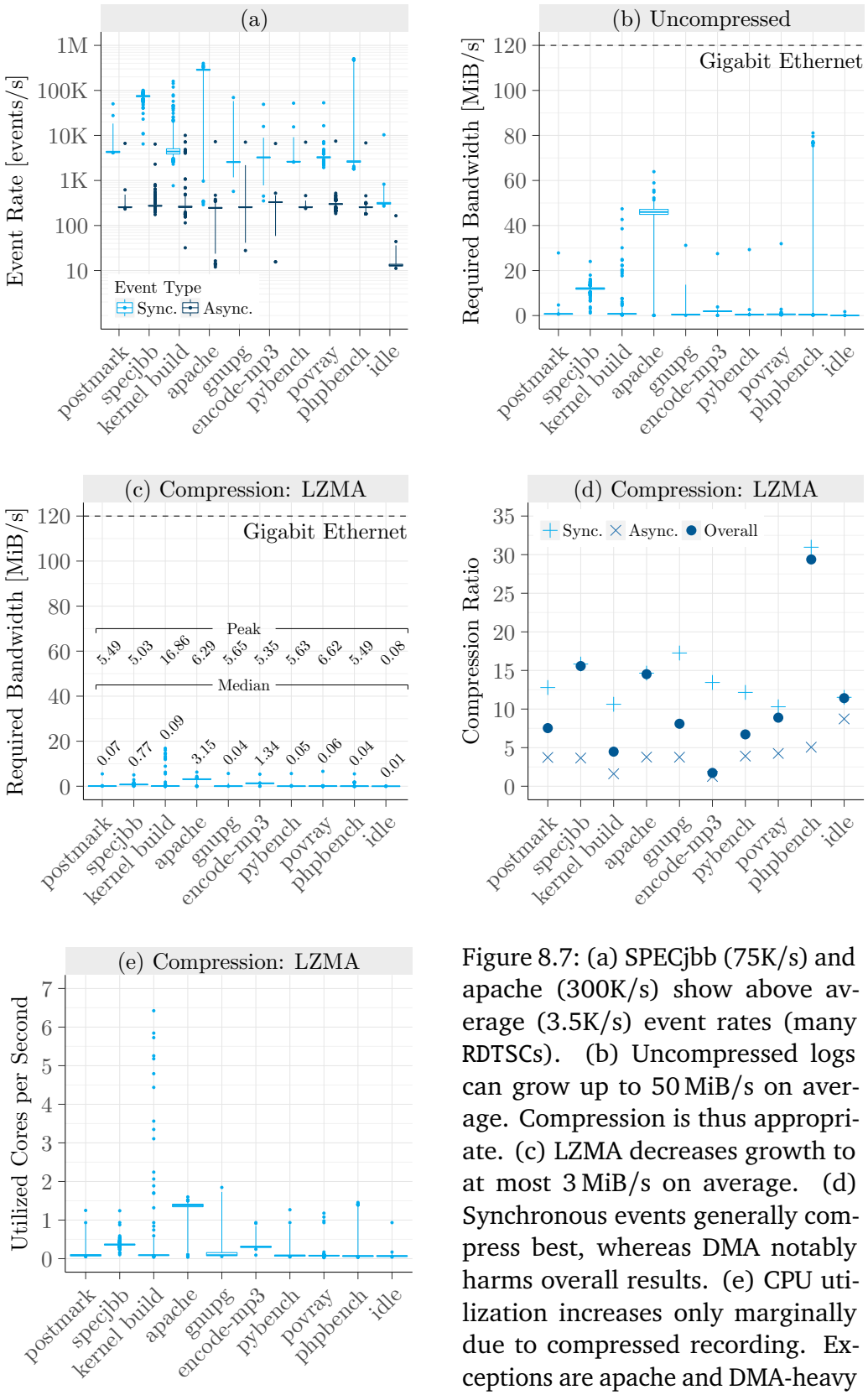


Figure 8.7: (a) SPECjbb (75K/s) and apache (300K/s) show above average (3.5K/s) event rates (many RDTSCs). (b) Uncompressed logs can grow up to 50 MiB/s on average. Compression is thus appropriate. (c) LZMA decreases growth to at most 3 MiB/s on average. (d) Synchronous events generally compress best, whereas DMA notably harms overall results. (e) CPU utilization increases only marginally due to compressed recording. Exceptions are apache and DMA-heavy phases in general.

for the increased data volume, the log for encode-mp3 comprises almost 70% DMA transactions (see Figure B.5b). Since DMA events include data read from secondary storage, their average compressibility is lower compared to other event types[9]. This is also clearly visible in Figure 8.7d, where the overall compression ratio tends to the lower ratio for asynchronous events (e.g., DMA) if the share of DMA in the log size is high. We can thus observe a clear correlation between increased DMA content and reduced overall compressibility. Nevertheless, even these logs can be transferred over network concurrently to the checkpoints of the respective workloads[10]. However, phases of increased DMA can reflect in short peaks. This is the case for the kernel build, where the outliers in Figure 8.7c are the manifestation of the Linux source code extraction at the beginning of the benchmark – i.e., the DMA events hold an already compressed archive.

The average CPU utilization due to the compressed recording is generally very low, remaining below 0.5 cores even for encode-mp3 and SPECjbb (see Figure 8.7e). Only apache consumes 1.5 additional cores on average. Although the aforementioned initialization phase of the kernel build has little impact on the average CPU utilization, the attempt to compress the already compressed DMA data temporarily (17 s) results in excessively high CPU utilization.

8.2 Simulation Refining

A major challenge with heterogeneous replay compared to homogeneous replay is that the simulation must be refined so that the *deterministic* parts behave exactly like in the hardware platform that was used for recording. In the following, we discuss some of the changes we made to QEMU to imitate our evaluation system.

Most of these (small) adaptations required hours of debugging to track down the code responsible for a single differing bit in memory (millions of instructions executed in between) or to find the spot in the instruction flow, where the replay started to diverge. In the latter case, we instrumented the VM enter and exit to generate debug events, delivering additional landmarks for state verification. Often, we had to further increase the frequency of landmarks with the help of the VMX preemption timer, which forces the CPU out of the guest after a certain number of CPU cycles. This way we could typically limit the search to less than a hundred instructions, albeit investigation remained tedious in code with many branches. For differences that showed up in memory only, we generally installed memory breakpoints, either in KVM or QEMU, depending on if QEMU implemented the modification in the first place. We can thus confirm previous work underlining the high development efforts for deterministic replay [50].

[9] This is also the case for encode-mp3, where the DMA events include the WAV input file. Compressing the raw audio with `gzip` leads to a 4% size reduction only.

[10] Compare Figure 7.4a on page 141.

8.2.1 Status Flag Computation

The x86 architecture stores status flags in the EFLAGS register, which is the lower 32-bit of the RFLAGS register on 64-bit platforms. The flags are manipulated by a total of 77 instructions [125], for example as a result of the CMP instruction or arithmetic computations. For 36 of the 77 instructions, at least one of the status bits is not defined, that is, the x86 specification does not include a calculation rule. As there is little use in working with undefined bits, programs generally do not access them and the instruction flow remains independent of their value. From this standpoint, undefined status flags are uncritical for deterministic replay. However, pushing the EFLAGS registers onto the stack using the PUSHF instruction can leak undefined values into memory. Linux, for example, typically backups the EFLAGS register this way before deactivating interrupts so that the register can be easily restored on exit. In consequence, when comparing the replayed contents of the guest physical memory to dumps from the original run, we have regularly seen diverging bits spread over the entire guest physical memory. In many cases, these turned out to be leaked undefined status flags that were differently implemented in QEMU. A few times this crashed the replay when the undefined flags ended up surfacing in one of the landmark registers. It is also straightforward for malware to deliberately exploit this divergence to undermine replay and eventually thwart inspection with SimuBoost. We therefore adapted the flag computation in QEMU to match our Xeon E5-2630 evaluation hardware.

To determine the computation rule, we wrote a small test application which fed the instructions with random values and corner cases and we inspected the actual status flags in the hardware's EFLAGS register. Table 8.1 lists some of the instructions for which we tested and/or changed the undefined flag computation in QEMU. We skipped 6 of the 36 instructions because they are not valid in 64-bit mode. In most cases, the auxiliary carry flag (AF), used in BCD arithmetic, is undefined in the specification but cleared by the hardware. This is also the default implementation in QEMU. For the bit scan instructions (BSF/BSR), QEMU uses the wrong operand to compute the parity flag (PF), which indicates an even number of 1-bits, so we modified it to use the first operand (OP1). For the shift operations,

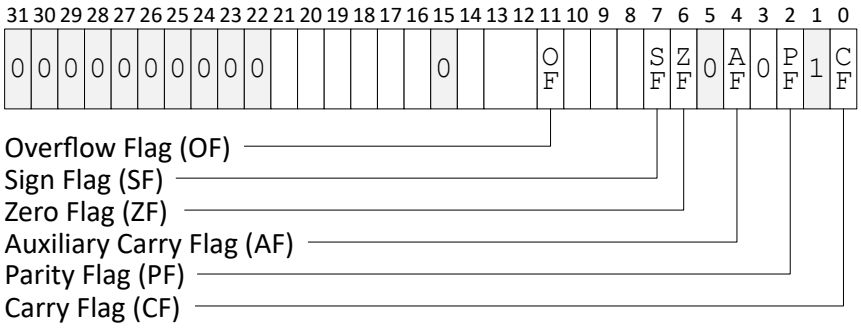


Figure 8.8: Status flags in the EFLAGS register [125].

Instruction	OF	SF	ZF	AF	PF	CF
AND				0		
ANDN				0	parity(DST)	
BSF/BSR	0	0†		0	parity(OP1)†	0
BZHI				0	0	
MUL/IMUL		sign(DST)	0†	0	parity(DST)	
OR/XOR				0		
SAL/SAR/SHL/SHR 1				0		
SAL/SAR/SHL/SHR cnt	OF(cnt=1)†			0		
TEST				0		

† Changed compared to default QEMU implementation.

Table 8.1: Undefined flag computation rules for selected x86 instructions. Blank fields denote defined values.

QEMU computes the overflow flag (OF), which indicates an overflow for signed integers, from the final shift result, whereas the hardware always sets the OF flag according to a shift by one. From a performance perspective, this change is most expensive because we have to perform two shift operations in software. In our prototype, we also compute the flag eagerly, even if it is not used.

8.2.2 Read-Write Instructions

When replaying the boot of a Linux kernel the simulation always crashed at a certain point. A first investigation revealed that the divergence happened in the guest's page fault handler in the course of an atomic compare-exchange (`CMPXCHG`). To aid debugging, we temporarily extended KVM to trap and log page fault exceptions. This eventually showed that the memory access mode between the simulation and the original execution differed. Whereas the simulation first performed a read and as a result caused a read page fault, the hardware raised a write page fault. We observed the same behavior for other instructions that both read and write memory; for instance `movs`, which copies a word directly in memory. We assume that the hardware first does a write probe as part of gaining exclusive access to the target cache line. As this can be omitted in software, QEMU does not integrate such probes.

Replicating x86 exception behavior is very complex in general and difficult to implement correctly in software. Hence, the authors of V2E decided to remove exceptions from QEMU and instead record and replay them [287]. While this simplifies simulation and makes replay more robust, it also means that guest page faults always have to trap in the hypervisor so they can be recorded. This takes away a great part of the performance that is gained from second level address translation and is a step back toward shadow page tables. We therefore decided

against this approach and adapted the simulation instead. Due to parallelization, we can tolerate run-time overhead in this phase, but not in the serial hardware-assisted virtualization run. In total, we extended 28 instructions or variants thereof that read *and* write to memory with a prior write probe in TCG:

- `adc`, `add`, `sbb`, `sub`, `xadd`
- `dec`, `inc`, `neg`

Integer Arithmetic

- `and`, `or`, `xor`, `not`
- `btr`, `bts`, `btc`
- `rcl`, `rcr`, `shr`, `shl`, `sar`, `sal`, `shrd`, `shrl`

Bit Arithmetic

- `xchg`, `cmpxchg`, `cmpxchg8b`, `cmpxchg16b`
- `movs`

Data Transfer

8.2.3 MMU-induced Non-Determinism

With second level address translation (SLAT) enabled, the host's MMU transparently translates guest virtual memory addresses to host physical memory using the guest-controlled page table and the EPT. To speed up subsequent accesses, the MMU caches translations in the TLB. As the different access times to cached and non-cached data can be measured in the guest, the TLB is not fully transparent. That means exits to the hypervisor or run time in other VMs can to some degree become visible through eviction of TLB entries. Nonetheless, from the perspective of deterministic replay, this is uncritical because the simulation accurately replays all time measurements, pretending that the same amount of time has passed. Although the TLB thus seemed unproblematic at first, we eventually found that it is not transparent to deterministic replay due to the following reason:

Whenever the MMU uses a PTE for address translation, it sets the PTE's accessed and dirty (A/D) bits accordingly. If, however, a matching translation is already present in the TLB, the MMU does not consult the page table and leaves the A/D-bits untouched. This becomes a severe problem for replay when the guest resets these bits in memory without also invalidating corresponding TLB entries – i.e., when the TLB and the guest page tables fall out of synchronization. In this case, the A/D-bits will only be updated on the next access if the TLB entries have been evicted in the meantime; either on purpose or as part of a replacement. The latter, in turn, depends on the size, architecture, and replacement policy of the TLB (all being CPU specific) as well as on the actual instruction flow and the memory accesses that accompany it.

As the A/D-bits are often used to estimate working sets and implement page replacement in operating systems, they directly affect the instruction flow. We observed this in the page aging algorithm used in Linux since version 3.16 [183]. When the kernel clears the accessed bit in PTEs, it does not invalidate the corre-

sponding TLB entries in order to avoid the potential performance hit resulting from a TLB miss. Instead, it tolerates inconsistencies between the TLB and the page table because the accessed bits only function as a loose page usage heuristic and the chance of a false prediction is considered "relatively low" by the maintainers [154]. In consequence, we had replays regularly crash when memory pressure in the VM started to rise and Linux invoked its page aging algorithm. We thus come to the conclusion that as soon as resetting A/D-bits and invalidating TLB entries is not synchronized, we see MMU-induced non-determinism that needs to be captured for replay.

Surprisingly, this source of non-determinism has not been widely discussed in the research. It is even more astonishing, considering that the problem is equally relevant to the more broadly researched homogeneous full system replay. In fact, we only found Bressoud and Schneider to describe a vaguely similar situation. In their case, the non-deterministic replacement in the software-controlled TLB on HP 9000/720 processors could lead to non-deterministic invocations of the guest's TLB miss handler [45]. Although we did not explicitly validate this, we assume that the commercial replay technology used in VMware products correctly handles MMU-induced non-determinism. This is because VMware published a patent (US9928180B2 [166]) that proposes a hardware extension to prevent the non-determinism by stale TLBs. In contrast to the research community, VMware engineers thus seem to be well aware of this problem.

For SimuBoost, we set out to quantify the overhead of feasible software alternatives. Our goal is to guarantee a faithful replay even in the presence of stale TLB entries. Since we cannot replicate the non-deterministic behavior of the TLB in the simulation, we are left with two options for tracking the state of A/D-bits in the hardware-assisted virtualization [183]:

1. We can prohibit the MMU write access to the guest page tables by enabling write protection for the corresponding guest physical pages in the EPT. Whenever the MMU then attempts to set bits in the guest page table, we receive a page fault in the hypervisor and can log the operation.

2. Alternatively, we prohibit the guest the read access to its own page tables. This way, we get a page fault when the guest tries to read the A/D-bits and we are able to log them before passing them on. This permits diverging A/D-bits in the page tables when replaying, but ensures that such differences are not relevant to the instruction flow as they are overridden by the logged state if necessary.

While the second approach would create a minimal log, disallowing the guest read access to the page tables also prohibits read access for the MMU. We thereby lose SLAT and would have to resort to shadow page tables. We therefore decided against it and evaluated the first approach instead [183].

In order to write-protect the guest page tables in the EPT, we have to identify the corresponding guest physical pages. We do this by trapping (guest) `CR3` changes, which indicates that a new page table hierarchy is to be configured. We then traverse this hierarchy to discover all involved guest physical pages. It is further necessary to track subsequent modifications to the active hierarchy such as the addition or removal of page tables. This is straightforward having the write protection established. For simplicity, we discard any information on previous hierarchies, but lazily keep the write protection until the next page fault.

The following step is to discern (deterministic) guest writes from (non-deterministic) MMU writes on an EPT violation. The first may entail changes in the guest page table hierarchy, whereas the latter needs to be recorded in the replay log. Fortunately, the VM exit qualification lets us infer this information.

In both cases, we finally have to determine the intended modification. We tried different approaches such as single-stepping but eventually settled on using KVM's instruction emulator being the fastest option. This way, we route offending instructions through the software page table walker where we can log modifications to A/D-bits. In addition, we can inspect modified PTEs for page table hierarchy changes before they become effective – i.e., before entering the virtual machine again. By recording the setting of A/D-bits as asynchronous events, we can use our existing infrastructure to precisely replay these operations in the simulation.

To evaluate if the approach works as anticipated, we booted a Linux VM with only 256 MiB of guest physical memory. This leads the kernel into activating the page aging algorithm and quickly creates stale TLB entries. Without recording MMU-induced non-determinism, the replay always failed to complete. This is because either diverging PTEs were loaded into CPU registers and resulted in fatal landmark mismatches, or the instruction flow diverged altogether. Tracking and replaying A/D-bits successfully avoids this problem. However, the recording comes at a considerable cost. We measured an additional run-time overhead of 51% over regular recording for a kernel build, 130% for apache, and 346% for a worst-case microbenchmark. Only for sqlite and a CPU stress test, the run-time overhead compared to regular recording was negligible. The overall log size increased between 30% and 100%. The replay time remained within a 1% to 3% increase.

We can thus conclude that, albeit being effective, recording MMU-induced non-determinism is an expensive operation. Considering that performance improvement for SLAT over shadow page tables has been reported with 38% for compiling the Apache server (compares best to the kernel build) [39], the question arises if, after all, resorting to the second approach is the more efficient solution. An alternative could also be to instrument guest page table reads using paravirtualization. However, this solution is not adequate for malware inspection.

Since MMU-induced non-determinism does not surface as long as the guest does not produce stale TLB entries, we regard tracking of A/D-bits as an optional feature that has to be activated for malicious and otherwise "misbehaving" guests only. We moreover see that further research is necessary to reduce the run-time overhead[11]. For the general evaluation of SimuBoost, we therefore did not integrate tracking of A/D-bits into the final prototype. Instead, we run the test VMs with 4 GiB of RAM, where we rarely observe a replay failing due to MMU-induced non-determinism. Nevertheless, the optimal solution would be to have a hardware extension in future processors as proposed by VMware [166].

8.2.4 Atomic Instructions (ARM only)

> The following issue has been observed on an Odroid-XU3 single-board computer, featuring a Samsung Exynos 5422 processor with four low-power Cortex-A7 and four high-performance Cortex-A15 cores. The (single-core) virtual machine for recording was pinned to one of the Cortex-A15 cores.

When experimenting with deterministic replay on the ARMv7 architecture, the replay of a Linux kernel boot always crashed at a certain point [262]. In almost all cases, the instruction count in the simulation was off by five instructions compared to the original recording in KVM. A comparison of the actual instruction stream during native execution and simulation revealed the cause (see Figure 8.9):

The code from `0x80132710` to `0x80132720` represents a typical use of exclusive instructions, for example, to implement synchronization primitives: load a value, modify it, attempt to write it back, check for success, retry if necessary (five instructions). The excerpt shown in the figure implements an atomic increment of the value at the address located in register `r3`. However, the store exclusive (`STREX`), which should write back the incremented value, fails during recording for no apparent reason and the whole operation is retried. The second attempt finally succeeds. In the simulation, on the other hand, the store exclusive does not fail, leading to a fatal divergence in the instruction count and preventing the replay of the next asynchronous event at `0x804630ec`.

The ARM Architecture Reference Manual states this to be valid behavior [32], even with no concurrent access to the locked memory address. The specification only dictates that progress has to be made at some point. For deterministic replay, this constitutes a major problem because `STREX` cannot be trapped or its result recorded. Furthermore, the instruction is not privileged and thus may also be freely used in user-mode code. This leaves even approaches based on paravirtualization unreliable. Up to this point, we are not aware of any generic solution to this problem.

[11] It should also be investigated if the issue could be used as a covert channel between VMs running on the same processor.

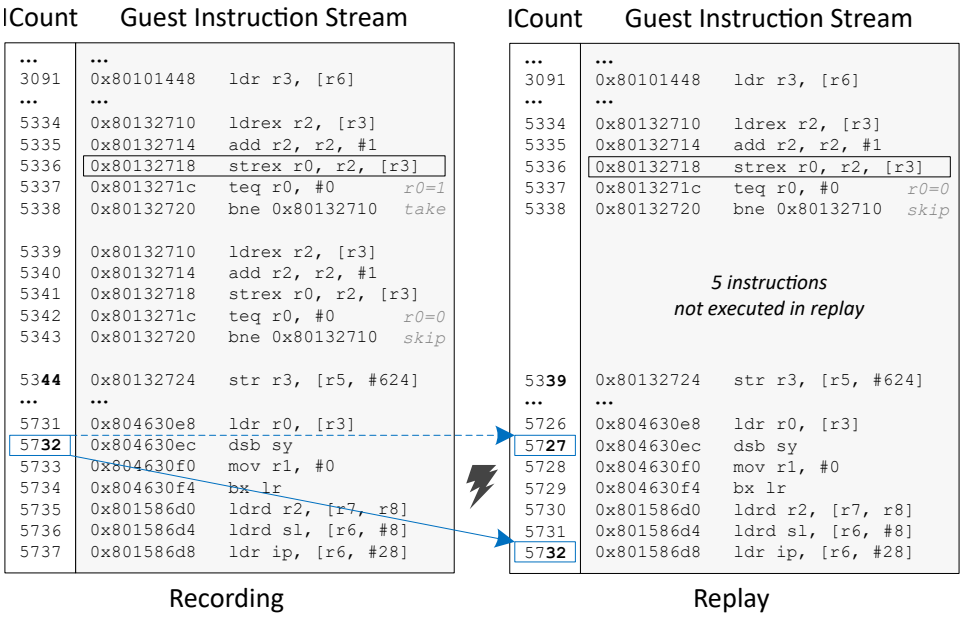


Figure 8.9: The store exclusive during recording fails for no apparent reason, causing a retry of the operation which is not present in the simulation (figure based on [262]).

Since most of the use cases for exclusive instructions in the Linux kernel follow a similar pattern to the presented excerpt, the difference in the instruction count is usually five instructions or multiples thereof – in case of further retries or more than one instance between consecutive landmarks. To some degree, adjusting the instruction counter as detailed in § 8.1.1 can compensate for the divergence. Nevertheless, this is not perfectly reliable, does require a rather large single-stepping window[12] and can still be exploited by malicious code to crash the replay. For the purpose of deterministic replay, we therefore would like to see the capability to trap STREX instructions. It may be possible to imitate this mechanism with the help of performance counters[13], but this remains open until future research.

8.2.5 Miscellaneous

In addition to the aforementioned issues, we had to perform numerous other changes to the simulation in QEMU in order to faithfully imitate the hardware behavior of our test platform. In the following, we describe three of the more prominent changes.

[12] The most common distance between recorded and replayed instruction counts were either 5 or 30 instructions [262].

[13] The Cortex-A15 supports the *Exclusive Instruction Speculatively Executed* – STREX *pass/fail* events.

Page Faults on Code Pages Besides faithfully reproducing page faults on data accesses, the simulation also has to accurately trigger page faults for code pages. In the case of QEMU, the binary translator reads guest code when it generates a translation block. This creates a fundamentally different access pattern compared to a physical CPU:

- A TB typically comprises multiple instructions but does not cross jumps.
- The translator does not perform speculative execution.
- Previously translated guest code is accessed again only when the respective TB has been removed from the code cache.

To prevent early page faults due to block translation, we adjusted QEMU to stop translation at page boundaries. Fortunately, we did not observe or find corresponding statements in the architecture manual that speculative execution triggers page faults. Our solution is thus consistent with existing heterogeneous replay systems [287]. The last point is uncritical because the physical CPU raises a page fault on a previously faulted code page only if the corresponding mapping is invalidated or changed in the meantime. These are cases that a binary translator has to cope with anyways. The last point thus does not constitute a problem for deterministic replay.

Resume Flag Bit 16 in the `EFLAGS` register is the so-called resume flag (RF), which controls whether the CPU stops at an instruction breakpoint [125]. The flag is intended to prevent recurrence of a breakpoint when a debugger continues execution. The CPU clears the RF flag after every instruction and sets it by pushing a corresponding `EFLAGS` value onto the stack when calling an event handler such as the breakpoint handler. This way, the set resume flag is popped from the stack on return. For interrupt handlers, the CPU sets the resume flag only if the interrupt arrives after any iteration of a repeated (`REP`) string instruction but the last iteration.

QEMU does not faithfully implement the resume flag, which leads to divergences in memory when the flag is pushed onto the stack. Furthermore, we observed that the physical CPU did not show fully deterministic RF states for interrupts during the last iteration of `REP`-prefixed instructions. We therefore replay the resume flag from the landmark in case of interrupts and replicate the behavior described in the architecture manual in all other cases.

Floating Point Unit (FPU) Besides the already mentioned reasons for diverging memory images, we found that many differences later in the system boot phase (i.e., starting of user-mode services) stemmed from an incomplete FPU implementation. This became apparent whenever the state of the FPU was written to memory using the `FSAVE` or `XSAVE` instructions, for instance on a context switch.

Deficiencies included:

- Missing reflection of FPU exceptions in the SSE status register (MXCSR).
- Missing implementation of the last FPU instruction pointer (FIP), the last FPU data pointer (FDP), and the last FPU instruction opcode (FOP) registers.
- Missing initialization of reserved words in the x87 FPU state and XSAVE areas.

We fixed these issues by extending QEMU and do not have to record any supplemental information.

8.3 Conclusion

Bootstrapping simulations based on periodic checkpoints alone does not reproduce the exact execution of the hardware-assisted virtualization. Instead, this requires recording and replay of non-deterministic events. In SimuBoost, we implemented a heterogeneous deterministic replay that collects events during hardware-assisted execution in KVM and precisely replays these events in QEMU's binary translator for simulation. To guarantee identical runs, we have to capture at least nine types of events, six synchronous (e.g., CPUID, RDTSC, IN) and three asynchronous (INT, SMI, Write DMA). Although strictly necessary only for asynchronous events, we also capture landmarks for synchronous events, which greatly simplifies debugging. The landmarks are built around the retired instruction count and the RCX register to differentiate individual iterations of repeated instructions (REP). Since, however, the hardware performance counter for instruction counting is not fully reliable on x86, the replay matches supplementary CPU registers in a window around the alleged target instruction to recognize the correct injection time.

A particular challenge with heterogeneous deterministic replay is that besides exact handling of non-deterministic events, the simulation needs to be refined to match the recorded hardware platform also in the execution of deterministic operations. In this course, we adapted QEMU's status flag computation, added memory write probing, and many more.

Interestingly, we found MMU-induced non-determinism to be entirely ignored in research – in contrast to the commercial products from VMware that seem to handle it. Our results confirm that a software-based solution is feasible, but only with considerable run-time overhead. We thus support VMware's proposal for a corresponding hardware extension. As MMU-induced non-determinism does only surface in conjunction with stale TLB entries, we regard recording and replaying it as optional in order to protect against malicious guests. We did not include it in our final prototype.

The evaluation of our implementation revealed that trapping RDTSC instructions is responsible for most of the run-time overhead. In consequence, benchmarks making frequent use of the timestamp counter suffer from notable slowdown (up to 90% for apache). All other benchmarks are barely affected by recording and show run-time overheads below 1%. Compression with LZMA proved to be very effective with recording logs, reducing the necessary network bandwidth to only 3 MiB/s for the most demanding workload. However, the compression ratio strongly correlates with the share of DMA events in the log, which can result in higher bandwidth consumption during DMA-heavy phases – e.g., up to 16 MiB/s during the initialization phase of the kernel build benchmark. Nevertheless, even with parallel checkpoint distribution Gigabit Ethernet generally provides enough bandwidth. Our replay solution thus fulfills the requirements of SimuBoost.

Chapter 9

Evaluation

In the previous chapters, we have described the four building blocks of SimuBoost: (1) the performance model, (2) continuous checkpointing, (3) checkpoint distribution, and (4) heterogeneous deterministic replay. In this chapter, we bring all these components together and evaluate to what extent SimuBoost is able to accelerate functional full system simulation.

In particular, the evaluation covers:

- **Achievable parallel simulation time and speedup**
- **Scalability and efficiency with increasing number of simulation nodes**
- **Applicability of the performance model**

We start with a detailed description of our evaluation setup in Section 9.1. In the following Section 9.2, we show that SimuBoost is able to drastically reduce the slowdown of functional full system simulation and that it is able to maintain this performance even with heavyweight instrumentation enabled. Section 9.3 demonstrates that SimuBoost delivers scalability beyond the limits of a single physical machine and that this is the basis for high acceleration. We further elaborate on the factors that determine the parallelization efficiency. In Section 9.4, we compare the predictions of the performance model with the actual results from our practical experiments and discuss conceptional weaknesses of the current model. We conclude the results in Section 9.5.

9.1 Evaluation Setup

As mentioned in § 7.3, our final prototype of SimuBoost does not yet integrate multicast distribution. Although this is a central component to perform *immediate* parallel simulation, we can do a representative evaluation of SimuBoost without

the live distribution. For this purpose, we separate the checkpointing and recording phase from the parallel simulation and instead manually copy all data to the simulation nodes in between. Our evaluation thus misses potential effects of multicast transmission delays because all data is already present on the target nodes. The results without live distribution will nevertheless be very close to what can be expected for a complete prototype. In Chapters 7 and 8, we demonstrate that the compression pipeline is typically able to reduce the data volume so that it remains below the bandwidth limit of Gigabit Ethernet. That means with actual distribution the simulation would start at most with the delay that is necessary for the compression and transmission of the first checkpoint. This delay is around 2 s in our experiments. All following checkpoints usually remain below the bandwidth limit and would not cause significant further delays.

Accordingly, we split each experiment into three consecutive phases:

1. Execution in a hardware-assisted virtual machine with active continuous checkpointing (copy-on-write, pre-scan, sparse) and recording of non-deterministic events. All data is compressed live, hence incurring comparable overhead as with multicast distribution. However, instead of sending the data into the network, we store everything on local storage.

2. Manual distribution of checkpoints and replay data to all systems in the simulation cluster (machine by machine) using `rsync`. Based on the aforementioned reasoning, we do not include this phase in the results. As stated in § 7.3, using a network file system to avoid the copy phase is not an option.

3. Parallel simulation using timed job submission to mimic the gradual availability of new checkpoints. We log the exact timing of checkpoints in the hardware-assisted run and then reproduce this very sequence.

Figure 9.1 illustrates our main evaluation setup consisting of five SMP systems with 108 physical cores – i.e., nodes for simulation – in total (see Tables 9.2 and 9.1 for details on the hardware and software configurations). Whereas the systems 2 to 5 only perform simulations, System V/1 serves as host for both the hardware-assisted execution at the beginning and simulations later on. To coordinate the parallel simulations, we establish a global job queue that is shared between all machines over the network. We use Python's capability for synchronized access to remote objects for this purpose. In this case, this is a regular Python multiprocessing queue hosted on System V/1. Since only a single process per machine dequeues jobs (i.e., five in our setup), contention on the queue is not an issue. For larger setups, a more sophisticated job assignment with a cluster scheduler such as SLURM [290] is certainly appropriate.

To attain a thorough picture of SimuBoost's speedup characteristics and to be able to verify the predictions of the performance model, we conduct experiments for all combinations of the following parameters:

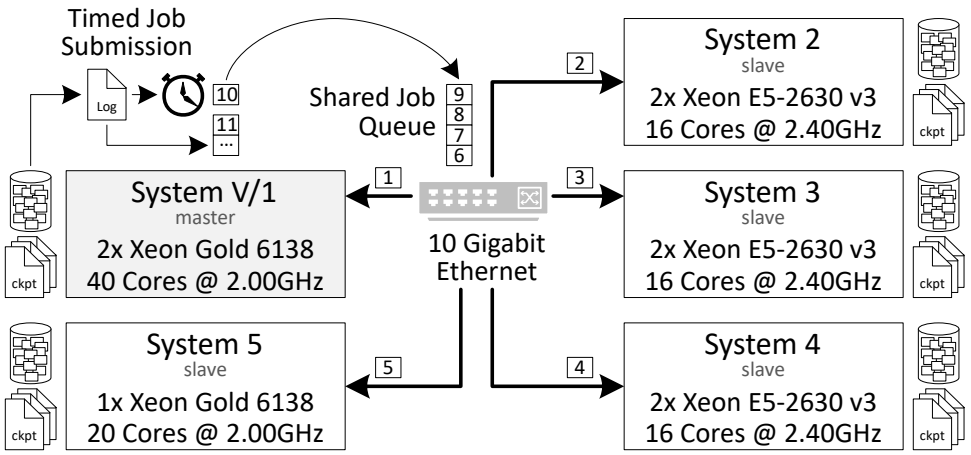


Figure 9.1: The hardware-assisted virtual machine runs on System V/1. Afterward, the checkpoints and replay data are copied to all other machines. Then, the benchmark framework starts parallel simulations that are coordinated with a shared job queue. A timed submission of jobs mimics the gradual availability of new checkpoints.

Workload In the previous chapters, we have found that the run-time overhead for continuous checkpointing and recording of non-deterministic events heavily differs between various benchmarks. As the run-time overhead defines a line below which decreasing the interval length does not lead to higher but lower speedup, we can expect considerable differences in maximum achievable speedup between workloads. We run applications from the same set of workloads as in the previous chapters.

Although we see truly interactive workloads such as user-driven desktop usage as an important workload category for operating system research, we do not explicitly include such a benchmark. In Chapter 6, we demonstrate that SimuBoost is able to maintain interactivity for real-world workloads with downtimes below 10 ms. The deterministic replay, in turn, injects all interactions into the simulation. Regarding interactive workloads, this is the decisive improvement of SimuBoost over conventional slow functional full system simulation, where it is impossible to faithfully capture and simulate realistic user behavior. As interactive workloads generally exhibit a disproportionate amount of idle phases, we include the *idle* workload for comparison. Depending on the applications run, a true interactive scenario will then be located between idle and one of the other workloads.

Number of Simulation Nodes Gradually increasing the number of nodes allows us to explore the scalability of SimuBoost. Although our simulation cluster provides a total of 216 logical cores due to hyperthreading, we only consider physical cores for simulation (i.e., 108). This is because preliminary experiments showed that running simulations on logical cores (i.e., two simulations per physical core) rarely increases performance. In fact, the opposite is often the case, which we

attribute to exceeding memory bandwidth limits and mutual cache pollution. This confirms results from Wallace and Hazelwood [267] who also did not observe further gain from using hyperthreads.

Accordingly, we repeat all parallel simulations with the following number of physical nodes: $N \in \{4, 8, 16, 24, 32, 48, 64, 80, 96, 108\}$. We use the same set of checkpoints and replay data for every configuration. In order to somewhat balance the load between the hosts, we assign each system a share of parallel jobs proportional to its number of physical cores.

Interval Length In order to find the optimal interval length for a certain number of nodes and to verify the predictions of the performance model, we run each benchmark configuration with different interval lengths: $L \in \{100, 300, 500, 1000, 2000, 3000, 4000\}$ms. We take the one with the highest speedup.

Simulation Slowdown While the simulation slowdown already varies between workloads due to each workload's individual instruction mix, we additionally run each simulation a second time with activated hooks for tracing memory writes (w), and a third time with tracing hooks for both memory reads and writes (r+w). This gives an impression on how speedup, scalability, and optimal interval length react to changes in simulation slowdown as it would be the case for different types of analyses connected to the simulation. Since collecting memory traces requires considerable computational resources (e.g., for compression) that we rather want to use for exploring larger simulation clusters, we do not perform actual tracing. Instead, we only activate the hooks. This adds a helper call to a tracing function for each memory access (depending on the tracing mode). Compared to actual tracing, the function only assembles the trace entry but leaves out submitting it. This slows down the simulation without taxing other CPU cores. We omit configurations with $N \in \{4, 8\}$ for the tracing runs as the higher slowdown makes such small setups less interesting.

For a more genuine comparison, we run conventional serial simulations without tracing with an unmodified version of TCG. Besides the tracing hooks, this version also misses the instruction counting and the refinements discussed in § 8.2. These are all features that further slow down the simulation. Both tracing runs, in contrast, add the hooks to our refined version of TCG.

We determine for each combination of workload and tracing mode the *best configuration* of L and N by gradually increasing N until a further increase does not provide any significant improvement in parallel simulation time but only harms efficiency[1]. Based on our results, we found a threshold of 10 percent points to be appropriate. If, for instance, selecting the next higher N only reduces the slowdown from 1.5x to 1.45x, we prefer the smaller setup with the higher slowdown but better efficiency.

[1] The ratio between the number of simulation nodes and the speedup over serial simulation.

9.1.1 Hardware and Software Configuration

The machines in our simulation cluster are connected with a 10 Gigabit Ethernet network in order to reduce the time for the copy of checkpoints and replay data. Since the transfer time is not taken into account, choosing 10 Gigabit Ethernet over Gigabit Ethernet does not affect the results.

In Chapter 8, we identified reading the timestamp counter (`RDTSC`) to be responsible for the majority of run-time overhead for event recording. When running multiple iterations of benchmarks that make heavy use of the timestamp counter such as apache, we first observed strong fluctuations in the run-time overhead between multiple iterations. Looking at the recording logs revealed that the guest Linux kernel sporadically changes its clock source from the timestamp counter (TSC) to the high performance event timer (HPET) or to the ACPI power management timer (PMT). Both the HPET and the PMT, however, do not just trap into KVM, but also require a round trip through the user-mode device emulation in QEMU, multiplying the overhead for each clock reading. Since this behavior heavily distorts benchmark results, we disable the HPET via QEMU's `—no—hpet` command line argument. We further adjust the fixed ACPI description table (FADT) in QEMU's ACPI implementation to not report the availability of an ACPI PMT. This can be done by setting the `PM_TMR_BLK` and `PM_TMR_LEN` fields to zero and is officially supported by ACPI [19, p. 130ff].

Software	Specification
Systems V/1 – 5	
Operating System	Ubuntu 16.04.6 LTS (64-bit)
Kernel (System V/1)	Linux 4.3.0 (KVM modified for SimuBoost)
Kernel (others)	Linux 4.4.0
QEMU	2.6.50 (modified for SimuBoost)
Simutrace	3.4.1
Guest VM	
Operating System	Ubuntu 16.04.5 LTS (64-bit)
Kernel	Linux 4.8.10[†]
Phoronix Test Suite	5.2.1[‡]
SPECjbb	2005
Java JRE	1.5.0_22
stress	1.0.4 (modified)

[†] KVM paravirtualization disabled to allow checkpoints to load in the binary translator.
[‡] See § 9.1.2 for the selected benchmarks and their respective versions. Configured to run one iteration only: `export FORCE_TIMES_TO_RUN=1`.

Table 9.1: Overview of Software Configuration

Component	Model & Specification
System V/1	
Primary evaluation system; used for all benchmarks in Chapters 6 to 8. Serves as host for the hardware-assisted virtualization as well as for parallel simulations.	
CPUs	2x Intel Xeon Gold 6138
Cores	40 (80 logical) @ 2.00 GHz†
Memory	128 GiB (8x16 GiB DDR4-2666)
Board	Supermicro X11DPi-NT
System Disk	Samsung SSD 960 EVO 256 GB
Data Disks	2x Samsung SSD 850 EVO 1 TB
Systems 2 – 4	
Used for parallel simulations only.	
CPUs	2x Intel Xeon E5-2630 v3
Cores	16 (32 logical) @ 2.40 GHz†
Memory	64 GiB (8x8 GiB DDR4-2133)
Board	Supermicro X10DRi
System Disk	Crucial SSD MX100 256 GB
Data Disk	Samsung SSD 850 EVO 1 TB
System 5	
Used for parallel simulations and runs the time-consuming serial simulations for comparison with SimuBoost.	
CPU	Intel Xeon Gold 6138
Cores	20 (40 logical) @ 2.00 GHz†
Memory	64 GiB (4x16 GiB DDR4-2666)
Board	Supermicro X11DPi-NT
System Disk	Samsung SSD 960 EVO 256 GB
Guest VM	
Benchmark virtual machine; executes all workloads. We checkpoint and replay this VM.	
CPU	P6 Pentium M compatible (`qemu64`)
Cores	1 (1 logical)
Ext. Features	64-bit long mode, NX, SSE3, CX16, MTRR CLFLUSH, huge pages
Memory	4 GiB
Board	Intel 440FX+PIIX3 chipset‡
System Disk	16 GB virtual disk file (`qcow2`)

† TurboBoost disabled for consistent performance between runs with few and many busy cores.
‡ High precision event timer (HPET) and ACPI power management timer (PMT) disabled to force more lightweight timestamp counter (TSC) as clock source.

Table 9.2: Overview of Hardware Configuration

9.1.2 Benchmark Scenarios

Throughout this thesis, we primarily use benchmarks from the Phoronix Test Suite [11], a popular automated open-source benchmark framework. This ensures reproducibility and embeds each application into the same environment. We have selected the benchmarks so that they execute a fixed amount of work (except idle and SPECjbb2005). This way we can determine the slowdown compared to hardware-assisted execution simply by measuring the run time. The version number identifies the exact Phoronix test profile.

pts/apache (v1.7.1) The apache benchmark starts a local instance of the Apache web server (version 2.4.7) and runs the `ab` tool to create a total of 1M requests, with 100 requests being carried out concurrently. The server responds with a static web page consisting of two files: a 3064 bytes HTML page containing a 4758 bytes PNG logo of the Phoronix Test Suite.

pts/build-linux-kernel (v1.6.0) The kernel build benchmark compiles and links the Linux kernel (version 4.3 - MD5:`737570130236c2256cfa67920fa721cf`) from `kernel.org` with `make defconfig`. The test starts by first extracting the source code from the compressed archive. Since we run a single-core VM only, the benchmark compiles one source file at a time (`-j1`).

pts/encode-mp3 (v1.7.1) This workload executes LAME 3.100 to encode a WAV rip of the 5-minute song *Demon Seed* by *Nine Inch Nails* to MP3 (`-h` option for high quality). The input file is 78.1 MiB in size. The encoded output is directly piped to `/dev/null`, thus creating no disk writes.

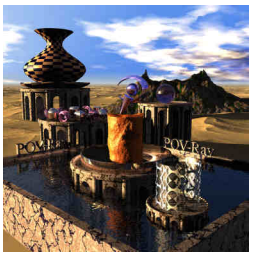
pts/gnupg (v2.4.0) The benchmark runs the GnuPG 1.4.22 cryptographic software suite to perform a symmetric AES128 encryption of a 2 GiB file, containing zeros only. The test dynamically creates the input file from `/dev/zero` using `dd`. As with encode-mp3, the output is directly dropped by sending it to `/dev/null`.

pts/phpbench (v1.1.5) The PHP benchmark measures the performance of the PHP interpreter by running 56 test cases which stress input parsing (e.g., removal of comments), language features (e.g., loops, assignments, typecasting), and various built-in functions such as `md5()` and `crc32()`. We increase the benchmark's run time by executing 10M iterations instead of the default 1M.

pts/postmark (v1.1.1) NetApp PostMark 1.5.1 simulates file operations similar to what a web or mail server would do. It first creates 500 files with random contents and size between 5 KiB and 512 KiB. Then, it executes 250K random transactions (e.g., read, append, delete). PostMark closes with removing all files.

pts/pybench (v1.1.2) The pybench workload performs 20 rounds of the Python Benchmark Suite 2.0 to tax the Python interpreter similarly to phpbench.

pts/povray (v1.2.1) A particularly challenging workload for simulation is POV-Ray 3.7.0.7. The ray tracer renders a 512x512 image of a standard 3D scene coming with the software. The benchmark heavily relies on FPU arithmetic, which in software is much slower than integer arithmetic. The povray benchmark thus experiences a tremendous slowdown, even with plain serial simulation. Hence, we skip the $N \in \{4, 8\}$ configurations.



Standard Scene

pts/sqlite (v2.0.1) The sqlite benchmark imitates a lightweight database workload that performs 7500 insertions into an SQLite 3.22 database. The final table has the format `[SMALLINT, TIMESTAMP, VARCHAR(4), VARCHAR(16)]`, thereby consuming around 200 KiB at 28 bytes per row.

idle The idle benchmark performs a 1-minute sleep. It is the optimal case in terms of checkpointing and recording overhead and allows the replay to skip most of its run time due to `HLT` instructions being ignored in replays. However, because its run time is predefined, the serial simulation also runs for only one minute, despite having simulated much less idle cycles.

Only used in previous chapters

stress-ram This benchmark serves as worst-case for the checkpointing component (not the compression pipeline) by allocating a 3 GiB buffer and touching it as fast as possible. It first runs over all pages and sets the first byte to `'Z'`. It then iterates a second time over the buffer, reading the character back in. We modified the underlying stress 1.0.4 application to allow the specification of a number of iterations rather than a target run time. We use 3000 iterations.

SPECjbb2005 This popular benchmark [14] evaluates the performance of server-side Java by emulating a client/server application that exercises the JVM, JIT compiler, garbage collection, and the operating system. The workload includes XML processing and large decimal computations. Just like the idle benchmark, SPECjbb runs for a predefined amount of time (i.e., 30 mins), irrespective of the virtualization technique. Instead, SPECjbb returns a number of transactions completed in this time frame. As we base our final evaluation on the run time of the workloads, we omit experiments with SPECjbb in this chapter.

9.2 Speedup

To get an impression of the speedup that we gain from SimuBoost, we compare the parallel simulation with a conventional serial simulation. In both cases, we give the slowdown relative to fast hardware-assisted virtualization with a regular *unmodified* KVM. When comparing SimuBoost with conventional serial simulation,

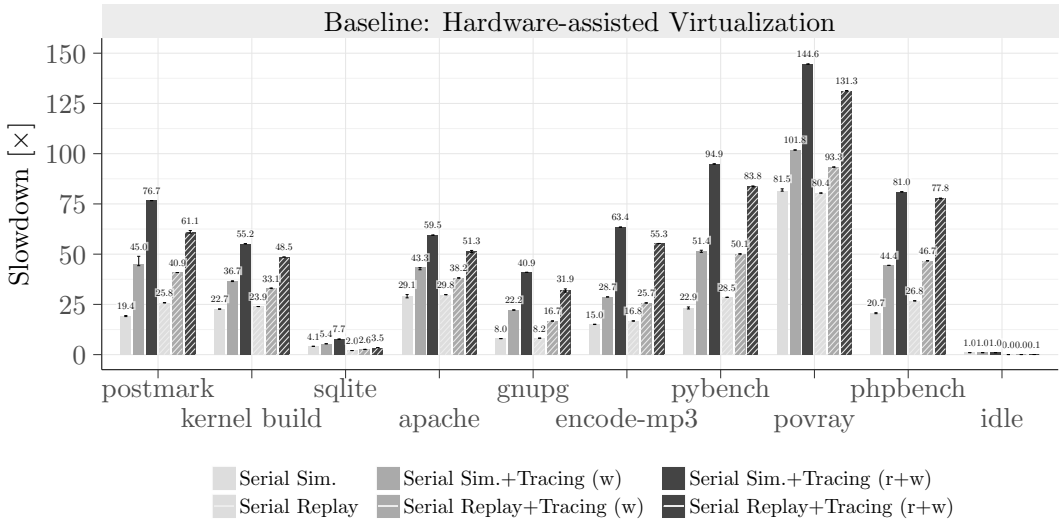


Figure 9.2: Without tracing, conventional simulation is typically faster as it does not include instruction counting and the refinements from § 8.2. With all refinements included and tracing enabled, replay is faster. The replay for the idle benchmark skips all halt cycles. The figure shows the replay of the best configuration (see Table B.6).

we have to bear in mind that a heterogeneous replay differs from a conventional simulation and that these differences may lead to varying slowdowns for the same combination of workload and degree of instrumentation. This is important as SimuBoost parallelizes a replay, not a conventional serial simulation, and we are using the slowdown as basis for our assessment. Before looking at the speedup of parallel simulation, we therefore first quantify the difference between *serial replay*[2] and conventional serial simulation.

> Detailed numbers for most results in this and the following section can be found in Table B.6. The table also contains additional information (e.g., N and L) on the selected configurations.

In Figure 9.2, we see that for both tracing iterations the slowdown of conventional simulation is between 8% and 25% higher than in the case of the serial replay. This stems from the fact that our conventional simulation does not employ a timing model, whereas the replay reproduces the recorded timing. With the overhead for instrumentation, it takes more time to simulate instructions. Albeit, the virtual timers in the conventional simulation keep firing at the same rate (relative to the wall-clock time), causing more kernel-mode code to be executed over the duration of the workload. Since all workloads process a fixed amount of work, the conventional simulation thus executes more instructions until the workload completes. A higher instrumentation overhead amplifies this effect.

[2]Replay without parallelization under the premise that event logs already exist. We sum the replay times of the 4 s intervals to get a total time even with potential replay failures.

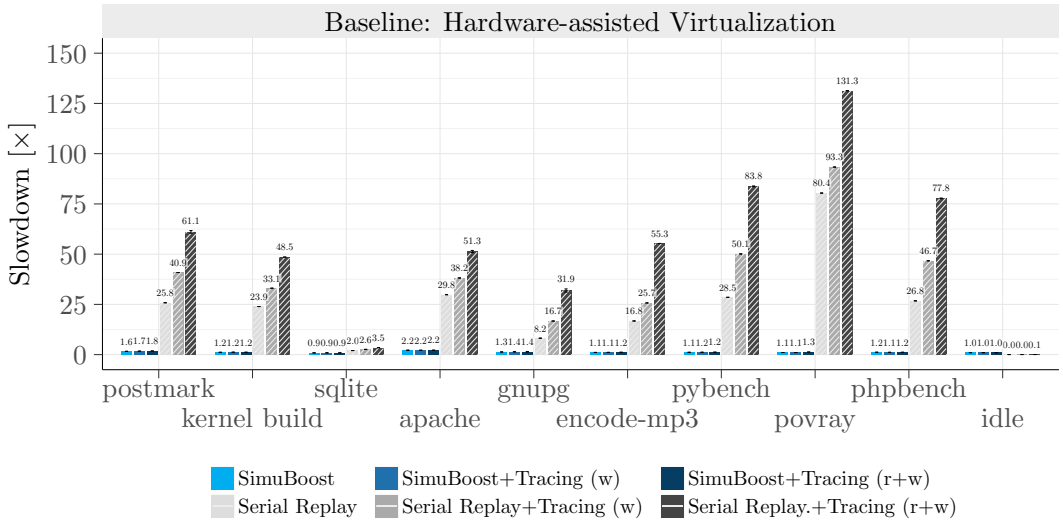


Figure 9.3: Parallel simulation with SimuBoost is much faster than conventional serial simulation. Baseline is a run with hardware-assisted virtualization using an unmodified KVM.

In case of plain simulation without tracing, the replay is between 2% and 33% slower because, as mentioned previously, the conventional simulation does not include the instruction counting and the refined TCG in this configuration.

In the following, we use the serial replay as reference for comparison as this is the actual simulation that is parallelized. Moreover, due to its higher representativeness of a timing-realistic execution irrespective of the instrumentation slowdown, the replay is typically also more interesting.

Figure 9.3 contrasts the slowdown of the serial replay with the drastically reduced slowdown attainable with a parallel simulation. SimuBoost is able to fully simulate a kernel build in just 16% more time ($N = 24$, $L = 2.0\,\mathrm{s}$) than required for an execution with fast hardware-assisted virtualization. The baseline is again an unmodified KVM. Even the povray benchmark with its high slowdown completes in only 6% more time ($N = 80$, $L = 0.5\,\mathrm{s}$). By choosing different configurations for N and L (typically higher N and shorter L), SimuBoost is able to deliver the same performance even in the tracing scenarios.

Nevertheless, the high run-time overhead for checkpointing and recording for some workloads reflects in a reduced speedup. This can be seen in the case of postmark, which possesses a 44% overhead in the best configuration ($N = 24$, $L = 0.5\,\mathrm{s}$) and therefore still shows a 63% slowdown with parallel simulation. The apache benchmark is even worse with a 102% overhead during the hardware-assisted virtualization and a final slowdown of 2.19x ($N = 16$, $L = 1.0\,\mathrm{s}$). From the evaluations in Chapters 6 and 8, we know that for postmark the run-time overhead is primarily caused by the checkpointing, whereas for apache the recording is the major cost factor.

These results underline the importance of fast checkpointing and recording technologies in this context. Furthermore, they illustrate that a higher speedup cannot be achieved by just using more nodes for simulation. In order to supply these additional nodes with work, we have to shorten the interval length. This increases the production rate of checkpoints but at the same time entails a higher run-time overhead. As described in Chapter 5 and as visible in the results for postmark and apache, this eventually leads to a higher parallel simulation time and less speedup. Only if the slowdown of the serial simulation increases, for example, due to higher instrumentation costs, the rise in run-time overhead for more frequent checkpointing is compensated.

The results for sqlite and idle represent special cases. As described in § 6.1.3, the sqlite benchmark completes faster with activated checkpointing due to the deactivation of transparent huge page support. This results in a slight speedup for parallel simulation compared to a run with an unmodified KVM (i.e., no checkpointing and active transparent huge page support). The idle benchmark, in turn, is slower with parallel simulation compared to a serial replay because for the serial replay we assume that event logs are already present. That allows the serial replay to skip the entire idle phase, completing the 60 s benchmark in just 2 s. In the case of the parallel simulation with SimuBoost, intervals are simulated individually and gradually as they are produced. This creates overhead for checkpoint loading and prevents the parallel simulation from skipping the entire idle phase at once. Instead, each simulation only skips the idle cycles covered by the corresponding interval. We therefore effectively get a slowdown compared to the serial replay. Nevertheless, the idle benchmark still completes 1 s faster than the reference execution with an unmodified KVM. This is due to saving the idle cycles in the last simulated interval.

9.3 Scalability and Efficiency

Depending on the characteristics of the scenario such as the workload's run time, the overhead for checkpointing and recording, and the slowdown for serial replay and instrumentation, SimuBoost is able to efficiently utilize a varying number of simulation nodes. Figure 9.4 depicts the attainable speedup over serial replay using an increasing number of simulation nodes. For each combination of workload, tracing mode, and N, the figure shows the configuration of L with the maximum speedup.

At first sight, the horizontal lines in the figure suggest low scalability. In fact, however, when the speedup stops rising and begins to progress horizontally, SimuBoost already removed most of the slowdown of the serial replay. In consequence, the breaks are located near the serial replay slowdown of the corresponding combination of workload and tracing mode. The difference is what we observe as

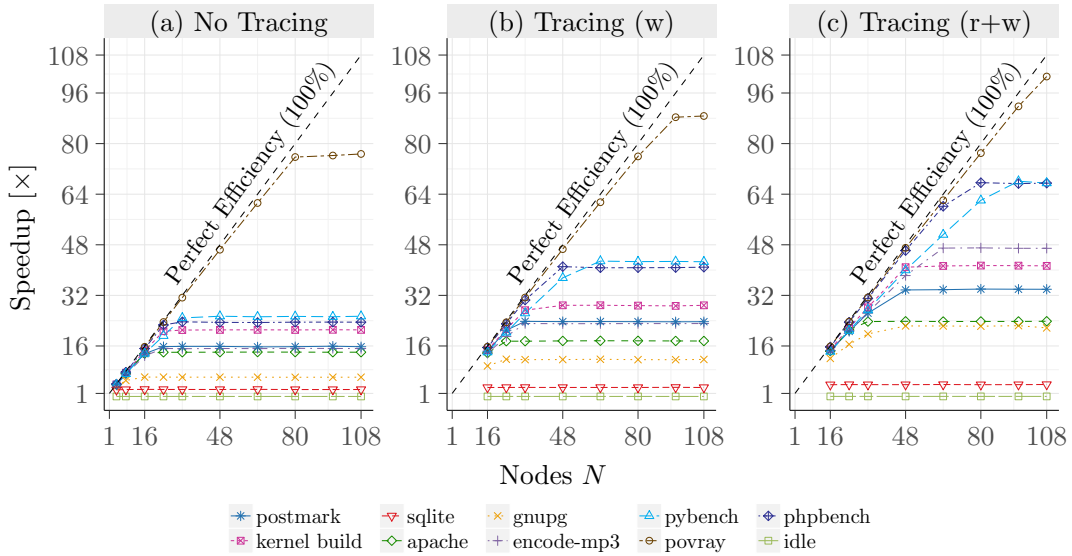


Figure 9.4: Breaks in scalability are located close to the serial replay slowdown of the respective workload and tracing mode combination (horizontal lines). The results for povray demonstrate that SimuBoost is able to efficiently scale beyond the limits of a single physical machine. The parallelization efficiency, however, differs between scenarios. N is the configured number of nodes, not the number of actually busy nodes (see below).

remaining slowdown as described in the last section. For instance, with tracing of writes enabled (w), we see a speedup of around 88x for povray. The remaining slowdown of the respective parallel simulation is 6% compared to a run with an unmodified KVM (Figure 9.3), which gives us the original slowdown for serial replay ($88 \cdot 1.06 \approx 93.3$).

Since the FPU-heavy computations performed by povray exhibit an exceeding simulation slowdown, SimuBoost utilizes more nodes for povray than for any other workload. With full memory tracing enabled, all 108 cores of our cluster are busy[3]. This demonstrates that SimuBoost can effectively scale beyond the capacity of a single physical machine.

A closely related metric in this context is the parallelization efficiency with which additional nodes can be leveraged for parallel simulation. With perfect efficiency, doubling the number of nodes results in double the speedup – at least to the point, where we reach the horizontal line and no further speedup can be made. Especially in Figure 9.4c, we can see that the parallelization of, for instance, pybench exhibits much lower efficiency than, for example, in the case of povray. The degree of efficiency depends on multiple factors that can be best understood graphically.

[3]The increase of the remaining slowdown from 6% to 30% (Figure 9.3) suggests that even more cores are required for maximum acceleration.

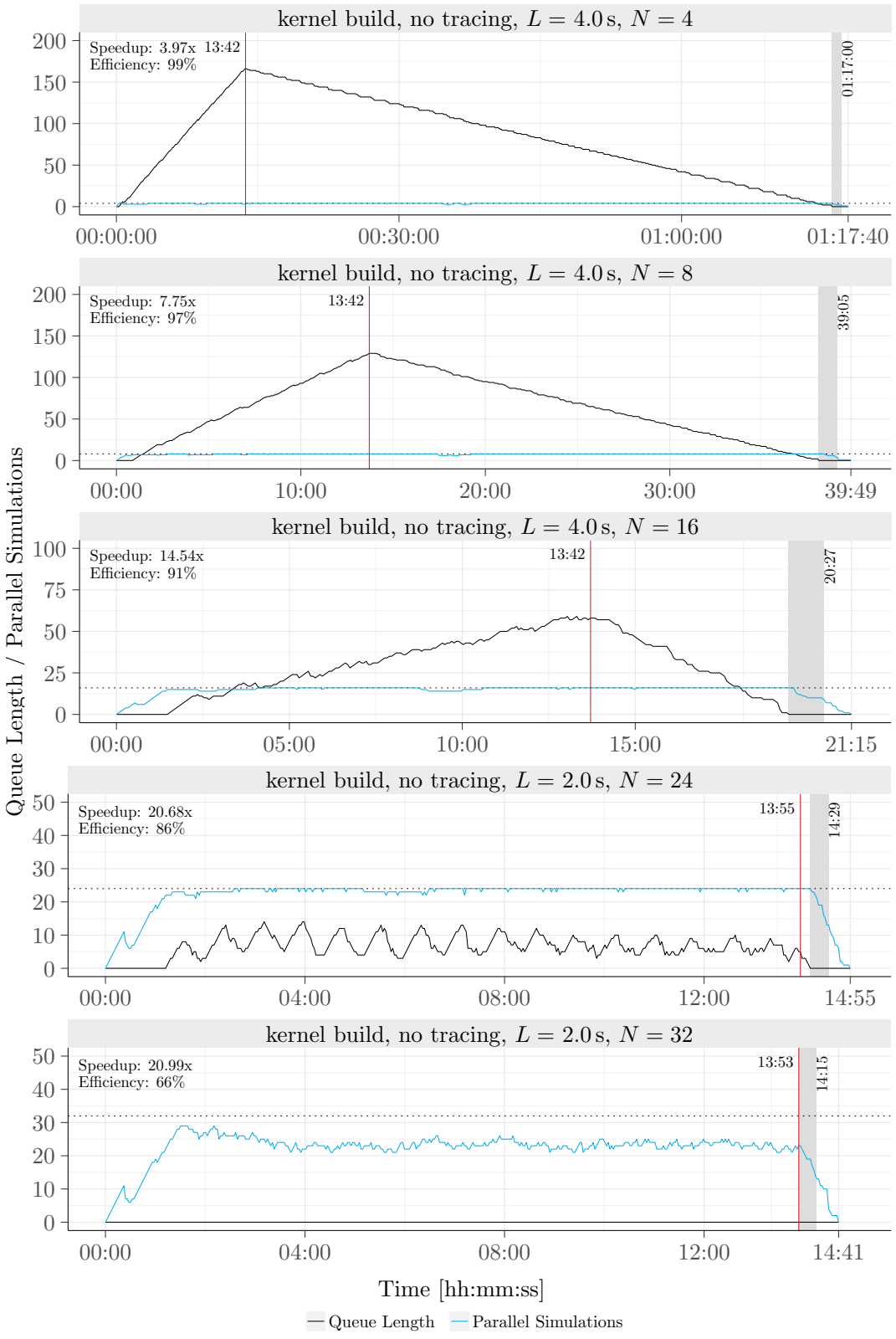


Figure 9.5: With too few nodes ($N < 24$), intervals need to be queued. With $N = 24$, production and consumption are in balance and utilization is still high. More nodes provide little benefit but notably reduce parallelization efficiency by just idling. The red line denotes the end of the virtualization run (i.e., end of production). The gray rectangle shows the simulation of the last produced interval. The dotted line marks the number of configured nodes N.

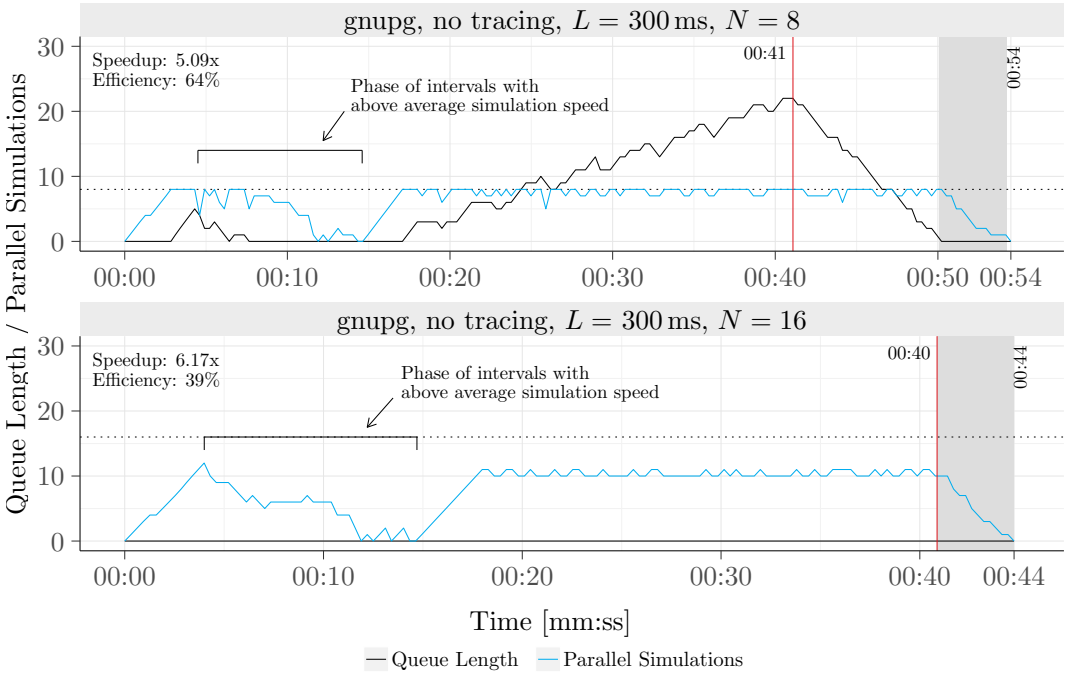


Figure 9.6: The intervals with above average simulation speed create a phase of under-utilization that reduces efficiency. See Figure 9.5 for a description of the graphical elements.

Figure 9.5 illustrates the node utilization and the length of the queue of outstanding intervals (i.e., checkpoints created but not yet simulated) for a set of parallel kernel build simulations with increasing number of nodes.

With only four nodes, we measure the lowest parallel simulation time with an interval length of 4 s. As the four nodes cannot keep up with the production rate, new checkpoints quickly accumulate and the queue length rises. It takes almost an hour until all intervals have been simulated. However, during this time all nodes are permanently busy. Increasing the number of nodes shortens the overall parallel simulation time and moves the end of the simulation closer to the end of the virtualization. The idle phases typical for SimuBoost can be clearly seen at the beginning and the end.

With $L = 2.0\,\text{s}$ and $N = 24$, the production and consumption rates are in perfect balance which reflects in the periodicity of the queue length. Except from the idle phases at the beginning and the end, all nodes are busy[4]. Further increasing the number of nodes to $N = 32$ provides only a marginal improvement in overall simulation time. As the consumption rate now exceeds the production rate, there is always a free node available and no interval gets ever queued. At this point, some of the simulation nodes remain permanently idle and the parallelization efficiency drops from 86% to 66%.

[4]Small fluctuations are caused by the job submission which our benchmark environment does not count as busy phase. Furthermore, it does not account simulation time for intervals that failed to replay.

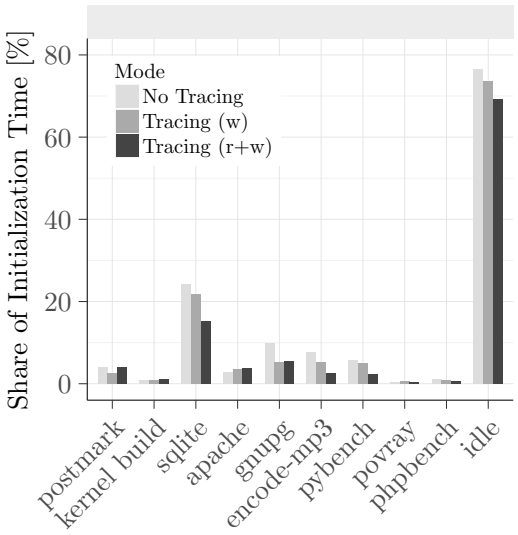


Figure 9.7: The initialization time is typically less than 10% of the busy time.

Idle Time We can conclude that everything creating space between the dotted line (the number of configured nodes) and the blue line (the number of busy nodes) entails a reduction in efficiency by causing idle time. Besides a general oversupply of nodes, this can also be the result of changing program behavior. We can observe this in particular with gnupg (see Figures 9.6 and B.7). The majority of simulations in the first 15 s complete on average in 1.2 s, all following simulations, in contrast, require on average 3.0 s. Whereas eight nodes are more than sufficient for achieving optimal speedup in the first phase, the cluster is clearly overloaded in the second phase (queue length rises). The growing backlog forces us to use more than eight nodes, otherwise we do not attain the overall minimum parallel simulation time. However, this creates (even more) idle time in the first program phase and notably reduces the efficiency.

Another factor lowering the efficiency is the share of busy time that is *not* spent on actual simulation. This factor consists of three components:

Initialization Time This is the time consumed for the initialization of the individual parallel simulations. Figure 9.7b shows that for our set of workloads (except sqlite and idle), the initialization accounts for less than 10% of the total busy time. The share shrinks with increasing simulation slowdown[5], but grows with the number of intervals (i.e., shorter intervals) and per-interval initialization cost, the latter being dominated by the checkpoint loading time (see § 6.3).

Per-Interval Simulation Overhead Figure 9.8a illustrates that with a fixed number of nodes ($N = 16$), the sum of the simulation time over all intervals increases with smaller interval lengths. This indicates that besides the explicit initialization, we have to pay further per-interval costs. We assume that most of this overhead

[5]Hence the high values for sqlite (2x - 3.5x) and idle (<1x), where the slowdowns are very low.

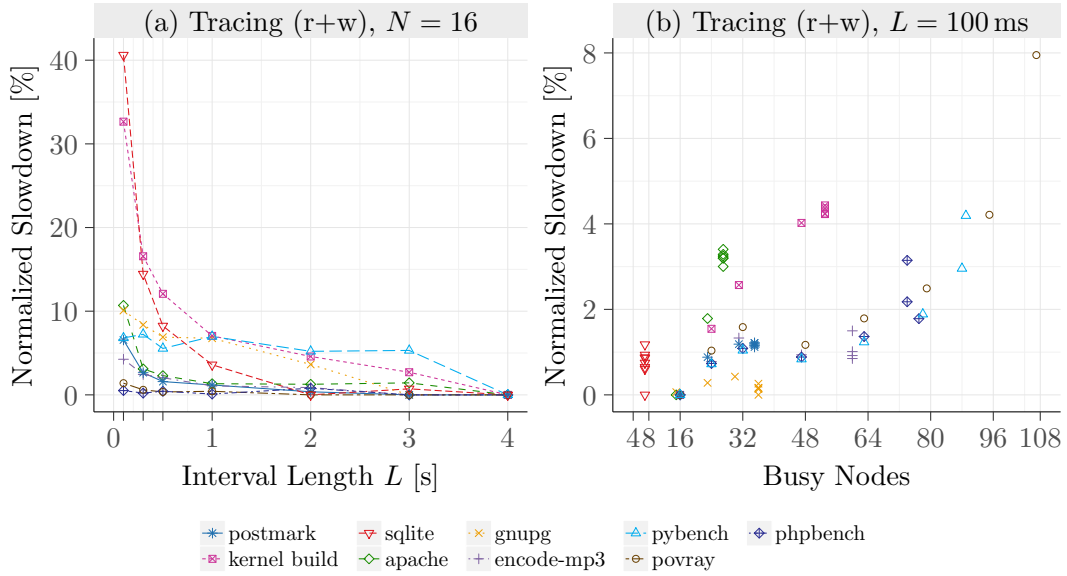


Figure 9.8: (a) Short intervals lead to higher accumulated simulation time. The graph shows the slowdown normalized to a simulation with 4 s intervals. (b) With increasing number of utilized nodes, simulations run slower due to the pressure on shared computing resources such as the memory bus and caches.

stems from the dynamic binary translation, that is, the actual code generation. Since each individual simulation starts with an empty code cache, shorter intervals increase the overall translation effort.

SMP Load Running a multitude of simulations in parallel on the same SMP system negatively affects the simulation speed. This relationship can be seen in Figure 9.8b. Especially povray, which is able to fully utilize our simulation cluster, suffers from this effect when going from 96 to 108 busy nodes. We attribute this to reaching limits in the memory bandwidth and the cache capacity. This is not surprising, considering that SimuBoost runs 40 simulations (i.e., virtual machines) in this configuration in parallel on our primary system alone.

Figure 9.7a depicts the efficiency for the various benchmark scenarios based on the median number of actually busy nodes, not the configured number of (potentially idle) nodes. This way, we remove surplus nodes and get an indication of the true efficiency values for parallelization despite the limited set of node numbers examined. Due to the described idle phase in gnupg, the efficiency is comparably low with only 62%. The efficiency for the idle benchmark is much worse (4% to 6%) as the speedup (the basis for calculating the efficiency) is effectively a slowdown compared to the serial replay. We can thus expect interactive workloads (e.g., user-driven desktop usage) to generally exhibit a rather low efficiency, depending on the share of idle cycles. The other measured scenarios have a parallelization efficiency between 72% (sqlite) and 95% (povray).

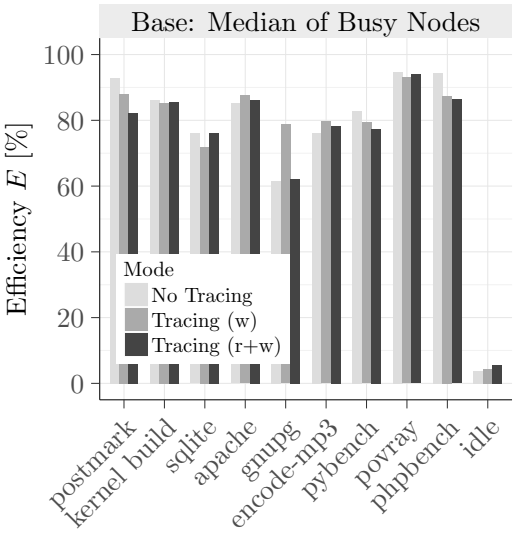


Figure 9.9: Measured parallelization efficiency based on the number of actually busy nodes is between 62% and 95%, except for idle (4% to 6%).

From the findings of this section, we can infer the following characteristics:

- Idle phases in the cluster (e.g., due to changing program behavior) have the most negative effect on the efficiency.
- Reducing the interval length is likely to decrease efficiency, especially for workloads with low code and memory access locality (high translation costs and checkpoint loading time).
- A high serial simulation slowdown originating from the instruction mix or instrumentation is likely to improve efficiency.

Remember that, in accordance with our performance model, the given results for the efficiency relate to the parallel simulation only. They can be used to weigh a reasonable size for the simulation cluster. As outlined in Chapters 6 and 7, additional CPU cores are needed for the compression of the checkpoints and the event log. For the measured optimal interval lengths and the workloads examined in our final evaluation, we have to provide between two and four additional cores. Furthermore, we require one core to run the workload with hardware-assisted virtualization. It is, moreover, likely that utilizing spare cores on the virtualization host for running parallel simulations induces a probe effect in the workload being recorded – similar to the reduction in parallelization efficiency we observe due to SMP load. Depending on the requirements of the specific research question, it may thus be recommended to use a physically separate host for the hardware-assisted virtualization (possibly wasting idle cores). These factors must be taken into account when estimating SimuBoost's overall resource requirements.

9.4 Performance Model

In Chapter 5, we have introduced a first formalization of the parallelization process in order to predict the parallel simulation time (T_{ps}) and find the optimal interval length (L_{opt}) for a particular scenario.

We begin the evaluation of the performance model by comparing the parallel simulation time that we measured in our experiments with the predictions of the model. We use the same measured best configurations of N and L as in the previous sections. We calculate T_{ps} with Equation 5.8 of the optimal model in case the configured number of nodes N is greater than $N(L)$, which is the model's estimated optimal number of nodes for the specified interval length (Equation 5.14)[6]. This describes a scenario in which our simulation cluster is large enough to provide optimal speedup for the given combination of workload and tracing mode. Therefore, the equations of the optimal setup apply. Note that we insert the interval length actually used in the practical experiments, not the model's estimated optimal interval length L_{opt}. This allows us to evaluate the accuracy of the T_{ps} estimation separately from the L_{opt} prediction. If our simulation cluster does not provide sufficient parallelism, that is, $N < N(L)$, we calculate T_{ps} with Equation 5.25 of the constrained model.

We compute the model output using the precise overheads and times observed during the corresponding practical experiment. The model thus receives the input parameters that describe the performance characteristics in the measured optimal configuration. The results thereby represent the minimum error.

Figure 9.10a illustrates the divergence of the model's estimated parallel simulation time from the measured one. We can see that for most benchmark scenarios the performance model delivers a good estimation, with errors ranging from -4.8% to +0.4%. In the majority of experiments, the model tends to underestimate the parallel simulation time. This is rooted in the fact that the model considers all intervals to be perfectly uniform. For example, it assumes the same simulation slowdown s_{sim} over the whole duration of the workload. In practice, however, this is rarely the case and intervals do show varying characteristics (slowdown, checkpoint loading time, etc.). Since variations are less likely to level out toward the end of the parallel simulation, variations concerning the last intervals have a strong influence on the model's estimation quality. In gnupg, for instance, the simulation of the last interval has a 50% higher slowdown than the average interval in the experiment. This delays the completion of the parallel simulation. The model, in contrast, assumes the same average simulation time for the last interval as for every other interval and consequently underestimates T_{ps}.

[6] See Table B.7 for results. In most cases, the prediction is close to the observed number of busy nodes (one to two cores difference).

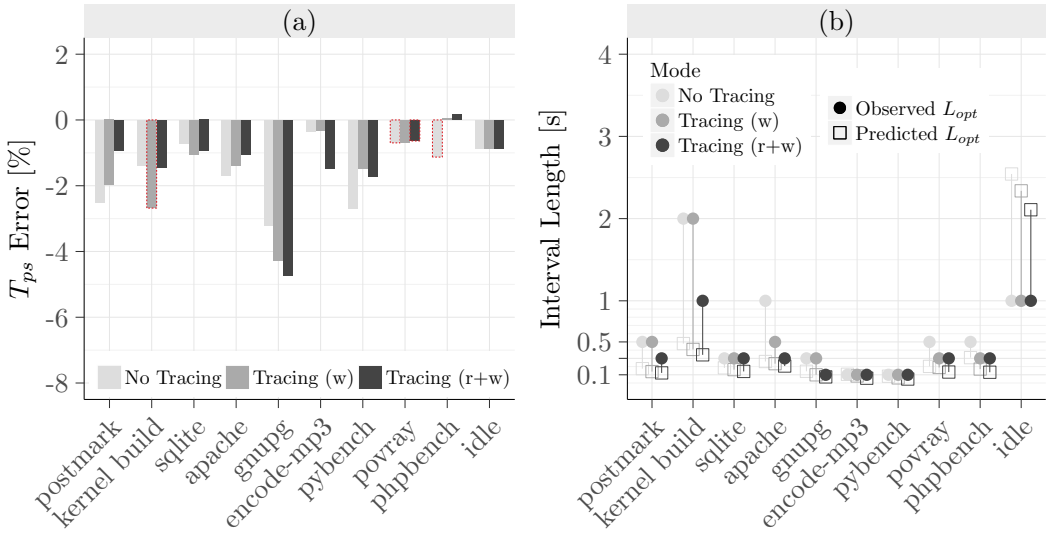


Figure 9.10: (a) The performance model tends to underestimate the parallel simulation time (T_{ps}). Uses of the constrained model (i.e., $N < N_{opt}$) are marked with the red dotted border. (b) Since the model does not incorporate the increasing costs for reducing the interval length, it typically recommends shorter intervals.

In our experiments, we also observe the opposite case, that is, the last interval completes earlier than expected by the model. That means the model's assumption that the parallel simulation always finishes with the completion of the last produced interval is not true (see kernel build in Figures 9.5). Nevertheless, this only marginally manifests as prediction error in the case of the kernel build because the overall completion time of the simulation barely changes (see Figure B.6).

Next, we review the accuracy of the model's estimation of the optimal interval length. Figure 9.10b compares the predicted L_{opt} with the actually measured one. In many cases, the model recommends a smaller interval length (e.g., kernel build, $L = 2\,\text{s}$ vs. $L_{opt} = 0.5\,\text{s}$). This reveals a decisive weakness of the model. It ignores how changing the parallelization degree and interval length influences the input parameters that describe overheads and delays:

- s_{vm}: Checkpointing and recording overhead
- s_{sim}: Simulation slowdown
- t_c: Checkpointing downtime
- t_s: Simulation start-up time

Following the model's recommendation[7] in case of the kernel build (no tracing) leads in practice to a 12% higher simulation time with the same number of nodes deployed (equals the model's N_{opt}). The simulation of sqlite even takes 73% longer in the predicted optimal configuration. This is because the model does

[7] We use the results from the experiments with the closest matching configuration.

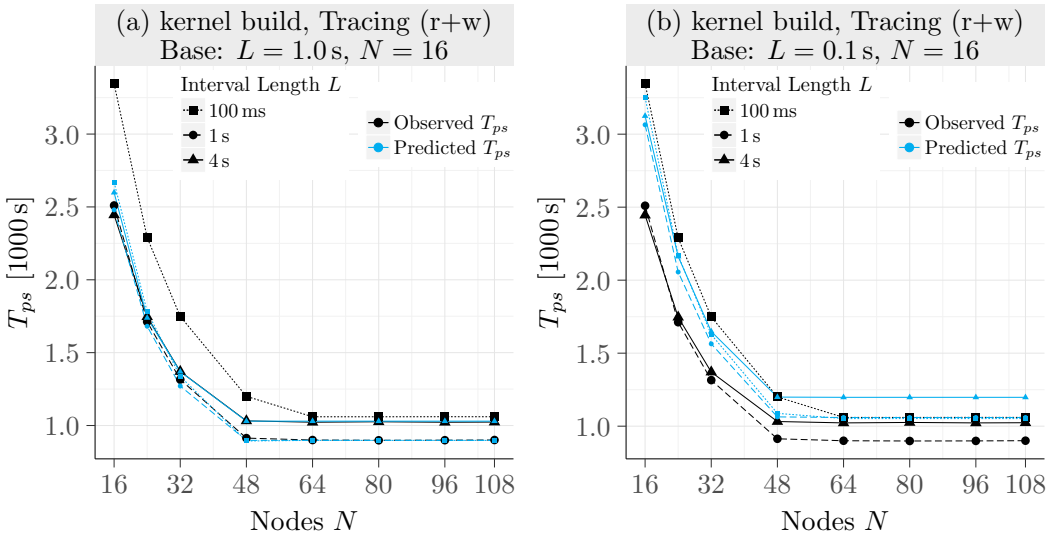


Figure 9.11: (a) The model noticeably underestimates the parallel simulation time for $L = 100\,\text{ms}$, if we use the overheads and delays measured in a run with 1 s intervals. (b) The model, in turn, overestimates T_{ps} for $L \in \{1, 4\}$ s, if we use the parameters measured with 100 ms intervals.

not incorporate the rising overhead (e.g., translation costs) when shortening the interval length. In consequence, the supposedly optimal interval length can, in fact, be too short. Alternatively, more nodes are needed than estimated to attain a comparable performance (i.e., N_{opt} is too small).

In practice, this shortcoming makes it difficult to properly characterize a workload and find representative values for the model parameters. The model does only deliver good estimations for interval lengths that exhibit similar overheads and delays as the interval length used to determine the parameters. In Figure 9.11a, we measure the parameters for a kernel build with 1 s intervals and 16 nodes and subsequently use the observed values to estimate the parallel simulation time for other configurations. While the model closely predicts T_{ps} with varying number of nodes and $L \in \{1, 4\}$, it fails to do so for $L = 100\,\text{ms}$, where the overheads are much larger. Conversely, if we determine the parameters with 100 ms intervals, the estimation quality for $L \in \{1, 4\}$ is unsatisfying, while the results for $L = 100\,\text{ms}$ are much better.

9.5 Conclusion

In this chapter, we demonstrated that SimuBoost is able to significantly reduce the slowdown of functional full system simulation. For most of the workloads, only a 10% to 30% slowdown compared to fast hardware-assisted virtualization re-

mains. SimuBoost efficiently scales beyond the limits of a single physical machine, thereby also presenting an alternative to user-mode tools such as SuperPin that are restricted in parallelization due to the use of forking instead of checkpointing.

We identified various factors that affect the efficiency of the parallelization. Idle times in the cluster that stem from phase behavior in the workload have the most impact. In addition, reducing the interval length generally entails a decrease in efficiency, which is especially notable for workloads with low code and memory access locality. We can confirm previous research in that the concept of partitioning and parallelization of simulation time is an effective and efficient acceleration approach. SimuBoost is the first project to apply this concept to functional full system simulation.

To estimate the parallel simulation time and determine the optimal interval length for a given scenario, we presented a performance model. The evaluation shows that while the model is generally able to provide estimations of the simulation time with low error (-4.5% to +0.4%), this heavily depends on the interval length used to measure the input parameters that describe overheads and delays (e.g., the simulation slowdown). In our benchmarks, we could only attain satisfying prediction quality for interval lengths that exhibit similar input parameters. Consequently, to get accurate estimations for the optimal case requires that the input parameters have been retrieved with this very optimal interval length. As the model infers the optimal interval length from its estimation of the parallel simulation time, this creates a dependency circle and, in turn, results in low prediction quality for the optimal interval length.

Chapter 10

Conclusion

Functional full system simulation is a powerful tool for gathering detailed information on the run-time behavior of applications and the operating system. A major downside of this technology is the immense slowdown that is bound to the software-driven execution model. Workloads that complete in minutes natively, can quickly run hours or even days with functional full system simulation, especially with heavyweight instrumentation. A slowdown between one and two orders of magnitude is common. This makes experiments not only time-consuming, but also prevents natural interaction with the examined workload and distorts timing whenever external input is involved (e.g, I/O).

SimuBoost leverages the concept of partitioning and parallelization of simulation time to drastically reduce the slowdown. The workload is first executed in a fast hardware-assisted virtual machine (VM). Periodic checkpoints serve as starting points for parallel simulations with one job per interval. Heterogeneous deterministic replay guarantees that the simulations repeat the exact same execution as in the hardware-assisted VM, including interactions and recorded timing.

With continuous checkpointing being a key technology for SimuBoost, we evaluated various checkpointing techniques by extending QEMU/KVM and found a combination of asynchronous (i.e., copy-on-write) and incremental checkpointing to provide the best performance. We further devised a fast method for tracking of modified pages, which we call pre-scan. Pre-scan asynchronously scans the EPT for set status bits before the downtime so as to reduce the time necessary for a consistent scan while the VM is suspended. This way, SimuBoost attains an average downtime of only 8 ms and preserves interactivity during checkpointing. The run-time overhead varies with the workload and interval length, ranging from less than 1% up to 100% in our experiments.

In order to distribute the checkpoints in a network of simulation nodes, we propose the use of a point-to-multipoint protocol such as IP multicast. This mitigates the bottleneck at the network interface of the checkpointing host. Since the

data volume generated during checkpointing can quickly exceed the bandwidth limit of contemporary networks, we demonstrated how a combination of data deduplication, delta compression, generic compression, and special-purpose compression is able to considerably reduce the size of continuous checkpoints. With a compression ratio of up to 39:1, SimuBoost can be used with Gigabit Ethernet.

The implementation of the heterogeneous deterministic replay turned out to be technically challenging. Eventually getting the intervals to faithfully replay in the simulation required various changes in the binary translation of QEMU. This also includes numerous adaptations to make the simulation work like the recorded hardware counterpart (e.g., status flag computation and instruction counting). Our prototype is capable of successfully replaying on average over 97% of intervals. The additional run-time overhead for recording is typically below 1%. However, we found that the trap for capturing timestamp counter readings can impose a considerable overhead if the workload performs excessive time keeping (e.g., apache: 90% run-time overhead).

Our final evaluation of SimuBoost confirms previous research that has shown significant speedup potential for the partitioning and parallelization of simulation time. SimuBoost, for the first time, demonstrates that the concept can also be very effectively used to accelerate continuous functional full system simulation. For most workloads, we measured a remaining slowdown for simulation over hardware-assisted virtualization of less than 30%, irrespective of the degree of instrumentation (tracing memory reads and writes). Only benchmarks with considerable run-time overhead during the checkpointing and recording phase show higher remaining slowdowns (apache: 120%, postmark: 63% – 81%).

10.1 Limitations and Future Work

While SimuBoost overcomes major limitations of previous work, some aspects have not yet been addressed or show room for further improvement:

Three-Stage SimuBoost

SimuBoost's property to accurately replay the execution from the hardware-assisted virtualization is one of its greatest strengths because it (re-)produces realistic timing in the simulation. This makes measurements more representative of real-world executions without the necessity of complex CPU and device timing models. However, this is only true as long as the overhead for checkpointing and recording remains low. Otherwise, the overhead can become visible in distorted timing and create a noticeable probe effect, which, for example, embodies in shorter effective time slices for guest processes due to intermittent VM exits. In Chapter 6, we demonstrated that, despite advanced checkpointing techniques,

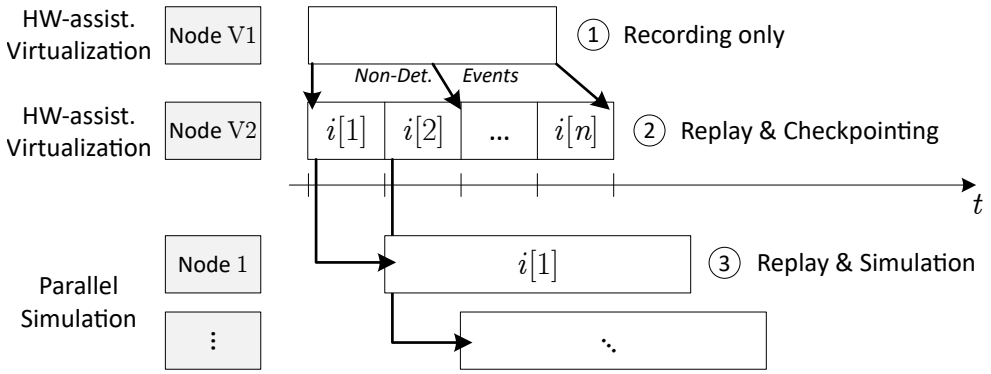


Figure 10.1: The first stage records non-deterministic events only. The execution is concurrently replayed in a second VM that periodically takes the checkpoints. The parallel simulations receive the same events.

heavyweight workloads such as postmark and SPECjbb may still experience high run-time overheads. In Chapter 8, we moreover showed that in case of frequent time stamps reads, recording of non-deterministic events can incur a notable further overhead. Nevertheless, the majority of overhead stems from the checkpointing and the overhead for recording remains below 1% for most workloads. It would thus be desirable to decouple the checkpointing from the recording so that the overhead of the checkpointing does not reflect in the recorded execution. This, in turn, would entirely eliminate any probe effect related to checkpointing from the simulations and produce more stable and realistic results.

We can accomplish this by splitting the checkpointing and recording into dedicated stages (see Figure 10.1). In the first stage, we only record non-deterministic events. We then feed these events to a concurrently running second hardware-assisted virtual machine. This VM consequently reproduces the low-overhead execution from stage one. Since the replay is not influenced by additional overhead (just like in the case of the much slower simulation), we can take checkpoints in the second VM without inflicting a probe effect.

It is, however, open to what extent the parallelization is negatively affected by this. Although the second VM may start without delay and immediately receive events[1], we can expect the virtual machine to run somewhat slower than in the original design. This is because we probably see additional overhead for asynchronous event injection, which on x86 requires multiple VM exits[2] [87]. The increased run-time overhead during checkpointing, in turn, possibly leads to slower parallelization and increased parallel simulation time. We already started exploring the advantages and disadvantages of the three-stage design in an ongoing bachelor's thesis.

[1] We can pause the VM as soon as the event queues run empty.

[2] Although the instruction counter can be configured to generate an interrupt, the stop does not occur instantaneously and we may miss the desired value. We thus have to stop earlier and step toward the correct instruction.

Multiprocessor Support

A major limitation of the current SimuBoost prototype is its restriction to single-core virtual machines. SimuBoost shares this shortcoming with many projects based on deterministic replay. As described in § 2.4.3, this originates from the difficulty to efficiently track accesses to shared memory regions in multiprocessor systems. While it is technically possible to accurately record and replay the order of memory accesses in software, for example, with a CREW protocol, this generally comes at a high run-time overhead. SMP-Revirt [88], for instance, incurs a 2x slowdown for a kernel build on two processors, and a 9x slowdown for four processors. Samsara [215] improves upon these values with better scalability (e.g., 6x slowdown for four cores). While we validated the general applicability of chunk-based recording to *heterogeneous* deterministic replay [293], the overhead still presents a considerable probe effect, which cannot even be avoided with the aforementioned three-stage SimuBoost design. The most promising solution is a CPU-integrated hardware recording logic, as demonstrated by Intel with QuickRec [205]. The FPGA-based prototype suffered only negligible performance degradation for tracking shared memory accesses in user processes.

Performance Model

Our evaluation of the performance model reveals that describing the overheads and delays in the model with nonconstant functions could improve the output quality. This is especially necessary to get good estimations when changing the interval length, which is indispensable for obtaining an accurate prediction of the optimal interval length. Since determining overhead functions is a cumbersome and nontrivial task, it is desirable to develop a method for fast automated workload characterization.

Appendix A

Deutsche Zusammenfassung

Für das Sammeln detaillierter Laufzeitinformationen, wie Speicherzugriffsmustern, wird in der Betriebssystem- und Sicherheitsforschung häufig auf die funktionale Systemsimulation zurückgegriffen. Der Simulator führt dabei die zu untersuchende Arbeitslast in einer virtuellen Maschine (VM) aus, indem er schrittweise Instruktionen interpretiert oder derart übersetzt, sodass diese auf dem Zustand der VM arbeiten. Dieser Prozess ermöglicht es, eine umfangreiche Instrumentierung durchzuführen und so an Informationen zum Laufzeitverhalten zu gelangen, die auf einer physischen Maschine nicht zugänglich sind.

Obwohl die funktionale Systemsimulation als mächtiges Werkzeug gilt, stellt die durch die Interpretation oder Übersetzung resultierende immense Ausführungsverlangsamung eine substanzielle Einschränkung des Verfahrens dar. Im Vergleich zu einer nativen Ausführung messen wir für QEMU eine 30-fache Verlangsamung, wobei die Aufzeichnung von Speicherzugriffen diesen Faktor verdoppelt. Mit Simulatoren, die umfangreichere Instrumentierungsmöglichkeiten mitbringen als QEMU, kann die Verlangsamung um eine Größenordnung höher ausfallen. Dies macht die funktionale Simulation für lang laufende, vernetzte oder interaktive Arbeitslasten uninteressant. Darüber hinaus erzeugt die Verlangsamung ein unrealistisches Zeitverhalten, sobald Aktivitäten außerhalb der VM (z. B. Ein-/Ausgabe) involviert sind.

In dieser Arbeit stellen wir **SimuBoost** vor, eine Methode zur drastischen Beschleunigung funktionaler Systemsimulation. SimuBoost führt die zu untersuchende Arbeitslast zunächst in einer schnellen hardwaregestützten virtuellen Maschine aus. Dies ermöglicht volle Interaktivität mit Benutzern und Netzwerkgeräten. Während der Ausführung erstellt SimuBoost periodisch Abbilder der VM (engl. Checkpoints). Diese dienen als Ausgangspunkt für eine parallele Simulation, bei der jedes Intervall unabhängig simuliert und analysiert wird. Eine heterogene deterministische Wiederholung (engl. heterogeneous deterministic Replay) ga-

rantiert, dass in dieser Phase die vorherige hardwaregestützte Ausführung jedes Intervalls exakt reproduziert wird, einschließlich Interaktionen und realistischem Zeitverhalten.

Unser Prototyp ist in der Lage, die Laufzeit einer funktionalen Systemsimulation deutlich zu reduzieren. Während mit herkömmlichen Verfahren für die Simulation des Bauprozesses eines modernen Linux über 5 Stunden benötigt werden, schließt SimuBoost die Simulation in nur 15 Minuten ab. Dies sind lediglich 16% mehr Zeit, als der Bau in einer schnellen hardwaregestützten VM in Anspruch nimmt. SimuBoost ist imstande, diese Geschwindigkeit auch bei voller Instrumentierung zur Aufzeichnung von Speicherzugriffen beizubehalten.

Die vorliegende Arbeit ist das erste Projekt, welches das Konzept der Partitionierung und Parallelisierung der Ausführungszeit auf die interaktive Systemvirtualisierung in einer Weise anwendet, die eine sofortige parallele funktionale Simulation gestattet. Wir ergänzen die praktische Umsetzung mit einem mathematischen Modell zur formalen Beschreibung der Beschleunigungseigenschaften. Dies erlaubt es, für ein gegebenes Szenario die voraussichtliche parallele Simulationszeit zu prognostizieren und gibt eine Orientierung zur Wahl der optimalen Intervalllänge. Im Gegensatz zu bisherigen Arbeiten legt SimuBoost einen starken Fokus auf die Skalierbarkeit über die Grenzen eines einzelnen physischen Systems hinaus. Ein zentraler Schlüssel hierzu ist der Einsatz moderner Checkpointing-Technologien. Im Rahmen dieser Arbeit präsentieren wir zwei neuartige Methoden zur effizienten und effektiven Kompression von periodischen Systemabbildern.

Appendix B

Additional Figures and Data

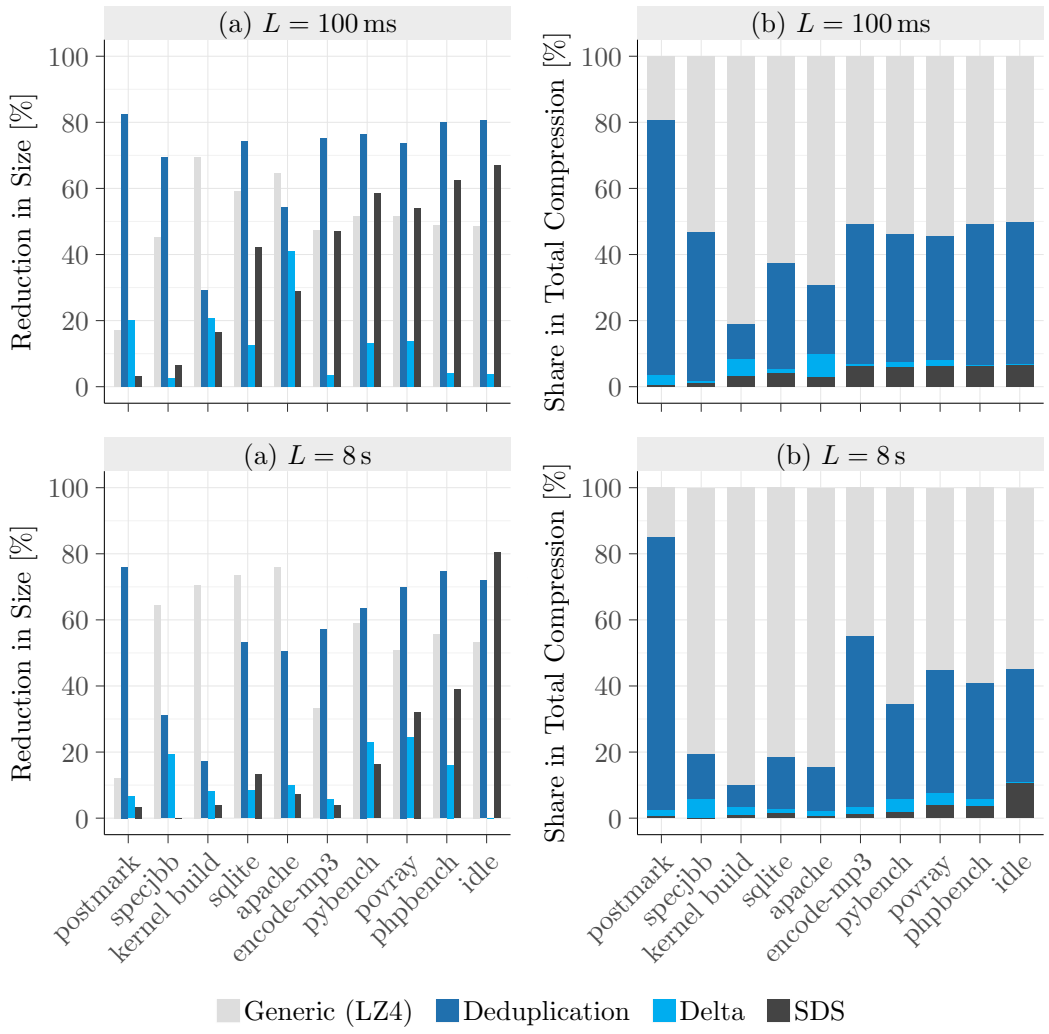


Figure B.1: On average, the specialized compression techniques profit from shorter intervals (i.e., more checkpoints), whereas longer intervals accumulate more entropy and make generic compression more effective.

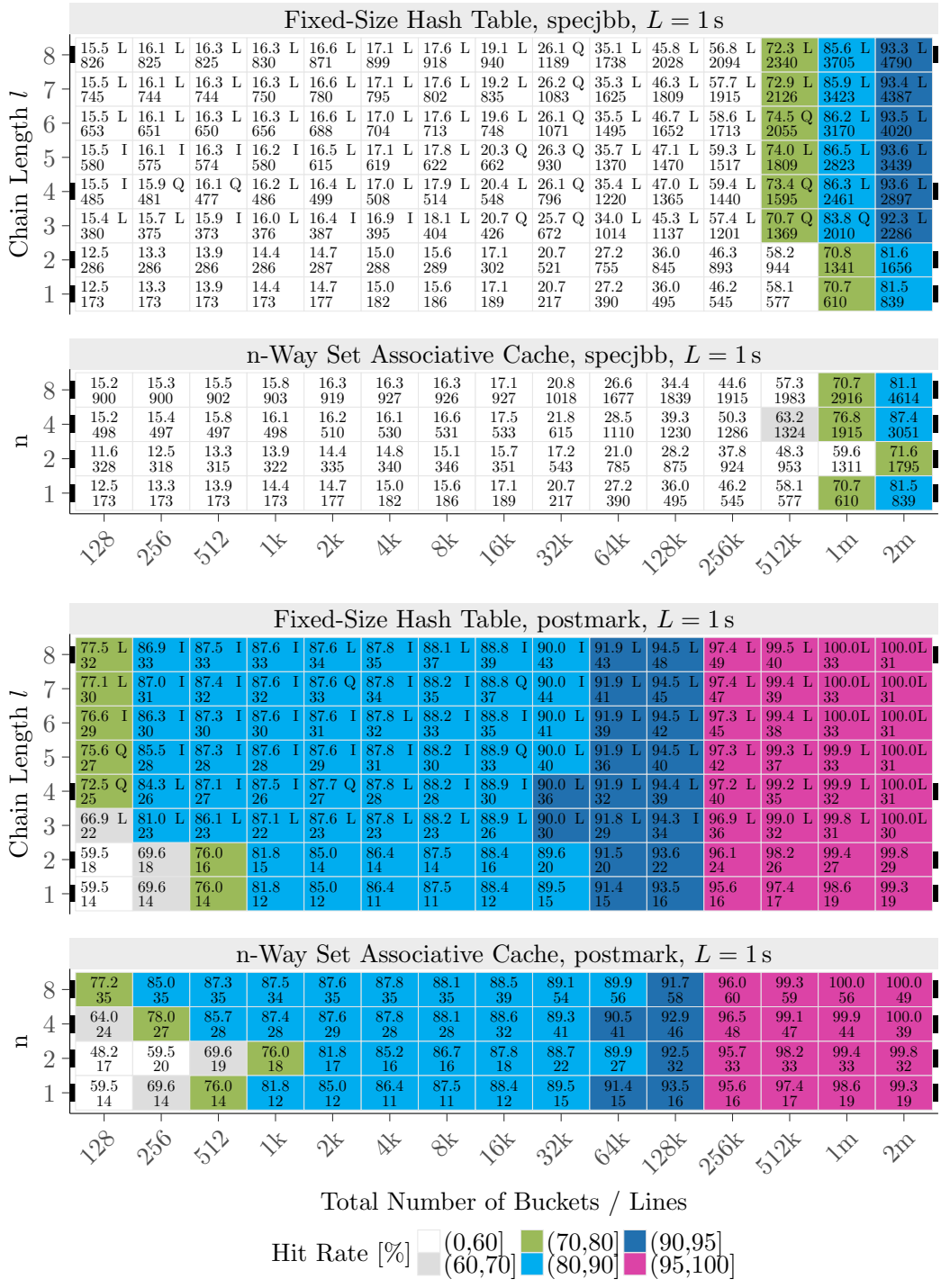
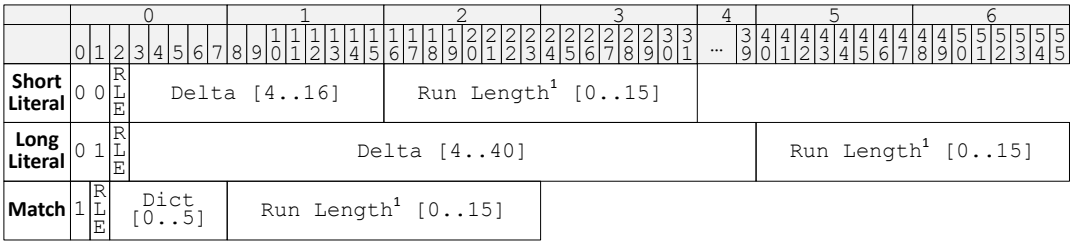


Figure B.2: Overview of fixed-size hash table and n-way associative cache hit rates for data deduplication in SPECjbb and postmark.



[1] Only with RLE bit set.

Figure B.3: SDS match and literal encoding. Short literals comprise deltas up to 2^{16} bytes. Long literals describe deltas that span at most 2^{40} bytes. That means the offsets in the checkpoint database for two consecutive pages in the state map may not exceed 1 TiB. Runs of up to 2^{16} identical deltas (a full table) can be encoded with the RLE extension.

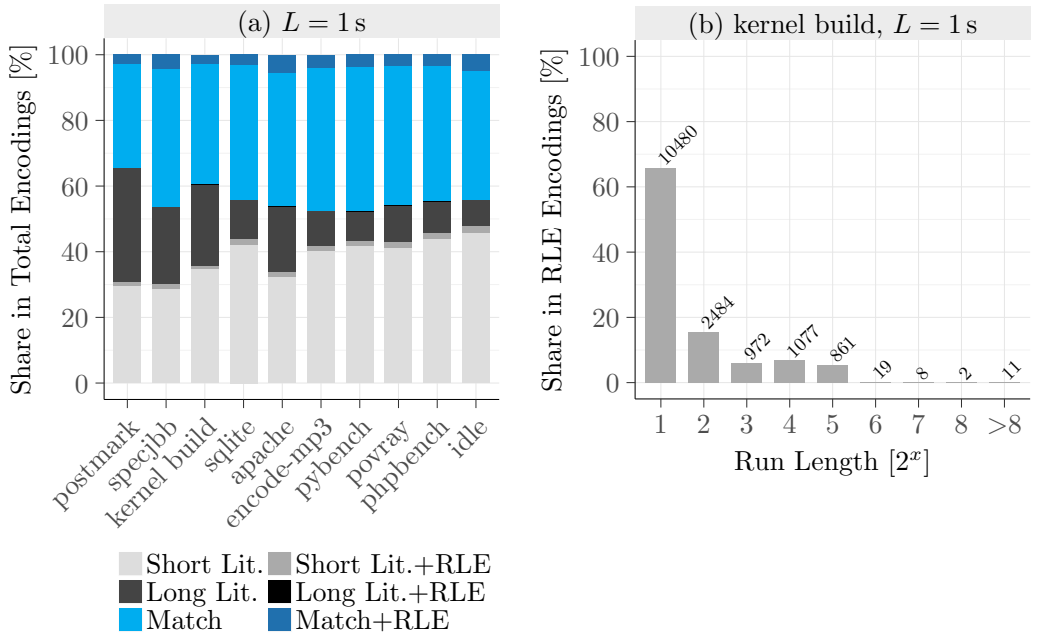


Figure B.4: (a) The final output of an SDS encoded RAM state map contains mostly short literals (2 bytes) and matches (1 byte). Long literals with RLE extension (7 bytes) are rarely used. (b) A run length of 2 (i.e., 1 repetition) is most frequent, making up 66% of all RLE encodings. Except for 2^1, values depict the share of run lengths between 2^{x-1} and up to 2^x. The annotations provide the average absolute number of occurrences for each run length in kernel build checkpoints for a 4 GiB VM.

Sync. Event	Site	Size	Description
`CPUID`	KVM	16	The return value of the x86 `CPUID` instruction. It contains information about supported CPU features and a CPU identification. Recorded information: `EAX`(4), `EBX`(4), `ECX`(4), `EDX`(4)
`RDTSC`	KVM	8	The time stamp returned by the `RDTSC` instruction. Recorded information: `EAX`(4), `EDX`(4)
`RDMSR` `WRMSR` [Opt.]	KVM	12	The value read from or written to a model specific register (MSR) using the `RDMSR` and `WRMSR` instructions. Needed to fake unimplemented hardware CPU features. `ECX` specifies the MSR register. Recorded information: `EAX`(4), `EDX`(4), `ECX`(4) [Opt.]
`IN` `OUT` [Opt.]	KVM	12	The value read or written using port I/O. Recorded information: Port(4), Value(4), Length(4) [Opt.]
Read APIC† Write APIC† [Opt.]	KVM	12	The value read from or written to the APIC memory at `0xFEE00000` – `0xFEEFFFFF` in guest physical memory. We inject interrupts from the log and do not faithfully emulate the APIC. Recorded information: Address(4), Value(4), Length(4) [Opt.]
Read HPET Write HPET [Opt.]	QEMU	12	The value read from or written to the APIC memory at `0xFED00000` – `0xFED003FF` in guest physical memory. Required for correct clock and timer replay. Recorded information: Address(4), Value(4), Length(4) [Opt.]

† Requires deactivation of direct interrupt delivery (APICv) to virtual machines.

Table B.1: Synchronous non-deterministic events recorded by SimuBoost. Site denotes recording location. Optional events or information are not needed for successful replay, but help detecting diverging replays. The value in parentheses indicates size in bytes. Total size is also in bytes and comprises event specific data only.

Async. Event	Site	Size	Description
INT†	KVM	4	Interrupt Recorded information: Vector(4)
SMI	KVM	0	System management interrupt (SMI) to enter system management mode (SMM). Recorded information: -
Write DMA	QEMU	8+X	Write from a virtual device into guest physical memory using (virtual) DMA. Recorded information: Address(8), Data(X)

† Requires deactivation of direct interrupt delivery (APICv) to virtual machines.

Table B.2: Asynchronous non-deterministic events recorded by SimuBoost. Site denotes recording location. The value in parentheses indicates size in bytes. Total size is also in bytes and comprises event specific data only.

Port Address	Description
`0x1CE` – `0x1D0`	VESA BIOS Extension (VBE)
`0x3B0` – `0x3DF`	Video Graphics Array (VGA)
`0x3F8` – `0x3FC`	COM1 Serial Port
`0x402` [Opt.]	SeaBIOS Debug Port
`0xCF8`	PCI Index
`0xCFC` – `0xCFF`	PCI Data Only if chipset or video adapter are selected via the PCI index port.

Table B.3: To make output over the video adapter or the serial port accessible during replay, the given ports must be forwarded to the corresponding virtual devices. This also allows proper configuration of the physical address space when loading a checkpoint or replaying the BIOS.

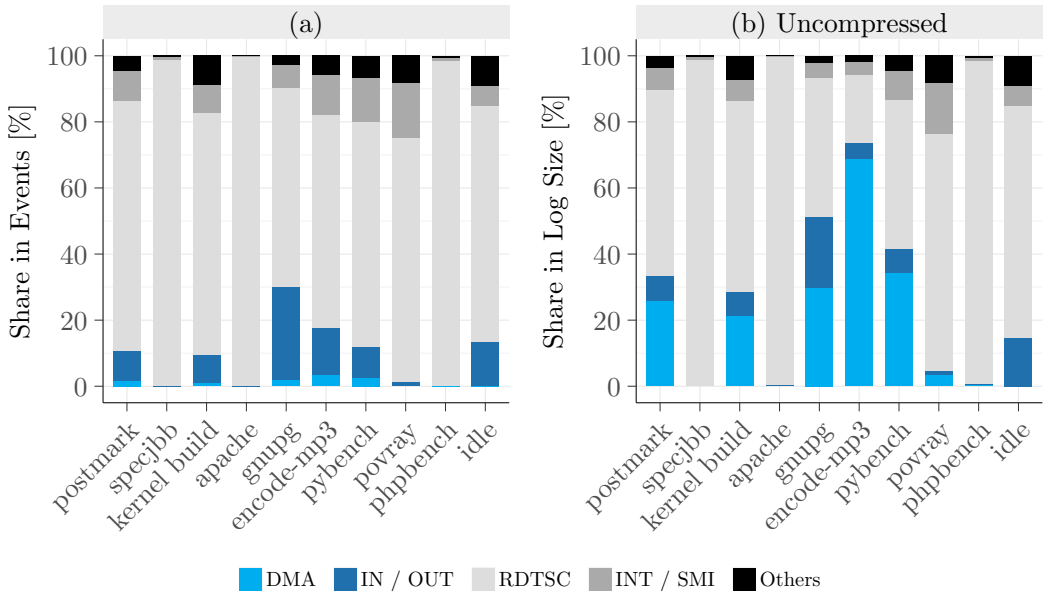


Figure B.5: (a) With a share of 60% to 99%, timestamp readings (RDTSC) are by far the most common events. (b) Looking at the share in the overall log size, however, reveals that DMA operations possess the highest per-event size. Shares encompass reads and writes. See Tables B.1 and B.2 for a complete list of event types.

postmark	specjbb	kernel build	apache	gnupg
76 ↘ 26	22996 ↘ 1560	1010 ↘ 319	9816 ↘ 699	71 ↘ 25

encode-mp3	pybench	povray	phpbench	idle
144 ↘ 105	60 ↘ 26	684 ↘ 135	3427 ↘ 146	5 ↘ 0.6

Table B.4: Total log size per workload before and after compression in MiB.

	L	Failed	Total	Rate		L	Failed	Total	Rate
sqlite	3.0 s	1	11	9.09%	phpbench	3.0 s	1	145	0.69%
encode-mp3	4.0 s	1	19	5.26%	pybench	0.1 s	1	647	0.15%
kernel build	4.0 s	8	208	3.85%	apache	0.3 s	1	843	0.12%
gnupg	0.5 s	1	86	1.16%	postmark	0.1 s	0	1040	0.00%
povray	3.0 s	4	380	1.05%	idle	0.1 s	0	587	0.00%
									∅ 2.14%

Table B.5: Maximum replay failure rate observed over all configurations according to Chapter 9.

	T_{vm}	T_{ps}		T_{sim} (s_{sim})			
	HW-Virt.	SimuBoost		Serial Replay		Serial Sim.	
apache	01:58	04:20	(2.19x)	59:03	(29.84x)	57:31	(29.06x)
L=1.0 s t_c=4.24 ms s_{vm} =2.02		n=241 N=16(16) t_s=432 ms S=13.62x E=85(85)%					
+ tracing (w)		04:18	(2.18x)	01:15:32	(38.17x)	01:25:41	(43.30x)
L=0.5 s t_c=3.87 ms s_{vm} =2.05		n=489 N=24(20) t_s=338 ms S=17.51x E=73(88)%					
+ tracing (r+w)		04:21	(2.20x)	01:41:36	(51.34x)	01:57:39	(59.45x)
L=0.3 s t_c=3.64 ms s_{vm} =2.09		n=825 N=32(27) t_s=292 ms S=23.30x E=73(86)%					
encode-mp3	01:04	01:10	(1.10x)	17:53	(16.75x)	16:00	(15.00x)
L=0.1 s t_c=2.73 ms s_{vm} =1.04		n=659 N=24(20) t_s=135 ms S=15.23x E=63(76)%					
+ tracing (w)		01:11	(1.11x)	27:29	(25.74x)	30:40	(28.72x)
L=0.1 s t_c=2.73 ms s_{vm} =1.04		n=659 N=32(29) t_s=136 ms S=23.10x E=72(80)%					
+ tracing (r+w)		01:15	(1.18x)	59:03	(55.30x)	01:07:44	(63.43x)
L=0.1 s t_c=2.73 ms s_{vm} =1.04		n=665 N=64(60) t_s=135 ms S=47.02x E=73(78)%					
gnupg	00:33	00:45	(1.33x)	04:37	(8.17x)	04:30	(7.97x)
L=0.3 s t_c=3.42 ms s_{vm} =1.20		n=135 N=16(10) t_s=227 ms S=6.17x E=39(62)%					
+ tracing (w)		00:48	(1.42x)	09:28	(16.73x)	12:34	(22.20x)
L=0.3 s t_c=3.42 ms s_{vm} =1.20		n=135 N=24(15) t_s=233 ms S=11.80x E=49(79)%					
+ tracing (r+w)		00:48	(1.43x)	18:03	(31.90x)	23:10	(40.94x)
L=0.1 s t_c=3.00 ms s_{vm} =1.23		n=417 N=48(36) t_s=153 ms S=22.34x E=47(62)%					
idle	01:00	00:59	(1.00x)	00:02	(0.04x)	01:00	(1.01x)
L=1.0 s t_c=4.11 ms s_{vm} =1.00		n=59 N=16(1) t_s=125 ms S=0.04x E=0(4)%					
+ tracing (w)		00:59	(1.00x)	00:02	(0.04x)	01:00	(1.00x)
L=1.0 s t_c=4.11 ms s_{vm} =1.00		n=59 N=16(1) t_s=127 ms S=0.04x E=0(4)%					
+ tracing (r+w)		00:59	(1.00x)	00:03	(0.06x)	01:00	(1.00x)
L=1.0 s t_c=4.39 ms s_{vm} =1.00		n=59 N=16(1) t_s=126 ms S=0.06x E=0(6)%					
kernel build	12:52	14:54	(1.16x)	05:08:12	(23.93x)	04:51:49	(22.66x)
L=2.0 s t_c=6.13 ms s_{vm} =1.08		n=410 N=24(24) t_s=439 ms S=20.68x E=86(86)%					
+ tracing (w)		15:35	(1.21x)	07:05:37	(33.05x)	07:53:02	(36.73x)
L=2.0 s t_c=6.13 ms s_{vm} =1.08		n=410 N=32(32) t_s=461 ms S=27.31x E=85(85)%					
+ tracing (r+w)		15:13	(1.18x)	10:25:10	(48.55x)	11:50:43	(55.19x)
L=1.0 s t_c=5.33 ms s_{vm} =1.10		n=838 N=48(48) t_s=516 ms S=41.06x E=86(86)%					
phpbench	06:31	07:42	(1.18x)	02:54:29	(26.78x)	02:14:56	(20.71x)
L=0.5 s t_c=2.87 ms s_{vm} =1.09		n=849 N=24(24) t_s=129 ms S=22.64x E=94(94)%					
+ tracing (w)		07:24	(1.14x)	05:04:21	(46.70x)	04:49:27	(44.42x)
L=0.3 s t_c=2.78 ms s_{vm} =1.09		n=1424 N=48(47) t_s=125 ms S=41.09x E=86(87)%					
+ tracing (r+w)		07:31	(1.15x)	08:26:53	(77.78x)	08:48:01	(81.02x)
L=0.3 s t_c=2.77 ms s_{vm} =1.09		n=1422 N=80(78) t_s=128 ms S=67.35x E=84(86)%					
postmark	00:59	01:37	(1.63x)	25:35	(25.82x)	19:12	(19.37x)
L=0.5 s t_c=6.51 ms s_{vm} =1.43		n=172 N=24(17) t_s=384 ms S=15.80x E=66(93)%					
+ tracing (w)		01:42	(1.72x)	40:29	(40.85x)	44:36	(45.01x)
L=0.5 s t_c=6.51 ms s_{vm} =1.43		n=172 N=32(27) t_s=387 ms S=23.78x E=74(88)%					
+ tracing (r+w)		01:47	(1.81x)	01:00:32	(61.09x)	01:15:58	(76.66x)
L=0.3 s t_c=5.67 ms s_{vm} =1.54		n=306 N=48(41) t_s=511 ms S=33.72x E=70(82)%					

Continues on next page →

	T_{vm}	T_{ps}		T_{sim} (s_{sim})			
	HW-Virt.	SimuBoost		Serial Replay		Serial Sim.	
povray	18:47	19:57	(1.06x)	25:11:21	(80.40x)	25:31:56	(81.49x)
L=0.5 s t_c=2.88 ms s_{vm}=1.01		n=2249 N=80(80) t_s=153 ms S=75.75x E=95(95)%					
+ tracing (w)		19:51	(1.06x)	29:14:22	(93.32x)	31:54:10	(101.82x)
L=0.3 s t_c=2.82 ms s_{vm}=1.01		n=3806 N=96(95) t_s=186 ms S=88.34x E=92(93)%					
+ tracing (r+w)		24:20	(1.29x)	41:08:52	(131.33x)	45:17:48	(144.57x)
L=0.3 s t_c=2.74 ms s_{vm}=1.01		n=3776 N=108(108) t_s=182 ms S=101.44x E=94(94)%					
pybench	01:02	01:11	(1.15x)	29:46	(28.51x)	23:55	(22.92x)
L=0.1 s t_c=2.91 ms s_{vm}=1.03		n=643 N=32(30) t_s=171 ms S=24.87x E=78(83)%					
+ tracing (w)		01:13	(1.17x)	52:19	(50.12x)	53:40	(51.42x)
L=0.1 s t_c=2.91 ms s_{vm}=1.03		n=643 N=64(54) t_s=253 ms S=42.84x E=67(79)%					
+ tracing (r+w)		01:17	(1.23x)	01:27:26	(83.76x)	01:39:03	(94.89x)
L=0.1 s t_c=3.01 ms s_{vm}=1.03		n=646 N=96(88) t_s=183 ms S=68.10x E=71(77)%					
sqlite	00:28	00:25	(0.89x)	00:57	(2.04x)	01:57	(4.15x)
L=0.3 s t_c=3.47 ms s_{vm}=0.84		n=79 N=16(3) t_s=233 ms S=2.28x E=14(76)%					
+ tracing (w)		00:25	(0.91x)	01:13	(2.61x)	02:32	(5.41x)
L=0.3 s t_c=3.47 ms s_{vm}=0.84		n=79 N=16(4) t_s=258 ms S=2.88x E=18(72)%					
+ tracing (r+w)		00:25	(0.92x)	01:38	(3.49x)	03:37	(7.70x)
L=0.3 s t_c=3.41 ms s_{vm}=0.84		n=79 N=16(5) t_s=224 ms S=3.81x E=24(76)%					

Table B.6: Detailed overview of SimuBoost evaluation results. The numbers given in parenthesis are the slowdown compared to T_{vm} for the corresponding execution mode. For N and E, the number in parenthesis denotes the median number of actually used nodes and the corresponding efficiency, respectively. Run times are in the format [hh:mm:ss]. Measurements reflect the best configuration according to Chapter 9. See Chapter 5 for a description of the symbols.

	No Tracing			Tracing (w)			Tracing (r+w)		
	N	Busy	$N(L)$	N	Busy	$N(L)$	N	Busy	$N(L)$
apache	16	16	16	24	20	21	32	27	27
encode-mp3	24	20	19	32	29	28	64	60	57
gnupg	16	10	9	24	15	17	48	36	29
idle	16	1	1	16	1	1	16	1	1
kernel build	24	24	24	32	32	33	48	48	48
phpbench	24	24	25	48	47	44	80	78	72
postmark	24	17	19	32	27	29	48	41	43
povray	80	80	81	96	95	94	108	108	133
pybench	32	30	32	64	54	54	96	88	89
sqlite	16	3	4	16	4	5	16	5	6

Table B.7: Estimated optimal number of nodes (Equation 5.14) is close to the measured number of busy nodes, when using the measured optimal interval length. In case of povray (r+w), our cluster is not large enough.

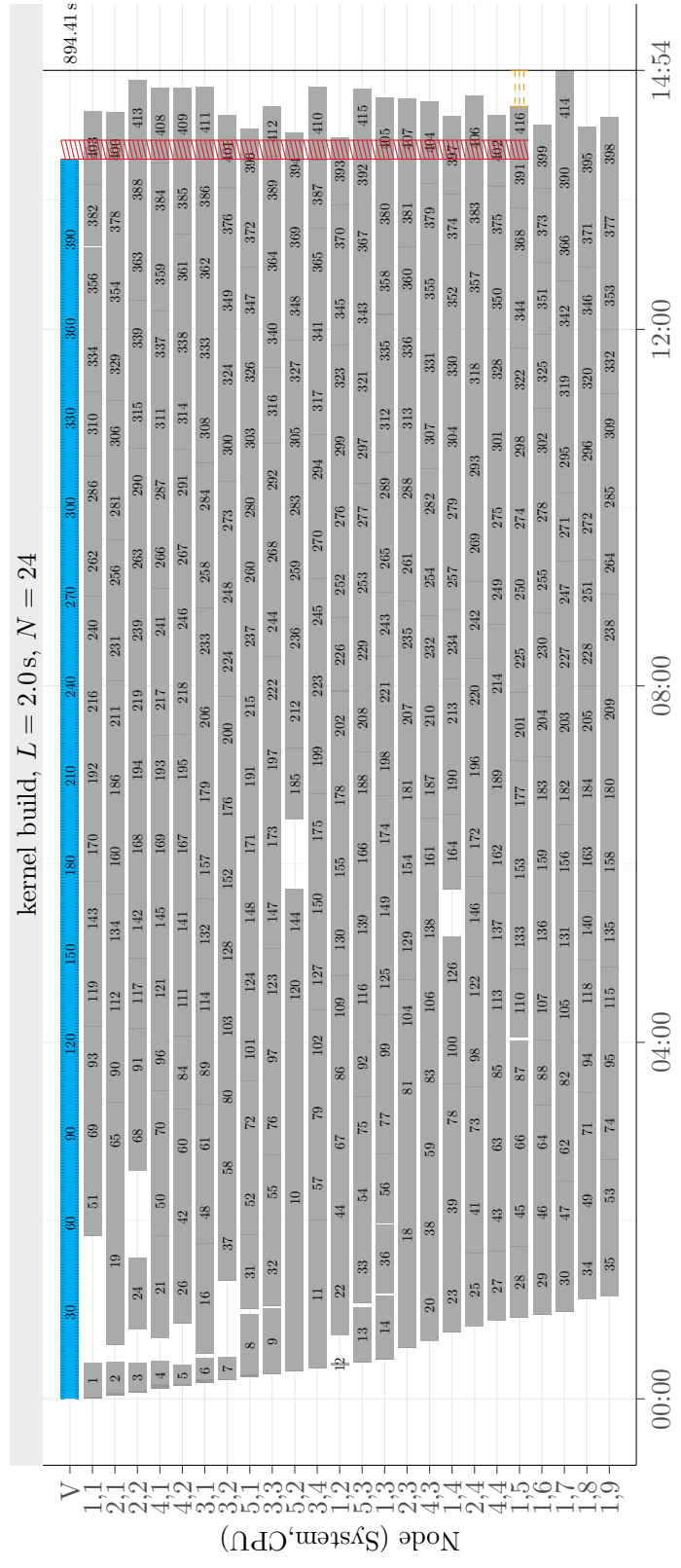


Figure B.6: Simulation diagram of a kernel build. Intervals at the beginning complete faster. The last produced interval takes only 50% of the average interval simulation time. However, due to the delayed simulation (red bar), the overall completion time only marginally changes. Missing intervals in the diagram failed to replay.

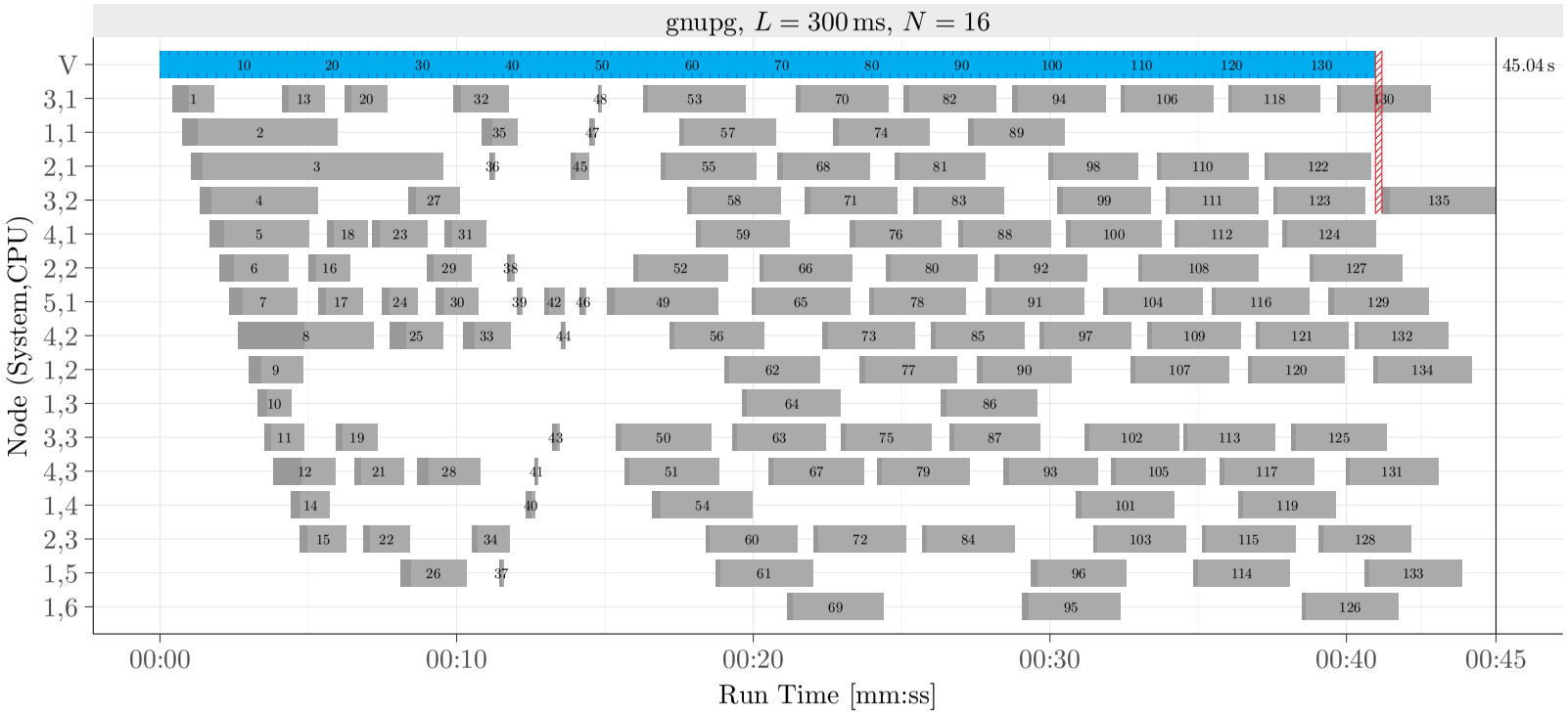


Figure B.7: Simulation diagram of gnupg. The two program phases are clearly visible in the interval simulation times. The last produced interval marks the end of the parallel simulation, but takes 50% longer to simulate than the average interval. As we use more nodes than necessary, a lot of idle time is created.

List of Tables

List of Figures

Bibliography

[1] Apache HTTP Server Project. http://httpd.apache.org/. [Online; retrieved June 11, 2019].

[2] Bochs IA-32 Emulator. http://bochs.sourceforge.net/. [Online; retrieved Apr. 17, 2018].

[3] FFmpeg. http://ffmpeg.org/. [Online; retrieved June 12, 2018].

[4] GNU Gzip. http://www.gnu.org/software/gzip/. [Online; retrieved June 11, 2018].

[5] Google FarmHash. https://github.com/google/farmhash. [Online; retrieved Jan. 28, 2019].

[6] Kyoto Cabinet. http://fallabs.com/kyotocabinet/. [Online; retrieved Dec. 31, 2018].

[7] Lempel-Ziv-Oberhumer (LZO). http://www.oberhumer.com/opensource/lzo/. [Online; retrieved June 12, 2018].

[8] LevelDB. https://github.com/google/leveldb/. [Online; retrieved Dec. 31, 2018].

[9] LZMA SDK. https://www.7-zip.org/sdk.html. [Online; retrieved Jan. 21, 2019].

[10] MongoDB. https://www.mongodb.com/. [Online; retrieved Dec. 31, 2018].

[11] Phoronix Test Suite. http://www.phoronix-test-suite.com/. [Online; retrieved May 29, 2018].

[12] Redis. https://redis.io/. [Online; retrieved Dec. 31, 2018].

[13] RUBiS. http://rubis.ow2.org/. [Online; retrieved June 11, 2018].

[14] SPECjbb2005. http://spec.org/jbb2005/. [Online; retrieved June 11, 2018].

[15] SPECweb99. http://www.spec.org/web99/. [Online; retrieved June 11, 2018].

[16] SuperFastHash. http://www.azillionmonkeys.com/qed/hash.html. [Online; retrieved June 8, 2018].

[17] Sysbench Benchmark Suite. https://github.com/akopytov/sysbench. [Online; retrieved June 12, 2018].

[18] The x86 KVM Shadow MMU. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/virtual/kvm/mmu.txt. [Online; retrieved May 15, 2018].

[19] *Advanced Configuration and Power Interface (ACPI) Specification - Version 6.3*, Jan. 2019.

[20] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the international Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'06, pages 2–13, San Jose, California, USA, Oct. 2006. URL: https://doi.org/10.1145/1168857.1168860.

[21] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual - Volume 2: System Programming*, Dec. 2017.

[22] O. Agesen, A. Garthwaite, J. Sheldon, and P. Subrahmanyam. The evolution of an x86 virtual machine monitor. *ACM SIGOPS Operating Systems Review*, 44(4):3–18, 2010. URL: https://doi.org/10.1145/1899928.1899930.

[23] O. Agesen, J. Mattson, R. Rugina, and J. Sheldon. Software techniques for avoiding hardware virtualization exits. In *Proceedings of the USENIX Annual Technical Conference*, USENIX ATC'12, pages 373–385, Boston, Massachusetts, USA, June 2012. URL: https://www.usenix.org/system/files/conference/atc12/atc12-final158.pdf.

[24] R. W. Ahmad, A. Gani, S. H. A. Hamid, M. Shiraz, F. Xia, and S. A. Madani. Virtual machine migration in cloud data centers: a review, taxonomy, and open research issues. *The Journal of Supercomputing*, 71(7):2473–2515, July 2015. URL: https://doi.org/10.1007/s11227-015-1400-5.

[25] J. H. Ahn, S. Li, O. Seongil, and N. P. Jouppi. McSimA+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, ISPASS'13, pages 74–85, Austin, Texas, USA, Apr. 2013. URL: https://doi.org/10.1109/ISPASS.2013.6557148.

[26] S. Akiyama, T. Hirofuchi, R. Takano, and S. Honiden. Fast wide area live migration with a low overhead through page cache teleportation. In *Proceedings of the International Symposium on Cluster, Cloud and Grid Computing*, CCGrid'13, pages 78–82, Delft, Netherlands, May 2013. URL: https://doi.org/10.1109/CCGrid.2013.57.

[27] S. Al-Kiswany, D. Subhraveti, P. Sarkar, and M. Ripeanu. VMFlock: Virtual machine co-migration for the cloud. In *Proceedings of the International Symposium on High Performance Distributed Computing*, HPDC'11, pages 159–170, San Jose, California, USA, June 2011. URL: https://doi.org/10.1145/1996130.1996153.

[28] A. R. Alameldeen, M. M. Martin, C. J. Mauer, K. Moore, M. Xu, M. D. Hill, D. A. Wood, and D. J. Sorin. Simulating a $2m commercial server on a $2k pc. *Computer*, 36(2):50–57, 2003. URL: https://doi.org/10.1109/MC.2003.1178046.

[29] A. R. Alameldeen and D. A. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, HPCA-9'03, pages 7–18, Anaheim, California, USA, Feb. 2003. URL: https://doi.org/10.1109/hpca.2003.1183520.

[30] G. Altekar and I. Stoica. Output-deterministic replay for multicore debugging. Technical Report UCB/EECS-2009-108, University of California at Berkeley, EECS Department, Aug. 2009. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-108.html.

[31] B. Amstadt and M. K. Johnson. Wine. *Linux Journal*, 1994(4es):3, Aug. 1994.

[32] ARM Limited. *ARM architecture reference manual - ARMv8, for ARMv8-A architecture profile*, Dec. 2017.

[33] J. W. Atwood. A classification of reliable multicast protocols. *IEEE Network*, 18(3):24–34, May 2004. URL: https://doi.org/10.1109/MNET.2004.1301019.

[34] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI'00, pages 1–12, Vancouver, British Columbia, Canada, June 2000. URL: https://doi.org/10.1145/349299.349303.

[35] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP'03, pages 164–177, Bolton Landing, New York, USA, Oct. 2003. URL: https://doi.org/10.1145/945445.945462.

[36] S. K. Barker, T. Wood, P. J. Shenoy, and R. K. Sitaraman. An empirical study of memory sharing in virtual machines. In *Proceedings of the USENIX Annual Technical Conference*, USENIX ATC'12, pages 273–284, Boston, Massachusetts, USA, June 2012. URL: https://www.usenix.org/system/files/conference/atc12/atc12-final226.pdf.

[37] N. Baudis. Deduplicating virtual machine checkpoints for distributed system simulation. Bachelor's thesis, Karlsruhe Institute of Technology (KIT), Operating Systems Group, Nov. 2013.

[38] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, USENIX ATC'05, pages 41–46, Anaheim, California, USA, Apr. 2005. URL: https://www.usenix.org/legacy/event/usenix05/tech/freenix/full_papers/bellard/bellard.pdf.

[39] N. Bhatia. Performance evaluation of Intel EPT hardware assist. Technical report, VMware, Inc, Mar. 2009. URL: https://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf.

[40] N. Bila, E. de Lara, K. Joshi, H. A. Lagar-Cavilla, M. Hiltunen, and M. Satyanarayanan. Jettison: Efficient idle desktop consolidation with partial VM migration. In *Proceedings of the European Conference on Computer Systems*, EuroSys'12, pages 211–224, Bern, Switzerland, Apr. 2012. URL: https://doi.org/10.1145/2168836.2168858.

[41] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, May 2011. URL: https://doi.org/10.1145/2024716.2024718.

[42] N. Boehr. Evaluating copy-on-write for high frequency checkpoints. Bachelor's thesis, Karlsruhe Institute of Technology (KIT), Operating Systems Group, Sept. 2015.

[43] E. B. Boyer, M. C. Broomfield, and T. A. Perrotti. GlusterFS one storage server to rule them all. Technical Report LA-UR-12-23586, Los Alamos National Lab, July 2012.

[44] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of the International Conference on Virtual Execution Environments*, VEE'07, pages 169–179, San Diego, California, USA, June 2007. URL: https://doi.org/10.1145/1254810.1254834.

[45] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 14(1):80–107, 1996. URL: https://doi.org/10.1145/225535.225538.

[46] K. Buchacker and V. Sieh. Framework for testing the fault-tolerance of systems including OS and network aspects. In *Proceedings of the International Symposium on High Assurance Systems Engineering*, HASE'01, pages 95–105, Oct. 2001. URL: https://doi.org/10.1109/HASE.2001.966811.

[47] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[48] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 15(4):412–447, Nov. 1997. URL: https://doi.org/10.1145/265924.265930.

[49] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang. Bringing virtualization to the x86 architecture with the original VMware workstation. *ACM Transactions on Computer Systems (TOCS)*, 30(4):12, Nov. 2012. URL: https://doi.org/10.1145/2382553.2382554.

[50] A. Burtsev, D. Johnson, M. Hibler, E. Eide, and J. Regehr. Abstractions for practical virtual machine replay. In *Proceedings of the International Conference on Virtual Execution Environments*, VEE'16, pages 93–106, Atlanta, Georgia, USA, Apr. 2016. URL: https://doi.org/10.1145/2892242.2892257.

[51] J. P. Buzen and U. O. Gagliardi. The evolution of virtual machine architecture. In *Proceedings of the National Computer Conference and Exposition*, AFIPS'73, pages 291–299, New York, New York, USA, June 1973. URL: https://doi.org/10.1145/1499586.1499667.

[52] J. Caballero, G. Grieco, M. Marron, and A. Nappa. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA'12, pages 133–143, Minneapolis, Minnesota, USA, July 2012. URL: https://doi.org/10.1145/2338965.2336769.

[53] H. W. Cain, K. M. Lepak, B. A. Schwartz, and M. H. Lipasti. Precise and accurate processor simulation. In *Workshop on Computer Architecture Evaluation using Commercial Workloads*, CAECW'02, pages 13–22, Boston, Massachusettes, USA, Feb. 2002.

[54] K. Chanchio, C. Leangsuksun, H. Ong, V. Ratanasamoot, and A. Shafi. An efficient virtual machine checkpointing mechanism for hypervisor-based HPC systems. In *High Availability and Performance Computing Workshop*, 2008.

[55] C.-J. Chang, J.-J. Wu, W.-C. Hsu, P. Liu, and P.-C. Yew. Efficient memory virtualization for cross-ISA system mode emulation. In *Proceedings of the International Conference on Virtual Execution Environments*, VEE'14, pages 117–128, Salt Lake City, Utah, USA, Mar. 2014. URL: https://doi.org/10.1145/2576195.2576201.

[56] J. B. Chen and B. N. Bershad. The impact of operating system structure on memory system performance. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP'93, pages 120–133, Asheville, North Carolina, USA, Dec. 1994. URL: https://doi.org/10.1145/168619.168629.

[57] P. M. Chen and B. D. Noble. When virtual is better than real. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS'01, pages 133–138, Elmau, Germany, May 2001. URL: https://doi.org/10.1109/HOTOS.2001.990073.

[58] S. Chen. Direct SMARTS: Accelerating microarchitectural simulation through direct execution. Master's thesis, Carnegie Mellon, Computer Architecture Lab, May 2004.

[59] Y. Chen and H. Chen. Scalable deterministic replay in a parallel full-system emulator. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPoPP'13, pages 207–218, Shenzhen, China, Feb. 2013. URL: https://doi.org/10.1145/2442516.2442537.

[60] Y. Chen, S. Zhang, Q. Guo, L. Li, R. Wu, and T. Chen. Deterministic replay: A survey. *ACM Computing Surveys (CSUR)*, 48(2):17, Nov. 2015. URL: https://doi.org/10.1145/2790077.

[61] A. Chernoff and R. Hookway. DIGITAL FX!32 running 32-bit x86 applications on Alpha NT. In *Proceedings of the USENIX Windows NT Workshop*, Seattle, Washington, USA, Aug. 1997. URL: https://www.usenix.org/legacy/publications/library/proceedings/usenix-nt97/full_papers/chernoff/chernoff.pdf.

[62] J.-H. Chiang, H.-L. Li, and T.-c. Chiueh. Introspection-based memory deduplication and migration. In *Proceedings of the International Conference on Virtual Execution Environments*, VEE'13, pages 51–62, Houston, Texas, USA, Mar. 2013. URL: https://doi.org/10.1145/2451512.2451525.

[63] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of the USENIX Annual Technical Conference*, USENIX ATC'08, pages 1–14, Boston, Massachusetts, USA, June 2008. URL: https://www.usenix.org/legacy/event/usenix08/tech/full_papers/chow/chow.pdf.

[64] J. Chow, D. Lucchetti, T. Garfinkel, G. Lefebvre, R. Gardner, J. Mason, S. Small, and P. M. Chen. Multi-stage replay with Crosscut. In *Proceedings of the International Conference on Virtual Execution Environments*, VEE'10, pages 13–24, Pittsburgh, Pennsylvania, USA, Mar. 2010. URL: https://doi.org/10.1145/1735997.1736002.

[65] C. Cifuentes and K. J. Gough. Decompilation of binary programs. *Software: Practice and Experience*, 25(7):811–829, July 1995. URL: https://doi.org/10.1002/spe.4380250706.

[66] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the Symposium on Networked Systems Design & Implementation*, NSDI'05, pages 273–286, Boston, Massachusetts, USA, May 2005. URL: https://www.usenix.org/legacy/events/nsdi05/tech/full_papers/clark/clark.pdf.

[67] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Fast Simulation of Computer Architectures*, pages 5–46. Springer, 1995. URL: https://doi.org/10.1007/978-1-4615-2361-1_2.

[68] Y. Collet et al. Lz4: Extremely fast compression algorithm. https://lz4.github.io/lz4/, Apr. 2011. [Online; retrieved Jan. 21, 2019].

[69] L. Cui, J. Li, B. Li, J. Huai, C. Hu, T. Wo, H. Al-Aqrabi, and L. Liu. VM-Scatter: Migrate virtual machines to many hosts. In *Proceedings of the International Conference on Virtual Execution Environments*, VEE'13, pages 63–72, Houston, Texas, USA, Mar. 2013. URL: https://doi.org/10.1145/2451512.2451528.

[70] L. Cui, T. Wo, B. Li, J. Li, B. Shi, and J. Huai. Pars: A page-aware replication system for efficiently storing virtual machine snapshots. In *Proceedings of the International Conference on Virtual Execution Environments*, VEE'15, pages 215–228, Istanbul, Turkey, Mar. 2015. URL: https://doi.org/10.1145/2731186.2731190.

[71] T. Cui, H. Jin, X. Liao, and H. Liu. A virtual machine replay system based on para-virtualized Xen. In *Proceedings on the International Conference on Network and Parallel Computing*, NPC'09, pages 44–50, Gold Coast, Queensland, Australia, Oct. 2009. URL: https://doi.org/10.1109/NPC.2009.29.

[72] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 161–174, San Francisco, California, USA, Apr. 2008. URL: https://www.usenix.org/legacy/events/nsdi08/tech/full_papers/cully/cully.pdf.

[73] C. Dall and J. Nieh. KVM/ARM: Experiences building the linux ARM hypervisor. Technical Report CUCS-010-13, Columbia University, Department of Computer Science, June 2013. URL: https://doi.org/10.7916/D8FN1FQS.

[74] C. Dall and J. Nieh. KVM/ARM: the design and implementation of the Linux ARM hypervisor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'14, pages 333–348, Salt Lake City, Utah, USA, Mar. 2014. URL: https://doi.org/10.1145/2541940.2541946.

[75] A. d'Antras, C. Gorgovan, J. Garside, J. Goodacre, and M. Luján. HyperMAMBO-X64: Using virtualization to support high-performance transparent binary translation. In *Proceedings of the International Conference on Virtual Execution Environments*, VEE'17, pages 228–241, Xi'an, China, Apr. 2017. URL: https://doi.org/10.1145/3050748.3050756.

[76] D. A. de Oliveira, J. R. Crandall, G. Wassermann, S. F. Wu, Z. Su, and F. T. Chong. ExecRecorder: VM-based full-system replay for attack analysis and system recovery. In *Proceedings of the Workshop on Architectural and System Support for Improving Software Dependability*, ASID'06, pages 66–71, San Jose, California, USA, Oct. 2006. URL: https://doi.org/10.1145/1181309.1181320.

[77] P. J. Denning. Virtual memory. *ACM Computing Surveys (CSUR)*, 2(3):153–189, Sept. 1970. URL: https://doi.org/10.1145/356571.356573.

[78] P. J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6(1):64–84, Jan. 1980. URL: https://doi.org/10.1109/TSE.1980.230464.

[79] U. Deshpande, X. Wang, and K. Gopalan. Live gang migration of virtual machines. In *Proceedings of the International Symposium on High Performance Distributed Computing*, HPDC'11, pages 135–146, San Jose, California, USA, June 2011. URL: https://doi.org/10.1145/1996130.1996151.

[80] R. Desikan, D. Burger, and S. W. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of the International Symposium on Computer Architecture*, ISCA'11, pages 266–277, Göteborg, Sweden, July 2001. URL: https://doi.org/10.1145/379240.565338.

[81] J.-H. Ding, P.-C. Chang, W.-C. Hsu, and Y.-C. Chung. PQEMU: A parallel system emulator based on QEMU. In *Proceedings of the International Conference on Parallel and Distributed Systems*, ICPADS'11, pages 276–283, Tainan, Taiwan, Dec. 2011. URL: https://doi.org/10.1109/icpads.2011.102.

[82] J.-H. Ding, C.-J. Lin, P.-H. Chang, C.-H. Tsang, W.-C. Hsu, and Y.-C. Chung. ARMvisor: System virtualization for ARM. In *Proceedings of the Ottawa Linux Symposium*, OLS'12, pages 93–107, Ottawa, Ontario, Canada, July 2012. URL: https://www.kernel.org/doc/ols/2012/ols2012-chang.pdf.

[83] B. Dolan-Gavitt, T. Leek, J. Hodosh, and W. Lee. Tappan zee (north) bridge: mining memory accesses for introspection. In *Proceedings of the Conference on Computer & Communications Security*, CCS'13, pages 839–850, Berlin, Germany, Nov. 2013. URL: https://doi.org/10.1145/2508859.2516697.

[84] X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, and Y. Xie. Leveraging 3D PCRAM technologies to reduce checkpoint overhead for future exascale systems. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC'09, page 57, Portland, Oregon, USA, Nov. 2009. URL: https://doi.org/10.1145/1654059.1654117.

[85] P. Dovgalyuk. Deterministic replay of system's execution with multi-target QEMU simulator for dynamic analysis and reverse debugging. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, CSMR'12, pages 553–556, Szeged, Hungary, Mar. 2012. URL: https://doi.org/10.1109/CSMR.2012.74.

[86] Y. Du and H. Yu. Paratus: Instantaneous failover via virtual machine replication. In *Proceedings of the International Conference on Grid and Cooperative Computing*, GCC'09, pages 307–312, Lanzhou, Gansu, China, Aug. 2009. URL: https://doi.org/10.1109/GCC.2009.58.

[87] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, OSDI'02, pages 211–224, Boston, Massachusetts, USA, Dec. 2002. URL: https://doi.org/10.1145/844128.844148.

[88] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the International Conference on Virtual Execution Environments*, VEE'08, pages 121–130, Seattle, Washington, USA, Mar. 2008. URL: https://doi.org/10.1145/1346256.1346273.

[89] B. Eicher. Virtual machine checkpoint storage and distribution for SimuBoost. Master's thesis, Karlsruhe Institute of Technology (KIT), Operating Systems Group, Sept. 2015.

[90] M. Ekman and P. Stenstrom. A robust main-memory compression scheme. In *Proceedings of the International Symposium on Computer Architecture*, ISCA'05, pages 74–85, Madison, Wisconsin USA, June 2005. URL: https://doi.org/10.1109/ISCA.2005.6.

[91] F. A. Endo, D. Couroussé, and H.-P. Charles. Micro-architectural simulation of in-order and out-of-order ARM microprocessors with gem5. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS'14, pages 266–273, Agios Konstantinos, Greece, July 2014. URL: https://doi.org/10.1109/SAMOS.2014.6893220.

[92] A. Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. http://www.agner.org/optimize/instruction_tables.pdf, 2018. [Online; retrieved Apr. 16, 2018].

[93] A. Fog. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. http://www.agner.org/optimize/microarchitecture.pdf, 2018. [Online; retrieved Apr. 13, 2018].

[94] J. Fotheringham. Dynamic storage allocation in the Atlas computer, including an automatic use of a backing store. *Communications of the ACM*, 4(10):435–436, Oct. 1961. URL: https://doi.org/10.1145/366786.366800.

[95] S.-Y. Fu, J.-J. Wu, and W.-C. Hsu. Improving SIMD code generation in QEMU. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition*, DATE'15, pages 1233–1236, Grenoble, France, Mar. 2015. URL: https://doi.org/10.7873/date.2015.0356.

[96] J.-l. Gailly and M. Adler. Zlib compression library. 2004.

[97] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Symposium on Network and Distributed System Security*, NDSS'03, pages 191–206, San Diego, California, USA, Feb. 2003.

[98] R. Garg. DéjàVu: Live record-replay of virtual machines for malware analysis. Technical report, Northeastern University, Khoury College of Computer Sciences, May 2012. URL: http://www.ccs.northeastern.edu/home/rohgarg/cs5650sp12-14457303-235210-130.pdf.

[99] T. Garnett. *Dynamic optimization of IA-32 applications under DynamoRIO*. PhD thesis, Massachusetts Institute of Technology, May 2003.

[100] B. Gerofi, Z. Vass, and Y. Ishikawa. Utilizing memory content similarity for improving the performance of replicated virtual machines. In *Proceedings of International Conference on Utility and Cloud Computing*, UCC'11, pages 73–80, Victoria, New South Wales, Australia, Dec. 2011. URL: https://doi.org/10.1109/UCC.2011.20.

[101] S. Girbal, G. Mouchard, A. Cohen, and O. Temam. DiST: A simple, reliable and scalable method to significantly reduce processor architecture simulation time. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS'03, pages 1–12, San Diego, California, USA, June 2003. URL: https://doi.org/10.1145/781027.781029.

[102] R. P. Goldberg. Architecture of virtual machines. In *Proceedings of the Workshop on Virtual Computer Systems*, pages 74–112, Cambridge, Massachusetts, USA, Mar. 1973. ACM. URL: https://doi.org/10.1145/800122.803950.

[103] R. P. Goldberg. Survey of virtual machine research. *Computer*, 7(6):34–45, June 1974. URL: https://doi.org/10.1109/MC.1974.6323581.

[104] J. Gray. Why do computers stop and what can be done about it? In *Proceedings of the Symposium on Reliability in Distributed Software and Database Systems*, SRDS'86, pages 3–12, Los Angeles, California, USA, Jan. 1986. URL: https://www.hpl.hp.com/techreports/tandem/TR-85.7.pdf.

[105] T. Gröninger. On statistical properties of duplicate memory pages. Diploma thesis, Karlsruhe Institute of Technology (KIT), Operating Systems Group, Oct. 2013.

[106] P. H. Gum. System/370 extended architecture: facilities for virtual machines. *IBM Journal of Research and Development*, 27(6):530–544, 1983. URL: https://doi.org/10.1147/rd.276.0530.

[107] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference Engine: Harnessing memory redundancy in virtual machines. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, OSDI'08, pages 85–93, San Diego, California, USA, Dec. 2008. URL: https://doi.org/10.1145/1831407.1831429.

[108] S. Hacking and B. Hudzia. Improving the live migration process of large enterprise applications. In *Proceedings of the International Workshop on Virtualization Technologies in Distributed Computing*, VTDC'09, pages 51–58, Barcelona, Spain, June 2009. URL: https://doi.org/10.1145/1555336.1555346.

[109] G. Hamerly, E. Perelman, J. Lau, and B. Calder. SimPoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism*, 7(4):1–28, 2005. URL: http://cseweb.ucsd.edu/~calder/papers/JILP-05-SimPoint3.pdf.

[110] S. Hassani, G. Southern, and J. Renau. LiveSim: Going live with microarchitecture simulation. In *Proceedings of the International Symposium on High Performance Computer Architecture*, HPCA'16, pages 606–617, Barcelona, Spain, Mar. 2016. URL: https://doi.org/10.1109/hpca.2016.7446098.

[111] P. Heidelberger and H. S. Stone. Parallel trace-driven cache simulation by time partitioning. In *Proceedings of the Winter 1990 Simulation Conference*, pages 734–737, New Orleans, Louisiana, USA, Dec. 1990. URL: https://doi.org/10.1109/wsc.1990.129605.

[112] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann, fourth edition, 2007. ISBN: 978-0123704900.

[113] M. R. Hines and K. Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the International Conference on Virtual Execution Environments*, VEE'09, pages 51–60, Washington, D.C., USA, Mar. 2009. URL: https://doi.org/10.1145/1508293.1508301.

[114] T. Hirofuchi, H. Nakada, S. Itoh, and S. Sekiguchi. Reactive Cloud: Consolidating virtual machines with postcopy live migration. *Information and Media Technologies*, 7(2):614–626, 2012. URL: https://doi.org/10.11185/imt.7.614.

[115] D.-Y. Hong, C.-C. Hsu, P.-C. Yew, J.-J. Wu, W.-C. Hsu, P. Liu, C.-M. Wang, and Y.-C. Chung. HQEMU: a multi-threaded and retargetable dynamic binary translator on multicores. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO'12, pages 104–113, San Jose, California, USA, Mar. 2012. URL: https://doi.org/10.1145/2259016.2259030.

[116] R. N. Horspool and N. Marovac. An approach to the problem of detranslation of computer programs. *The Computer Journal*, 23(3):223–229, Aug. 1980. URL: https://doi.org/10.1093/comjnl/23.3.223.

[117] K.-Y. Hou, K. G. Shin, and J.-L. Sung. Application-assisted live migration of virtual machines with Java applications. In *Proceedings of the European Conference on Computer Systems*, EuroSys'15, page 15, Bordeaux, France, Apr. 2015. URL: https://doi.org/10.1145/2741948.2741950.

[118] K.-Y. Hou, K. G. Shin, Y. Turner, and S. Singhal. Tradeoffs in compressing virtual machine checkpoints. In *Proceedings of the International Workshop on Virtualization Technologies in Distributed Computing*, VTDC'13, pages 41–48, New York, USA, June 2013. URL: https://doi.org/10.1145/2465829.2465834.

[119] C.-C. Hsu, P. Liu, C.-M. Wang, J.-J. Wu, D.-Y. Hong, P.-C. Yew, and W.-C. Hsu. LnQ: Building high performance dynamic binary translators with existing compiler backends. In *The International Conference on Parallel Processing*, ICPP'11, pages 226–234, Taipei City, Taiwan, Sept. 2011. URL: https://doi.org/10.1109/ICPP.2011.57.

[120] C.-C. Hsu, P. Liu, J.-J. Wu, P.-C. Yew, D.-Y. Hong, W.-C. Hsu, and C.-M. Wang. Improving dynamic binary optimization through early-exit guided code region formation. In *Proceedings of the International Conference on Virtual Execution Environments*, VEE'13, pages 23–32, Houston, Texas, USA, Mar. 2013. URL: https://doi.org/10.1145/2451512.2451519.

[121] J. Huang and T.-C. Peng. Analysis of x86 instruction set usage for DOS/Windows applications and its implication on superscalar design. *IEICE Transactions on Information and Systems*, 85(6):929–939, June 2002.

[122] W. Huang. Introduction of AMD advanced virtual interrupt controller. XenSummit, San Diego, California, USA, Aug. 2012.

[123] K. Z. Ibrahim, S. Hofmeyr, C. Iancu, and E. Roman. Optimized pre-copy live migration for memory intensive applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'11, page 40, Seattle, Washington, Nov. 2011. URL: https://doi.org/10.1145/2063384.2063437.

[124] IEEE. *Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7*, Jan. 2018.

[125] Intel. *Intel 64 and IA-32 architectures software developer's manual - combined volumes*, Dec. 2017.

[126] C. Irvin and J. Robin. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *Proceedings of USENIX Security Symposium*, USENIX SEC'00, pages 129–144, Denver, Colorado, USA, Aug. 2000. URL: https://www.usenix.org/legacy/events/sec00/full_papers/robin/robin.pdf.

[127] E. Itskova. Echo: A deterministic record/replay framework for debugging multithreaded applications. Technical report, Imperial College, London, June 2006. URL: http://www3.imperial.ac.uk/pls/portallive/docs/1/18619711.PDF.

[128] S. Jaffer, P. Kedia, and S. Bansal. Improving remote desktopping through adaptive record/replay. In *Proceedings of the International Conference on Virtual Execution Environments*, VEE'15, pages 161–172, Istanbul, Turkey, Mar. 2015. URL: https://doi.org/10.1145/2731186.2731193.

[129] N. Jia, C. Yang, J. Wang, D. Tong, and K. Wang. SPIRE: improving dynamic binary translation through SPC-indexed indirect branch redirecting. In *Proceedings of the International Conference on Virtual Execution Environments*, VEE'13, pages 1–12, Houston, Texas, USA, Mar. 2013. URL: https://doi.org/10.1145/2451512.2451516.

[130] H. Jin, L. Deng, S. Wu, X. Shi, and X. Pan. Live virtual machine migration with adaptive, memory compression. In *Proceedings of the International Conference on Cluster Computing*, CLUSTER'09, pages 1–10, New Orleans, Louisiana, USA, Aug. 2009. URL: https://doi.org/10.1109/CLUSTR.2009.5289170.

[131] C. Jo, E. Gustafsson, J. Son, and B. Egger. Efficient live migration of virtual machines using shared storage. In *Proceedings of the International Conference on Virtual Execution Environments*, VEE'13, pages 41–50, Houston, Texas, USA, Mar. 2013. URL: https://doi.org/10.1145/2451512.2451524.

[132] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: Monitoring the buffer cache in a virtual machine environment. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'06, pages 14–24, San Jose, California, USA, Oct. 2006. URL: https://doi.org/10.1145/1168857.1168861.

[133] M. Jurczyk and G. Coldwind. Bochspwn: Identifying and exploiting windows kernel race conditions via memory access patterns. In *The Symposium on Security for Asia Network*, SyScan'13, Singapore, Sept. 2013. URL: https://ai.google/research/pubs/pub42189.

[134] A. Kangarlou, P. Eugster, and D. Xu. VNsnap: Taking snapshots of virtual networked environments with minimal downtime. In *Proceedings of the International Conference on Dependable Systems & Networks*, DSN'09, pages 524–533, Lisbon, Portugal, July 2009. URL: https://doi.org/10.1109/DSN.2009.5270298.

[135] S. Kim, F. Liu, Y. Solihin, R. Iyer, L. Zhao, and W. Cohen. Accelerating full-system simulation through characterizing and predicting operating system performance. In *Proceedings of the International Symposium on Performance Analysis of Systems & Software*, ISPASS'07, pages 1–11, San Jose, California, USA, May 2007. URL: https://doi.org/10.1109/ISPASS.2007.363731.

[136] S. T. King, G. W. Dunlap, and P. M. Chen. Operating system support for virtual machines. In *Proceedings of the USENIX Annual Technical Conference, General Track*, USENIX ATC'03, pages 71–84, San Antonio, Texas, USA, June 2003. URL: https://www.usenix.org/legacy/events/usenix03/tech/full_papers/king/king.pdf.

[137] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the USENIX Annual Technical Conference, General Track*, USENIX ATC'05, pages 1–15, Anaheim, California, USA, Apr. 2005. URL: https://www.usenix.org/events/usenix05/tech/general/king/king.pdf.

[138] J. Kiszka. Architecture of the kernel-based virtual machine (KVM). In *17th International Linux System Technology Conference*, Nuremberg, Germany, Sept. 2010. URL: http://www.linux-kongress.org/2010/slides/KVM-Architecture-LK2010.pdf.

[139] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, pages 225–230, Ottawa, Ontario, Canada, June 2007. URL: https://www.kernel.org/doc/ols/2007/ols2007v1-pages-225-230.pdf.

[140] A. KleinOsowski and D. J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1(1):7, 2002. URL: https://doi.org/10.1109/L-CA.2002.8.

[141] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *Symposium on Security and Privacy*, S&P'19, San Francisco, California, USA, May 2019. URL: https://arxiv.org/pdf/1801.01203.

[142] R. Koller and R. Rangaswami. I/O Deduplication: Utilizing content similarity to improve I/O performance. *ACM Transactions on Storage (TOS)*, 6(3):13, Sept. 2010. URL: https://doi.org/10.1145/1837915.1837921.

[143] A. Koto, H. Yamada, K. Ohmura, and K. Kono. Towards unobtrusive VM live migration for cloud computing platforms. In *Proceedings of the Asia-Pacific Workshop on Systems*, APSYS'12, page 7, Seoul, Republic of Korea, July 2012. URL: https://doi.org/10.1145/2349896.2349903.

[144] M. Kozuch, M. Satyanarayanan, T. Bressoud, and Y. Ke. Efficient state transfer for Internet suspend/resume. Technical Report IRP-TR-02-03, Intel Research Pittburgh, Apr. 2002. URL: http://www.cs.cmu.edu/afs/cs.cmu.edu/user/satya/Web/docdir/kozuch-irp-tech-report-may-2002.pdf.

[145] C. Krintz and R. Wolski. Using phase behavior in scientific application to guide Linux operating system customization. In *19th International Symposium on Parallel and Distributed Processing*, IPDPS'05, page 219, Denver, Colorado, USA, Apr. 2005. URL: https://doi.org/10.1109/IPDPS.2005.449.

[146] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. De Lara, M. Brudno, and M. Satyanarayanan. SnowFlock: Rapid virtual machine cloning for cloud computing. In *Proceedings of the European Conference on Computer Systems*, EuroSys'09, pages 1–12, Nuremberg, Germany, Apr. 2009. URL: https://doi.org/10.1145/1519065.1519067.

[147] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems (TOCS)*, 10(4):265–310, Nov. 1992. URL: https://doi.org/10.1145/138873.138874.

[148] R. Lantz. Fast functional simulation with parallel Embra. In *Proceedings of the Workshop on Modeling, Benchmarking and Simulation*, MoBS'08, page 5, Beijing, China, June 2008. URL: https://www-cs.stanford.edu/~rlantz/papers/lantz-mobs08-final.pdf.

[149] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO'04, pages 75–88, Palo Alto, California, USA, Mar. 2004. URL: https://doi.org/10.1109/cgo.2004.1281665.

[150] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely. PEBIL: Efficient static binary instrumentation for Linux. In *Proceedings of the International Symposium on Performance Analysis of Systems & Software*, ISPASS'10, pages 175–183, White Plains, New York, USA, Mar. 2010. URL: https://doi.org/10.1109/ISPASS.2010.5452024.

[151] G. Lauterbach. Accelerating architectural simulation by parallel execution of trace samples. In *Proceedings of the Hawaii International Conference on System Sciences*, HICSS'94, Wailea, Hawaii, USA, Apr. 1994. URL: https://doi.org/10.1109/hicss.1994.323171.

[152] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with Instant Replay. Technical Report ADA179902, Rochester University, Department of Computer Science, Sept. 1986. URL: https://apps.dtic.mil/dtic/tr/fulltext/u2/a179902.pdf.

[153] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: Efficient online multiprocessor replay via speculation and external determinism. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'10, pages 77–90, Pittsburgh, Pennsylvania, USA, Mar. 2010. URL: https://doi.org/10.1145/1736020.1736031.

[154] S. Li. Linux v4.18 Page Aging. https://github.com/torvalds/linux/blob/v4.18/arch/x86/mm/pgtable.c#L522. [Online; retrieved Mar. 4, 2019].

[155] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java virtual machine specification*. Oracle, twelfth edition, Feb. 2019.

[156] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium*, USENIX Security'18, Baltimore, Maryland, USA, Aug. 2018. URL: https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-lipp.pdf.

[157] H. Liu, H. Jin, X. Liao, L. Hu, and C. Yu. Live migration of virtual machine based on full system trace and replay. In *Proceedings of the International Symposium on High Performance Distributed Computing*, HPDC'09, pages 101–110, Garching, Germany, June 2009. URL: https://doi.org/10.1145/1551609.1551630.

[158] P. Liu, Z. Yang, X. Song, Y. Zhou, H. Chen, and B. Zang. Heterogeneous live migration of virtual machines. In *International Workshop on Virtualization Technology*, IWVT'08, 2008.

[159] G. H. Loh, S. Subramaniam, and Y. Xie. Zesto: A cycle-level simulator for highly detailed microarchitecture exploration. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, ISPASS'09, pages 53–64, Boston, Massachusetts, USA, Apr. 2009. URL: https://doi.org/10.1109/ISPASS.2009.4919638.

[160] M. Lu and T.-c. Chiueh. Fast memory state synchronization for virtualization-based fault tolerance. In *Proceedings of the International Conference on Dependable Systems & Networks*, DSN'09, pages 534–543, Lisbon, Portugal, Sept. 2009. URL: https://doi.org/10.1109/DSN.2009.5270295.

[161] P. Lu, B. Ravindran, and C. Kim. VPC: Scalable, low downtime checkpointing for virtual clusters. In *Proceedings of the International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD'12, pages 203–210, New York, New York, USA, Oct. 2012. URL: https://doi.org/10.1109/SBAC-PAD.2012.31.

[162] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI'05, pages 190–200, Chicago, Illinois, USA, June 2005. URL: https://doi.org/10.1145/1065010.1065034.

[163] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, Aug. 2002. URL: https://doi.org/10.1109/2.982916.

[164] P. S. Magnusson, F. Larsson, A. Moestedt, B. Werner, J. Nilsson, P. Stenström, F. Lundholm, M. Karlsson, F. Dahlgren, and H. Grahn. Simics/sun4m: A virtual workstation. In *Proceedings of the USENIX Annual Technical Conference*, USENIX'98, pages 119–130, New Orleans, Louisiana, USA, June 1998. URL: https://www.usenix.org/legacy/publications/library/proceedings/usenix98/full_papers/magnusson/magnusson.pdf.

[165] P. S. Magnusson and B. Werner. Some efficient techniques for simulating memory. Technical Report SICS-R–94/16–SE, Sept. 1994. URL: http://soda.swedish-ict.se/2136/.

[166] V. V. Malyugin, B. Weissman, G. Venkitachalam, and M. Xu. Synchronizing a translation lookaside buffer with page tables, Mar. 2018. US Patent 9,928,180.

[167] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. Testing CPU emulators. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA'09, pages 261–272, Chicago, Illinois, USA, July 2009. URL: https://doi.org/10.1145/1572272.1572303.

[168] C. May. Mimic: a fast System/370 simulator. In *Proceedings of the Symposium on Interpreters and Interpretive Techniques*, pages 1–13, St. Paul, Minnesota, USA, June 1987. URL: https://doi.org/10.1145/29650.29651.

[169] A. M. G. Maynard, C. M. Donnelly, and B. R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'94, pages 145–156, San Jose, California, USA, Oct. 1994. URL: https://doi.org/10.1145/195473.195524.

[170] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the USENIX Conference on File and Storage Technologies*, FAST'03, pages 115–130, San Francisco, California, USA, Mar. 2003. URL: https://www.usenix.org/legacy/event/fast03/tech/full_papers/megiddo/megiddo.pdf.

[171] T. Merrifield and H. R. Taheri. Performance implications of extended page tables on virtualized x86 processors. In *Proceedings of the International Conference on Virtual Execution Environments*, VEE'16, pages 25–35, Atlanta, Georgia, USA, Apr. 2016. URL: https://doi.org/10.1145/2892242.2892258.

[172] R. A. Meyer and L. H. Seawright. A virtual machine time-sharing system. *IBM Systems Journal*, 9(3):199–218, 1970. URL: http://www.eecs.harvard.edu/~margo/cs261/papers/meyer-1970.pdf.

[173] Microsoft. Hyper-V. https://msdn.microsoft.com/de-de/library/mt169373(v=ws.11).aspx. [Online; retrieved Apr. 4, 2018].

[174] D. Mihocka and S. Shwartsman. Virtualization without direct execution or jitting: Designing a portable virtual machine infrastructure. In *Workshop on Architectural and Microarchitectural Support for Binary Translation*, AMAS-BT'08, Beijing, China, June 2008. URL: http://www.emulators.com/docs/VirtNoJit_Paper.pdf.

[175] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *Proceedings of the International Symposium on High Performance Computer Architecture*, HPCA'10, pages 1–12, Bangalore, India, Jan. 2010. URL: https://doi.org/10.1109/hpca.2010.5416635.

[176] K. Miller. *Efficient Main Memory Deduplication Through Cross Layer Integration*. PhD thesis, Karlsruhe Institute of Technology (KIT), Operating Systems Group, July 2014.

[177] K. Miller, F. Franz, M. Rittinghaus, M. Hillenbrand, and F. Bellosa. XLH: More effective memory deduplication scanners through cross-layer hints. In *Proceedings of the 2013 USENIX Annual Technical Conference*, USENIX ATC'13, pages 279–290, San Jose, California, USA, June 2013.

[178] R. B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the Fall 1968 Joint Computer Conference*, pages 267–277, Dec. 1968.

[179] G. Miłós, D. G. Murray, S. Hand, and M. A. Fetterman. Satori: Enlightened page sharing. In *Proceedings of the USENIX Annual Technical Conference*, USENIX ATC'09, pages 1–14, San Diego, California, USA, June 2009. URL: http://static.usenix.org/events/usenix09/tech/full_papers/milos/milos.pdf.

[180] U. F. Minhas, S. Rajagopalan, B. Cully, A. Aboulnaga, K. Salem, and A. Warfield. RemusDB: Transparent high availability for database systems. *The International Journal on Very Large Data Bases (VLDB)*, 22(1):29–45, Feb. 2013. URL: https://doi.org/10.1007/s00778-012-0294-6.

[181] F. F. Moghaddam and M. Cheriet. Decreasing live virtual machine migration down-time using a memory page selection based on memory change PDF. In *Proceedings of the International Conference on Networking, Sensing and Control*, ICNSC'10, pages 355–359, Chicago, Illinois, USA, Apr. 2010. URL: https://doi.org/10.1109/ICNSC.2010.5461517.

[182] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the International Symposium on Computer Architecture*, ISCA'08, pages 289–300, Beijing, China, June 2008. URL: https://doi.org/10.1109/ISCA.2008.36.

[183] B. Morbach. Accurate record and replay of x86 MMU behavior for SimuBoost. Master's thesis, Karlsruhe Institute of Technology (KIT), Operating Systems Group, Sept. 2018.

[184] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, and R. Peri. Shadow Profiling: Hiding instrumentation costs with parallelism. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO'07, pages 198–208, San Jose, California, USA, Mar. 2007. URL: https://doi.org/10.1109/cgo.2007.35.

[185] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the International Symposium on Computer Architecture*, ISCA'05, pages 284–295, Madison, Wisconsin USA, June 2005. URL: https://doi.org/10.1109/ISCA.2005.16.

[186] S. Nathan, U. Bellur, and P. Kulkarni. Towards a comprehensive performance model of virtual machine live migration. In *Proceedings of the Symposium on Cloud Computing*, SoCC'15, pages 288–301, Kohala Coast, Hawaii, Aug. 2015. URL: https://doi.org/10.1145/2806777.2806838.

[187] S. Nathan, U. Bellur, and P. Kulkarni. On selecting the right optimizations for virtual machine migration. In *Proceedings of the International Conference on Virtual Execution Environments*, VEE'12, pages 37–49, Atlanta, Georgia, USA, Apr. 2016. URL: https://doi.org/10.1145/2892242.2892247.

[188] S. Nathan, P. Kulkarni, and U. Bellur. Resource availability based performance benchmarking of virtual machine migrations. In *Proceedings of the International Conference on Performance Engineering*, ICPE'13, pages 387–398, Prague, Czech Republic, Apr. 2013. URL: https://doi.org/10.1145/2479871.2479932.

[189] M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the USENIX Annual Technical Conference, General Track*, USENIX ATC'05, pages 391–394, Anaheim, California, USA, Apr. 2005. URL: http://static.usenix.org/legacy/events/usenix05/tech/general/full_papers/short_papers/nelson/nelson.pdf.

[190] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI'07, pages 89–100, San Diego, California, USA, June 2007. URL: https://doi.org/10.1145/1250734.1250746.

[191] R. H. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. Technical Report CS-93-15, Brown University, Department of Computer Science, Sept. 1993. URL: ftp://ftp.cs.brown.edu/pub/techreports/93/cs93-15.pdf.

[192] A.-T. Nguyen, P. Bose, K. Ekanadham, A. Nanda, and M. Michael. Accuracy and speed-up of parallel trace-driven architectural simulation. In *Proceedings of the International Parallel Processing Symposium*, IPPS'97, pages 39–44, Geneva, Switzerland, Aug. 1997. URL: https://doi.org/10.1109/ipps.1997.580842.

[193] K. Nguyen. APIC virtualization performance testing and Iozone. https://software.intel.com/en-us/blogs/2013/12/17/apic-virtualization-performance-testing-and-iozone, Dec. 2013. [Online; retrieved May 23, 2018].

[194] O. Oppitz. A particular bug trap: Execution replay using virtual machines. In *Workshop on Automated and Algorithmic Debugging*, AADEBUG'03, Ghent, Belgium, Sept. 2003. URL: https://arxiv.org/pdf/cs/0310030.

[195] Oracle. VirtualBox. http://www.virtualbox.org. [Online; retrieved Apr. 4, 2018].

[196] V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM: An execution-driven simulator for ILP-based shared-memory multiprocessors and uniprocessors. *Technical Committee on Computer Architecture Newsletter*, Oct. 1997. URL: https://scholarship.rice.edu/handle/1911/20182.

[197] D. Z. Pan and M. A. Linton. Supporting reverse execution for parallel programs. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, PADD'88, pages 124–129, Madison, Wisconsin, USA, May 1988. URL: https://doi.org/10.1145/68210.69227.

[198] E. Park, B. Egger, and J. Lee. Fast and space-efficient virtual machine checkpointing. In *Proceedings of the International Conference on Virtual Execution Environments*, VEE'11, pages 75–86, Newport Beach, California, USA, Mar. 2011. URL: https://doi.org/10.1145/1952682.1952694.

[199] A. Patel, F. Afram, and K. Ghose. MARSS-x86: A QEMU-based microarchitectural and systems simulator for x86 multicore processors. In *Proceedings of the International QEMU Users' Forum*, pages 29–30, Grenoble, France, Mar. 2011. URL: http://adt.cs.upb.de/quf/quf11/QUF11-papers/quf2011-10.pdf.

[200] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO'10, pages 2–11, Toronto, Ontario, Canada, Apr. 2010. URL: https://doi.org/10.1145/1772954.1772958.

[201] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, fifth edition, 2013. ISBN: 978-0124077263.

[202] PCI-SIG. *Single Root I/O Virtualization and Sharing Specification, Revision 1.1*, 2010.

[203] N. Penneman, D. Kudinskas, A. Rawsthorne, B. De Sutter, and K. De Bosschere. Formal virtualization requirements for the ARM architecture. *Journal of Systems Architecture*, 59(3):144–154, Mar. 2013. URL: https://doi.org/10.1016/j.sysarc.2013.02.003.

[204] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under UNIX. In *Proceedings of the USENIX Technical Conference*, USENIX TCON'95, New Orleans, Louisiana, USA, Jan. 1995. URL: https://www.usenix.org/legacy/publications/library/proceedings/neworl/plank.html.

[205] G. Pokam, K. Danne, C. Pereira, R. Kassa, T. Kranich, S. Hu, J. Gottschlich, N. Honarmand, N. Dautenhahn, S. T. King, and J. Torrellas. QuickRec: Prototyping an Intel architecture extension for record and replay of multithreaded programs. In *Proceedings of the International Symposium on Computer Architecture*, ISCA'13, pages 643–654, Tel-Aviv, Israel, June 2013. URL: https://doi.org/10.1145/2485922.2485977.

[206] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, July 1974. URL: https://doi.org/10.1145/361011.361073.

[207] A. Portero, A. Scionti, Z. Yu, P. Faraboschi, C. Concatto, L. Carro, A. Garbade, S. Weis, T. Ungerer, and R. Giorgi. Simulating the future kilo x86-64 core processors and their infrastructure. In *Proceedings of the Simulation Symposium*, ANSS'12, Orlando, Florida, USA, Mar. 2012. URL: http://www3.diism.unisi.it/~giorgi/papers/Portero12a.pdf.

[208] A. Pranckevičius. More hash function tests. https://aras-p.info/blog/2016/08/09/More-Hash-Function-Tests/, Aug. 2016. [Online; retrieved Jan. 29, 2019].

[209] A. Pusch. Checkpoint distribution for simuboost. Master's thesis, Karlsruhe Institute of Technology (KIT), Operating Systems Group, Oct. 2017.

[210] W. Qin, J. D'Errico, and X. Zhu. A multiprocessing approach to accelerate retargetable and portable dynamic-compiled instruction-set simulation. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS'06, pages 193–198, Seoul, Korea, Oct. 2006. URL: https://doi.org/10.1145/1176254.1176302.

[211] A. Quinn, D. Devecsery, P. M. Chen, and J. Flinn. JetStream: Cluster-scale parallelization of information flow queries. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, OSDI'16, pages 451–466, Savannah, Georgia, USA, Nov. 2016. URL: https://www.usenix.org/system/files/conference/osdi16/osdi16-quinn.pdf.

[212] T. Raffetseder, C. Kruegel, and E. Kirda. Detecting system emulators. In *Proceedings of the International Conference on Information Security*, ISC'07, pages 1–18, Valparaíso, Chile, Oct. 2007. URL: https://doi.org/10.1007/978-3-540-75496-1_1.

[213] S. Rajagopalan, B. Cully, R. O'Connor, and A. Warfield. SecondSite: Disaster tolerance as a service. In *Proceedings of the Conference on Virtual Execution Environments*, VEE'12, pages 97–108, London, England, Mar. 2012. URL: https://doi.org/10.1145/2151024.2151039.

[214] B. Randell and C. Kuehner. Dynamic storage allocation systems. *Communications of the ACM*, 11(5):297–306, May 1968.

[215] S. Ren, L. Tan, C. Li, Z. Xiao, and W. Song. Samsara: Efficient deterministic replay in multiprocessor environments with hardware virtualization extensions. In *Proceedings of the USENIX Annual Technical Conference*, USENIX ATC'16, pages 551–564, Denver, Colorado, USA, June 2016. URL: https://www.usenix.org/system/files/conference/atc16/atc16_paper-ren.pdf.

[216] P. Riteau, C. Morin, and T. Priol. Shrinker: Improving live migration of virtual clusters over WANs with distributed data deduplication and content-based addressing. In *Proceedings of the European Conference on Parallel Processing*, Euro-Par'11, pages 431–442, Bordeaux, France, Aug. 2011. URL: https://doi.org/10.1007/978-3-642-23400-2_40.

[217] M. Rittinghaus. Runtime benefits of memory deduplication. Diploma thesis, Karlsruhe Institute of Technology (KIT), Operating Systems Group, July 2012.

[218] M. Rittinghaus, T. Groeninger, and F. Bellosa. Simutrace: A toolkit for full system memory tracing. White paper, Karlsruhe Institute of Technology (KIT), Operating Systems Group, May 2015.

[219] M. Rittinghaus, K. Miller, M. Hillenbrand, and F. Bellosa. SimuBoost: Scalable parallelization of functional system simulation. In *Proceedings of the International Workshop on Dynamic Analysis*, WODA'13, Houston, Texas, USA, Mar. 2013.

[220] M. Ronsse and K. De Bosschere. RecPlay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems (TOCS)*, 17(2):133–152, 1999. URL: https://doi.org/10.1145/312203.312214.

[221] M. Rosenblum. The reincarnation of virtual machines. *Queue - Virtual Machines*, 2(5):34, July 2004. URL: https://doi.org/10.1145/1016998.1017000.

[222] M. Rosenblum and T. Garfinkel. Virtual machine monitors: Current technology and future trends. *Computer*, 38(5):39–47, May 2005. URL: https://doi.org/10.1109/MC.2005.176.

[223] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: The SimOS approach. *IEEE Parallel & Distributed Technology: Systems & Applications*, 3(4):34–43, 1995. URL: https://doi.org/10.1109/88.473612.

[224] J. Ruh. Analyzing duplication in incremental high frequency checkpoints. Bachelor's thesis, Karlsruhe Institute of Technology (KIT), Operating Systems Group, Sept. 2015.

[225] J. Ruh. Optimizing continuous checkpoints for deterministic replay. Master's thesis, Karlsruhe Institute of Technology (KIT), Operating Systems Group, July 2018.

[226] R. Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review - Research and Developments in the Linux Kernel*, 42(5):95–103, July 2008. URL: https://doi.org/10.1145/1400097.1400108.

[227] M. Russinovich and B. Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI'96, pages 258–266, Philadelphia, Pennsylvania, USA, May 1996. URL: https://doi.org/10.1145/231379.231432.

[228] Y. Saito. Jockey: A user-space library for record-replay debugging. In *Proceedings of the International Symposium on Automated Analysis-driven Debugging*, AADEBUG'05, pages 69–76, Monterey, California, USA, Sept. 2005. URL: https://doi.org/10.1145/1085130.1085139.

[229] A. Sandberg, N. Nikoleris, T. E. Carlson, E. Hagersten, S. Kaxiras, and D. Black-Schaffer. Full speed ahead: Detailed architectural simulation at near-native speed. In *Proceedings of the International Symposium on Workload Characterization*, IISWC'15, pages 183–192, Atlanta, Georgia, USA, Oct. 2015. URL: https://doi.org/10.1109/IISWC.2015.29.

[230] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, OSDI'02, pages 377–390, Boston, Massachusetts, USA, Dec. 2002. URL: https://doi.org/10.1145/844128.844163.

[231] D. J. Scales, M. Nelson, and G. Venkitachalam. The design of a practical system for fault-tolerant virtual machines. *ACM SIGOPS Operating Systems Review*, 44(4):30–39, 2010. URL: https://doi.org/10.1145/1899928.1899932.

[232] M. Schmidt, N. Fallenbeck, M. Smith, and B. Freisleben. Efficient distribution of virtual machines for cloud computing. In *Proceedings of the International Conference on Parallel, Distributed and Network-Based Processing*, PDP'10, pages 567–574, Pisa, Italy, Feb. 2010. URL: https://doi.org/10.1109/PDP.2010.39.

[233] T. Schmidt. Evaluating techniques for full system memory tracing. Bachelor's thesis, Karlsruhe Institute of Technology (KIT), Operating Systems Group, Oct. 2017.

[234] J. Schoetterl-Glausch. Intel page modification logging for lightweight continuous checkpointing. Bachelor's thesis, Karlsruhe Institute of Technology (KIT), Operating Systems Group, Oct. 2016.

[235] J. Schötterl-Glausch. Exploring pre-scan, parallel copy, and large pages for continuous checkpointing. Master's thesis, Karlsruhe Institute of Technology (KIT), Operating Systems Group, Nov. 2018.

[236] P. Sharma, S. Lee, T. Guo, D. Irwin, and P. Shenoy. SpotCheck: Designing a derivative IaaS cloud on the spot market. In *Proceedings of the European Conference on Computer Systems*, EuroSys'15, page 16, Bordeaux, France, Apr. 2015. URL: https://doi.org/10.1145/2741948.2741953.

[237] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'02, pages 45–57, San Jose, California, USA, Oct. 2002. URL: https://doi.org/10.1145/605397.605403.

[238] S. Shwartsman and D. Mihoka. *How Bochs Works Under the Hood*. Second edition, June 2012.

[239] B. Smith. ARM and Intel battle over the mobile chip's future. *Computer*, 41(5):15–18, May 2008. URL: https://doi.org/10.1109/MC.2008.142.

[240] J. Smith and R. Nair. *Virtual machines: versatile platforms for systems and processes*. Morgan Kaufmann, July 2005. ISBN: 978-1558609105.

[241] X. Song, J. Shi, R. Liu, J. Yang, and H. Chen. Parallelizing live migration of virtual machines. In *Proceedings of the International Conference on Virtual Execution Environments*, VEE'13, pages 85–96, Houston, Texas, USA, Mar. 2013. URL: https://doi.org/10.1145/2451512.2451531.

[242] SPARC International, Inc. *The SPARC Architecture Manual*, eights edition, 1992.

[243] T. Spink, H. Wagstaff, and B. Franke. Hardware-accelerated cross-architecture full-system virtualization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(4):36, Dec. 2016. URL: https://doi.org/10.1145/2996798.

[244] D. Srinivasan and X. Jiang. Time-traveling forensic analysis of VM-based high-interaction honeypots. In *Proceedings of the International Conference on Security and Privacy in Communication Systems*, SecureComm'11, pages 209–226, London, England, Sept. 2011. URL: https://doi.org/10.1007/978-3-642-31909-9_12.

[245] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the USENIX Annual Technical Conference, General Track*, USENIX ATC'04, pages 29–44, Boston, Massachusetts, USA, July 2004. URL: https://www.usenix.org/legacy/events/usenix04/tech/general/full_papers/srinivasan/srinivasan.pdf.

[246] J. Stevens, P. Tschirhart, and B. Jacob. Fast full system memory checkpointing with SSD-aware memory controller. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS'16, pages 96–98, Alexandria, Virginia, USA, Oct. 2016. URL: https://doi.org/10.1145/2989081.2989126.

[247] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *Proceedings of the USENIX Annual Technical Conference, General Track*, USENIX ATC'01, pages 1–14, Boston, Massachusetts, USA, June 2001. URL: http://usenix.org/publications/library/proceedings/usenix01/sugerman/sugerman.pdf.

[248] M. H. Sun and D. M. Blough. Fast, lightweight virtual machine checkpointing. Technical report, Georgia Institute of Technology, 2010. URL: http://hdl.handle.net/1853/36671.

[249] D. Sundmark and H. Thane. Pinpointing interrupts in embedded real-time systems using context checksums. In *Proceedings of the International Conference on Emerging Technologies and Factory Automation*, ETFA'08, pages 774–781, Hamburg, Germany, Sept. 2008. URL: https://doi.org/10.1109/ETFA.2008.4638487.

[250] P. Svärd, B. Hudzia, J. Tordsson, and E. Elmroth. Evaluation of delta compression techniques for efficient live migration of large virtual machines. In *Proceedings of the International Conference on Virtual Execution Environments*, VEE'11, pages 111–120, Newport Beach, California, USA, Mar. 2011. URL: https://doi.org/10.1145/1952682.1952698.

[251] P. Svärd, J. Tordsson, B. Hudzia, and E. Elmroth. High performance live migration through dynamic page transfer reordering and compression. In *Proceedings of the International Conference on Cloud Computing Technology and Science*, CloudCom'11, pages 542–548, Athens, Greece, Nov. 2011. URL: https://doi.org/10.1109/CloudCom.2011.82.

[252] P. Ta-Shma, G. Laden, M. Ben-Yehuda, and M. Factor. Virtual machine time travel using continuous data protection and checkpointing. *ACM SIGOPS Operating Systems Review*, 42(1):127–134, Jan. 2008. URL: https://doi.org/10.1145/1341312.1341341.

[253] Y. Tamura, K. Sato, S. Kihara, and S. Moriai. Kemari: Virtual machine synchronization for fault tolerance. In *USENIX Annual Technical Conference (Poster Session)*, USENIX ATC'08, Boston, Massachusetts, USA, June 2008. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.580.7704&rep=rep1&type=pdf.

[254] A. S. Tanenbaum and H. Bos. *Modern Operating System*. Pearson Education, Inc, fourth edition, 2015. ISBN: 978-1292061429.

[255] M. M. Theimer, K. A. Lantz, and D. R. Cheriton. Preemptable remote execution facilities for the V-system. Technical Report ADA166948, Stanford University, Department of Computer Science, Sept. 1985. URL: https://apps.dtic.mil/dtic/tr/fulltext/u2/a166948.pdf.

[256] X. Tong, T. Koju, M. Kawahito, and A. Moshovos. Optimizing memory translation emulation in full system emulators. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):60, Jan. 2015. URL: https://doi.org/10.1145/2686034.

[257] D. Trendafilov, N. Memon, and T. Suel. zdelta: An efficient delta compression tool. Technical Report CIS-2002-02, Polytechnic University, Brooklyn, Department of Computer and Information Science, June 2002.

[258] J. Tröger and D. Mihocka. Fast microcode interpretation with transactional commit/abort. In *Workshop on Architectural and Microarchitectural Support for Binary Translation*, AMAS-BT'11, San Jose, California, USA, June 2011. URL: http://www.emulators.com/docs/amas-bt2011.pdf.

[259] C.-C. Tu, M. Ferdman, C.-t. Lee, and T.-c. Chiueh. A comprehensive implementation and evaluation of direct interrupt delivery. In *Proceedings of the International Conference on Virtual Execution Environments*, VEE'15, pages 1–15, Istanbul, Turkey, Mar. 2015. URL: https://doi.org/10.1145/2731186.2731189.

[260] R. Uhlig, R. Fishtein, and O. Gershon. SoftSDV: A presilicon software development environment for the IA-64 architecture. *Intel Technology Journal*, Nov. 1999.

[261] G. Vallee, T. Naughton, H. Ong, and S. L. Scott. Checkpoint/restart of virtual machines based on Xen. In *High Availability and Performace Computing Workshop*, HAPCW'06, Santa Fe, New Mexico, USA, Oct. 2006. URL: https://www.cct.lsu.edu/~estrabd/LACSI2006/workshops/workshop1/papers/cr-xen-hapcw06-final.pdf.

[262] S. Veith. Towards heterogeneous record and replay on the ARM architecture. Master's thesis, Karlsruhe Institute of Technology (KIT), Operating Systems Group, Jan. 2017.

[263] N. Viennot, S. Nair, and J. Nieh. Transparent mutable replay for multicore debugging and patch validation. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'13, pages 127–138, Houston, Texas, USA, Mar. 2013. URL: https://doi.org/10.1145/2451116.2451130.

[264] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP'05, pages 148–162, Brighton, United Kingdom, Oct. 2005. URL: https://doi.org/10.1145/1095810.1095825.

[265] C. Waldspurger and M. Rosenblum. I/O virtualization. *Queue - Virtualization*, 9(11):30–39, Nov. 2011. URL: https://doi.org/10.1145/2063166.2071256.

[266] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, OSDI'02, pages 181–194, Boston, Massachusetts, USA, Dec. 2002. URL: https://doi.org/10.1145/844128.844146.

[267] S. Wallace and K. Hazelwood. SuperPin: Parallelizing dynamic instrumentation for real-time performance. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO'07, pages 209–220, San Jose, California, USA, Mar. 2007. URL: https://doi.org/10.1109/cgo.2007.37.

[268] K. Wang, Y. Zhang, H. Wang, and X. Shen. Parallelization of IBM mambo system simulator in functional modes. *ACM SIGOPS Operating Systems Review*, 42(1):71–76, Jan. 2008. URL: https://doi.org/10.1145/1341312.1341325.

[269] L. Wang, Z. Kalbarczyk, R. K. Iyer, and A. Iyengar. VM-μCheckpoint: Design, modeling, and assessment of lightweight in-memory VM checkpointing. *IEEE Transactions on Dependable and Secure Computing*, 12(2):243–255, Mar. 2015. URL: https://doi.org/10.1109/TDSC.2014.2327967.

[270] Z. Wang, J. Li, C. Wu, D. Yang, Z. Wang, W.-C. Hsu, B. Li, and Y. Guan. HSPT: Practical implementation and efficient management of embedded shadow page tables for cross-ISA system virtual machines. In *Proceedings of the International Conference on Virtual Execution Environments*, VEE'15, pages 53–64, Istanbul, Turkey, Mar. 2015. URL: https://doi.org/10.1145/2731186.2731188.

[271] Z. Wang, R. Liu, Y. Chen, X. Wu, H. Chen, W. Zhang, and B. Zang. COREMU: A scalable and portable parallel full-system emulator. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPoPP'11, pages 213–222, San Antonio, Texas, USA, Feb. 2011. URL: https://doi.org/10.1145/1941553.1941583.

[272] V. M. Weaver and S. A. McKee. Can hardware performance counters be trusted? In *Proceedings of the International Symposium on Workload Characterization*, IISWC'08, pages 141–150, Seattle, Washington, USA, Sept. 2008. URL: https://doi.org/10.1109/IISWC.2008.4636099.

[273] V. M. Weaver and S. A. McKee. Using dynamic binary instrumentation to generate multi-platform SimPoints: Methodology and accuracy. In *Proceedings of the International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC'08, pages 305–319, Göteborg, Sweden, Jan. 2008. URL: https://doi.org/10.1007/978-3-540-77560-7_21.

[274] V. M. Weaver, D. Terpstra, and S. Moore. Non-determinism and overcount on modern hardware performance counter implementations. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, ISPASS'13, pages 215–224, Austin, Texas, USA, Apr. 2013. URL: https://doi.org/10.1109/ISPASS.2013.6557172.

[275] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, OSDI'06, pages 307–320, Seattle, Washington, USA, Nov. 2006. URL: https://www.usenix.org/legacy/events/osdi06/tech/full_papers/weil/weil.pdf.

[276] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe. Simulation sampling with live-points. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, ISPASS'06, pages 2–12, Austin, Texas, USA, Mar. 2006. URL: https://doi.org/10.1109/ispass.2006.1620785.

[277] J. Werner. Assessment of virtual machine working-sets in SimuBoost. Bachelor's thesis, Karlsruhe Institute of Technology (KIT), Operating Systems Group, Mar. 2018.

[278] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, OSDI'02, pages 195–209, Boston, Massachusetts, USA, Dec. 2002. URL: https://doi.org/10.1145/844128.844147.

[279] F. Wilhelm. Tracing privileged memory accesses to discover software vulnerabilities. Master's thesis, Karlsruhe Institute of Technology (KIT), Operating Systems Group, Nov. 2015.

[280] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The case for compressed caching in virtual memory systems. In *Proceedings of the USENIX Annual Technical Conference, General Track*, USENIX ATC'99, pages 101–116, Monterey, California, USA, June 1999. URL: https://www.usenix.org/legacy/events/usenix01/cfp/wilson/wilson.pdf.

[281] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS'96, pages 68–79, Philadelphia, Pennsylvania, USA, May 1996. URL: https://doi.org/10.1145/233013.233025.

[282] T. Wood, K. Ramakrishnan, P. Shenoy, and J. Van der Merwe. CloudNet: dynamic pooling of cloud resources by live WAN migration of virtual machines. In *Proceedings of the International Conference on Virtual Execution Environments*, VEE'11, pages 121–132, Newport Beach, California, USA, Mar. 2011. URL: https://doi.org/10.1145/1952682.1952699.

[283] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. D. Corner. Memory Buddies: Exploiting page sharing for smart colocation in virtualized data centers. *ACM SIGOPS Operating Systems Review*, 43(3):27–36, July 2009. URL: https://doi.org/10.1145/1618525.1618529.

[284] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the International Symposium on Computer Architecture*, ISCA'03, pages 84–95, San Diego, California, USA, June 2003. URL: https://doi.org/10.1109/ISCA.2003.1206991.

[285] M. Xu, R. Bodik, and M. D. Hill. A flight data recorder for enabling full-system multiprocessor deterministic replay. In *Proceedings of the International Symposium on Computer Architecture*, ISCA'03, pages 122–133, San Diego, California, USA, June 2003. URL: https://doi.org/10.1145/859618.859633.

[286] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. ReTrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the Workshop on Modeling, Benchmarking and Simulation*, MoBS'07, page 3, San Diego, California, USA, jun 2007. URL: http://www-mount.ece.umn.edu/~jjyi/MoBS/2007/program/01C-Xu.pdf.

[287] L.-K. Yan, M. Jayachandra, M. Zhang, and H. Yin. V2E: combining hardware virtualization and software emulation for transparent and extensible malware analysis. In *Proceedings of the Conference on Virtual Execution Environments*, VEE'12, pages 227–238, London, England, Mar. 2012. URL: https://doi.org/10.1145/2151024.2151053.

[288] S. Yang. Extending KVM with new Intel Virtualization technology. Napa Valley, California, USA, June 2008. URL: https://www.linux-kvm.org/images/c/c7/KvmForum2008%24kdf2008_11.pdf.

[289] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, and D. M. Hawkins. Characterizing and comparing prevailing simulation techniques. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, HPCA'05, pages 266–277, San Francisco, California, USA, Feb. 2005. URL: https://doi.org/10.1109/HPCA.2005.8.

[290] A. B. Yoo, M. A. Jette, and M. Grondona. Slurm: Simple linux utility for resource management. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, JSSPP'03, pages 44–60, Seattle, Washington, USA, June 2003. URL: https://doi.org/10.1007/10968987_3.

[291] P. Yosifovich, A. Ionescu, M. E. Russinovich, and D. A. Solomon. *Windows Internals Part 1*. Microsoft Press, seventh edition, 2017. ISBN: 978-0735684188.

[292] M. T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Proceedings of the International Symposium on Performance Analysis of Systems & Software*, ISPASS'07, pages 23–34, San Jose, California, USA, Apr. 2007. URL: https://doi.org/10.1109/ISPASS.2007.363733.

[293] M. Zangl. Towards heterogeneous deterministic replay for symmetric multiprocessors. Master's thesis, Karlsruhe Institute of Technology (KIT), Operating Systems Group, Nov. 2017.

[294] H. Zeng, M. Yourst, K. Ghose, and D. Ponomarev. MPTLsim: A cycle-accurate, full-system simulator for x86-64 multicore architectures with coherent caches. *ACM SIGARCH Computer Architecture News*, 37(2):2–9, May 2009. URL: https://doi.org/10.1145/1577129.1577132.

[295] X. Zhang, Q. Guo, Y. Chen, T. Chen, and W. Hu. HERMES: a fast cross-ISA binary translator with post-optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO'15, pages 246–256, San Francisco, California, USA, Feb. 2015. URL: https://doi.org/10.1109/CGO.2015.7054204.

[296] X. Zhang, Z. Huo, J. Ma, and D. Meng. Exploiting data deduplication to accelerate live virtual machine migration. In *Proceedings of the International Conference on Cluster Computing*, CLUSTER'10, pages 88–96, Heraklion, Crete, Greece, Sept. 2010. URL: https://doi.org/10.1109/CLUSTER.2010.17.