Samsara: Efficient Deterministic Replay with Hardware Virtualization Extensions

Shiru Ren, Chunqi Li, Le Tan, and Zhen Xiao *

School of Electronics Engineering and Computer Science, Peking University

{rsr, lcq, tanle, xiaozhen}@net.pku.edu.cn

Abstract

Deterministic replay, which provides the ability to travel backward in time and reconstructs the past execution flow of a multi-processor system, has many prominent applications including cyclic debugging, intrusion detection, malware analysis, and fault tolerance. Previous software-only schemes cannot take advantage of modern hardware support for replay and suffer from excessive performance overhead. They also produce huge log sizes due to the inherent drawbacks of the point-to-point logging approach used. In this paper, we propose a novel approach, called Samsara, which uses hardware-assisted virtualization (HAV) extensions to achieve an efficient software-based replay system. Unlike previous software-only schemes that record dependences between individual instructions, we record processors' execution as a series of chunks. By leveraging HAV extensions, we avoid the large number of memory access detections which are a major source of overhead in the previous work and instead perform a single extended page table (EPT) traversal at the end of each chunk. We have implemented and evaluated our system on KVM with Intel's Haswell processor. Evaluation results show that our system incurs less than 3X overhead when compared to native execution with two processors while the overhead in other state-of-the-art work is much more than 10X. Our system improves recording performance dramatically with a log size even smaller than that in hardware-based scheme.

* The contact author is Zhen Xiao.

APSys 2015, July 27–28, 2015, Tokyo, Japan.
Copyright © 2015 ACM 978-1-4503-3554-6/15/07…$15.00.
http://dx.doi.org/10.1145/2797022.2797028

1. Introduction

Current multi-processor architectures are non-deterministic. When supplied with the same inputs, they cannot be expected to reproduce the past execution flow exactly. The lack of repeatability on a multi-processor system complicates debugging, security analysis, and fault tolerance. It greatly restricts the development of parallel programming and some security applications.

Deterministic replay helps to reconstruct non-deterministic multi-processor executions. It is extensively used in several different applications. For program debugging, it can reproduce bugs and allow a programmer to inspect the program state. For security analysis, it can facilitate a quick analysis of an attack. For fault tolerance, it provides the ability to recover whole system states, which can be used in hot-standby system [18, 19] or data recovery.

Current research on deterministic replay for single processor is relatively mature with some commercial products available [4, 17]. However, the emergence of multi-processor systems poses a new challenge to deterministic replay. Since memory accesses from multiple processors to a shared memory object may interleave in some arbitrary order, it has become a major source of non-deterministic events, which may affect the processor's execution.

Existing researches on deterministic replay for multi-processor can be divided into two categories: software-only schemes and hardware-based schemes. Research in the first category achieves deterministic replay by modifying the OS, the compiler, the runtime libraries or the virtual machine manager (VMM) [3, 5, 11, 14, 15]. Some of them work at the application-level and cannot handle non-deterministic and I/O events in the OS [1, 8, 11, 12, 15]. To achieve full-system level replay, virtualization-based approaches were proposed in [3, 5, 13] which leverage the CREW protocol to serialize and log the total order of the memory access interleaving [7]. Each memory access operation must be checked for logging before execution which results in serious performance degradation and huge log sizes.

To reduce the recording overhead, research in the second category add special hardware components into the processor to detect and record memory interleaving [9, 10, 16].

Typically, they redesign the cache coherence protocol to identify and record coherence messages between processors [6]. These schemes require modifications to the existing hardware, which increases the complexity of the circuits. Thus, most of them have been modeled only using software simulations and are largely impractical for use in real systems.

Hence, we believe software-only schemes will be the main viable approach in the foreseeable future. Although there is no commercial processor with dedicated hardware-based record and replay features, some modern hardware characteristics in these processors are available to boost performance of the software-based deterministic replay systems significantly. The main motivation of our work is to take advantage of these characteristics to achieve efficient record and replay in commodity hardware.

The emergence of hardware-assisted virtualization (HAV) provides the possibility to meet the above requirements. Take Intel Virtualization Technology as an example. Intel's recent Haswell microarchitecture introduces accessed and dirty flags for EPT that provides hardware support to detect which memory pages have been accessed or updated during past execution. By leveraging this characteristic, we avoid the large number of memory access detections in previous approaches and instead obtain working set by a single EPT traversal. Moreover, HAV provides a more efficient full-system virtualization platform after years of development. Therefore, it is the most promising platform that can be used to implement an efficient replay system.

To the best of our knowledge, our prototype, Samsara, is the first software-based deterministic replay system implemented on the HAV platform, which support record and replay in the multi-processor environment on commodity hardware. By leveraging the chunk-based recording strategy on the HAV platform, we obtain significant improvements in performance and log size reduction compared to the previous software schemes.

The rest of the paper is organized as follows. Section 2 describes the general architecture of our system and gives a brief description of how to record and replay non-deterministic events. Section 3 illustrates the specific implementation on recording and replaying the memory interleaving with HAV extensions. Then, we evaluate the space overhead and performance slowdown of our prototype in Section 4. Finally, Section 5 concludes the paper.

2. Design Overview

As a software-only approach, Samsara is implemented inside the virtual machine manager (VMM), which has access to the entire virtual machine and can take full advantage of HAV extensions. Unlike application-level approaches, Samsara is designed to support record and replay the execution of the whole virtual machine. This design extensively expands the application range of deterministic replay.

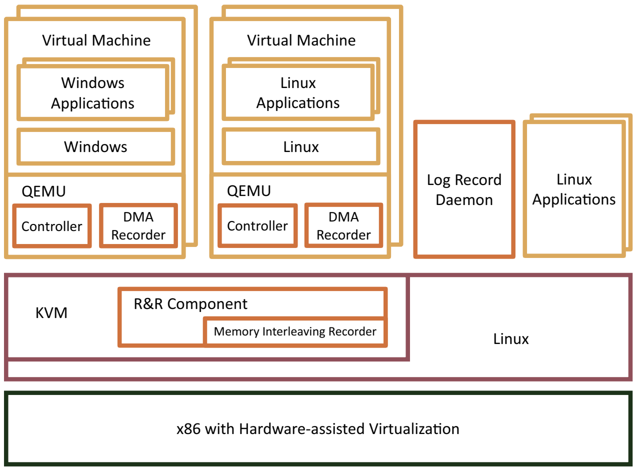

Figure 1. Architecture overview.

In this section, we first illustrate the general architecture of our system. Then, we discuss how to record and replay all non-deterministic events.

2.1 System Architecture

Samsara implements the deterministic replay as an extension to VMM, as illustrated in Figure 1. The architecture of Samsara consists of four principal components, namely, the Controller, the DMA recorder, the R&R (record and replay) component, and the log record daemon. The controller is in charge of all policy enforcement. It provides a control interface to users, manages R&R component in KVM, and is in charge of log transfer. The R&R 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 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 different log files for multiple VMs' recording and replaying simultaneously.

Before recording, controller initializes a snapshot of the whole VM states. Then all non-deterministic events and the exact point in the instruction stream where these events occurred will be logged by the R&R component during recording. Meanwhile, it transfers these log data to the userspace log record daemon, which is responsible for the persistent storage and management of the logs.

The replay phase is initialized by loading the snapshot to restore all VM states. During replay, virtual processors' execution is controlled by the R&R component, therefore all external events will be ignored, and each recorded event will be injected at the exact same point.

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 take place on the exact point 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 need to inject logged data to where the I/O (or RDTSC) instruction is trapped into the VMM.

Asynchronous Events. These events originate from external devices toward the VM at arbitrary times, such as external interrupt. The result of them is deterministic; but their timing is not. Therefore, to replay them, all these events must be identified with a three-tuple timestamp like the approach proposed by ReVirt [4]. 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 timing and result. DMA is an example of these events, because DMA completion 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 contents of a DMA event. In practice, the interrupt will be properly injected during replay, so we need to guarantee the DMA transfer is done prior to the interrupt injection. Unfortunately, the access orders to the DMA memory between the emulated device and virtual processors may induce non-deterministic inter-leavings during replay. Hence, we treat DMA device as a virtual processor with the highest priority.

Memory interleaving. Memory interleaving is also an asynchronous event. Moreover, the number of memory interleavings is an order of magnitude larger than the sum of all other non-deterministic events. Therefore, how to record and replay memory interleaving is the most important challenge in a replay system.

3. Record and Replay Memory Interleaving with HAV Extensions

As a software-only scheme, Samsara leverages some HAV features to achieve a chunk-based memory interleaving recording approach. The following illustrates the specific implementation and optimizations of this scheme.

3.1 Chunk-based Strategy

Previous software-only schemes leverage CREW protocol to serialize and log the total order of the memory access interleaving [7], which produces huge log size and excessive performance overhead because of each memory access must be checked for logging before execution. Therefore, chunk-based approach has been proposed on the hardware-based replay system to reduce this overhead [9]. However, this

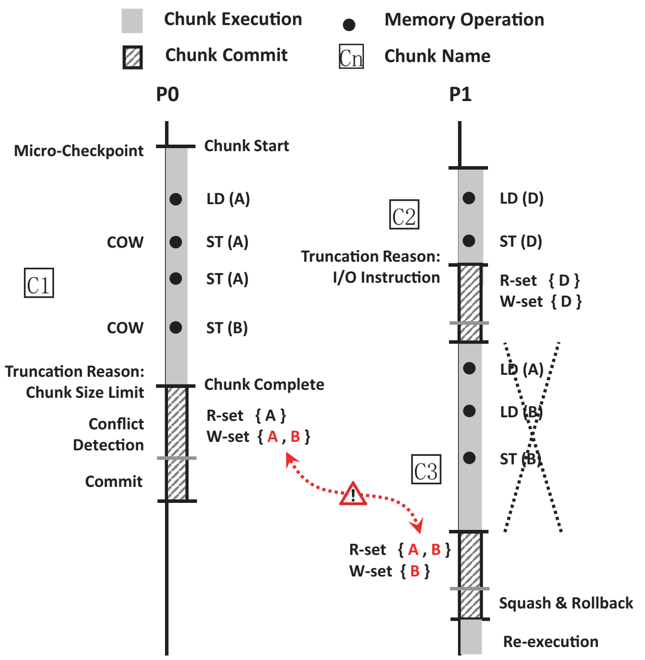

Figure 2. The execution flow of our chunk-based approach.

approach is difficult to be applied to a software-only replay system, because it still needs to trace each memory access to obtain the working set during chunk execution which is also a time-consuming process similar to the point-to-point logging approach. By leveraging HAV extensions, we avoid these detections and instead perform a single EPT traversal at the end of each chunk.

To implement a chunk-based recoding scheme, we need to restrict virtual processors' execution into a series of chunks. In our system, a chunk is defined as a finite sequence of machine instructions. 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 a same system state as if these chunks were executed serially.

To enforce serializability, 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, it either commits, copying all local copies back to shared memory instantly, or squashes, 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 recoding performance, we leverage lazy conflict detection. Namely, we defer detection until chunk completion. When a chunk completes, we obtain the read- and write-set (R&W-set) of this chunk. Afterwards, we intersect all write-sets of other concurrent chunks with this R&W-set. If the intersection is not empty, which means there are collisions, then this chunk must be squashed and re-executed.

Finally, there are some instructions that may violate atomicity because they lead to externally observable behaviors (such as I/O instructions that may modify device status and control activities on a device). Once these instructions are executed in a chunk, this chunk could not be squashed and re-executed. Therefore, we truncate a chunk when one of these instructions is encountered. Then its execution 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 virtual processor's status 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 modification to this page will be per-formed 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 chunk completion, we obtains the R&W-set of this chunk. 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 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 instantaneously, and then we rollback the virtual processor status with the micro-checkpoint we made at beginning and re-execute this chunk.

3.2 Obtain R&W-set Efficiently via HAV

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

Like the accessed and dirty flags in ordinary paging-structure entries, processors with Intel VT also support corresponding flags in EPT entries. Whenever the processor uses an EPT entry as part of 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 will be set. Therefore, we can obtain the R&W-set by gathering all leaf entries which the accessed or dirty flag is set.

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 parts of memory to be absent. Similar feature is also present for the accessed and 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 brings practically negligible overhead.

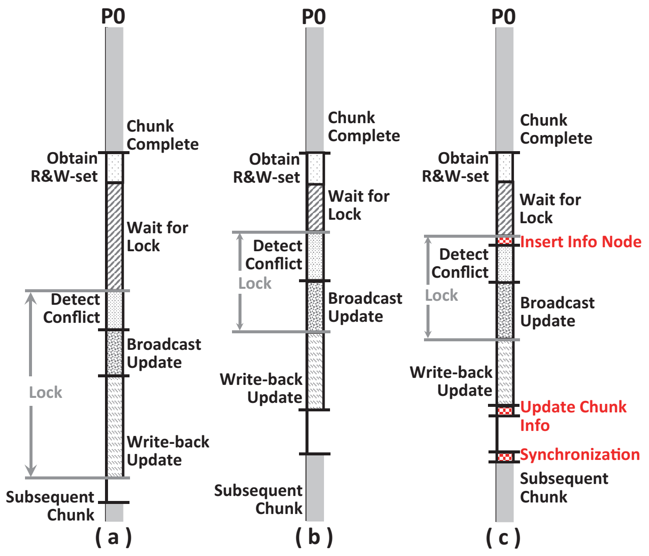

Figure 3. General design of decentralized three-phase commit protocol: a) chunk timeline of a naïve design, b) moving update write-back operation out of the synchronized block, and c) a design of decentralized three-phase commit protocol.

3.3 A Decentralized Three-Phase Commit Protocol

Apart from obtaining R&W-set, chunk commit is another time-consuming process. In this subsection, we will discuss how to optimize this part for a better performance by utilizing a decentralized three-phase commit protocol.

Some hardware-based solutions add a centralized arbiter module to processors to ensure that chunks commit one at a time, without overlap [9]. However, when it comes to software-only schemes, an arbiter may occupy plenty of processing resources as a long-running process. Thus, we propose a decentralized three-phase commit protocol to perform chunk commit efficiently.

The chunk commit process should include at least three steps, conflict detection that determines whether this chunk can be committed, update broadcast that notifies other processors which memory pages are modified, update write-back that copies all updates back to shard 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. The whole commit process is performed while holding this lock.

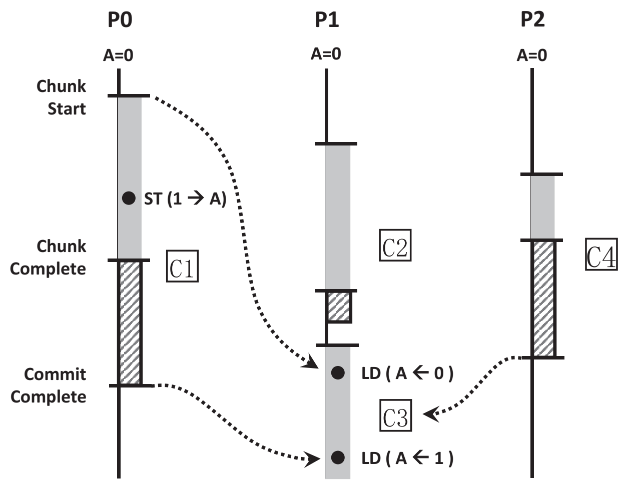

Figure 4. An example of out-of-order commit.

Unfortunately, our analysis indicates that the time consumed on waiting for this lock is excessive, which takes up nearly 40% of the whole commit time. This severely limits the performance and scalability of our system. To address this issue, the straightforward idea is to reduce the lock granularity. We observe that the update write-back operation involves serious performance degradation due to lots of page copies, and all these pages committed con-currently by different chunks have no intersection, which is guaranteed by conflict detection. Based on these observations, we move this operation out of the synchronized block to reduce lock granularity, as shown in Figure 3 b). This, on the one hand noticeably reduces expensive lock operations, and on the other, also improves parallelism because multiple chunks can commit concurrently.

Unfortunately, this design may result in chunk committing out-of-order, thus violates serializability. One example is shown in Figure 4. C1 writes A, then finishes its execution first and starts to commit. Afterwards, 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 completed its commit, C3 gets two different values of a same memory object, which violates serializability obviously. To avoid this problem, we need to guarantee that before starting C3, P2 waits until all the other chunks which start committing prior to the commit point of C2 (e.g., C1) complete their commit.

To avoid this potential issue induced by out-of-order commit, we propose a decentralized three-phase commit protocol to support parallel commit while ensuring serializability. To eradicate out-of-order commit, we introduce a global linked-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. To avoid multiple chunk update this list concurrently, operations on it are responsible for acquiring a lock. This protocol consists of three phases as shown in Figure 3 c).

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

2) The commit phase, in which 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 at this phase, because pages committed by different chunks certainly have no intersection.

3) The synchronization phase, in which 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 there is at least one flag is 0, then this processor must be blocked. Otherwise, the processor removes its own info node from the linked-list and begins to execute next chunk. In practice, this blocking almost never happens, because a virtual processor tends to exit to QEMU to emulate some device operations before executing the next chunk, which happens to provide sufficient time for other chunks to complete their commit. This protocol can satisfy the serializability requirement because it strictly guarantees that the processor which first begins to commit this chunk will execute the subsequent chunk preferentially.

This design noticeably improves performance via reducing lock granularity. It also reduces the time spent on waiting for lock, because the shorter the time a chunk holds a lock, the lower the probability that other chunks will request it while the first chunk is holding it.

3.4 Replay Memory Interleaving

Replay memory interleaving is relatively simple under chunk-base strategy. To enable practical replay, we merely need to guarantee all chunks will be properly re-built and executed in the original order. Therefore, a hardware performance counter is required to confirm that a chunk can be truncated at the recorded timestamp. Besides, we also need to ensure that all preceding chunks have been committed successfully before the subsequent chunk starts.

Compared with some previously-proposed schemes that are not designed for parallel replay, Samsara greatly boosts replay performance via allowing processors execute concurrently in replay. Moreover, unlike fine-grained approaches which perform too much constraint that will hurt replay efficiency, we avoid these restrictions by leveraging chunk-based strategy to replay interleaving in coarse-grain.

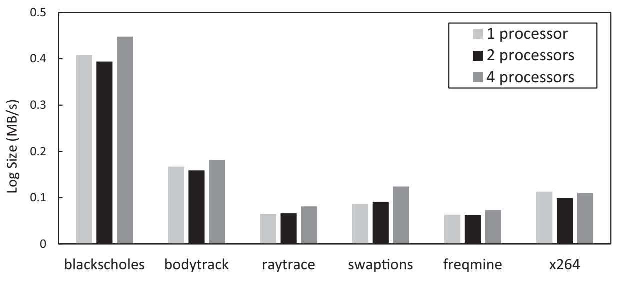

Figure 5. Log size produced by Samsara during recording (2GB memory, compressed with gzip).

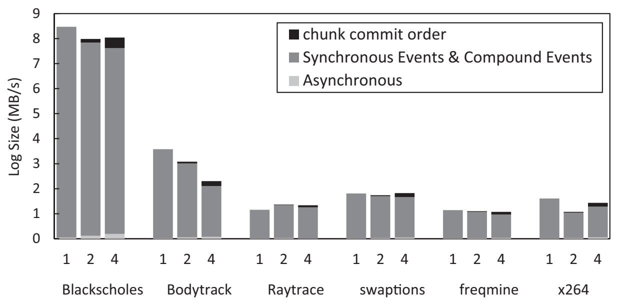

Figure 6. The relative proportions of each type of non-deterministic events in a log file (2 GB memory, compressed with gzip).

4. Evaluation

This section discusses our evaluation of Samsara. We first illustrate our experimental setup and the workloads we choose. Then, we assess two different aspects of our system.

4.1 Experimental Setup and Workloads

All the experiments are based on a Dell Precision T1700 Workstation, with a 4-core Intel Core i7-4790 processor running Ubuntu 12.04 with Linux kernel version 3.11.0 and QEMU-1.2.2. The host machine has 12GB memory.

To evaluate how well traditional chip-multiprocessors workloads run under our system, we ran PARSEC [2] benchmark suite on our testbed. This is a well-studied benchmark composed of multithreaded programs. We choose six workloads from PARSEC: blackscholes, bodytrack, raytrace, swaptions, freqmine, and x264. We choose these because they exhibit diverse characteristics that represent the different worst-case applications.

4.2 Log Size

Log size is an important consideration of the replay systems. Usually, recoding non-deterministic events will generate huge space overhead that limits the duration of the recording interval. Some previous approaches produce approximately 1-2 MB log/1GHz-processor/second after compression. Some can support only a few seconds' recording [16], which is difficult to satisfy long-term recording needs.

Experiments show that Samsara produces a much smaller log size than previous approaches. Figure 5 shows the log sizes for all the benchmarks applications. The experiment indicates that the log size generated by recoding does not increase commensurately with the number of processors. Samsara generates log at an average rate of 0.127 MB/s and 0.151 MB/s for recoding two and four processors, respectively. For comparison, the average log size generated by a single processor recoding (which does not need to record memory interleaving) is 0.131 MB/s.

We achieved a significant reduction in log size because the size of the chunk commit log is practically negligible compared with other non-deterministic events. Figure 6 illustrates the relative proportions of each type of non-deterministic events in the log file. In most workloads, the chunk commit log represents a tiny fraction (2.33% with 2 processors and 7.37% with 4 processors on average).

The log size in our system is even smaller than hardware-based schemes, since we can further reduce it via increasing the chunk size which is impossible in hardware due to the risk of cache overflow [9].

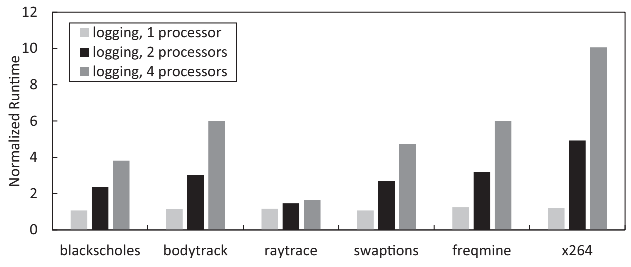

Figure 7. Record overhead compared to Native Execution.

4.3 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 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 7.

We can see from this figure that the average performance overhead introduced by Samsara are 2.6X and 5.0X for recording these workloads on two and four processors. This overhead is much smaller than previous software-only schemes which may cause about 16X or even 80X overhead when recoding similar workloads with two cores [3, 13].

Among these workloads, x264 has a relatively large overhead (nearly 5X with two processors), while retrace has a negligible overhead (less than 0.4X with two processors) in contrast. By analyzing the shared memory access pattern of these two workloads, we find that retrace contains an immense amount of read operations than write. These read operations will not bother our system, because we do not trace any read access. However, x264 contains a lot of shared memory writes, and the coarse-grained parallelism result in some false sharing issues which may cause the performance to suffer.

5. Conclusion

In this paper, we made the first attempt to leverage HAV extensions to achieve an efficient software-based replay system. Unlike previous schemes that record dependences between individual instructions, we record processors' execution as a series of chunks and avoid the large number of memory access detections by performing a single EPT traversal at the end of each chunk. Moreover, we propose a decentralized three-phase commit protocol to reduce the lock granularity of the chunk commit process, which boost performance significantly. Evaluation shows that our system improves recording performance dramatically with a log size even smaller than that in hardware-based scheme.

Acknowledgments

The authors would like to thank the anonymous reviewers for their comments. This work was supported by the National Natural Science Foundation of China (Grant No. 61170056), the National High Technology Research and Development Program (863 Program) of China (Grant No. 2013AA013203).

References

[1] G. Altekar and I. Stoica. Odr: Output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, 2009.

[2] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[3] Y. Chen and H. Chen. 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.

[4] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation*, 2002.

[5] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2008.

[6] N. Honarmand and J. Torrellas. 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.

[7] T. LeBlanc and J. Mellor-Crummey. Debugging parallel programs with instant replay. *Computers, IEEE Transactions on*, C-36(4):471–482, April 1987.

[8] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. 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.

[9] P. Montesinos, L. Ceze, and J. Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the International Symposium on Computer Architecture*, 2008.

[10] S. Narayanasamy, C. Pereira, and B. Calder. Recording shared memory dependencies using strata. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[11] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[12] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. Pres: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, 2009.

[13] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2010.

[14] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference, General Track*, 2004.

[15] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. Doubleplay: Parallelizing sequential logging and replay. *ACM Trans. Comput. Syst.*, 30 (1):3:1–3:24, Feb. 2012.

[16] M. Xu, R. Bodik, and M. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the International Symposium on Computer Architecture*, 2003.

[17] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the Third Annual Workshop on Modeling, Benchmarking and Simulation*, 2007.

[18] J. Zhu, Z. Jiang, and Z. Xiao. Twinkle: A fast resource provisioning mechanism for internet services. In *Proceedings of the IEEE INFOCOM*, 2011.

[19] J. Zhu, Z. Jiang, Z. Xiao, and X. Li. Optimizing the performance of virtual machine synchronization for fault tolerance. *IEEE Transactions on Computers*, 60(12):1718–1729, Dec 2011.