Samsara: Efficient Deterministic Replay in Multiprocessor Environments with Hardware Virtualization Extensions

Shiru Ren, Le Tan, Chunqi Li, and Zhen Xiao, ***Peking University;***
Weijia Song, ***Cornell University***

https://www.usenix.org/conference/atc16/technical-sessions/presentation/ren

This paper is included in the Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC '16).

June 22–24, 2016 • Denver, CO, USA

978-1-931971-30-0

Open access to the Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC '16) is sponsored by USENIX.

Samsara: Efficient Deterministic Replay in Multiprocessor Environments with Hardware Virtualization Extensions

Shiru Ren[1], Le Tan[1], Chunqi Li[1], Zhen Xiao[1], and Weijia Song[2]

[1]*Department of Computer Science, Peking University*
[2]*Department of Computer Science, Cornell University*

Abstract

Deterministic replay, which provides the ability to travel backward in time and reconstruct the past execution flow of a multiprocessor system, has many prominent applications. Prior research in this area can be classified into two categories: hardware-only schemes and software-only schemes. While hardware-only schemes deliver high performance, they require significant modifications to the existing hardware which makes them difficult to deploy in real systems. In contrast, software-only schemes work on commodity hardware, but suffer from excessive performance overhead and huge logs caused by tracing every single memory access in the software layer.

In this paper, we present the design and implementation of a novel system, Samsara, which uses the hardware-assisted virtualization (HAV) extensions to achieve efficient and practical deterministic replay without requiring any hardware modification. Unlike prior software schemes which trace every single memory access to record interleaving, Samsara leverages the HAV extensions on commodity processors to track the read-set and write-set for implementing a chunk-based recording scheme in software. By doing so, we avoid all memory access detections, which is a major source of overhead in prior works. We implement and evaluate our system in KVM on commodity Intel Haswell processor. Evaluation results show that compared with prior software-only schemes, Samsara significantly reduces the log file size to 1/70th on average, and further reduces the recording overhead from about 10×, reported by state-of-the-art works, to 2.3× on average.

1 Introduction

Modern multiprocessor architectures are inherently non-deterministic: they cannot be expected to reproduce the past execution flow exactly, even when supplied with the same inputs. The lack of reproducibility complicates software debugging, security analysis, and fault-tolerance. It greatly restricts the development of parallel programming.

Deterministic replay helps reconstruct non-deterministic processor executions. It is extensively used in a wide range of applications. For software debugging, it is the most effective way to reproduce bugs, which helps the programmer understand the causes of the bug [1, 33]. For security analysis, it can help the system administrator analyze the intrusions and investigate whether a specific vulnerability was exploited in a previous execution [17, 18, 7, 11, 37]. For fault-tolerance, it provides the ability to replicate the computation on processors for building the hot-standby system or data recovery [6, 43, 42, 32].

In the multiprocessor environment, memory accesses from multiple processors to a shared memory object may interleave in any arbitrary order, which become a significant source of non-determinism and pose a formidable challenge to deterministic replay. To address this problem, most of the existing research focuses on how to record and replay the memory access interleaving using either a pure hardware scheme or a pure software scheme.

Hardware-only schemes record memory access interleaving efficiently by embedding special hardware components into the processors and redesigning the cache coherence protocol to identify the coherence messages among processors [38, 22, 21, 15, 9, 29, 14, 35, 23, 30]. The advantage of such a scheme is that it allows efficient recording of memory access interleaving in a multiprocessor environment. On the down side, it requires extensive modifications to the existing hardware, which significantly increases the complexity of the circuits and makes them largely impractical in real systems.

In contrast, software-only schemes achieve deterministic replay on the existing hardware by modifying the OS, the compiler, the runtime libraries or the virtual machine manager (VMM) [19, 13, 8, 34, 33, 34, 25, 2,

26, 20, 41, 4]. Among them, virtualization-based deterministic replay is one of the most promising approaches which provides full-system level replay by leveraging the concurrent-read, exclusive-write (CREW) protocol to serialize and log the total order of the memory access interleaving [13, 8, 27]. While these schemes are flexible, extensible, and user-friendly, they suffer serious performance overhead (about 10× compared to the native execution) and generate huge logs (approximately 1 MB/s on a four core processor after compression). The poor performance can be ascribed to the numerous page fault VM exits led by tracing every single memory access in the software layer.

To summarize, it is inherently difficult to record memory access interleaving efficiently by software alone without proper hardware support. Although there is no commodity processor with dedicated hardware-based record and replay capability, some advanced hardware features in these processors are available to boost the performance of the software-based deterministic replay systems. Therefore, we argue that the software scheme can be a viable approach in the foreseeable future if it can take advantages of advanced hardware features.

In this paper, the main goal is to implement a software approach that can take full advantages of the latest hardware features in commodity processors to record and replay memory access interleaving efficiently without introducing any hardware modifications. The emergence of hardware-assisted virtualization (HAV) provides the possibility to meet our requirements. Although HAV cannot be used for tracing memory access interleaving directly, we have found a novel use of it to track the read-set and write-set, and bypass the time-consuming process in traditional software schemes. Specifically, we abandon the inefficient CREW protocol that records the dependence between individual instructions, and instead use a chunk-based strategy that records processors' execution as a series of chunks. By doing so, we avoid all memory access detections, and instead obtain each chunk's read-set and write-set by retrieving the accessed and the dirty flags of the extended page table (EPT). These read and write sets are used to determine whether a chunk could be committed, and the determinism is ensured by recording the chunk size and the commit order. To further improve the system performance, we propose a decentralized three-phase commit protocol, which significantly reduces the performance overhead by allowing chunk commits in parallel while still ensuring serializability.

We implement our prototype, Samsara, which, to the best of our knowledge, is the first software-based deterministic replay system that can record and replay memory access interleaving efficiently by leveraging the HAV extensions on commodity processors. Experimental results show that compared with prior software schemes based on the CREW protocol, Samsara reduces the log file size to 1/70th on average (from 0.22MB/core/second to 0.003MB/core/second) and reduces the recording overhead from about 10× to 2.3× compared to the native execution.

Our main contributions are as follows:

- We present a software-based deterministic replay system that can record and replay memory access interleaving efficiently by leveraging the HAV extensions. It improves the recording performance dramatically with a log size much smaller than all prior approaches.
- We design a decentralized three-phase commit protocol, which further improves the performance by enabling the chunk commit in parallel while ensuring serializability.
- We build and evaluate our system in KVM on Intel Haswell processor, and we plan to open-source our system to the community.

The rest of the paper is organized as follows. Section 2 describes the general architecture and shows how Samsara achieves deterministic replay. Section 3 illustrates how to record and replay the memory access interleaving. Section 4 presents the optimization and the implementation details. We evaluate Samsara in section 5. Section 6 reviews related work and section 7 concludes the paper.

2 System Overview

In this section, we present the system overview of Samsara. We first outline the overall architecture of Samsara. Then, we discuss how it records and replays all non-deterministic events.

2.1 System Architecture

Samsara implements the deterministic replay as an extension to VMM, which has access to the entire virtual machine and can take full advantage of the HAV extensions, as illustrated in Figure 1. The architecture of Samsara consists of four principal components, namely, the Controller, the record and replay component, the DMA recorder, and the log record daemon as shown in orange boxes in the figure. The controller is in charge of all policy enforcement. It provides a control interface to users, manages the record and replay component in KVM, and is in charge of the log transfer. The record and replay component acts as a part of VMM working in the kernel space being responsible for recording and replaying all

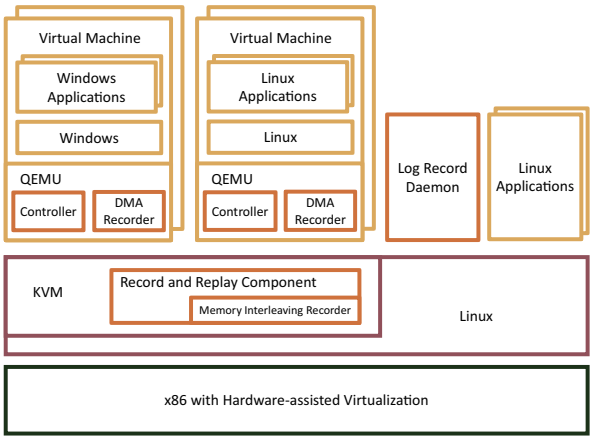

Figure 1: Architecture overview.

non-deterministic events, especially the memory access interleaving. The DMA recorder records the contents of DMA events as part of QEMU. Finally, we optimize the performance of logging by utilizing a user-space log record daemon. It runs as a background process that supports loading and storing log files.

Samsara implements deterministic replay by first logging all non-deterministic events during the recording phase and then reproducing these events during the replay phase. Before recording, the controller initializes a snapshot of the whole VM states. Then all non-deterministic events and the exact points in the instruction stream where these events occurred will be logged by the record and replay component during recording. Meanwhile, it transfers these log data to the userspace log record daemon, which is responsible for the persistent storage and the management of the logs. The replay phase is initialized by loading the snapshot to restore all VM states. During replay, the execution of the virtual processors is controlled by the record and replay component which ignores all external events. Instead each recorded event will be injected at the exact same point as in the recorded execution.

2.2 Record and Replay Non-deterministic Events

Non-deterministic events fall into three categories: synchronous, asynchronous, and compound. The following illustrates what events will be recorded and how recording and replaying is done in our system.

Synchronous Events. These events are handled immediately by the VM when they occur. They always take place at the exact same point where they appear in the instruction stream, such as I/O events and RDTSC instructions. The key observation is that they will be triggered by the associated instructions at the fixed point if all previous events are properly injected. Therefore, we just record the contents of these events. During replay, we merely inject logged data to where the I/O (or RDTSC) instruction is trapped into the VMM.

Asynchronous Events. These events are triggered by external devices, such as external interrupts, so they may appear at any arbitrary time from the point of view of the VM. Their impact to the state of the system is deterministic, but the timing of their occurrences is not. To replay them, all such events must be identified with a three-tuple timestamp (including program counter, branch counter, and the value of ECX) like the approach in ReVirt [12]. The first two are used to uniquely identify the instruction where the event appears in the instruction stream. However, the x86 architecture introduces the REP prefixes to repeat a string instruction the number of times specified in the ECX. Therefore, we also need to log the value of ECX which stores how many iterations remain at the time of this event takes place [12]. During replay, we leverage a hardware performance counter to guarantee that the VM stops at the recorded timestamp to inject them.

Compound Events. These events are non-deterministic in both their timing and their impact on the system. DMA is an example of such events: the completion of a DMA operation is notified by an interrupt which is asynchronous, and the data copy process is initialized by a series of I/O instructions which are synchronous. Hence, it is necessary to record both the completion time and the content of a DMA event.

Memory Access Interleaving. In the multiprocessor environment, memory accesses from multiple processors to a shared memory object may interleave in any arbitrary order, which become a significant source of non-determinism. More specifically, if two instructions both access the same memory object and at least one of them is write, then the access order of these two instructions should be recorded during the recording phase. Unfortunately, the number of such events is orders of magnitude larger than all the other non-deterministic events combined. Therefore, how to record and replay these events is the most challenging problem in a replay system.

3 Record and Replay Memory Access Interleaving with HAV Extensions

How to record and replay memory access interleaving efficiently is the most significant challenge we face during the design and implementation of Samsara. In this section, we describe on how Samsara uses HAV extensions to overcome this challenge. Firstly, we show the specific design of our chunk-based strategy, then we discuss two

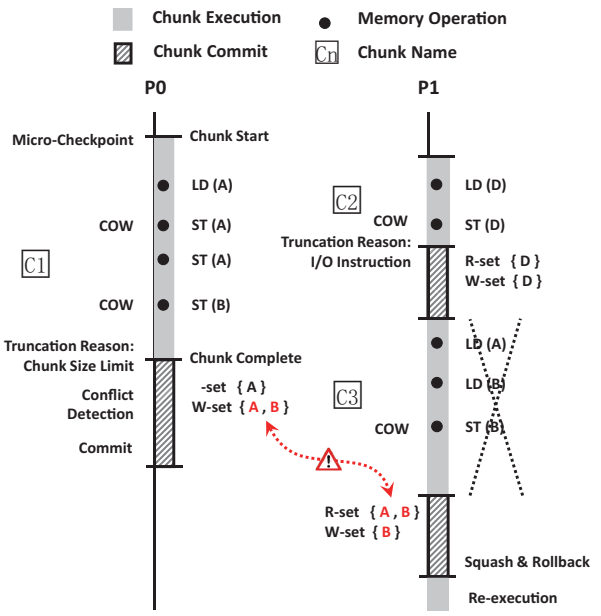

Figure 2: The execution flow of our chunk-based approach.

technical issues when implementing this strategy in software: how to obtain the read-set and write-set efficiently and how to reduce the commit overhead. Finally, we give a brief description on how to replay the memory access interleaving in Samsara.

3.1 Chunk-based Strategy

Previous software-only schemes leverage CREW protocol to serialize and log the total order of the memory access interleaving [19], which produces huge log size and excessive performance overhead because every single memory access needs to be checked for logging before execution. Therefore, chunk-based approach has been proposed on the hardware-based replay system to reduce the log size [21]. In this approach, each processor executes instructions grouped into chunks. Thus, it just needs to record the total order of chunks. However, this approach is not directly applicable to a software-only replay system, because tracing every single memory access to obtain the read-set and write-set during chunk execution in software will still be as time-consuming as directly logging the memory access interleaving itself. To eliminate this performance overhead, we find HAV extension extremely useful. Instead of tracing every single memory access, HAV offers a fast shortcut to track the read-set and write-set, which can be used to implement the chunk-based approach in software layer.

To implement a chunk-based recording scheme, we need to divide the execution of virtual processors into a series of chunks. In our system, a chunk is defined as a finite sequence of machine instructions. Similarly to the database transaction, chunk execution must satisfy the atomicity and serializability requirements. Atomicity requires that the execution of each chunk must be "all or nothing". Serializability requires that the concurrent execution of chunks have to result in the same system state as if these chunks were executed serially.

To enforce serializability, firstly, we must guarantee no update within a chunk is visible to other chunks until it commits. Thus, on the first write to each memory page within a chunk, we create a local copy on which to perform the modification by leveraging copy-on-write (COW) strategy. When a chunk completes execution, it either gets committed, copying all local data back to the shared memory, or gets squashed, discarding all local copies. Moreover, an efficient conflict detection strategy is necessary to enforce serializability. Particularly, an executing chunk must be squashed and re-executed when its accessed memory pages have been modified by a newly committed chunk. To optimize recording performance, we leverage lazy conflict detection. Namely, we defer detection until chunk completion. When a chunk completes, we obtain the read-set and write-set (R&W-set) of this chunk. We intersect all write-sets of other concurrent chunks with this R&W-set afterwards. If the intersection is not empty, which means there are collisions, then this chunk must be squashed and re-executed. Note that the write-write conflict must be detected even if there is no read in these chunks. Specifically, the conflict detection is implemented at the page-level granularity, therefore any attempts to make the write-conflicting chunks serial may overwrite uncommitted data and cause a lost update. Finally, there are certain instructions that may violate atomicity because they lead to externally observable behaviors (e.g., I/O instructions may modify device status and control activities on a device). Once any of such instructions has been executed in a chunk, this chunk could no longer be rolled back. Therefore, we truncate a chunk when any of such instructions is encountered. Then the execution of such instructions must be deferred until this chunk can be committed.

Figure 2 illustrates the execution flow of our chunk-based approach. First, we make a micro-checkpoint of the status of a virtual processor at the beginning of each chunk. During chunk execution, the first write to each memory page will trigger a COW operation that creates a local copy. All the following modifications to this page will be performed on this copy until chunk completion. A currently running chunk will be truncated when an I/O operation occurs or if the number of instructions executed within this chunk reaches the size limit. When a chunk completes, we obtain its R&W-set. Then the con-

flict detection is done by intersecting its own R&W-set with all W-sets of other chunks which just committed during this chunk execution. If the intersection is empty (as C1 or C2 in Figure 2), this chunk can be committed. Finally, we record the chunk size and the commit order which together are used to ensure that this chunk will be properly reconstructed during replay. Otherwise (as C3 in Figure 2), all local copies will be discarded and we rollback the status of the virtual processor with the micro-checkpoint we made at the beginning and re-execute this chunk.

In our design, there are two major challenges: 1) how to obtain the R&W-set (section 3.2); 2) how to commit the chunks in parallel while ensuring serializability (section 3.3).

3.2 Obtain R&W-set Efficiently via HAV

The biggest challenge in the implementation of a chunk-based scheme in software is how to obtain the R&W-set efficiently. Hardware-based schemes achieve this by tracing each cache coherence protocol message. However, doing so in software-only schemes will result in serious performance degradation.

Fortunately, the emergence of HAV provides the possibility to reduce this overhead dramatically. HAV extensions enable efficient full-system virtualization utilizing the help from hardware capabilities. Take Intel Virtualization Technology (Intel VT) as an example. It provides hardware support for simplifying x86 processor virtualization. The EPT that provided in HAV is a hardware-assisted address translation technology, which can be used to avoid the overhead associated with software managed shadow page tables. Intel Haswell microarchitecture also introduces the accessed and dirty flags for EPT, which enables hardware to detect which page has been accessed or updated during execution. More specifically, whenever the processor uses an EPT entry as part of the address translation, it sets the accessed flag in that entry. In addition, whenever there is a write to a guest-physical address, the dirty flag in the corresponding entry will be set. Therefore, by utilizing these hardware features, we can obtain the R&W-set by gathering all leaf entries where the accessed or the dirty flag is set, which can be archived by a simple EPT traversal.

Moreover, the tree-based design of EPT makes it possible to further improve performance. EPT uses a hierarchical, tree-based design which allows the subtrees corresponding to some unused part of the memory to be absent. A similar feature is also present for the accessed and the dirty flags. For instance, if the accessed flag of one internal entry is 0, then the accessed flags of all page entries in its subtrees are definitely 0. Hence, it is not necessary to traverse these subtrees. In practice, due to locality of reference, the access locations of most chunks are adjacent. Thus, we usually just need to traverse a tiny part of EPT, which incurs negligible overhead.

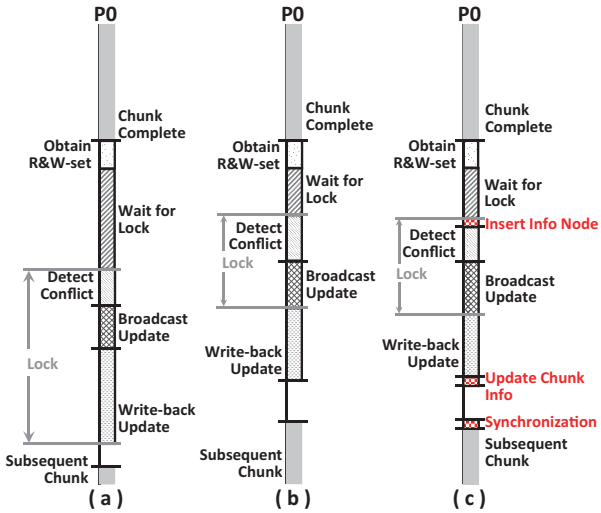

Figure 3: General design of decentralized three-phase commit protocol: a) chunk timeline of a naïve design, b) moving update write-back operation out of the synchronized block, and c) a design of decentralized three-phase commit protocol.

3.3 A Decentralized Three-Phase Commit Protocol

Apart from obtaining the R&W-set, chunk commit is another time-consuming process. In this section, we discuss how to optimize this part using a decentralized three-phase commit protocol.

Some hardware-based solutions add a centralized arbiter module to processors to ensure that one chunk gets committed at a time, without overlapping [21]. However, when it comes to software-only schemes, an arbiter will be slow. Thus, we propose a decentralized commit protocol to perform chunk commit efficiently.

The chunk commit process includes at least three steps in our design: 1) conflict detection that determines whether this chunk can be committed, 2) update broadcast that notifies other processors which memory pages are modified, 3) update write-back that copies all updates back to shared memory. A naïve design of the decentralized commit protocol is shown in Figure 3 a). Without a centralized arbiter, we leverage a system-wide lock to enforce serializability. Each virtual processor maintains three bitmaps: an access bitmap, a dirty bitmap, and a conflict bitmap. The first two bitmaps help mark which memory pages were accessed or updated dur-

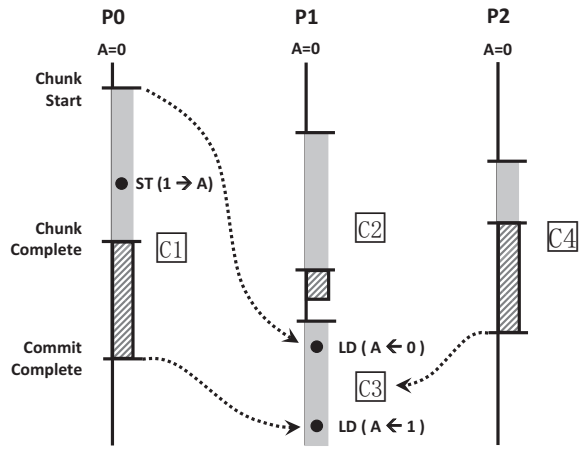

Figure 4: An example of out-of-order commit.

ing the chunk execution (same as the R&W-set). Each bit in the conflict bitmap indicates whether its corresponding memory page was updated by other committing chunks. To detect conflict, we just need to intersect the first two bitmaps with the last one. If the intersection is empty which means this chunk can be committed, this virtual processor broadcasts its W-set to notify others which memory pages have been modified by performing a bitwise-OR operation between the other virtual processors' conflict bitmaps and its own dirty bitmap. Then it copies its local data back to the shared memory. Finally, it clears its three bitmaps before the succeeding chunk starts. This whole commit process is performed while holding this lock.

However, lock contention turns out to cause significant performance overhead. In our experiments, it contributes to nearly 40% of the time spent on committing the chunks. To address this issue, we redesign the commit process to reduce the lock granularity. We observe that the write-back operation involves serious performance degradation due to lots of page copies, and all these pages committed concurrently by different chunks have no intersection, which is already guaranteed by conflict detection. Based on this observation, we move this operation out of the synchronized block to reduce the lock granularity, as shown in Figure 3 b). This not only reduces the cost of the locking operation substantially, but also increases parallelism because multiple chunks can now commit concurrently.

However, one side effect of this design is that chunks may get committed out-of-order, thereby violating serializability. One example is shown in Figure 4. C1 writes A, then finishes its execution first and starts to commit. Then, C2 starts committing as well and finishes before C1. Meanwhile C3 starts to execute and happens to read A immediately. Unfortunately, C1 may not accomplish its commit process in such a short period, thus C3 fetches the obsolete value of A. Suppose C3 reads A again and gets a new value after C1 completes its commit. Then C3 gets two different values of the same memory object, which violates serializability. To maintain serializability, we need to guarantee that before starting C3, P1 waits until all the other chunks which start committing prior to the commit point of C2 (e.g., C1 and C4) complete their commit.

We develop a decentralized three-phase commit protocol to support parallel commit while ensuring serializability. To eradicate out-of-order commits, we introduce a global linked list, *commit_order_list*, which maintains the order and information of each current committing chunk. Each node of this list contains a commit flag field to indicate whether the corresponding chunk has completed its commit process. Moreover, this list is kept sorted by the commit order of its corresponding chunk. A lock is used to prevent multiple chunks from updating this list concurrently. This protocol consists of three phases as shown in Figure 3 c):

1) The pre-commit phase: In this phase, each processor must register its commit information by inserting an info node at the end of the *commit_order_list*. The commit flag of this info node will be initialized to 0, which means this chunk is about to be committed.

2) The commit phase: In this phase, the memory pages updated by this chunk will be committed (i.e., written back to shared memory). Then the processor must set the commit flag of its info node to 1 at the end of this phase, which means it has completed its commit process. Chunks can commit in parallel in this phase, because pages committed by different chunks have no intersection.

3) The synchronization phase: In this phase, this virtual processor is blocked until all the other chunks which start committing prior to the commit point of its preceding chunk have completed their commit. To enforce this, it needs to check all commit flags of those chunk info nodes which are ahead of its own node. If at least one flag is 0, then this processor must be blocked. Otherwise, the processor removes its own info node from the *commit_order_list* and begins executing the next chunk. In practice, this blocking almost never happens, because a virtual processor tends to exit to QEMU to emulate device operations before executing the next chunk, which happens to provide sufficient time for other chunks to complete their commit.

This design noticeably improves performance via reducing the lock granularity. In brief, only the conflict de-

tection and the update broadcast operation are protected by a system-wide lock. Furthermore, It also reduces the time spent on waiting for the lock, because the shorter the time a chunk holds a lock, the lower the probability that other chunks requesting it have to wait is. The most important characteristic is that this protocol can satisfy the serializability requirement because it strictly guarantees that the processor starting to commit a chunk first will execute the subsequent chunk preferentially. The following of this section presents a formal proof on how our decentralized three-phase commit protocol ensures serializability.

Assume for the sake of contradiction that it does not guarantee serializability. Then there exists a set of chunks $C_0, C_1 \ldots C_{n-1}$ which obey our three-phase commit protocol and produce a non-serializable schedule. In order to know whether this chunk schedule is serializable or not, we can draw a precedence graph. This is a graph in which the vertices are the committed chunks and the edges are the dependencies between these committed chunks. A dependence $C_i \rightarrow C_j$ exists only if one of the following is true: 1) C_i executes $Store(X)$ before C_j executes $Load(X)$; 2) C_i executes $Load(X)$ before C_j executes $Store(X)$; 3) C_i executes $Store(X)$ before C_j executes $Store(X)$.

A non-serializable chunk schedule implies a cycle in this graph, and we will prove that our commit protocol cannot produce such a cycle. Assume that a cycle exists in the precedence graph like this: $C_0 \rightarrow C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_{n-1} \rightarrow C_0$, for each chunk C_i, we define T_i to be the time when C_i has been committed, and the corresponding processor begins executing its next chunk C_{i+1}. Then for chunks such that $C_i \rightarrow C_j$, $T_i < T_j$. This is because the *commit_order_list* maintains the total order of these current committing chunks on all processors, and the three-phase commit protocol guarantees that all chunks will be processed in FIFO order. Specifically, The pre-commit phase guarantees that the chunk will be inserted in the *commit_order_list* in execution order, and the synchronization phase guarantees that the chunk will be blocked until all the other chunks which start committing prior to it have completed their commits. Moreover, the conflict detection ensures that an executing chunk will be squashed and re-executed later when there are collisions between it and a newly committed chunk, therefore, will not affect the commit order. Then for this cycle, we have: $T_0 < T_1 < T_2 < \ldots < T_{n-1} < T_0$, which is a manifest contradiction. Hence, our three-phase commit protocol can ensure serializability.

3.4 Replay Memory Access Interleaving

It is relatively simple and efficient to replay memory access interleaving under a chunk-base strategy. Unlike the CREW protocol which must restrict every single memory access to reconstruct the recorded memory access interleaving, we just need to make sure that all chunks will be re-built properly and executed in the original order. In other words, our replay strategy is more coarse-grained.

When we design the replay mechanism of Samsara, a design goal is to maintain the same parallelism as the recoding phase. Since the atomicity and the serializability have already been guaranteed in recording phase, both the conflict detection and the update broadcast operations are no longer required during replay. We just need to ensure that all the preceding chunks have been committed successfully before the current chunk starts. More specifically, during replay, the processors generate chunks according to the order established by the chunk commit log. Then they use the chunk size in that log to determine when they need to truncate these chunks. Here, we use the same approach as above to confirm that a chunk can be truncated at the recorded timestamp. During chunk execution, the COW operation is also required to guarantee that the other concurrently executing chunks will not access the latest data updated by this chunk. To ensure chunk commit in the original order, we will block the commit of a chunk until all the preceding chunks have been committed successfully.

4 Optimizations and Implementation Details

This section describes several optimizations for our chunk-based strategy to improve the recording performance and some implementation details of Samsara.

4.1 Caching Local Copies

In our chunk-based strategy, a copy-on-write (COW) operation will be triggered to create a local copy on the first write to each memory page within a chunk. In the original design, these local copies will be destroyed at the end of this chunk. However, we find that these COW operations can cause a significant amount of performance overhead, especially when recording computation intensive applications.

By analyzing the memory access patterns, we observe that the write accesses of successive chunks exhibit great temporal locality with a history-similar pattern, which means they incline to access roughly the same set of pages. Particularly, when a rollback occurs, the re-executed chunk will follow a similar instruction flow and access the exact same set of pages in most instances.

Based on this observation, we decide to retain local copies at the end of each chunk and use them as a cache of hot pages. By doing so, when a processor modifies a page which already has a copy in the local cache, it

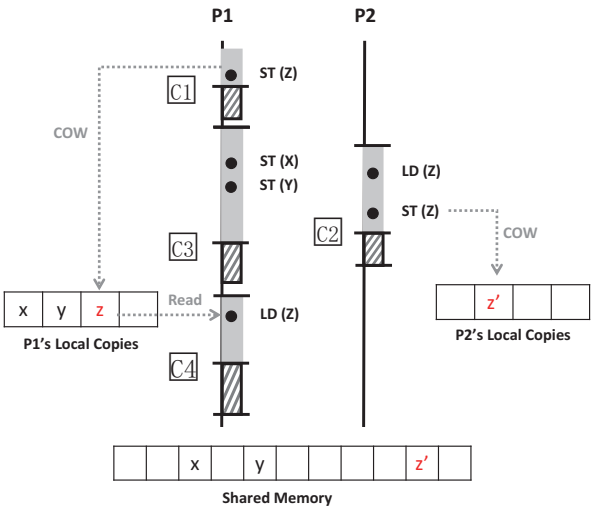

Figure 5: An example of reading outdated data from local copies.

acts just like it does in the unmodified VM with hardware acceleration, and no other operations will be necessary.

However, this design may cause chunks to read outdated data. One example is shown in Figure 5: chunk C4 reads z from its local cache, and meanwhile this page is modified to z' by another committed chunk C2 and copied back to the shared memory. This does not cause any collision, but unfortunately, chunk C4 reads the outdated data z.

These outdated copies can be simply detected by checking the corresponding bit in the conflict bitmap for each local copy. However, the crucial issue remains as how to deal with these outdated copies. We can either update local copies with the latest data in the shared memory or simply discard these outdated copies which have been modified by other committed chunks. These two strategies both have their own advantages and shortcomings: the former reduces the number of COW operations but leads to relatively high overhead due to frequent memory copy operations, while the latter avoids this overhead but still retains some COW operations. We combine the merits of these two strategies as follows: we update outdated copies when a rollback occurs, and discard them when a chunk is committed.

This optimization is essentially equivalent to adding a local cache to buffer the hot pages which are modified by successive chunks. In the current implementation, we limit this cache to a fixed size (0.1% of the main memory size) with a modified LRU replacement policy.

4.2 Adaptive Chunk Size

The chunk size is also a critical factor to the performance of the replay system. If the chunk size is too small, its execution time will not be long enough to amortize the cost of a chunk commit. On the other hand, if the chunk size is too large, the corresponding processor may experience repeated rollbacks due to the increased risk of collision during its commitment, which will eventually cause starvation. Moreover, different applications or even different execution regions in the same application can exhibit varying memory access patterns, which makes it difficult to seek the sweet spot for chunk size, because the optimal size might not be a constant, but rather change during execution. Therefore, one of the major challenges in our implementations is how to adjust chunk size adaptively to achieve a good balance.

Therefore, we propose an adaptive additive-increase/multiplicative-decrease (AIMD) algorithm to adjust the chunk size dynamically during runtime. In this algorithm, a processor will increase its chunk size by a fixed amount after each successful commit to probe for longer execution time. When collision is detected, the processor decreases its chunk size by a multiplicative factor. The idea is similar to the feedback control algorithm in TCP congestion avoidance [10]. The decrease must be multiplicative because it is the only effectual way to ensure that at every step the fairness either increases or stays the same [10]. In a nutshell, when a conflict takes place, this adaptive AIMD algorithm ensures that all processors will quickly converge to use equal chunk size. Therefore, each chunk has same probability to be committed or squashed. One limitation of this algorithm is that it is less effective for some I/O intensive workloads due to the frequent chunk truncations caused by the large number of concurrent I/O requests.

4.3 Double Buffering

In our decentralized commit protocol, the conflict detection and the update broadcast operation are both protected by a system-wide lock to enforce the serialization requirement. Since the conflict bitmap will be modified by other chunks due to the update broadcast operation, while being read by its own chunk for the conflict detection, it has to acquire this system-wide lock whenever one chunk try to set the corresponding bits in other chunks' conflict bitmap to broadcast its updates. Similarly, it has to wait for this lock to be released whenever its own chunk needs to read this bitmap for conflict detection.

Double buffering mitigates this problem and can further increase parallelism. Instead of using a single

bitmap, we use two bitmaps simultaneously to implement double buffering. One of them serves as a write bitmap and the other as the read bitmap. Both bitmaps can be accessed at any time. By doing so, we avoid locking the bitmap while reading and writing to it. We switch these two bitmaps when the succeeding chunk starts its conflict detection, so it can read the bitmap directly when other chunks are free to set this bitmap simultaneously. In our design, only this switch operation is protected by a lock, and neither bitmap requires any locking at all.

5 Evaluation

This section discusses our evaluation of Samsara. We first illustrate the experimental setup and our workloads. Then we evaluate different aspects of Samsara and compare it with a CREW approach.

5.1 Experimental Setup

All the experiments are conducted on a Dell Precision T1700 Workstation with a 4-core Intel Core i7-4790 processor (running at 3.6GHz, with 256KB L1, 1MB private L2 and 8MB shared L3 cache) running Ubuntu 12.04 with Linux kernel version 3.11.0 and QEMU-1.2.2. The host machine has 12GB memory. The Guest OS is an Ubuntu 14.04 with Linux kernel version 3.13.1.

5.2 Workloads

To evaluate our system on a wide range of applications, we choose two sets of benchmarks that represent very different characteristics, including both computation intensive and I/O intensive applications.

The first set includes eight computation intensive applications chosen from PARSEC and SPLASH-2 benchmark suites (four from each): blackscholes, bodytrack, raytrace, and swaptions form PARSEC [5]; radiosity, water_nsquared, water_spatial, and barnes from SPLASH-2 [36]. We choose both PARSEC and SPLASH-2 suites because each of them has its own merits, and no single benchmark can represent the characteristics of all types of applications. PARSEC is a well-studied benchmark suite composed of emerging multithreaded programs from a broad range of application domains. In contrast, SPLASH-2 is composed mainly of high-performance computing programs which are commonly used for scientific computation on distributed shared-address-space multiprocessors. These eight applications come from different areas of computing and are chosen because they exhibit diverse characteristics and represent the different worst-case applications due to the burdensome shared memory accesses.

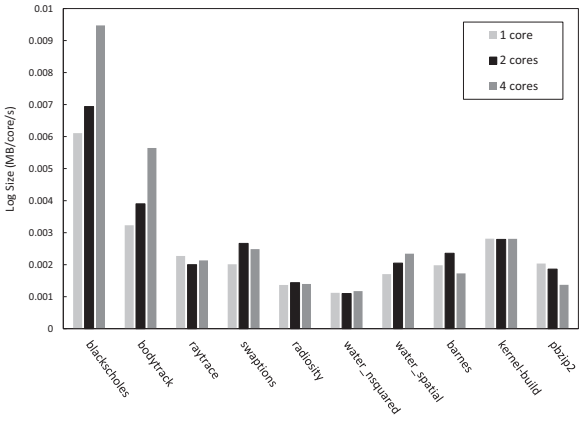

Figure 6: Log size produced by Samsara during recording (compressed with gzip).

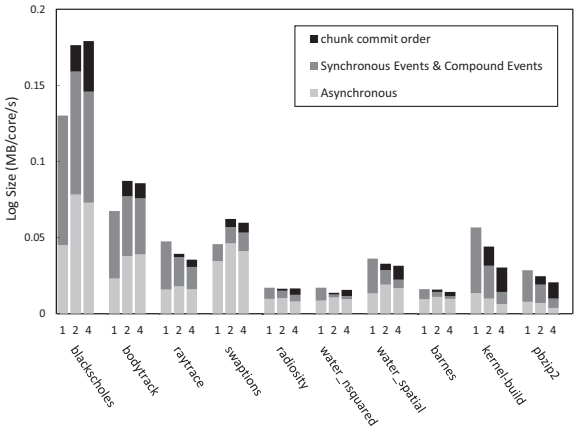

Figure 7: The proportion of each type of non-deterministic events in a log file (without compression).

Although there are applications in the first set that perform certain amount of I/O operations, most of them are disk read only. In the other set of benchmarks, we select two more I/O intensive applications (kernel-build and pbzip2) to further evaluate how well Samsara handle I/O operations. Kernel-build is a parallel build of the Linux kernel version 3.13.1 with the default configuration. In order to achieve maximum degree of parallelism we use the *-j* option of make. Usually, *make -j n+1* produces a relatively high performance on a VM with *n* virtual processors. This is because the extra process makes it possible to fully utilize the processors during network delays and general I/O accesses such as loading and saving files to disk [13]. Pbzip2 is a parallel file compressor which uses pthreads. We use pbzip2 to decompress a 111MB Linux-kernel source file.

5.3 Log Size

Log size is an important consideration of the replay systems. Usually, recording non-deterministic events will generate huge space overhead which limits the duration of the recording. The log size of some prior works is approximately 2 MB/1GHz-processor/s [38]. Some can support only a few seconds' recording which is difficult to satisfy long-term recording needs [38].

Experiment results show that Samsara produces a much smaller log size which is orders of magnitude smaller than the ones reported by prior work in software-based schemes, and even smaller than some reported in hardware-based schemes. Figure 6 shows the compressed log sizes generated by each core for all the applications. The experiments indicate that Samsara generates logs at an average rate of 0.0027 MB/core/s and 0.0031 MB/core/s for recording two and four cores, respectively. For comparison, the average log size with a single core, which does not need to record memory interleaving, is 0.0024 MB/s.

To compare the log size of Samsara and the previous software or hardware approaches, this experiment was designed to be as similar as possible to the ones in the previous papers. SMP-ReVirt generates logs at an average rate of 0.18MB/core/s when recording the workloads in SPLASH-2 and kernel-build on two dual-core Xeons [13]. DeLorean generates logs at an average rate of 0.03MB/core/s when recording the workloads in SPLASH-2 on eight simulated processors [21].

We achieve a significant reduction in the log size because the size of the chunk commit log is practically negligible compared with other non-deterministic events. Figure 7 illustrates the proportions of each type of non-deterministic events in each log file. In most workloads, the interleaving log represents a small fraction of the whole log (approximately 9.36% with 2 cores and 19.31% with 4 cores). For the I/O intensive applications, this proportion is higher, because the large number of concurrent I/O requests leads to more chunk truncations.

Another reason is we avoid recording all disk reads. In Samsara, we use QEMU's qcow2 (QEMU Copy On Write) disk format to create a write protected base image and an overlay image on top of it to perform disk modifications during recording and replay. By doing so, we can present the same disk view for replay without logging any disk reads or creating another copy of the whole disk image.

In summary, the use of chunk-based strategy makes it possible to significantly reduces the log file size by 98.6% compared to the previous software-only schemes. The log size in our system is even smaller than the ones reported in hardware-based solutions, since we can further reduce the log size via increasing the chunk size which is impossible in hardware-based approaches due to the risk of cache overflow [21].

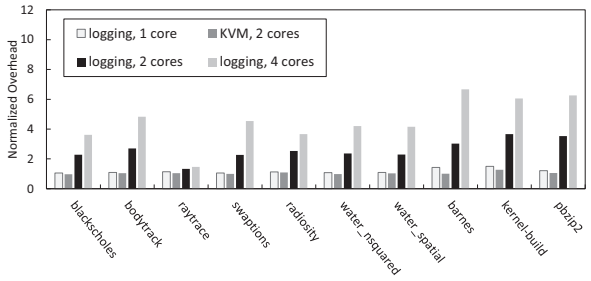

Figure 8: Recording overhead compared to the native execution.

5.4 Performance Overhead Compared to Native Execution

The performance overhead of a system can be evaluated in different ways. One way is to measure the overhead of the system relative to the base platform (e.g., KVM) it runs on. The problem with this approach is that the performance of different platforms can vary significantly and hence the overhead measured in this manner does not reflect the actual execution time of the system in real life. Consequently, we decide to compare the performance of our system to native execution, as shown in Figure 8.

As shown in the figure, the average performance overhead introduced by Samsara is 2.3× for recording computation intensive applications on two cores, and 4.1× on four cores. For I/O intensive applications, the overhead is 3.5× on two cores and 6.1× on four cores. This overhead is much smaller than the ones reported by prior works in software-only schemes, which cause about 16× or even 80× overhead when recording similar workloads on two or four cores [8, 27]. Samsara improves the recording performance dramatically because we avoid all memory access detections which are a major source of the overhead. Further experiment reveals that only 0.83% of the whole execution time is spent on handling page fault VM exits in Samsara, while prior CREW approaches suffer from more than 60% execution time spent on handling page fault VM exits.

Among the computation intensive workloads, barnes has a relatively high overhead (more than 3× on two cores), while retrace has a negligible overhead (about 0.3× on two cores). After analyzing the shared memory access pattern of these two workloads, we find that retrace contains many more read operations than write. Since Samsara does not trace any read accesses, these read operations do not cause any performance overhead. In contrast, barnes contains a lot of shared mem-

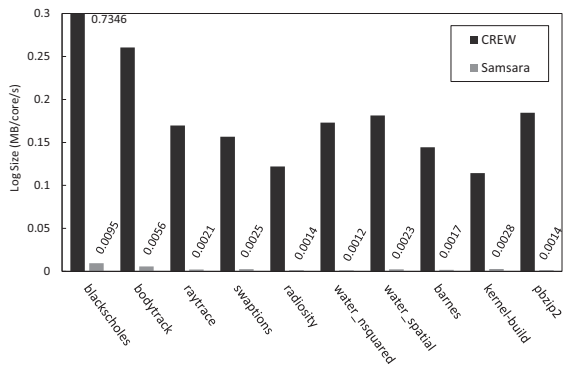

Figure 9: A comparison of the log file size between Samsara and CREW (4 cores, compressed with gzip).

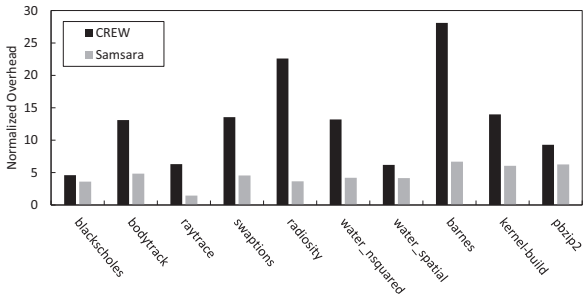

Figure 10: A comparison of recording overhead between Samsara and CREW (4 cores).

ory writes, and the unstructured communication pattern negates the effects of our hot page cache. Moreover, our page-level conflict detection may cause false conflicts (i.e., false sharing in SMP-ReVirt), which may lead to unnecessary rollback and increase performance overhead. When compared to computation intensive workloads, I/O intensive workloads incur relatively high overhead. This is also caused by the large number of concurrent I/O requests, which keep the chunk size quite small. Therefore, the execution time is not long enough to amortize the cost of the chunk commits in these workloads.

5.5 A Comparison with Prior software Approaches

To further evaluate our chunk-based strategy in Samsara against prior software-only approaches, we implement the original CREW protocol [13] in our testbed.

Log Size: Figure 9 shows the comparison against CREW protocol in log file size, in which Samsara reduces the log file size by 98.6% (i.e., from 0.22MB/core/s to 0.003MB/core/s). To understand the

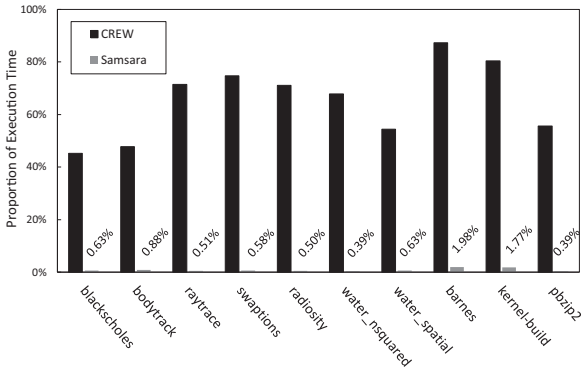

Figure 11: Proportion of the execution time consumed on handling page fault VM exits (4 cores).

improvement that Samsara achieves, we measure the proportions of each type of non-deterministic events in the log file. In this measurement, we find that nearly 99% of the events are memory access interleaving in CREW protocol, while only 10% of the events in Samsara are chunk commit orders.

Performance Overhead: We also compare the performance overhead of Samsara and the CREW protocol. The results in figure 10 illustrate that with four cores Samara reduces the overhead by up to 76.8% and the average performance improvement is 58.6% compared to the native execution.

Time Consumed on Handling Page Fault VM Exits: To understand why Samsara improves the recording performance so dramatically, we evaluate the time consumed on handling page fault VM exits in both approaches, since it is one of the primary contributors to the performance overhead. Figure 11 shows that 65.6% of the whole execution time is spent on handling page fault VM exits in the CREW protocol. In contrast, this proportion is only 0.83% in Samsara due to the HAV and chunk-based strategy we used.

5.6 Benefits of Performance Optimizations

Benefits of Caching Local Copies: Experimental results show that the average performance benefits contributed by caching local copies are 15.2% for recording computation intensive applications on four cores. For I/O intensive applications, the benefits increase to 22.1%. The effect of this optimization is highly dependent on the amount of temporal locality the local cache can exploit and the frequency of write operations. This explains why, for the applications, like water_nsquared and water_spatial, which perform few write operations and exhibit poor temporal locality, the benefits of this optimization are less (-1.24% and 2.76%).

Benefits of Double Buffering: The performance benefits contributed by double buffering are less significant. Empirically, the average performance increase is 4.61% when recording computation intensive applications on four cores. For I/O intensive applications, the improvement is 7.43%.

The improvement of the adaptive chunk size optimization is constrained by the frequent chunk truncations caused by I/O requests, thus is heavily application-specific. Among all the applications we experiment with, only a small subset of the computation intensive applications (e.g., raytrace from PARSEC, radiosity from SPLASH-2) is shown to have statistically significant benefit from this optimization.

6 Related Work

Deterministic Replay in Virtualization Environment: The idea of achieving deterministic replay based on virtualization environment was first proposed by Bressoud, et al. [6]. Similarly, ReVirt [12] can replay entire OSes deterministically and efficiently by recording all non-deterministic events within the VMM. ReTrace [40] is a trace collection tool based on the deterministic replay of the VMware hypervisor. However, both of them only work for uniprocessors and cannot be applied to multiprocessor environment. SMP-Revirt [13] is the first deterministic replay system that records and replays a multiprocessor VM on commodity hardware by leveraging CREW protocol. ReEmu [8] refines the CREW protocol with a seqlock-like design to achieve scalable deterministic replay in a parallel full-system emulator. While these virtualization-based schemes are flexible, extensible, and user-friendly, they suffer serious performance degradation and generate huge logs. In contrast, Samsara can leverage the latest HAV extensions in commodity multiprocessors to achieve efficient and practical deterministic replay. A preliminary description of this work was in [31].

Hardware-based Deterministic Replay: Hardware-based deterministic replay uses special hardware support for recording memory access interleaving. These schemes require modifications to the existing hardware, which increases the complexity of the circuits. FDR [38] records interleaving between pairs of instructions, and it improves the performance by implementing the Netzer's Transitive Reduction optimization [24] on hardware. RTR [39] extended FDR by only recording the logical time orders between memory access instructions. However, they still generate huge space overhead, which limits the duration of the recording. Strata [22] redesigns the recording strategy and records a stratum when a dependence occurs. Each stratum contains many memory operations issued by the corresponding processor since the last stratum is logged. Delorean [21] goes even further on this idea. Rather than logging individual dependence, it records memory access interleaving as series of chunks. By doing so, it allows out-of-order execution of instructions. IMMR [28] designs a chunk-based strategy for memory race recording in modern chip multiprocessors. Rerun [16] introduces an intermediate approach where it traces each data access but does not record this dependence. Instead, it records the number of instructions between two dependences. However, Rerun does not scale well during replay. To improve replay performance, Karma [3] is proposed as a chunk-based approach that aims to increase replay parallelism. Compared to chunk-based strategies in hardware schemes, Samsara improves the recording performance in VMM without requiring any hardware modification. Firstly, by leveraging HAV extensions, we avoid tracing every single memory access, instead perform a EPT traversal to obtain the read and write set. Secondly, we remove the centralized arbiter in Delorean, and propose a decentralized three-phase commit protocol to perform chunk commit efficiently.

7 Conclusion

In this paper, we have made the first attempt to leverage HAV extensions to achieve an efficient and practical software-based deterministic replay system on commodity multiprocessors. Unlike prior software schemes that trace every single memory access to record interleaving, we leverage the HAV extensions to track the read and write-set, and implement a chunk-based recording scheme in software. By doing so, we avoid all memory access detections, which are a major source of overhead in the prior work. In addition, we propose a decentralized three-phase commit protocol which significantly reduces the performance overhead by allowing chunk commits in parallel while still ensuring serializability. By evaluating our system on real systems, we demonstrate that Samsara can reduce the recording overhead from 10× to 2.3× and reduce the log file size to 1/70th on average.

Acknowledgments

The authors would like to thank Jon Howell, our shepherd Andreas Haeberlen, and the anonymous reviewers for their insightful comments. We also thank Yunqi Zhang for his valuable feedback on the earlier drafts of this paper. This work was supported by the National Natural Science Foundation of China (Grant No. 61170056), the National Grand Fundamental Research 973 Program of China (Grant No. 2014CB340405).

References

[1] AGRAWAL, H., DE MILLO, R., AND SPAFFORD, E. An execution-backtracking approach to debugging. *Software, IEEE 8*, 3 (1991), 21–26.

[2] ALTEKAR, G., AND STOICA, I. Odr: Output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (2009), pp. 193–206.

[3] BASU, A., BOBBA, J., AND HILL, M. D. Karma: Scalable deterministic record-replay. In *Proceedings of the International Conference on Supercomputing* (2011), ICS '11, pp. 359–368.

[4] BHANSALI, S., CHEN, W.-K., DE JONG, S., EDWARDS, A., MURRAY, R., DRINIĆ, M., MIHOČKA, D., AND CHAU, J. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments* (2006), VEE '06, pp. 154–163.

[5] BIENIA, C. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[6] BRESSOUD, T. C., AND SCHNEIDER, F. B. Hypervisor-based fault tolerance. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (1995), SOSP '95, pp. 1–11.

[7] CHEN, A., MOORE, W. B., XIAO, H., HAEBERLEN, A., PHAN, L. T. X., SHERR, M., AND ZHOU, W. Detecting covert timing channels with time-deterministic replay. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), pp. 541–554.

[8] CHEN, Y., AND CHEN, H. Scalable deterministic replay in a parallel full-system emulator. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2013), pp. 207–218.

[9] CHEN, Y., HU, W., CHEN, T., AND WU, R. Lreplay: A pending period based deterministic replay scheme. In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (2010), ISCA '10, pp. 187–197.

[10] DAH-MING, C., AND RAJ, J. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN systems 17*, 1 (1989), 1–14.

[11] DEVECSERY, D., CHOW, M., DOU, X., FLINN, J., AND CHEN, P. M. Eidetic systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), pp. 525–540.

[12] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation* (2002), pp. 211–224.

[13] DUNLAP, G. W., LUCCHETTI, D. G., FETTERMAN, M. A., AND CHEN, P. M. Execution replay of multiprocessor virtual machines. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (2008), pp. 121–130.

[14] HONARMAND, N., DAUTENHAHN, N., TORRELLAS, J., KING, S. T., POKAM, G., AND PEREIRA, C. Cyrus: Unintrusive application-level record-replay for replay parallelism. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2013), ASPLOS '13, pp. 193–206.

[15] HONARMAND, N., AND TORRELLAS, J. Relaxreplay: Record and replay for relaxed-consistency multiprocessors. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (2014), pp. 223–238.

[16] HOWER, D. R., AND HILL, M. D. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the 35th Annual International Symposium on Computer Architecture* (2008), ISCA '08, pp. 265–276.

[17] JOSHI, A., KING, S. T., DUNLAP, G. W., AND CHEN, P. M. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles* (2005), SOSP '05, pp. 91–104.

[18] KING, S. T., AND CHEN, P. M. Backtracking intrusions. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (2003), SOSP '03, pp. 223–236.

[19] LEBLANC, T., AND MELLOR-CRUMMEY, J. Debugging parallel programs with instant replay. *Computers, IEEE Transactions on C-36*, 4 (April 1987), 471–482.

[20] LEE, D., WESTER, B., VEERARAGHAVAN, K., NARAYANASAMY, S., CHEN, P. M., AND FLINN, J. Respec: Efficient online multiprocessor replayvia speculation and external determinism. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems* (2010), pp. 77–90.

[21] MONTESINOS, P., CEZE, L., AND TORRELLAS, J. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the International Symposium on Computer Architecture* (2008), pp. 289–300.

[22] NARAYANASAMY, S., PEREIRA, C., AND CALDER, B. Recording shared memory dependencies using strata. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2006), pp. 229–240.

[23] NARAYANASAMY, S., POKAM, G., AND CALDER, B. Bugnet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture* (2005), ISCA '05, pp. 284–295.

[24] NETZER, R. H., AND XU, J. Adaptive message logging for incremental program replay. *IEEE Concurrency 1*, 4 (1993), 32–39.

[25] OLSZEWSKI, M., ANSEL, J., AND AMARASINGHE, S. Kendo: Efficient deterministic multithreading in software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (2009), pp. 97–108.

[26] PARK, S., ZHOU, Y., XIONG, W., YIN, Z., KAUSHIK, R., LEE, K. H., AND LU, S. Pres: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (2009), pp. 177–192.

[27] PATIL, H., PEREIRA, C., STALLCUP, M., LUECK, G., AND COWNIE, J. 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* (2010), pp. 2–11.

[28] POKAM, G., PEREIRA, C., DANNE, K., KASSA, R., AND ADL-TABATABAI, A.-R. Architecting a chunk-based memory race recorder in modern cmps. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture* (2009), MICRO 42, pp. 576–585.

[29] QIAN, X., HUANG, H., SAHELICES, B., AND QIAN, D. Rainbow: Efficient memory dependence recording with high replay parallelism for relaxed memory model. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on* (Feb 2013), pp. 554–565.

[30] QIAN, X., SAHELICES, B., AND QIAN, D. Pacifier: Record and replay for relaxed-consistency multiprocessors with distributed directory protocol. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (2014), ISCA '14, pp. 433–444.

[31] REN, S., LI, C., TAN, L., AND XIAO, Z. Samsara: Efficient deterministic replay with hardware virtualization extensions. In *Proceedings of the 6th Asia-Pacific Workshop on Systems* (2015), APSys '15, pp. 9:1–9:7.

[32] SCALES, D. J., NELSON, M., AND VENKITACHALAM, G. The design of a practical system for fault-tolerant virtual machines. *SIGOPS Oper. Syst. Rev. 44*, 4 (Dec. 2010), 30–39.

[33] SRINIVASAN, S. M., KANDULA, S., ANDREWS, C. R., AND ZHOU, Y. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference, General Track* (2004), pp. 29–44.

[34] VEERARAGHAVAN, K., LEE, D., WESTER, B., OUYANG, J., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. Doubleplay: Parallelizing sequential logging and replay. *ACM Trans. Comput. Syst. 30*, 1 (Feb. 2012), 3:1–3:24.

[35] VOSKUILEN, G., AHMAD, F., AND VIJAYKUMAR, T. N. Timetraveler: Exploiting acyclic races for optimizing memory race recording. In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (2010), ISCA '10, pp. 198–209.

[36] WOO, S. C., OHARA, M., TORRIE, E., SINGH, J. P., AND GUPTA, A. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture* (1995), ISCA '95, pp. 24–36.

[37] WU, X., AND MUELLER, F. Elastic and scalable tracing and accurate replay of non-deterministic events. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing* (2013), ICS '13, pp. 59–68.

[38] XU, M., BODIK, R., AND HILL, M. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the International Symposium on Computer Architecture* (2003), pp. 122–133.

[39] XU, M., HILL, M. D., AND BODIK, R. A regulated transitive reduction (rtr) for longer memory race recording. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (2006), ASPLOS XII, pp. 49–60.

[40] XU, M., MALYUGIN, V., SHELDON, J., VENKITACHALAM, G., AND WEISSMAN, B. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the Third Annual Workshop on Modeling, Benchmarking and Simulation* (2007).

[41] YANG, Z., YANG, M., XU, L., CHEN, H., AND ZANG, B. Order: Object centric deterministic replay for java. In *USENIX Annual Technical Conference* (2011).

[42] ZHU, J., JIANG, Z., AND XIAO, Z. Twinkle: A fast resource provisioning mechanism for internet services. In *Proceedings of the IEEE INFOCOM* (2011), pp. 802–810.

[43] ZHU, J., JIANG, Z., XIAO, Z., AND LI, X. Optimizing the performance of virtual machine synchronization for fault tolerance. *IEEE Transactions on Computers 60*, 12 (Dec 2011), 1718–1729.