Karlsruher Institut für Technologie

Towards Heterogeneous Deterministic Replay for Symmetric Multiprocessors

Masterarbeit
von

Michael Zangl

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Frank Bellosa
Zweitgutachter:	Prof. Dr. Wolfgang Karl
Betreuender Mitarbeiter:	Marc Rittinghaus

Bearbeitungszeit: 1. Juni 2017 – 12. November 2017

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

Karlsruhe, den 12. November 2017

Abstract

A program can be recorded and then deterministically replayed. This is used for debugging applications and detecting the cause of bugs or errors after they were observed.

During recording, all non-deterministic events are stored for later replay. Upon replay, this allows the program to behave exactly like the recording, down to the instruction level. A detailed analysis of the program is possible during replay. To improve the analysis results, it can be done in a different environment than the recording while preserving the exact instruction flow.

In this master thesis, the ability to extend a heterogeneous record and replay system with multi-core support is evaluated. To make memory race conditions deterministic, a chunk based protocol is evaluated with which a transactional memory is simulated in the hypervisor. No hardware modifications are required for this.

A prototype is implemented based on QEMU that records a virtual machine and replays the recording in an emulator. It is shown that recording on a virtual machine causes low memory overhead and acceptable performance overhead. The scalability of the system is shown to be good for processor numbers that are currently used in virtual machines, although for high numbers of processors this solution does not scale. It is validated that a correct replay is possible in an emulated environment.

Contents

Chapter 1

Introduction

Debugging programs often requires to go back in time and to trace the real cause of an error. This can be the case for a normal program bug that is only recognizable by the errors it causes in the following program execution. In this case, finding the source of it is difficult. It may be useful to re-execute the program and trigger the bug again. But reproducing the bug is not always possible for the user, since small differences in program execution may lead to the bug not being triggered. The same way, finding the entry path into the system of malicious applications proves difficult, since those programs can alter system logs and other information after they have infected a system. Once the user realizes that such a malicious application is or was on the system, it may not be possible any more to find out where it came from and what the application did. A record and deterministic replay system that records the running system and then allows to re-simulate the execution down to the instruction level can be used to debug such defective or malicious applications [4].

When recording a system, all non-deterministic events are captured. They are written to a log which is used during replay to make the replay deterministic. To allow for high performance record and replay, hardware support can be used for recording. This is done using virtualization extensions of modern CPUs and running the system that should be analyzed as a virtual machine. During replay, the same hardware and the same environment can be used in order to ensure correctness [10, 19].

But software emulation allows for a more detailed program analysis than hardware virtualization [4]. So to allow the recording to be run in background with high performance while having powerful tools to analyze it, a heterogeneous approach can be used. The recording of the system is done in a virtualized environment while the replay is done in an emulator.

For single-core systems, this has been implemented in V2E [24]. But modern processors have multiple processors to increase performance through parallelism.

When executing programs in parallel, new problems occur. A common problem are memory race conditions. Small differences in the time at which instructions are executed on different cores may cause differing program results, including crashes. Those problems cannot easily be reproduced since they do not occur deterministically. To analyse them, record and replay can be used as well. This is why a record and replay solution should support tracing and replaying multiple processors. Such a multi-core record and replay has not yet been implemented in a heterogeneous environment. Evaluating the possibility of such an implementation will be the scope of this work.

Chapter 2

Background

This work is on recording and replaying applications in virtualized environments. For this, the a basic knowledge of virtual environments is required. Special attention is on the support for multiple processors and memory synchronization, since this will be one key aspect of this work.

2.1 Virtualization

Originally, an operating system is meant to run on the bare hardware. The operating system expects to have full control over the hardware. This prevents multiple operating systems from running in parallel.

For debugging purposes, it is often required to pause an operating system and to access it's memory. In this case, there needs to be an mechanism that takes care of the hardware in those situations. One solution to this problem is to run an other operating system on the hardware and then run the system that should be analyzed in a virtual, simulated environment.

2.1.1 Software Emulation

An emulator is a software that simulates a the complete computer hardware. It allows the user to specify virtual IO devices like a serial console or a hard drive that this computer then uses. The emulator then simulates a system start from that hard drive. All instructions the CPU would execute to do this are interpreted in software.

This full system emulation allows a operation system and normal applications to be run inside the emulator without any modification. The program behaves exactly as it would on a real system.

Since the emulator is written in software, it can be instructed to perform additional operations in addition to simply running the program. This includes debugging the application by pausing on a specific breakpoint. The current state of the emulated CPU can then be inspected, including all register contents and the hardware state. That way, full systems can be debugged without the need for hardware debuggers.

The emulator can even do more complex operations like tracing all memory accesses and analyzing them.

The big advantage of software emulators is their flexibility and the ability to run any target architecture on any host architecture. An example for such an emulator is QEMU running in the TCG mode [3].

Multi-core Emulation

There are two approaches to emulate multi-core applications.

The first approach is to run the processors in sequence and frequently switch between them [3]. This approach is known from classic multi tasking applications. From a guest application point of view, all processors make progress. Since the guest system usually does not expect an exact timing of the inter processor communication, it will still work as it normally would on a full parallel system.

The switch may not happen during instructions that are considered atomic on the emulated platform. A memory write of an 64 bit word is an example for such an instruction on 64 bit systems. The emulator might need several native instructions to emulate this write. During those instructions, the memory state may only be half updated. To prevent this, a switch between processors is only possible at the end of such instructions.

Since the emulator has full control over the environment, this is not a problem in practice. The emulator does not rely on external interrupts to do the switch but it can instead use an internal counter that counts the instructions executed on the current core. It can switch to an other core after executing a given number of instructions.

If the switch is not frequent enough, programs that rely heavily on inter processor communication may be behaving differently. A spin lock for example may block the program execution by a long time until the emulator switches to the processor holding the spin lock. That way, real parallel program execution cannot be emulated reliably.

An alternative to the sequential emulation is to emulate all processors in parallel on different host processors. [6, 8] For this, each host processor is emulating one target processor. The host processors use the inter processor communication mechanisms of the host to simulate inter processor communication of the emulated system.

Special care needs to be taken for memory accesses. Memory accesses on the host system may not have the same granularity as the ones on the target system. This may require the host system to use special locking in order to guarantee the atomic instructions of the guest system to be atomic on the host system.

The emulator state synchronization needs to be taken care of as well. Pausing the emulator to analyze it's state requires to pause all processors simultaneously. This requires a special management layer for the emulator in which it manages the processors and pauses or resumes them as needed.

Emulation Performance Optimizations

While software emulation allows for an exact simulation of the target system, it has a major disadvantage in speed. CPUs can perform complex operations in one cycle. To simulate those operations, multiple CPU instructions on the host system may be required. Emulating hardware access adds additional CPU time which would not be required on the target system. This is why software emulation is significantly slower than natively running the program.

There are various optimizations available to counter this. The most important one is a just in time compilation of the guest code. The code is not interpreted but instead dynamically translated to machine code for the host system. Using this method, a program still needs several times longer to run than it would natively. [3]

An other major performance disadvantage of software emulation is the access to hardware. System hardware runs in parallel to the main processor. The program flow of a normal program can continue while the hardware is doing work. A software emulator needs to emulate this hardware. This simulation is done in software and is thus running on the CPU. Therefore, the CPU cannot continue to emulate the guest program.

To improve this situation, the hardware access can be translated instead of emulated. It then uses the resources of the host hardware and allows the CPU to continue with the guest program. This makes hardware accesses - like disk reads - asynchronous on the emulated system. While this is the same behavior as on the host system, it is an unpredictable behavior since the hardware access time is not deterministic.

2.1.2 Hardware Assisted Virtualization

While software emulation allows for a lot of flexibility, it has the major disadvantage that it is several times slower than native execution. To avoid this issue and allow for a new native performance when using virtualisation, hardware manufacturers added virtualisation support. This allows to run unmodified guest systems directly on the host processor.

Those hardware extensions allow the guest operating system and it's application to use the CPU. The virtual memory layout can be controlled by the guest system, although it does not have complete access to the physical memory.

The part of the host system that controlls the virtual machines is called the hypervisor. From a user perspective, it can be used in a similar way the emulator is used to run the guest system. Instructions that operate on the virtual memory can run without the interference of the hypervisor. When the guest operating system accesses hardware, the hypervisor needs to filter the request to emulate a real hardware access as if the guest operating system was running directly on the hardware.

Hardware virtualisation requires the guest program to use the correct instruction set architecture for the processor. It is not possible to run programs that were compiled for other architectures.

Guest Memory Access

While the guest system is running, will use instructions that access the main memory. The guest system should have it's own, isolated memory area with contiguous addresses starting at 0 for this.

If the guest would be given direct access to the host memory, this would lead to conflicts. Therefore, a translation layer is added that translates between guest physical and host physical addresses.

On modern systems like the Intel processors with virtualisation extensions, this is done by mapping the memory addresses on a page granularity level. A page table like the one for normal applications is used to map the addresses.

This table is called the extended page table (EPT). It is set up and controlled by the virtual machine monitor. The guest can itself use the normal page table to isolate it's applications, which leads to a two level address translation as shown in Figure .

The guest operating system itself uses page tables to distribute the guest physical memory among it's processes. This lads to a two level translation in which the virtual address of the guest process is mapped to the guest physical address first and then to the host physical address, as described in Figure 4.1. Non-virtualized processes running directly on the host do not need this two stage address translation. For them, the extended page table is not used.

The flags of the EPT and the guest page tables are independent from each other. If the guest accesses a page, the corresponding access flags of both tables are set. The guest handles the access flags in it's page table. Since the EPT cannot be seen by the guest, the access flags of the EPT can be evaluated and reset by the host. The same applies to the dirty flag on page writes. In the Intel implementation of the EPT, the access flags are set hirarchically on all page table levels. The dirty flag is only set on the last level of the EPT.

The EPT has read, write and execute flags as well. When a guest attempts

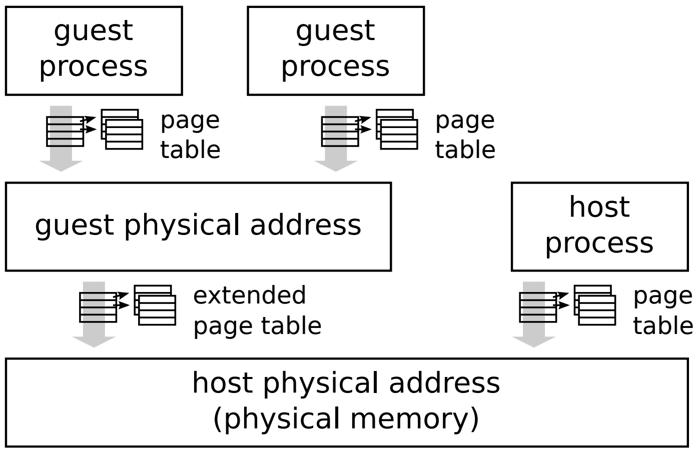

Figure 2.1: The memory addresses of guest applications are mapped to the guest physical address. This memory address is then mapped to the host physical address. Hardware can provide full support for this two stage process.

to perform any operation on a page, those flags are evaluated in the guest page table first. If a violation is detected by the hardware, the processor traps. This trap is forwarded to the guest operating system to handle the page fault. After the permission tests for the guest page table have passed, the flags in the EPT are evaluated. If there is a violation in those flags, the host operating system needs to handle that page fault. It can then fix that corresponding page table entry and re-evaluate the page table miss to provide a page to the client or it can emulate an invalid memory access as if the guest would have accesses a non-existing physical address.

Current hypervisors like KVM are built to run unmodified guest operating systems. Those systems expect to have a contiguous memory starting at an address near 0 and ending at a fixed size. This is why those hypervisors are designed to give the guest system a large, contignuous block of memory. This restricts the largest guest physical address to the total amount of memory given to that guest.

However, as virtualisation becomes more common, alternative approaches use the flexibility of not running directly on the hardware to give sparse memory to the guest system. This allows the guest to make use of the full set of theoretically available physical addresses on modern systems. [2]

Trapping

Whenever the hardware is not able to directly handle an instruction the guest system executes, it traps into the hypervisor. The hypervisor is then able to handle this instruction of the guest system.

Most instructions a guest application executes do not trap and can completely be handled by the hardware. This includes memory accesses and some accesses to machine specific registers.

Hardware accesses cannot be handled by the hardware itself directly because there may be many guest operating systems running in parallel. This is why every access to the hardware traps into the hypervisor. The hypervisor can then decide to pass the request on to the real hardware or to emulate the hardware feature.

An example for this is network traffic: If a guest wants to send a network package, it communicates with a virtual network device. That device is simulated by the hypervisor. As soon as the guest instructs that virtual device to send the package, the hypervisor decides whether to pass it on to the real network interface or to handle the send request internally and e.g. send it on to an other virtual machine.

An other emulated instruction is the CPUID instruction. This instruction allows to get the id of the current core. Since a guest application can have a different view of the cores and this view should be transparent to the guest, the id given to the guest needs to be determined by the hypervisor. It returns the id of the virtual CPU (vCPU) in this case.

Chapter 3

Analysis

The more complex applications get, the more complex it gets to reproduce errors and trace the source of errors. To debug such an application in which erroneous or malicious behavior was observed, a record and replay approach can help. Using this, the application is recorded. The recording can then be executed again to observe and analyze the exactly same program behavior.

3.1 User-Space Record and Replay

Conventional debuggers allow the user to set breakpoints to pause the program on a specific condition. The program can then be inspected and the execution can be resumed. They only allow the user to see the current program state and then to go forward in time. It is not possible to go back in time. If an error did not directly trigger the breakpoint but only the later effects of the error did, the cause of the error cannot be examined any more.

A solution to this problem is to find a test case that always triggers the error and then to run this test case and set breakpoints earlier in the instruction flow.

For modern, complex software, not all bugs can be reproduced. This especially applies to fuzzing, a modern testing method in which the program is fed with random input and the program behavior is examined. If a fuzzer triggers a race condition or any other rare problem, the error may not happen in the next run even if the same input parameters are used.

For this, developers require debuggers that can go back in time [4, 15]. Such debuggers allow the analysis of a previous program state. They record a program by storing the initial program state. The debugger then stores every non-deterministic behavior of the application. That way, the debugger can replay the exact program flow starting at a program start or a checkpoint. To go back in time, the debugger restores the last checkpoint before the desired time frame and then replays the program until the desired state is reached.

One of those user-space record and replay systems is Jockey [21]. It runs on an unmodified Linux by linking it into an existing binary. It then intercepts all library calls of that binary that are non deterministic. On a Linux system, those are the wrappers for system calls. All other program execution is expected to be deterministic. This limits Jockey to single-core applications that do not use any form of shared memory and that do not receive external signals.

The problem with receiving asynchronous events like signals is that they need to be replayed at the exact same time they occurred during recording. A user-space recording library has no possibility to record an exact time reference at which the signal occurred. This prevents an exact insertion of the signal in the replay stream.

The second problem that Jockey faces is with concurrent memory access. If there are multiple threads in the running program or if there are multiple processes working on a shared memory segment, the order in which they access the shared memory must be recorded to make a deterministic replay possible. This requires context switches to be recorded. This would only be possible for a user-space, non-preemptive threading library. Any preemptive thread switch would be an asynchronous event that cannot be recorded in user-space.

Recording multiple processes at once and recording their interactions or even recording a whole operating system is not possible, since there is no shared timing information between the processes. That way, no synchronization during replay is possible and each process could only be replayed independently.

3.2 Full System Record and Replay

An alternative to recording only a single process is to record the whole system that process runs in. Such a recording would include the process itself but would also allow to debug it's operating system interactions and the communication with other processes. This has the advantage that it is more platform independent since it does not rely on a target application interface. In addition to this, it may cause a lower overhead since there is no need to include the contents of inter-process communications or the communication between a process and the operating system.

3.2.1 Emulation

An emulator allows the implementation of record and replay relatively easily. Since the emulator is completely simulating the target system, it has full control over it. All interrupts are triggered by the emulator and all communication of the target system is doen through the emulator.

A naive approach would be to let the emulator record every single instruction it executes. This would lead to a very big log file and a lot of logging overhead. Therefore, instructions that are fully deterministic given the current emulator state do not need to be recorded. This includes instructions that read or write normal memory, arithmetic instructions and branching instructions.

For recording an emulated system, all actions of the emulator that do not depend on the current target state only need to be logged. This includes the times at which the emulator decides to trigger an interrupt vector and the results the emulator returns for IO operations that are mapped to external hardware like network cards.

During replay, the interrupts are then inserted at the same place in the instruction flow. Since the emulator has full control over the target and emulates every single instruction, it can count the instructions to determine the right place to insert the interrupt.

Other events that were synchronous to the target execution are triggered by it in the replay at exactly the same time as they were during recording. For those events, the resulting operation of the emulator is read from the log. That way, the replay behaves exactly as the recording system did.

3.2.2 Virtualization

The problem with emulators is that they are slow in comparison to running the program on the original hardware. To trigger rare bugs that would require a replay mechanism to reproduce them, the program needs to run a relatively long time. It would even be best to run the recording in a production environment to be able to trace bugs in production. Using an emulator there is not an option, since the slowdown would severely reduce the throughput of the production system.

To get a performance close to a system that is running natively on the hardware, virtualization can be used. A virtualized system can be recorded at low memory overhead and a speed close to the one without recording [10, 23]

An additional disadvantage of emulators is that malware often contains mechanisms to detect an emulator and refuses to work in this environment. Virtualized environments are used for web servers, so malicious software often targets those environments. Therefore, no preventive measures against virtual machines are implemented in many such applications [7].

In a virtualized environment, the hypervisor manages all communication of the virtual machine with the outside world. If designed correctly, the hypervisor is the only source of indeterministic behavior of the virtual machine. Therefore, no modifications to the guest system are necessary to be able to do a full system record and replay.

During recording, a sequential log of hypervisor events is generated. Those events record the non-deterministic input values and the timing and values of non-deterministic events. Hardware performance counters are used to count the number of instructions the virtual machine executed. That way, the exact time of asynchronous events can be recorded.

Since all asynchronous events are delivered by the hypervisor by default, there is no additional interrupt or other hardware interception required. The event can directly be stored by the hypervisor in it's memory space. This allows a recording overhead of approximately 5% on typical workloads [23].

During replay, the virtual machine is restored to the initial state. Then, it is resumed. Since the virtual machine state is exactly the same as during the recording, all future events are guaranteed to happen at exactly the same state during replay. Whenever the virtual machine makes a synchronous call that involves the hypervisor that requires a non-deterministic input value, that value is read from the recorded log.

Asynchronous events are inserted by instructing the CPU to pause the guest at a given instruction count. This is the same count that was stored in the recording log. When the CPU hits that instruction count, the hypervisor is invoked. The hypervisor reads the event information from the log and simulates it for the guest.

3.2.3 Heterogeneous Record and Replay

In a virtualized environment, recording introduces only a low overhead. But during replay, there are no complex analysis tools available. For debugging, it is required to extend the replay mechanism and run further analysis on the program [5, 22, 23].

To combine the advantages of virtualisation during recording and emulation during replay, a heterogeneous approach can be used. The recording is done in a virtualized environment. The emulator is configured to exactly match that environment. This is not trivial, since the emulator needs to be able to emulate all instructions of the source platform for every possible input parameters exactly, including the flags registers and the unspecified behavior [6]. Current implementations have slight differences between the real hardware and the emulator. This is the case for instructions that are not used by the C compiler or instruction result bits that are usually ignored [13]. So for normal applications, there should be no difference in the execution due to those differences in interpreting the instructions. But a exact replay cannot be guaranteed. It is then loaded with the initial state of the virtual machine. During replay, the events to be emulated are read from the replay log.

In *Decoupling dynamic program analysis from execution in virtual environments*, a ht erogenous record and replay solution called *Aftersight* has been implemented and evaluated. The new aspect here is that the replay environment differs from the source environment since replay can be done on an other machine. In the previous virtual machine to virtual machine case, it was sufficient to record the input parameters to external hardware as long as that hardware would behave deterministically. Since the replay is done on a different hardware now, those hardware components may not be available or may not behave exactly as they did on the source system. To solve this problem, the output of the hardware needs to be recorded as well [24].

It has been shown that a correct replay can be achieved for full modern operating systems like windows or Linux. The recording overhead was approximately 10% for a Linux kernel compilation.

Aftersight was used to detect bugs in the Linux kernel. One of their achievements was the detection of a uninitialized stack use in the network stack that was present there for several years.

3.3 Multicore Record and Replay

The previous projects focused on record an replay on single-core systems. With the arise of systems that contain multiple CPUs, new problems arise. For those systems, the relative order of events that may influence other processors nees to be recorded [12, 17]. The non-deterministic effects of concurrent memory access is the greatest challenge.

3.3.1 Software Memory Tracing

An emulator can run multiple CPUs in parallel by running them in sequence and periodically switching between them. The emulated system has the impression that all CPUs are making constant progress. Since there is no actual parallel execution, memory race conditions cannot happen. The emulator records the times at which it switches processors during recording. During replay, the processor is switched at exactly the same time. This ensures a consistent replay.

The disadvantage of this approach is that it makes no real use of multi-core hardware and adds to the already present slowness of the emulator.

A faster emulation can be achieved by running the emulated CPUs on parallel physical CPUs. [8] The memory model of the emulator needs to allow concurrent access, especially for atomic instructions like the test-and-set instruction. This can be achieved by synchronizing the access of the processors to the memory.

The problem with this is that the synchronization is non deterministic. An example for this is if a guest uses a spin lock, there is no guarantee which emulated processor will get the lock. To ensure a consistent replay, this information needs to be recorded.

Since the emulator does not know about the semantics of a memory access the target system does, the emulator needs to record the order of every single memory access of the application to replay them in the correct sequence. Creating one log entry for every memory access and synchronizing every single memory access would result in a big log size and in a huge recording time overhead.

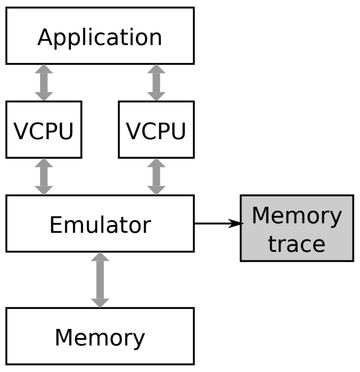

Figure 3.1: The emulator intercepts all memory accesses and traces them.

During replay, the emulator only needs to ensure the correct ordering of the single memory accesses. Each CPU is paused before the memory access until the time for this particular access is reached. This makes a replay straight forward in this situation.

While this ensures correct recording and replay, it adds a lot of performance overhead to memory accesses. Since every executed instruction needs to be fetched from memory, this adds a lot of performance overhead. Even when using a reader-writer lock that allows for parallel and low-overhead read locks, there would be a lot of writes to memory cells that would need to be locked. There would also be a global clock that is shared by all processors and that would ensure the correct ordering of memory accesses. All this reduces the scalability of this approach and reduces the performance benefits from using multiple processors.

The disadvantage can be counteracted by grouping multiple memory accesses. Those groups of instructions are called chunks by most implementations [14, 18, 19]. The memory is then not locked for each single memory access but for the chunk as a whole.

A global time frame those chunks are referenced against is introduced [16]. If a processor does a memory access that may cause conflicts with other processors, that access is detected. On each such access, the global time is captured and increased. This denotes the end of a chunk. During replay, the instructions that were part of the chunks are replayed in the same order in which they were encountered during recording.

While this reduces the size of the record, it still introduces a lot of overhead since the emulator needs to check every single memory access for collisions. This check is done in software and adds several CPU instructions per emulated instruction.

3.3.2 Hardware Assisted Memory Tracing

To avoid the significant performance overhead of emulation, the memory tracing mechanisms can be implemented in hardware. The computer hardware is extended by modules that trace the interleaving memory accesses for later replay.

Using this approach, it is not necessary to trace every access to the CPU. CPUs do a sort of memory synchronization on their own using a cache coherence protocol. Hooking into the cache coherence is sufficient to track each CPUs view of the main memory. [11, 18]

Modern processors use a snoop cache coherence policy. The caches snoop on the memory address bus to detect cache conflicts.

In *Architecting a Chunk-based Memory Race Recorder in Modern CMPs* [18], hardware modifications that allow the recording of interleaved memory sequences are proposed. For this, the cache coherence protocol is extended. A logger for relevant memory events that synchronize the memory between the CPUs is added, as shown in Figure 3.2.

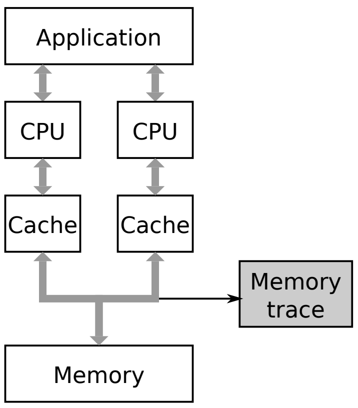

Figure 3.2: The communication between memory and CPU caches is monitored and the memory access order is recorded.

Events on the cache coherence bus do not contain any timestamp. Since there is no global synchronization on that bus as well, the order in which the messages are viewed by each processor cannot be determined by simply recording the messages.

To solve this issue, a logical clock is introduced for each processor. It tracks the oder of the individual events. By appending the logical clock to the coherence messages, the processors can record at which time the other processor sent the event and at which time they received it.

The instructions that cause memory accesses are then grouped into chunks between the cache coherence messages. Each instruction in the chunk has the same view of the main memory except for the changes the local processor does. Chunk termination is synchronized using the logical clocks which allows for a replay in the correct order. The changes a processor does to main memory then become visible to other processors after chunk termination, since this is the time the processors first may have synchronized their caches.

In *Rerun* [11], such an approach was implemented. The single chunks are named episodes in Rerun. An episode lasts at most as long as the core does not access any memory that is referenced by the current episodes of other cores. To detect such conflicts, the the cache coherence bus is monitored. Similar to Intel, the local clock of each processor has been added to the coherence protocol to be able to record the order in which such events happened. This may detect false positives if two bytes on the same cache line are updated, but those false positives do not influence the validity of the result.

If such a reference happens, the core ends it's current episode by writing the current time to the log. During replay, the episodes are replayed sequentially in the same order in which they have ended. This ensures a consistent view of the memory for all processors during replay. There is no ability for a parallel replay in this method since no chunk dependency information is stored.

In *Karma* [1], the concept or Rerun was enhanced by allowing a parallel replay. For this, the predecessors and the successors of each episode need to be recorded. They can be extracted from the cache coherence protocol messages that lead to a episode end.

During relay, this dependency graph is analyzed. Instead of ordering the episodes by a global time frame and running them in sequence, the episodes of each recorded processor are now run on their own processor. Episodes that do not conflict can then be run in parallel. For conflicts, processors need to pause until all episodes that their next episode depends on have ended. For this system, a recording overhead on of 1% for typical applications and 10% for memory conflict intensive benchmarks has been shown. The replay speed cold be increased to be between 19% and 28% slower than the original recording on the same hardware.

Although this approach causes a low overhead during recording, the recording is relatively complex. In *DeLorean* [14], a new coherence protocol that focuses on recording is proposed. Implicit memory transactions are suggested to make the deterministic recording easier. Within each chunks, the instructions are considered to be atomic as seen from other processors. If there is a conflict, chunks may be rolled back and the processor is reset to it's previous view of the memory.

To allow for detecting conflicts and rolling back, changes to the processor hardware and the memory bus are required. Therefore this approach requires specialized hardware.

Some processor instructions, like accesses to machine specific registers, are hard to roll back. To avoid this problem, the current chunk is committed before each of those instructions and a new chunk is started afterwards. This ensures that they do not need to be rolled back.

For a correct replay, the commit order of the chunks needs to be stored. The chunks can then later be replayed in the same order in which they were committed to ensure a correct view of the main memory. During replay, a commit protocol needs to be used as well. In addition to this, *DeLorean* uses two separate logs per processor to store non-deterministic events. One log is used for asynchronous events like interrupt requests. For those events, the system needs to be paused at a specific time during replay. The other log is used to store the results of synchronous operations like I/O operations. Those events are triggered by the guest during replay at the exact same state as they were recorded.

The granularity in which memory accesses were logged are cache lines. This makes two memory accesses conflict not only if they affect the same memory cell but also if they are on the same cache line. This greater granularity is required because the chunk coherence protocol only works on cache lines.

3.3.3 Virtualized Memory Tracing

With the emerge of virtualization technologies, an additional layer is added between the hardware and the operating system. It allows to run an unmodified operating system in a virtualized environment. This allows the memory tracing mechanisms to be moved to the hypervisor layer (Figure 3.3). Since the hypervisor is implemented in software, no hardware modifications are neccessary.

In *Samsara* [19], such a recording system was implemented. It uses a memory chunk protocol that is similar to *DeLorean*. In contrast to *DeLorean*, the recording is able to run on an state of the art system and does not require hardware modifications.

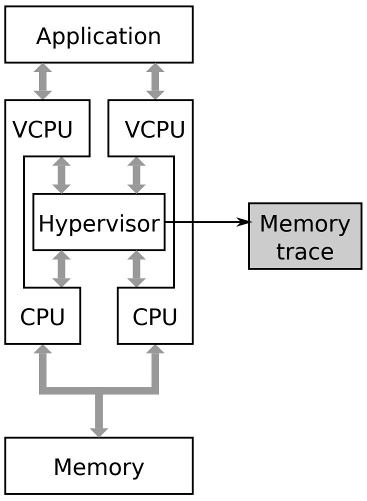

Figure 3.3: The hypervisor instructs the CPU to provide it with tracing information. It then writes the traces to the log.

Modern processors with virtualization support use a separate extended page table to map the memory of the guest to the physical memory. Normally, hypervisors use one table per virtual machine for this mapping. *Samsara* instead uses one table for each processor. This allows the tracking of the accessed and dirty bits of each individual processor to detect reads and writes to memory pages.

To simulate transactional memory, copy-on-write is used. On each write to memory, that write operation is only performed to a copy of the actual memory page. This page is then written back to main memory during the commit operation. Since the extended page tables are per processor, this allows a different view of the main memory for each processor so that each processor sees it's own writes but not those of other uncommited chunks.

A chunk is ended when the processor switches from guest to hypervisor. Page fault switches are ignored, since they occur frequently due to the copy-on-write required for writing memory. Other switches trigger a commit attempt of the current chunk. If the chunk is committed, the set of accessed pages is determined. It is then compared to the dirty map for the current processor. That map contains a list of pages that were written to by other processors. If there is any intersection because any of the pages the current processor accessed were written to by an other processor, the commit is aborted and the processor is rolled back to the state at which the chunk started. If there is no conflict, the chunk is committed by writing all pages that were modified back to the main memory. Then, the written pages are added to the dirty map of all other processors so that they get a conflict if they attempt to commit a conflicting access.

Samsara puts a lot of effort in optimizing the recording and reducing the recording overhead. This includes an optimized copy-on-write strategy that re-uses copies from the previous chunk, double buffering of the conflict page set for reduced locking conflicts and a commit protocol that reduces the time a global lock needs to be held to a minimum. With those optimizations, *Samsara* adds an overhead between 310% and 510% for a four core simulation. Although this is a significant overhead, it is relatively low compared to other software based solutions [17].

During replay, the same mechanism can be used to simulate a transactional memory. To ensure a correct ordering, each chunk may only start if all previous chunks have been committed. For this, the index of the chunk that was committed last before the current chunk needed to be recorded. For page writes, copy-on-write needs to be used as well. The changes may only be written back to main memory on the current chunk commit. This commit may only happen after all previous commits of the other processores happened to ensure a correct replay order.

While *Samsara* puts a lot of focus on efficient recoding, the replay was not implemented in their prototype and not evaluated in their paper.

3.4 Conclusion

Various ways of recording a system and replaying it later have been evaluated in the past. The ability to record a virtual machine using normal consumer grade hardware has only been evaluated lately. While the recording on this platform was evaluated to work, it was not yet evaluated if such a recording can be analyzed using an emulator to get a more detailed view on the recorded program.

For recording, a chunk based recording approach has been shown to have a low overhead in general. In *Samsara*, such an approach was implemented for recording a virtual machine. The ability to replay of such a multi-core recording using an emulator needs yet to shown. This will be the primary goal of this work.

Chapter 4

Design

In this work, a record and replay system will be implemented that supports replay in a different execution environment than the recording. The focus of this work is on supporting multiple CPUs that run in parallel in this heterogeneous environment.

4.1 Recording

During the recording phase, non-deterministic events are captured for later replay.

4.1.1 Initialisation

When starting a recording session, the recoding infrastructure needs to be set up before executing any guest code. A storage needs to be created and made accessible by the host kernel. This can be done before starting the virtual machine.

The recording state is only bound to the virtual machine. This allows the recording of multiple virtual machines simultaneously. Therefore, the required data structures need to be added initialized for the virtual machine. For each processor, a event log storage that records the processor events needs to be created.

The initial state at which the system starts is usually the system reset state. In this state, the processors are in a predetermined state. This state is set up by the hypervisor and can be set up fully deterministically. Therefore, it is not required to record the initial system state.

The virtual machine is then started. The loading of the BIOS, the boot vector, and other IO events are then recorded so that the system boot can be replayed exactly.

4.1.2 Processor Events

During the execution of the virtual machine, the processor may generate events that cannot be emulated during replay. Those events need to be detected and recorded to an event log for later replay.

Synchronous Processor Events

Many processor instructions are deterministic. This includes memory accesses, disk access and instructions that do register computations. The result of those instructions can be fully emulated if the current state of the CPU, RAM and disk are known. This is why those instructions do not need to be recorded.

The output of the remaining instructions cannot be fully determined by the emulator. This includes reads and writes to IO memory and machine specific registers. Since they are synchronous to the program flow, they will happen at the exact same time during replay. Their input parameters are the same as well, since they depend on the current CPU state which is fully known to the emulator. So only the registers that were changed as a result of the instruction need to be changed. A synchronous processor event containing those results is added to the event log for those instructions.

For verification, it can be useful to record the input parameters and the processor state when running the instruction as well. That way, divergences between the recorded program and the replayed one can be detected early without adding much overhead.

The processor events are then scheduled to be written to the event for synchronous log. There is one log for each virtual processor. This avoids synchronization problems.

Asynchronous Processor Events

Interrupts are asynchronous events that are passed on to the guest system. Whenever those interrupts are generated, the guest application is paused by the hardware and the hypervisor has to forward the interrupt to the guest. The time at which this interrupt happens is independent from the current processor state. This means that those events won't be happening automatically during replay.

To be able to replay those events, they need to be recorded in a way that allows inserting them at the exact same instruction during replay. To ensure this, the CPU is instructed to count the number of instructions it executes in guest mode. This count is then used to store the time at which an interrupt happened.

In addition to the instruction count, a part of the guest processor state is stored as well. This allows a validation of the replay processor state on interrupts. Due to limitations of the hardware, the instruction count may be off by a few cycles. [9] During replay, the processor state can be used to re-align the instruction counts of the replay and the recording.

The event is then scheduled to be written to the asynchronous event log. The event log for asynchronous events is separated from the one for synchronous events. This makes handling the different cases during replay easier.

4.1.3 Chunk Based Commit Protocol

On single-core systems where the CPU is the only entity that accesses the main memory, memory accesses are fully deterministic. A multi-core record and replay requires to record the interleaving of the memory accesses. This is done using a commit and rollback procedure that ensures that concurrent memory accesses can be tracked.

Defining a Chunk

A chunk is defined as a sequence of instructions whose view of the main memory is not influenced by other processors. Form the view of the instructions inside a chunk, they have exclusive access to the memory. From a outside view, all changes a chunk makes to main memory happen atomically.

The memory state at the beginning of a chunk is the memory state the instructions operate on. The instructions can then modify the memory and following instructions in the same chunk see the modifications of the current CPU only. Written memory is not made available to other processors but only stored locally. Those dirty pages are visible globally after a chunk has ended successfully. Such a successful end of a chunk is called commit.

Implementing this in practice introduces difficulties, since there is no hardware mechanism that ensures this atomic protocol. A roll back approach has been chosen to avoid this issue: Each chunk that cannot commit using those invariants is rolled back instead. This may happen on conflicting writes. If two concurrent chunks write to the same memory cell, their real write order is unknown. This is why those chunks need to be rolled back.

For avoiding a full memory snapshot when starting a chunk, the chunk reads are always done on the current main memory. This introduces the problem that the chunk may read a memory area that was altered by an other processor after the chunk started. This violates the assumption that a chunk always has an uninfluenced view of the main memory. For this reason, chunks such accesses are traced and chunks with such accesses are rolled back.

To avoid overhead, a page wise approach is used for tracking memory accesses. This has the additional advantage of being able to track page reads and writes using the accessed bits of the page tables.

Processor Isolation

Whether a chunk is committed or rolled back is determined at the end of each chunk. Therefore, other processors may not see any effects of the other processors until after they have committed.

To ensure this, all changes the to the main memory during a chunk is not written directly to memory but to a temporary memory area. To implement this efficiently, copy-on-write is used.

All processors share a common memory that represents all guest memory contents that have been committed. Each processor has it's own page table that maps in all this guest memory as read only. As soon as the guest writes to a memory page, that memory page is duplicated for the guest and the writes are written to the duplicate. The guest then does further reads and writes to that local page. That way, changes to memory contents are not visible to other processors.

Handling machine specific registers and accesses to IO memory cannot be delayed using this mechanism. The processor requires the hardware to do those accesses immediately. To make this direct hardware interaction possible, it needs to be done in a state in which a rollback may not happen. For this, the current chunk is committed before such an access is done. The access is then executed and the next chunk starts with the result of the access. That way, the access only needs to be done once on the hardware. If the next chunk is rolled back later, the results of the access can be re-used. For this, it is stored in the processor local data structure that restores the processor state on rollbacks.

Tracking Chunk Memory Accesses

In order to determine whether a chunk can be committed, the pages that were accessed in the chunk need to be tracked.

This is done by using the accessed flag of the extended page table. This flag is set by the processor MMU on each EPT lookup at the corresponding page table entry and all parent entries.

At the beginning of each chunk, the flag is reset. That way, the flag was set by the hardware at exactly the memory pages that were accessed during the chunk once the chunk has ended.

The the flags are read at the end of each chunk. For this, the EPT is walked hierarchically. Since the accessed flags are set on all page table levels, a full scan of all page tables is not required. The scan can be done efficiently in linear time relative to the number of accessed pages.

For memory writes, copy-on-write needs to be used. This requires all pages to be set to read only. That way, the hypervisor is trapped on the first write access to each page. The hypervisor then copies that page to a new memory page and alters the EPT entry to point to the new page. It sets the accessed flag and sets a flag in the dirty page set to indicate that this page was written to in the current chunk.

For storing the set of accessed and dirty pages, a data structure that holds a set of pages and allows for a fast iteration of the page numbers and a fast check if it contains a single page number needs to be used. *Samsara* uses a bitmap in combination with a linked list. Both data structures always contain the same information. This redundancy allows the fast lookup times of a bitmap to be combined with the linear walk time of a linked list.

The disadvantage of this approach is that a bitmap takes as much space as the highest possible virtual address is. For virtual machines with a contiguous memory, this is not a big problem since the maps are still comparably small. But as the virtual memory gets bigger, the size of the bit maps increases. Modern systems may not even require a contiguous virtual memory. On those systems, a memory bitmap mapping the full 48 bit address space that a modern Intel EPT could address requires 2^{48} bits (35 terrabytes) of memory.

To reduce this amount, that information can directly be stored in the page tables. One of the reserved bits of the EPT can be used for this. The dirty flags is set in the EPT of the current processor when a copy-on-write operation occurs. The insert operation for this is no performance overhead, since the EPT entry needs to be written to to update the page address any way.

An efficient walk of the dirty pages is only possible if all parent pages have the dirty bit set correctly. The dirty bit cannot be set on the first page table walk that determines the page address on a page fault, since it is unknown then if the page should be copied. Therefore, the dirty bit is only set in the last level of the page tables. For each page that was marked as dirty, the accessed flag is set as well. Since the accessed flag is set in the complete page table hirarchy, the accessed pages can be iterated efficiently. During the commit phase, such a walk of the accessed pages is required. During that walk, the dirty pages can be tracked as well with low overhead.

Detecting Chunk Conflicts

At the end of each chunk, a decision needs to be made on whether to commit the chunk or to roll it back. For this, a conflict page set is used for each processor that tracks the pages on which it might conflict with other processors.

For this, the commit conflict page sets are first locked globally. This ensures that no other processors influence the computation. It also ensures that no other commits happen while the current chunk is committed.

Then, the read and write actions of the chunk are compared to other chunks that were active while it was running. The chunk needs to be rolled back if any chunk it has accessed was written to by an other processor during time the chunk was active.

For this, each processor has a processor local map of pages that were modified by other processors during the chunk. Other processors did modify this map when they did commits, so this information does not need to be derived.

The processor then computes a list of pages it has accessed. It is then tested if that list intersects with the conflict page set. If they do intersect, a conflict happened and the chunk needs to be rolled back.

Chunk Commits

After determining that a chunk needs to be committed, the data required to replay it is written to the replay log and it's memory modifications are made available to other processors.

First, the commit index is determined. For this, a global counter is used that is incremented atomically. This commit index is used for the chunk end event so that the chunks can be ordered correctly during replay.

Then, the list of the copy-on-write pages of the current chunk is determined. Those pages can be found by either walking the extended page table or by tracking them in a separate copy-on-write list. Those pages are written back. For each of thoise pages, the corresponding page is added to the conflict page set of all other processors.

During the chunk, all events were written to the temporary event log. On a commit, this temporary event log is written to the persistent event log. In addition to the processor events, a chunk start and a chunk end event is added. Those events are added to the asynchronous log, since they have not been triggered by the program flow. In Figure 4.1, this mechanism and the temporary event logs are visualized.

The chunk start event contains the index of the last commit that happened before the chunk was started. Since a chunk's view of the main memory is determined by the time it started, this information is required during later replay to determine the time at which the chunk may start running. Due to the delayed addition of the events to permanent storage, the current commit counter can be added to the start of the chunk as well. That way, the replay system can determine the commit time at which a chunk ends easily. This makes ordering the chunks for later replay on single-core processors easier.

At the end of the chunk, a chunk end event is added. It contains the commit index of the chunk.

It would also be possible to store the list of accessed and written pages of

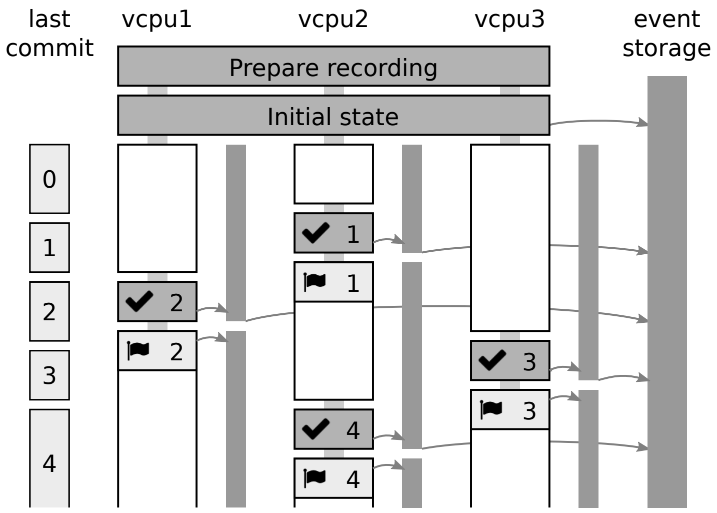

Figure 4.1: The execution of the guest CPUs is periodically paused. The chunk is then committed by storing it's events to the event log. The chunk end event contains an index that indicates the order in which the commits happend.

the current chunk. This might allow for further performance optimizations during replay. Since replay performance is not the focus and a fast recording with low memory overhead is desired, this is not done. A correct replay is possible without this information.

Chunk Rollbacks

If a conflict was detected during the commit phase of a chunk, the commit is aborted. The chunk is rolled back and the processor state is restored so as if no instructions were executed since the last chunk.

All changes made to the memory are discarded by removing the copy-on-write pages and restoring them with the current memory contents of those pages.

The processor state is then restored to the state that was stored at the chunk start. This state includes the registers of the processor with the instruction pointer. That way, the program resumes it's execution at the point where the previous chunk ended.

There might have been interrupts on the current processor for the chunk that is to be rolled back. Since the chunk is rolled back, the guest needs to see a state in which those interrupts have not been handled. For this to happen, the interrupt flags are set again. This causes the processor to execute the interrupt handler for

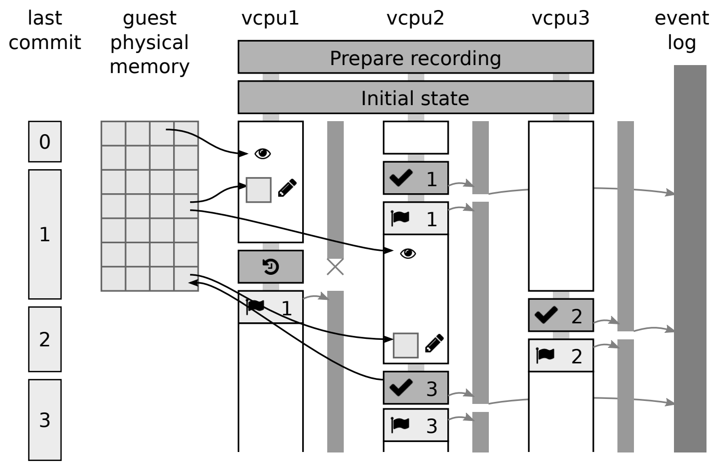

Figure 4.2: On memory writes, a local copy of the memory page is created. That copy is committed at the end of each chunk. If it was accessed by other processors in between, the chunk is rolled back and the changes are discaded. Chunk starts and commits are stored to the event log.

the flag with the highest priority as soon as the processor starts executing the next chunk. From a guest perspective, this causes the impression that the processor was only paused by the hypervisor and the interrupt occurred while the processor was paused. The guest will then execute the interrupt handler as if that interrupt handler was never executed.

Forced Commits

Commits are normally done on a regular basis every several milliseconds. This ensures that changes a chunk made become visible to other processors after a few milliseconds. For this, a timer is used that traps the vCPU into the hypervisor after a given time. The current chunk is then committed.

But there are situations that require an instant commit of the current chunk. One such situation is the interaction with hardware. When the guest system accesses hardware, the hardware is not accessed directly. Instead, the hypervisor is trapped to emulate the hardware access. The hypervisor normally forwards those calls to the real hardware. This may be a network device or a hard drive. For most devices, this forwarding means that the hardware state changes in a way that cannot easily be undone.

State from one processors should only be visible to other processors after a chunk commit. Since the hardware device is shared between processors, it would also mean that the commit protocol would need to be extended to hardware registers. The hardware would need to have the ability to be visible to other processors in an older state and conflicts on the hardware would need to be detected.

When rolling back such hardware operations, all state needs to be restored. This includes communication with external devices. For the network interface, packages could be delayed. But other interfaces like the serial port might not be able to delay the messages if the timing is critical or if the processor awaits a direct response.

Therefore, the commit and roll back protocol cannot be extended to generic hardware devices. To avoid this issue, whenever the hypervisor traps for an IO operation, a commit is forced. The processor then either commits or rolls back the current chunk. If the chunk was rolled back, the processor is restored to the old state and the IO operation request is ignored by the hypervirsor. The processor resumes the virtual machine execution at an earlier point. The guest chan then attempt the IO operation again in hope that no conflict occurs.

If the commit was successful, the IO operation is performed. The hypervisor then sets the guest registers to to correspond with the new hardware state and loads the return value from the hardware into the desired register. After that, the snapshot for later roll back is done. This ensures that a roll back always rolls back to directly after the IO operation. The IO operation itself is not rolled back.

Since the guest was not resumed, no memory accesses may have been done in the time between the commit and the rollback checkpoint. That way, this short time does not need to be protected by the memory commit protocol.

4.1.4 Direct Memory Access

When other hardware has direct access to the main memory, the same problems as with multiple processors arise. Therefore, that hardware needs to participate in the commit protocol as well.

This may even happen in systems with only one CPU that support direct memory access (DMA). DMA allows the hardware to directly access the main memory. In virtualized environments, the guest operating system has no direct control over the DMA controller. Instead, all accesses are trapped by the hypervisor. It then checks the access permissions and forwards the request to the controller. That controller can then both read from and write to main memory.

The memory that is written by the DMA controller on a write operation can be seen as deterministic as long as the state of the source hardware is known or recorded. But while the read operation is in progress, the destination memory area is written to by the DMA controller at an unknown time. Therefore all read of the running CPU to the affected memory area can be considered non-deterministic.

For operations where the DMA controller reads from memory, a similar problem may arise. While the DMA controller reads the memory, the processor may modify that memory area. Therefore the values read by the controller are non-deterministic.

Guest applications normally require a deterministic behavior of the DMA controller. This is why they do not access the memory that was passed on to the DMA controller while a DMA operation is still in progress. But this behavior cannot be relied on to ensure a deterministic replay. It can be enforced by the hypervisor by locking the affected pages for read/write access while the DMA operation is in progress. That way, the guest application will need to be halted if it attempts to access one of those pages.

An alternative is to buffer the DMA controller memory by using copy-on-write while the controller reads from memory. For DMA writes, the data is written to background pages and the pages are then made visible to the guest by re-mapping them. This has the advantage of making the DMA operations atomic and allowing a simpler replay implementation at the cost of reducing the DMA performance and increasing memory requirements while recording.

4.1.5 Checkpointing

An alternative to starting the recording at the system start is to allow for it to be started at a user defined time. This has the advantage that the log size is smaller.

To create a checkpoint and start a recording session from there, the virtual machine needs to be paused. A memory snapshot needs to be taken. A hypervisor supports two modes to create a snapshot. In the stop-and-copy mode, the virtual machine is paused, the snapshot is written to disk and the machine is then resumed. In the copy-on-write mode, the virtual machine is only paused a short time to save the processor state. The memory is then to set read only and uses copy-on-write if the virtual machine attempts to write to a memory page. That way, the virtual machine can continue to run while the snapshot is written to disk. Once the snapshot is completely written, copy-on-write can be disabled again. Since copy-on-write is used during the chunks, a stop-and-copy checkpoint can to be done to not interfere with that protocol.

A copy-on-write snapshot is not possible the way copy-on-write snapshots are usually implemented. Current implementations assume that there is only one EPT per virtual machine. If a page write occurs during the snapshot, that EPT can be changed to point to a new page location and preserve the original page contents. This approach does not scale well since there is now one EPT per virtual processor. To avoid this issue, the write back on chunk commits can be intercepted. Before a page is written back, it is first checked if it needs to be preserved for an ongoing checkpoint write. If it needs to be preserved, the original page content is copied to a different memory location before the new page content is written. The copied page is then used for the snapshot.

After taking the snapshot, the system initializes the additional data structures for recording. The extended page table is set up for the copy-on-write mechanism. Then, all processors add the initial event to their recording log. That event contains information about the initial processor state. This may include additional information about the current CPU features to match this processor automatically in later replay.

When allowing to start the recording while a virtual machine is running, it may be desired to stop the recording after some time. To stop recording, all processors need to be paused so that none of them writes to memory any more. The hypervisor then does a final commit or roll back operation for each processor. Since the processors are paused, all processors have the same view of the main memory after this is done. The hypervisor then closes the event log and removes all event hooks. The EPT is restored to remove the copy-on-write flags.

After all processors are done with this, the virtual machine can be resumed. The processors then continue to run the virtual machine in the normal, non-recording execution mode from the point where the recording ended. A recording can be started again at any time. This makes the recording system flexible.

4.2 Replay

The replay is done in an emulator that is able to exactly emulate the source system. The emulator is started at the same state the recorded system was when recording started. It then replays the exact instruction flow of that system. To replay non-deterministic instructions and the correct order of memory accesses, the emulator uses the recorded event log.

4.2.1 Initial state

The initial state of the replay needs to be exactly the same as the initial state of the source system.

If the recording was started with the virtual machine construction, the emulator needs to create the exact same system state. It needs to make the same number of CPUs available. Each CPU needs to have the correct virtual CPU ID set and the CPU features need to be matched to those of the source system.

The memory state is set to the initial, reset memory state.

No further setup is required since the non-deterministic actions that may occur during the system boot are all handled by the recording mechanism. This also includes the start of the non-primary processors that is triggered by the guest operating system in the early boot process.

4.2.2 Chunk Ordering

During recording, the vCPUs were running in parallel. If memory race conditions between the processors occurred during recording, the order in which the memory was accessed was recorded by recording the order in which the processors committed their memory. When replaying, the memory accesses need to be replayed with an order that ensures that the result of the memory accesses is the same.

For correctness, an exact ordering of all memory accesses is not required. It is only required that all memory accesses to a fixed memory address happen in the exact same order as they happened during recording.

During recording, the commit protocol ensured that no two threads accessed the same memory at the same time except for reading. Moreover, it is even guaranteed that if a processor writes to a memory page, no other processor writes to or reads from that page during the chunk. If this would have happened, one of the two processors would have been rolled back.

This means that during replay, it needs to be ensured that all chunks that write to a fixed memory address x are replayed in the correct order. If two there are two chunks $c_{1,2}$ accessing the memory page of x and c_1 was committed before c_2 started, the writes of c_1 need to be replayed before the replay of c_2 can be started.

Since there can only be one active chunk writing to that page, this ordering is ensured if a chunk only starts to be emulated after all chunks that were executed prior to it during recording have finished.

An emulator can emulate parallel processors by running them in sequence and switching between them. This ensures that there are no direct race conditions between the processors.

For such a single-core replay, those constraints can be met if all chunks are executed in the order in which they committed. If the commit order was stored along with the chunk start event during recording, the processors are always yielded whenever a chunk ends. The event logs of all processors then contain chunk start events at their head. Since those chunk start events contain the commit index, one of them needs to contain the next commit index. The corresponding processor is then scheduled for execution.

That way, the replay can be done without any need for further, complex conflict logic.

4.3 Conclusion

A multi-core virtual machine can be recorded using the virtualization hardware extensions. During the recording, a transactional memory is simulated by splitting the program execution in chunks of instructions. The changes each chunk does to the main memory are then committed atomically. Conflicts are detected and rollback mechanism is used in this case to drop the conflicting chunk.

This recording can then be replayed in an emulator down to the exact instruction level. The emulator needs to be able to exactly emulate the behavior of the source system. For the multi-core support, only minor additions to a single-core emulator are required. The chunks of the individual processors can then be scheduled in their commit order on that single-core emulator.

Chapter 5

Implementation

To evaluate the possibilities of heterogenous, multi-core record and replay systems, a prototype is implemented. It supports recording a virtual machine with multiple processors and then replaying it using an emulator. To evaluate the features of this prototype, a special guest system is implemented that triggers memory conflicts and makes evaluating race conditions possible.

5.1 Architecture

The implementation is based on QEMU using the KVM kernel module for a virtualized recording and using QEMU in TCG mode for the emulated replay.

QEMU has a record and replay feature that supports recording and replaying in the TCG mode only. This feature only supports a single vCPU. Therefore, a new recording mechanism needs to be created that supports recording in KVM and a replay mechanism that

5.2 Chunk Protocol Data Structures

To implement the chunk based commit protocol, additional data structures need to be added to KVM. Those data structures are added to the virtual machine and to the virtual processors. No host global data structures are used to allow for multiple virtual machines to be recorded in parallel.

5.2.1 Page Set

The implementation needs to handle a set of accessed, dirty and conflicting pages for each processor.

For this, a generic interface is written. Each page is identified by it's global frame number. A library implementing the interface needs to support insertions of single pages, a union of one page set into an other and intersection tests of two page sets.

Since the page set data structures are used on each commit, the operations should be as performant as possible. For this, a single page insertion time in the order of $O(1)$ is desired to allow for a linear run time scaling with the number of pages written. The union operation and the intersection test should be possible in $O(n)$ where n denotes the number of pages in the page sets. The size of each page set should be linear to the number of pages that are contained in the set. It should not relate to the total size of the virtual address space of the guest, since full 48 bit address spaces should be supported.

In addition to those operations, a fast mechanism that clears the whole page set is required. This mechanism is used when the chunk is committed and the page sets need to be reset.

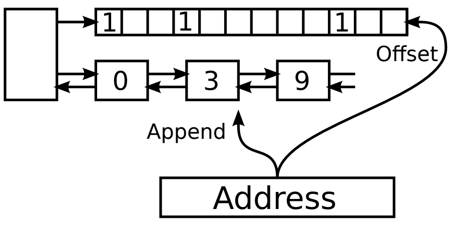

Figure 5.1: *Samsara* uses a bitset and a linked list to store the conflict map. Both data structures contain the same information.

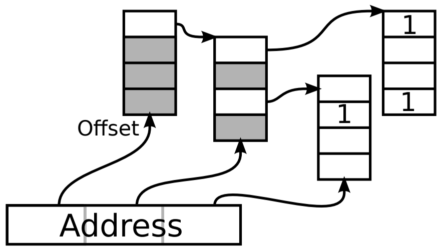

Figure 5.2: A hirarchical bitmap solves the size problem of large address spaces and removes unneeded redundancy.

In *Samsara* [19], a data structure that provides those operations was implemented. The data structure consists of a linked list that contains the indexes of the pages that are in the set and a bit map in which the pages contained are represented as set bits. This data structure is shown in Figure 5.1. Both parts of the data structure store the same information and are updated simultaneously. The linked list can then be used to walk though the list of pages efficiently while the bitmap allows for an efficient contains query.

On each addition of a page to the set, the corresponding bit in the bit map is set. If that bit was not set before, the page index is added to the list. That way, an add operation is possible in $O(1)$ while maintaining the invariant that the page may only be added to the linked list once.

To add all pages from one set to an other, the list of pages in the set that needs to be added is traversed. They are then added to the other page set sequentially.

This results in a total time for a union of $O(n)$.

The intersection of two pages is computed by walking the linked list of one page set. Each of the pages present in that list are then tested in the bit map of the other page set. Since that test ist possible in $O(1)$, the total page walk time is $O(n)$

In *Samsara*, only a contiguous address range is used as guest physical memory. That way, the bit map that spans all of the guest physical memory is reasonably small. While this is sufficient for virtualizing a normal operating system, virtualized systems can be more flexible and do not require the guest phyiscal memory to be in one chunk [2]. In those situations, the bitmap would need to span the full address space of the guest, which results in a bit map size of $8GiB$ on the 64 bit system with 48 bit addresses that was used for the implementation. Although the test system has a memory capacity of $64GiB$, the space would not be sufficient since *Samsara* uses four maps per processor. Therefore, the size of the page set data structure needs to be reduced to allow for a recording on a consumer grade host system.

The total number of pages in a 48 bit address space is approximately 69 million. Since most tables are expected to have an entry count in the order of thousands, this leads to a huge amount of wasted memory space. To reduce the size of the data structure, the bitmap can be converted into a sparse tree structure that resembles a page map with a bit set for it's last level, as shown in Figure 5.2. The granularity can be tuned by the number of levels the data structure has. For easy convertibility between the page set and a page table, the same structure as a page table is used. A 4kiB table with 64 bit pointers can hold a total of 512 pointers. 9 bits of the page address can be used as index in this table to access the next layer, as it is done in the page table. But instead of a page table entry, the last level only stores a bit set for the pages. This bit set has a size of 512 bit (64 bytes).

A read or write access to a existing page entry in this page set can be done using a total of 4 memory accesses. New entries can be added using at most 3 memory allocations. Therefore, setting a bit in the table can be done in $O(1)$. For pages that are in contiguous areas of the memory, this number is smaller.

A further optimization could be to reduce the number of levels in the table by one to reduce the walk time and while increasing the memory that is occupied by a table with few entries.

Walking such a structure can be done efficiently in $O(n)$ since the total walk time is limited by the number of tables in the structure, which is itself limited by the number of previous insertions times four. A union can be computed efficiently by walking one table and inserting the entries into the other table. A intersection test is possible by walking one table and testing against the other as well. It can however be done more efficiently by testing the intersection in each level. For the leaf bitmap, a bitwise comparison of the 64 bytes can be used. This is more efficient than walking the table for each bit of those 64 bytes. If an entry is not set in a top level table of either page set, there can not be any intersection in the leaf nodes of that part of the tree. That way, leaf nodes can be skipped entirely if their parents are not in the other set.

5.2.2 Memory Access Maps

The memory accesses, writes and conflicts of each processor need to be stored during each chunk.

For the access/dirty bits, the corresponding bits of the EPT the hardware defines are used. On Intel, this is the bit 8 for the access and bit 9 for the dirty flags.

KVM uses a separate EPT for each virtual processor. Normally, tables are shared between those EPT structures. To prevent this and to have a accessed flag per processor, KVM is instructed not to share tables between processors.

The access bits in the EPT can be set by hardware. This feature is activated in this implementation to track read accesses without overhead.

For write accesses, the dirty bit is not tracked by the hardware. Since all pages that are written need to be copied, that bit is set during the copy operation. The accessed bit is set in the same operation, that way a dirty page is always marked as accessed. Those copy-on-write pages are marked with an additional copy-on-write flag. For this, bit 52 is used. That bit is ignored by the hardware and free for custom use.

To store the conflicts with other processors, a page set is used. The set of pages with accessed or dirty bits can be seen as such a page set as well. But using this for operations like the intersection or union requires page table walks that are not already implemented in KVM. To allow or a cleaner application implementation, those sets are extracted to a separarte page set first. For this, the page table is scanned. Since the accessed flag is hierarchical, unaccessed leaf tables do not need to be scanned. That way, scanning is possible in linear time compared to the number of accessed pages. Those pages are then added to the temporary accessed page set.

Each accessed page is added to the accessed page set. Since only accessed pages may be dirty, it is sufficient to check each accessed page if it was marked as dirty as well. Those pages are then added to the dirty page set.

That way, there are three page sets present at the end of each chunk. The accessed and dirty page sets contain the pages accessed or written by the current processor. The dirty page set contains the pages that other processors have committed to during the current chunk.

5.2.3 Event Log

The event log needs to support the addition of events that have a small but variable size. For each new event, a event object is allocated. The event meta data is written to that object. That meta data contains flags about the event nature, especially if it is an asynchronous event or not.

When the event is added to the event log, this flag is checked first. Then, the correct temporary synchronous or asynchronous event log for the given virtual processor is searched. That log is implemented as linked list. The event is appended to that list.

Upon commit, the all events from the temporary event log are copied over to a transfer event log. That log contains the events that should be moved to the permanent storage on disk. This copy operation can be implemented fast since the transfer event log is a linked list and moving items between linked lists does not require any memory allocations. In this case, since the whole temporary list is appended to the transfer log, the list does not even need to be walked and only the pointers to last/first items need to be changed.

QEMU regularly sends a request to KVM to get the latest event log contents. QEMU provides the kernel with a user-space memory are to write the new events to for this. The kernel removes the events from the transfer queue after this. Those events are then written on to the *Simustore* afterwards. That transfer mechanism out of the kernel is based on the implementation in the *Simutrace* [20] project.

In addition to the events that are recorded for the replay, events that help restoring the system state on rollback needs to be stored. For this, a list of APIC interrupts that were triggered during the current chunk is used for each processor. This list is cleared on each successful commit, since the events for replay are recorded separately.

5.3 Recording

The recording is based on qsimu-linux from the *Simutrace* project [20]. It is adjusted to support recording multiple processors. The chunk based commit protocol is added while parts from *Samsara* [19] could be re-used, although differences in the KVM versions required a rewrite of some parts.

5.3.1 Creating an Initial Checkpoint

The implementation assumes that all processors are initialized upon recording start. On system boot, only the first processor is started by the hardware. Therefore, the recording can only be started once the guest operating system has initialized all cores. This is one of the first things modern operating systems do, so a large part of the operating system startup and all of the running operation system can be analyzed in this implementation.

The reason for starting the recording with all cores active is that the protocol to start processors is complex and not well-documented by Intel. An emulator implementation of this start mechanism that behaves exactly like the Intel implementation for a given processor family is difficult to achieve. Since there were differences in the way QEMU and the real hardware behave, it was decided to skip this initialization and to focus the implementation on a running operating system instead. The advantage of this is that a multi-core checkpointing mechanism can be evaluated together with the chunk based commit protocol.

A command is added to QEMU that starts the recording. This start command first checks whether the virtual machine is paused. If it is not, it pauses all processors and waits for them to halt.

Then, a checkpoint is done using *Simuboost*. The checkpoint is done synchronously without use of the copy-on-write mechanism to not interfere with the later commit protocol. Therefore, the start command needs to wait until the checkpoint is done.

After creating the checkpoint, a recording session is started. The virtual machine can then be resumed afterwards if it was paused by the command.

5.3.2 Starting a Recording Session

A recording session can only be started directly after a snapshot. The processors are still paused when starting the recording so that the system state is exactly the same as the one in the snapshot.

The recording data structures are then initialized and cleared. This initialization is split into two parts. First, the virtual machine global data structures are initialized. This includes the global locks used to synchronize the commits.

After that, each virtual processor initializes it's local data structures. Each processor has four event queues. They are used to store the temporary and the committed events, each synchronous and asynchronous. They are simple linked list and can be initialized on default.

Then, the data structures required to record the copy-on-write pages are initialized. They are then set to the empty, initial values for each chunk.

While recording, KVM needs to detect write accesses to pages. For this, pages need to be set to read only. This flag is not updated directly on startup. Instead, the shadow page table is invalidated. This makes KVM re-create the shadow page table as soon as the virtual processor accesses memory. A flag is set internally to indicate that newly created EPT entries should be set to read only.

The bit 52, which holds the copy-on-write flag, is added to the list of allowed bits in the shadow map. Otherwise, the page table validation of KVM would not allow setting the bit. Since the validation is per processor, this needs to be done in the processor start routine as well.

The state flag of the memory chunk protocol is then set to idle to indicate that recording has just started. The virtual processor is now in the state in which it can be resumed and starts with the chunk based memory protocol.

The preemption timer is set up to preempt the running virtual machine after it executed for ea given time. This is used to restrict the chunk size. The chunk end handling is done on VM exit, therefore the premption timer callback does not need to be changed.

5.3.3 Processor Memory Access

As long as the virtual machine is recorded, each processor has it's own view of the main memory. Pages may be mapped differently on the different processors for this. While the processor is executing guest code, this is achieved by separating the EPT for each processor. But as soon as the guest traps to the hypervisor, the hypervisor may need to read guest memory, too. KVM internally uses the kvm_read_guest and kvm_write_guest functions to do such reads and writes. Those functions translate the guest address into the host physical address and then write to that address. For this, the functions receive the current virtual machine as parameter.

Those functions are altered so that they do receive the virtual processor instead of the virtual machine instead. That way, the memory access can be done using the view of the current processor. The current processor that triggered the read or write by entering the hypervisor needs to be tracked when those functions are invoked.

There are some global operations in KVM that are not run out of a vCPU context. They include copy-on-write remappings when doing a copy-on-write snapshot. The mechanism used by those snapshots now is to modify the corresponding EPT mapping if a guest writes to a page that is under copy-on-write protected. Using separate EPTs for each processor, a copy-on-write operation without a vCPU context would require all EPTs to be changed simultaneously. While this can be implemented, it would require a lot of changes to the way KVM handles those operations. Since copy-on-write snapshots are not required for the evaluation, they are disabled in this implementation.

5.3.4 Recording Events

After the recording session was initialized on all processors, the execution of the virtual machine is resumed. Counters are installed to count the number of instructions the processor executes in guest mode. This count is used to allow for a replay of asynchronous events.

All events that are non-deterministic need to be handled by KVM directly. Therefore, no direct hardware access is granted to the virtual machine.

Whenever the virtual machine does an action or receives an event that introduces non-deterministic behaviour, KVM needs to be hooked so that an event is added to the replay log. For this, the kernel is modified to record the relevant information for each of those events.

For interrupts, the number of the interrupt vector that was triggered and the current instruction counter is stored. Storing only the instruction count is not sufficient for locating the exact instruction at which the interrupt was triggered, since that instruction count can be off by a few instructions on Intel platforms [9]. To allow for an exact location, the current instruction pointer and the exc register are stored. The exc register is required for determining the iteration of a rep instruction. Adding more registers to this landmark increases the accuracy of locating the right time to insert the event during replay. In some cases this is not enough, since the processor can be looping in an infinite loop using a relative jump instruction with an offset of 0. In this case, the processor would execute the exact same instruction over and over and the other registers of the processor would not change. An exact insertion of the event may not be possible then. With the current hardware, this situation cannot be prevented. In this case, an earlier insertion of the event by a few cycles would probably not change the future instruction flow. Although the risk of such a situation occurring in practice is low, there is still the possibility of such a situation.

The interrupts do not only need to be recorded for the replay, they also need to be recorded for rolling back the current chunk. If the chunk is rolled back, the interrupt flags need to be set again so that the current processor handles the interrupt again after the roll back operation. For this, all interrupts of the current chunk are stored to a separate interrupt log that only stores the number of the interrupt that occurred and not the time at which it occurred.

The results of several instructions that are non-deterministic are recorded to the synchronous event log. The result of each accesses to machine specific registers and to the APIC controller is recorded. The content of writes does not need to be stored since the writes will occur in exactly the same way during replay. Only the values read back from those instructions are therefore added to the event log. This also includes the $cpuid$ instruction, the $rtdsc$ instruction, and the $rtdscp$ instruction. The result of those instructions then does not need to be emulated by the emulator completely. In case of the time stamp counter, computing the exact value might even be impossible for the emulator. This way, it is ensured that the emulator can exactly reproduce the instruction results that occurred during the recording.

After recording the events, they are added to one of the temporary event queues. This queue stores the events for the current memory chunk. They are passed on to the permanent storage once the chunk is committed.

5.3.5 Capturing Memory Access

On each access to memory though the page table, the MMU sets an accessed flag for the given page. Having a table for each processor allows to track the accesses of each processor.

Extended page tables allow huge pages with $2MiB$ or $1GiB$ instead of the default $4kiB$. KVM uses this optimization when a large, contiguous memory area is allocated to the virtual machine. If such a large page would be written to, a lot of data would need to be copied in order to perform the copy-on-write operation. To avoid this situation, KVM is changed to only use small pages by disabling the merge of such areas.

The memory writes need to be intercepted to do a copy-on-write operation. For this, each memory page is set to read only as long as it is mapped to the shared memory of the virtual machine. As soon as the guest attempts to write to any page, the hypervisor is trapped to handle that page fault. It then checks if the page could normally be written to. If this is the case, a processor local copy of the page is made. The page table entry is then changed to point to that local copy and to allow writes to that local copy. The virtual machine is then resumed. It performs the memory access again, while this time it succeeds and accesses the copied page. Further reads also read from the copied page and therefore read the changes the current processor did to the memory but not the ones other processors did.

Processor local data structures store the accessed, dirty and conflict state of each memory page, as shown in Figure 5.3. Upon commit, the accessed and dirty flags of the processor are collected into page sets. The accessed page set is then compared with the local dirty page set. If they intersect, there are conflicts that prevent the chunk from being committed. After the commit, the set of local dirty pages is added to the dirty sets of all other processors. The sets are then reset for the current processor after each chunk.

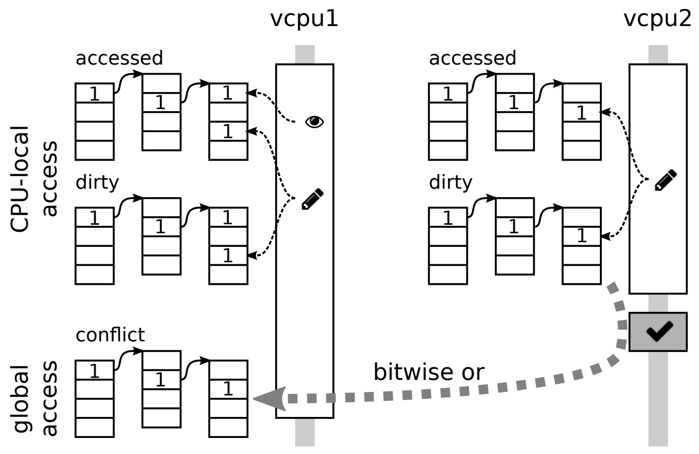

Figure 5.3: When a vCPU accesses a memory page, the corresponding flag in the accessed and dirty bitsets is set. On a commit, all dirty pages are committed to memory and visible to other vCPUS. The dirty pages are therefore marked as conflict on each other vCPU. vCPUs can then compare this conflict map with their local accessed map.

5.3.6 Ending a Chunk

When the processor traps into the hypervisor, the function *vcpu_enter_guest* is called. This happens for special instructions, page faults and a number of other reasons. In this function, KVM handles the exit from the guest code. Guest operation is resumed upon function exit. Therefore, the commit and rollback protocol needs to be hooked into this function.

At first, the action that should be taken for the current VM exit is determined. This can either be to roll back, to commit, or to do nothing. Roll back and commit both end the current chunk.

No action is taken on VM exits that occurred because of a EPT violation. Because of the copy-on-write mechanism used during recording, those EPT violations are quite frequent. They should not end the current chunk.

If an action should be taken, it is determined if there are conflicts that prevent a commit. As stated above, the memory accesses during the chunk are recorded by the hypervisor for this. An evaluation function is called that returns whether there is a conflict in the current chunk. A commit action is taken, if all the processor has no conflicts with other processors. Otherwise, the processor is rolled back.

Determining this action requires to access the dirty page set. That page set is guarded by a global commit lock which needs to be accquired for the test. After determining the action that is to be taken, it is executed while still holding the lock.

Commit

For a commit, all pages for which copy-on-write was used are stored in a list. In this list, the original and the copied page location are stored. Since a page was only copied on a write page fault, the hypervisor can be sure that all pages are modified and does not need to check for further modifications. The pages are copied to the original position and thus become visible to all other processors. The copies are then discarded and the original EPT entry is restored.

In addition to writing back the pages, the dirty sets of all other processors are updated. For this, the list of processors is walked, skipping the current processor. Since the list of copy-on-write pages is the same as the set of dirty pages, the set of dirty pages is unioned into the conflict page set of each processor. Operations on the dirty page set are guarded by the global commit lock, which is held in this phase. It can be relased after the pages are written back and all conflict page sets are updated.

The current processor state is then saved for possible future roll backs. This includes the register contents, the contents of machine specific registers, the FPU registers, and the MMU registers.

Roll Back

If a roll back occurs, the virtual machine state needs to be restored to the state at which the chunk started. For this, all copy-on-write pages are discarded and the original EPT entries restored for them. The accessed, dirty and conflict page sets are cleared.

Then, the processor registers are read from the values that were stored when the chunk was started. This value was stored on commit.

5.3.7 Stopping the Recording Session

To stop the recording and still continue the virtual machine execution, there may not be any copy-on-write pages left. This is why disabling the chunk based protocol is only possible if all chunks were committed or rolled back. For this, the virtual machine is paused by QEMU. QEMU then instructs KVM to set a global flag that the virtual machine should stop recording. Then, it traps all processors into the hypervisor. This triggers a commit of each of the chunks. Since the commits are guarded by the global commit lock, they are done in sequence. The commit or rollback operations then check the global exit flag. Since it is set, no preparations for starting the next chunk are done. Instead, the data structures are freed. To unregister the copy-on-write mechanism, KVM flushes the EPT so that it is re-created on the next page faults.

The events are then pulled by QEMU and written to the storage. The storage for the event log can then be closed.

The virtual machine can then be resumed. All record and replay hooks are then skipped.

5.4 Replay

For replaying a recorded virtual machine, QEMU is used in the TCG mode. This this mode, QEMU uses binary translation to emulate the execution of the virtual machine.

5.4.1 Restoring the Initial State

Before starting the replay, the emulator state needs to be restored from the checkpoint that was created for the virtual machine during the start of the recording. For this, the emulator is put in a paused state. The checkpoint is then loaded. This restores the memory and register contents of the processor to the information stored in the checkpoint.

Then, the event log store is opened. The event log contains two event streams per processor, one for the synchronous and one for the asynchronous events. Those streams are then matched to the correct processor. Matching the right processor is required because each processor was restored to a different state when loading the checkpoint and needs to be fed with the correct events for it's state. To match the processors, the virtual CPU ids are used. Both KVM and QEMU in TCG mode use vCPU-ids that start at 0 and are consecutive. By computing the stream index using the virtual CPU id, the same mapping the hypervisor used while creating the event log is achieved.

After opening the event logs, the replay data structures are set up. No commit protocol is used during the replay. Therefore, no data structures to hold copy-on-write pages or page access information is required. Instead, only the instruction count needs to be tracked in oder to emulate the asynchronous events at the right instruction.

The current chunk commits are counted by the emulator. This count is set to the intial value of zero. The emulator can then resume the gust system.

5.4.2 Replay the Chunks

When emulating multiple processors, QEMU schedules the processors in sequence instead of running them in parallel. This is required because the internals of the emulator are not designed to work in a parallel environment [8]. A simple scheduler loops over the virtual CPUs and lets each CPU run a short time. This creates the impression for the emulated system that it is running on a parallel hardware.

This scheduler is adjusted so that the order in which the processors are executed is determined by the replay mechanism. The scheduler now always selects the processor that will do the next commit. For this, the index of the commit event at the end of each chunk was written to the start event of the chunk. Each asynchronous event log contains such a chunk start event as head event. Therefore, the scheduler can walk though all processors and select the one whose commit index matches the next chunk that should be executed. This processor is then running directly on the memory without any copy-on-write mechanism. The results of non-deterministic instructions is read from the synchronous event log.

After each instruction that is executed, the processor checks if it has reached the correct insertion point of the next asynchronous events. If it is reached, it checks the type of the event. For interrupt events, an interrupt is simulated. For chunk commit events, the global chunk counter is increased and processor is yielded. The system is now at the same memory state at which it was when the chunk was committed during recording. The scheduler then searches for the next processor to execute, which is the one that committed next.

5.5 Conclusion

An implementation based on an existing hypervisor for recording and an existing emulator for replay has been achieved.

The implementation showed that there are several changes to the hypervisor required in order to use the commit protocol and that many mechanisms of the hypervisor need to be adjusted in order to support the new per-processor view on the memory. If those special features that would require further adjustment are disabled, the changes to the hypervisor are mostly restricted to hooking into the VM exit event. Depending on the exit type, that event is recorded for later replay and memory is committed or rolled back in this event.

The multi-core replay implementation was shown to be possible with only minor adjustments to the emulator in comparison to a single-core emulator. To support the commit events, only the scheduler for the different cores needed to be adjusted to schedule the cores in the right order.

Chapter 6

Evaluation

The concept for heterogenous report and replay needs to be evaluated in situations in which it will be useful. For this, scenarios that require full system debugging are tested on the implementation.

The evaluation should show that a multi-core system can be recorded and that the replay on the emulator matches the recording. Since scalability is an important factor when supporting multiple cores, it should be evaluated how good the implementation scales when using multiple cores.

The test is done using a Intel Xenon E5-2630 CPU with 16 cores. This processor is based on the 64 bit architecture and supports 48 bit virtual addresses using it's extended page table.

6.1 Evaluation Scenarios

Analyzing a full virtual machine makes sense in places in which there is not just a single user-space program that needs to be analyzed. For this work, two scenarios that both provide such a situation are used.

In the first scenario, a booting operating system is analyzed. When an operating system boots, it first runs some bootstrap code to set the processor and the basic hardware to a known state. This first part of the boot process runs on the primary CPU. It is mostly the same for all system boots and always follows the same pattern. The operating system then starts the other processors and the scheduler. It then initializes the drivers, starts the system services, checks the environment and loads additional modules. All those tasks are done in parallel to improve overall throughput. This makes analyzing this phase of the system boot interesting. This first scenario is operating system heavy and is expected to do a lot of different hardware interactions.

For the evaluation, an ubuntu 16.04.2 guest was used. It was run on QEMU using two cores and 8GiB of memory.

An alternative scenario to analyze would be a running operating system with multiple active processes. The nature of those processes severely influences the evaluation results. Applications that rely heavily on spin locks in shared memory to synchonize their memory access may trigger a lot of memory conflicts. Most applications however have longer time frames in which they do not directly communicate with other processors but only do local computations. This includes server applications, in which the main source of inter-processor communication is the scheduler and the network or disk traffic. Those are the applications this work focuses on. An example application that provides such a typical workload needs to be found.

Building a Linux kernel provides such a typical and mixed workload. It contains both hardware (disk) accesses and memory accesses from multiple processes running in parallel. The individual builds are run as separate processes without much interference, so only few memory conflicts can be expected from them. But the underlying operating system has a disk cache and does a lot of process forking, both of which creates memory conflicts. The scheduler and other parts of the operating system require synchronization mechanisms between the CPUs. That way, this scenario provides a range of access patterns in a realistic balance.

The Linux kernel build is often used in evaluations. It is especially used int he evaluation of *Samsara* [19], on which this recording implementation is based. Therefore, it provides a good base for comparison with other solutions.

6.1.1 Analyzing the Scenarios

Analyzing the system boot may cause unreproducible results, since each boot may be different depending on hardware reaction times and the internal scheduling of the operating system. This makes it difficult to reliably analyze the influences of the memory commit protocol.

Running a parallel kernel build is indeterministic as well. The build process consists of several phases - from parsing the input files, compiling them to binary code and then linking it - that all have different access patterns. The chunk based commit protocol influences the timing for the processors any may delay code execution that causes memory conflicts. This change in processor behavior might itself change the behavior of the running program.

To be able to use a realistic evaluation scenario that does not rely on DMA, the scenarios are first analyzed. They are run once and the memory access patterns of this run are saved. A simple program is then written that mimics those access patterns. That way, the overhead of the recording system can be analyzed in a reproducible way..

To analyze them, the scenarios are started in a virtual machine. This machine then keeps track of the accessed and dirty pages by the reference bits in the EPT. The preemption timer is set so that the thread is preempted when a commit would happen. Additionally, the IO operation functions are instrumented. Since those are the times at which a commit would happen, this allows to determine the number of accessed and written pages in each chunk.

Chunks are not rolled back in this scenario. It is assumed that a processor resumes it's operation after a roll back and therefore most processors do the same memory accesses again after a roll back. Therefore, disabling roll backs should only have a minimal impact on the recorded memory access statistics. The advantage of this is that the complex copy-on-write protocol can be disabled when recording memory accesses. This allows for recording the access pattern of complex programs such as a Linux during boot time without the need for a fully working DMA implementation.

The exact set of pages that was accessed does not need to be tracked. It is sufficient to track the number of pages that were accessed since this corresponds to the sizes of the page sets and to the number of copy-on-write operations.

Therefore, only the total number of accessed and dirty pages is logged for each page table level.

For both scenarios - the Linux system boot and the Linux kernel build - this list of chunks was created. For comparison, the same values were recorded on a idle Linux system. Table 6.1 shows the numbers that were recorded on the test system using two cores.

The number of accesses in the first level of the EPT corresponds to the number of accessed pages. One can see that the number of accessed pages is below 10 in most chunks.

The peak accesses were fluctuating a lot. The top values are around 400 accessed and 300 written pages but they only happening every few thousand accesses. In each scenario, less than 5% of the chunks was above 50 accessed and less tan 5% was above 25 written chunks. This shows that most chunks only access a few pages but that there are peak chunks in which a lot of pages are written. The copy-on-write mechanism and the page set mechanisms need to be able to handle this high number of pages. In the implementation, flexible data structures that do not restrict the number of pages are used so the implementation is able to handle this load.

In addition to the number of touched pages, the number of instructions that were executed was recorded by recording the difference in the icount value.

Table 6.2 shows the icount differences for the chunks in the different test cases. In the idle case, the processors spend most time in a halt instruction, which is why only the active time of the processors is used in this statistic. The non-halted time resembles the one at system boot, since the operating system is active then.

Scenario	Linux boot		Kernel build		Linux idle	
Sample size	62976 chunks		27136 chunks		20992 chunks	
1st level EPT	accessed	written	accessed	written	accessed	written
75% median	16	12	17	14	10	9
Median	6	5	5	4	6	5
25% median	1	1	2	2	2	1
2nd level EPT	accessed	written	accessed	written	accessed	written
75% median	8	6	10	8	6	5
Median	4	3	3	3	4	3
25% median	1	1	2	2	2	1
3rd level EPT	accessed	written	accessed	written	accessed	written
75% median	1	1	2	2	2	2
Median	1	1	2	2	2	1
25% median	1	1	2	2	1	1

Table 6.1: Number of access or dirty bits set per chunk. Level 4 is ignored since the test address space of $8GiB$ only requires three levels.

Scenario	Linux boot	Kernel build	Linux idle
Minimum	0	0	0
Median	255	873	239
75%-Median	3298	27927	1531
Maximum	749875	1436033	1436033

Table 6.2: Instruction counts per chunk in the different evaluation scenarios

In each scenario, there are chunks that have a length of zero. Those are chunks in which the virtual machine monitor was trapped twice for the same instruction. This may happen e.g. if the guest trapps into the hypervisor because of a device IO operation while an interrupt happens. The interrupt is then handled by the VMM directly and a chunk with zero length is inserted.

The implementation is able to handle those zero length chunks by adding an empty commit event. They cannot be dropped because they may indicate a barrier at which the memory state needs to be updated by the other cores. Therefore, even those zero length commits are recorded.

Most chunks have a length of a few hundred instructions. This happens for example during IO operations of the operating systems. The OS communicates with the hardware using several IO operations in sequence with a short sequence in the virtual machine to read the IO results and compute the next IO operation.

Computation intensive chunks could probably have a length of several million instructions. The preemption timer is used to restrict that maximum size. The way it was set during the tests, the maximum achievable chunk size was 1436033. Since the timer interrupt is not exact, there are various chunks a few cycles shorter than this. While those chunks are few, they account for most of the computation time and most of the memory accesses during a kernel build.

The maximum length of a chunk can be tuned by altering the preemption timer value to ensure a good commit performance. Such tuning was done in Samsara. In this work, the commit performance was not a primary objective and therefore the chunk length parameter was not tuned.

Using those values, a small boot environment is built that simulates those access patterns while not relying on unimplemented features like DMA. This allows the validation of the prototype implementation.

6.2 Validation

The most important part of a replay is to ensure that the replay behaves exactly as the recording does.

In a valid replay, all instructions should be executed in exactly the same way as they were during recording. Validating this directly is difficult, since a trace of every executed instruction would need to be done during recording. Doing this using the hypervisor would interfere with the recording mechanism since the hypervisor needs to be instructed to pause after every instruction. A valid evaluation would not be possible this way.

There are several alternatives to this. One is to use a hardware tracing mechanism. Hardware tracing is very complex as tracing the memory accesses requires spcialized hardware.

A simpler approach is to design the test program in a way that the execution of the instrucitons can be validated by not looking at the instructions themselves but only at the program results. For this, the test program is written so that it's instructions modify the main memory depending on the way or order they were executed. For instructions that do not modify main memory directly, a subsequent instruction can write their changes to main memory.

Instructions that access IO registers can for example write the results of that access and the processor registers that were read to a predefined area in main memory.

To evalute the correctness of the replay, the recording is paused at a random point. A memory snapshot is taken at the end of the recording. That snapshot is then compared to the memory state after the replay. Since the test program was written in a way that the snapshots would differ if the execution of the program was different, the validity of the replay is proven if the two snapshots match.

This test only needs to tell that the two snapshots contain exactly the same memory content. Since no information about the individual memory cells is required, it is sufficient to compare a checksum over the memory content and compare that checksum. For good checksums like SHA256, the likelihood of a checksum collision is minimal. This test is repeated multiple times. Since the point at which recording is ended is chosen randomly, intermediate memory differences may also be found with this approach.

Doing a snapshot during the recording is not possible. Each processor has it's own view of the main memory when recording. Only at a commit, the processor memory state is the same as the main memory state. One could take the memory state the committing processor sees, but this state does not include the changes done by other processors since those processors use copy-on-write. Therefore, all processors would need to be paused and a commit would need to be forced.

This is the same operation that is done at the end of the recording. Therefore, instead of pausing the recording, the recording can be ended. Then, a checksum on the main memory is computed. By repeating the whole recording multiple times in different lengths the same effect as when pausing and resuming the recording can be achieved.

To run this test, a small boot environment that boots multiple processors was written. Each processor uses a pseudo-random number generator to compute the memory pages it should access. This creates race conditions between the processors. Delay loops that only operate on the stack are used to simulate processor local computations.

For this program, a recording session was started at a random point. It was then running for a random time. Then, it was stopped. The memory checksum was computed and printed to the serial output. Then, the program was replayed and the same memory checksum computed. This step was repeated 10 times to ensure for a valid replay.

In all those iterations, the checksum did match. This showed that the chunks were recorded in the correct order.

6.3 Scalability

With the trend of increasing numbers of processors in consumer computers, it is possible to allocate more processors to a virtual machine. The record and replay mechanism works with a unlimited number of processors in theory. However, it needs to be evaluated how well it scales with the number of processors.

6.3.1 Performance

There are two ways performance can be reduced when increasing the number of processors.

The first way is by an increased chance for chunk conflicts which would increase the number of rollbacks. If the client program uses more processors, the chances of the processors producing a conflict may be higher. This depends on the amount of shared memory accesses the processors do. Depending on the application, the performance impact can be none if the processors do not work on the same data or very high if it often uses synchronization mechanisms between processors. Therefore, the commit protocol may scale well with some applications like web servers but may not scale well with applications that have a lot of shared memory access like database servers.

The second way the performance is reduced is by longer commit times. To commit, the processor needs to acquire a global lock for the commit data structures and for updating the conflict maps of other processors. Holding this lock, the processor needs to update the conflict map of each other processor and set it's own dirty bits in those maps. This way, the time the lock needs to be held by each processor increases linear with the number of processors.

Assuming the chunk size stays the same when increasing the number of processors, each processor still needs to acquire this lock in the same interval. Since the lock is global, the number of times an entity wants to acquire the lock increases linear with the number of processors.

So the total increase of the time the lock is held is squared when doubling the number of processors. *Samsara* [19] evaluated this lock to be one of the main performance problems. Their results showed that the time consumption while waiting for the lock is 40% on a system with four cores for an unoptimized case and still considerable high when using a parallel write back of the copy-on-write pages.

Event when using more fine-grained locking during the commit, the time to update the dirty page sets scales linear with the number of processors. Therefore, this may be the main issue when increasing the number of processors and an alternative to the conflict detection protocol needs to be found in order to maintain performance.

6.3.2 Memory Consumption

There are both global and processor-local data structures required for the commit and rollback protocol.

The global data structures are the commit index counter and the commit synchronization mechanism. They are of constant size, so an increased number of processors does not increase the required memory.

The remaining data structures are required for each processor.

Each processors requires a local accessed, dirty and conflict map. The size of the accessed and dirty maps is linear to the number of touched memory pages in a chunk. That number is determined by the length of the chunk and the type of work the processor did in the chunk. During recording, the a timer interrupt is used to limit the length of the chunk. If it's parameter is not changed, the maximum length of a chunk stays the same if the number of processors is increased. Assuming the frequence of IO events that trigger a forced commit is not increased by the target application, the same program can be expected to produce the same number of touched pages on one CPU. The accessed and dirty page set each use $4kiB$ per set page in their last level if the bits cannot be grouped. The previous levels of the map are ignored for this evaluation, since this is just a rough estimate and previous evaluations showed that they are close to 1 page per level. As previous evaluation has shown, most chunks are well below 100 accessed pages. By triggering a commit event when this limit is reached, that bond could even be enforced. Assuming a maximum of 100 accessed pages per chunk, the upper bound for the memory used by the accessed page set M_a and by the dirty page set M_d is shown in equation 6.1.

$$M_a = M_d = 4kiB \cdot 100 = 400kiB \tag{6.1}$$

The conflict page set is a union of all pages that were touched by other processors during the current chunk. For this evaluation, it is assumed that any other processor accesses 100 pages during the current chnk as well. As whit the local accesses, this bound could also be enforced by rolling back chunks that have grown to big or by using inter-processor communication to force a commit on the other processor. Because it is a union, the size of the conflict set increases linear to the number of other processors in the system. The memory consumption per processor for the conflict page set M_c is shwon in equation 6.2, where n is the total number of virtual processors.

$$M_c = ((n-1) \cdot 100) \cdot 4kiB \tag{6.2}$$

For an increased memory, this depends on the way the program handles more allocated memory. The previous evaluations showed that most chunks have a memory access count of less than 100 for real world programs. For this evaluation, a memory access count of 100 is assumed and the worst case scenario - that those hits are distributed evenly over the memory and that there are no space saving effects from storing multiple neighboring pages in one page set entry. This number orders of magnitudes smaller than the total number of pages in the evaluated system. So even increasing the main memory size will probably not increase the total number of pages touched by the program in one chunk.

Each processor holds it's own set of copy-on-write memory pages. Each of those pages has a size of $4kiB$. The number of extra pages a processor requires is the same as the number of bits set in the dirty page set. As shown above, the size of this page set is not expected to increase with an increased number of processors or a increased memory size. Therefore, the total size of those pages M_{cow} can be computed by multiplying the page size with the number of pages copied, as shown in equation 6.3. For each processor, a constant overhead can be assumed for the copy-on-write pages.

$$M_{cow} = 4kiB \cdot 100 = 400kiB \tag{6.3}$$

In addition to the constant size data structures and the memory maps, each processor requires it's own extended page table. This is required because the accessed flags in those tables need to be tracked per processor and each processor needs to have it's copy-on-write pages mapped separately.

For each extra memory page that is made available to the guest, the EPT needs an additional entry. If the guest uses a contiguous, like normal operting systems do, no sparse entries are in the page tables. So the total size of the last leven page tables is 8 bytes per such page, rounded up to the full page size since the table for the highest address may not be fully filled. Additional memory is required for the page tables with a higher level, but their total memory consumption is limited in the same way by the number of lower level page tables. That way, their number decreases by 512 on each level for contiguous memory allocations, so their total number is far less than the number of last level page tables. Therefore, the extended page table grows linear to the total number of pages that are made available to the guest. For a contiguous area of m memory pages, the size required for a contiguous EPT allocation M_{ept} is stated in equation 6.4.

$$M_{ept} = (\left\lceil \frac{m}{512} \right\rceil + \left\lceil \frac{m}{512^2} \right\rceil + \left\lceil \frac{m}{512^3} \right\rceil + 1) \cdot 4kiB < m \cdot 4kiB \qquad (6.4)$$

The linear growth of the page table size would also be the case if a sparse memory is used. In that case, at most one page table per level is used per guest memory page, which summs up to a page table size that is limited by 4 (the number of levels) times the memory size of the guest.

$$\begin{aligned} & n \cdot (M_a + M_d + M_c + M_{cow} + M_{ept}) \\ = & \, n \cdot (2 \cdot 100 \cdot 4kiB + (n-1) \cdot 100 \cdot 4kiB + 100 \cdot 4kiB + M_{ept}) \\ = & \, n \cdot (200 \cdot 4kiB + n \cdot 100 \cdot 4kiB + M_{ept}) \\ < & \, (200 \cdot n + 100 \cdot n^2 + n \cdot m) \cdot 4kiB \end{aligned} \qquad (6.5)$$

Since the processors each require their own data structures, the memory overhead for recording can be put in relation to n processors on m pages of contiguous memory, as seen in equation 6.5. The memory consumption for the chunk based memory data structures increases squared with the number of processors due to the conflict page sets. It increases linear with the number of memory pages, since the size of the EPT needs to be increased as well then.

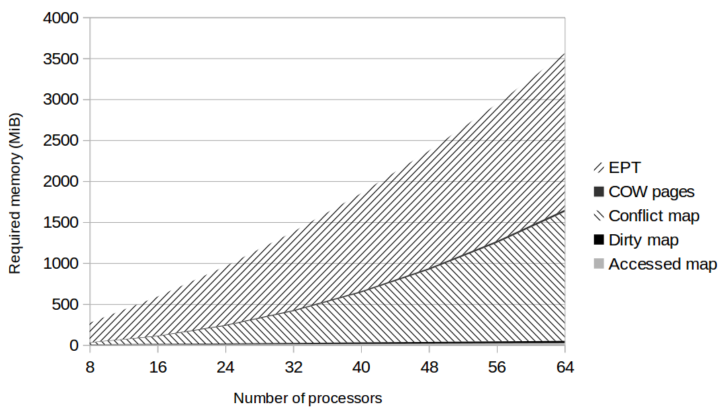

Figure 6.1: This chart shows the additional memory required when recording a virtual machine with $16GB$ of guest physical memory. The memory consumption increases more than linear as the number of processors increases.

This is illustrated in Figure 6.1. For this virtual machine with $16GB$ of memory, the additional memory overhead would be 1.77% for a virtual machine with eight cores but 23% with 64 cores. For a $64GB$ virtual machine, the memory overhead would only be 15%. The memory overhead is relatively high but still well within acceptable sizes.

The memory for the temporary event lists is not evaluated, since the temporary queue only contains the events for a few millisecodns. The events are written back to disk in the order of seconds. The event buffer is required per processor. It can be written back once it is full, so the total size occupied by the event buffer increases linear to the number of processors.

6.4 Conclusion

The evaluation has shown that a correct replay of a recorded program is possible on a different target platform.

The performance overhead for number of processors of a current consumer PC is acceptable. For a higher number of cores, difficulties may be expected.

The memory overhead of the recording mechanism is low enough so that it will scale with the number of cores that are to be expected within the next decade.

Chapter 7

Conclusion

A heterogeneous record and replay is possible even with multiple processors. A virtual machine can be recorded and later replayed using an emulator.

A chunk based memory access protocol can be used during the recording phase to ensure a deterministic access to the main memory. While this can be implemented in hardware [11, 18], a software only approach is possible using the virtualisation features of modern processors [19]. While the software based approach was evaluated in theory before, only a recording prototype was build previously. No replay of such a recording has been done, neither on the source system nor on a different system.

In this work, the possibility to replay such a recording was evaluated. With a focus on memory race conditions, a prototype was build that records the memory access orders and makes them deterministic that way. They were then replayed using an emulator that simulates the source system. It could be shown that the memory race conditions were detected successfully and replayed in the right order.

Since a key feature of multi-core programs is scalability, the scalability of the record and replay solution was evaluated. It could be shown that the memory required for tracking the memory access patterns lies within an acceptable size in comparison to the memory size the virtual machine uses, even when tracking a large number of cores.

The performance has been shown to not perform well on many processors, mainly because of a global lock that prevents the application from scaling. In further work, a more fine grained locking mechanism could be evaluated to solve this issue.

The mechanisms to replay the application both on multiple hardware processors and on a emulated multi-processor system have been discussed. The replay was implemented using an emulator that schedules the virtual processors all on one physical processor. It has been shown that a correct replay can be achieved using this. The ability to run the replay in parallel to achieve higher replay performance was shown to be possible. It can be further evaluated in future work on the subject to measure the real performance benefits of this approach.

Bibliography

[1] Arkaprava Basu, Jayaram Bobba, and Mark D Hill. Karma: scalable deterministic record-replay. In *Proceedings of the international conference on Supercomputing*, pages 359–368. ACM, 2011.

[2] Adam Belay, Andrea Bittau, Ali José Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Osdi*, volume 12, pages 335–348, 2012.

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

[4] Sanjay Bhansali, Wen-Ke Chen, Stuart De Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 154–163. ACM, 2006.

[5] Jim Chow, Tal Garfinkel, and Peter M Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 1–14, 2008.

[6] E. G. Cota, P. Bonzini, A. BennÃ©e, and L. P. Carloni. Cross-isa machine emulation for multicores. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 210–220, Feb 2017.

[7] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62. ACM, 2008.

[8] Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, and Yeh-Ching Chung. Pqemu: A parallel system emulator based on qemu. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 276–283. IEEE, 2011.

[9] George Dunlap. Execution replay for intrusion analysis. *D thesis, University of Michigan*, 2006.

[10] George W Dunlap, Samuel T King, Sukru Cinar, Murtaza A Basrai, and Peter M Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review*, 36(SI):211–224, 2002.

[11] Derek R Hower and Mark D Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *ACM SIGARCH computer architecture news*, volume 36, pages 265–276. IEEE Computer Society, 2008.

[12] Thomas J LeBlanc and John M Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, (4):471–482, 1987.

[13] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing cpu emulators. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 261–272. ACM, 2009.

[14] Pablo Montesinos, Luis Ceze, and Josep Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution ef? ciently. In *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*, pages 289–300. IEEE, 2008.

[15] Mozilla. rr-project: lightweight recording deterministic debugging. `http://rr-project.org/`. Accessed: 2017-10-20.

[16] Satish Narayanasamy, Cristiano Pereira, and Brad Calder. Recording shared memory dependencies using strata. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 229–240. ACM, 2006.

[17] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 2–11. ACM, 2010.

[18] Gilles Pokam, Cristiano Pereira, Klaus Danne, Rolf Kassa, and Ali-Reza Adl-Tabatabai. Architecting a chunk-based memory race recorder in modern cmps. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 576–585. ACM, 2009.

[19] Shiru Ren, Le Tan, Chunqi Li, Zhen Xiao, and Weijia Song. Samsara: Efficient deterministic replay in multiprocessor environments with hardware virtualization extensions. In *USENIX Annual Technical Conference*, pages 551–564, 2016.

[20] 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*, 2015.

[21] Yasushi Saito. Jockey: a user-space library for record-replay debugging. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 69–76. ACM, 2005.

[22] Dinesh Subhraveti. *Record and vPlay: Problem Determination with Virtual Replay Across Heterogeneous Systems*. Columbia University, 2012.

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

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