Lessons Learned During the Development of the CapoOne Deterministic Multiprocessor Replay System

Pablo Montesinos, Matthew Hicks, Wonsun Ahn, Samuel T. King and Josep Torrellas
Department of Computer Science
University of Illinois, Urbana-Champaign
{pmontesi, mdhicks2, dahn2, kingst, torrellas}@cs.uiuc.edu

Abstract

Current schemes for deterministic replay of parallel applications can be of great help for programmers. Software-only replay systems, while great at operating at the abstraction of user applications, incur large CPU overheads. Hardware-only systems produce minimal, if any, CPU overhead, but because their focus is the low-level hardware primitive that records and replays the memory access interleaving, they do not integrate well with user applications. The limitations of both the software- and hardware-only schemes render them impractical for most production uses.

Capo is a hardware-assisted deterministic replay framework that combines the performance of hardware-only schemes with the flexibility of the software-only ones. *CapoOne* is the first implementation of the Capo framework. It has modest overheads and shows that practical deterministic replay of production parallel applications is feasible. This paper brings together a set of lessons learned during the development of the CapoOne prototype so that designers of future hardware-assisted deterministic replay systems can apply them, lessening their pain.

1 Introduction and Motivation

Debugging parallel applications is a complicated task because certain bugs only appear under hard-to-replicate timing conditions. Consequently, programmers would benefit greatly from tools and techniques that could help them debugging these programs.

One such technique is *deterministic replay of execution*. Current software-only deterministic replay systems [1, 9, 3, 10, 4] are flexible but they perform slowly on (or do not work with) multiprocessors. Thus, hardware-based schemes have been proposed [11, 8, 12, 6, 5]. Even though they record multiprocessor execution efficiently, they impose restrictions on how the users can record and replay applications, rendering them impractical. Capo [7] defines a set of OS-level abstractions and a software-hardware interface for *practical hardware-assisted* deterministic replay. It combines the best of both worlds: it is flexible like the software-only schemes and is efficient like the hardware-only techniques.

CapoOne is our first implementation of the Capo interface. The current prototype uses DeLorean-like hardware [6] as the hardware replay substrate. On top of this hardware, CapoOne runs a standard Linux operating system with a modified kernel. CapoOne has good performance and it is able to record and/or replay standard Linux applications. Moreover, it can independently record and/or replay two or more user applications that run simultaneously. CapoOne required a 18 man-month development effort. In this time, we i) implemented a new hardware simulator for DeLorean, ii) modified the Linux kernel and iii) developed the *Replay Sphere Manager*, a complicated piece of software that manages the hardware/software interaction of the replay system.

In this paper, we describe some practical lessons we learned during the development of CapoOne. In hindsight, our biggest mistake was to underestimate the time and complexity of the software components. Consequently, our intention with this paper is to expose some key aspects that, even though they might look intuitive now, were time-consuming because they are full of subtleties. We hope that by exposing them here, we can help researchers, saving precious development time.

This paper first focuses on the issues that we faced when we converted a full-system, hardware-based replay system such as DeLorean [6] into a hardware-assisted replay system for user applications. The key element was to make sure that CapoOne only recorded and replayed instructions that belonged to the target application. The paper then discusses how to deal with interrupts and exceptions. Some of these events are non-deterministic and require saving some state information into a log so that they can be replayed. Others, even though they are non-deterministic events as well, can be treated in a way that does not require CapoOne to record them in a log —perhaps at the cost of some minor performance degradation.

In this paper we also describe in detail how CapoOne ensures that copying data from the kernel into the application is deterministic. This was arguably the hardest piece of code that we had to write. We give insights on why our first approach did not work and what was required to get it working correctly.

This paper is organized as follows. Section 2 presents our hardware-assisted replay system. Section 3 describes CapoOne's hardware changes. Section 4 describes user-to-kernel transitions and Section 5 describes other system issues.

2 System Overview

This section presents our system. It first describes the concept of the *Replay Sphere* and its interactions with the system. Then, it shows our first implementation CapoOne.

2.1 Capo

Capo [7] presents a software-hardware interface and a set of abstractions for practical *hardware-assisted* deterministic replay of execution. We refer to Capo as *hardware-assisted* because in Capo, the software is the driving force of the replay mechanism while some specialized replay hardware is in charge of recording the memory access interleaving of different threads.

The main abstraction in Capo is the *Replay Sphere*. A replay sphere is a set of user-level threads that are recorded and replayed as a unit together with their address space. Each replay sphere is independent. Therefore, there is no information on how a thread running inside a sphere (an *R-thread*) interleaves with R-threads from other spheres or with regular threads. Thus, the replay sphere isolates processes that are being recorded or replayed from the rest.

The replay sphere also helps separating the responsibilities of the software and hardware components. The hardware records and replays the R-thread interleaving in a per-sphere log called the *Interleaving Log*. For its part, the software records any other source of non-determinism that may affect the execution path of any of the R-threads of a sphere. This per-sphere log —called the *Input Log*— includes signals, return values from system calls and, in general, any data copied into the replay sphere. In a sense, the hardware records the non-determinism that occurs inside the sphere while the software records non-deterministic events that originate outside the sphere but that might affect its execution. The reader should observe that, in the absence of any input from the kernel or other process in the system, the only source of non-determinism during the execution of a user-level parallel application is its memory access interleaving.

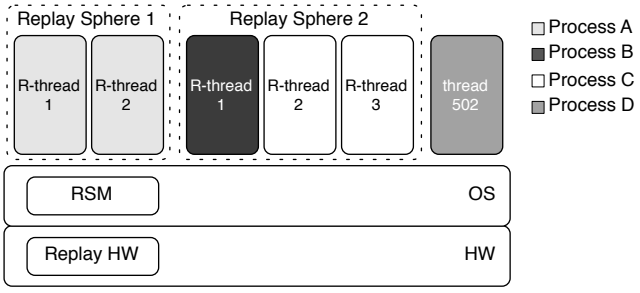

Figure 1: Capo's architecture.

Figure 1 depicts Capo's architecture. The figure shows *Replay Sphere 1* recording Process *A* and *Replay Sphere 2* replaying Process *B* and Process *C* simultaneously. Process D is executing normally and does not need to run within any replay sphere. The figure also shows Capo's main software component: the *RSM* or *Replay Sphere Manager*. The RSM is in charge of managing logs, assigning spheres to the replay hardware, etc. The RSM uses a small hardware interface to interact with the replay hardware and, as a result, Capo can work with any replay hardware scheme [11, 8, 12, 6, 5].

R-threads are part of the replay sphere for as long as they stay in user land. If one of them executes a system call, it leaves the sphere and its interleaving with the other R-threads in the sphere is not recorded until it returns from the system call. We decided to set the replay sphere boundary at the system call level because Capo's main goal is to record and replay applications and the code in the kernel is much more unpredictable and harder to track than the application's code. Alternatively, we could have set the replay sphere boundary at the library level. That would mean that any Capo implementation would have to interpose all library calls. Moreover, libraries would have to be modified so that the data copied from the library into the application could be recorded.

2.2 CapoOne

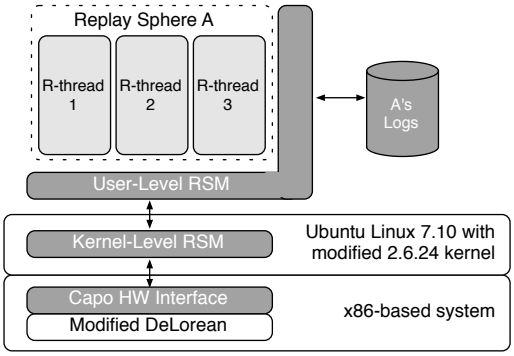

Figure 2: Capo's first implementation:CapoOne.

CapoOne is the first implementation of the Capo framework. As Figure 2 shows, CapoOne's replay hardware is based on DeLorean [6]. DeLorean uses a new execution model [2] where processors continuously execute large groups of consecutive dynamic instructions —called *chunks*— atomically and in isolation. These chunks are separated by processor checkpoints, like in hardware transactional memory. In this environment, recording the execution for replay requires that the hardware records the total commit order of chunk commits. CapoOne augments DeLorean with Capo's hardware interface so that the RSM can manage it. Section 3 gives more insight into the changes made to DeLorean.

Figure 2 also shows that CapoOne uses a standard, off-the-shelf Linux operating system. We extended the kernel to add support for replay spheres and R-threads. We also implemented new kernel functions for copying data from the kernel into the replay sphere —and vice versa. They ensure that the interleaving between the kernel and the R-threads is deterministic.

Finally, it can be seen in the figure that CapoOne's RSM has components in both user and kernel lands. The user-level component manages both the interleaving and input logs, interposes R-threads' system calls and delivers signals. For its part, the kernel-level portion manages the replay hardware, schedules spheres and records the data copied between the kernel and the replay sphere.

3 From Full-System Replay to Sphere-Based Replay

Previous hardware-based deterministic proposals [11, 8, 12, 6, 5] record and replay the entire system. CapoOne just records and replays processes running inside replay spheres. Consequently, CapoOne does not log non-deterministic events such as interrupts or DMA operations. Instead, it logs i) all inputs to the sphere and ii) the interleaving of the chunks executed by the R-threads of the same sphere. Information about other non-deterministic events is discarded. Because of ii), CapoOne requires some changes to the DeLorean hardware —in addition to augmenting to meet Capo's hardware interface specification.

In DeLorean, it is possible for a chunk to include instructions from both a user-level thread and the OS. For example, if a thread issues a system call, the last instructions the thread executed before the system call and the first few from the system call handler can end in the same chunk. CapoOne must prevent this from happening because its interleaving log can only consist of user-level instructions from the same sphere. As a result, CapoOne's hardware chunks the dynamic instruction stream differently than does DeLorean. Thus, in CapoOne, chunks can only contain instructions from one R-thread.

Moreover, when a processor detects that a chunk to be committed is the last one that belongs to an R-thread before the OS takes over the processor, the commit request includes a bit indicating that i) this is the last chunk of the R-thread for now, and that ii) the following OS chunks should not be included in the log. This is necessary because the OS can execute one or more chunks before it executes the instructions that manage the replay hardware. CapoOne must ensure that these OS chunks are not part of the interleaving log.

Lesson learned: Making sure that the interleaving log only contains chunks where all the instructions belong to the R-threads in the sphere can be challenging. However, the alternative solution, making sure that the instructions that do not belong to the sphere are also deterministic, is much more complicated.

4 User to Kernel Transitions

Section 3 described a very important design principle in CapoOne: all chunks executed by an R-thread must only include instructions from the same R-thread. This section describes how CapoOne transitions from user to kernel land so that it upholds that design principle while maintaining current semantics. We focus on user-to-kernel transitions and not on kernel-to-user ones because, in the former, CapoOne must ensure that the status of the replay sphere when R-threads leave is replayable. Returning to the sphere is always deterministic if the replay sphere exits were deterministic as well.

4.1 Interrupts

Interrupts are asynchronous events generated by hardware devices that alter the instruction stream of the processor. They are inherently non-deterministic. Full-system hardware-based replay systems [11, 8, 12, 6, 5] record, for each interrupt, when it arrived and its kind. This is not the case in CapoOne because interrupts do not directly affect the execution path of the R-threads inside a sphere. However, CapoOne must ensure that kernel code in the interrupt handler is not recorded as part of the sphere's interleaving log.

Due to CapoOne's chunk-based execution, the interrupt delivery policy balances three conflicting demands: size of the interleaving log, interrupt latency and amount of wasted work due to squashes. Thus, there are three different approaches to interrupt handling, that we call: *Finish First*, *Commit First* and *Squash First*.

In *Finish First*, a processor does not service an interrupt until the in-flight chunk reaches its predefined size and commits. Completing and committing the chunk might take some time —chunks are thousands of instructions long— so this increased interrupt latency can hurt performance if the system is executing under a heavy interrupt load. On the other hand, because the committed chunk reached the predefined size, CapoOne does not need to record anything in the interleaving log.

In *Commit First* the processor does not wait for the chunk to reach the desired chunk size. Instead, it tries to commit the not-yet-completed chunk first and it then starts servicing the interrupt. As a result, the chunk's final size is non-deterministic and it must be recorded in the interleaving log. However, the interrupt response time is better than in *Finish First*.

Finally, in *Squash First*, when an interrupt arrives, the processor squashes any in-flight chunk that has not sent its commit request to the arbiter yet. This approach is simple and allows for a fast and relatively constant interrupt response time. As a drawback, it can cause some performance degradation due to frequent squashes.

Lesson learned: It is sometimes possible to treat highly non-deterministic events —such as interrupts— as deterministic events. For example, *Squash First* and *Finish First* do not require any information to be recorded in the log. This is why we use *Squash First* in CapoOne. Moreover, it is simple to implement and did not cause any significant overhead in our experiments.

4.2 Exceptions

Exceptions are *synchronous* events that alter the dynamic instruction stream of the processor. Some of them are raised when the processor detects an anomalous condition while executing an instruction —*faults* and *traps*— and others are generated at the request of the programmer —*programmed exceptions*. Obviously, exceptions must be treated differently than interrupts, because exceptions must be delivered at precisely the right moment. CapoOne's design goal is to maintain correct exception semantics; this

section describes how CapoOne handles each exception type.

4.2.1 Handling Faults

A fault is a type of exception where the instruction that caused it re-executes after the anomalous condition has been solved. In order to maintain correct exception semantics, CapoOne must finish and commit the chunk containing all the dynamic instructions preceding the one causing the fault. Executing the fault handling code means that the processor leaves the replay sphere until the kernel completes the fault handling and the user-level execution is resumed. Therefore, the faulting instruction becomes the first instruction of the new chunk when it re-executes.

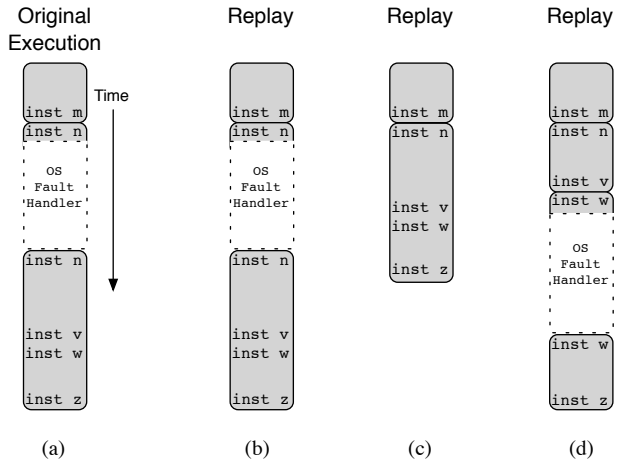

Figure 3: Fault handling in CapoOne.

Although faults are synchronous events, they are not necessarily deterministic. Consequently, CapoOne logs the size of the last chunk committed before the faulting instruction and uses that size information to recreate the same chunk during replay.

Consider Figure 3(a). Instruction n causes a page fault, so CapoOne must prematurely commit a chunk that finishes at instruction m. It must also log the chunk's size in the interleaving log. The OS takes over and services the page fault. Once the R-thread resumes, instruction n becomes the first instruction of the new chunk and it re-executes. This new chunk commits normally when it reaches the predefined chunk size —which happens right after instruction z. Figures 3(b), 3(c) and 3(d) show that there are three possible behaviors when CapoOne replays a fault.

In Figure 3(b), the processor uses the log to produce a chunk whose last instruction is instruction m. Afterwards, the processor starts a new chunk, and the same instruction n causes a page fault, making the system proceed in the same way as in the original execution.

Consider now Figure 3(c). As in Figure 3(b), the processor decides —based on the information in the log— to chunk the dynamic instruction stream at instruction m. However, instruction n does not fault this time. Then, the processor continues executing the chunk normally until it reaches the predefined chunk size.

Finally, Figure 3(d) shows the case where a fault occurs during replay but not during the initial execution. In the figure, instruction w causes an unexpected fault so the processor commits a second chunk whose last instruction is v. Note that CapoOne does not log anything now because it is replaying. Also, notice that in order for the replay to be deterministic, no other processor can commit any chunk belonging to another R-thread in the sphere until the rest of the instructions in the original chunk are committed. Thus, the processor creates a new chunk starting at instruction w and ending at instruction z. Once this new chunk commits, the replay sphere state is identical to the one at the end of Figure 3(a).

Lesson learned: Faults are non-deterministic events that must be handled properly. The most difficult case is depicted in Figure 3(d), where a fault occurs in replay and not in the initial execution.

4.2.2 Handling Traps and Programmed Exceptions

Processors raise traps and programmed exceptions after the execution of a trapping instruction. Once the OS handles them and the application resumes, the next instruction to execute is the one following the one that caused the trap. Processors must exit the replay sphere to let the OS handle traps and exceptions, so these events also cause an early commit of a processor's in-flight chunk.

As with faults, CapoOne tries to maintain the correct trap and programmed exception semantics. They differ from faults in two main aspects. First, the instruction that raises them is always the last one of the chunk. And second, they are deterministic and, therefore, CapoOne does not need to record the "irregular-sized" chunks they produce.

Lesson learned: Traps and programmed exceptions are deterministic events and require no extra logging.

5 System Issues

We learned lessons about several other system-wide issues. We describe them in this section.

5.1 Moving Data Between the Kernel and the Replay Sphere

In CapoOne, copying data from the kernel into the replay sphere is a delicate matter. First, the RSM must record in the input log the data about to be copied. And second, it must ensure that the interleaving between the kernel thread injecting the data and the R-threads in the sphere is deterministic. This is because some R-thread might concurrently access the same memory region where the kernel is copying data into, and the hardware does not record the interleaving of code outside the replay sphere. CapoOne addresses this problem by temporarily inserting the kernel's function in charge of the copy (`copy_to_user`) into the sphere. Once the kernel thread executing the function is inside the sphere, the hardware records the interleaving of the kernel thread chunks with the chunks of the R-threads of the sphere. After the copy is over, `copy_to_user` exits

the replay sphere.

Making sure that the interleaving between `copy_to_user` and the R-threads in the sphere was deterministic gave us many headaches. In our first implementation, the RSM associated the data copied by `copy_to_user` with the system call exit event corresponding to the system call that executed `copy_to_user`. At that time, it made sense that the input log entry corresponding to the system call exit event would also contain the data that the system call copied into the replay sphere. During replay, right before a system call returned, the RSM would inject the data into the sphere. We believed that from the point of view of an R-thread it did not matter at which point of the system call execution the OS copied data into the sphere.

This simple and intuitive approach worked for many of our applications. Unfortunately, it would cause deadlock in certain situations due to circular dependences between the input log and the interleaving log. To understand why, the reader must note that the RSM records in the input log the order in which R-threads enter and exit system calls. During replay, the RSM enforces the same ordering.

Consider Figure 4(a). During initial execution, R-thread A executes a system call that copies some data into the replay sphere. After `copy_to_user` is over, but before the OS exits the system call and resumes A execution, B executes a system call as well. In the input log, B's system call enter comes before A's system call exit event. In the interleaving log, the `copy_to_user` chunks come before B's chunk finishing in the system call enter instruction. The arrows in the figure show these dependences

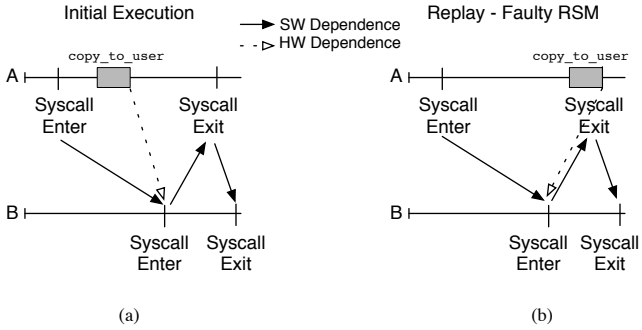

Figure 4: Circular dependences between the input log and the interleaving log cause deadlocks during replay.

Consider now Figure 4(b). In the figure, the system deadlocks while trying to replay the execution from Figure 4(a). The reason is that the RSM associated the `copy_to_user` event with the system call exit event in the input log and it is waiting for thread B to execute a system call. At the same time, the hardware follows the interleaving log and cannot let B commit the chunk that executes the system call until the `copy_to_user` have been executed.

To solve this problem in a new RSM implementation, `copy_to_user` events are their own entity and they are no longer associated with system call exit events. In hindsight, it is an obvious solution, but this was not the case during the development because our first solution works well as long as there is only one R-thread executing a system call and all the other R-threads are in user mode (i.e. no system calls, exceptions, etc).

Lesson learned: It is possible to have circular dependences between the input log and the interleaving log. These dependences can cause deadlocks under certain interleavings. We recommend making `copy_to_user` events a first-order event in the input log.

5.2 Cache Overflows

BulkSC-based [2] systems such as CapoOne keep the current chunk's speculative data in the cache until commit time. However, a chunk may access more lines mapping to a cache set than ways the set has. Certain transactional memory schemes allow storing speculative state in main memory. This is not the case for the current BulkSC-based systems. When a cache would overflow, CapoOne commits the in-flight chunk independently of its size.

Caches are not part of the replayable state, and therefore cache overflows are non-deterministic. Thus, in the event of a cache overflow, CapoOne must record the size of the prematurely-committed chunk. During replay, the processors use the information in the log to create chunks of the same size of those prematurely committed due to cache overflows.

Lesson learned: The instruction that caused the overflow does not necessarily become the first instruction of the next chunk. In CapoOne, processors can freely reorder memory operations within a chunk and even across consecutive chunks of the same processor. As a result, a memory operation not at the top of the reorder buffer can cause a cache overflow.

5.3 Self-Modifying Code

Just as strict isolation of data accesses within a chunk must be enforced by hardware, isolation of instruction memory must also be enforced. In the case of CapoOne, this is done by adding all instruction fetches to the read set of the chunk. Thus, at commit time, a code-modifying chunk is able to detect other chunks that have read and executed a stale version of the code and squash them, maintaining correct ordering. In addition, instruction cache coherence is maintained by having a committing chunk flash-invalidate its write set in all instruction caches, as well as the data caches. No recording of the code modification event has to take place in the interleaving log since maintaining the same ordering of chunks at replay time is sufficient for ensuring that the self-modifying event is deterministically replayed. During replay, signatures are also used to invalidate the instruction caches.

Besides remote code modification, local code modification must also be detected. In the case of CapoOne, all instruction fetches are checked against the local write set to make sure they are not fetching stale instructions. On detection, the current chunk is immediately committed to make the modified code memory available for execution

and a new chunk is started. The new chunk would then fetch the correct modified instruction from its data cache which has the most recent copy. Again, no recording needs to take place in the replay log since local code modification is deterministic.

Lesson learned: Self-modifying code events are fully deterministic and they can be handled seamlessly with the same hardware support (signatures and squashes). Moreover, these event do not need to be stored in any log.

6 Conclusions

CapoOne is a hardware-assisted replay system that allows the user to record and deterministically replay parallel applications running on shared-memory multiprocessors. It has modest overheads and shows that practical deterministic replay of applications is possible. This paper is a collection of lessons that we learned during its development.

For example, this paper described how isolating instructions that belong to an application that is being recorded or replayed can be a delicate issue. It also showed how interrupts and exceptions are important sources of nondeterminism and how CapoOne deals with them. Another key issue during CapoOne's development was to make sure that data copied from the kernel into the application memory was deterministic. In this paper, we described how our first implementation of the software that manages the replay system could deadlock during replay due to dependences between the hardware and software logs, and we show how we fixed it in the second implementation.

Finally, we believe that these lessons can be of use for researchers building their hardware-assisted deterministic replay systems.

References

[1] T. C. Bressoud and F. B. Schneider, "Hypervisor-Based Fault-Tolerance," in *Proceedings of the 1995 Symposium on Operating Systems Principles*, December 1995.

[2] L. Ceze, J. M. Tuck, P. Montesinos, and J. Torrellas, "BulkSC: Bulk Enforcement of Sequential Consistency," in *International Symposium on Computer Architecture*, June 2007.

[3] G. W. Dunlap, S. T. King, S. Cinar, M. 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 (OSDI)*, December 2002.

[4] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen, "Execution Replay of Multiprocessor Virtual Machines," in *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ACM, March 2008.

[5] D. R. Hower and M. D. Hill, "Rerun: Exploiting Episodes for Lightweight Memory Race Recording," in *Proceedings of the 35th International Symposium on Computer Architecture (ISCA)*, June 2008.

[6] P. Montesinos, L. Ceze, and J. Torrellas, "DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently," in *International Symposium on Computer Architecture*, June 2008.

[7] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas, "Capo: a Software-Hardware Interface for Practical Deterministic Multiprocessor Replay," *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 44, no. 3, 2009.

[8] S. Narayanasamy, C. Pereira, and B. Calder, "Recording Shared Memory Dependencies Using Strata," in *Internationl Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006.

[9] M. Russinovich and B. Cogswell, "Replay for Concurrent Non-Deterministic Shared-Memory Applications," in *Proceedings of the 1996 Conference on Programming Language Design and Implementation*, May 1996.

[10] S. Srinivasan, S. Kandula, C. Andrews, and Y. Zhou, "Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging," in *Proceedings of the USENIX Annual Technical Conference, General Track*, 2004.

[11] M. Xu, R. Bodik, and M. D. Hill, "A "Flight Data Recorder" for Enabling Full-System Multiprocessor Deterministic Replay," in *International Symposium on Computer Architecture*, June 2003.

[12] M. Xu, M. D. Hill, and R. Bodik, "A Regulated Transitive Reduction (RTR) for Longer Memory Race Recording," in *Internationl Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006.