Capo: A Software-Hardware Interface for Practical Deterministic Multiprocessor Replay *

Pablo Montesinos Matthew Hicks Samuel T. King Josep Torrellas

University of Illinois at Urbana-Champaign
{pmontesi, mdhicks2, kingst, torrellas}@cs.uiuc.edu

Abstract

While deterministic replay of parallel programs is a powerful technique, current proposals have shortcomings. Specifically, software-based replay systems have high overheads on multiprocessors, while hardware-based proposals focus only on basic hardware-level mechanisms, ignoring the overall replay system. To be practical, hardware-based replay systems need to support an environment with multiple parallel jobs running concurrently — some being recorded, others being replayed and even others running without recording or replay. They also need to manage limited-size log buffers.

This paper addresses these shortcomings by introducing, for the first time, a set of abstractions and a software-hardware interface for practical hardware-assisted replay of multiprocessor systems. The approach, called *Capo*, introduces the novel abstraction of the *Replay Sphere* to separate the responsibilities of the hardware and software components of the replay system. In this paper, we also design and build *CapoOne*, a prototype of a deterministic multiprocessor replay system that implements Capo using Linux and simulated DeLorean hardware. Our evaluation of 4-processor executions shows that CapoOne largely records with the efficiency of hardware-based schemes and the flexibility of software-based schemes.

Categories and Subject Descriptors C [*Computer Systems Organization*]: C.0 General. **Subjects:** Hardware/Software Interfaces; C.1 [*Processor Architectures*]: C.1.0 General.

General Terms Design, Measurement, Performance.

Keywords Capo, CapoOne, Deterministic Replay, Replay Sphere.

1. Introduction

Recording and deterministically replaying execution gives computer users the ability to travel backward in time, recreating past states and events in the computer. Time travel is achieved by recording key events when the software runs, and then restoring to a previous checkpoint and replaying the recorded log to force the software down the same execution path. This alluring mechanism has enabled a wide range of applications in modern systems. First, programmers can use time travel to help debug programs [27, 8, 2, 1, 5, 24], including programs with non-determinism [22, 15], since time travel can provide the illusion of reverse execution and reverse debugging. Second, system administrators can use time travel to replay the past execution of applications looking for exploits of newly discovered vulnerabilities [13] or to inspect the actions of an attacker [14]. Third, system designers can use replay as an efficient mechanism for recreating the state of a system after a crash [3].

To maximize the utility of a deterministic replay scheme, it should have six desirable traits: (1) record an initial execution at production-run speeds, to minimize timing distortions; (2) have minimal log space requirements, to support long recording periods; (3) replay executions at similar speeds as initial executions, to maximize potential uses; (4) require only modest hardware support; (5) operate on unmodified software and binaries; and (6) given the popularity of multi-core processors, efficiently record and replay multithreaded software on multiprocessor systems.

Current software-based replay systems [3, 22, 6, 24, 7] make major strides toward achieving these goals, but fall short in one or more areas. One recent technique, ReVirt [6], implements replay support within a virtual machine monitor (VMM) and can replay entire OSes deterministically and efficiently. ReVirt works by recording all inputs from the VMM into the OS, including interrupts and other sources of non-determinism, to force the OS and applications down the same execution path when replaying. However, ReVirt only works for uniprocessors and the extensions of this technique for replaying multi-threaded applications on multiprocessors have substantial logging requirements and poor recording and replay speeds [7].

Current hardware-based replay proposals also make major advances toward achieving these goals, but are limited. The idea in these systems is to use special hardware to de-

* This work was supported in part by the National Science Foundation under grants CNS 07-20593, CNS 08-34738 and CCF 08-11268; and gifts from IBM, Intel, Sun, and the Internet Services Research Center (ISRC) of Microsoft Research.

ASPLOS'09, March 7–11, 2009, Washington, DC, USA.
Copyright © 2009 ACM 978-1-60558-406-5/09/03. . . $5.00

tect how the memory accesses of the different threads interleave during the execution — typically leveraging cache coherence protocol messages — and save such information in a log. Later, the log is used to replay, recreating the same memory access interleaving. Recent schemes include FDR [25], BugNet [19], RTR [26], Strata [18], DeLorean [17], and Rerun [11].

While these hardware-based replay proposals achieve low run time overhead, have low logging space overhead, and can cope with multiprocessor systems efficiently, they are largely impractical for use in realistic systems. The main problem is that they focus only on the hardware implementation of the basic primitives for recording and, sometimes, replay. They do not address key issues such as how to separate software that is being recorded or replayed from software that should execute in a standard manner (i.e., without being recorded or replayed), or from other software that should be recorded or replayed separately. This limitation is problematic because practical replay systems require much more than just efficient hardware for the basic operations. For example, they likely need a software component to manage large logs (on the order of gigabytes per day), and a way to mix standard execution, recorded execution, and replayed execution of different applications in the same machine concurrently. Unfortunately, providing this functionality requires a redesign of the hardware-level mechanisms currently proposed, and a detailed design and implementation of the software components that manage this hardware.

This paper addresses this problem. It presents *Capo*, the first set of abstractions and software-hardware interface for practical hardware-assisted deterministic replay of programs running on multiprocessor systems. A key abstraction in Capo is the *Replay Sphere*, which allows system designers to separate cleanly the responsibilities of the hardware and the software components. To evaluate Capo, we design and build a hardware-software deterministic replay system called *CapoOne*. It is an implementation of Capo using Linux on top of simulated DeLorean hardware [17].

Our evaluation of 4-processor executions running parallel applications shows that CapoOne largely records with the efficiency of hardware-based schemes and the flexibility of software-based schemes. Specifically, compared to the DeLorean hardware-based replay scheme, CapoOne increases the average log size by only 15% and 38% for engineering and system applications, respectively. Moreover, recording under CapoOne increases the execution time of our engineering and system applications by, on average, a modest 21% and 41%, respectively. If two parallel applications record concurrently, their execution time increase is, on average, 6% and 40% for the two classes of applications. Finally, replaying the engineering applications takes on average 80% more cycles than recording them. This is a low replay overhead, and it can be further reduced by implementing a replay-aware thread scheduler in CapoOne.

The contributions of this paper are as follows:

1. It is the first paper to design a set of abstractions and a software-hardware interface to enable practical hardware-assisted deterministic replay of multiprocessors (Capo).

2. As part of the interface, it introduces the concept of Replay Sphere for separating the responsibilities of hardware and software.

3. It is the first paper to enable hardware-assisted replay systems to mix recording, replaying, and standard execution in the same machine concurrently.

4. It builds and evaluates a prototype of a hardware-software multiprocessor replay system (CapoOne).

This paper is organized as follows. Section 2 gives a background; Sections 3 and 4 present Capo and CapoOne; Section 5 evaluates CapoOne; Section 6 presents a discussion; and Section 7 reviews related work.

2. Background on Deterministic Replay

This section gives a brief overview of how current hardware and software systems implement deterministic replay.

Deterministic replay of an initial parallel execution *E* is a new execution *E'* where, starting from the same initial state as *E* and given the same inputs at the same points in the execution, each thread (or processor, under full-system replay) executes the same instructions and the interleaving of instructions from different threads is the same. The initial execution is the *recording phase*, when the inputs to the execution, their timing, and the interleaving of instructions from different threads are collected in a *replay log*. A *replay system* is an overall system that includes hardware and software components for recording and replaying execution. Finally, *standard execution* refers to an execution that is neither being recorded nor replayed.

2.1 Hardware-Based Deterministic Replay

There are several recent schemes for hardware-based deterministic replay of multiprocessor systems [25, 19, 18, 26, 17, 11]. They propose special hardware to detect the interleaving of memory accesses or instructions from the different processors during execution, and save the key information in a log. This log is later used to replay the same interleaving.

Of these schemes, FDR [25], BugNet [19], Strata [18], and RTR [26] are able to reproduce the interleaving of memory accesses by recording (a subset of) the data dependences between processors. Data dependences trigger cache coherence protocol messages, which these schemes intercept. DeLorean [17] uses a different approach. Rather than recording data dependences, it records the total order of instructions executed. To support this operation efficiently, each processor in DeLorean executes instructions grouped into chunks, and only the total sequence of chunks is recorded. An arbiter serializes and records chunk commits. Finally, Rerun [11] uses an intermediate approach. It also detects data dependences between processors but stores sequences of instruction groups in the logs.

Most of these schemes log and replay the full system execution, including application, OS and other threads. The exception is BugNet [19], which focuses on recording only one application. BugNet presents a hardware solution for recording the inputs to the application. It does not focus on how to log the interleaving of the application threads.

Overall, these efforts focused on devising basic hardware primitives for recording and, sometimes, replay. There has been no emphasis on understanding either the modifications required on the software layers (OS, libraries, or VMM) that should drive and manage these hardware primitives, or how software and hardware should work together. Our paper addresses these specific topics.

2.2 Software-Based Deterministic Replay

There are several recent proposals for software-based deterministic replay [3, 22, 6, 24, 7]. They propose software systems that record all sources of non-determinism, such as network inputs or interrupts, and use this information to guide the software down the same execution path during replay.

As an example, Flashback [24] records and replays processes by modifying the OS to log all sources of non-determinism during recording. This includes logging all results of system calls, plus any data the kernel copies into the process. For example, if a process makes a *read* system call, Flashback records the return value of the system call and the data that the kernel copies into the read buffer. When replaying, Flashback injects this same data back into the process when it encounters this specific system call.

Although software-based schemes work well for uniprocessor systems, they have high run time overhead on multiprocessor machines because current techniques for interposing on shared-memory accesses are inefficient. Fortunately, hardware-based schemes can record shared-memory interleavings efficiently. However, none of the current software-based schemes uses these efficient hardware-based mechanisms. In our paper, we combine the best of the hardware-based and software-based schemes.

3. Capo: Abstractions and Interface

We propose *Capo*, a novel software-hardware interface and a set of abstractions for building practical schemes for hardware-assisted deterministic replay in multiprocessors. In this section, we describe the main concepts in Capo. Note that both the hardware and software components of Capo can be implemented in a variety of ways.

In our discussion, for clarity, we refer to a replay system where an OS provides the ability to record and replay processes or groups of processes. The same ideas also apply to any privileged software layer that records and replays unprivileged software running above — e.g., a VMM that records and replays VMs or groups of VMs.

3.1 Replay Sphere

To enable practical replay, Capo provides an abstraction for separating independent workloads that may be recording, replaying, or executing in a standard manner concurrently. For this abstraction to be useful, it must provide a clean separation between the responsibilities of software and hardware mechanisms. Moreover, it must also be meaningful to software components and yet still low-level enough to map efficiently to hardware.

This novel abstraction is called the *Replay Sphere* or *Sphere* for short (Figure 1). A replay sphere is a group of threads — together with their address spaces — that are recorded and replayed as a cohesive unit. All the threads that belong to the same process must run within the same replay sphere. It is possible, however, to include different processes within the same replay sphere. In our work, we call a thread that runs within a replay sphere an *R-thread*. Each R-thread is identified by an *R-threadID* and each sphere has its own set of R-threadIDs. Figure 1 shows a system with two replay spheres and four processes. In Replay Sphere 1, all R-threads belong to Process A, whereas Processes B and C run within Replay Sphere 2.

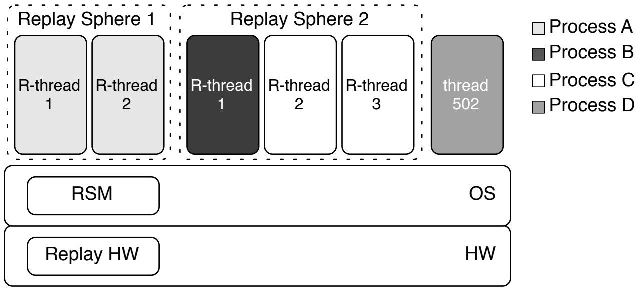


Figure 1. Architecture of Capo for an OS-level replay system. The replay system includes user-level threads running within replay spheres and a kernel-level Replay Sphere Manager (RSM) that manages the underlying replay hardware and provides the illusion of infinite amounts of replay hardware.

Execution enters a replay sphere when code from one of its R-threads begins execution; execution leaves the replay sphere when the R-thread stops execution and the OS takes over. These transitions can be explicit or implicit, depending on the type of event that occurs. Explicit transitions are triggered by privileged calls (e.g., system calls) or by a special instruction designed to enter or exit the replay sphere explicitly. Implicit transitions result from hardware-level interrupts or exceptions that invoke the OS automatically.

In Capo, the hardware records in a log the interleaving of the R-threads running within the same sphere. The software records the other sources of non-determinism that may affect the execution path of the R-threads, such as system call return values and signals.

Our decision to use software-level threads (R-threads) as the main principal in our abstractions, instead of hardware-level processors, represents a departure from current proposals for hardware-based replay. The extra level of indirection of using software-level threads instead of hardware-level processors allows us to track the same thread across different processors during recording and during replay, providing the flexibility needed to integrate cleanly with current OS scheduling mechanisms.

3.2 Separation of Responsibilities

With the replay sphere abstraction, Capo separates the duties of the software and the hardware components as shown in Table 1.

Software Duties (Per Sphere)
Assign the same R-threadIDs during recording and replay
Assign the same virtual addresses during recording and replay
Log the inputs to the replay sphere during recording
Inject the logged inputs back to the replay sphere during replay
Squash the outputs of the replay sphere during replay
Manage the buffers for the Interleaving and Sphere Input logs

Hardware Duties (Per Sphere)
Generate the Interleaving log during recording
Enforce the ordering in the Interleaving log during replay

Table 1. Separation of the duties of the software and the hardware components in Capo.

Before recording an execution, the software allocates the buffers that will store the logs. The software also identifies the threads that should be recorded together, which are referred to as R-threads. In addition, the software assigns R-threadIDs to R-threads using an algorithm that ensures that the same assignment will be performed at replay. Finally, the software must guarantee that the same virtual addresses are assigned during recording and during replay. This is necessary to ensure deterministic re-execution when applications use the addresses of variables — for example, to index into hash tables.

During the recording phase, the software logs all replay sphere inputs into a *Sphere Input Log*, while the hardware records the interleaving of the replay sphere R-threads into an *Interleaving Log*[1] (Table 1). During the replay phase, the hardware enforces the execution interleaving encoded in the Interleaving log, while the software provides the entries in the Sphere Input log back into the replay sphere and squashes all outputs from the replay sphere. During both recording and replay, the software also manages the memory buffers of the Interleaving log and the Sphere Input log by moving data to or from the file system.

Next, we describe Capo's software (Section 3.3) and hardware (Section 3.4) components.

3.3 Software Support: The Replay Sphere Manager

Capo's main software component is the Replay Sphere Manager (RSM). For each sphere, the RSM supports the duties shown in the top half of Table 1. In addition, the RSM multiplexes the replay hardware resources between spheres, giving users the illusion of supporting unlimited spheres.

Figure 2 shows a logical representation of a four-processor machine where the RSM manages three replay spheres. Each sphere has its own log, which includes the Interleaving and the Sphere Input logs. There are two replay spheres currently scheduled on the hardware — Sphere 1 recording and Sphere 3 replaying. Sphere 1 has two R-threads running on CPUs 1 and 2, while Sphere 3 has one R-thread running on CPU 4. There is a third sphere (Sphere 2) that is waiting to run due to lack of hardware — there is no free Replay Sphere Control Block (RSCB), a hardware structure that will be described in Section 3.4. As a result, CPU 3 is idle.

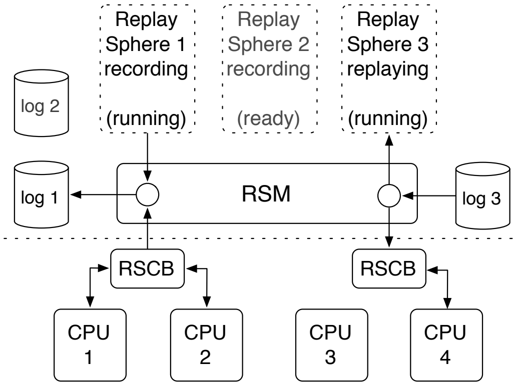

Replay Sphere 1 recording (running)
Replay Sphere 2 recording (ready)
Replay Sphere 3 replaying (running)
log 2
log 1
RSM
log 3
SW
HW
RSCB
RSCB
CPU 1
CPU 2
CPU 3
CPU 4

Figure 2. Logical representation of a system where the RSM manages three replay spheres.

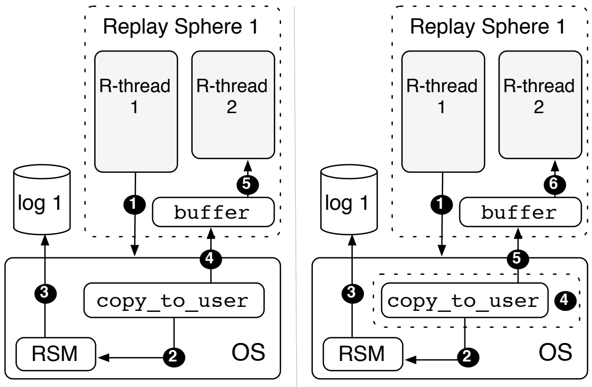
For correct and efficient recording and replay, we claim that the RSM must address three key challenges. First, it must maintain deterministic re-execution when copying data into a replay sphere. Second, it must determine when a system call needs to be re-executed and when it can be emulated. Third, it must be able to replay a sphere on fewer processors than were used during recording.

3.3.1 Copying Data into a Replay Sphere

When the OS copies data into a location within a replay sphere, extra care is needed to ensure correct deterministic replay. This is because the memory region being written to by the OS may be accessed by R-threads at the same time. In this case, the interleaving between the OS code that performs the writes and the R-threads that access the region may be non-deterministic. Figure 3(a) illustrates this case. In the figure, R-thread 1 makes a system call that will cause the kernel to copy data into the user-mode *buffer* via the *copy_to_user* function (1). Before the actual copy, the RSM logs the input to the *copy_to_user* function ((2) and (3)). The kernel then copies the data into *buffer* (4). Finally, while the copy is taking place, R-thread 2 accesses *buffer*, thus causing a race condition between *copy_to_user* and R-thread 2 (5).

To ensure deterministic inputs into a replay sphere, we consider two possibilities. First, we could inject inputs into the sphere atomically. However, ensuring atomicity requires blocking all R-threads that may access the region. As a result, this approach penalizes R-threads even if they do not access the region. A second, better, approach that we use is to include the *copy_to_user* function within the replay sphere directly. This inclusion allows the hardware to *log the interleaving* between the *copy_to_user* code and the R-threads' code. This approach is symbolically depicted in Figure 3(b). Although this approach is efficient, it creates a less clear boundary between the code running within a

[1] This log is often called the memory-interleaving, memory-ordering, or processor-interleaving log in the literature [25, 18, 26, 17, 11].



(a) (b)

Figure 3. Race condition between the OS *copy_to_user* function and R-thread 2 (a). The data race is avoided by including *copy_to_user* in the replay sphere (b).

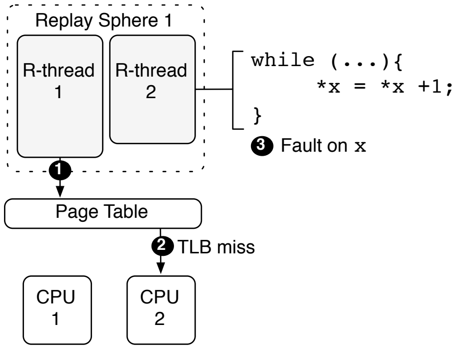
replay sphere and the code running outside of it — since the *copy_to_user* function runs within both the replay sphere and the OS. Still, we use this approach because most system calls cause inputs into replay spheres, and blocking all R-threads during these system calls would be inefficient.

3.3.2 Emulating and Re-Executing System Calls

In Capo, the RSM emulates the majority of system calls during replay. To do this, the RSM first logs the system call during recording. Then, during replay, the RSM squashes the system call and injects its effects back into the replay sphere, thus ensuring determinism.

However, the RSM needs to re-execute some system calls during replay. These are system calls that modify select state outside of the replay sphere, which affects R-threads running within the sphere. They include system calls that modify the address space of a process, process management system calls (e.g., *fork*), and signal handling system calls. By re-executing these system calls, we modify external states directly during replay, which would require substantial additional functionality in the RSM to emulate correctly.

As a result of re-executing system calls, we must ensure that external state changes affect R-threads deterministically. One subtle issue that the RSM must handle arises from modifications to shared address spaces. R-threads form *implicit dependences* when one R-thread changes the mapping or the protection of a shared address space, and another R-thread accesses this changed address space. Figure 4 shows an example where one R-thread changes the protection of a page that a second R-thread is using. In the figure, R-thread 1 and R-thread 2 run on CPU 1 and CPU 2, respectively, and share the same page table. In the example, R-thread 1 first issues an *mprotect* system call to change the protection of a page, and the system call modifies the page table (1). Eventually, both CPUs will cache the new protection (2). After CPU 2 caches the new protection, the effects of the page table modification will become visible to R-thread 2, and R-thread 2 suffers a page fault (3). To ensure deterministic replay, the interleaving between the page table modification and the use of the affected addresses must be recorded and reproduced faithfully during replay.



```
while (...){
      *x = *x +1;
}
```

Figure 4. Example of potential non-determinism due to an implicit dependence.

This issue largely disappears in replay schemes that use full-system recording and replay [25, 18, 26, 17, 11]. This is because they naturally record most OS actions related to implicit dependencies (e.g., page fault handling). However, in Capo, code that carries out address space changes resides outside of the replay sphere and, therefore, is not invoked deterministically. As a result, the address space modifications may not be injected into the replay at the *precise* same point of execution (e.g., the exact same loop iteration in Figure 4), thus violating the correctness of our replay system.

To solve this problem, Capo gives the RSM the ability to explicitly express implicit dependencies to the hardware. Interactions like the one in the example are recorded in the log and can be replayed deterministically.

3.3.3 Replaying with a Lower Processor Count

Resource availability in a system is unpredictable and can change from the time recording takes place to the time when the user replays the execution. As a result, the system may have fewer processors available during replay. In this case, an R-thread that the hardware needs to replay next may be unable to run because it is not scheduled on any processor.

To cope with this potential source of inefficiency, we consider three possible solutions. First, we could rely on hardware support to detect when an R-thread that is currently unassigned to a processor needs to run, and trigger an interrupt. This approach provides software with immediate notification on replay stalls, but requires additional hardware support. Second, the RSM could periodically inspect the Interleaving log to predict which R-threads need to be run in the near future, and schedule them accordingly. However, this approach requires the hardware-level log to include architecturally-visible states, making its format less flexible. Also, even with this careful planning, the OS scheduling algorithms may override the RSM. The third approach is for the RSM to simply ensure that all the R-threads get frequent and fair access to the CPUs. In this case, there will be some wasted time during which all of the running R-threads may be waiting for a preempted R-thread. Our design uses this approach because it is simple and has low overhead.

3.4 Hardware Support

Capo's hardware components are shown in Table 2. They are a structure called *Replay Sphere Control Block (RSCB)* per *active* replay sphere (i.e., per sphere that is currently using at least one processor), a structure called *R-Thread Control Block (RTCB)* per processor that is currently being used by a replay sphere, and an interrupt-driven buffer interface. These components can be implemented in different ways to support any of the proposals for hardware-based deterministic replay such as FDR [25], BugNet [19], RTR [26], Strata [18], DeLorean [17], or Rerun [11].

Replay Sphere Control Block (RSCB): <per-sphere structure>
Mode register: Current execution mode of the replay sphere
Base, Limit, and Current registers: Pointers to the replay sphere log
R-Thread Control Block (RTCB): <per-processor structure>
R-ThreadID register: ID of the R-thread running on the processor
RSID register: ID of the replay sphere using the processor
Interrupt-driven buffer interface

Table 2. Capo's hardware components.

The RSCB is a hardware structure that contains information about an active replay sphere. When a sphere is not using any processor, like Sphere 2 in Figure 2, the state in the RSCB is saved to memory. An ideal machine configuration would have as many RSCBs as processors. Having more RSCBs than processors does not make sense.

The RSCB consists of a *Mode* register and log pointer registers. The Mode register specifies the sphere's execution mode: *Recording*, *Replaying*, or *Standard*. The log pointer registers are used to access the sphere's log. At a high level, they need to enable access to the *Base* of the log, its *Limit*, and its *Current* location — where the hardware writes to (during recording) or reads from (during replay). The Current pointer is incremented or decremented automatically in hardware. Depending on the log implementation, there may be multiple sets of such pointers.

The per-processor RTCB consists of two registers. The first one contains the R-threadID of the R-thread that is currently running on the processor. The R-threadID is per replay sphere. It is short and generated deterministically in software for each R-thread in the replay sphere. It starts from zero and can reach the maximum number of R-threads per replay sphere. The R-threadID is saved in the Interleaving log, tagging the log entries that record events for that R-thread. The second RTCB register contains the ID of the replay sphere that currently uses the processor (*RSID*). The hardware needs to know, at all times, which processors are being used by which replay spheres because each replay sphere interacts with a different log.

The size of the R-threadID is given by the maximum number of R-threads that can exist in a replay sphere. Such number can be high because multiple R-threads can time-share the same processor. However, given that log entries store R-threadIDs, their size is best kept small. In general, the size of the RSID register is determined by the number of concurrent replay spheres that the RSM can manage. Such number can potentially be higher than the number of RSCBs, since multiple replay spheres can time-share the same hardware resources.

Depending on the implementation, the RTCB and RSCB structures may or may not be physically located in the processors — they may be located in other places in the machine. At each context switch, privileged software updates them if necessary.

Finally, Capo also includes an interrupt-driven interface for the Interleaving log. Such a log may or may not be built using special-purpose hardware. However, in all of the hardware-based deterministic replay schemes proposed, it is at least *filled* in hardware, transparently to the software. In Capo, we propose that, to use modest memory resources, it be assigned a fixed-size memory region and, when such a region is completely full (during recording) or completely empty (during replay), an interrupt be delivered. At that point, a module copies the data to disk and clears the region (during recording) or fills the region with data from disk (during replay) and restarts execution.

4. CapoOne: An Implementation of Capo

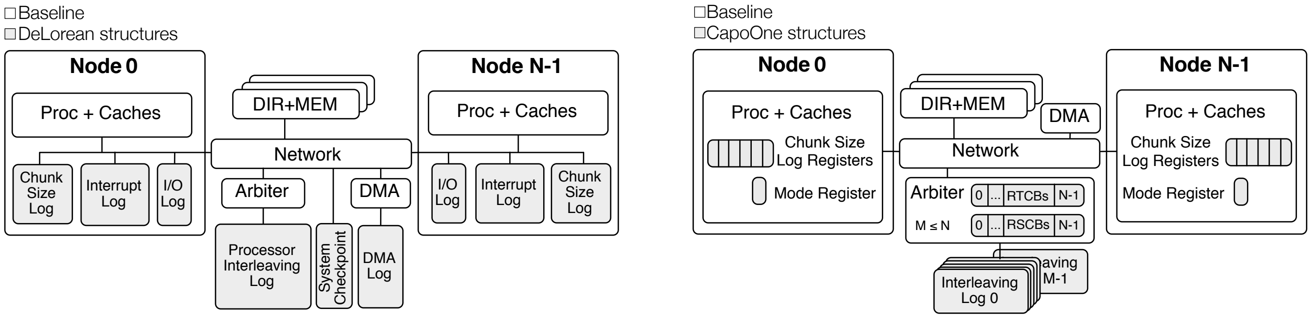
To evaluate Capo, we design and build a prototype of a deterministic replay system for multiprocessors called *CapoOne*. It is an implementation of Capo using Linux and simulated DeLorean replay hardware [17]. It records and replays user-level processes and not the OS. In this section, we discuss the software and hardware implementation in turn.

4.1 Software Implementation

We modify the 2.6.24 Linux kernel by adding a data structure per replay sphere called *rscb_t* that stores the hardware-level replay sphere context. We also add a per R-thread data structure called *rtcb_t* that stores the R-threadID and replay sphere information for the R-thread. The RSM manages these structures by saving and restoring them into hardware RSCBs and RTCBs during context switches.

We change the kernel to make sure that *copy_to_user* is the only kernel function used to copy data into replay spheres. Then, we modify *copy_to_user* so that it records all inputs before injecting them into the replay spheres. To help make *copy_to_user* deterministic, we inject data one page at a time, and make sure that no interrupt or page fault can occur during the copy. We also change the kernel to track implicit dependences. We make page table modifications and the resulting TLB flushes atomic, to avoid race conditions with R-threads running on other CPUs.

Our RSM tracks processes as they run using the Linux *ptrace* process tracing mechanism. Ptrace gives processes the ability to create child processes, receive notification on key events, and access arbitrary process states as the child process runs. Using the ptrace mechanism, we implement much of our RSM in user mode.



(a) (b)

Figure 5. Multiprocessor with the DeLorean hardware as presented in the DeLorean paper [17] (a), and as implemented in CapoOne (b).

4.2 Hardware Implementation

In CapoOne, we implement the hardware interface of Table 2 for DeLorean. This section describes the implementation in detail and, to give more insight, Section 4.2.1 outlines the implementation for FDR [25] and similar schemes.

Figure 5(a) shows a multiprocessor with the DeLorean-related hardware. CapoOne implements the interface of Table 2 mostly in the arbiter module. In addition, there are a few other changes to make to the DeLorean architecture of Montesinos *et al.* [17] because that architecture was a full-system replayer, while CapoOne is not. We consider the two issues separately.

In CapoOne, the RTCB and RSCB structures are placed in the arbiter module. Specifically, as shown in Figure 5(b), the arbiter contains an array of N RTCBs and an array of N RSCBs, where N is the number of processors in the machine. Each RTCB corresponds to one processor. If the processor is currently used by a replay sphere, its RTCB is not null. The RTCB contains the R-threadID of the R-thread currently running on the processor and, in the RSID field, a pointer to the entry in the RSCB array corresponding to the replay sphere currently using the processor. In this implementation, therefore, the size of the RSID field is given by the maximum number of concurrent active replay spheres. Finally, each active replay sphere has a non-null RSCB. Each RSCB contains the sphere's mode, and the current, base, and limit pointers to the sphere's Interleaving log. In Figure 5(b), we show the case when there are M active replay spheres and, therefore, M Interleaving logs. Note that $M \leq N$. The Interleaving log is DeLorean's Processor Interleaving log.

For performance reasons, each node also has some special hardware registers, namely the Mode register and the Chunk Size Log registers. The former contains the mode of the sphere that is currently using the processor; the latter contains the top of DeLorean's Chunk Size log for the R-thread currently running on the processor.

With this hardware, every time that the OS schedules an R-thread on a processor, the OS potentially updates the node's Mode and Chunk Size Log registers, the processor's RTCB, and the RSCBs of the spheres for this R-thread and for the preempted R-thread — recall that spheres that currently use no processor have a null RSCB. As the R-thread executes, it writes or reads the Chunk Size Log registers depending on whether the Mode register indicates Recording or Replaying mode, respectively. If the Chunk Size Log registers are used up before the OS scheduler is invoked again, an interrupt is delivered, and the OS saves their contents and clears them (during recording) or loads then with new data (during replay). The Chunk Size log is used very infrequently, so a few registers are enough.

During execution, when the arbiter receives a message from a processor requesting a chunk commit, the arbiter checks the RTCB for the processor. From that RTCB, it reads its current R-threadID and, through the RSID pointer, the mode of the sphere that currently uses the processor. If the mode is Standard, no action is taken. Otherwise, the Interleaving log is accessed. If the mode is Recording and the chunk can commit, the hardware adds an entry in the Interleaving log and tags it with the R-threadID. If, instead, the mode is Replaying, the hardware reads the next Interleaving log entry and compares the entry's tag to the R-threadID. If they are the same, the chunk is allowed to commit and the Interleaving log entry is popped. There are no changes to the encoding of messages to or from the arbiter.

Since the DeLorean architecture of Montesinos *et al.* [17] is a full-system replayer and CapoOne is not, we need to make additional changes to the original architecture. First, there is no need for the hardware-generated logs for DMA, I/O, and interrupts shown in Figure 5(a). In CapoOne, during recording, privileged software records all the inputs to the replay sphere in the Sphere Input log; during replay, privileged software plays back these log entries at the appropriate times. This Sphere Input log is invisible to the hardware; it is managed by the software. For this reason, we do not show it in Figure 5(b). Moreover, since the checkpointing is now per replay sphere, it is likely performed and managed by privileged software and, therefore, it is not shown in the figure either. Finally, since we only record chunks from replay spheres, the algorithm for creating chunks changes slightly. Specifically, at every system call, page fault, or other OS invocation, the processor terminates the current chunk and commits it. Interrupts are handled slightly differently: for ease of implementation, they squash the current chunk and execute right away. In all cases, as soon as the OS completes execution, a new application chunk starts.

4.2.1 Hardware Implementation for FDR-like Schemes

We now outline how the hardware interface of Table 2 can be implemented for FDR [25] and similar replay schemes. The idea is to tag cache lines with R-thread and, potentially, sphere information. In this case, each processor has R-threadID, RSID, and Mode registers. These registers are updated every time that the OS schedules a new thread on the processor. During recording, when a processor accesses a line in its cache, in addition to tagging it with its current dynamic instruction count, it also tags it with its current R-threadID and, potentially, RSID. This marks the line as being accessed by a given R-thread of a given replay sphere. The line can remain cached across context switches. At any time, when a data dependence between two processors is detected, the message sent from the processor at the dependence source to the processor at the dependence destination includes the R-threadID and, potentially, the RSID of the line. The receiver processor then selects a local log based on its own current R-threadID and RSID. In that log, it stores a record composed of the R-threadID of the incoming message plus the dynamic instruction counts of the source and destination processors.

5. Evaluation of CapoOne

This section discusses our evaluation of CapoOne. We first describe our experimental setup. Then, we evaluate different aspects of CapoOne: log size, hardware characteristics, and performance overhead during recording and replay.

5.1 Evaluation Setup

To evaluate CapoOne, we use two different environments, which we call *Simulated-DeLorean* and *Real-Commodity-HW*. Both environments run the same Ubuntu 7.10 Linux distribution with a 2.6.24 kernel that includes CapoOne's software components. In the *Simulated-DeLorean* environment, we use the Simics full-system architecture simulator enhanced with a detailed model of the DeLorean hardware. We model a system with four x86 processors running at 2 GHz, 1,000 instructions per chunk, DeLorean's *OrderOnly* logging method, and the latency and bandwidth parameters used in the DeLorean paper [17]. We use this environment to evaluate a complete CapoOne system.

In the *Real-Commodity-HW* environment, we use a 4-core HP workstation with an Intel Core 2 Quad processor running at 2.5GHz with 3.5GB of main memory. We use this environment to evaluate the software components of CapoOne on larger problem sizes than are feasible with *Simulated-DeLorean*, allowing us to take more meaningful timing measurements.

We evaluate CapoOne using ten SPLASH-2 applications configured to execute with four threads, a web server, and a compilation session. We call the SPLASH-2 applications *engineering applications* and the rest *system applications*. We run all the applications from beginning to end. The SPLASH-2 applications use the standard input data sizes for the *Simulated-DeLorean* environment and larger sizes for the *Real-Commodity-HW* environment. The web server application is an *Apache* web server exercised with a client application. In the *Real-Commodity-HW* environment, it downloads 150MB of data via 1KB, 10KB, and 100KB file transfers, and uses five or ten concurrent client connections depending on the experiment. In the *Simulated-DeLorean* environment, we run the same experiments except that we only download 50MB of data. We call the applications *apache-1K*, *apache-10K*, and *apache-100K* depending on the file transfer size. The compilation application for the *Real-Commodity-HW* environment is a compilation of a 2.6.24 Linux kernel using the default configuration values. For the *Simulated-DeLorean*, it is a compilation of the SPLASH-2 applications. We run the compilation with a single job (*make*) or with four concurrent jobs (*make-j4*).

Our experimental procedure consists of a warm-up run followed by six test runs. We report the average of the six test runs. In all experiments, the standard deviation of our results is less than three percent. All log sizes we report are for logs compressed using *bzip*.

5.2 Log Size

Figure 6 shows the size of CapoOne's logs, namely the Interleaving plus the Chunk Size logs (bottom) and the Sphere Input log (top), measured in bits per committed kilo-instruction. Although not seen in the figure, the contribution of the Chunk Size log is practically negligible. In the figure, SP2-G.M. is the geometric mean of SPLASH-2, while SYS-G.M. is the geometric mean of the system applications. This experiment uses the *Simulated-DeLorean* environment.

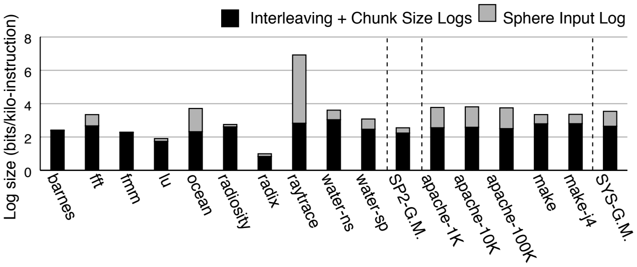


Figure 6. CapoOne's log size in bits per kilo-instruction.

The figure shows that CapoOne generates a combined log of, on average, 2.5 bits per kilo-instruction in the engineering applications and 3.8 bits in the system applications. In most applications, the Interleaving log contributes with most of the space. This is especially true for the engineering applications because most of them interact with the OS infrequently. The one exception is *raytrace*, which issues more file system reads than the other engineering applications, thus requiring a larger Sphere Input log. As expected, the system applications require larger Sphere Input logs because they execute more system calls.

Overall, we find that the size of the Interleaving plus Chunk Size logs is comparable to the size reported by Montesinos *et al*. [17]. When comparing the size of these logs to the total logging overhead of CapoOne, we see that the

Sphere Input log increases the average logging requirements of the hardware only modestly: by 15% and 38% for the engineering and system applications, respectively.

5.3 Hardware Characterization

Table 3 characterizes the CapoOne hardware during recording under the *Simulated-DeLorean* environment. The first two columns show the average number of dynamic instructions per chunk and the percentage of all chunks that attain the maximum chunk size (full size chunks). The *Truncated Chunks* columns show the three main reasons why the hardware had to truncate the chunk: cache overflows, system calls, and page faults.

Application	Avg. Chunk Size (# of insts)	Full Size Chunks (%)	Truncated Chunks		
			Cache Overflows (%)	System Calls (%)	Page Faults (%)
barnes	999	99.8	40.7	29.7	29.4
fft	981	97.8	0.0	14.4	85.4
fmm	998	99.6	30.4	6.4	63.0
lu	996	99.6	0.3	53.8	45.7
ocean	977	97.8	0.9	63.7	35.3
radiosity	994	99.1	5.2	44.8	49.9
radix	982	95.1	78.7	1.9	19.3
raytrace	993	99.0	9.3	38.5	52.1
water-ns	953	91.6	85.9	0.8	13.1
water-sp	989	97.2	93.7	1.9	4.3
SP2-AVG	986	97.6	34.5	25.5	39.7
apache-1K	785	65.9	1.3	93.2	5.3
apache-10K	781	66.2	1.1	92.3	6.6
apache-100K	773	65.0	0.9	93.4	5.5
make	993	96.5	16.6	54.9	28.3
make-j4	993	96.6	14.3	58.2	27.6
SYS-AVG	865	78.5	6.7	78.4	14.6

Table 3. CapoOne's hardware characterization.

The data shows that, on average, 98% and 97% of the chunks in the engineering and compilation applications, respectively, are full-sized. For the web server applications, only 66% of the chunks reach full size because of the high system call frequency in the Apache application. In theory, applications with short chunks in Table 3 should match those with long Interleaving plus Chunk Size logs in Figure 6. However, the correlation is not that clear due to the effect of log compression.

5.4 Performance Overhead during Recording

We are interested in CapoOne's execution time overhead during recording in two situations, namely when there is a single replay sphere in the machine and when there are multiple. For these experiments, we use the *Real-Commodity-HW* environment. This is because its larger application problem sizes help us get more meaningful results. Moreover, all of the recent works on hardware-based deterministic replay schemes indicate that the execution time overhead caused by the recording hardware is negligible [25, 18, 26, 17, 11].

We first consider a single sphere in the machine. Figure 7 shows the execution time of the applications running on four processors when they are being recorded. The bars are normalized to the execution time of the same applications under standard execution — therefore, execution time equal to 1.0 means that there is no CapoOne overhead. The figure shows that, on average, recording under CapoOne increases the execution time of our engineering and system applications by 21% and 41%, respectively. This is a modest overhead that should affect the timing of concurrency bugs little.

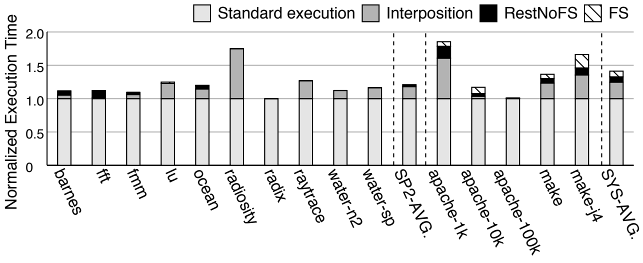


Figure 7. Execution time overhead of CapoOne during recording for a single replay sphere in the machine.

Figure 7 breaks down this overhead into three basic parts. One is the interposition overhead, namely the overhead caused by the ptrace mechanism to control the execution of the application. The next one is the rest of the RSM and kernel overhead without exercising the file system (*RestNoFS*). Finally, there is the overhead of storing the logs into the file system (*FS*). From the figure, we see that practically all of the overhead in the engineering applications, and most of the one in the system ones comes from interposition overhead. Thus, improving the performance of interposition will likely improve the results for these applications significantly. The system applications also have noticeable overhead due to *RestNoFS* and *FS*. In these applications, the OS is invoked more frequently, and there is more file system activity.

We now measure CapoOne's execution time overhead when two spheres record simultaneously. In this experiment, we measure *pairs* of applications. A pair consists of two instances of the same application running concurrently on the four-processor machine with two threads each. Consequently, we change the compilation to run with two concurrent jobs (*make-j2*). For the Apache applications, we cannot always control the number of threads and, therefore, there may be more threads than processors.

We test three scenarios. In the first one, both applications run under standard execution — therefore, there is no CapoOne overhead. In the second one, one application runs under standard execution and the other is being recorded. In the third scenario, both applications are being recorded. Figure 8 shows the resulting normalized execution times. For each application, we show three pairs of bars, where each pair corresponds to one of the three scenarios described, in order, and the two bars in a pair correspond to the two applications. We will call these bars Bar 1 to Bar 6, starting from the left. For a given application, all the bars are normalized to the execution time of a single, 2-threaded instance of the application running alone in the machine.

We make two observations. First, consider the second scenario, where one sphere is not being recorded and one is (Bars 3-4 of each application). We see that, generally, the

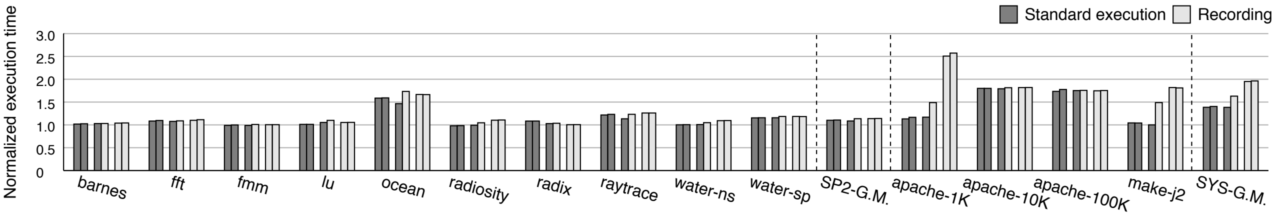


Figure 8. Execution time overhead of CapoOne during recording when two replay spheres share the machine.

two spheres do not induce overhead on each other. To see this, note that, on average, Bar 3 is no higher than Bar 2. Moreover, the ratio of Bar 4 to Bar 3 is even lower than the height of the bars in Figure 7, where there was a single sphere in the whole machine.

For the second observation, we focus on Bars 5-6, where both spheres are being recorded. Comparing these two bars to Bars 1-2, we see that recording two parallel applications concurrently increases their execution time over standard execution by an average of 6% and 40% for engineering and system applications, respectively. This is also a very modest overhead. The overhead is high in only two system applications, namely *apache-1K* and *make-j2*. This result mirrors the overheads of *apache-1K* and *make-j4* (which is roughly equivalent to two concurrent *make-j2* instances) from the single-sphere experiments.

5.5 Performance Overhead during Replay

Finally, we measure CapoOne's performance overhead during replay. We perform two sets of experiments. In the first one, we record and replay the SPLASH-2 applications using the *Simulated-DeLorean* environment. Then, since the processes in our Apache applications do not share data with each other, we note that our RSM can replay the Apache applications without assistance from the DeLorean hardware. Consequently, in our second experiment, we record and replay the *apache-1K*, *apache-10K*, and *apache-100K* applications using the *Real-Commodity-HW* environment. Finally, we are unable to provide replay performance results for the compilation applications at this point.

Figure 9 compares the execution of the SPLASH-2 applications during recording and during replay. For simplicity, our simulator models each instruction to take one cycle to execute, and reports execution time in number of cycles taken to execute the application. Consequently, for each application, the figure shows two bars, corresponding to the number of cycles taken by recording (*Rc*) and by replaying (*Rp*) the application. In each application, the bars are normalized to *Rc*. The bars show the *Execution* cycles and, in the replay bars, the cycles that a processor is stalled, having completed a chunk and waiting for its turn to commit it, according to the order encoded in the Interleaving log. Such stall is broken down based on whether the completed chunk contains user code (*User Stall*) or code from the kernel or RSM (*Kernel+RSM Stall*). The latter occurs when, in the execution of the *copy_to_user* function, code from the RSM or the kernel needs to be ordered relative to user code.

The figure shows that, in all applications, replay takes longer than recording. On average, the replay run takes 80% more cycles than the recording run. The *Execution* cycles generally vary little between the two runs — although the RSM and kernel activity can be different in two runs, for example when system calls are emulated rather than reexecuted. However, the main difference between the *Rc* and *Rp* bars is the presence of stall cycles during replay. Such stall cycles are dominated by *User Stall*.

The stall cycles are substantial because, often, the R-thread that needs to commit the next chunk is not even running on any processor. This occurs even though the application has no more R-threads than processors in the machine. A better thread scheduling algorithm that tries to schedule all the threads of the application concurrently should reduce these cycles substantially. Overall, we consider that our initial version of CapoOne replay has a reasonably low performance overhead, and that future work on thread scheduling will reduce the overhead more.

We now consider the web server applications. Recall that we run them using the *Real-Commodity-HW* environment and, therefore, measure their performance in elapsed time. Table 4 shows the execution time of the replay relative to the execution time of a standard run. On average for the Apache applications, replay only takes 54% of the time taken by the standard execution run. In *apache-1K*, replay takes nearly as much time as the standard execution run. This is expected because *apache-1K* is both CPU intensive and issues system calls frequently. However, in *apache-10K* and *apache-100K*, CapoOne replays execution significantly faster than the standard execution run. The reason is that these applications are both network bound and have a low CPU utilization. When CapoOne replays these applications, the RSM injects the results of network system calls into the replay sphere *without* accessing the network, resulting in a faster execution. This phenomenon is related to *idle time compression* [6], where any CPU idle time is skipped during replay, causing replay to outperform initial executions that have significant amounts of idle time. Overall, CapoOne replays system applications at high speed.

Application	Normalized Replay Execution Time
apache-1K	0.92
apache-10K	0.57
apache-100K	0.14
AP-AVG.	0.54

Table 4. Replay performance of the web server applications.

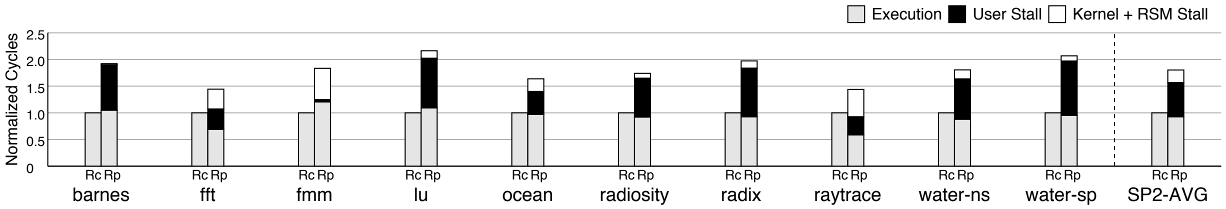


Figure 9. Normalized number of cycles taken by the SPLASH-2 applications during recording (*Rc*) and replay (*Rp*).

6. Discussion

The development of the CapoOne hardware-software prototype has given us some insights into key issues in experimental replay systems. One such issue relates to RSM implementation. For simplicity, we decided to implement the RSM in user mode, using the *ptrace* process tracing mechanism of Linux to interpose on recording and replaying processes. Although this decision made our initial prototype easier to implement, our first iteration added substantial overhead during recording. To reduce this overhead, we included two optimizations that buffer data for system calls within the kernel, thus minimizing costly traps to the RSM. These optimizations improved performance significantly, but increased the complexity of the kernel-mode portion of the RSM. Looking back, the optimizations were complicated enough that a kernel-mode RSM implementation may have been both cleaner and higher performing.

One surprising aspect of our RSM implementation is that we did not need to use a versioning file system [10, 23] to include the hard disk state within our checkpoints. To reduce log sizes, software-only replay systems commonly include disk state within checkpoints and allow replaying software to recreate disk state without logging it explicitly in the same way replaying software recreates memory state [6]. However, experiments showed that the hardware-level Interleaving log accounts for the majority of data within our replay logs, thus obviating the need to introduce the complexities of including disk state within our checkpoints.

On the hardware side, we found that using DeLorean as the underlying recording hardware scheme made the transition from per-processor replay to per-thread replay surprisingly easy. This is due to DeLorean's unique way of creating the log as a total order of program chunks. This method easily lends itself to combining per-thread chunk sequences rather than per-processor chunk sequences. However, since we wanted to record and replay applications rather than a full system, we had to modify DeLorean's chunking policies substantially. In particular, chunks had to be truncated at system calls and page faults. This added non-trivial complexity.

7. Related Work

Several software-based deterministic replay schemes have been proposed. In Recap [21], the compiler inserts code for every read that may access a shared memory location. Agora [9] uses write-once memory and a history log for maintaining the correct state of a program. Instant Replay [16] logs execution by using Reader-Writer locking semantics for accessing shared memory. DejaVu [12] records scheduling invocations of a Java Virtual Machine to support deterministic replay of multithreaded Java applications on uniprocessors. Russinovich and Cogswell [22] propose modifying the OS scheduler of a uniprocessor so that it logs each thread switch. Flashback [24] uses shadow processes to enable efficient rollback of the memory state of a process while a lightweight logger records the process interaction with the OS.

Other researchers propose using virtual machine monitors to replay entire virtual machines. Bressoud and Schneider [3] modify a hypervisor to replay virtual machines on Alpha-based computer systems, and the ReVirt project [6] modifies a VMM to replay virtual machines on a modern x86-based computer system. Dunlap *et al.* extend ReVirt to work for multiprocessor virtual machines [7].

Several hardware-based schemes have been proposed for deterministic multiprocessor replay. The Flight Data Recorder (FDR) [25] is a full-system recorder for directory-based multiprocessors. It augments the hardware in the caches and in the cache coherence protocol to identify and record coherence messages between processors. It implements a hardware version of Netzer's Transitive Reduction (TR) optimization [20] to reduce the number of recorded dependences. Xu *et al.* [26] extend FDR by introducing Regulated Transitive Reduction (RTR). This scheme introduces artificial dependencies so that Netzer's TR can eliminate additional dependencies. BugNet [19] records user processes by storing the result of all load instructions within a hardware-based dictionary. Strata [18] maintains a per-processor counter that records the number of memory operations issued by a processor. Before a dependence-forming memory operation completes, Strata logs the memory operation counts of all the processors. Rerun [11] does not record data dependences in its log. Instead, it records the number of instructions executed by a processor between dependences. The section of dynamic instructions without dependences is called an Episode.

DeLorean [17] is the scheme used in this paper. It uses the BulkSC execution mode proposed by Ceze *et al.* [4], where processors continuously execute large blocks of instructions atomically and in isolation. This execution mode can be thought of as processors executing software-invisible transactions all the time. Each of these blocks is called a Chunk, and includes a fixed number of dynamic committed instructions (e.g., 1,000).

Every time that a processor completes a chunk, it sends a request to commit to a central module called the Arbiter. The request includes a hardware signature summarizing the footprint of the chunk. The arbiter uses this signature to decide immediately whether the chunk can commit. To record the execution of a parallel program, DeLorean only needs to record the ordered list of chunk commits.

Under the *OrderOnly* recording mode, the arbiter logs the ID of the processors committing the chunks in a *Processor Interleaving* log. There are a few, somewhat rare, events that truncate a currently-running chunk and force it to commit early. They include cache overflow. For these "short" chunks, a per-processor *Chunk Size* log stores the chunk size and their position in the sequence of local chunks.

During replay, all processors execute normally, creating chunks and requesting the arbiter to let them commit the chunks. The arbiter uses the Processor Interleaving log to delay or to accept the commit of individual chunks. Moreover, each processor also reads its Chunk Size log to identify the position and size of small chunks.

8. Conclusions

Current proposals for hardware-based deterministic replay of multiprocessors focus only on the implementation of the basic primitives for recording and, sometimes, replay. A practical system additionally requires a software component that interfaces with these primitives, manages large logs, and enables the concurrent execution of multiple parallel applications that mix standard, recorded, and replayed execution. To solve this problem, this paper introduced Capo, the first set of abstractions and software-hardware interface for deterministic replay of multiprocessors. A key abstraction in Capo is the Replay Sphere, which separates the responsibilities of the hardware and the software. To evaluate Capo, we built a prototype called CapoOne based on Linux and DeLorean.

We evaluated CapoOne with 4-processor executions. Compared to the DeLorean hardware-only scheme, CapoOne increased the average log size by only 15% and 38% for engineering and system applications, respectively. Moreover, recording under CapoOne increased the execution time of the engineering and system applications by, on average, only 21% and 41%, respectively. If two parallel applications record concurrently, their execution time increase was, on average, 6% and 40% for the two classes of applications. Finally, replaying the engineering applications took on average a modest 80% more cycles than recording them. With these modest overheads, we argue that deterministic replay of multiprocessor systems is a powerful and practical tool to debug concurrency bugs.

References

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford, "An Execution-Backtracking Approach to Debugging," *IEEE Software*, vol. 8, May 1991.

[2] B. Boothe, "Efficient Algorithms for Bidirectional Debugging," in *PLDI*, June 2000.

[3] T. C. Bressoud and F. B. Schneider, "Hypervisor-Based Fault-Tolerance," in *SOSP*, Dec. 1995.

[4] L. Ceze, J. M. Tuck, P. Montesinos, and J. Torrellas, "BulkSC: Bulk Enforcement of Sequential Consistency," in *ISCA*, June 2007.

[5] S.-K. Chen, W. K. Fuchs, and J.-Y. Chung, "Reversible Debugging Using Program Instrumentation," *IEEE Transactions on Software Engineering*, vol. 27, August 2001.

[6] 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 *OSDI*, Dec. 2002.

[7] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen, "Execution Replay of Multiprocessor Virtual Machines," in *VEE*, Mar. 2008.

[8] S. I. Feldman and C. B. Brown, "IGOR: A System for Program Debugging Via Reversible Execution," in *PADD*, Nov. 1988.

[9] A. Forin, "Debugging of Heterogeneous Parallel Systems," in *PADD*, May 1988.

[10] D. Hitz, J. Lau, and M. Malcolm, "File System Design for an NFS File Server Appliance," in *USENIX Technical Conference*, Jan. 1994.

[11] D. R. Hower and M. D. Hill, "Rerun: Exploiting Episodes for Lightweight Memory Race Recording," in *ISCA*, June 2008.

[12] J. Choi and H. Srinivasan, "Deterministic Replay of Java Multithreaded Applications," in *SPDT*, Aug. 1998.

[13] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen, "Detecting Past and Present Intrusions Through Vulnerability-Specific Predicates," in *SOSP*, Oct. 2005.

[14] S. T. King and P. M. Chen, "Backtracking Intrusions," in *SOSP*, Oct. 2003.

[15] S. T. King, G. W. Dunlap, and P. M. Chen, "Debugging Operating Systems with Time-Traveling Virtual Machines," in *USENIX Technical Conference*, April 2005.

[16] T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *IEEE Transactions on Computers*, vol. 36, April 1987.

[17] P. Montesinos, L. Ceze, and J. Torrellas, "DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently," in *ISCA*, June 2008.

[18] S. Narayanasamy, C. Pereira, and B. Calder, "Recording Shared Memory Dependencies Using Strata," in *ASPLOS*, Oct. 2006.

[19] S. Narayanasamy, G. Pokam, and B. Calder, "BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging," in *ISCA*, June 2005.

[20] R. H. B. Netzer, "Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs," in *PADD*, May 1993.

[21] D. Z. Pan and M. A. Linton, "Supporting Reverse Execution for Parallel Programs," in *PADD*, Jan. 1988.

[22] M. Russinovich and B. Cogswell, "Replay for Concurrent Non-Deterministic Shared-Memory Applications," in *PLDI*, May 1996.

[23] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir, "Deciding When to Forget in the Elephant File System," in *SOSP*, Dec. 1999.

[24] S. Srinivasan, S. Kandula, C. Andrews, and Y. Zhou, "Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging," in *USENIX Technical Conference*, 2004.

[25] M. Xu, R. Bodik, and M. D. Hill, "A "Flight Data Recorder" for Enabling Full-System Multiprocessor Deterministic Replay," in *ISCA*, June 2003.

[26] M. Xu, R. Bodik, and M. D. Hill, "A Regulated Transitive Reduction (RTR) for Longer Memory Race Recording," in *ASPLOS*, Oct. 2006.

[27] M. V. Zelkowitz, "Reversible Execution," *Communications of the ACM*, vol. 16, Sept. 1973.