Karlsruhe Institute of Technology

Towards Three-Stage Parallelization of System Simulation

Bachelor's Thesis
submitted by

Marco Schlumpp

to the KIT Department of Informatics

Reviewer:	Prof. Dr. Frank Bellosa
Second Reviewer:	Prof. Dr. Wolfgang Karl
Advisor:	Dr.-Ing Marc Rittinghaus

17. Juni 2019 – 16. Oktober 2019

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any source or auxiliary means other than these referenced. This thesis was carried out in accordance with the Rules for Safeguarding Good Scientific Practice at Karlsruhe Institute of Technology (KIT).

Karlsruhe, October 16, 2019

Abstract

Full system simulation is a flexible tool for dynamic analysis of computer systems. However, it causes a large slowdown for most workloads. This slowdown can also change the behavior of the observed system. SimuBoost addresses this by recording the workload in a hardware-assisted virtual machine and replaying it in a full system simulator. Therefore, the behavior of the workload is not affected by the large slowdown of a full system simulator. To speed up the analysis, continuous checkpointing is used to create independently analyzable segments. These segments can be analyzed by multiple workers in parallel.

However, the continuous checkpointing introduces a new overhead source into the recording process. An additional replay stage can be used to separate the recording from the checkpointing. So far, SimuBoost has only implemented a replay engine for full system simulation. However, running the checkpointing in a full system simulation with its associated slowdown would cause significant delays in the parallelized analysis. Therefore, a replay for hardware-assisted virtual machines is necessary.

We implemented the necessary replay engine and compared its run time with the run time of the recording. Ideally, the difference should be small so the move to the stage does not affect the checkpointing. We measured that in most workloads the run-time overhead is less than 5%.

Contents

Chapter 1

Introduction

Full system simulation is a powerful tool to explore and analyze the activity of a computer system. The drawback of these simulators is the high slowdown caused by them. Additionally, the slowdown can increase when logging interesting information of the running workload such as memory accesses. There are two consequences of this slowdown. Firstly, the turnaround time increases, as it is necessary to wait for the analysis to finish. Secondly, the slowdown distorts the measurements and can also change the behavior of the system. Thus, it can be difficult to gain meaningful results out of a full system simulator.

SimuBoost [28] attempts to address the high run-time overhead of full system simulators by using deterministic record and replay. First, the workload is recorded on a hardware-assisted virtual machine. This process captures all non-deterministic events and is much more lightweight than a full system simulation. In a second stage the events are injected into a full system simulation. This way a reconstruction of the execution can be created. Therefore, only the run-time overhead of the recording can influence the execution, which is much lower than the run-time overhead of a full system simulator[28]. However, the injection of each event depends on the previous state. Therefore, the replay and thus the simulation is at first not parallelizable. To enable a parallelization of the analysis, SimuBoost makes use of continuous checkpointing, which creates checkpoints in regular intervals. The intervals enclosed between two consecutive checkpoints are distributed to a worker pool, which does the actual analysis. Each worker can reconstruct the execution independently by injecting the events of the interval on its starting checkpoint.

While the checkpointing makes it possible to parallelize the analysis, it causes a notable run-time overhead, thus thwarting the idea of a lightweight recording phase. Rittinghaus proposes a three-stage design to remove the checkpointing overhead from the recording [28]. An additional stage is inserted between the recording and analysis stage, which

replays the workload simultaneous to the recording and is responsible for the checkpoint creation. Because the parallelization factor now depends on the checkpoint production in the new stage, the replay engine should be able to keep up with the recording. Therefore, it should be based on a hardware-accelerated virtual machine.

The goal of this work is to implement a replay engine for hardware-assisted virtual machines on top of KVM, which can host the checkpoint production for the three-stage approach.

Chapter 2 gives an overview of the necessary technologies. An analysis of the runtime overhead caused by the checkpointing is done in Chapter 3. Chapter 4 gives an architectural overview of the replay engine. A description of the implementation is given in Chapter 5. After evaluating the implemention in Chapter 6, we conclude this work with a short summary in Chapter 7.

Chapter 2

Background

There are many approaches to measure performance data of a workload. For instance, by using the *performance monitoring unit* (PMU) of modern processors [9]. This unit usually contains multiple counters, which can be configured to count occurrences of specific events such as cache misses or retired instructions. While they can provide an insight into what the processor is doing, they are limited to low-level events and cannot be customized. For example, it is possible to count how many times a cache miss occurs, but it is impossible to record the accessed address.

A more flexible approach is to statically instrument the program code at interesting locations. This can be either done manually or in an automated fashion by extending the compiler [4]. A downside of this approach is that the resulting binary is specific to the inserted instrumentation. If something different should be measured, the whole process has to be redone. An alternative is to dynamically instrument applications at runtime [21]. Both approaches are only feasible for the scope of a few programs and not of a whole system.

When flexibility and elaborate probe capabilities on a system-wide scope are necessary, *full system simulation* (FSS) can be used instead. Because the hardware is only simulated using software, arbitrary measurements can be done. The core of a FSS is the processor simulation. A simple but flexible technique is to use an interpreter to execute the guest's code. Each instruction is fetched, decoded, executed and the result is applied to the machine model. This is repeated as long as the simulation is running. The benefit of this approach is the independence of the underlying host architecture.

However, to execute a single guest instruction the host must execute many instructions, which makes it inherently slower than a native execution. A technique with better performance for many workloads is *dynamic binary translation*. Instead of looking at a single instruction, a whole block of instructions is considered. This block is translated

into machine code suitable for execution by the host's processor. The translated code is usually cached, as most code is executed multiple times and the translation itself takes some time. As part of the translation, the code is also modified to change the state of the guest. For example, memory addresses cannot be directly used and need to be modified to point to the virtualized guest's memory. Privileged instructions such as `lgdt` on x86 are translated into unprivileged code, which just modifies the guest state, instead of acting on the physical hardware. Usually, simulators also include emulations of various common devices such as video cards or chipsets. Therefore, guest operating systems can in most cases run without any modifications in a FSS.

The level of detail of the simulation can vary depending on what information is needed. In the most basic form, a full system simulator may only simulate the functionality and will not try to re-create the timings of real hardware. This is called a *functional simulator* and a well-known example for this is QEMU [3]. When timing accuracy is necessary, the relevant parts of the hardware can be simulated with an accurate timing model. Some simulators such as Simics [20] can be used either as a functional simulator or a simulator with an accurate timing model.

The price for the flexibility of a full system simulation is that the simulation overhead is very high. Rittinghaus measured a slowdown of factor 43 for an Apache web server benchmark when running it on the functional simulator QEMU compared to running it on a hardware-assisted virtual machine (VM) [28].

This slowdown can be problematic for workloads, where multiple systems are involved, such as a web server running on a simulated system. The clients could be run outside the simulation, but the results might not be representative anymore because the web server needs more time to answer incoming requests. Alternatively, the clients could be part of the simulation, but this would increase the slowdown even more.

The same problem applies to interactive workloads. While the system might not notice that it is running slowly, a user interacting with the simulated system will notice the slowdown. A consequence is that the user might interact more quickly from the system's point of view when using a simulation. Thus, the system might be more busy and spend less time being idle. Unlike the non-interactive example above, the user cannot be integrated into the simulation.

In both cases, the workload running in the simulated system may not reflect the behavior of the workload on real hardware because the differences in the timing can influence behavior aspects such as scheduling decisions of the operating system.

SimuBoost [28] is an approach to reduce the slowdown accompanying full system simulation. The central techniques involved are deterministic record and replay as well as continuous checkpointing of virtual machines. The following section gives an overview

of deterministic record and replay. The concepts of virtual machine checkpoints and continuous checkpoints are explained in Section 2.2. The approach of SimuBoost is described in Section 2.4.

2.1 Deterministic Record and Replay

While the flexibility of a full system simulation is often desirable for an in-depth analysis of a system, the resulting measurement may not be representative of measuring the workload on real hardware because of the slowdown incurred by running on a full system simulator. On top of this, any instrumentation can add additional overhead.

An approach to eliminate the slowdown associated with full system simulation and instrumentation is to use deterministic record and replay [38]. The basic idea is that the instruction execution is generally deterministic except for a small set of interactions with the external environment such as I/O operations or hardware interrupts. By giving the exact same input to a system from the same starting point, original execution flow can be reproduced. Therefore, it is possible to record and replay a system similar to how a video can be recorded and played.

The replay does not have to be done on the same platform as the recording. For example, a workload can be recorded in a hardware-assisted virtual machine but replayed in a full system simulator. This combination is called a *heterogeneous replay*. In contrast to this, when either both phases use hardware-assisted virtual machines or both use a software simulation, it is called *homogeneous replay*.

Homogeneous replay can be used for many purposes, such as time travel debugging [19] or replicating virtual machines [11]. The benefit of a heterogeneous record and replay is the higher flexibility, as the platforms for both stages can be chosen independently based on the requirements of each stage. On the other hand, a homogeneous record and replay system will be easier to implement because the platform's behavior will be the same across the two stages.

The events can be categorized into two main classes, which are necessary to reconstruct the execution. For both classes it is enough to store the non-deterministic information introduced into the system. The first class are events, which happen *synchronously* to the instruction execution. For instance, reading the timestamp counter (`rdtsc` instruction [8]) is a synchronous event and only the observed timestamp counter value needs to be stored. When replaying the resulting recording, it is enough to use the previously stored values in the same order they were produced instead of executing the instruction itself. The implementation is straightforward given an existing recording implementation because the event can be injected at the same location where the events are recorded.

The second class consists of events happening *asynchronously* to the execution, such as hardware interrupts or DMA writes. This class is more difficult to replay efficiently because the injection point has to be recorded and replayed with instruction granularity. The instruction pointer is not sufficient because the same instruction might be executed multiple times without any other event happening between each occurrence (e.g., as part of a loop). Even considering the whole register file is not enough as the vital location identifying information might reside on the stack or more generally somewhere in memory. Considering the whole memory, however, would simply be too expensive because hashing gigabytes of memory takes a considerable amount of time. A very simple and efficient alternative is to count the number of executed instructions until an event happens. This can be accomplished by using a performance counter configured to count the executed instructions. However, this requires that the counter is accurate, which is not the case in practice [22].

A workaround for these issues is to use a combination of mentioned strategies. For instance, Rittinghaus used an instruction counter to roughly determine the location and used register comparisons for the precise location [28]. A weakness of the approach is that it is possible to craft a program, which can cause a divergence at the replay phase and making it impossible to continue.

An example for this, is a simple loop which only contains code to keep track of the iteration count. Instead of storing the counter in a register, the counter is stored in memory. By doing so, the replay engine cannot distinguish between the iterations. Suppose an asynchronous event such as an interrupt happens when this loop is executed. When the interrupt handler is executed while recording, it will be able to read the current iteration counter and do something based on the value. If a replay of this situation is attempted, the replay engine might inject the asynchronous event too early because the register content stays the same across all iterations and the instruction counter is only usable for rough locations. When the interrupt handler reads the counter in this case, the value might differ from the value read in the original recording causing a divergence. Because SimuBoost was not designed with security-focused applications, this failure mode is not considered a problem.

Over the years, many deterministic record and replay systems have been developed in the industry and in the research community.

ReVirt [15] is an implementation of homogeneous record and replay for post-attack analysis of system intrusions. The resulting log of non-deterministic events grows about $0.08\,\mathrm{GB/d}$ while causing a run-time overhead of 8% when compiling a Linux kernel. Their implementation is based on UMLinux [6]. UMLinux is a modified Linux allowing the kernel to run in user space. As such, a significant limitation of the used design is that it cannot run arbitrary operating systems. A similar project [19] uses homogeneous

deterministic record and replay to implement time travel debugging. This work uses a modified user-mode Linux kernel, which has the same limitation as the UMLinux-based approach.

A variant of *ReVirt* with support for multiprocessor systems called *SMP-ReVirt* [14] was developed in 2008. It is based on the Xen hypervisor and supports only paravirtualized guests. It uses a homogeneous record and replay, too. The authors measured a runtime overhead of 10% when recording a single processor guest building the Linux kernel. In this case, the system generated compressed log files at $0.562\,\mathrm{GB/d}$. Burtsev et al. [7] propose a general blueprint for implementing deterministic record and replay systems. They also provided an example implementation on top of Xen. While the use of paravirtualized hardware reduces the overhead of the virtualization, it also makes it harder to use in a heterogeneous record/replay scenario, where the analysis phase runs in a full system simulator. The paravirtualized hardware is usually hypervisor specific and not available in other hypervisors.

ExecRecorder [23] is a homogeneous record and replay engine for post-attack analysis. It uses a modified Bochs emulator, which is an x86 interpreter. Therefore, it is too slow for recording a real world workload with representative timings. The tool generates log files at $5.4\,\mathrm{GiB/h}$ on average over multiple workloads.

PANDA [12] is a tool for reverse engineering whole systems. As part of the tool, the authors wrote a record/replay engine based on QEMU. A similar implementation based on QEMU was done by Dovgalyuk [13] but with focus on time travel debugging. The record engine generates log files at $30\,\mathrm{kB/s}$. Another deterministic record/replay engine with a security focus was written by Srinivasan and Jiang [31]. They used a slightly different approach for their implementation. Instead of modifying QEMU itself, they largely left it unchanged and rather record all library calls QEMU makes, which are later used to replay QEMU and the contained virtual machine. They measured a slowdown of 62% for an Apache web server benchmark.

However, these QEMU-based approaches use the dynamic binary translation mode for the recording and replay. While it is faster than a pure interpreter such as Bochs, a Linux kernel build, for example, still takes 22.7 times longer than on a hardware-assisted virtual machine [28].

ReTrace [38] was a commercial record/replay implementation, which was part of the VMware hypervisor. It was removed because the feature was not considered important enough [17]. The purpose was to create detailed execution traces of a system, which could be used for further analysis. ReTrace introduced the previously described idea of using deterministic record and replay to reduce the simulation overhead. The workload is first recorded in a hardware-assisted virtual machine. The second phase replays the system based on the recording. The user can toggle the generation of an execution log

for interesting parts of the recording. When the logging mode is activated, the hypervisor uses an interpreter to execute the guest's code. After each instruction, the current system state is logged to the execution log. It also includes information about exceptions and interrupts, which happen during the recording. The resulting log describes the complete execution, which can be used for further analysis.

When using this approach it is essential that the simulated system behaves exactly the same when being recorded and replayed. If the simulated system differs even slightly when replaying, it can diverge at some point and cannot be replayed from that point onwards. For example, the behavior of some instructions differs between QEMU and real x86 processors [28]. Real processors perform a write probe before executing instructions which read and then write memory. If the destination is not writable, the processor triggers a write page fault. In contrast, QEMU directly attempts to read the memory, which results in a read page fault in case of an inaccessible memory location. A consequence is that it is hard to build a heterogeneous record/replay, which runs the two phases on different hardware implementation or simulators.

2.2 Checkpointing

Checkpoints can be created to capture the state of a virtual machine at a given point in time. The saved state contains the values of the CPU registers, the contents of the main memory and may also include device state such as the framebuffer and configuration of the video card.

The captured state can be used for many purposes. For example, it can be used to transfer virtual machines to different hosts without stopping the VM. They can also be used to restore a virtual machine in case of a failure.

When the checkpoints are created in regular intervals, the process is called *continuous checkpointing*. For example, this can be used to build a *flight data recorder* [37]. If the virtual machine crashes, the system state can be reconstructed afterwards to investigate the cause of the crash. Continuous Checkpointing can also be used to ensure high availability of virtual machines [11]. For this purpose, the virtual machine checkpoints are continuously copied to a secondary host. If the primary host goes down for some reason, the replicated virtual machine can substitute the lost VM.

To create a consistent checkpoint, the whole system needs to be temporarily stopped. While the system is paused, the state can be captured. The duration for which the system is paused is called *downtime*. During the downtime, the CPU registers and device state are captured. Because registers are very small and there are only a few, this can be done in about $3\,\mathrm{ms}$ [28]. But the main memory can be fairly large and thus takes more time to

copy than the other state. For example, creating a copy of the main memory of a virtual machine with 8 GiB may take around 2 s [28]. For continuous checkpointing the resulting downtime would be too long if the copying happened while the machine is paused.

Depending on the circumstances in which checkpointing is used, different optimizations for this time-consuming and storage-intensive process can be applied. When continuous checkpointing is used, an observation is that usually only some memory regions change their content between two checkpoints. The method of copying only the modified pages is called *incremental checkpointing*. This lowers the size of the incremental checkpoints for many real-world workloads considerably. For example a Linux kernel build modifies approximately 156 MiB on average when every 8 s a checkpoint is created [28]. When using this method the downtime depends on how many pages the guest modifies. However, the consequence is that RAM-heavy workloads are still impacted by a large downtime [28].

The next optimization is to use *copy-on-write* checkpointing [32, 5]. The goal is to delay the actual copying and only mark the to be copied pages as read-only while the VM is paused. After the execution is resumed, the copy process can happen asynchronously to the execution. If the guest attempts to write into a page which is scheduled to be copied, a page fault occurs and the hypervisor can copy the page ahead of schedule. Afterwards, it can restore the original permissions, allowing modifications to the page. By using this technique the downtime can be further reduced, but the overall slowdown is still high for memory-heavy workloads. Because of the additional page faults, the slowdown can even increase for synthetic benchmarks compared to a synchronous approach [28].

An important part of efficient incremental checkpointing is to keep track of modified pages. The most basic way is to use the write-protection bits in the page tables. When the guest attempts to write to a page, the hypervisor can mark the page as modified and remove the write-protection of the page. The execution can continue afterwards. The disadvantage is that there are many additional page faults. Taking checkpoints with a 1 s interval and write-protection-based tracking of modified pages incurs a run-time overhead of 25% for postmark.

An alternative is to consider the *dirty* bits in the page table entries. By walking through the page table tree during the downtime, all modified pages can be collected. The intermediate layers of the page table on x86 only store an accessed bit. But this bit can still be used to prune the tree because no child table will have a dirty bit set if the parent was not accessed [28]. This way, many branches can be skipped while scanning through the page table. While the *scan* method reduces the overhead to 20%, it almost doubles the length of the downtime for postmark [28].

Similar to memory copying, some work of the *dirty logging* can be moved outside the downtime. In this case, the so-called *pre-scan* method asynchronously walks through

the page table right before the downtime begins [28]. Afterwards, the normal downtime starts and the page table is scanned again to detect pages that have been modified in the meantime. The second scan takes less time because the previously mentioned pruning optimization. The *pre-scan* can make the downtime duration comparable to the write-protection approach for many workloads while keeping the low overall overhead benefit of the scan method [28]. The overall overhead of copy-on-write, pre-scan, and incremental checkpointing combined is around 10% for a Linux kernel build when using a checkpoint interval of 1 s. For more memory intensive benchmarks such as postmark the combination causes a 20% slowdown in the same configuration [28].

2.3 QEMU/KVM

QEMU [27] is an open source emulator, which can run operating systems either in a full system simulator or hardware-assisted virtualization environment using KVM. For full system simulation QEMU uses dynamic binary translation. In this mode the guest's instructions are dynamically translated to work on the virtual guest state instead of the physical host. This may also include a translation between instruction sets if the guest uses a foreign ISA. In addition to the processor emulation, QEMU also emulates a variety of hardware components such as network interfaces and graphics cards.

However, the dynamic binary translation mode of QEMU is not fast enough for virtualization purposes, where performance is more important than flexibility. To provide support for virtualization on x86 QEMU relies on hardware support. For Intel processors this functionality is called *virtual-machine extensions* (VMX) [9]. Other architectures have similar mechanisms. For example, AMD calls it *secure virtual machine* [18] (SVM).

The VMX extension introduces two modes of operation for the processor. The first mode is called *root mode*. This mode is mostly the same as the normal operating mode of the processor with some small restrictions but can still access the hardware. Hypervisors run in this mode and control the execution of virtual machines. The guest itself runs in *non-root mode*, to which the processor switches upon the request of the hypervisor. After the transition (called *VMX entry*) is completed, the processor will run the code of the guest.

The configuration for this mode is stored in the *virtual-machine control structure* (VMCS). The access to real hardware is usually restricted in non-root mode by setting appropriate settings in the VMCS. When the guest accesses a restricted function such as port I/O, the processor will transition back to root mode. This transition is called *VMX exit*. Based on the exit code the hypervisor can determine the appropriate action. For instance, the emulation of port I/O on a virtual device.

To expose these different mechanisms under a uniform interface, Linux comes with *Kernel-based Virtual Machine* (KVM). The scope of KVM is mostly limited to virtualize bare processor cores and related chips such as interrupt controllers. Hypervisors have to supplement the functionality provided by KVM to implement a complete virtual machine environment.

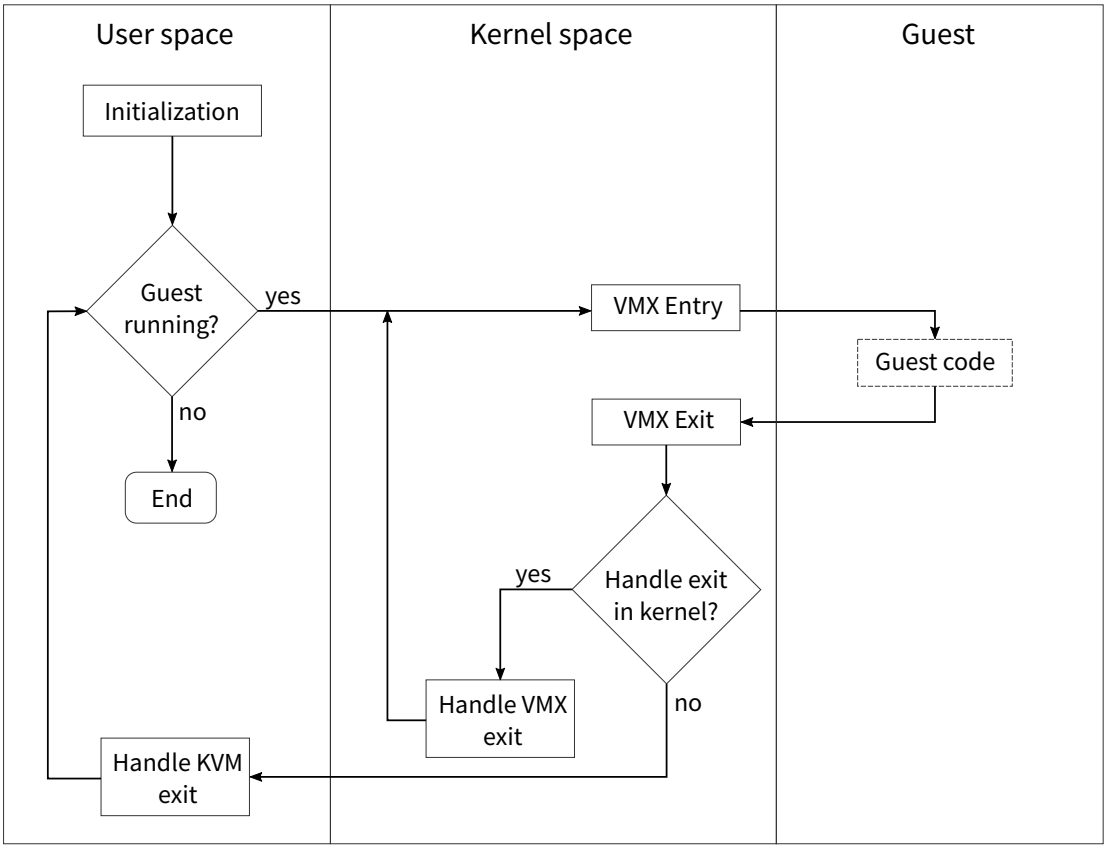

Figure 2.1: Hypervisors can provide a particular environment for guests by handling KVM exits.

The user space API is based on *ioctl*s [33]. An overview of the typical interaction with it is shown in Figure 2.1. After opening `/dev/kvm`, the `KVM_CREATE_VM` ioctl can be used on the open descriptor to create a virtual machine. Multiple virtual CPUs can be created afterwards using `KVM_CREATE_VCPU`. Most other ioctls are used to manipulate the state of the virtual machine or the virtual processors. For example, the registers can be accessed or interrupts can be injected. The execution can be started by using the `KVM_RUN` ioctl. KVM then uses one of the previously mentioned hardware mechanisms such as VMX to execute the code of the guest. When a VMX exit happens, KVM first decides whether the exit can be handled in kernel space. For instance, accessing a swapped out page will be handled in the kernel. If the exit cannot be handled in kernel

space, the request is passed to user space by returning from the `KVM_RUN` ioctl system call. The user space program receives a description of the event which caused the exit and can further handle it.

As described previously, QEMU was originally a pure emulator. It was later extended to be able to use KVM instead of using its hybrid translation backend. In this configuration, QEMU responds to the KVM exits by forwarding the requests to the emulated devices.

2.4 SimuBoost

SimuBoost combines record and replay with checkpointing to address the slowdown of classical full system simulation. Figure 2.2 shows an architectural overview of SimuBoost. The high run-time overhead of a full system simulation is reduced by using heterogeneous record and replay. The workload runs in a hardware-assisted virtual machine using QEMU with KVM. This reduces the overhead when running the workload compared to running it on a full system simulator, which preserves the representativeness [30]. The analysis is done by replaying the previously captured non-deterministic events in QEMU in the emulator mode. The benefit of this separation is having the flexibility of a full system simulation without the distortion in execution behavior that would be caused by the high slowdown of the simulation.

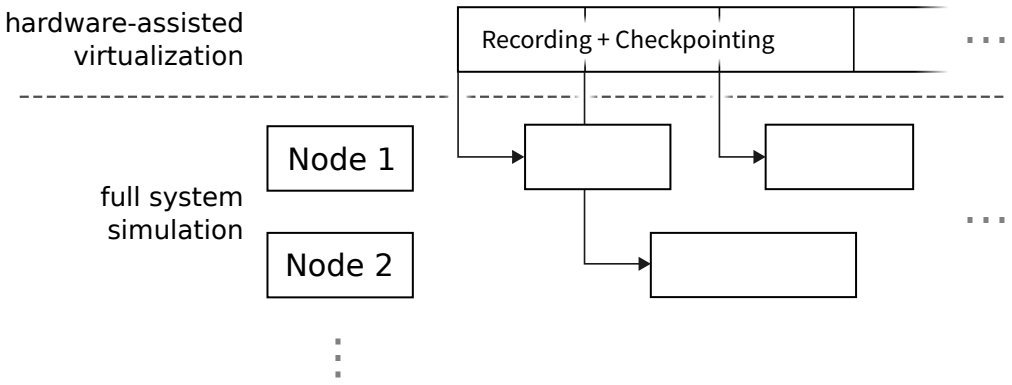

Figure 2.2: The workload is recorded on a hardware-assisted virtual machine and checkpoints are created in regular intervals. The resulting segments are distributed to a pool of worker nodes.

By using continuous checkpointing the analysis can be parallelized and drastically reduces the overall time required for the simulation. Each time a checkpoint is created, the interval between the previous and the current checkpoint can be replayed. Because each interval is independent of each other, the parallelism is mainly limited by the creation of new checkpoints and the availability of simulation nodes.

2.4.1 Simutrace

SimuBoost uses Simutrace [29] to store the recorded non-deterministic events, checkpoints, and analysis traces. It uses a client/server architecture to allow access to the data. When clients run on the same host as the server, the clients can communicate with the server using a named pipe. The data itself is transported using shared memory. When the client sends a request to append a new segment to a stream, the Simutrace server hands out an empty $64\,\mathrm{MiB}$ segment in a shared memory region. When the client is either done with writing data or the segment is full, the client can tell the server about the completion.

The process for reading segments is similar. The client can request segments and the server will write the content of the requested segment into a shared memory region, which can be then accessed by the client. The benefit of this approach is that the data needs to be copied less often as it would when sent over a pipe or socket. However, clients can also access the server via network if the server runs on a different host. In this setup it is no longer possible to use shared memory and the data is sent over network sockets.

After Simutrace receives a new segment of data, it can apply different compression and storage schemes depending on the stream type. For example, the non-deterministic event streams of SimuBoost are compressed using LZMA. Checkpoints, on the other hand, use a mixture of custom and general-purpose compression methods as well as data deduplication [28]. After all filters have run, the output is written to storage. In case of a local shared memory buffer, the buffer is reused for further client requests.

Chapter 3

Analysis

The term *probe effect* describes the symptom of influencing measurements by doing the measurements. For example, a profiler might add instrumentation to the beginning and ending of every function. By doing so every function call will be more expensive, because the inserted code will consume some time. The resulting slowdown is more significant for small functions than for more time-consuming ones. Therefore, some parts might appear more expensive than they are in a non-instrumented build.

The effect is not necessarily caused by additional executed code only, but bottlenecks in the system can also induce a probe effect. For example, the main memory bandwidth is limited and a concurrently executed memory-intensive analysis application can negatively affect the performance of other programs even when they run on different cores.

Because any conclusions based on imprecise measurements can be misleading, it is crucial to minimize the probe effect. The probe effect also affects the analysis of a system in full system simulators, because many operations are slower than on a real machine. By splitting the analysis into two phases, a lightweight recording and a slow analysis, SimuBoost already moved the simulation and instrumentation overhead away from the main execution. However, the goal of parallelizing the analysis workload makes it necessary to use checkpointing. While a well-optimized checkpointing implementation can reduce the overall slowdown, it will still cause a probe effect. However, any probe effect in the recording is carried into the analysis. In this case, the checkpoint creation slows down the simulated system causing an unnecessary probe effect. In the following, we investigate impact of checkpointing by measuring the overall run-time overhead. Section 3.2 describes an approach to remove the probe effect caused by checkpointing.

3.1 Checkpointing Overhead

To measure the actual impact of checkpointing, we first run a workload on a virtual machine without checkpointing and then run it on a virtual machine with checkpointing enabled. This is based on the assumption that the induced probe effect will increase the time needed for the affected operations. This should be the case because the instrumentation will always add additional operations to the operations executed inside the simulated system. Thus, when comparing the two runs, the difference in the total time spent is approximately the sum of the time taken for these additional operations.

We run following benchmarks on the same setup as described in Section 6.2 but using an unmodified version of SimuBoost. SimuBoost is configured to record the workload and to use copy-on-write, pre-scan, and incremental checkpointing (see Section 2.2). Multiple checkpoints intervals are tested because shorter intervals can be scheduled more tightly onto the analysis worker pool. The downside of a higher checkpoint frequency is the increased overhead. Additionally, each benchmark is run with only the recording active. The results are compared against a benchmark run without recording the non-deterministic events and without creating checkpoints. A more detailed description of our methodology can be found in Section 6. The results are shown in Figure 3.1.

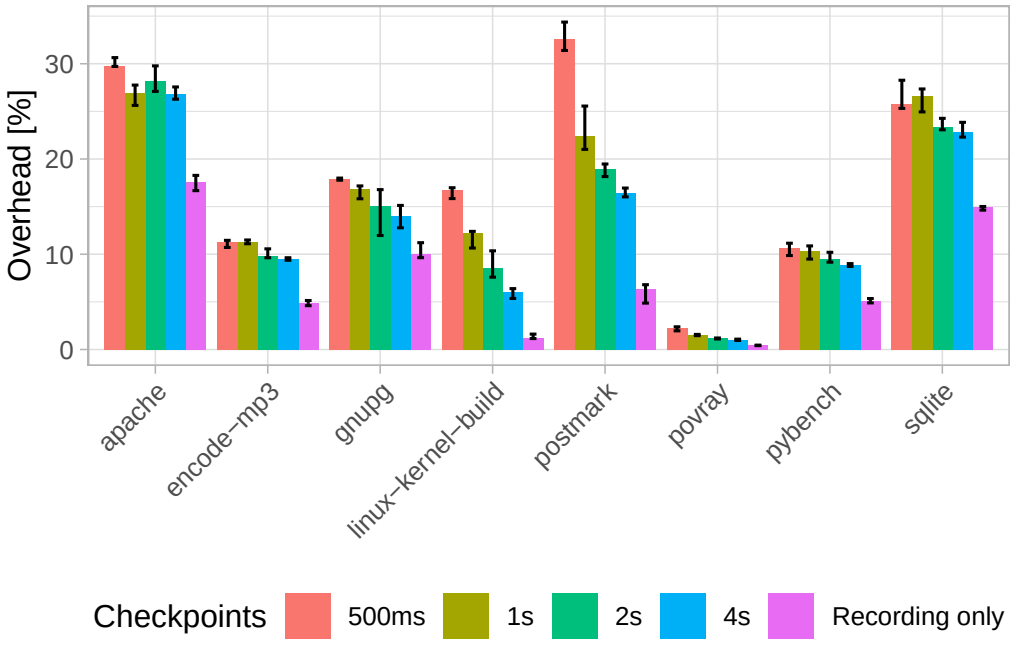

Figure 3.1: Checkpointing can cause a considerable slowdown. Memory-intensive benchmarks such as postmark are particularly affected by the overhead.

The first benchmark is the Apache web server benchmark. In the benchmark the current time is often retrieved using the `rdtsc` instruction [28]. By doing so many events are generated, which causes the recording to have an overhead of 17% compared to other benchmarks. While activating the checkpointing increases the overhead to 27%, increasing the checkpoint frequency does only slightly increase the overhead.

In contrast, the *encode-mp3* and *pybench* benchmarks only experience an overhead of 5% when being recorded. Enabling checkpointing increases the overhead to 10%, while increasing the checkpointing interval does only have a small effect similar to the Apache benchmark. For the Linux kernel compilation the overhead caused by the recording is less than 1%. However, creating checkpoints in $1\,\mathrm{s}$ intervals raises the overhead to 12%.

Postmark touches large amounts of memory, which causes a large overhead of 32% in combination with checkpointing in $500\,\mathrm{ms}$ intervals. The event recording itself only causes a 6% overhead. Workloads which are CPU-heavy, such as POV-ray are only slightly affected by both recording and checkpointing. Even when creating checkpoints in $500\,\mathrm{ms}$ intervals, the overhead is only 2%.

In summary, creating checkpoints causes a non-negligible probe effect and can cause distortions even with low checkpoint intervals. Using checkpointing has a considerable effect for all measured benchmarks, except for purely processor-bound benchmarks. As the parallelization approach of SimuBoost requires high frequency checkpointing, the overhead can very high for memory-intensive workloads.

3.2 Three Stage Parallelization

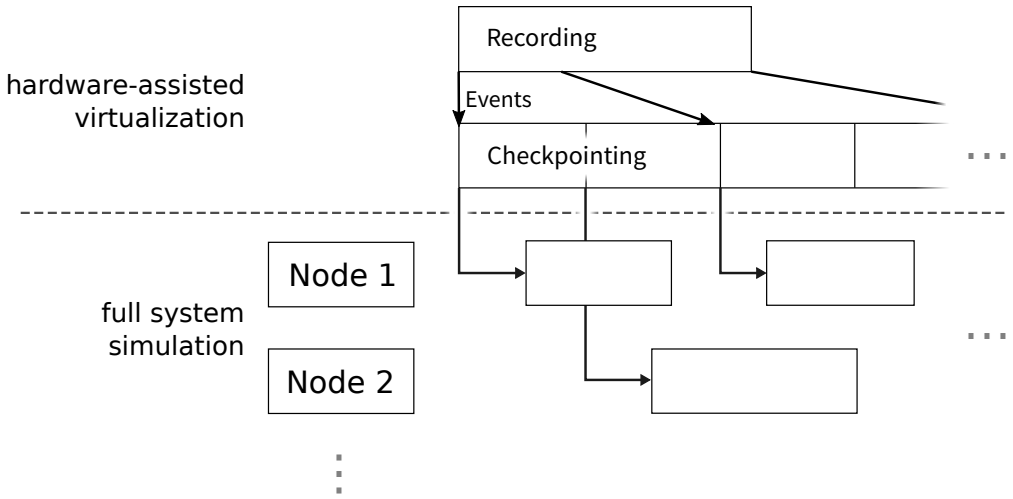

Figure 3.2: Using three stages to remove the checkpointing overhead from the recording. The probe effect caused by the checkpointing is no longer affecting the measurements in the analysis.

A proposed improvement to remove the probe effect induced by checkpointing is to move the checkpointing to a new intermediate phase between the record and analysis phase [28].This phase directly replays the previously recorded workload, thereby producing the exact same execution irrespective of any additional overhead. Creating checkpoints in this phase does not add to the probe effect. As shown in Figure 3.2, the analysis of an interval can start as soon as the new replay phase creates the checkpoint at the start of an interval. This matches the current approach used by SimuBoost.

To maintain a fast and effective parallelization for the interval simulations, the replay phase must not be slower than the original recording phase. Thus, the new checkpointing phase cannot use the existing replay engine of SimuBoost built into the slow full system simulation. Instead, we need to replay the execution in a hardware-assisted virtual machine, which is the same environment already used for the recording in the first phase. As part of this work, such a replay engine will be implemented.

Chapter 4

Design

In the previous section, we described the approach of creating checkpoints in an additional phase to remove the probe effect caused by checkpointing from recordings. This approach requires a replay implementation for hardware-assisted virtual machines to maintain the parallelizability of SimuBoost. As the newly written replay engine has to be integrated into the existing SimuBoost system, the implementation has to consume the event format output by the current recording engine.

As explained in Section 2.1, program execution is mostly deterministic. By recording the few remaining non-deterministic operations, a replay of the original workload can be created. I/O is usually non-deterministic because the retrieved values depend on factors not controlled by the hypervisor such as key presses or network communication. Events which expose some sort of timing to the guest must also be recorded such as the occurrence of interrupts or reading a timestamp.

Event	Logged data
`in`, `out` instructions	address, data
`cpuid` instructions	`eax`, `ebx`, `ecx`, `edx` registers
`rdtsc` instructions	`eax`, `edx`
`rdmsr`, `wrmsr` instructions	`ecx`, `eax`, `edx`
APIC accesses	offset within configuration page, length, and value
HPET accesses	offset, length, and value

Table 4.1: Synchronous non-deterministic events on x86 processors

Some events happen synchronously to the program execution. Table 4.1 lists these events for the x86 architecture. Besides port-based I/O using the `in` and `out` instructions, another possibility to communicate with external devices is to use memory-mapped I/O.

Both the *Advanced Programmable Interrupt Controller* (APIC) and the *High Precision Event Timer* (HPET) use this approach. The HPET provides a timer accessible in a memory region, which makes it necessary to record the interactions with it. The APIC is responsible for the interrupt delivery and the configuration can be accessed in a special memory page. Instead of emulating the APIC, we decided to replay any interactions with it. This includes accessing the configuration page and also interrupts. This data was already logged by the original SimuBoost implementation for simplicity reasons, because the behavior needs to stay consistent on both hardware-assisted virtual machines and full system simulators.

While `cpuid` is deterministic and not necessary for the replay on a hardware-assisted virtual machine if the same processor model is used, it is useful for the following analysis phase. The values reported by a full system simulator will differ from the ones reported in the hardware-assisted virtual machine leading to a divergence. By recording the results, this situation is avoided. The `rdmsr` and `wrmsr` instructions, that are used to access the model specific registers, have the same problem and are also recorded for that reason.

Event	Logged data
DMA writes	Address and written data
Hardware interrupts	Interrupt vector number
System Management Interrupts	–

Table 4.2: Asynchronous non-deterministic events on x86 processors

There are also non-deterministic events, which can happen asynchronously to the instruction execution. A list of those events is shown in Table 4.2. DMA writes are used to asynchronously write data to the main memory without the involvement of the CPU. For instance, this can be used for asynchronously reading data from the hard disk. The completion of the asynchronous operation can be signaled with a hardware interrupt, which is another asynchronous event source.

Both DMA writes and interrupts depend on the timing outside the simulated system. For example, in the hard disk scenario, the hypervisor might have to wait for the data to arrive from a real hard disk, which can take an unknown amount of time. Additionally, DMA writes can introduce non-deterministic events into the guest.

The following section gives a short overview of the existing recording implementation because some mechanisms of it are reused for the replay engine. Section 4.2 describes the high-level design of the replay engine itself and how it integrates into the existing SimuBoost architecture.

4.1 Recording

The recording engine was implemented on top of QEMU and KVM to reduce the execution overhead as described in Section 3.2. This was done as part of SimuBoost and the implementation will be used for this work.

The port I/O instructions are trapped within KVM because the guest should not be able to directly communicate with devices of the host. Instead, KVM will forward these requests to QEMU, which emulates the connected devices.

The `cpuid` instruction is also trapped, even when the CPU model is passed through because KVM needs to filter out some features, which cannot be used in a guest system [36]. Thus, the existing code only needs to be adjusted to record the return values. The VMX extension also exposes a way to trap `rdtsc` instructions, which can be used to implement recording of timestamp counter accesses. Likewise, accessing the model specific registers using `rdmsr` and `wrmsr` is also trapped for recording or replaying.

As hardware interrupts are already injected by the hypervisor itself, recording them can be done in the function, which actually injects the interrupt. The same holds true for DMA writes, which are done by QEMU by writing into the address space of the guest.

To identify the correct location of an event, the recording implementation creates a landmark each time an event happens. A landmark contains the values of a subset of registers and the count of the so far executed instructions. While only strictly necessary for asynchronous events, this landmark is also used for verification purposes to ensure that the replay progresses correctly.

A peculiarity of the x86 instruction set is the existence of the `rep`eat instruction prefixes, which can be combined with string instructions. This allows a compact representation of loops common in string manipulation. Examples of such loops include comparing two null-terminated strings or filling a memory block with a value [8]. After each iteration the counter register is decremented and the pointers to the string are moved forward. The iteration stops when the counter reaches zero. Depending on the used string instruction, the equality of the current characters or if a zero is encountered, can also be a condition to stop the iteration.

Because these instructions can run for extended periods, it is important that they can be interrupted. When an interruption happens, the CPU stores the current iteration in the input registers. When the control returns to the instruction after the interrupt was handled, the instruction will continue where it previously stopped. However, these instructions count as a single instruction for the performance counter. To uniquely identify the correct time to inject an asynchronous event in a replay, the counter register `rcx`, string source register `rsi` and string destination register `rdi` are also required.

4.2 Replay

The replay implemented as part of this work uses the recorded data to reconstruct the execution flow and is also based on KVM.

While any additional overhead does not affect the execution anymore, it is still important to have a low overhead. If replaying entails a considerable slowdown in the checkpointing run, the parallelization in the analysis phase is negatively affected and the attainable speedup might decrease notably. To avoid any delay in the parallel simulation, the replay should also be able to run concurrently to the recording.

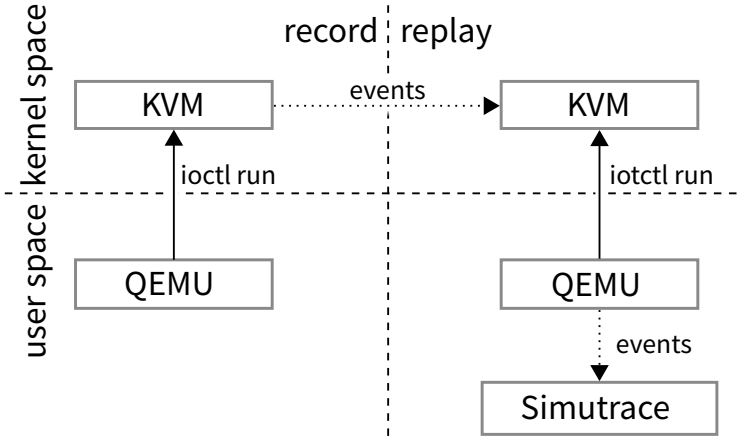

Figure 4.1: The recording is modified to pass any generated events to the replay. Afterwards, the replay engine writes them into Simutrace.

The high-level architecture of the setup is shown in Figure 4.1. In the current implementation the checkpointing depends on being able to add special marker events into the recording log to identify interval boundaries. The new checkpointing phase implementation should also allow the insertion of these checkpoint events because Simustore streams are append-only. If the recording phase had passed the events to Simutrace directly, the checkpointing stage would not be able to insert the markers anymore. By passing all events through the replay engine, the additional events can be inserted when a checkpoint is created.

The event transport itself can be adapted depending on the scenario. If the replay runs on the same machine, the recorded events can be directly shared with some additional synchronization. Alternatively, the data could be sent over the network if the replay runs on a separate system.

Chapter 5

Implementation

The replay is implemented on top of the modified Linux 4.3.0 kernel of SimuBoost. This kernel already contains the KVM patches necessary to record workloads in a hardware-assisted virtual machine and create continuous checkpoints. Section 5.1 describes the transport used to move the recorded data to the replay. The implementation of the replay itself is described in Section 5.2.

5.1 In-kernel Event Transport

The recording, transport, and replay use the same binary representation for the events. The benefit is that the events can be easily moved around without any conversion involved. A single event consists of an event type, a landmark, and an optional payload data (see Section 4.1. While the event itself has the same length across all event types, the payload can vary in size depending on the event type or in case of DMA writes even vary between events.

When an event is created while recording a workload, a linked list node is allocated from a `kmem_cache`. Because the event structure has a fixed size, it is directly embedded into the list node. The variable-length payload is stored into a separate allocation, referenced by a pointer in the node structure. The node is inserted into the local synchronous or asynchronous event list depending on the event type.

After a certain number of events, the contents of both lists are moved into a queue shared with the replay process. Figure 5.1 visualizes the resulting structure. The reference-counted queue is created when a recording is started and is shared with the replay process. To maintain a low overhead, the transport does not use a normal list protected by a mutex but rather a wait-free single-producer single-consumer queue. It is implemented

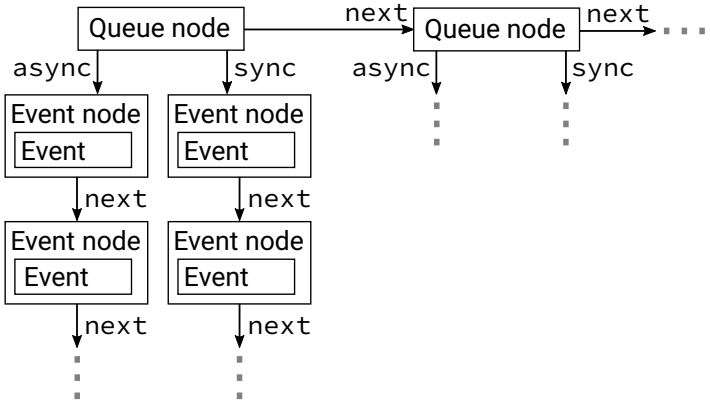

Figure 5.1: The events are transported using a queue, which contains a list for synchronous and asynchronous events. Batches of multiple events are wrapped into a queue node to reduce the run-time cost using atomic instructions to insert new events into the queue.

using a single-linked list and is based on a design by Vyukov [34]. Pushing an element with this implementation requires one allocation out of a `kmem_cache` and one atomic instruction.

The replay process on the other side of this queue also has two local event lists in addition to a pointer to the shared queue. Every time an event has been successfully replayed, the corresponding event will be removed from one of the local lists depending on the whether the event was asynchronous. When neither a synchronous nor asynchronous event is left and the recording is still running, new data is fetched from the shared queue. If the queue is empty, the replay will wait for the availability of more events in a busy-loop. When a new batch of events was successfully fetched from the shared queue, the events contained in the queue node are appended to the local replay lists. Because the structure of the lists stored in the shared queue nodes is the same as the local list, the new batch of events can be spliced to the local lists by just changing a few pointers.

If only one of the local lists is empty, it is safe to continue the replay and wait until the other list is also empty. In other words, until the last event of the current event batch is processed, no event of the empty list's category can occur. If an event violating this property would exist, the event would have to have happened before the batch's last event. But this event would have been contained in the current batch, as the two lists are flushed at once.

This is important for asynchronous events because missing one of them could silently cause a divergence, which would only surface on a later event. Because the event batch does not have any gaps, it is safe to not schedule any asynchronous events for injection if only the asynchronous list is empty.

Finally, the event node is moved into another pair of lists after being replayed. These lists represent the replayed but not yet permanently stored events. The events in these lists are moved to user space every time a KVM exit happens.

After every KVM exit, the user space part of SimuBoost collects all accumulated replayed events using an `ioctl` on the virtual CPU. This `ioctl` mechanism was already implemented and previously used for collecting the events from the recording. This collection procedure is now done in the replay, because the checkpoint markers must be inserted. The `ioctl` directly writes into the 64 MiB buffers provided by Simutrace. This is also the point where the list node memory is freed.

The required modifications to inject checkpoint marker events are small because the already replayed events and the incoming events are stored separately. The user space only receives events which have already been applied and have contributed to the state, which is made persistent when a checkpoint is created.

A checkpoint background thread simply pauses the virtual machine, which causes a KVM exit and triggers the previously mentioned collection of replayed events. The background task waits until this is done and creates the checkpoint itself. Afterwards, it emits an event directly to the Simutrace stream and resumes the virtual machine. The emitted event will be consistent with the surrounding events because the deterministic nature of the replay.

5.2 Event Replay

The next step is to use the acquired events to recreate the original execution. For the synchronous events this is straightforward because the event can be replayed at the same location in the code where it was originally recorded. The general structure of these locations is shown in Algorithm 1 using the example of `rdtsc`.

When the CPU encounters an `rdtsc` instruction and VMX is configured to trap it, the CPU causes a VMX exit. In this case, if the recording is active, the current timestamp value will be read, written into a new event, and also into the corresponding registers with the virtual machine. When the replay is active, the hypervisor fetches the next synchronous event. If the event type matches, the hypervisor fetches the associated event data, too. This data is then injected into the virtual machine. If the event type does not match, something in the replay process went wrong and must be aborted as the state already differs from the expected state.

In both the recording and replay, the `rdtsc` instruction is skipped, by using the instruction length provided by VMX. Afterwards, the execution of the virtual machine

Algorithm 1 Handling of a synchronous event using `rdtsc` as an example

```
procedure HANDLE_RDTSC
    if replay_active then
        event ← consume_event()                    ▷ Fetch and check landmark
        eax ← event.eax
        edx ← event.edx
    else
        eax, edx ← rdtsc()                         ▷ Wrapped Operation
        add_event(EVENT_RDTSC, eax, edx)
    end if
    vcpu.eax, vcpu.edx ← eax, edx
    move_instruction_pointer_forward()
end procedure
```

continues. If no recording is active the `add_event` call is a no-op and does not cause any additional overhead on top of the overhead caused by the VMX exit. The same process applies to the other synchronous events.

Injecting asynchronous events is more complex than injecting the synchronous events. Every instruction can be a potential location for an asynchronous event. For the replay implementation, we attempt to inject the next asynchronous event every time a VMX exit happens. With this design, it is sufficient to cause a VMX exit to inject an asynchronous event. When the replay engine attempts to inject an event, the landmark is compared to the current processor state. A landmark matches the current state, when all registers values stored in the landmark are equal to the current register values. If the state matches, the event is injected. This is repeated for the following events, until a non-matching event is encountered.

The stored instruction count is also compared to the current instruction count. Unlike the registers the value is not checked for exact equality because the counter can diverge from the real value in some circumstances [35]. To counteract this symptom, it is tested whether the current replay instruction count is within a window around the landmark's instruction count. This window should be as small as possible to reduce the chance of early false positives. For instance, these can happen when injecting an interrupt in a busy loop waiting for that specific interrupt. While the execution can continue in this situation, it does change the list of executed instructions. However, these false positives can also cause divergences if the window was too large.

The problem is to cause a VMX exit at the correct location. There are two values contained in the landmark usable for this task: the instruction pointer and the instruction count. Accordingly, we considered two methods to trigger VMX exits for event injection:

Breakpoints The x86 architecture has support for hardware breakpoints. The primary use of these is for debugging. When one of the configured breakpoints is hit, the processor jumps to the corresponding exception handler. This mechanism is also available for virtual machines and VMX can be configured to cause an exit when a breakpoint is hit. They can be used to debug the code running in virtual machines but it is also possible to use them to cause the necessary VMX exits for the asynchronous event injection. In both cases, the delivery of the exception to the virtual machine is suppressed, so the execution of the guest remains unmodified.

However, there are some problems with this approach. Because the configuration registers are part of the architectural state and the number of configurable breakpoints is limited to four, the guest cannot fully use the breakpoint functionality if the host is also using it. In addition, the debug functionality also interacts with other parts of the processor's behavior. For example, the normal x86 flags register contains a *resume* flag (`RF`). Before an instruction is executed, the processor checks whether there is an active breakpoint at the instruction's address. The CPU only raises a breakpoint exception if the resume flag is not set. [1] Using breakpoints to inject asynchronous events is thus problematic: In some cases, when the processor enters an exception or interrupt handler, it saves the flags register on stack with `RF` set [9]. When the interrupt handler returns control to the interrupted code, the flags register is restored from the value on the stack, which has the resume flag set. If an asynchronous event happens at this location, the CPU will ignore the breakpoint due to the set resume flag, thereby preventing asynchronous event injection and eventually breaking the replay. In addition, there are cases, in which it is impossible to trigger a breakpoint at a certain instruction. For example, if a `pop ss` instruction is executed, the delivery of breakpoint exceptions is inhibited [9].

Performance Monitoring Interrupt (PMI) The recording uses the performance counters to gather the instruction count for the landmarks. The `INST_RETIRED.ANY` counter can be accessed as a model specific registers. Further, the counter can also be configured to raise an interrupt on a counter overflow. By setting the counter to the maximum value minus n, a *performance monitoring interrupt* (PMI) can be scheduled after the measured event happened n times. This configuration is also commonly used for sampling profilers to regularly stop the observed program and sample the current instruction pointer. The PMI will not cause an interrupt handler to be called by itself. Therefore, the APIC needs to be configured to raise a *non-maskable interrupt* (NMI) when a PMI happens and VMX can be configured to exit when a NMI is delivered. Both VMX and the APIC are already configured this way in KVM because the perf subsystem of the Linux kernel relies on this configuration.

[1] Since the flag is cleared after each instruction, it allows debuggers to resume execution without removing the breakpoint.

Due to the mentioned shortcomings of breakpoints, we decided to use PMIs to exit non-root mode at the next landmark. This is done by comparing the current executed instruction count and the instruction count stored in the next asynchronous event. The difference between the two is the number of instruction to execute until the event location is reached. When an asynchronous event is scheduled for injection, the value of the performance counter is set right before the switch into VMX non-root mode.

However, the delay between the counter overflow and the interrupt arrival can be large [7]. Within this time frame, the processor can retire further instructions because of the out-of-order nature of modern processors. By the time the hypervisor handles the VMX exit, it may already be too late to inject the asynchronous event. A workaround for this problem is to stop the execution before the landmark is anticipated. We determined the size of this window experimentally to be around 120 instructions (see Section 6.4.1). The remaining instructions between the PMI interruption and the next landmark can be single stepped. As already mentioned, using the trap flag in the guest processor flags is too unreliable for this purpose. But there is a similar flag unaffected by these issues in the VMX extension, which is called *monitor trap flag*. It cannot be circumvented with the resume flag and this flag is not observable within virtual machine. Furthermore, the flag has another useful property: Whereas the normal trap flag steps over repeated string operations as if they were a single instruction, the monitor trap flag stops every iteration. This makes it possible to single-step through these single instruction loops, which is required to precisely inject asynchronous events.

Chapter 6

Evaluation

In the previous chapters we described our implementation of a replay engine for the three-stage parallelization approach. In this chapter, we evaluate our implementation. First, we describe our evaluation methodology in Section 6.1 and the environment in which all measurements were done in Section 6.2. In Section 6.3, we demonstrate that the simultaneous replay itself only causes a small run-time overhead on the recorded workload. Finally, we present the performance of our replay implementation, which is important for a proper parallelization, in Section 6.4.

6.1 Methodology

To assess the performance of our implementation we used the *phoronix test suite* [24] within the guest. The following benchmarks of the test suite were used:

phoronix-apache (v1.7.1) This benchmark starts an Apache web server and uses *Apache Benchmark* (ab) to send one million requests to the web server [2]. This benchmark is classified as system test. A notable observeration of this test is the high frequency of `rdtsc` instructions [28].

phoronix-encode-mp3 (v1.7.1) The benchmark encodes a WAV file into an MP3 file using LAME and measures how long this takes. Because there is only hardly I/O involved and LAME is mostly processor-bound, the benchmark is considered a processor test.

phoronix-gnupg (v2.4.0) Encrypts a 2 GiB file using GnuPG [16]. This benchmark is also a processor focused test.

phoronix-postmark (v1.1.1) This benchmark attempts to recreate the workload performed by web and mail servers [25]. It is doing I/O operations to 500 files across 25 thousand transactions. While the benchmark is primarily a disk test, it also modifies large amounts of memory.

phoronix-povray (v1.2.1) The benchmark renders a scene using the povray ray-tracer and is a processor test.

phoronix-pybench (v1.1.2) This benchmark runs the benchmark suite of the official Python project [26]. It can give an estimate of the performance of a typical Python program.

phoronix-sqlite (v2.0.1) This disk benchmark repeatedly inserts rows into a table of an SQLite database. This represents a light database workload.

phoronix-linux-kernel-build (v1.6.0) The benchmark measures how long it takes to compile the Linux 4.3 kernel [1]. The compilation involves I/O and the processor. Thus the benchmark is classified as a system test.

Additionally, we used the following workloads, which are not part of the phoronix test suite:

SPECjbb2005 (v1.07) This is a benchmark, which emulates a three-tier Java server commonly used in businesses [10].

Idle For this workload the system is left idle for $60\,\mathrm{s}$. Therefore, the system mostly waits for interrupts.

We did all measurements using automated tests scripts. Instead of using the score reported by the benchmarks themselves, we used the wall-clock run-time. The benefit is that it can be used for measuring the run-time overhead of both the recording and the replay. Using the benchmark score for evaluating the replay is not possible, because the score is simply replicated.

To assess the overhead of a configuration, we compare the measured run time to a baseline. The baseline is, unless stated otherwise, an execution in a VM with the modified KVM, but without recording and checkpointing. Even if the workload is not recorded, it will trap instructions such as `rdtsc` [28]. In comparison to a vanilla kernel this can reduce the performance of some workloads, which frequently uses the trapped functionality. Among those workloads are benchmark applications such as SPECjbb2005 or ab, which assume that `rdtsc` is a cheap operation.

All run-time overhead measurements were repeated six times. The 25% and 75% percentiles are represented as error bars in the plots.

6.2 Evaluation Environment

The hardware and software configuration, which was used for our measurements is shown in Table 6.1. Hyper-threading was disabled to avoid any disturbances of the performance counters, which are used for the record and replay process.

Component	Specifications
Processor	2x Intel® Xeon® CPU E5-2630 v3 with 2.40 GHz base frequency
Main Memory	64 GiB
Storage	Crucial CT256MX1, 256 GB
Guest Main Memory	2 GiB
Software	**Version**
Host Operating System	Ubuntu 16.04.6 LTS
Host Kernel	Linux 4.3.0, modified for SimuBoost and three-stage parallelization
Guest Operating System	Ubuntu 16.04.5 LTS
Guest Kernel	Linux 4.8.10
QEMU	2.6.50, modified for SimuBoost and this work

Table 6.1: The hardware and software used to evaluate the replay engine

Linux has the ability to choose from multiple clock sources to provide the current time. Among those sources are the High Performance Event Timer (HPET) and the CPU's timestamp counter, the latter also being the preferred source on modern x86 hardware. If Linux deems a clock source as unreliable, it can fall back to another one such as the HPET. However, these need to be emulated by QEMU. Because of the additional exit into user space, they can cause a high slowdown. Rittinghaus has observed this switch in Linux guests when being recorded. Therefore, QEMU was modified to not expose any clock sources to the guest except the timestamp counter [28].

6.3 Recording

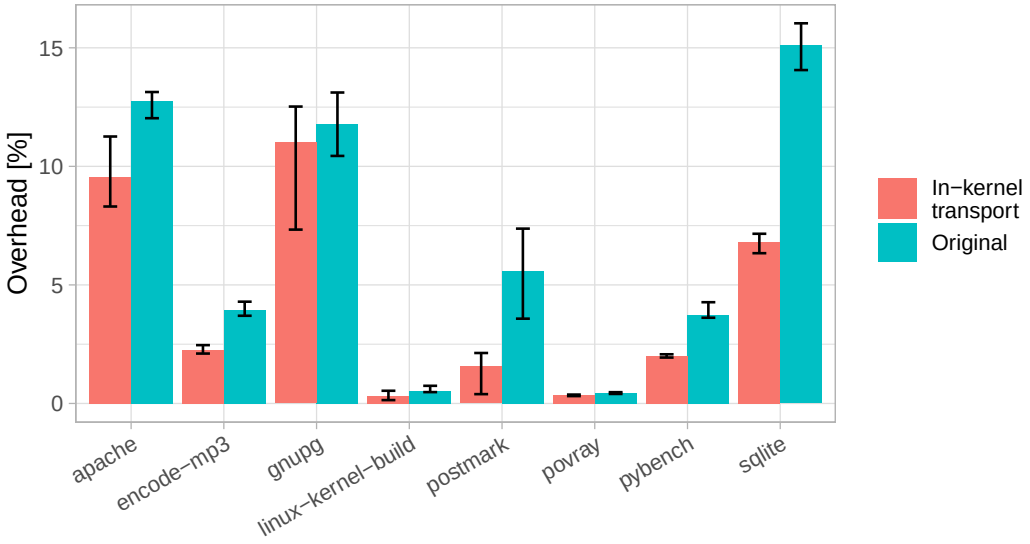

Figure 6.1: We compared the run-time overhead induced by our simultaneous replay with the in-kernel transport against the original implementation. The in-kernel transport reduces the recording overhead in all cases. In some benchmarks the overhead halves.

As described in Section 5, the recording was modified to transport the generated events within the kernel to the replay process. We measured the run-time overhead of the recording with these changes and compared them with the original version in Figure 6.1. In both cases the generated events were also stored in Simutrace. The event transport changes improve the recording overhead in all measured workloads. Some benchmarks such as SQLite and postmark incur only half of the overhead compared to the original implementation.

The SQLite database relies on `fsync` on the guest side to ensure the newly written data is persistent. This request is passed forward to the emulated disk by the guest kernel. Because the disk emulation happens in QEMU, a KVM exit is triggered. The original implementation collects all events after a KVM exit using an `ioctl`. The collection procedure copies the data into Simutrace segments and also deallocates the event and the buffer used for additional data. Because the benchmark involves many small insertions into a table, this happens very often. While this still occurs when the in-kernel transport is used, the additional `ioctl` call is no longer necessary on the recording side. This results in a lower overhead.

While this approach reduces the overhead of the event transport itself, it now requires a

second task to retrieve the events. In our implementation, the simultaneous replay takes care of persisting them. To investigate the effect of the additional load, we implemented two new modes in the recording engine. The first mode frees the event instead of pushing it into the shared queue. The second mode, starts a worker kernel thread, which merely fetches any incoming events in a busy loop. Any events received are directly freed. This approximates the behavior of the replay, without actually doing the replay itself. Additionally, we evaluate whether there is a difference between running the replay on the same CPU socket or on a different one. Figure 6.2 shows the results of these measurements.

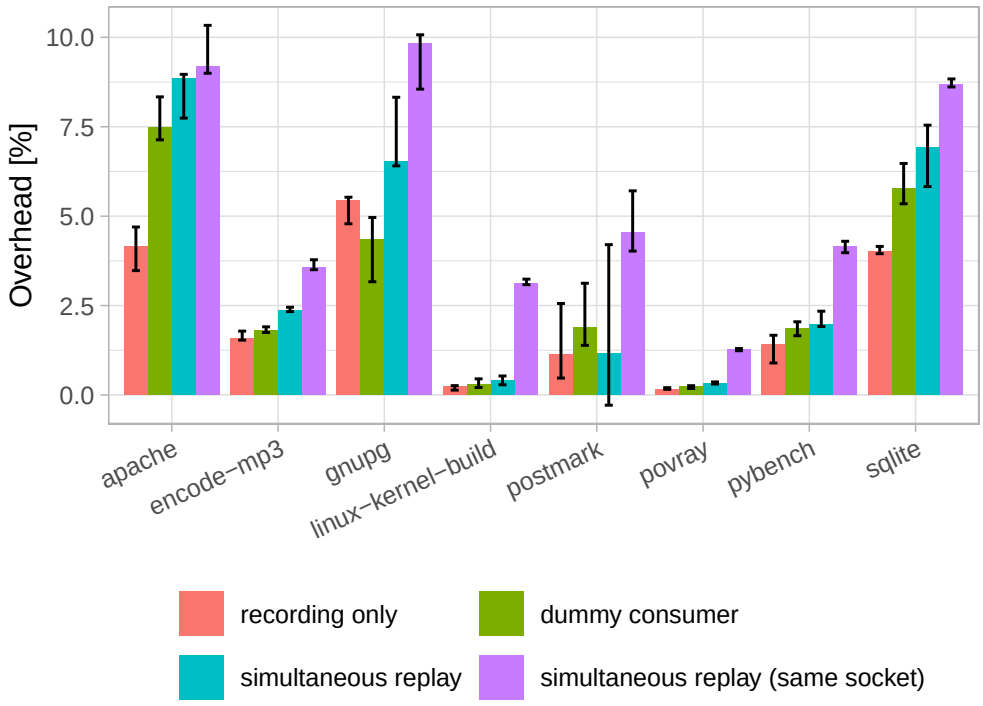

Figure 6.2: Probe effect on recording caused by simultaneous replay. The run-time overhead induced by the replay itself is about 2.5%.

The Apache web server workload is mostly slowed down by the queue, because all three variants involving the queue are much slower than the recording-only variant. The overhead caused by the queue is about 5%. The effect is also present in the SQLite benchmark albeit weaker. Running the replay on a separate CPU socket improves the performance considerably for the other workloads. This suggests there is a bottleneck within the processor, which can be avoided on dual socket systems.

6.4 Replay

In Section 3.2, we outlined the necessity of a replay engine based on a hardware-assisted virtual machine to offload the checkpoint generation. However, the analysis worker pool now also depends on the replay to produce enough work. If it is too slow, it can be the bottleneck of the setup. We address the replay performance in Section 6.4.2. It also needs to work correctly and must not fail, because the analysis cannot continue without the checkpoints.

6.4.1 Correctness

To verify whether the replay works correctly, we replayed all benchmarks and additionally SPECjbb2005, which proved to be valuable to expose bugs in the replay engine.

We mentioned previously in Section 5.2, that there is a delay between the performance counter overflow and interrupt arrival. To counteract this behavior, we schedule the PMI a certain number of instructions earlier — the *PMU window*. After the VMX exit, we continue single-stepping towards the target location using the monitor trap flag, while trying to inject the asynchronous event.

The single-stepping is very slow, because of the repeated VMX entries and exits. Therefore, it is important to keep the PMU window small. However, setting it too small can lead to missed injections. We determined the optimal window size experimentally: First, we set the window to a large value. Every time a performance counter overflow was handled, we inspect the value of the counter. In the optimal case, the value in the counter would be zero after an overflow happens, because then the execution would have stop immediately. However, as the processor executes further instructions until the overflow interrupt arrives, the value will be higher. By logging the value, it is possible to quantify how many instructions the target instruction was overshot.

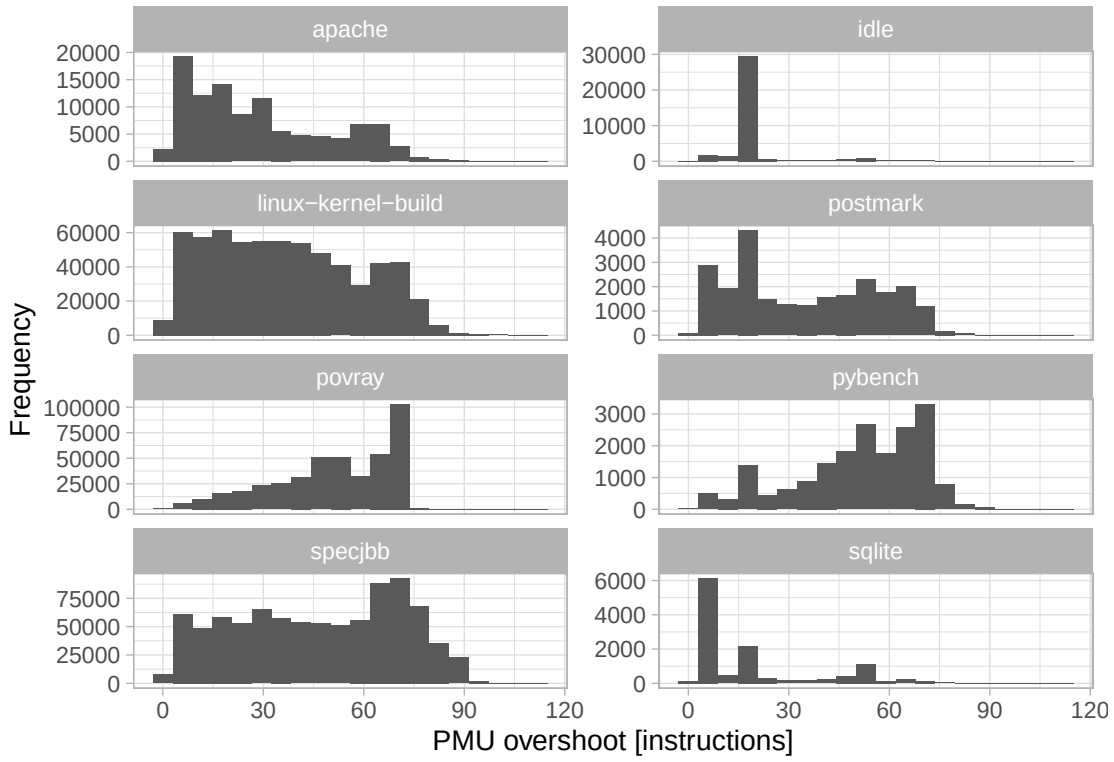

Figure 6.3: The histogram shows that the delay between performance counter overflow and interrupt arrival stays below 114 instructions.

Figure 6.3 shows the distribution of the delay in various workloads. All workloads cover a wide range of values, but none of them passed 120 instructions. We set the PMU window to this value instead of the actual maximum to include a safety margin. This window is likely micro-architecture specific and might have to be determined experimentally for other processor models.

Each of the workload creates a distinctive shape, which could indicate that the out-of-order execution of modern processors is the cause for the delay. The idle workload has a uniform instruction mix and does not fill the processor pipeline as much as the other workloads, because it is mostly waiting for (timer) interrupts. In contrast to other workloads, idle overshoots the target almost always by about 20 instructions. POV-ray and pybench, which are purely processor bound, incur a higher interrupt delay than the other benchmarks. Possibly, the processor pipeline is filled when the counter overflows, causing this delay.

6.4.2 Replay Overhead

To evaluate whether the replay can keep up with the recording, we measured the run time of both using the previously determined PMU window. Additionally, we measured some larger PMU windows to investigate the impact of the window size. The results of our measurements are shown in Figure 6.4.

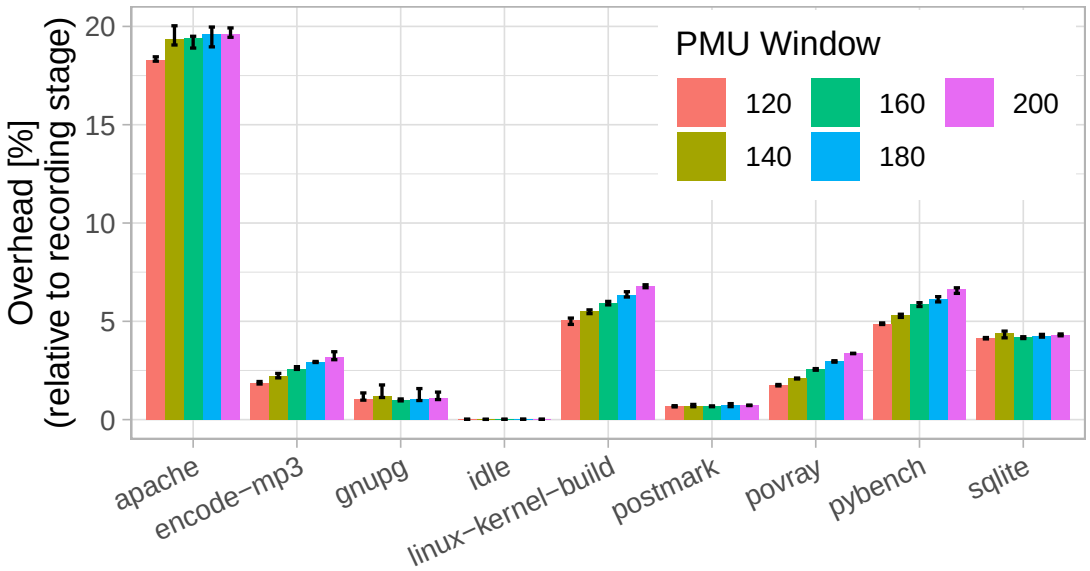

Figure 6.4: Our replay engine has an overhead of about 5% for most workloads compared to the recorded execution. Only Apache, which produces a large amount of events, incurs an overhead of 18%.

Idle represents the optimal case of a deterministic replay, because when the guest intents to wait for interrupts with a halt instruction (e.g., `hlt` on x86), the next interrupt can be injected immediately. Therefore, the replay should be able to catch up quickly after the boot sequence is completed. The measurements confirm this behavior. The Apache workload, which already had a large impact on the overhead while recording incurs the largest slowdown for this scenario.

Reducing the PMU window from 200 instructions to 120 instructions reduces the runtime overhead by about 2.5% for most benchmarks. Some benchmarks such as SQLite and Postmark are not affected of the window adjustments.

Because moving the event collection out of the recording already has a substantial effect on the overhead, we also investigated the impact on the replay. For this purpose, we modified the replay engine to free the events instead of queuing them for the collection after they have been successfully injected. While this configuration cannot be used for

the three-stage approach anymore, it is possible to gauge the performance of the replay in isolation. Figure 6.5 compares this modification with the normal version, which persists the events.

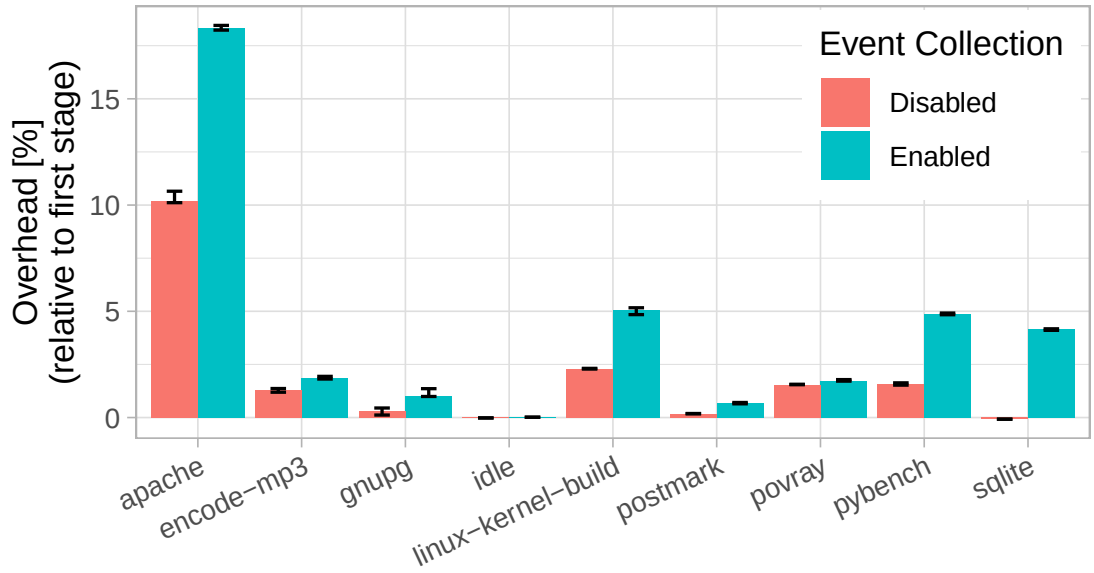

Figure 6.5: The event collection can impose a significant overhead on the replay. In some cases it more than halves.

The resulting measurements show that the event collection procedure also causes a considerable overhead on the replay. For example, without the event collection the SQLite workload is able to catch up with the recording. However, for processor-bound workloads such as POV-ray or encode-mp3 the modification improves the overhead by less than a percent.

6.5 Conclusion

While the primary goal of the in-kernel transport was to simplify the data movement for the replay, it is also beneficial for the overhead and probe effect in the recording. Copying the data to user space seems to be time-intensive enough that it is worthwhile doing it outside the recording process. We collect all events in the replay process but it incurs an additional overhead in the same way the recording previously did. While this does not result in a probe effect, it can have a detrimental effect on the parallelization of the simulation phase.

The replay itself affects the recording overhead slightly even if it is running on a separate processor. It might be beneficial for some workloads to avoid this effect by running the replay on a different system. But comparing the dummy consumer and the simultaneous replay on the same systems suggests that the yield of this approach may be only about 1%.

Chapter 7

Conclusion

Full system simulation is a powerful and flexible tool to gain in-depth insight into a system. The catch is that the simulation adds a large overhead. Depending on the information logged (e.g., memory accesses), the overhead usually increases even more. Further, the differences in timing cause a probe effect, which influences the system behavior and can distort measurements or even render them unusable.

Deterministic record and replay can be used to reduce this probe effect. By creating a lightweight recording of the non-deterministic events of a workload, enough information can be gathered to reconstruct the execution. The resulting log can be used to replay the workload in a full system simulator. This way the large slowdown of a FSS does not cause a probe effect on the workload.

The analysis speed is largely limited by its naturally single-threaded nature. SimuBoost tries to address this slowdown by using continuous checkpointing to break the recording into independent segments. Each of the created checkpoints serves as the starting point of an analysis by a separate worker.

However, continuous checkpointing introduces a significant additional run-time overhead on top of the run-time overhead of the lightweight recording, resulting in a probe effect. A promising approach to remove the checkpointing overhead from the recording is to use a three-stage setup. The new stage between the record and analysis stage replays the recording in a hardware-assisted virtual machine and is responsible for the checkpoint creation. In this stage, the checkpointing does not cause any probe effect. We showed that this replay has an overhead of less than 5% for many workloads compared to the run time of the recording itself.

7.1 Future Work

While we implemented the necessary components to use the three-stage approach, we did not evaluate the full setup. It remains open how much of an effect the replay overhead has on the parallelization in SimuBoost.

An interesting observation was the noticeable effect of the event collection (see Section 6.4.2). This could be improved by using an additional thread to collect the replayed events. A queue similar to the in-kernel transport queue described in Section 5.1 could be used for this purpose.

Bibliography

[1] *Apache Benchmark 1.6.0*. URL: https://openbenchmarking.org/innhold/dda952e40113b276e205198ef88adb6007eee885 (visited on 09/09/2019).

[2] *Apache Benchmark 1.7.1*. URL: https://openbenchmarking.org/innhold/e506cd77d5e3299aba7157128e0321596d4cf797 (visited on 09/09/2019).

[3] Fabrice Bellard. „QEMU, a Fast and Portable Dynamic Translator.“ In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005, pp. 41–41. URL: http://dl.acm.org/citation.cfm?id=1247360.1247401.

[4] Dean Michael Berris et al. *XRay: A Function Call Tracing System*. Tech. rep. A white paper on XRay, a function call tracing system developed at Google. 2016.

[5] Nico Boehr. „Evaluating Copy-On-Write for High Frequency Checkpoints.“ Bachelor Thesis. Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, Sept. 2015.

[6] K. Buchacker and V. Sieh. „Framework for testing the fault-tolerance of systems including OS and network aspects.“ In: *Proceedings Sixth IEEE International Symposium on High Assurance Systems Engineering. Special Topic: Impact of Networking*. Oct. 2001, pp. 95–105. DOI: 10.1109/HASE.2001.966811.

[7] Anton Burtsev et al. „Abstractions for Practical Virtual Machine Replay.“ In: *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '16. Atlanta, Georgia, USA: ACM, 2016, pp. 93–106. ISBN: 978-1-4503-3947-6. DOI: 10.1145/2892242.2892257. URL: http://doi.acm.org/10.1145/2892242.2892257.

[8] Intel Corporation. *Intel® 64 and IA-32 Architectures Developer's Manual*. May 2019. URL: https://software.intel.com/sites/default/files/managed/a4/60/325383-sdm-vol-2abcd.pdf (visited on 09/10/2019).

[9] Intel Corporation. *Intel® 64 and IA-32 Architectures Developer's Manual*. May 2019. URL: https://software.intel.com/sites/default/files/managed/a4/60/325384-sdm-vol-3abcd.pdf (visited on 09/09/2019).

[10] Standard Performance Evaluation Corporation. *SPEC JBB2005*. URL: https://www.spec.org/jbb2005/ (visited on 09/10/2019).

[11] Brendan Cully et al. „Remus: High Availability via Asynchronous Virtual Machine Replication." In: *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. NSDI'08. San Francisco, California: USENIX Association, 2008, pp. 161–174. ISBN: 111-999-5555-22-1. URL: http://dl.acm.org/citation.cfm?id=1387589.1387601.

[12] Brendan Dolan-Gavitt et al. „Repeatable Reverse Engineering with PANDA." In: *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*. PPREW-5. Los Angeles, CA, USA: ACM, 2015, 4:1–4:11. ISBN: 978-1-4503-3642-0. DOI: 10.1145/2843859.2843867. URL: http://doi.acm.org/10.1145/2843859.2843867.

[13] Pavel Dovgalyuk. „Deterministic Replay of System's Execution with Multi-target QEMU Simulator for Dynamic Analysis and Reverse Debugging." In: *2012 16th European Conference on Software Maintenance and Reengineering*. Mar. 2012, pp. 553–556.

[14] George W. Dunlap et al. „Execution Replay of Multiprocessor Virtual Machines." In: *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '08. Seattle, WA, USA: ACM, 2008, pp. 121–130. ISBN: 978-1-59593-796-4. DOI: 10.1145/1346256.1346273. URL: http://doi.acm.org/10.1145/1346256.1346273.

[15] George W. Dunlap et al. „ReVirt: Enabling Intrusion Analysis Through Virtual-machine Logging and Replay." In: vol. 36. SI. New York, NY, USA: ACM, Dec. 2002, pp. 211–224. DOI: 10.1145/844128.844148. URL: http://doi.acm.org/10.1145/844128.844148.

[16] *GnuPG Benchmark 2.4.0*. URL: https://openbenchmarking.org/innhold/e5631bf709cafbe64906a764728f235faa2088ed (visited on 10/07/2019).

[17] *Goodbye, Replay Debugging...* Sept. 2011. URL: http://www.replaydebugging.com/2011/09/goodbye-replay-debugging.html (visited on 09/20/2019).

[18] Advanced Micro Devices Inc. *AMD64 Architecture Programmers's Manual*. July 2019. URL:

https://www.amd.com/system/files/TechDocs/24593.pdf (visited on 09/09/2019).

[19] Samuel T. King, George W. Dunlap, and Peter M. Chen. „Debugging Operating Systems with Time-traveling Virtual Machines.“ In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005, pp. 1–1. URL: http://dl.acm.org/citation.cfm?id=1247360.1247361.

[20] P. S. Magnusson et al. „Simics: A full system simulation platform.“ In: *Computer* 35.2 (Feb. 2002), pp. 50–58. DOI: 10.1109/2.982916.

[21] Nicholas Nethercote and Julian Seward. „Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation.“ In: *SIGPLAN Not.* 42.6 (June 2007), pp. 89–100. ISSN: 0362-1340. DOI: 10.1145/1273442.1250746. URL: http://doi.acm.org/10.1145/1273442.1250746.

[22] Robert O'Callahan et al. „Engineering Record and Replay for Deployability.“ In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, July 2017, pp. 377–389. ISBN: 978-1-931971-38-6. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/ocallahan.

[23] Daniela A. S. de Oliveira et al. „ExecRecorder: VM-based Full-system Replay for Attack Analysis and System Recovery.“ In: *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*. ASID '06. San Jose, California: ACM, 2006, pp. 66–71. ISBN: 1-59593-576-2. DOI: 10.1145/1181309.1181320. URL: http://doi.acm.org/10.1145/1181309.1181320.

[24] *Phoronix Test Suite - Linux Testing*. URL: https://www.phoronix-test-suite.com/ (visited on 10/07/2019).

[25] *PostMark Benchmark 2.4.0*. URL: https://openbenchmarking.org/innhold/7875f3df74d35e5aa0b08ceb067f31c888c1c624 (visited on 10/07/2019).

[26] *PyBench Benchmark 2.4.0*. URL: https://openbenchmarking.org/innhold/f28e7b617e3c4614a76336ed4097fcf6ce94918b (visited on 10/07/2019).

[27] *QEMU*. Oct. 1, 2019. URL: https://www.qemu.org/.

[28] Marc Rittinghaus. „SimuBoost: Scalable Parallelization of Functional System Simulation.“ PhD thesis. Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, 2019.

[29] Marc Rittinghaus, Thorsten Groeninger, and Frank Bellosa.
Simutrace: A Toolkit for Full System Memory Tracing. White Paper.
Karlsruhe Institute of Technology (KIT), Operating Systems Group, May 2015.

[30] Marc Rittinghaus et al.
„SimuBoost: Scalable Parallelization of Functional System Simulation.“
In: *Proceedings of the 11th International Workshop on Dynamic Analysis*.
Houston, Texas, Mar. 2013.

[31] Deepa Srinivasan and Xuxian Jiang.
„Time-Traveling Forensic Analysis of VM-based High-Interaction Honeypots.“
In: *Security and Privacy in Communication Systems*.
Springer Berlin Heidelberg, 2012, pp. 209–226.

[32] Michael H Sun and Douglas M Blough.
Fast, lightweight virtual machine checkpointing. Tech. rep.
Georgia Institute of Technology, 2010.

[33] *The Definitive KVM (Kernel-based Virtual Machine) API Documentation*.
URL: `https://www.kernel.org/doc/Documentation/virtual/kvm/api.txt` (visited on 09/09/2019).

[34] Dmitry Vyukov. *Unbounded SPSC Queue*.
URL: `http://www.1024cores.net/home/lock-free-algorithms/queues/unbounded-spsc-queue` (visited on 09/10/2019).

[35] V. M. Weaver, D. Terpstra, and S. Moore. „Non-determinism and overcount on modern hardware performance counter implementations.“
In: *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Apr. 2013, pp. 215–224.

[36] Stefan Weil. *QEMU version 4.1.0 User Documentation*.
URL: `https://qemu.weilnetz.de/doc/qemu-doc.html` (visited on 08/30/2019).

[37] Min Xu, Rastislav Bodik, and Mark D. Hill. „A "Flight Data Recorder" for Enabling Full-system Multiprocessor Deterministic Replay.“
In: *SIGARCH Comput. Archit. News* 31.2 (May 2003), pp. 122–135.
ISSN: 0163-5964. DOI: `10.1145/871656.859633`.
URL: `http://doi.acm.org/10.1145/871656.859633`.

[38] Min Xu et al.
„Retrace: Collecting execution trace with virtual machine deterministic replay.“
In: *Proceedings of the Third Annual Workshop on Modeling, Benchmarking and Simulation*. 2007.