# Leveraging Hardware-Assisted Virtualization for Deterministic Replay on Commodity Multi-Core Processors

Shiru Ren, Le Tan, Chunqi Li, Zhen Xiao, *Senior Member, IEEE*, and Weijia Song

**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. In contrast, software-only schemes work on commodity hardware, but suffer from excessive performance overhead and huge logs. In this article, we present the design and implementation of a novel system, Samsara, which uses the hardware-assisted virtualization (HAV) extensions to achieve efficient deterministic replay without requiring any hardware modification. Unlike prior software schemes which trace every single memory access to record interleaving, Samsara leverages HAV 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. 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 $10\times$, reported by state-of-the-art works, to $2.1\times$ on average.

**Index Terms**—Deterministic replay, virtualization, multi-core

## 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 has been 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], [2]. For security analysis, it can help the system administrator analyze the intrusions and investigate whether a specific vulnerability was exploited in a previous execution [3], [4], [5], [6]. For fault-tolerance, it provides the ability to replicate the computation on processors for building the hot-standby system or data recovery [7], [8], [9].

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 [10], [11], [12], [13], [14], [15], [16]. 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 operating system, the compiler, the runtime libraries or the virtual machine manager (VMM) [2], [17], [18], [19], [20]. 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 [18], [19], [21]. While these schemes are flexible, extensible, and user-friendly, they suffer serious performance overhead (about $10\times$ compared to the native execution with two processors) 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. Further experiment


- *S. Ren, L. Tan, C. Li, and Z. Xiao are with the Department of Computer Science, Peking University, Beijing 100871, China. E-mail: {rsr, tanle, lcq, xiaozhen}@net.pku.edu.cn.*
- *W. Song is with the Department of Computer Science, Cornell University, Ithaca, NY 14853. E-mail: ws393@cornell.edu.*


*Manuscript received 28 Oct. 2016; revised 30 June 2017; accepted 5 July 2017. Date of publication 16 July 2017; date of current version 19 Dec. 2017. (Corresponding author: Zhen Xiao.) Recommended for acceptance by G. Heiser. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TC.2017.2727492*

0018-9340 © 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

reveals that approximately 60 percent of the whole execution time is spent on handling page fault VM exits. By contrast, it takes only less than 0.1 percent in original VMM.

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 advantage of advanced hardware features.

In this article, the main goal is to implement a software approach that can take full advantage 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. Therefore, in our design, what we need to record are just the chunk sizes and commit orders which are practically negligible compared with other non-deterministic events.

Moreover, HAV provides a transparent and more efficient full-system virtualization platform after years of improvement. With deterministic replay extension, this platform is promising to be the most practical solution for applications like cyclic debugging, malware analysis, and intrusion detection. This poses a new opportunity to extensively expand the applicability and practicability of deterministic replay.

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. Through moving most time-consuming operations out of the synchronized block, we reduce the lock contention and improve scalability and performance. We also present a formal proof on how our decentralized three-phase commit protocol ensures 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 0.22 MB/core/second to 0.003 MB/core/second) and reduces the recording overhead from about $10\times$ to $2.1\times$ compared to the native execution.

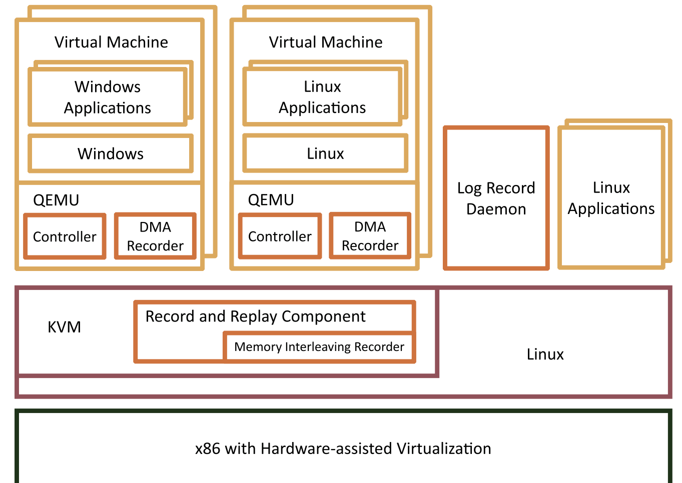



Fig. 1. Architecture overview.

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. We also introduce several optimizations for our chunk-based strategy to improve the performance.

The rest of the article 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. Sections 4 and 5 discuss several implementation issues as well as some optimizations critical to performance but not covered in previous works. We evaluate Samsara in Section 6. Section 7 reviews related work and Section 8 concludes the article.

## 2 System Overview

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

### 2.1 System Architecture

Samsara implements the deterministic replay in original VMM, which has access to the entire virtual machine and can take full advantage of the HAV extensions, as illustrated in Fig. 1. The architecture of Samsara consists of four principal components, namely, the Controller, the record and replay component, the Direct Memory Access (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 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 need to 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 [22]. 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 [22]. 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. During replay we need to guarantee the DMA transfer is done prior to the interrupt injection. Moreover, the order of memory accesses from the DMA devices and the virtual processors may interleave non-deterministically during replay. Hence, we treat a DMA device as a virtual processor with the highest priority.

*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 how Samsara uses HAV extensions to overcome this challenge.

### 3.1 Chunk-Based Strategy

Previous software-only schemes leverage CREW protocol to serialize and log the total order of the memory access interleaving [17], 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 [12]. 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. As with the database transaction, chunk execution must satisfy the atomicity and serializability requirements. Atomicity requires that the

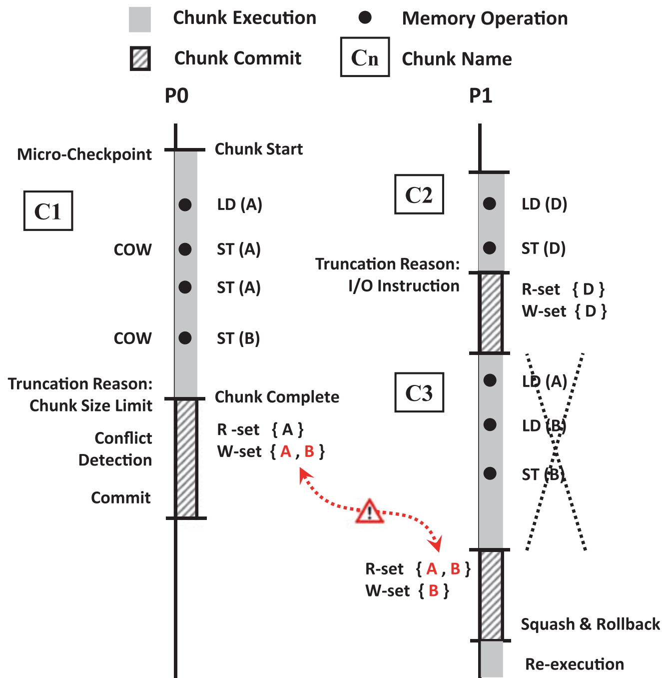



Fig. 2. The execution flow of our chunk-based approach.

execution of each chunk must be "all or nothing", therefore, prevents partial occurrence of updates. Serializability requires that the concurrent execution of chunks have to result in the same system state as if these chunks were executed serially. Namely, we must guarantee that all virtual processors will observe the same chunk commit order simultaneously, and the effects of an incomplete chunk might not even be visible to other chunks. Serializability is the highest level of isolation, and the major correctness criterion for chunk execution [23]. Under these requirements, chunk appears to the system as a single memory access because the interleaving between memory accesses of the different processors occurs only at chunk boundaries [12]. Therefore, executing chunk atomically and in serializable isolation level in a deterministic total order can properly reconstruct the memory access interleaving.

To enforce serializability, first, 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.

Fig. 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 conflict 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 Fig. 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 Fig. 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 our 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 [24]. 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.

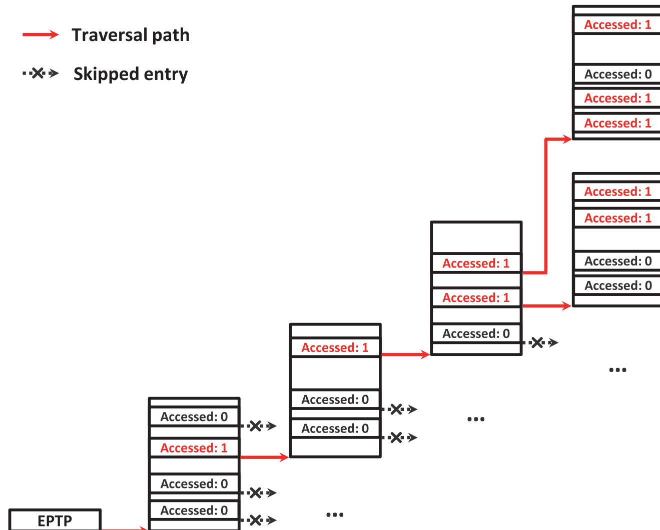



Fig. 3. The partial traversal of EPT.

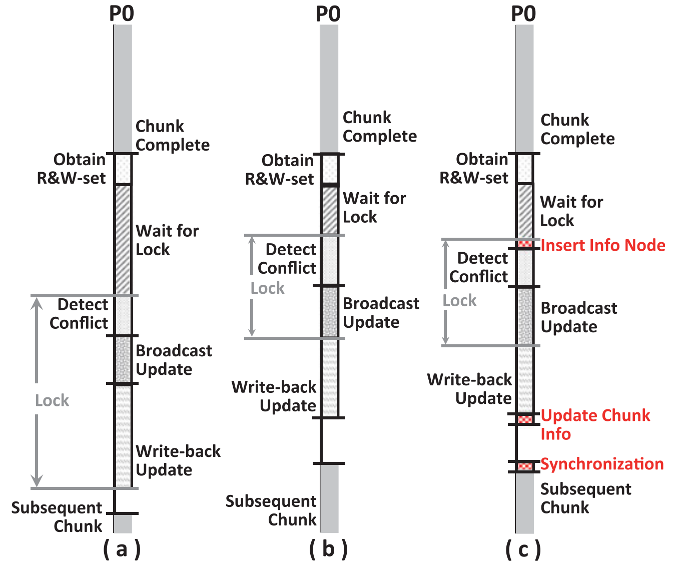



Fig. 4. General design of decentralized three-phase commit protocol: a) chunk timeline of a naïve design, b) moving update write-back operation out of the synchronized block, and c) a design of decentralized three-phase commit protocol.

Therefore, by utilizing these hardware features, we can obtain the R&W-set by simply gathering all leaf entries where the accessed or the dirty flag is set, which can be archived by an EPT traversal. Specifically, we maintain a dedicated EPT alone with an access bitmap and a dirty bitmap for each virtual processor, so that intermediate state of a chunk is invisible to other chunks. When a chunk completes execution, the corresponding virtual processor will first traverse its own EPT until it finds all entries where the accessed or the dirty flag is set, and then updates its access bitmap and dirty bitmap accordingly. This design avoids detecting all memory accesses, which is the primary overhead in prior works, and instead obtains each chunk's read-set and write-set by retrieving the accessed and the dirty flags of the EPT.

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. An example is given in Fig. 3. In the first level of the EPT, there is only one table entry whose accessed flag is set to 1. Therefore, we just traverse the subtree pointed by this entry. Thanks to locality, the access locations of most chunks are adjacent, which means the 1 bits are often clustered together, while most of the rest bits are 0. In most cases, we only traverse a tiny part of the EPT with negligible overhead.

### 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 [12]. 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 Fig. 4a. 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 during 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 percent 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 Fig. 4b. This not only reduces the cost of the locking operation substantially, but also increases parallelism because multiple chunks can now commit concurrently.

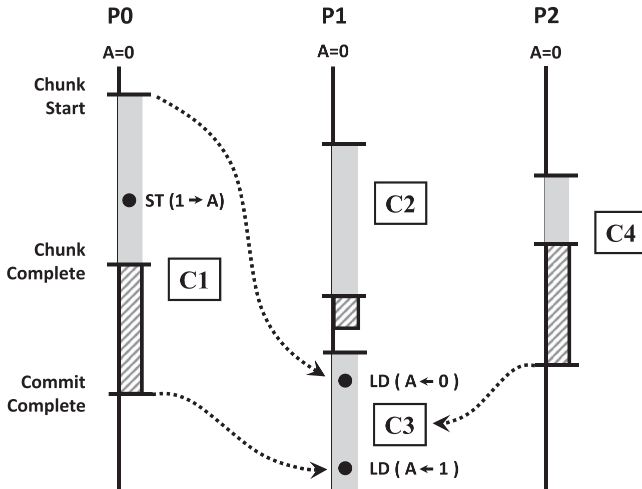



Fig. 5. An example of out-of-order commit.

However, one side effect of this design is that chunks may get committed out-of-order, thereby violating serializability. One example is shown in Fig. 5. 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 Fig. 4c:

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 detection 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 this design 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: $C_0 \rightarrow C_1 \rightarrow C_2 \rightarrow \cdots \rightarrow C_{n-1} \rightarrow C_0$, 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: $T_0 < T_1 < T_2 < \cdots < T_{n-1} < T_0$, which is a manifest contradiction. Hence, our three-phase commit protocol can ensure serializability.

TABLE 1
The Optimized Bitmap Operations in Our Revised Design

| Bitmap operations | Descriptions |
|---|---|
| *bitmap_clear ()* | Traverses all nodes through the linked list and uses them to quickly locate and clear all bits in the bitmap. |
| *bitmap_or (dst, src)* | Traverses all nodes through the linked list of *src* and uses them to perform the bitwise OR operation with the bitmap of *des*. Then inserts these nodes into the corresponding linked list of *des*. |
| *bitmap_intersects (src1, src2)* | Traverses all nodes through the linked list of *src1* and, for each of them, checks whether the corresponding bit in the bitmap of *src2* is set. |
| *test_bit ()* | Checks the bitmap to determine whether a bit is set. |
| *set_bit ()* | Sets a bit in bitmap and inserts a node into the corresponding linked list. |

### 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 Implementation

This section describes the implementation details of our chunk-based strategy as well as several performance-critical design choices.

### 4.1 Accelerating Bitmap Operations

Recall that each virtual processor maintains three bitmaps to mark the R&W-set and perform the conflict detection. We observe that these bitmaps are overly sparse and exhibit significant spatial locality, which means the 1 bits are often clustered together, while most of the rest bits are 0. Therefore, the bitmap operations that involve a traversal of the whole bitmap (e.g., *bitmap_or*, *bitmap_clear*, and *bitmap_intersects*) are expensive.

The above observation provides the insight to use an auxiliary data structure to accelerate these operations. In our revised design, we combine each bitmap with a linked list. Each node of the linked list represents a 1 bit in its associated bitmap, and contains the same Guest Frame Number (GFN) of the memory page which this bit represents. This simplifies most bitmap operations as shown in Table 1. In general, the linked list is used to traverse all 1 bits efficiently, while the bitmap is useful for retrieving a bit.

This optimization is essentially trading space for time, which can significantly reduce the time consumed by bitmap operations while only increasing space slightly. Although there are other data structures such as segment tree which also match the access patterns we observed, using both a bitmap and a linked list is probably the simplest and most efficient solution, since we do not need to support arbitrary deletion.

### 4.2 Starvation Avoidance

As with the other systems which execute instructions grouped into chunks [25], Samsara also suffers from starvation. In some special scenarios, a virtual processor may be unable to make progress when a chunk is being repeatedly squashed. Suppose a chunk C1 reads an array in loop. At the same time, this array is being written by all other executing chunks simultaneously. The read-set of C1 includes all elements of this array, but each write-set of other chunks only includes a subset of this array. Hence, if any of the other chunks gets committed (which is a high probability event), C1 must be squashed and the corresponding virtual processor rollbacks repeatedly without making progress.

Samsara introduces two mechanisms to avoid starvation. When rollback is detected, the virtual processor decreases its chunk size by a multiplicative factor, therefore, significantly increasing the probability of committing success. We call this mechanism multiplicative-decrease, because the idea is similar to the feedback control algorithm in TCP congestion avoidance [26]. However, we restore the chunk size immediately once a chunk commits. Otherwise, if the chunk size is too small, the execution time will not be long enough to amortize the cost of a chunk commit.

Although the multiplicative-decrease design reduces rollbacks, it does not guarantee a chunk to be committed. If a virtual processor cannot make progress after several iterations of multiplicative-decrease, Samsara makes it switch to another execution mode, which we call protected-commit. As illustrated in Fig. 6, P0 switches into protected-commit mode for chunk C5, it first broadcasts the R&W-set of its last squashed chunk C1 to other concurrently executing chunks. The reason for broadcasting the previous R&W-set is that the re-executed chunk will follow a same instruction flow in most instances, which means that the R&W-set of re-executed chunk C5 is the same as that of C1. Therefore, any of the other executing chunks which is in conflict with C5 (as C6 and C7 in this figure) will wait until C5 commit, but all of the other irrelevant chunks will not be affected (as C4 in this figure).

The practice indicates that these two mechanisms allow chunks to commit successfully in most cases while still

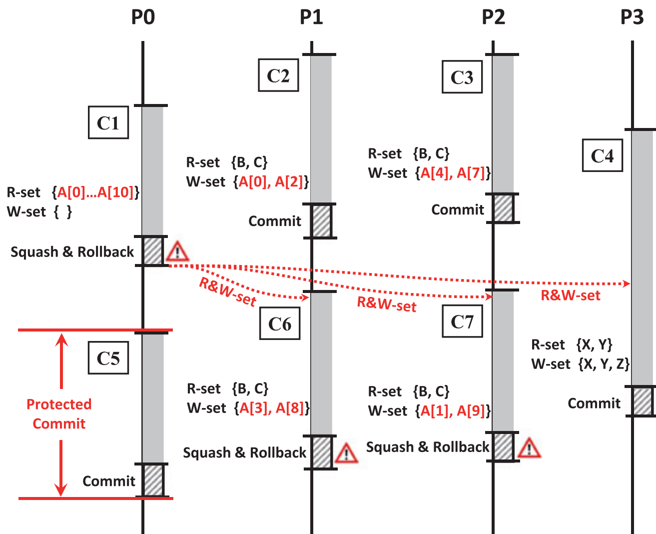


Fig. 6. The protected-commit execution mode.

retaining sufficient parallelism. However, in rare cases the memory pages updated by other newly committed chunks (as C2 or C3 in Fig. 6) may affect the instruction flow of re-executed chunk C5 and result in a conflict between C5 and other chunks. If the protected-commit cannot guarantee forward progress, Samsara will prohibit other chunks from committing until this chunk commit.

### 4.3 Early Rollback

Samsara leverages the lazy conflict detection which defers the detection until chunk completion to provide better throughput. Lazy conflict detection can expose more concurrency than the eager detection which resolves conflicts at the access time [27]. Moreover, the delayed conflict detection decreases the probability of a livelock [27]. However, postponing the conflict detection until commit-time may lead to wasted work by conflicting chunks, since a chunk may continue running even though other virtual processors have committed conflicting updates. In practice, once a conflicting update is committed, the affected chunks are doomed to abort and will be wasting not only its work performed so far but also the work which will be done until commit. This insight suggests we should abort a chunk as early as possible to avoid further waste.

We implement an early chunk rollback strategy as an additional mechanism to alleviate the risks of wasted work associated with the lazy conflict detection. During chunk execution, if a write operation triggers a VM exit and traps to VMM, we check whether the corresponding bit of this page in the conflict bitmap is set. If the result is positive, which means this chunk modifies a page which has already been updated by other committed chunks, thus, is doomed to abort. In this case, instead of waiting until the rollback occurs, we rollback this chunk immediately. By allowing chunks to perform early rollback, we significantly reduce the extent of rollback, and therefore, uncover more parallelism.

### 4.4 Chunk Truncation

In our chunk-based strategy, we need to guarantee that a chunk will be truncated if the number of instructions executed within it reaches the size limit. Otherwise, if the chunk size is too large, the corresponding processor may experience repeated rollbacks due to the increased risk of collision. In order to further take advantage of the latest hardware features, we use the VMX-preemption timer that provided in the Intel VT to automatically truncate a chunk. The VMX-preemption timer provide a generic mechanism for VMM to preempt VM execution after a specified amount of time [24]. Specifically, we program the initial chunk size limit into the timer. Then, it will count down in the VMX non-root operation according to a ratio of TSC and CPU will save the timer value on each successive VM exit. When the timer counts down to zero, a VM exit will be triggered and captured by us to truncate this chunk.

Although the VMX-preemption timer does not always guarantee instruction-level precision, we do not need to guarantee a chunk will be properly truncated at an exact point in the instruction stream during recording. The VMX-preemption timer, therefore, appears opportune to truncate a chunk during recording.

## 5 Optimizations

This section describes two more optimizations for our chunk-based strategy to further improve the performance.

### 5.1 Caching Local Copies

During recording, a COW operation will be triggered to create a local copy on the first write to each memory page. In our 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.

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 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 Fig. 7: 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

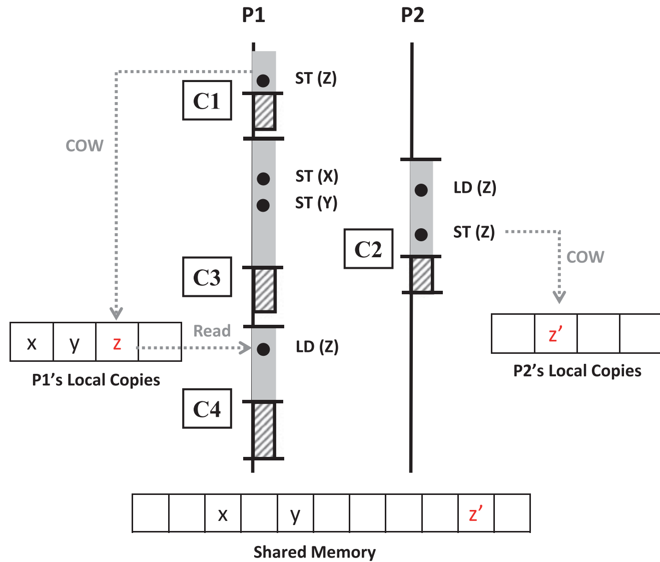



Fig. 7. An example of reading outdated data from local copies.

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. Though caching local copies can avoid repeatedly triggering COW operations, it requires extra work like traversing through the local cache to find and update the outdated copies. Therefore, in the current implementation, we limit this cache to a fixed size (0.1 percent of the main memory size) with a modified Least Recently Used (LRU) replacement policy.

### 5.2 Double Buffering

In our decentralized commit protocol, the bitmap clearing operation 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 bitmap clearing operation, both these operations must be done with the lock held.

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 public bitmap and the other as the private bitmap. By doing so, we avoid locking the bitmap while clearing it and therefore make it possible to clear and update the bitmap at the same time. More specifically, one virtual processor can clear its private bitmap without the lock held when other chunks are free to set the public bitmap simultaneously. We switch these two bitmaps right after the virtual processor broadcasts its update and still holds the lock. Therefore, the correctness is guaranteed since this switch operation is protected by the lock and the old private bitmap becomes the new public bitmap when the corresponding virtual processor is ready to receive the updates.

## 6 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.

### 6.1 Experimental Setup

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

### 6.2 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 [28]; radiosity, water_nsquared, water_spatial, and barnes from SPLASH-2 [29]. In our evaluation, all workloads are tested with simlarge or native input sets which contain bigger working sets and more parallelism. 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.

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 [18]. Pbzip2 is a parallel file compressor which uses pthreads. We use pbzip2 to decompress a 111 MB Linux-kernel source file.

### 6.3 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

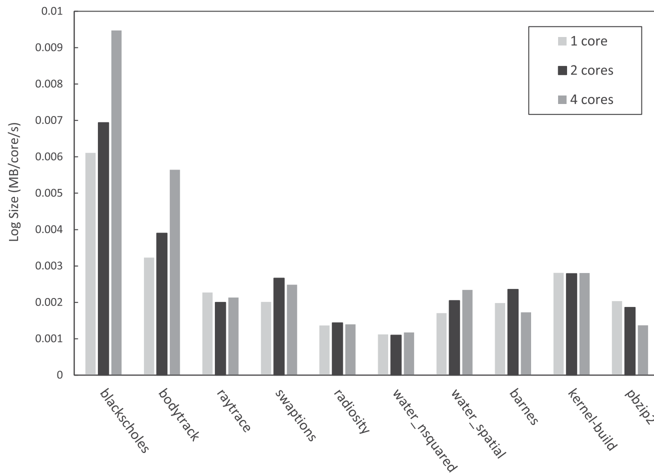



Fig. 8. Log size produced by Samsara during recording (compressed with gzip).

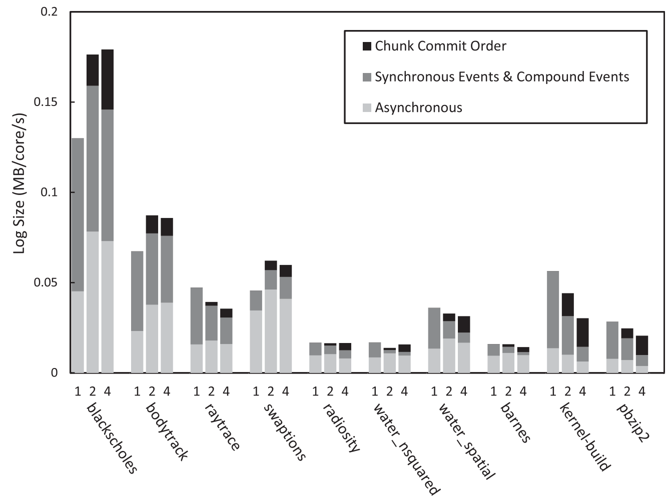



Fig. 9. The proportion of each type of non-deterministic events in a log file (without compression).

recording. The log size of some prior works is approximately 2 MB/1 GHz-processor/s [10]. Some can support only a few seconds' recording which is difficult to satisfy long-term recording needs [10].

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. Fig. 8 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 and 0.0032 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 0.0025 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 0.18 MB/core/s when recording the workloads in SPLASH-2 and kernel-build on two dual-core Xeons [18]. DeLorean generates logs at an average rate of 0.03 MB/core/s when recording the workloads in SPLASH-2 on eight simulated processors [12].

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. Fig. 9 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 8.89 percent with 2 cores and 18.47 percent 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 percent 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 [12].

### 6.4 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 Fig. 10.

The average performance overhead introduced by Samsara is $2.1\times$ for recording computation intensive applications on two cores, and $4.1\times$ on four cores. For I/O intensive applications, the overhead is $3.4\times$ on two cores and $5.9\times$ on four cores. This overhead is much smaller than the ones reported by prior works in software-only schemes, which cause about $16\times$ or even $80\times$ overhead when

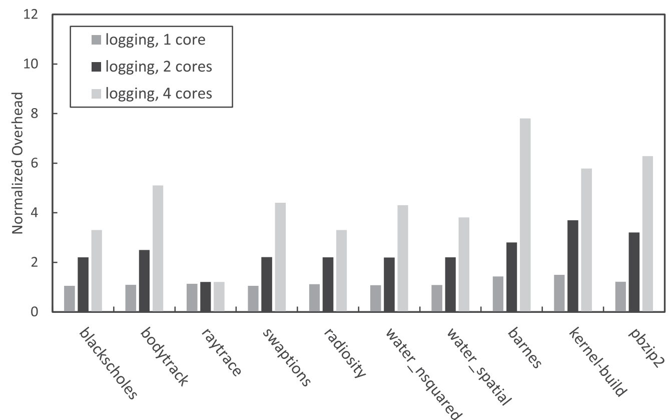



Fig. 10. Recording overhead compared to the native execution.

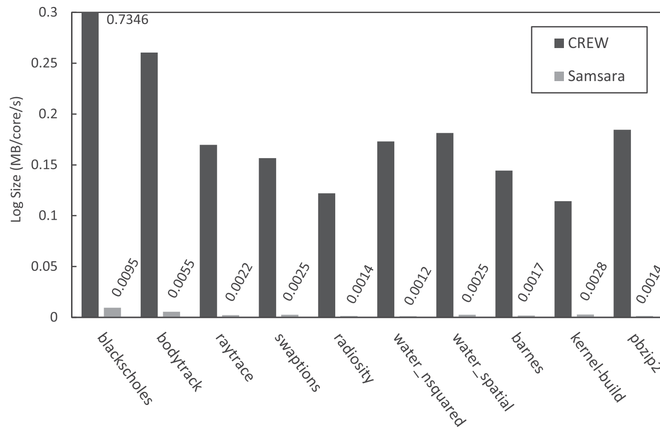



Fig. 11. A comparison of the log file size between Samsara and CREW (4 cores, compressed with gzip).

recording similar workloads on two or four cores [19], [21]. 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 percent of the whole execution time is spent on handling page fault VM exits in Samsara, while prior CREW approaches suffer from more than 60 percent execution time spent on handling page fault VM exits.

Among the computation intensive workloads, barnes has a relatively high overhead (about $2.8\times$ on two cores), while retrace has a negligible overhead (about $0.2\times$ 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 memory 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 [18]), which may lead to unnecessary rollback and increase performance overhead, and our page-level COW approach may cause write-amplification, which may lead to cache pollution and also increases memory traffic and therefore decreases the performance. 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.

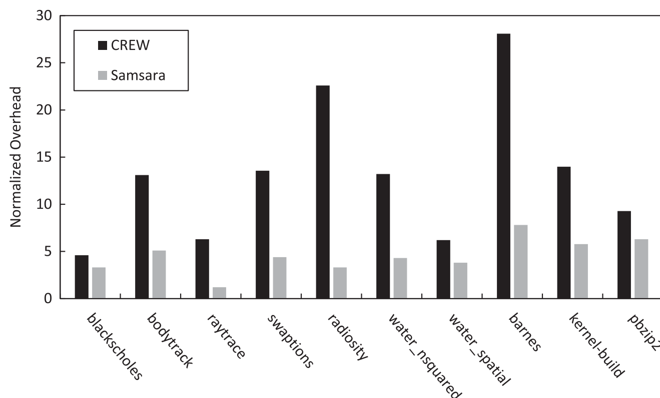



Fig. 12. A comparison of recording overhead between Samsara and CREW (4 cores).

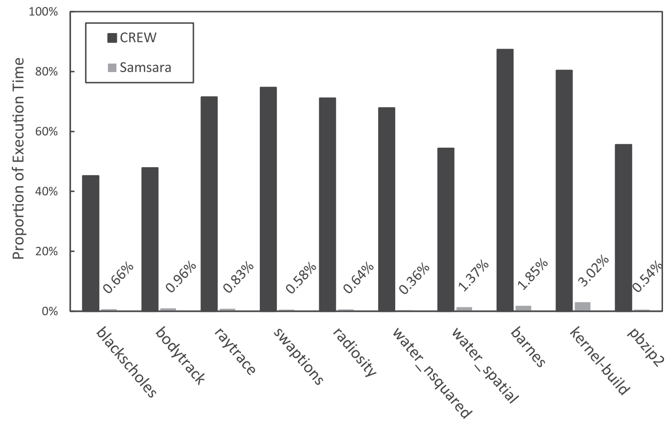



Fig. 13. Proportion of the execution time consumed on handling page fault VM exits (4 cores).

Therefore, the execution time is not long enough to amortize the cost of the chunk commits in these workloads.

## 6.5 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 [18] in our testbed.

*Log Size.* Fig. 11 shows the comparison against CREW protocol in log file size, in which Samsara reduces the log file size by 98.6 percent (i.e., from 0.22 to 0.003 MB/core/s). To understand the 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 98 percent of the events are memory access interleaving in CREW protocol, while only 8.9 percent of the events in Samsara are chunk commit orders (on two cores).

*Performance Overhead.* We also compare the performance overhead of Samsara and the CREW protocol. The results in Fig. 12 illustrate that with four cores Samara reduces the overhead by up to 81.0 percent and the average performance improvement is 62.2 percent 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. Fig. 13 shows that 65.6 percent of the whole execution time is spent on handling page fault VM exits in the CREW protocol. In contrast, this proportion is only 1.1 percent in Samsara due to the HAV and chunk-based strategy we used.

## 6.6 Benefits of Innovations and Optimizations

The next evaluation focuses on quantifying the performance improvement due to each of Samsara's innovations and optimizations.

*Obtaining R&W-Set via HAV.* We first evaluate the performance improvement of using the accessed and the dirty flags of HAV. The results in Fig. 14 illustrate that with four cores Samara significantly reduces the overhead from $13.2\times$ to $4.5\times$ on average by leveraging HAV. This improvement results from avoiding the read and the write page faults. Further experiment reveals that compared with the

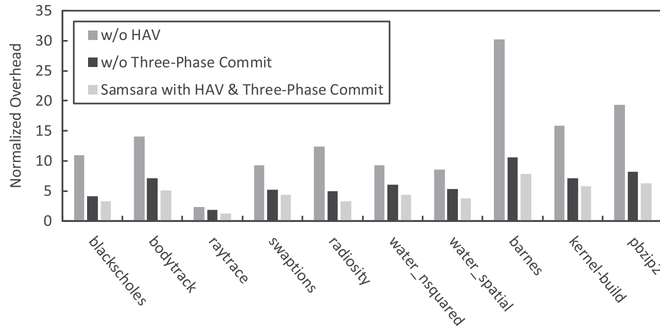



Fig. 14. The performance improvement due to Samsara's two innovations (4 cores).

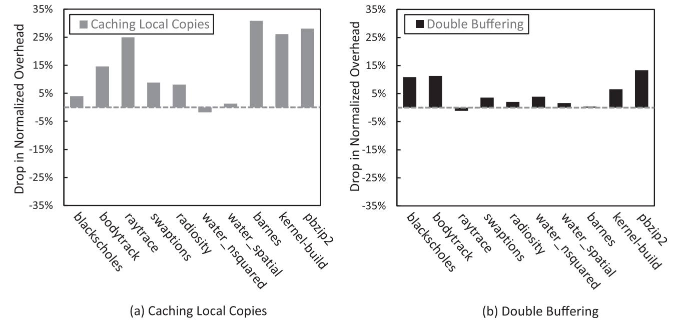



Fig. 15. Benefits of the performance optimizations (4 cores).

approach that obtaining R&W-set via tracing memory accesses, Samsara significantly reduces the read page faults by 98.3 percent, and further reduces the write faults by 78.7 percent.

*Decentralized Three-Phase Commit Protocol.* Fig. 14 illustrates that the average performance benefits contributed by the decentralized three-phase commit protocol are 26.9 percent for recording computation intensive applications on four cores. For I/O intensive applications, the benefits decrease to 20.6 percent. This improvement results from improving parallelism via reducing the lock granularity. Further experiment shows that the average time spent on waiting for the commit lock is reduced by 96.6 percent on average.

*Caching Local Copies.* Fig. 15a shows that the average performance benefits contributed by caching local copies are 12.4 percent for recording computation intensive applications on four cores. For I/O intensive applications, the benefits increase to 26.9 percent. 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 applications, like water_nsquared and water_spatial, which exhibit poor temporal locality, benefit less from this optimization. In contrast, since I/O intensive applications contain much more write operations and usually exhibit strong locality, they can benefit markedly from this optimization.

*Double Buffering.* As illustrated in Fig. 15b, the performance benefits contributed by double buffering are less significant. Empirically, the average performance improvement is 4.2 percent when recording computation intensive applications on four cores. For I/O intensive applications, the improvement is 9.9 percent. The effect of this optimization is variable depending on the number of chunk commit. This explains why I/O intensive applications, which contain a lot of chunk commits due to the frequent chunk truncations caused by the large number of concurrent I/O requests, experience significant improvement from this optimization.

## 7 RELATED WORK

The idea of achieving deterministic replay based on virtualization environment was first proposed by Bressoud, et al. [7]. Similarly, ReVirt [22] can replay entire operating systems by recording all non-deterministic events within the VMM. ReTrace [30] 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 [18] is the first deterministic replay system that records and replays a multiprocessor VM on commodity hardware by leveraging CREW protocol. ReEmu [19] 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. The preliminary descriptions of this work were in [31], [32].

To further reduce recording overhead, some efficient approaches log only synchronization operations and therefore is unable to replay programs that contain data races, such as RecPlay [33], Arnold [5], and Castor [34]. Compared to these approaches, Samsara is designed to support record and replay the execution of the whole VM at the instruction-level granularity and therefore can correctly replay operating systems, device drivers and all applications that contain data races.

Hardware-based deterministic replay uses special hardware support for recording memory access interleaving. FDR [10] records interleaving between pairs of instructions, and it improves the performance by implementing the Netzer's Transitive Reduction optimization [35] on hardware. RTR [36] 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 [11] 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 [12] 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 [37] designs a chunk-based strategy for memory race recording in modern chip multiprocessors. To improve replay performance, Karma [38] 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.

Moreover, we believe that many other works can potentially benefit from the innovations introduced in Samsara. For example, an important design choice in software

transactional memory systems is how to address the conflict detection problem [39]. Prior works provide several methods, like using locks [40], Bloom filters [41], and word-based implementations [39], [40]. The idea of tracking read/write set by leveraging the HAV extensions may provide a new feasible method for the conflict detection.

## 8 Conclusion

In this article, 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. We also discuss several implementation issues as well as some optimizations critical to performance but not covered in previous works. By evaluating our system on real systems, we demonstrate that Samsara can reduce the recording overhead from $10\times$ to $2.1\times$ and reduce the log file size to 1/70th on average.

## Acknowledgments

This work was supported by the National Natural Science Foundation of China (Grant No. 61572044 and Grant No. 61170056). The contact author is Zhen Xiao.

## References

[1] H. Agrawal, R. De Millo, and E. Spafford, "An execution-backtracking approach to debugging," *IEEE Softw.*, vol. 8, no. 3, pp. 21–26, May 1991.

[2] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou, "Flashback: A lightweight extension for rollback and deterministic replay for software debugging," in *Proc. USENIX Annu. Tech. Conf.*, 2004, pp. 29–44.

[3] S. T. King and P. M. Chen, "Backtracking intrusions," in *Proc. 19th ACM Symp. Operating Syst. Principles*, 2003, pp. 223–236.

[4] A. Chen, et al., "Detecting covert timing channels with time-deterministic replay," in *Proc. 11th USENIX Symp. Operating Syst. Des. Implementation*, Oct. 2014, pp. 541–554.

[5] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen, "Eidetic systems," in *Proc. 11th USENIX Symp. Operating Syst. Des. Implementation*, Oct. 2014, pp. 525–540.

[6] X. Wu and F. Mueller, "Elastic and scalable tracing and accurate replay of non-deterministic events," in *Proc. 27th Int. ACM Conf. Int. Conf. Supercomputing*, 2013, pp. 59–68.

[7] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," in *Proc. 15th ACM Symp. Operating Syst. Principles*, 1995, pp. 1–11.

[8] J. Zhu, Z. Jiang, Z. Xiao, and X. Li, "Optimizing the performance of virtual machine synchronization for fault tolerance," *IEEE Trans. Comput.*, vol. 60, no. 12, pp. 1718–1729, Dec. 2011.

[9] J. Zhu, Z. Jiang, and Z. Xiao, "Twinkle: A fast resource provisioning mechanism for internet services," in *Proc. IEEE INFOCOM*, 2011, pp. 802–810.

[10] M. Xu, R. Bodik, and M. Hill, "A "flight data recorder" for enabling full-system multiprocessor deterministic replay," in *Proc. Int. Symp. Comput. Architecture*, 2003, pp. 122–133.

[11] S. Narayanasamy, C. Pereira, and B. Calder, "Recording shared memory dependencies using strata," in *Proc. Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2006, pp. 229–240.

[12] P. Montesinos, L. Ceze, and J. Torrellas, "DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently," in *Proc. Int. Symp. Comput. Archit.*, 2008, pp. 289–300.

[13] N. Honarmand and J. Torrellas, "RelaxReplay: Record and replay for relaxed-consistency multiprocessors," in *Proc. 19th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2014, pp. 223–238.

[14] N. Honarmand, N. Dautenhahn, J. Torrellas, S. T. King, G. Pokam, and C. Pereira, "Cyrus: Unintrusive application-level record-replay for replay parallelism," in *Proc. 18th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2013, pp. 193–206.

[15] G. Voskuilen, F. Ahmad, and T. N. Vijaykumar, "Timetraveler: Exploiting acyclic races for optimizing memory race recording," in *Proc. 37th Annu. Int. Symp. Comput. Archit.*, 2010, pp. 198–209.

[16] X. Qian, B. Sahelices, and D. Qian, "Pacifier: Record and replay for relaxed-consistency multiprocessors with distributed directory protocol," in *Proc. 41st Annu. Int. Symp. Comput. Archit.*, 2014, pp. 433–444.

[17] T. LeBlanc and J. Mellor-Crummey, "Debugging parallel programs with instant replay," *IEEE Trans. Comput.*, vol. C-36, no. 4, pp. 471–482, Apr. 1987.

[18] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen, "Execution replay of multiprocessor virtual machines," in *Proc. 4th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environments*, 2008, pp. 121–130.

[19] Y. Chen and H. Chen, "Scalable deterministic replay in a parallel full-system emulator," in *Proc. 18th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2013, pp. 207–218.

[20] K. Veeraraghavan, et al., "DoublePlay: Parallelizing sequential logging and replay," *ACM Trans. Comput. Syst.*, vol. 30, no. 1, pp. 3:1–3:24, Feb. 2012.

[21] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs," in *Proc. 8th Annu. IEEE/ACM Int. Symp. Code Generation Optimization*, 2010, pp. 2–11.

[22] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "ReVirt: Enabling intrusion analysis through virtual-machine logging and replay," in *Proc. Symp. Operating Syst. Des. Implementation*, 2002, pp. 211–224.

[23] J. Devietti, B. Lucia, L. Ceze, and M. Oskin, "DMP: Deterministic shared memory multiprocessing," in *Proc. 14th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2009, pp. 85–96.

[24] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Santa Clara, CA, USA, Jun. 2016, no. 325462–059US.

[25] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "BulkSC: Bulk enforcement of sequential consistency," in *Proc. 34th Annu. Int. Symp. Comput. Archit.*, 2007, pp. 278–289.

[26] C. Dah-Ming and J. Raj,"Analysis of the increase and decrease algorithms for congestion avoidance in computer networks," *Comput. Netw. ISDN Syst.*, vol. 17, no. 1, pp. 1–14, 1989.

[27] A. Shriraman and S. Dwarkadas, "Refereeing conflicts in hardware transactional memory," in *Proc. 23rd Int. Conf. Supercomputing*, 2009, pp. 136–146.

[28] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Dept. Comput. Sci., Princeton Univ., Princeton, NJ, USA, Jan. 2011.

[29] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *Proc. 22nd Annu. Int. Symp. Comput. Archit.*, 1995, pp. 24–36.

[30] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman, "ReTrace: Collecting execution trace with virtual machine deterministic replay," in *Proc. 3rd Annu. Workshop Model. Benchmarking Simul.*, 2007.

[31] S. Ren, C. Li, L. Tan, and Z. Xiao, "Samsara: Efficient deterministic replay with hardware virtualization extensions," in *Proc. 6th Asia-Pacific Workshop Syst.*, 2015, pp. 9:1–9:7.

[32] S. Ren, L. Tan, C. Li, Z. Xiao, and W. Song, "Samsara: Efficient deterministic replay in multiprocessor environments with hardware virtualization extensions," in *Proc. USENIX Annu. Tech. Conf.*, Jun. 2016, pp. 551–564.

[33] M. Ronsse and K. De Bosschere, "RecPlay: A fully integrated practical record/replay system," *ACM Trans. Comput. Syst.*, vol. 17, no. 2, pp. 133–152, May 1999.

[34] A. J. Mashtizadeh, T. Garfinkel, D. Terei, D. Mazières, and M. Rosenblum, "Towards practical default-on multi-core record/replay," in *Proc. 22nd Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2017, pp. 693–708.

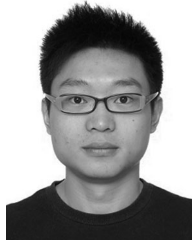
[35] R. H. Netzer and J. Xu, “Adaptive message logging for incremental program replay,” *IEEE Concurrency*, vol. 1, no. 4, pp. 32–39, Nov. 1993.
[36] M. Xu, M. D. Hill, and R. Bodik, “A regulated transitive reduction (RTR) for longer memory race recording,” in *Proc. 12th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2006, pp. 49–60.
[37] G. Pokam, C. Pereira, K. Danne, R. Kassa, and A.-R. Adl-Tabatabai, “Architecting a chunk-based memory race recorder in modern CMPs,” in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2009, pp. 576–585.
[38] A. Basu, J. Bobba, and M. D. Hill, “Karma: Scalable deterministic record-replay,” in *Proc. Int. Conf. Supercomputing*, 2011, pp. 359–368.
[39] P. Felber, C. Fetzer, and T. Riegel, “Dynamic performance tuning of word-based software transactional memory,” in *Proc. 13th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2008, pp. 237–246.
[40] D. Dice, O. Shalev, and N. Shavit, *Transactional Locking II*. Berlin, Germany: Springer, 2006, pp. 194–208.
[41] M. F. Spear, M. M. Michael, and C. von Praun, “RingSTM: Scalable transactions with a single atomic instruction,” in *Proc. 20th Annu. Symp. Parallelism Algorithms Archit.*, 2008, pp. 275–284.

**Shiru Ren** is currently working toward the PhD degree in the School of Electronics Engineering and Computer Science, Peking University. His research interests include virtualization technologies, operating system, fault tolerance, and distributed system. His recent research aims to implement efficient deterministic replay system on commodity multi-core processors.

**Le Tan** received the master’s degree from the School of Electronics Engineering and Computer Science, Peking University, in 2016. During his student life, he focused on exploring system level developing techniques, especially the operating system and the virtualization technology. Currently, he is a software engineer working at Microsoft.

**Chunqi Li** received the master’s degree from the School of Electronics Engineering and Computer Science, Peking University, in 2015. His research focuses on virtualization, operating system, and distributed system. Currently, he is a software engineer working at Google.

**Zhen Xiao** is currently a professor in the Department of Computer Science at Peking University, Beijing, China. He received the PhD degree from Cornell University, Ithaca, NY, in January 2001. After that he joined as a senior technical staff member at AT&T Labs—Research at Florham Park, NJ, and then as a research staff member at IBM T.J. Watson Research Center. His current research interests include cloud computing, virtualization, and various distributed systems issues. He is a senior member of the ACM and the IEEE.



**Weijia Song** received the PhD degree from Peking University, in 2014. He is a research associate in the Computer Science Department, Cornell University. His research is in the area of distributed and network system, focusing on cloud file/storage systems, cloud scheduling.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.