# PRACTICAL TIME TRAVEL OF MULTIPROCESSOR SYSTEMS

BY

PABLO MONTESINOS ORTEGO

B.S., Universidad de León, 2001
M.S., University of Illinois at Urbana-Champaign, 2005

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2009

Urbana, Illinois

Doctoral Committee:

Professor Josep Torrellas, Chair
Assistant Professor Sam King, Co-Director of Research
Professor Marc Snir
Professor Wen-mei Hwu
Professor Yuanyuan Zhou
Professor Christoph von Praun, Georg-Simon-Ohm University

# ABSTRACT

With the arrival of multicore chips as the commodity architecture for a wide range of platforms, there is a growing pressure to make parallel programming the norm. Unfortunately, most current programmers find parallel programming too complex. Therefore, we need tools, models, and architectures that make multiprocessors more programmable.

One compelling way to improve programmability is to enable back-and-forth time travel of multiprocessor execution. Such ability simplifies parallel code debugging, and is possible using a technique called Deterministic Replay of Execution. This thesis presents *DeLorean*, a novel hardware substrate for deterministic replay of multiprocessor systems. DeLorean advances the state of the art in that it enables high-speed recording and replay of parallel execution and in that its space requirements are orders of magnitude smaller than those of current schemes.

To be practical, DeLorean and other hardware replay systems need to support an environment with multiple parallel jobs running concurrently — some being recorded, others being replayed and possibly many others running without recording or replay. To address this issue, this thesis presents *Capo*, a novel software-hardware interface for practical high-speed multiprocessor replay. It also introduces the novel abstraction of *Replay Sphere* to separate the responsibilities of the hardware and software components of a replay system. Finally, this thesis describes *CapoOne*, a prototype of a deterministic multiprocessor replay system that implements Capo using Linux running on simulated DeLorean hardware.

*To Shelley, for her love.*

*To my parents, for everything they have done —and still do — for me.*

*To my brother, for being there whenever I need him and for making me laugh so much.*

*To my grandfather, for making me the happiest kid on earth.*

*A Shelley, por su amor.*

*A mis padres, por todo lo que han hecho —y todavía hacen— por mí.*

*A mi hermano, por estar a mi lado cuando le necesito y por hacerme reir tanto.*

*A mi abuelo, por hacerme el niño más feliz de la tierra.*

# ACKNOWLEDGMENTS

It would not have been possible to write this thesis without the help and support of the wonderful people around me, to only some of whom it is possible to give particular mention here.

Fist and foremost, I would like to thank my advisor, Josep Torrellas, for his guidance and support during my time at the University of Illinois. He taught me how to do quality research and persuaded me to stay in the Ph.D. program after I had basically decided to leave. In particular, I am very grateful to him for believing in me from the very beginning, when he called me at my office in Spain to convince me to join his research group.

I would also like to thank my co-advisor, Samuel T. King, for helping me so much with Capo and CapoOne. I cannot even begin describing how much I have learned from him during such a project. I would have never been able to finish on time without his help.

In addition, I would like to show my gratitude to the members of my Ph.D. committee, Prof. Marc Snir, Prof. Wen-mei Hwu, Prof. Yuanyuan Zhou and Prof. Christoph von Praun, for their feedback and their recommendation letters.

As a member of the i-acoma group, I am indebted to many of my colleagues who made our office a great workplace. We worked hard, but we had lots of fun as well. I would like to thank Brian Greskamp for his friendship and for all the discussions we had during the last five years. I would also like to thank Radu Teodorescu and Jun Nakano, with whom I co-invented the office-badminton. I am also grateful to Luis Ceze, with whom I co-authored several papers. He was my mentor and remains a very good friend. I

must also mention Karin Strauss and James Tuck, as I learned a great deal from our office discussions. I had the pleasure of working with Wonsun Ahn, who helped me building the CapoOne infrastructure.

I would not have made it to this point without the support of many friends. I would like to thank my dear friends from Spain Elena Cuevas, Iván García, Octavio Silva and Isabel Villa for their long-distance friendship and moral support. I would also like to thank my friend and fine mathematician Rafael Santamaría for believing in me, even more than I did at times. Also, thanks to my friends Xavier Llorá, Ana Vivancos, Angel Yanguas-Gil and Elsa Alvaro for making Urbana not only bearable, but in fact a great place to be.

I owe my deepest gratitude to my parents, Juan José and Mercedes, and my brother, Nacho, for their support throughout my life. I have no words that can describe their unconditional love for me. This dissertation is simply unimaginable without them. They supported me every step of the way, even though that meant that I would move thousands of miles away from our home in León, Spain.

Finally, I would like to thank my wife Shelley because she is the reason I am still a sane person (for the most part, at least). She gave up doing fun stuff for countless weeknights and weekends so that I could work on my thesis. She stood by me when research did not go as planned and cheered me up every single day. She is the love of my life and my best friend. To her I dedicate this thesis.

# TABLE OF CONTENTS

# LIST OF FIGURES

# CHAPTER 1

# Introduction and Motivation

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 logging 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 [2, 6, 8, 15, 23, 39, 43, 57, 67] — including non-deterministic ones [17, 32, 53],— because 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 [30] or to inspect the actions of an attacker [31]. Third, system designers can use replay as an efficient mechanism for recreating the state of a system after a crash [9, 20, 62].

Current software-only deterministic replay systems [9, 20–22, 39, 53, 57] are flexible and integrate well with the rest of the software stack. However, they perform slowly on (or do not work with) multiprocessors. Thus, hardware-based schemes have been proposed [5, 28, 41, 64, 65]. Even though they are much more efficient, they generate large logs and their performance is limited because they constrain the reordering of memory operations. Furthermore, they impose restrictions on how the users can record and replay applications, rendering them largely impractical.

The work presented in this thesis makes several contributions to deterministic replay

of execution. The first one, presented in Chapter 4, explores a novel hardware approach to higher-performance multiprocessor replay that provides substantial reductions in log size and a much improved recording and replaying performance. This scheme is called *DeLorean*. It uses a new substrate for deterministic replay where processors execute *chunks* of consecutive dynamic instructions atomically and in isolation, as in transactional memory or speculative execution. In this environment, recording an execution no longer involves logging the dependences between individual memory accesses. Instead, in its simplest form, the hardware just records the total order in which processors commit chunks. Because these chunks can be thousands of instructions long, the resulting log size is very small. Moreover, because each chunk is automatically executed in isolation, memory instructions can be reordered freely inside a chunk, which enables high performance execution and replay. Chapter 4 also proposes more advanced execution modes for DeLorean that allow users to trade performance for log size. One of them predefines the processor commit order: it requires almost *no log* and it is about as fast as previous hardware-based replay schemes.

The second contribution of this thesis is a novel software-hardware interface for practical hardware-assisted deterministic replay. A limiting issue of hardware-based schemes is that they are largely *impractical* for use in real machines. The main reason is that they are *hardware centric* designs: they do not distinguish between software that it is being recorded or replayed and the rest. As a result, users cannot record or replay on an per-application basis, nor they can record or replay different applications concurrently. Moreover, they must replay under the control of a complex virtual machine monitor or inside a full-system simulator. To address these issues, Chapter 5 presents Capo, a software-hardware interface that enables *practical and efficient* multiprocessor replay. Capo's key abstraction is the *Replay Sphere*: a set of threads that are replayed or recorded as a unit, together with their address space. Replay spheres isolate processes that are being recorded or replayed from the others, and provide a clean separation between the responsibilities

of the software and the hardware components. In Capo, the replay hardware captures the memory access interleaving of a sphere's threads in a per-sphere log. The software, in turn, records any input to the replay sphere (i.e. input data from the network) in a different per-sphere log. During the replay phase, the hardware uses its log to enforce the same memory access interleaving while the software injects the logged inputs back to the replay sphere.

The final contribution of this thesis is the development and evaluation of *CapoOne*, an implementation of Capo based on Ubuntu Linux [10] running on top of a simulated x86 implementation of DeLorean. CapoOne can record and/or replay two or more unmodified multithreaded applications, both independently and simultaneously. Moreover, our evaluation of 4-processor executions running parallel applications shows that CapoOne largely records with the efficiency of hardware-based techniques and the flexibility of software-based schemes. As a result, a user could record an application while replaying another one without impacting the performance of the rest of the executing applications.

This thesis is organized as follows. Chapters 2 and 3 present related work and give some necessary background. Chapters 4 and 5 introduce DeLorean and Capo, respectively. Chapter 6 describes CapoOne and discusses a series of lessons learned during its development. Chapters 7, 8 and 9 describe our simulation environments and evaluate DeLorean and CapoOne. Finally, Chapter 10 discusses some possible avenues for extension to the work presented in this thesis.

# CHAPTER 2

# Background: Deterministic Replay of Execution

## 2.1 Introduction

Deterministic replay is a well-known technique that enables time travel of computer systems. It consists of two phases. During the *Recording* phase —also known as the *Initial Execution* phase— the software runs while the deterministic replay mechanism records into a log certain non-deterministic events. Different replay schemes log different events. For example, a compiler-based deterministic replay system might need to record the values returned by all load instructions; an OS-based scheme might just require logging the values returned by system calls and the OS scheduler decisions.

During the second phase, known as the *Replay* phase, the replay system first restores the software to a previous checkpoint. From there, the software restarts execution, but this time the replay system uses the log generated during the initial execution phase to make sure that the software follows the same execution path.

Researchers have devised many different deterministic replay schemes, and all of them guarantee that the outcome of the execution is the same in both recording and replay under certain assumptions. Each of them, however, has a different understanding of what replayed execution means. For example, consider a processor spinning on a lock. Some schemes make sure that the processor spins the same number of times on the lock during initial execution and during replay. Others, on the other hand, do not care whether the processor spins a different number of times as long as the execution outcome is not affected. The reader should notice that no deterministic replay scheme guarantees that the

timing of the replay is the same as the timing of the initial execution.

In this thesis we classify the deterministic replay schemes into software-based and hardware-based. We say that a scheme is hardware-based if it requires hardware structures or devices that have been specifically designed for replay. Otherwise, we classify it as software-only.

## 2.2 Six Desirable Traits of Deterministic Replay Schemes

In this thesis, we argue that schemes for deterministic replay have six desirable traits. First, to be able to be used in production-run systems, they should record with little or no overhead. Second, to support long recording periods, their logging requirements should be minimal. Third, to maximize potential uses, their replay speed should be similar to the initial execution speed. Fourth, they should require modest or none hardware support. Fifth, they should operate on unmodified software and binaries. Finally, given the popularity of multicore processors, they should efficiently record and replay multithreaded software running on multiprocessor systems.

Sections 2.3 and 2.4 give an overview of both software-based and hardware-based schemes, and describe how different schemes measure along our six desirable traits.

## 2.3 Software-based Deterministic Replay Schemes

A number of software-based deterministic replay schemes have been proposed. In *Recap* [45], Pan and Linton propose using the compiler to record how user-level threads access shared-memory. In their scheme, the compiler inserts at least two additional instructions for every read that may access a shared location: one of them saves the loaded value into a log and the other increments a log pointer. In their paper, Pan and Linton do not provide any experimental data. However, because every value loaded from a poten-

tially shared location must be recorded, Recap generates a large log and thus it can only replay the last few seconds of execution.

*Instant Replay* [36] uses a programming model where threads can *only* communicate via controlled accesses to common data structures that live in shared memory. Instant Replay uses a concurrent-read-exclusive-write (CREW) protocol [18] to ensure valid serialization of accesses to shared data, and can be used in both message-passing and shared-memory machines. In such environment, before a thread can write to a shared object *S*, it executes a library procedure that i) waits until no other thread is reading *S*, ii) locks *S*, iii) logs *S*'s version number and the total number of readers for this version and, iv) updates *S*'s version number. Similarly, before a thread can read from *S*, it must execute a procedure that i) waits until no other thread is modifying it, ii) locks *S*, iii) logs *S*'s version, and iv) increases the concurrent and total readers counters. After a thread finishes writing or reading the object version, it unlocks the object and performs some other housekeeping operations. This approach works well when programs access shared memory in a coarse grain fashion. However, the overhead is significant if the programs do fine-grain accesses. Furthermore, code must be reshaped to use Instant Replay's shared-object access procedures. A similar scheme, *Agora* [24], also uses version numbers to implement record and replay, except that it does not employ the CREW protocol to manage version numbers.

At the virtual machine level, Choi and Srninivasan [16] propose *DejaVu*. In their paper, the authors focus on recording and replaying Java multithreaded applications running on uniprocessors, although their approach could also be used —under certain assumptions and with a large overhead— in multiprocessor systems. DejaVu records the schedule of the threads running on top of a Java VM. Instead of modifying the underlying thread scheduler, DejaVu captures the thread scheduling by looking at two types of events, namely synchronization operations and shared-variable accesses. Suppose that thread $T1$ accesses shared memory location $X$ and later $T2$ acquires lock $L$. With this information, DejaVu can infer that $T1$ was scheduled before $T2$. Because each of these

events require acquiring a global lock and incrementing a shared variable, DejaVu can incur in considerable overhead even in uniprocessor systems.

Researchers have also proposed modifying the operating system to record and replay certain applications running on it. For example, the *Flashback* [57] system records and replays processes by modifying the implementation of an 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. The Flashback paper focuses on single-threaded applications, and the authors propose adopting DejaVu's approach to deterministically replay multithreaded applications in uniprocessors. For multiprocessor systems, they acknowledge that some architectural support would be needed to replay shared-memory accesses. Russinovich and Cogswell [53] propose *Repeatable Scheduling* to record and replay multithreaded applications running on a uniprocessor system. Like DejaVu, their approach consists in recording the thread scheduling. However, it does so by modifying the Mach [50] operating system's scheduler so that it informs the replay system of each thread switch. Unfortunately, this technique would never work on a multiprocessor system because it cannot record accesses to shared memory locations. Moreover, their paper only focuses on replaying the scheduling decisions and it does not study how to replay applications that read inputs from the network or from files that change from execution to execution.

Other researchers propose using virtual machine monitors (VMMs) to replay entire virtual machines. Bressoud and Schneider [9] modify a hypervisor to replay virtual machines on Alpha-based computer systems, and the *ReVirt* [21] and *ReTrace* [66] projects modify the VMM to replay virtual machines on modern x86-based computer systems, provided that the VMs run on single virtual processors. These projects record external in-

put from the network and other I/O by logging the calls that read these devices. To replay interrupts, these schemes log the dynamic instructions at which interrupts are delivered and ensure that they are redelivered at precisely the same point during replay. The authors use performance counters in the processor to uniquely identify such points .

In a recent paper, Dunlap *et al.* present *SMP-ReVirt* [22], an extension of ReVirt that works with multiprocessor virtual machines. SMP-ReVirt guarantees that reads and writes to shared memory are ordered with respect to each other by using the CREW [18] protocol at page granularity. Thus, pages can be in either *concurrent read* or in *exclusive-write* states. For example, if a virtual processor attempts to write into a page marked as concurrent-read, then the CREW system sends messages to the other processors to decrease their permissions before it may increase the requester's. This can be done rather straightforwardly using the VMM's shadow page tables [55]. SMP-ReVirt records into a log the CREW events so that they can be replayed. Unfortunately, it is very common that two or more processors write into the same page but without writing into shared variables, resulting in many unnecessary CREW events, which are not cheap. Consequently, SMP-ReVirt can impose considerable overhead.

*iDNA* [6] uses dynamic binary translation and interpretation to record and replay multithreaded applications, even in multiprocessor systems. iDNA emulates the instruction stream by breaking the guest instructions into sequences of simpler operations that are fed into a simulator. A tracer connected to the simulator records all the memory values read by the executing instructions, together with other relevant info (i.e. the register state). As expected, recording each value read by an application produces a large log. Thus, even though iDNA compresses its logs very efficiently, its space overhead is still large. Moreover, because iDNA runs the applications within a simulator, its runtime overhead is high (6X). Still, iDNA has been successfully used within Microsoft to detect and classify data races in mature, large applications.

Musuvathi *et al.*'s *CHESS* [39] uses a form of deterministic replay to debug applica-

tions. CHESS does not implement a complete log and replay mechanism, being up up to the user to provide most of the deterministic inputs — it does log, however, input values that are not easily controlled by the user, such as accesses to the timer or the processor ID. During the program execution, CHESS redirects calls to concurrency primitives to alternative implementations in a wrapper library, and records the scheduler decisions. To avoid having two threads concurrently accessing shared-memory locations, CHESS enforces single-threaded execution, which of course has an severe effect on performance. During testing, CHESS first replays a sequence of scheduling choices from the log. If the replay phase does not find any bug, CHESS continues exploring other schedules. CHESS is a successful tool and it is now used at Microsoft to test production codes.

Finally, *RecPlay* [52] uses record and replay to detect data races. During recording, RecPlay only records information (i.e. outcomes) about the synchronization events — completely ignoring shared data accesses. As a result, RecPlay records execution very efficiently and requires a small log. Unfortunately, recording such little information also prevents RecPlay from replaying applications with data races.

### 2.3.1 Summary of Software-Based Replay Systems

Table 2.1 summarizes how the software-based deterministic replay schemes described in Section 2.3 measure along the six axes that Section 2.2 argued are key for replay schemes: initial execution speed, log size, replay speed, hardware needed, ability to use unmodified binaries and support for multiprocessor execution.

It can be seen in the table that software-based techniques work well for uniprocessor execution, but that their performance drops significantly when they deal with multiprocessors. This is the case of SMP-ReVirt and iDNA. RecPlay [52],on the other hand, has outstanding multiprocessor performance but only because it does not record shared data accesses and, therefore, cannot replay applications with data races. Moreover, their imple-

| Replay Scheme | Recording Overhead (%) | Estimated Log Size (b/kilo-inst) | Replay Overhead (%) | Hardware Support | Unmodified Binaries | SMP Support |
|---|---|---|---|---|---|---|
| Recap | N/R | N/R | N/R | None | N | Y |
| Instant Replay | $<5$ | 1.1 | $< 5$ | None | N | Msg. Passing |
| Agora | N/R | N/R | N/R | None | N | Msg. Passing |
| DejaVu | 47.2 | 0.01 | 32.6 | None | Y | N |
| Flashback | $<10$ | N/R | N/R | None | Y | N |
| Repeatable Scheduling | 9.75 | 0.09 | 11.91 | None | N | N |
| ReVirt | 35.2 | 0.04 | 27.1 | None | Y | N |
| ReTrace | 5.9 | 4 | N/R | None | Y | N |
| SMP-ReVirt (4 procs) | 425 | 0.62 | N/R | None | Y | Y |
| iDNA | 600 | 300 | 1000 | None | Y | Y |
| CHESS | N/R | N/R | N/R | None | N | Y |
| RecPlay | 2.1 | N/C | 91 | None | Y | Y |

**Table 2.1:** Comparing the main issues in software-based deterministic replay schemes. "N/R" stands for "Not Reported". "N/C" stands for "Could Not Convert", meaning that we did not have enough information to convert their reported log size into bits per kilo-instruction.

mentation assumes that the data returned by system calls is always deterministic; adding support for recording and replaying system calls would add more overhead. Instant replay also has very little overhead, but it requires the application to be modified so that shared data accesses are done using special functions. If the programmer forgets to use these functions, Instant Replay might not be able to deterministically replay the application.

Column 3 of Table 2.1 shows our estimation of the size of the log generated by each scheme in bits per kilo-instruction. Note that each software-based technique uses different units to measure their log size and runs different benchmarks on completely different processors. Thus, we have converted all the results to be in terms of bits per kilo-

instruction, which is a very common metric in the hardware-based replay community. The table shows that most schemes have very small logging requirements, with the exceptions of iDNA. As we discussed earlier, iDNA logs every single value loaded by the application, so its large log is expected.

## 2.4 Hardware-based Deterministic Replay Schemes

The software-based techniques described in Section 2.3 are flexible, integrate well with the rest of the software stack and have been proven effective. Unfortunately, Table 2.1 shows that some of them do not work with multiprocessor execution and, the ones that do, perform very slowly. This is because current software-only techniques for interposing on shared-memory accesses are inefficient.

Fortunately, hardware can record very efficiently how processors access shared memory. As a result, several hardware-based schemes have been proposed. These schemes propose special hardware to detect the interleaving of memory accesses or instructions from different processors during execution, and save the key information into a log, which is later used to enforce the same interleaving. This log is often called the memory interleaving, memory ordering or memory access interleaving log in the literature. In this thesis, we will refer to it as the *Memory Ordering Log*.

Bacon and Goldstein [5] were the first to propose using hardware to record the interleaving of multiprocessor execution. Their key insight was that, in a sequentially consistent multiprocessor, it is possible to capture the dependences between concurrent threads by logging the coherence messages in the bus. They used a board attached to the bus to record all such messages in a log stored in dedicated memory and/or disk. Although their scheme did not degrade performance during normal execution, their log size was substantial because every bus transaction was recorded. During replay, Bacon and Goldstein's approach allows the first processor in the log to run up to the instruction preceding the

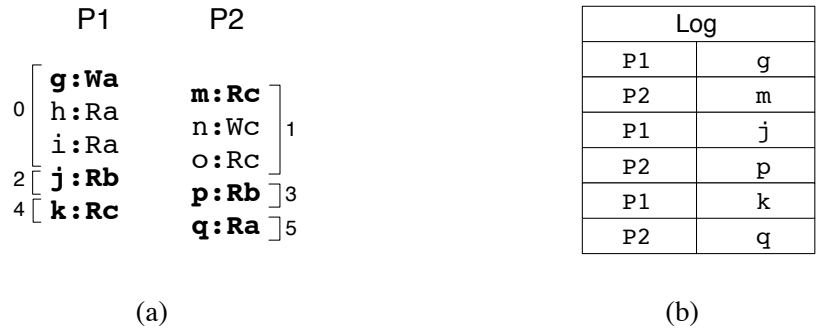



**Figure 2.1:** Recording multiprocessor execution using Bacon and Goldstein's scheme: initial execution (a) and resulting Memory Ordering log (b). Instructions in bold indicate cache coherence events in the bus. The segments labeled *0* to *5* show the replay schedule.

instruction that caused that processor's next bus transaction, effectively serializing the execution.

Figure 2.1 shows Bacon and Goldstein's scheme. At instruction *g*, processor *P1* writes into location *a*. This creates a coherence message in the bus that is recorded into the log. The next two instructions from *P1* read from *a*, and therefore they do not originate coherence messages. At instructions *j* and *p*, processors *P1* and *P2* read from *b*, thus creating two coherence messages that must be logged as well.

Figure 2.1(a) also shows the replay schedule for the example code. During replay, the system executes instructions *g*, *h* and *i* from processor *P1* (segment *0*). Then, it executes *m*, *n* and *o* from *P2*. This process continues until the end of the log (Figure 2.1(b). It can be seen in the figure that Bacon and Goldstein's approach leads to unnecessary serialization, as segments *0* and *1* could have been executed in parallel because they do not access the same locations.

The *Flight Data Recorder* (FDR) [64] records and replays directory-based multiprocessors under SC. Like Bacon and Goldstein's scheme [5], FDR observes coherence messages between processors. However, it only records a subset of them, which are chosen using a hardware version of Netzer's Transitive Reduction (TR) algorithm [44]. Netzer's optimization is based in two observations. First, that the order between instructions that access separate memory locations does not need to be recorded. And second, that depen-

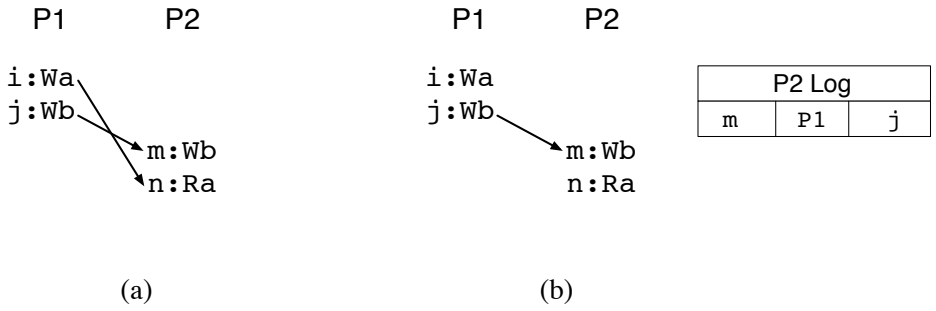



**Figure 2.2:** FDR's transitive reduction optimization: dependences in initial execution (a), recorded dependence (b).

dences that can be transitively implied by others do not need to be recorded either.

Figure 2.2 illustrates FDR's transitive reduction optimization. In Figure 2.2(a), instructions *i* and *j* of processor *P1* write locations *a* and *b*, respectively. Later, *P2*'s instructions *m* and *n* access *b* and *a*, creating the dependences *i:Wa→n:Ra* and *j:Wb→m:Wb*. However, the dependence *i:Wa→n:Ra* does not need to be recorded because it is transitively implied by *i:Wa→j:Wb*, *j:Wb→m:Wb*, and *m:Wb→n:Ra*. Consequently, FDR only records *j:Wb→m:Wb* — Figure 2.2(b). For this, FDR saves the processor ID and instruction count of the two instructions in *P2's* log. The sum of all per-processor logs constitutes FDR's Memory Races Log. During replay, the system makes sure that *P1* has executed *j:Wb* before *P2* can execute instruction *m:Wb*.

FDR augments each cache block with the count of the last instruction that accessed it. FDR increases the area of the caches by 6.25% and generates a compressed log size of 2MB per 1GHz processor per second [64].

BugNet [42] reuses FDR's hardware to replay user code and shared libraries. Its key insight is that it is possible to replay a parallel application by recording the register file contents at any point in time and then record the values returned by all the load instructions executed afterwards. BugNet efficiently records the output of all load instructions by compressing them with a hardware-based dictionary scheme.

Xu *et al.* [65] extend FDR in several ways, although this section focuses on the two main extensions. The first one is the *Regulated Transitive Reduction* (RTR). The idea is

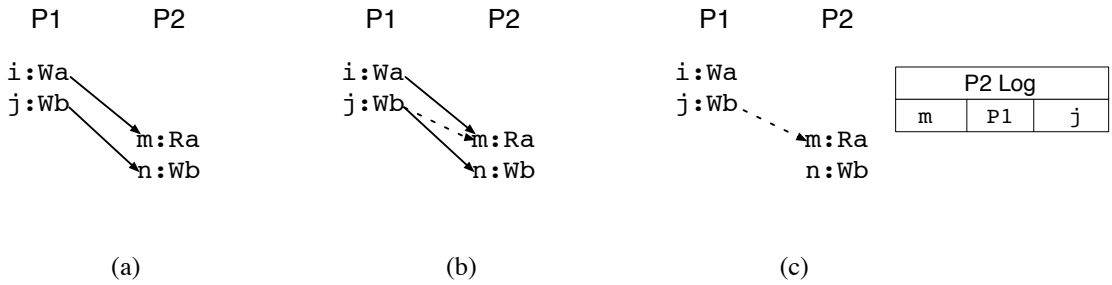


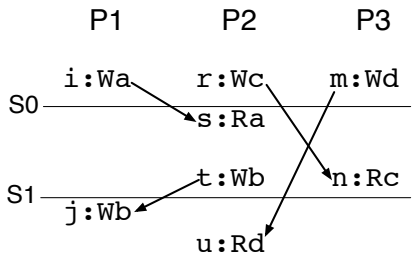

**Figure 2.3:** RTR optimization: dependences in initial execution (a), dependences in initial execution plus artificial dependence (b), final recorded dependence (c).

to judiciously introduce artificial dependences so that Netzer's TR can eliminate other dependences. Figure 2.3 illustrates this process. The original execution — Figure 2.3(a) — has dependences *i:Wa→m:Ra* and *j:Wb→n:Wb*. FDR would record both because none of them could be transitively implied by the other. In Figure 2.3(b), RTR introduces a new, stricter artificial dependence: *j:Wb→m:Ra*. Applying RT to this set of dependences eliminates the need to record the two original dependences — Figure 2.3(c). Although not shown in the figure, RTR also saves space by representing recurring dependences with a vector notation.

The second contribution of Xu *et al.*'s work is a recording algorithm for a TSO machine [65]. It extends FDR's algorithm as follows. There is hardware in the processor that detects when a load has violated SC. In this case, the dependence that FDR would log (assuming SC) is not logged. Instead, the hardware logs the value read by the load, which is later fed to the replayer. Supporting TSO is significant because TSO is used in real machines. However, the authors do not evaluate the impact of the new algorithm on execution speed or on log size [65].

This thesis refers to Xu *et al.*'s work [65] as the RTR system, and distinguish between the Base (no TSO) and Advanced (TSO) support. The Base RTR support logs about 1B per processor per kilo-instruction (compressed).

The *Strata* [41] replay scheme records dependences differently than FDR/RTR. Rather than logging individual dependences with a pair of instruction counts, each entry




(a)

| | Log | | |
|---|---|---|---|
| S0 | i | r | m |
| S1 | 0 | t-r | n-m |

(b)

**Figure 2.4:** Recording multiprocessor execution using Strata: dependences in initial execution (a), Strata's Memory Ordering Log(b).

(a "Stratum") in Strata's Memory Races Log is a vector of as many counters as processors. Each counter is the number of memory operations issued by the corresponding processor since the last stratum was logged. A stratum is logged before a processor issues the second access of an inter-processor dependence. Therefore, a stratum separates at least two dependent memory operations executed in different processors.

Figure 2.4(a) shows a reference trace with the points (*S0* and *S1*) where strata are logged. Right before the second reference of the dependence *i:Wa→s:Ra* is issued, Strata logs the memory reference counts of all 3 processors. The same process is repeated right before the second reference of the dependence *t:Wb→j:Wb*. The other two dependences in the figure do not require the creation of a new stratum: each of them already has its two references in different strata regions. Figure 2.4(b) shows the Strata's Memory Ordering Log corresponding to the memory interleaving found in Figure 2.4(a).

Strata works with directory- and snoop-based systems — both under SC. Strata can choose to ignore WAR dependences when building the log. In this case, WAR dependences are uncovered at replay at the cost of slowing down the replay with multiple re-executions [41]. The compressed log for 4 processors is 2.2KB per 1M memory references. Although Strata's log size for 4 processors is comparable to RTR's, it is proportional to the number of processors in the system. As a result, Strata's log is significantly larger than RTR's on systems with 8 or 16 processors [28].

*Rerun* [28] takes yet another approach to generating the Memory Ordering Log. In-

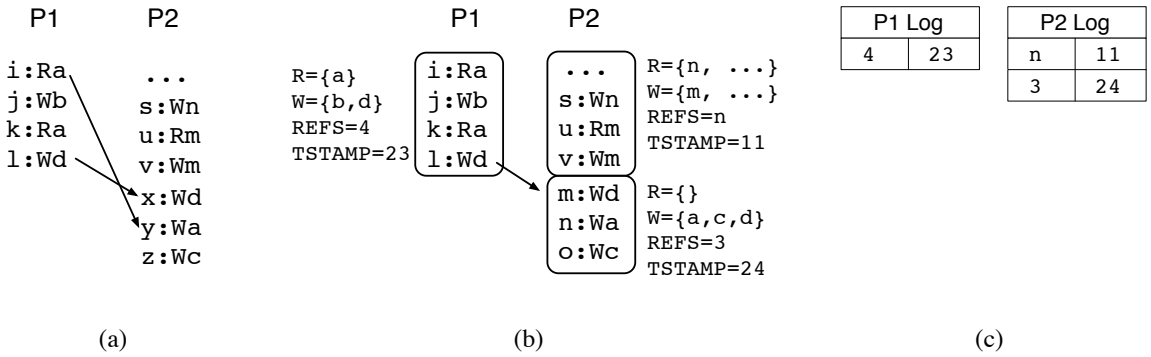



**Figure 2.5:** Recording multiprocessor execution using Rerun: dependences in initial execution (a), Rerun's Episodes (b) and Rerun's Memory Ordering Log (c).

stead of focusing on individual dependences between instructions of different threads, it focuses on regions of *independence*. Rerun breaks the dynamic instruction stream into consecutive series of independent execution regions, called *Episodes*. Thus, each episode is a set of consecutive dynamic instructions from the same thread that execute without conflicting with any other thread of the system. For each processor, Rerun uses two bloom filters [7] to track each episode's read and write sets. In addition, Rerun uses a per-processor counter to record the number of memory references per episode and a per-processor register to store the current episode's Lamport clock.

Figure 2.5 shows Rerun in action. In the original execution —Figure 2.5(a)— there are two dependences: *i:Ra→y:Wa* and *l:Wd→x:Wd*. The latter causes P2 to complete the episode with Timestamp 11 and start a new one with Timestamp 24 —Figure 2.5(b). The counters $REFS$ in Figure 2.5(b) store the number of memory accesses in each episode. Figure 2.5(c) shows the final recorded Memory Ordering Log. Notice that the first dependence, *i:Ra→y:Wa*, does not create a new episode because instruction *i:Ra* belongs to an episode that has already finished and has a timestamp that is smaller than the current episode's.

During replay, episodes are ordered and replayed using the recorded timestamps. Unfortunately, this makes Rerun replay mostly sequential. On the other hand, Rerun has very modest hardware requirements and its final Memory Ordering Log size is similar to

RTR's.

ReEnact [47] leverages thread-level speculation (TLS) mechanisms to debug data races in multithreaded programs. As soon as ReEnact detects a data race, it rolls back execution and enters a data race characterization phase. During this phase, ReEnact deterministically replays the instructions leading to the data race and collects information such as the instructions and memory locations involved in it. CORD [46] is a data race detector and replayer that is both simpler and faster than ReEnact. Its main idea is to record RAW dependences in a bus-based system.

### 2.4.1 Hardware-Based Full-System Replay

FDR, RTR, Strata and Rerun are *full-system* hardware-based replay schemes. This means that they record and replay both the operating system and applications running on them. Because of this, recording the memory interleaving is not enough to guarantee a correct replay execution: all of them must also record other sources of non-determinism, such as interrupts, DMA operations and inputs to the system — i.e. incoming network packets.

Therefore, in addition to the Memory Ordering Log, all these schemes require an *Interrupt Log*, a *DMA Log* and an *Input Log*. In this thesis, when we refer to the log size of full-system replay schemes, we only consider the Memory Ordering Log. This is because the other logs, such as the Input and DMA logs, are less critical [65] and are handled similarly by all hardware-based full-system schemes.

### 2.4.2 Summary of Hardware-Based Replay Systems

Columns 2-5 of Table 2.2 show our estimation of how Bacon and Goldstein, FDR, RTR, Strata and Rerun measure along the six traits described in Section 2.2. Bacon and Goldstein, FDR, Strata, Base RTR and Rerun have been shown to affect execution speed negligibly. Consequently, we list as their execution speed that of the memory consistency

| Replay Scheme | Initial Execution Speed | Estimated Memory Ordering Log Size (b/kilo-inst) | Replay Speed | Hardware Needed | Unmodified Binaries | SMP Support |
|---|---|---|---|---|---|---|
| Bacon and Goldstein | SC | N/C | N/R | Board attached to system bus | Y | Y |
| FDR | SC | 200 | N/R | Cache hierarchy | Y | Y |
| RTR BASE | SC | 8 | N/R | Cache hierarchy | Y | Y |
| Advanced RTR | TSO? | N/R | N/R | Cache hierarchy + Processor | Y | Y |
| Strata | SC | N/C ($\approx$ RTR) | N/R | Cache hierarchy | Y | Y |
| Rerun | SC | 8 | N/R | Cache hierarchy | Y | Y |

**Table 2.2:** Comparing the main issues in hardware-based, full-system replay schemes. "N/R" stands for "Not Reported". "N/C" stands for "Could Not Convert", meaning that we did not have enough information to convert their reported log size into bits per kilo-instruction. All schemes use a 4-processor configuration.

model supported, namely SC. Advanced RTR supports TSO but its execution speed has not been measured.

It can be seen in the table that, in general, hardware-based systems require larger logs than software-based schemes. Because it logs almost all coherence messages, Bacon and Goldstein requires the largest log. In their paper, Narayanasami *et al* estimate that Strata's log size is similar to RTR's. However, Strata scales very poorly with the number of processors. In fact, when using Strata on an 8 processor configuration, its log size more than doubles. The log size for Rerun is similar to the one for Base RTR, and there is no information for Advanced RTR. There is also no information on the replay speed of these schemes, but we estimate that, in their current shape, replay is significantly slower than the initial execution. In the case of Strata, there are three reasons: (i) the log strata likely act as synchronization barriers for replaying processors, (ii) the presence of WARs (if not recorded) requires multiple replays of the same stratum region, and (iii) the replay

under directory schemes needs a prepass to combine the multiple logs. In the case of FDR and RTR, every dependence requires a communication between two replaying processors. Moreover, the conservative dependences introduced by RTR may potentially cause processor stalls. Finally, replay in Bacon and Goldstein and Rerun is significantly slower because both schemes serialize re-execution.

Most schemes require changes in the cache hierarchy, with Advanced RTR requiring changes in the processor as well. Strata and Rerun have very few hardware requirements. Finally, all of them can operate on unmodified binaries and support SMP systems.

To conclude, Figure 2.6 shows a taxonomy of the replay schemes that have been discussed in this chapter.

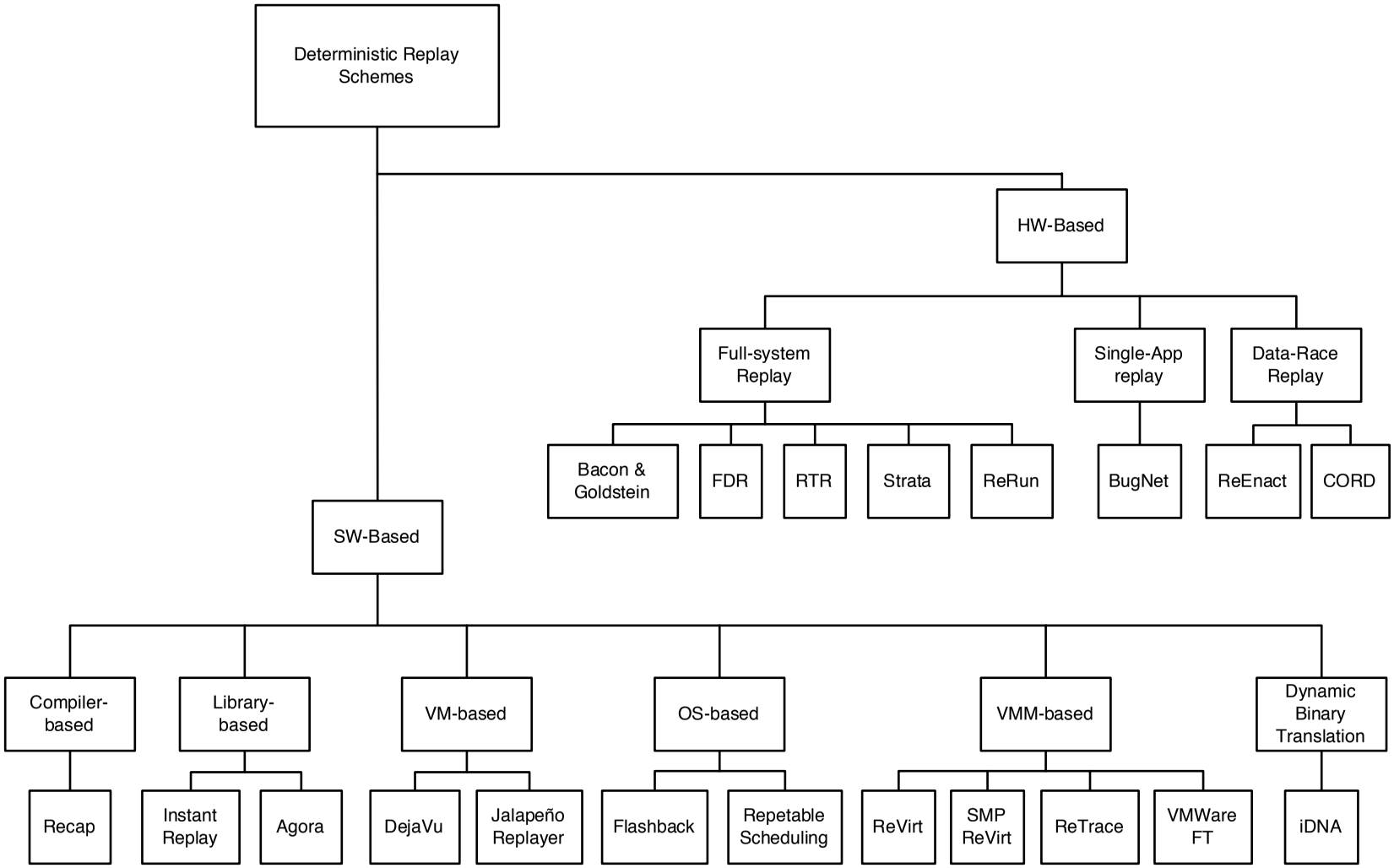



**Figure 2.6:** Deterministic replay schemes classification.

# CHAPTER 3

# Background: BulkSC

It is widely accepted that the most intuitive memory consistency model, and the one that most programmers assume is Sequential Consistency (SC) [35]. Despite this advantage of SC, manufacturers such as Intel, IBM, AMD, Sun and others have chosen to support more relaxed memory consistency models [1]. Such models have been largely defined and used to facilitate implementation optimizations that enable memory access buffering, overlapping, and reordering. A straightforward implementation of the requirements imposed by SC on the outstanding accesses of a processor impairs performance too much. Moreover, the hardware extensions that are required for a processor to provide the illusion of SC at a performance level that is competitive with relaxed models are too expensive.

*Bulk Enforcement of SC* or *BulkSC* [13] proposes a new implementation of SC that delivers performance comparable to relaxed models while being simple to realize. The key idea is to dynamically group sets of consecutive instructions into chunks that appear to execute atomically and in isolation. Then, the hardware enforces SC at the coarse grain of chunks rather than at the conventional, fine grain of individual memory accesses.

In BulkSC, processors execute sets of consecutive dynamic instructions (we will call them *Chunks* in this thesis) as a unit, in a way that each chunk appears to execute atomically and in isolation. To ensure this perfect encapsulation, BulkSC enforces two rules. First, updates from a chunk are not visible to other chunks until the chunk completes and commits. And second, loads from a chunk have to return the same value as if the chunk was executed at its commit point —otherwise, the chunk would have "observed" a changing global memory state while executing. BulkSC supports SC because chunks from in-

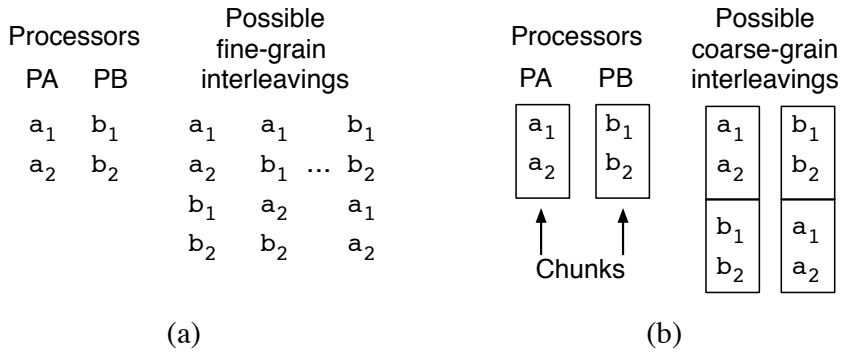



**Figure 3.1:** Fine (a) and coarse-grain (b) access interleaving.

dividual processors maintain program order and chunks from all processors maintain a single sequential order.

As Figure 3.1 shows, some global interleavings of memory accesses from different processors are not possible in BulkSC. However, all the resulting possible executions are sequentially consistent at the individual memory access level. In a sense, BulkSC reduces the number of different possible interleavings in the machine. As we will see in Chapter 4, this is a great asset for deterministic replay.

BulkSC is a key component of the hardware-based deterministic replay system presented in this thesis. Therefore, the rest of this chapter outlines the main components of BulkSC.

## 3.1 Overview of the BulkSC Operation

In BulkSC, a processor takes a checkpoint every *N* committed instructions. The instructions between two checkpoints, or *Chunk*, are executed speculatively until either they get squashed due to a data dependence with another chunk, or they all commit at once, after the chunk has completed. All processors repeatedly (and only) execute chunks.

As a processor executes a chunk speculatively, it buffers the updates in the cache. In addition, the addresses read and written by a chunk are hash-encoded [7] in hardware into

a Read (R) and Write (W) signature. When a processor wants to commit a chunk, it sends its signatures to a simple hardware state machine called the *arbiter*. The arbiter intersects the signatures with those of the chunks that are currently committing. If the intersection is empty, the arbiter keeps the W signature, forwards it to the directory to make the commit visible to all processors, and grants permission to commit to the processor. While a processor is requesting permission to commit a chunk, it continues executing subsequent chunks — each has its own signatures. If the intersection is not empty, the permission-to-commit request from the processor is denied, and the processor will later retry. Since this process is very fast, the arbiter is not a bottleneck in a modest-sized machine.

A processor can have more than one speculative chunk of a thread executing at a time. Memory accesses by a processor are allowed to fully reorder and overlap both within chunks and across chunks. In fact, wihin-chunk execution proceeds with all the memory access reordering and overlapping possible in uniprocessor code.

Chunks from one or multiple processors must appear to commit in a total order. In practice, for high performance, multiple chunks are allowed to commit concurrently as long as the addresses they have accessed do not overlap. Overall, BulkSC execution supports SC, although its performance is practically the same as that of RC execution [13].

## 3.2 Hardware Requirements

BulkSC uses Bulk [12] to perform memory disambiguation. Bulk is a set of hardware mechanisms that simplify the support of common operations in an environment with multiple speculative tasks such as Transactional Memory (TM) and Thread-Level Speculation (TLS). A hardware module called the Bulk Disambiguation Module (BDM) generates the per-chunk signatures described in 3.1. It also includes units that perform signature operations such as intersection, expansion, etc. Signature intersection is used to detect conflicts between chunks. Signature expansion finds the set of lines in a cache that belong to

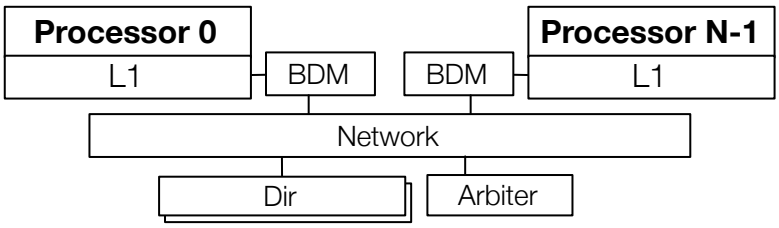



**Figure 3.2:** BulkSC architecture.

a signature withoug traversing the cache, and it is used to perform *bulk invalidation* of the relevant lines from a cache during chunk squashes.

A second key component in the BulkSC architecture is the support for efficient processor checkpointing, which is already feasible today [3, 11, 19, 33, 34, 38, 40, 58]. Chunk squashes leverage the mechanisms of checkpointed processors: a register checkpoint is restored and all the speculative state generated by the chunk is efficiently discarded from the cache.

The L1 cache requires minimum changes because task speculation and address disambiguation are supported by a Bulk Disambiguation Module (BDM) connected to the L1 controller. There is no need to watch for cache displacements to enforce consistency. Clean lines can be displaced from the cache —the R signature records them— and the BDM prevents the displacement of written lines until commit [12]. Thanks to the BDM, the cache tag and data array are unmodified; they do not know if a given line is speculative or what chunk it belongs to.

Figure 3.2 shows an overview of the architecture for a BulKSC system with a distributed directory protocol and a generic network.

# CHAPTER 4

# DeLorean: A New Approach to Hardware-based Deterministic Replay

## 4.1 Introduction

Chapter 2 presented a wide array of deterministic replay schemes. While the software-based ones make major strides toward achieving the six traits described in Section 2.2, they fall short in one or more areas. In general, software-based schemes have poor recording and replaying speeds on multiprocessor systems.

As Table 2.2 shows, the hardware-based schemes also make major strides in these directions. However, they still fall short of our goals in some axes. First, they require Sequential Consistency (SC) — a strict consistency model whose typical implementations have relatively low performance and, therefore, can distort the timing of bugs relative to production-run execution. The exception is RTR [65], which introduces an algorithm to record under Total Store Order (TSO) [59]. However, the impact of this algorithm on execution speed or log size is not evaluated. Secondly, most existing hardware-based schemes capture shared-memory dependences by logging them individually [64] or in groups [41, 65]. For this, they need to log about one byte per kilo-instruction after compression, which limits the duration of the recorded interval. Finally, it is unclear how fast these schemes replay.

In this chapter we present *DeLorean*, a new approach to deterministic replay that provides substantial advances in these six traits. DeLorean uses a new execution substrate: one where processors execute large blocks of instructions atomically, separated by processor checkpoints, like in transactional memory or thread-level speculation. To

capture a multithreaded execution, DeLorean only needs to record the total *order* in which blocks from different processors commit — not individual shared-memory dependences. This results in a substantial reduction in log size compared to previous hardware-based schemes. Moreover, since the memory accesses of a processor can overlap and reorder within and across the same-processor blocks, DeLorean can record execution at the speed of the most aggressive consistency models used today — and replay at a comparable speed. While the hardware used is not standard in today's current systems, the required changes are mostly concentrated in the memory system.

## 4.2 Deterministic Replay of Chunk-Based Systems

Chapter 3 described an environment where processors continuously execute blocks of consecutive dynamic instructions (which in Chapter 3 we called *Chunks*) atomically and in isolation. Other proposals on systems with all-the-time software-annotated transactions such as TCC [26] or checkpointed multiprocessors with all-the-time hardware-based transactions such as Implicit Transactions (IT) [61] have described similar environments. Moreover, such an environment can also be supported in systems with thread-level speculation or with coarse-grain memory ordering support such as ASO [63].

In this environment, the updates made by a chunk only become visible when the chunk commits. When two concurrently-executing chunks conflict — there is a data dependence across the two chunks — one of the chunks is typically squashed and retried. The net effect is that the interleaving between the memory accesses of different processors appears to occur *only* at chunk boundaries. Furthermore, all the memory accesses of a committing chunk appear to occur after those of the chunks committed earlier and before those of the chunks that will commit later. In this thesis, we call this execution environment *chunk-based* execution, and the systems that implement it *chunk-based* systems.

In chunk-based systems, recording the execution for replay involves logging the se-

quence of chunk commits. This provides two fundamental advantages over conventional recorders. The first one is that we can record and replay executions where the memory accesses issued by a processor within a chunk (and in fact across chunks [13]) are fully reordered and overlapped. Recording under such conditions has been a major stumbling block for this area's research, and is recognized by Xu *et al.* [65] as an open problem. The significance of this is that both execution and replay can now proceed at a speed similar to that of a highly-relaxed memory consistency model such as RC. This enables the recording of memory interleavings in *true production runs*. It also enables high-speed deterministic replay. In our view, this has major implications on the applicability of deterministic replay in debugging, fault tolerance and security.

The second fundamental advantage is that the Memory Ordering Log is now very small. Indeed, rather than recording individual dependences, or even groups of them such as in Strata [41] and RTR [65], the Memory Ordering Log in a chunk-based system only needs to record the *total order* in which chunks from different processors commit. This means that each log entry is short (naively, the ID of the committing processor and the chunk size) and the log is updated infrequently (chunks can be thousands of instructions long). Furthermore, in an aggressive design, we can predefine when to finish a chunk and start a new one, and even the chunk commit order. In this case, the log size is similar to that of software-based schemes.

In the rest of this section, we examine the design space of chunk-based deterministic replay systems, present our proposed chunk-based replay architecture called DeLorean, and then put it in the context of conventional replayers.

### 4.2.1 Design Space of Chunk-Based Deterministic Replay Systems

In a chunk-based system, the Memory Ordering Log does not store individual or groups of dependences; it only needs to store the total order of chunk commits. In the simplest

design, each log entry contains the ID of the processor committing the chunk and the chunk size — measured in number of retired instructions.

We can reduce the log size by either reducing the number of entries or reducing the size of each entry. To reduce the number of entries, we can increase the chunk size — i.e., include more instructions in each chunk. However, increasing the chunk size beyond a certain point is counter-productive. First, we may hurt performance because long chunks increase the chances of inter-chunk conflicts and resulting squashes. Second, we may be unable to increase the effective chunk size. Indeed, a long chunk may access more lines mapping to a cache set than ways the cache has — risking the cache overflow of speculatively updated lines. Before this happens, the chunk has to be forcefully finished and committed.

To reduce the size of each log entry, we can omit the chunk size or the ID of the committing processor from the entry. To be able to omit the chunk size, we need to make "chunking" — i.e., the decision of when to finish a chunk — deterministic. We can accomplish this in different ways. One could be to finish chunks at software annotation points — perhaps similar to what is done in transactional memory systems. Another is to finish chunks when a certain number of memory operations or instructions have been committed — like it is done in BulkSC or IT.

In reality, there may be events that truncate a currently-running chunk and force it to commit before it has reached its "expected" size. This is fine as long as the event reappears deterministically in the replay. An example is an uncached load to an I/O port. The chunk is truncated but its log entry does not need to record its actual size because the uncached load will reappear in the replay and truncate the chunk at the same place.

There are, however, a few events that truncate a currently-running chunk and are not deterministic — e.g., cache overflow as discussed above. We will examine these rare events in Section 4.3.2. When one such event occurs, the log is augmented with information on: (i) what chunk gets truncated and (ii) its size. With this information, the ex-

act chunking can be reproduced during replay. This mode of execution makes little sense with checkpointed processors that, for performance reasons, checkpoint (and start a new chunk) at certain non-deterministic events such as cache misses [34] or low-confidence branches [3]. Missing loads and low-confidence branches in the initial execution may become hits and predictable branches in the replay. Consequently, checkpoints and the resulting chunk boundaries may not appear in the same program positions in both executions.

To be able to omit the ID of the committing processor from the log entry, we need to "predefine" the chunk commit interleaving. This can be accomplished by enforcing a given commit policy — e.g., pick processors round-robin, allowing them to commit one chunk at a time. The drawback is that, by delaying the commit of completed chunks until their turn, we may slow down execution and replay.

Based on all these ways to reduce the log size, we have three *Execution Modes* in chunk-based deterministic replay systems (Table 4.1). In the following discussion, we assume that the machine has an *Arbiter* module that observes the order of chunk commits. The arbiter can be associated with the bus controller in a bus-based machine or be an independent module in a machine with a generic network as in BulkSC (Chapter 3) or Scalable TCC [14].

In the *Order&Size* Mode (top left), chunking is not deterministic and the chunk commit interleaving is not predefined. During execution, the arbiter logs the sequence of committing processor IDs in a *Processor Interleaving* (PI) Log. In addition, processors log the size of the chunks they commit in the per-processor *Chunk Size* (CS) Log. During replay, each processor generates chunks that are sized according to its CS Log, and the arbiter enforces the commit order present in the PI Log. The combination of a single PI Log and per-processor CS Logs constitutes the Memory Ordering Log. The table shows the estimated size of the Memory Ordering Log, where *max_chunksize* and *chunksize* are the maximum and average chunk size, respectively.

| | Non-Deterministic Chunking | | Deterministic Chunking | |
|---|---|---|---|---|
| **Non-Predefined Chunk Commit Interleaving** | Name | – *Order&Size* | Name | – *OrderOnly* |
| | Execution | – Arbiter logs committing processors<br>– Processors log chunk sizes | Execution | – Arbiter logs committing processors |
| | Replay | – Arbiter consumes Processor Interleaving Log<br>– Arbiter enforces order in Processor Interleaving Log<br>– Processors consume private Chunk Size Log<br>– Processors chunk according to private Chunk Size Log | Replay | – Arbiter consumes Processor Interleaving Log<br>– Arbiter enforces order in Processor Interleaving Log<br>– Processors execute chunks normally |
| | Log size (bits) | $\approx (\log(\#\text{procs}) + \log(\text{max\_chunksize})) \times \frac{\#\text{ dyn. insts}}{\text{chunksize}}$ | Log size (bits) | $\approx \log(\#\text{ of procs}) \times \frac{\#\text{ dyn. insts}}{\text{chunksize}}$ |
| **Predefined Chunk Commit Interleaving** | — | | Name | – *PicoLog* |
| | | | Execution | – Arbiter enforces predefined commit order |
| | | | Replay | – Arbiter enforces predefined commit order<br>– Processors execute chunks normally |
| | | | Log size (bits) | $\approx 0$ |

**Table 4.1:** Execution modes in chunk-based deterministic replay systems.

In the *OrderOnly* Mode (top right), the commit interleaving is not predefined, but chunking is deterministic. Therefore, there is no need to log the chunk size. During execution, the arbiter logs the committing processor IDs in the PI Log; during replay, it uses the PI Log to enforce the same commit interleaving. The log size is smaller because we only have the PI Log. In reality, each processor also keeps a very small CS Log where, for each of its few chunks that were truncated non- deterministically, it records both the position in the sequence of chunks committed by the processor and the size.

In the *PicoLog* Mode (bottom right), chunking is deterministic and the commit interleaving is predefined. During both execution and replay, the arbiter enforces a given commit order. There is no PI Log. Each processor keeps the very small CS Log discussed for *OrderOnly*. The Memory Ordering Log is largely eliminated.

Note that a mode where the chunking is not deterministic but the chunk commit interleaving is predefined (bottom left) is unattractive. We save log space in the arbiter only to use more in the processors.

### 4.2.2 DeLorean: A Chunk-Based Execution-Replay Architecture

DeLorean is our architecture for chunk-based execution-replay (Figure 4.1). It takes a machine that supports a chunk-based execution environment with a generic network and an arbiter for chunk commit as in BulkSC [13] or Scalable TCC [14], and augments it with the three typical mechanisms for replay: the Memory Ordering Log, the input logs, and system checkpointing.

The Memory Ordering Log consists of the PI and CS Logs. Together, they replace the Memory Races Log Buffer in FDR [64] and RTR [65], and the Strata Log in Strata [41]. They are configured differently depending on which of the three execution modes of Table 4.1 is desired — allowing for different trade-offs between speed and log size. For each execution mode, Table 4.2 lists the log entry formats and the time when the logs are up-

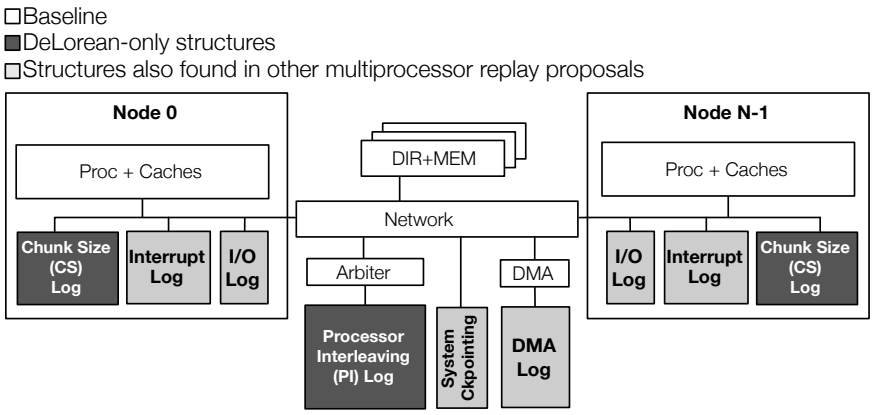



**Figure 4.1:** DeLorean architecture.

| | PI Log | | CS Log | |
|---|---|---|---|---|
| Execution Mode | Log Entry Format | When Updated | Log Entry Format | When Updated |
| *Order&Size* | `procID` | Chunk | `size` | Chunk Commit |
| *OrderOnly* | `procID` | Chunk Commit | `chunkID, size` | Chunk Truncation |
| *PicoLog* | - | - | `chunkID, size` | Chunk Truncation |

**Table 4.2:** PI and CS log in each execution modes.

dated.

In the *Order&Size* and *OrderOnly* modes, when the arbiter gives commit permission to a processor during execution, it also saves the processor's `procID` in the PI Log. During replay, the arbiter uses the sequence of `procID`s in the PI Log to give commit permissions to processors in the correct order.

We can reorganize the PI Log according to the Strata [41] design and reduce its size by half (Section 4.4).

In the *Order&Size* mode, when a processor gets permission to commit a chunk during execution, it records the number of dynamic instructions in the chunk (`size`) in its CS Log. In the *OrderOnly* and *PicoLog* modes, the processor only updates its CS Log when the chunk to be committed has been truncated non-deterministically. In this case, it stores the processor-local sequence order of the chunk (`chunkID`) and its `size`. During replay,

each processor uses its CS Log to determine when it needs to terminate each chunk (in *Order&Size*), or only those that were truncated non-deterministically in the initial execution (in *OrderOnly* and *PicoLog*).

It is possible to combine the PI Log and the per-processor CS Logs into a single, larger PI Log. Such option has two drawbacks. First, during execution, the commit request message between processors and the arbiter will have to include the chunk size and its ID — which consumes extra bandwidth. Second, during replay, the processor will have to wait for the arbiter to supply back this information before it can proceed with the execution of a chunk.

The input logs are similar to those in previous replay schemes. As shown in Figure 4.1, they include one shared log (*DMA* Log) and two per-processor logs (*Interrupt* and *I/O* logs). The DMA Log records the data that the DMA writes to memory. During the initial execution, the DMA acts like another processor in that, before it updates memory, it needs to get commit permission from the arbiter. When the arbiter grants permission, the DMA writes to memory and a copy of the data is saved in the DMA Log. Moreover, the arbiter creates an entry in the PI Log with the DMA's `procID`. Note that, in the *PicoLog* mode, there is no PI Log. In this case, the arbiter records the "commit slot" of the DMA operation, namely the current value of a counter that counts the total number of chunk commits since recording started. Later, during replay, when the arbiter finds the DMA's `procID` in its PI Log — or, in the *PicoLog* mode, when the arbiter's count of chunk commits matches a saved DMA commit slot — the data in the next entry of the DMA Log is consumed.

The per-processor Interrupt Log stores, for each interrupt, the time it is received, its type, and its data. Time is recorded as the processor-local `chunkID` of the chunk that initiates execution of the interrupt handler. The per-processor I/O Log records the values obtained by I/O loads. Section 4.3.2 provides more details.

Like previous replay schemes, DeLorean includes system checkpointing support,

| Replay Scheme | Initial Execution Speed | Estimated Memory Ordering Log Size (b/kilo-inst) | Replay Speed | Hardware Needed | Unmodified Binaries | SMP Support |
|---|---|---|---|---|---|---|
| Bacon and Goldstein | SC | N/C | N/R | Board attached to system bus | Y | Y |
| FDR | SC | 200 | N/R | Cache hierarchy | Y | Y |
| RTR BASE | SC | 8 | N/R | Cache hierarchy | Y | Y |
| Advanced RTR | TSO? | N/R | N/R | Cache hierarchy + Processor | Y | Y |
| Strata | SC | N/C ($\approx$ RTR) | N/R | Cache hierarchy | Y | Y |
| Rerun | SC | 8 | N/R | Cache hierarchy | Y | Y |
| DeLorean's *OrderOnly* | RC | 1.3 | 0.82$\times$RC | BulkSC, IT or TCC | Y | Y |
| DeLorean's *PicoLog* | RC | 0.05 | 0.72$\times$RC | BulkSC, IT or TCC | Y | Y |

**Table 4.3:** Comparing the main issues in hardware-based, full-system replay schemes. "N/R" stands for "Not Reported". "N/C" stands for "Could Not Convert", meaning that we did not have enough information to convert their reported log size into bits per kilo-instruction. All schemes use a 4-processor configuration, except *OrderOnly* and *PicoLog*, which use an 8-processor one.

possibly with schemes such as ReVive [48] or SafetyNet [56]. We do not focus on this issue in this thesis.

As a summary, Table 4.3 augments Table 2.2 with DeLorean in *OrderOnly* and *PicoLog* modes. In Chapter 8, we will see that DeLorean's Memory Ordering Log is orders of magnitude smaller than previous hardware-based proposals. We will also see that DeLorean executes at a speed similar to that of RC execution in *OrderOnly* mode and only modestly slower in *PicoLog* mode. Replay speed will also be shown to be high. The hardware needed is that of a chunk-based system like BulkSC, IT, or TCC, which modifies the memory hierarchy more than the conventional schemes. Processor modifications are largely confined to supporting checkpointing.

### 4.2.3 DeLorean in the Context of Other Replayers

#### 4.2.3.1 Initial Execution

Recall that the advantages of DeLorean in the initial execution are that: (i) it records an environment where memory accesses reorder and overlap substantially, delivering a performance similar to that of a relaxed-consistency machine, and (ii) its log is minuscule. The first advantage comes at the cost of potential squash and re-execution of code sections. In most cases, the squash frequency is very small and the execution time is largely unaffected. In theory, however, squashing can noticeably slow down an application. This issue is not present in conventional replay schemes.

The minuscule log is the combination of two facts: DeLorean needs few log entries and each entry is small. For this discussion, consider the *OrderOnly* mode. DeLorean naturally combines multiple dependences between two processors into a single dependence — something that RTR does at a smaller scale by creating stricter dependences artificially. This is shown in Figure 4.2(a), where *all* the dependences between the instructions in the chunks executed by *P4* and *P5* (shown with arrows in the figure) are combined into a *single* PI Log entry. Moreover, as shown in the figure, such log entry is *simply P4*'s ID.

Similarly, like Strata, DeLorean naturally combines multiple dependences across several processors into a single one. Indeed, as shown in Figure 4.2(b), when a chunk finishes, it is like a *single-processor* stratum: in the figure, the three dependences are summarized into a *single* log entry, which is *simply P5*'s ID. Unlike Strata, though, DeLorean is unable to combine executions from multiple processors into a single stratum. This is shown in Figure 4.2(c), where the chunk-commit log entries for *P4* and *P6* are not combined as they would in Strata. However, while a Strata log entry is very wide — it is a vector of as many reference counters as processors the machine has — a DeLorean log entry is only a processor ID. Interestingly, we can reorganize DeLorean's log according to Strata's design and save space. This is shown in Section 4.4.

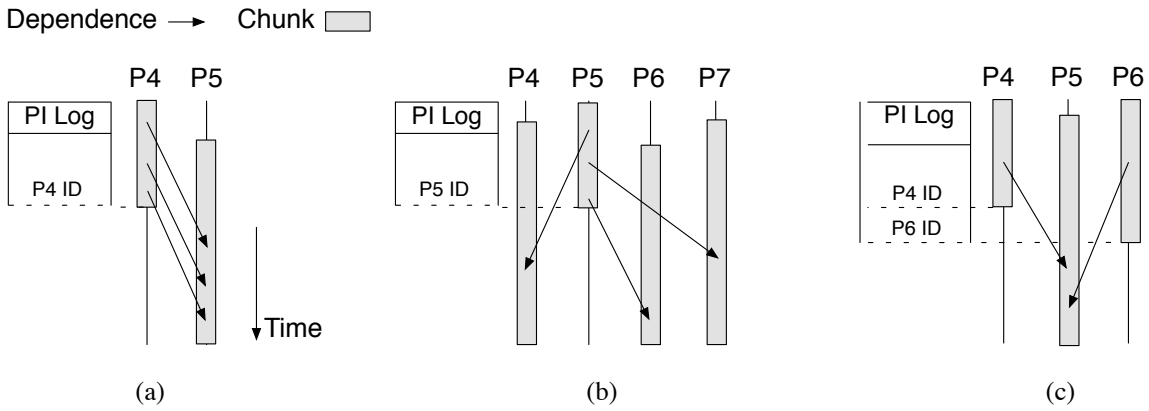



**Figure 4.2:** Comparing DeLorean to RTR and Strata.

Finally, we explained in Section 2.4 how Rerun breaks the dynamic instruction stream into consecutive series of dynamic instructions, called episodes. Rerun's episodes bear some resemblance with DeLorean's chunks. Indeed, DeLorean and Rerun are based in the idea that it is possible to record multiprocessor execution by recording the regions where threads execute without conflicting with any other thread in the system. However, both schemes approach the problem differently: in Rerun, the hardware i) detects when these regions of independence begin and end and ii) logs each region's size and sequence number. DeLorean's hardware, on the other hand, i) manufactures these regions of independence by committing chunks atomically and in isolation and ii) logs their interleaving.

#### 4.2.3.2 Replay

An advantage of DeLorean's replay over previously-proposed schemes is its high speed: all processors execute concurrently, with each processor fully reordering and overlapping its memory accesses. Chunk commit involves a fast check with the arbiter, which is overlapped with the computation of the next chunk, as in BulkSC. Intuitively, therefore, replay speed is likely to be high.

In comparison, the processors in the other schemes replay at most at SC speed (or

TSO in Advanced RTR). They require more communication between the replaying processors: FDR and RTR require a cross-processor communication for each dependence in the log, while Strata requires the replaying processors to synchronize in a barrier at every log stratum. Moreover, As discussed in Section 2.4.2, Strata has other potential sources of replay overhead. Both Bacon and Goldstein and Rerun replay sequentially, making their overhead during replay very significant, especially in systems with many processors.

In practice, recording and replay are likely to proceed on top of a hypervisor layer or maybe inside a simulator. This and other issues will be discussed in Chapter 5.

## 4.3 DeLorean Implementation

We now consider three implementation aspects of DeLorean: implementation choices, exceptional events, and an optimization to further reduce the log size.

### 4.3.1 Implementation Choices and Operation

Fundamentally, a chunk-based execution-replay system needs support for speculative tasking and cross-task address disambiguation — the support needed for transactional memory and thread-level speculation. Such support can be implemented in software, hardware, or in a hybrid way. Moreover, there are multiple degrees of freedom, including whether conflict detection and version management are done lazily or eagerly. In addition, the network can be a bus or generic. If generic, we need an arbiter module — which can be designed in a distributed manner to avoid bottlenecks [13].

DeLorean can be implemented in any of these ways. In this thesis, we choose to implement DeLorean using the signature-based BulkSC [13] architecture that we described in Chapter 3. Our DeLorean system uses a generic network with a directory and a single arbiter module.

We choose BulkSC because its signatures enable fast and efficient memory disambiguation, and an additional log optimization that we will discuss in Section 4.4. Also, BulkSC chunks are automatically created by the hardware, eliminating any need to add software annotations to indicate when the current chunk should finish. Specifically, a chunk in *OrderOnly* and *PicoLog* modes finishes when the processor has committed a certain fixed number of instructions since the chunk started. We call such chunk size the *standard chunk size*.

With this support, Figure 4.3 summarizes DeLorean's operation. During the initial execution in *Order&Size* mode, when a processor such as *P0* or *P1* finishes a chunk, it sends its ID and signature to the arbiter (messages *1* and *2*). Suppose that the arbiter grants permission to *P0* first (message *3*). In this case, the arbiter logs *P0*'s ID (*4*) and propagates the commit operation to the rest of the machine (*5*). While this is in progress, if the arbiter determines that both chunks can commit in parallel, it sends a commit grant to *P1* (*6*), logs *P1*'s ID (*7*), and propagates the commit (*8*). As each processor receives commit permission, it logs the chunk size (*9* and *10*). In *OrderOnly*, steps *9* and *10* are skipped. In *PicoLog*, steps *4*, *7*, *9* and *10* are skipped, and the arbiter grants commit permission to processors according to a predefined order policy, irrespective of the order in which it receives their commit requests. In all cases, a processor does not stall when requesting commit permission; it continues executing its next chunk(s).

During replay, suppose that *P1* finishes its chunk first, and the arbiter receives message *2* before *1*. The arbiter checks its PI Log (or its predefined order policy in *PicoLog*) and does not grant permission to commit to *P1*. Instead, it waits until it receives the request from *P0* (message *1*). At that point, it grants permission to commit to *P0* (*3*) and propagates its commit (*5*). The rest of the operation is as in the initial execution but without logging. In addition, in *Order&Size*, processors use the information in their CS Log to decide when to finish each chunk.

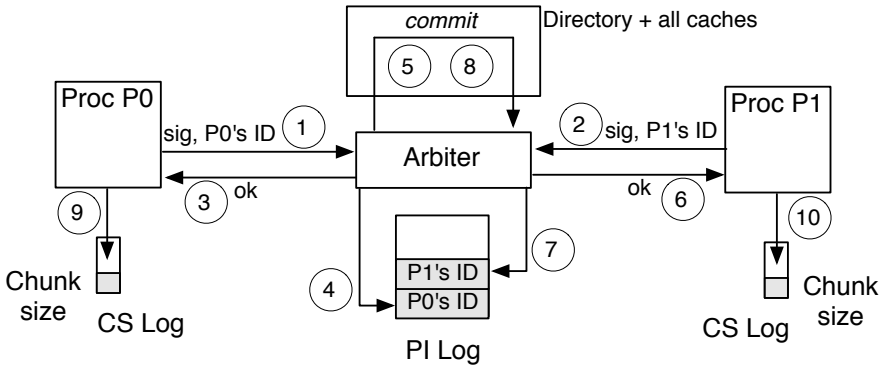



**Figure 4.3:** DeLorean's operation.

### 4.3.2 Exceptional Events

In DeLorean, the same instruction in the initial and the replayed execution must see exactly the same full-system architectural state. Only then can the stream of committed instructions be guaranteed to be the same in both runs. This means, for example, that the two runs perform the same number of spins on a spin-lock, and the same number of system calls and I/O operations.

On the other hand, it is likely that structures that are not visible to the software such as the cache and branch predictor will contain different state in the two runs. This is because, in the two runs, the relative timing of some events may be different, the number of chunk squashes may be different, and structures like the cache and branch predictor may diverge.

Unfortunately, chunk construction is affected by the cache state — through cache overflow that requires finishing the chunk — and by the branch predictor — through wrong-path speculative loads that may cause spurious dependences and induce chunk squashes. Consequently, we need to be careful that chunks are still replayed deterministically.

This section addresses this problem. Table 4.4 lists the exceptional events that might affect chunk construction during the initial execution. In the following, we consider each

| Do not Truncate | Truncate | |
|---|---|---|
| | Deterministically | Non-deterministically |
| 1) Interrupts<br>2) Traps | 1) Reach limit number of instructions<br>2) Uncached accesses (e.g., I/O initiation)<br>3) Special system instructions | 1) Attempt to overflow the cache<br>2) Repeated chunk collision (Not for *PicoLog* and not during replay) |

**Table 4.4:** Exceptional events that may affect chunk construction.

one in turn.

#### 4.3.2.1 Interrupts and Traps

An interrupt during the initial execution does not truncate the current chunk (Table 4.4). If the interrupt has low priority, the processor waits until the current chunk completes; if the interrupt has high priority or the current chunk has recently started, the processor squashes the current chunk. In either case, after this, the processor starts a new chunk while initiating execution of the interrupt handler. Moreover, an entry in the Interrupt Log is created with: (i) the ID of the new chunk — namely, the number of chunks committed by this processor up to now plus one — and (ii) the interrupt's type and data[1].

During replay, interrupts are replayed in the same way in all execution modes. Specifically, each processor keeps a count of the chunks it has committed so far. When such count is one lower than the chunk ID in the next entry of its Interrupt Log, the processor starts a new chunk by consuming the Interrupt Log entry.

A trap does not truncate the current chunk (Table 4.4). Instead, the current chunk simply continues to grow, now executing instructions from the trap handler. In addition,

[1] In *PicoLog* mode, if the interrupt has high priority, the processor can request that the arbiter commit the chunk that handles the interrupt immediately — rather than for the processor to wait until it is its turn to commit. If so, the arbiter records the "commit slot" of the interrupt chunk like it does for DMA requests (Section 4.2.2), to know when to consume it during replay.

the trap is not logged, since it will deterministically reappear during replay. Consider, for example, a page fault trap. The instruction that caused it will cause it again in the replay because the memory state is the same — since wrong-path speculative loads cannot trigger the fault, and squashed chunks cannot modify memory state. The TLB state may be different during replay, but we assume hardware-managed TLBs.

#### 4.3.2.2 Deterministic Truncation of a Chunk

There are certain events that truncate the chunk that is currently-executing in the processor and that will reappear deterministically during replay (Table 4.4). They include the trivial case when the number of instructions committed by the chunk reaches the size limit. More importantly, they include instructions that are hard to undo in a speculative environment, like uncached accesses (such as those that initiate I/O operations) and special system instructions (such as those that change the processor frequency or mask/unmask interrupts).

Following BulkSC [13], when one these hard-to-undo instructions is encountered, the currently-running chunk is truncated, the instruction is executed, and a new chunk starts. Typically, the execution of the instruction is not initiated until the previous chunk commits, and the subsequent chunk does not start until the instruction commits. There is no need to log the size of the truncated chunk in the *OrderOnly* and *PicoLog* modes because the event will recur in the replay and truncate the chunk at exactly the same instruction. The event itself is not logged either. The only exception is that we must log in the I/O Log the value loaded by I/O loads. Such values will be provided to the I/O loads when they are encountered again in the replay.

#### 4.3.2.3 Non-Deterministic Truncation of a Chunk

Finally, there are two events that truncate the currently-executing chunk and are not deterministic (Table 4.4). They are the attempt to overflow the cache and repeated chunk collision.

When a chunk accesses more memory lines mapping to a cache set than ways the cache has, there is the danger that speculatively written data may overflow. Before this happens, execution has to stop. Squashing the chunk and re-executing it does not help because the problem will typically recur. Instead, we need to truncate the chunk, initiate its commit process, and start a new chunk. Note that the actual point in the chunk where overflow is detected is not deterministic. It depends on the actual reordering of loads and stores; e.g., a wrong-path speculative load may trigger the attempt to displace dirty speculative data. Moreover, multiple speculative chunks of a thread concurrently running may interfere and cause the overflow. Consequently, when a chunk is truncated due to attempted overflow during initial execution, the processor records in its CS Log the truncated size of the chunk and, in *OrderOnly* and *PicoLog*, the `chunkID` as well.

During replay, processors use their CS Log to identify which chunks need to be truncated and at which instruction. It is possible that, due to timing differences between initial execution and replay, one such chunk would not have caused overflow during replay — still, it will be truncated to preserve determinism. It is also possible that, during replay, a chunk unexpectedly attempts to cause overflow and has to be committed sooner than in the initial execution. In this case, the processor, as it commits the shorter chunk, requests the arbiter to let it commit a second chunk immediately after. This second chunk contains the rest of the instructions in the original chunk and no other processor can commit until it commits.

The second non-deterministic event, repeated chunk collision, occurs when, during the initial execution, a chunk is repeatedly squashed by other chunks. The simplest solu-

tion proposed in [13] is to progressively reduce the size of the chunk until it can commit. This final size is not deterministic. Consequently, the processor records in its CS Log the truncated size of the chunk and, in *OrderOnly*, the `chunkID` as well. Note that repeated chunk collision cannot occur in *PicoLog*. This is because a chunk can only be squashed by a committing chunk and, in *PicoLog*, there is a predefined chunk commit order.

During replay, processors use their CS Log to truncate chunks that were truncated due to collisions in the initial execution. Note that, during replay, chunks may suffer a different set of collisions. However, the problem of repeated chunk collisions cannot occur because chunks have now a predefined commit order.

Overall, even in the presence of all these types of exceptional events, DeLorean's replay is deterministic. Section 4.5 outlines a proof for why DeLorean's replay is deterministic.

## 4.4 Optimization: Reducing the PI Log Size by Stratifying It

We can reduce the size of the PI Log in *Order&Size* and *OrderOnly* by applying Strata's [41] approach to log construction. Specifically, we design the PI Log to record *chunk strata*. Thus, an entry in this newly designed PI Log is a stratum. Each stratum is a vector of counters that tell the number of chunks committed per processor since the previous stratum. These chunks have no cross-processor conflicts — although they may have within-processor cross-chunk conflicts. Consequently, we need not record their exact commit sequence because, during replay, those chunks among them that belong to different processors can be executed and committed in any order; those chunks that belong to the same processor will serialize their commit by construction.

DeLorean creates a new stratum when the chunk to log next has these properties: (i) it conflicts with chunks committed by other processors since the last stratum or (ii) it would overflow the counter of chunks committed by this processor since the last stratum.

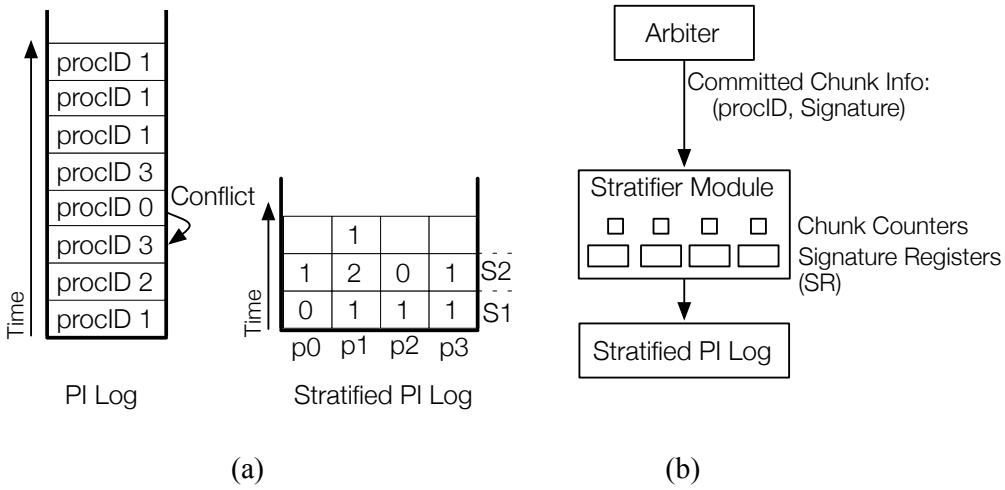



**Figure 4.4:** PI Log stratification: example (a) and design (b).

We call this log optimization *stratifying* the PI Log.

Figure 4.4(a) shows an example of this technique. Assume that there is a conflict between the chunks from processors *3* and *0* whose PI Log entries are connected with an arrow. The other chunks do not have cross-processor conflicts. Also, assume that each counter in the vector can at most reach *2*. The figure shows that stratum *S1* is created when the chunk from processor *0* is next to be logged, while *S2* is created when the last chunk from processor *1* is next to be logged.

We implement this optimization without affecting DeLorean's recording speed as follows. Chunks commit as usual. However, after a chunk commits, rather than dumping its `procID` into the PI Log, we pass its signature *S* to a *Stratifier Module* (Figure 4.4(b)). The Stratifier contains: (i) the vector of chunk counters and (ii) one Signature Register (SR) per processor. The latter contain the logical-OR of the signatures of all the chunks from a given processor committed since the last stratum. When a new chunk arrives at the Stratifier, if the corresponding counter is at its maximum value, the system creates a new stratum by dumping the counters into the PI Log. Then, it sets the corresponding SR and counter to *S* and 1, respectively, and clears the other SRs and counters. Otherwise, *S* is

logically ANDed with the other processors' SRs — without updating the latter. If there is a conflict, the system creates a new stratum as above. Otherwise, *S* is logically ORed into the corresponding SR and the corresponding counter is incremented.

## 4.5 Why DeLorean's Replay is Deterministic

We outline a proof for why DeLorean's replay is deterministic assuming that we use a BulkSC implementation. For brevity, we focus only on the *OrderOnly* mode, but a similar reasoning can be followed for the other two.

In DeLorean, a chunk executes atomically and in isolation. It cannot see any state change while it executes — otherwise it is squashed. The only state it observes is the state of the system when it is about to commit. Thus, we make the following observations.

**Observation 1** *The execution path taken inside a chunk only depends on the state of the system when the chunk is about to commit.*

**Observation 2** *Non-deterministic events that can modify a processor's instruction stream happen at chunk boundaries, and they are logged. They include both external events (e.g., I/O and interrupts) and internal ones (stores executed by other processors' chunks, which are made visible when those chunks commit).*

**Observation 3** *Chunk sizes in the initial execution and in the replay are the same because the decision of when to truncate a chunk is either deterministic (it depends on the instruction stream itself) or reproducible (it is based on information found in the CS Log).*

We now define deterministic replay and show that DeLorean's replay is deterministic.

**Definition** The *Global Commit Count* (GCC) is the number of chunks committed by all processors since execution began.

**Definition** An *Interval I(n,m)* of execution is the period between GCC=n and GCC=n+m,

where m$\geq$1.

**Definition** The *deterministic replay* of interval *I(n,m)* is a new interval *I'(n,m)* such that the initial and final system states, the number of chunks executed by each processor, the instructions in each chunk and the chunk interleaving are the same in *I* and *I'*.

**Theorem** *Assuming that a system checkpoint was taken at GCC=n, DeLorean can deterministically replay an execution for the interval I(n,m).*

**Proof** We use induction on *m*. Start with *m=1*. The PI Log has a single entry (say, processor $P_i$) and the system state has been restored to that at GCC=$n$. As replay starts, all processors execute, but the arbiter only allows $P_i$ to commit. Because no other processor can commit, the system state that $P_i$ observes is the one at the checkpoint. As $P_i$ replays its chunk, Observation 1 tells us that the path taken by the execution inside the chunk will be the same as in the initial execution. Moreover, as per Observation 3, the number of instructions in the chunk will also be the same as in the initial execution. Finally, if the chunk was affected by an external event in the initial execution, Observation 2 tells us that the event was logged. In the replay of the chunk, we simply reproduce the logged event. Overall, the system has replayed deterministically.

We now assume that the system replayed deterministically for the first *k* committed chunks (*k*<*m*) and show that it will also do so for the *k+1* commit. At GCC=k, we know that: (i) the next processor in the PI Log (say, $P_i$) is the one that executed next in the initial execution; (ii) $P_i$ is at the same instruction as it was in the initial execution; and (iii) the system state that $P_i$'s chunk observes now in the replay is the same as it observed in the initial execution. We can use observations 1, 2 and 3 like in *m=1* to show that chunk *k+1* is replayed deterministically. Therefore, the system replays deterministically.

# CHAPTER 5

# Capo: A Software-Hardware Interface for Hardware-Assisted Deterministic Replay

## 5.1 Introduction

Chapter 4 presented DeLorean, a high-performance hardware-based deterministic replay system. Unlike software-based replay systems, DeLorean and the other hardware-based schemes can cope with multiprocessor execution efficiently: they achieve low run-time overhead and can have small logging requirements.

Unfortunately, these hardware-based systems cannot replace software-based replay systems yet because they are largely impractical for use in realistic systems. Their main problem is that they only focus on the hardware implementation of the basic primitives for recording and 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. They likely need a software layer to manage the hardware components and to handle large logs (on the order of a few gigabytes per day, in the case of DeLorean). The question is where such layer should go. It cannot be part of the operating system because its execution would be recorded in the Memory Ordering Log and that would be an issue during replay. This is why in Section 4.2.3 we said that record and replay in hardware-based systems is likely to proceed on top of a hypervisor layer — or inside a simulator [64, 65].

Moreover, hardware-based replay systems will likely also need a way to mix standard execution, recorded execution, and replayed execution of different applications in the same machine concurrently —which all software-based schemes already do. Consider, for example, a developer trying to find a bug on a small application using a hardware-based scheme. Even though she only cares about that piece of code, she would have to record and replay the entire system.

Unfortunately, providing such 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 chapter addresses this problem. It presents *Capo*, the first set of abstractions and a software-hardware interface for practical hardware-assisted deterministic replay. 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 ordering of different threads. A key abstraction in Capo is the *Replay Sphere*, which allows system designers to separate cleanly the responsibilities of the hardware and software components and isolates the software that is being recorded or replayed from the software executing in a standard manner.

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.

## 5.2 Capo's Key Abstraction: The 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.

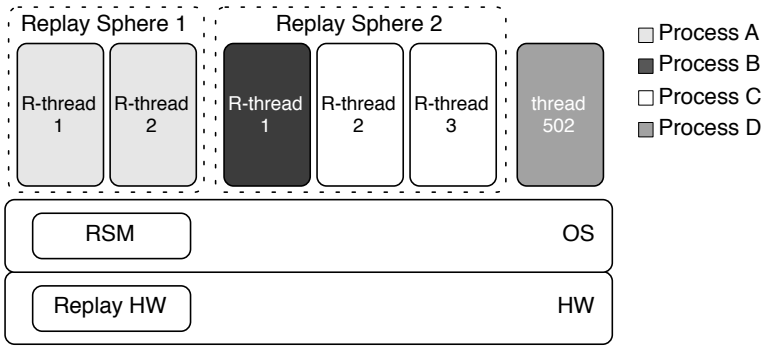



**Figure 5.1:** Architecture of Capo for an OS-level replay system. The replay system includes user-level threads running within replay spheres and a kernel-level Replay Sphere Manager (RSM) that manages the underlying replay hardware and provides the illusion of infinite amounts of replay hardware.

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.

We call this abstraction the *Replay Sphere* or *Sphere* for short (Figure 5.1). 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 this thesis, 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 5.1 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.

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.

| | |
|---|---|
| Software Duties (Per Sphere) | Assign the same R-threadIDs during recording and replay<br>Assign the same virtual addresses during recording and replay<br>Log the inputs to the sphere during recording in the Sphere Input Log<br>Inject the logged inputs back to the replay sphere during replay<br>Squash the outputs of the replay sphere during replay<br>Manage the buffers for the Memory Ordering and Sphere Input Logs |
| Hardware Duties (Per Sphere) | Generate the Memory Ordering Log during recording<br>Enforce the interleaving in the Memory Ordering Log during replay |

**Table 5.1:** Separation of the duties of the software and the hardware components in Capo.

In Capo, the replay hardware records in a log the memory ordering of the R-threads running within the same sphere. It does not record the ordering of R-threads from different spheres, nor the ordering of regular threads with respect to R-threads. The software, in turn, 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. In a sense, the hardware is in charge of recording the way the sphere's address space is modified from within the sphere, and the software is in charge of recording how the replay sphere's address space is modified from the outside.

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 most 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.

### 5.2.1 Separation of Responsibilities

With the replay sphere abstraction, Capo separates the duties of the software and the hardware components as shown in Table 5.1.

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 memory interleaving of the replay sphere R-threads into a *Memory Ordering Log*. Both of them are per-sphere logs.

During the replay phase, the hardware enforces the interleaving encoded in the Memory Ordering Log, while the software injects 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 Memory Ordering Log and the Sphere Input Log by moving data to or from the file system.

The next two sections describe in detail Capo's software (Section 5.3) and hardware (Section 5.4) components, respectively.

### 5.2.2 Finding The Replay Sphere in Other Replay Systems

This section has introduced the concept of replay sphere, a *logical boundary* between the software that is being recorded or replayed and the rest of the software in the system. To the best of our knowledge, CapoOne (a Capo-based deterministic replay system that we will describe in Chapter 6) is the first one to explicitly employ the replay sphere concept. It is possible, however, to find the replay sphere in all other deterministic replay schemes.

Consider ReVirt [21], for example. It can be seen that the VM being recorded constitutes the replay sphere. And because the VMM logs any input into the guest VM, it effectively plays the RSM's role. Notice that ReVirt does not need a Memory Ordering Log because the VM only uses one virtual processor and thus, any non-determinism must

come from outside the sphere. Of course, in software-based schemes that support multiprocessor execution, the per-sphere hardware duties that we specified in Table 5.1 are also the software's responsibility. This is the case of SMP-ReVirt [22], where the hypervisor uses the CREW protocol to record the guest's Memory Ordering Log. Note that both ReVirt and SMP-ReVirt could record and/or replay several VMs —replay spheres— simultaneously.

Consider now DeLorean —or any of the other full-system replay schemes. It is easy to see that they only record or replay a single sphere at a time, and that such sphere includes both the operating system and the applications running on top of it. This is why, in all of them, the hardware records the interleaving of all the OS and user threads in a single Memory Ordering Log. Together, their I/O and Interrupt Logs constitute the Sphere Input Log. In Section 4.2.3 we mentioned that all hardware-based systems will likely run on top of a hypervisor. This is because they need a piece of software not running in the OS —that is, outside the sphere— to manage the replay hardware and handle the Sphere Input Log and the Memory Ordering Log: *they need an RSM*.

Finally, consider Recap [45]. In Recap, the compiler inserts instructions to log the value of every load that may access shared data. In that case, the replay sphere simply includes the threads and their private memory. Consequently, shared memory is outside the sphere and thus, any read from shared data must be logged because it crosses the sphere's logical boundary.

## 5.3 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 5.1. In addition, the RSM multiplexes the replay hardware resources between spheres, giving users the illusion of supporting unlimited spheres.

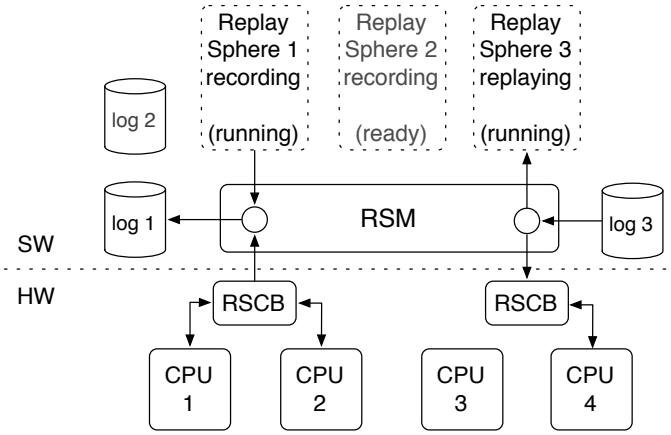



**Figure 5.2:** Logical representation of a system where the RSM manages three replay spheres. Even though CPU 3 is free, no R-thread from Replay Sphere 2 can run on it because the system only has two RSCBs, which are being used by the other two running spheres.

Figure 5.2 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 Memory Ordering 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) whose R-threads are 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 5.4. As a result, CPU 3 is idle or executing an application that is not being recorded nor replayed.

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.

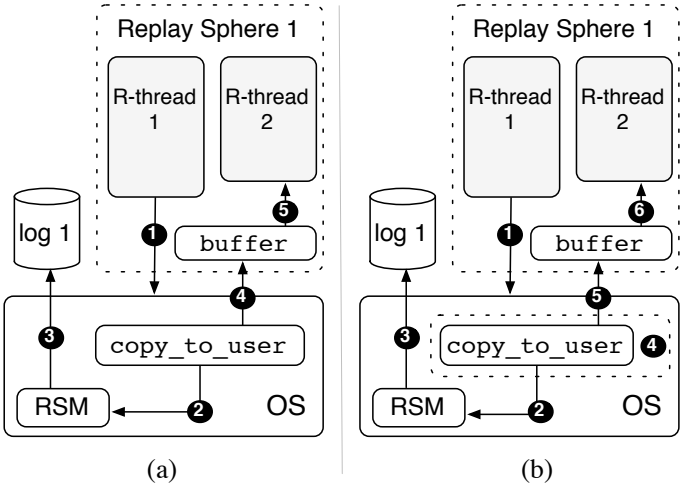



**Figure 5.3:** Race condition between the OS *copy_to_user* function and R-thread 2 (a). The data race is avoided by including *copy_to_user* in the replay sphere (b).

### 5.3.1 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[1]. This is because the memory region being written to by the OS may be accessed by R-threads at the same time. In this situation, the interleaving between the OS code that performs the writes and the R-threads that access the region may be non-deterministic. The reason is that the kernel code performing the copy is outside the sphere and, as we mentioned in Section 5.2, the replay hardware only records the ordering of R-threads running within the same replay sphere.

Figure 5.3(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.

[1] A similar argument can be made for when the OS copies data from a location within a replay sphere into the kernel.

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 5.3(b). Although this approach is efficient, it creates a less clear boundary between the code running within a 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.

### 5.3.2 Emulating and Re-Executing System Calls

In Capo, the RSM emulates most of the 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. This approach is very reminiscent of the way Flashback [57] records and replays single-threaded applications.

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

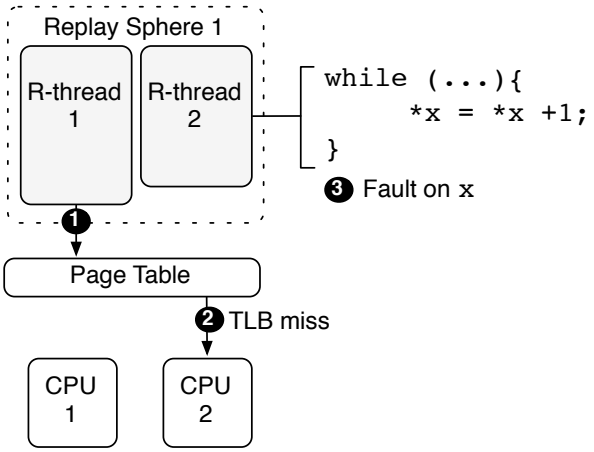



**Figure 5.4:** Example of potential non-determinism due to an implicit dependence.

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 5.4 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.

Such issue largely disappears in DeLorean and other replay schemes that use full-system recording and replay [5, 28, 41, 64, 65]. 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 5.4), 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.

### 5.3.3 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, and the R-thread that the hardware needs to replay next according to the Memory Ordering Log 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 Memory Ordering 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.

| |
|---|
| **Replay Sphere Control Block (RSCB)**: <per-sphere structure><br>Stores state and information about a running sphere.<br>Registers Mode: Current execution mode of the replay sphere<br>Base, Limit & Current: Pointers to Memory Ordering Log |
| **R-Thread Control Block (RTCB)**: <per-processor structure><br>Identifies the R-thread and sphere running on the processor<br>Registers R-ThreadID: ID of the R-thread running on the processor<br>RSID:ID of the replay sphere using the processor |
| **Interrupt-driven buffer interface**<br>Raises interrupt when the Memory Ordering Log gets full during initial execution or empty during replay |

**Table 5.2:** Capo's hardware components.

## 5.4 Hardware Support

Capo augments the replay hardware with the hardware components shown in Table 5.2. 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 are very simple and can be implemented in different ways to support DeLorean or any other of the proposals for hardware-based deterministic replay: FDR [64], BugNet [42], RTR [65], Strata [41], or Rerun [28].

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 5.2, its state in the RSCB is saved to memory. To prevent the situation depicted in the figure, where an sphere cannot execute even though there are free processors in the system, an ideal machine configuration would have as many RSCBs as processors. If the machine depicted in Figure 5.2 had four RSCBs, Sphere 2 could be running as well. Note that 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 Memory Ordering 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 Memory Ordering 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 Memory Ordering 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 hard-

ware, 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.

# CHAPTER 6

# CapoOne: A DeLorean-based Implementation of Capo

## 6.1 Introduction

To evaluate Capo, we design and build a prototype of a deterministic replay system for multiprocessors that we call *CapoOne*. CapoOne is the first implementation of Capo's abstractions and interfaces described in Chapter 5. The current prototype uses a DeLorean-based replay hardware substrate and, on top of it, it runs a standard Ubuntu Linux [10] operating system with a slightly modified kernel.

CapoOne combines the best of the hardware- and software-based deterministic replay schemes: it has good multiprocessor performance and it is able to record and/or replay unmodified standard Linux applications such as the Apache [4] web server. Moreover, it can record and/or replay two or more user applications that run both independently and simultaneously. In this chapter, we first describe CapoOne's software and hardware components and then we present some practical lessons that we learned during its development.

## 6.2 Software Implementation

Figure 6.1 depicts CapoOne's architecture. It can be seen in the figure that CapoOne's RSM is split into two different components, one in user space and another inside the kernel. The user-level RSM tracks processes as they run using the Linux *ptrace* process tracing mechanism. Ptrace gives processes the ability to create child processes, receive no-

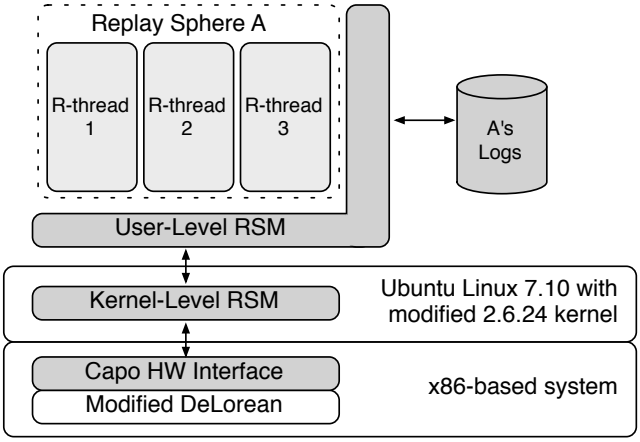
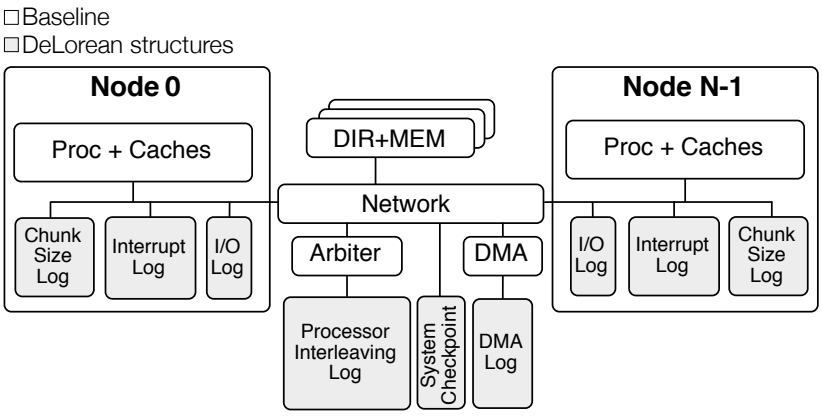



**Figure 6.1:** CapoOne's architecture.

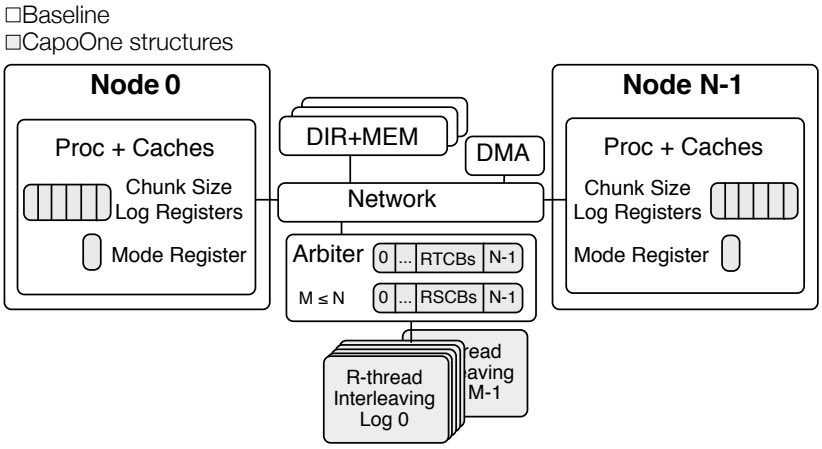
tification on key events, and access arbitrary process states as the child process runs. In addition, the user-level RSM is in charge of storing the logs into the disk during recording and retrieving them during replay. The kernel-level RSM, in turn, manages the replay hardware, schedules replay spheres and records the data copied between the kernel and the replay sphere. The kernel-level RSM is developed as a kernel module to minimize the changes to the kernel as much as possible. In Section 6.4.1 we will talk more about such RSM design and explain its consequences

Figure 6.1 also shows that CapoOne runs on Ubuntu Linux 7.10 with a custom 2.6.24 kernel. We modify the 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 kernel-level RSM manages these structures by saving and restoring them into hardware RSCBs and RTCBs during context switches.

We also modify 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 depen-




(a)




(b)

**Figure 6.2:** Multiprocessor with the DeLorean hardware as presented in Chapter 4 (a), and as implemented in CapoOne (b).

dences. We make page table modifications and the resulting TLB flushes atomic, to avoid race conditions with R-threads running on other CPUs.

## 6.3 Hardware Implementation

In CapoOne, we use DeLorean's *OrderOnly* as the underlying replay hardware and we augment it with the Capo interface of Table 5.2. This section describes the implementation in detail and, to give more insight, it also outlines a possible implementation for FDR [64] and similar schemes.

Figure 6.2(a) shows a DeLorean multiprocessor. The figure is similar to Figure 4.1, but we reproduce it here so it is easier to compare to Figure 6.2(b). In general, CapoOne's extends DeLorean so that it can handle multiple Memory Ordering Logs simultaneously, one per sphere. Thus, in CapoOne, each sphere has a Memory Ordering Log. And each sphere's Memory Ordering Log includes a R-thread Interleaving Log (RI Log) and a per-R-thread Chunk Size Log (CS Log). The figure also shows that CapoOne implements the interface of Table 5.2 mostly in the arbiter module. In addition, there are a few other changes to make to the DeLorean architecture because DeLorean is 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 6.2(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 RI Log. In Figure 6.2(b), we show the case when there are $M$ active replay spheres and, therefore, $M$ Memory Ordering Logs. Note that $M \leq N$.

For performance reasons, each node also has some special hardware registers, namely the Mode register and the CS Log registers. The former contains the mode of the sphere that is currently using the processor; the latter contains the top of DeLorean's CS 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 CS 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 CS Log registers depending on whether the Mode register indicates Recording or Replaying mode, respectively. If the CS 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 CS 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 RI Log is accessed. If the mode is Recording and the chunk can commit, the hardware adds an entry in the RI Log and tags it with the R-threadID. If, instead, the mode is Replaying, the hardware reads the next RI 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 RI Log entry is popped. There are no changes to the encoding of messages to or from the arbiter.

Since the DeLorean architecture 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 6.2(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 6.2(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.

### 6.3.1 Hardware Implementation for FDR-like Schemes

We now outline how the hardware interface of Table 5.2 can be implemented for FDR [64] 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.

## 6.4 Lessons Learned During the Development of CapoOne

The development of the CapoOne prototype required a 18 man-month effort. In this time, we: i) implemented a new hardware simulator for the DeLorean underlying architecture, ii) modified the Linux Kernel and iii) developed the RSM. During that period we gained some insights into key issues in experimental replay systems that we describe now.

This section first discusses the problems we faced because of our split-RSM im-

plementation. In hindsight, our biggest mistake was to underestimate the time and complexity of the software components. The section then describes the issues that we faced when we converted a full-system, hardware-based replay system such as DeLorean into a hardware-assisted replay system for user applications. A key issue was to make sure that CapoOne only recorded and replayed instructions that belonged to the target applications.

The section also explains how CapoOne deals 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, 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. Finally, this section details 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.

### 6.4.1 Implementing the RSM

As we mentioned in Section 6.2, we implemented a large portion of the RSM in user mode using the *ptrace* process tracing mechanism of Linux to interpose on recording and replaying processes. Implementing the RSM in user mode makes it easier to develop, since writing user mode code gives us access to more libraries and debugging tools, and greater flexibility compared to writing kernel mode code. Indeed, splitting the RSM made our initial prototype easier to implement. Unfortunately, it had two unintended consequences.

First, it added substantial overhead during recording to our very first prototype. For example, when a user mode RSM interposes on a system call invocation of a recording or replaying process, it requires four context switches and a significant number of kernel-level subsystems to carry out the interposition. In contrast, a kernel mode RSM would

interpose on system calls by simply adding an extra function call on the system call path within the OS to divert control to the RSM. 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.

Fortunately, most of our user-level RSM overhead occurs only during system call invocations. Therefore, if the system call frequency is low, the overhead of our implementation is small. Even if the system call frequency is high, if the application is such that CPUs are often idle, RSM execution does not cause much overhead. However, if the application is both CPU bound and system call intensive, our implementation does add some overhead.

The second consequence of using ptrace for our RSM implementation is that it made debugging much harder. Ptrace's interface and kernel paths are sometimes obscure and, towards the end of our development effort, most of the hard-to-debug bugs would have been avoided if we had implemented our RSM exclusively inside the kernel.

One surprising aspect of our RSM implementation is that we did not need to use a versioning file system [27, 54] 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 [21]. However, experiments showed that the RI Log generated by the hardware 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. Note that, had we used DeLorean's *PicoLog* — whose Memory Ordering Log is an order of magnitude smaller than *OrderOnly*'s — a versioning file system might have been required.

**Lessons learned:** Splitting the RSM into user-level and kernel-level components increases its complexity and makes it harder to debug in the long run, even though it can greatly help bringing up the first version of the prototype. A versioning file system might not be required if the Memory Ordering Log is much larger than the Sphere Input Log.

### 6.4.2 From Full-System Replay to Sphere-Based Replay

As we discussed in Section 2.4.1, previous hardware-based deterministic proposals record and replay the entire system. CapoOne, on the other hand, 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 it 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 Memory Ordering 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 Memory Ordering Log.

**Lesson learned:** Making sure that the Memory Ordering 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.

### 6.4.3 User to Kernel Transitions

Section 6.4.2 described a very important design principle in CapoOne: all chunks 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.

#### 6.4.3.1 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 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 Memory Ordering Log.

Due to CapoOne's chunk-based execution, the interrupt delivery policy balances three conflicting demands: size of the Memory Ordering 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 Now* and *Squash Now*.

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 CS Log.

In *Commit Now* 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 CS Log. However, the interrupt response time is better than in *Finish First*.

Finally, in *Squash Now*, 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 Now* and *Finish First* do not require any information to be recorded in the Sphere Input Log — *Finish First* does write into the RI Log as any other normal chunk commit, though. This is why we use *Squash Now* in CapoOne. Moreover, it is simple to implement and did not cause any significant overhead in our experiments.

#### 6.4.3.2 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

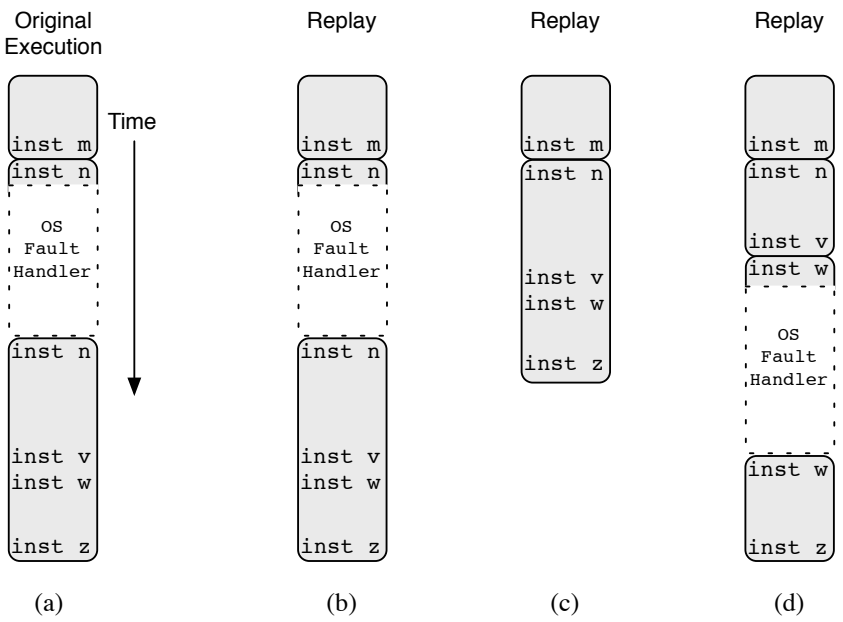



**Figure 6.3:** Fault handling in CapoOne.

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; we now describe them in turn.

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.

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 6.3(a). Instruction $n$ causes a page fault, so CapoOne must prema-

turely commit a chunk that finishes at instruction $m$. It must also log the chunk's size in the CS 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 6.3(b), 6.3(c) and 6.3(d) show that there are three possible behaviors when CapoOne replays a fault.

In Figure 6.3(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 6.3(c). As in Figure 6.3(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 6.3(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 6.3(a).

CapoOne handles traps and programmed exceptions differently. Unlike faults, processors raise them *after* the execution of a trapping instruction. Therefore, the next instruction to execute after the OS handles the event is the instruction following the one that caused the exception. Because processors must exit the replay sphere to let the OS man-

age these events, they 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:** Faults are non-deterministic events that must be handled properly. The most difficult case is depicted in Figure 6.3(d), where a fault occurs in replay and not in the initial execution. However, traps and programmed exceptions are deterministic events and require no extra logging.

### 6.4.4 System Issues

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

#### 6.4.4.1 Moving Data Between the Kernel and the Replay Sphere

As we described in Section 5.3, copying data from the kernel into the replay sphere is a delicate matter in Capo-based systems such as CapoOne. First, the RSM must record in the Sphere 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.

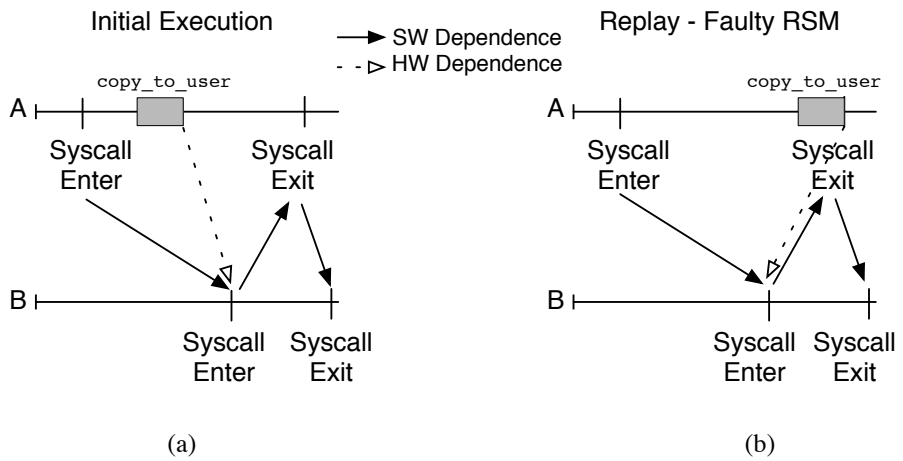
Making sure that the interleaving between `copy_to_user` and the R-threads in the sphere was deterministic gave us many headaches. In our first implementation, the RSM associated the data copied by `copy_to_user` with the system call exit event corresponding to the system call that executed `copy_to_user`. At that time, it made sense that the Sphere 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 Sphere Input Log and the Memory Ordering Log. To understand why, the reader must note that the RSM records in the Sphere Input Log the order in which R-threads enter and exit system calls. During replay, the RSM enforces the same ordering.

Consider Figure 6.4(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 Sphere Input Log, $B$'s system call enter comes before $A$'s system call exit event. In the Memory Ordering 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.

Consider now Figure 6.4(b). In the figure, the system deadlocks while trying to replay the execution from Figure 6.4(a). The reason is that the RSM associated the `copy_to_user` event with the system call exit event in the Sphere Input Log and it is waiting for thread $B$ to execute a system call. At the same time, the hardware follows the Memory Ordering 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




(a) (b)

**Figure 6.4:** Circular dependences between the Sphere Input Log and the Memory Ordering Log cause deadlocks during replay.

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 Sphere Input Log and the Memory Ordering Log. These dependences can cause deadlocks under certain interleavings. We recommend making `copy_to_user` events a first-order event in the Sphere Input Log.

#### 6.4.4.2 Cache Overflows

BulkSC-based [13] 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. While certain transactional memory schemes allow storing speculative state in main memory, this is not the case for 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.

#### 6.4.4.3 Handling 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. Self-modifying code is usually defined as code that alter its own instructions as it executes. Intel microprocessor manuals [29] distinguish between self- and cross-modifying code. They define the former as:

> The act of a processor writing data into a currently executing code segment with the intent of executing that data as code

And the latter as:

> The act of one processor writing data into the currently executing code segment of a second processor with the intent of having the second processor execute that data as code.

In this thesis, we will adopt this classification. CapoOne uses signatures to detect cross-modifying code by adding all instruction fetches to the chunk's read signature. 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 Memory Ordering Log since maintaining the same ordering of chunks at replay time is sufficient for ensuring that the cross-modifying event is deterministically replayed. During replay, signatures are also used to invalidate the instruction caches.

Detecting and handling self-modifying code is harder because modern processors can have many in-flight instructions. Thus, it is possible that the code being modified by a retiring instruction is already inside the pipeline; in fact, it might have executed already. Because of this, modern processors require a serializing instruction before the processor can start executing the new code [29]. Similarly, in a BulkSC based system such as CapoOne, such instruction also causes a deterministic, early chunk commit. The problem, however, is how to replay execution if the serializing instruction is missing. We consider three different approaches to this issue.

The first one is to do nothing and, therefore, do not support recording and replaying of non-deterministic self-modifying code. The reasoning behind this approach is that it is not the role of the replay subsystem to deal with undefined behavior within the processors.

The second approach is to modify the processor so that it can detect when an instruction is writing into instruction memory. In CapoOne, this can be done by using a new *In-flight Instruction Signature* (IIS) that stores the addresses of the in-flight instructions. The IIS is a counter-based Bloom filter so that instructions can be removed from it once they retire. It is similar to the In-flight Conflict Detector (ICD) structure proposed by Tuck *et al.* [60], except that the ICD only stores the addresses of the in-flight loads. Whenever a store retires, it intersects its target address with both the IIS and the read signature. If the intersection is not null, the processor early commits the current chunk —whose last

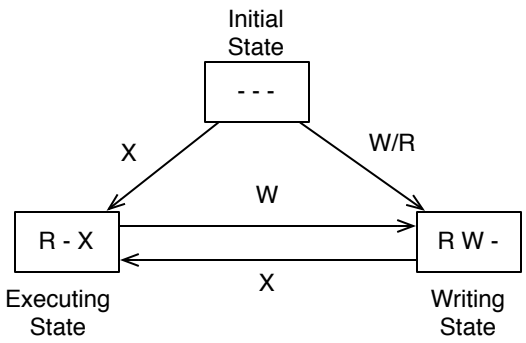



**Figure 6.5:** Permission transition diagram for a software-only, self-modifying code detection mechanism using page protections.

instruction is the retiring store— and logs its size in the CS Log. Unfortunately, this approach requires non-trivial changes to the processor.

The final approach is to rely on software to detect self-modifying code. In this case, the RSM ensures that no page can be written and executed at the same time. Thus, when an instruction tries to write into an executable page, the processor raises a fault, which causes an early chunk commit. The RSM then changes the permissions of the page from $(r, -, x)$ to $(r, w, -)$ and execution continues. If the processor tries to execute code from the same page, a new page fault is raised, the chunk is truncated early and the page's permissions are set back to $(r, -, x)$. Figure 6.5 illustrates this mechanism. The main drawback of this approach is that it can hurt performance significantly.

**Lesson learned:** Cross-modifying code events they can be handled seamlessly with the same hardware support (signatures and squashes). Self-modifying code, on the other hand, can be hard to record and replay deterministically due to existing non-determinism in current processors.

# CHAPTER 7

# Evaluation Setup

## 7.1 Introduction

In this chapter we describe the simulation environments that we use to evaluate DeLorean and CapoOne. For DeLorean, we use the SESC [51] cycle-accurate execution-driven simulator. SESC models the processor and memory subsystem in detail. Unfortunately, it does not support OS execution, which is required to evaluate CapoOne. Consequently, we evaluate CapoOne using the SIMICS [37] full-system architecture simulator enhanced with a detailed model of the DeLorean hardware.

## 7.2 DeLorean's Evaluation Environment

We use the SESC to evaluate an 8-processor DeLorean Chip Multiprocessor (CMP). We compare the speed of DeLorean's execution and replay to three other systems. The first one is the CMP of Table 7.1 under RC with speculative execution across fences and hardware exclusive prefetching for stores. We call it *RC*. The second is the CMP of Table 7.1 under an aggressive SC implementation that includes speculative execution of loads and hardware exclusive prefetching for stores. We call it *SC*. Note that neither the RC nor the SC systems support bulk execution, speculative tasking, or logs. Finally, our third baseline architecture is a BulkSC system without support for deterministic replay. We call it *BulkSC* and it uses the architectural parameters shown in Table 7.2. These parameters are largely like those used in [13].

| Processor | Memory |
|---|---|
| Processors: 8 in a CMP<br>Frequency: 5.0 GHz<br>Fetch/issue/comm width: 6/4/5<br>I-window/ROB size: 80/176<br>LdSt/Int/FP units: 3/3/2<br>Ld/St queue entries: 56/56<br>Int/FP registers: 176/90<br>Branch penalty: 17 cyc (min) | Private wback D-L1:<br>32KB/4-way/32B-lines<br>Round trip: 2 cycles<br>MSHRs: 8 entries<br>Shared L2:<br>8MB/8-way/32B-lines<br>Round trip: 13 cyc min<br>MSHRs: 32 entries<br>Mem round trip: 300 cyc |

**Table 7.1:** Configuration for the two non-chunked baseline architectures, RC and SC.

| Processor | Memory | BulkSC |
|---|---|---|
| Processors: 8 in a CMP<br>Frequency: 5.0 GHz<br>Fetch/issue/comm width: 6/4/5<br>I-window/ROB size: 80/176<br>LdSt/Int/FP units: 3/3/2<br>Ld/St queue entries: 56/56<br>Int/FP registers: 176/90<br>Branch penalty: 17 cyc (min) | Private wback D-L1:<br>32KB/4-way/32B-lines<br>Round trip: 2 cycles<br>MSHRs: 8 entries<br>Shared L2:<br>8MB/8-way/32B-lines<br>Round trip: 13 cyc min<br>MSHRs: 32 entries<br>Mem round trip: 300 cyc | Signature: 2 Kbits<br>Commit arbitration<br>latency: 30 cyc<br>Max. concurrent<br>commits: 4<br># Simultaneous chunks<br>per processor: 2<br># of arbiters: 1<br># of directories: 1 |

**Table 7.2:** Configuration for the BulkSC baseline architecture.

We use the BulkSC configuration shown in Table 7.2 as the foundation of our DeLorean system. We evaluate the three execution modes discussed in Section 4.2.1 — *Order&Size*, *OrderOnly* and *PicoLog* — using the parameters shown in Table 7.3. Note that all three execution modes also use the BulkSC configuration . Specifically, *Order&Size* uses chunks of at most 2,000 instructions, variable-sized CS Log entries (1 bit if the chunk has maximum size or 12 bits otherwise), and 4-bit PI Log entries. The latter encode the IDs of the 8 processors and the DMA. To model an environment with variable-sized chunks, we artificially truncate the chunk probabilistically: we truncate 25% of the chunks in *Order&Size*, giving them a size between 1 and the maximum size using a uniform probability distribution.

| | *Order&Size* | *OrderOnly* | *PicoLog* |
|---|---|---|---|
| Chunk Size | 2000 inst. max.<br>25%chunks $<$ 2000inst | 2000 inst. | 1000 inst. |
| CS Log Entry | Variable-sized:<br>1 bit if chunk size==2000inst<br>12 bit otherwise | 21 bit distance<br>11 bit distance | 22 bit distance<br>10 bit distance |
| PI Log Entry | 4 bit procID | 4 bit procID | – |

**Table 7.3:** DeLorean's preferred configurations.

*OrderOnly* uses 2,000-instruction chunks, 32-bit CS Log entries (which include 11 bits for the truncated chunk size and, in lieu of `chunkID`, 21 "distance" bits for the number of chunks committed by the processor since its most-recent truncated chunk), and 4-bit PI Log entries. *PicoLog* uses 1,000-instruction chunks, 32-bit CS log entries, and round-robin processor commit order. In our experiments with different chunk sizes in *OrderOnly* and *PicoLog*, we keep the CS Log entry size constant, thus changing the distance bits. Our simulator models both initial execution and replay.

All log buffers are enhanced with compression hardware that uses the LZ77 algorithm [68].

We assume that the performance of the initial execution in FDR [64], Strata [41], and Basic RTR [65] is similar to that of *SC* — therefore assuming that recording induces no overhead on these schemes. This is consistent with the results reported in their papers. Finally, we estimate the performance of Advanced RTR using data on Processor Consistency (PC) performance.

As applications, we use SPLASH-2, SPECjbb2000 and SPECweb2005. The SPLASH-2 codes are evaluated without system references. They run to completion, and include all applications but Volrend (which fails in our infrastructure). Both SPECjbb2000 and SPECweb2005 are evaluated by interfacing the SIMICS full-system simulator as a front-end to our SESC simulator. Therefore, we capture system references as well. SPECjbb2000

is configured to use 8 warehouses, while SPECweb2005 runs the e-commerce workload. Each runs for over 1 billion instructions after warm-up.

## 7.3 CapoOne's Evaluation Environment

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 SIMICS to 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 from Table 7.2. 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*.

# CHAPTER 8

# DeLorean Evaluation

## 8.1 Log Size

Figure 8.1 shows the size of the PI Log and CS Logs in *OrderOnly*, measured in *bits per kilo-instruction*. We evaluate configurations with standard chunk sizes of 1,000, 2,000 and 3,000 instructions. For each of them, we report the size of both logs with and without compression. In the figure, the CS Log contribution is stacked atop the PI Log's, but it is too small to be seen. The SP2-G.M. bars correspond to the geometric mean of SPLASH-2. For comparison, the figure shows a line with the average size of the compressed Memory Ordering Log in Basic RTR from [65]. We will use this line as a reference, although we note that the set of applications measured here and in RTR [65] are different.

The figure shows that our preferred 2,000-inst. *OrderOnly* configuration uses on average only 2.1 bits (or 1.3 bits if compressed) per kilo-instruction to store both the PI Log and CS Logs (that is, the Memory Ordering Log). This means that these compressed logs use only 16% of the space that we estimate is needed by the compressed Memory Ordering Log in Basic RTR.

Figure 8.1 also shows that the size of the CS Log is negligible. Moreover, as we increase the standard chunk size, the size of the PI Log decreases. This is because there are fewer chunks to commit. However, chunks are also more likely to conflict, and the potentially higher number of squashes may affect performance.

Figure 8.2 shows the space required by the CS Log in *PicoLog*. Recall that *PicoLog* has no PI Log. We see that the CS Log needs 0.37 bits or fewer per kilo-instruction in

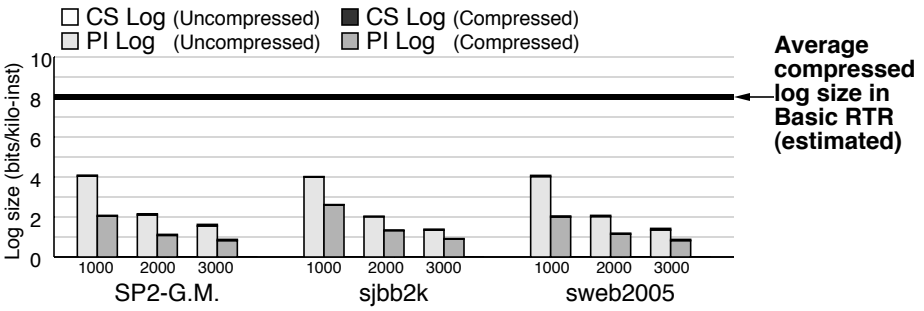



**Figure 8.1:** Size of the PI Log and CS Logs in *OrderOnly*. The numbers under the bars are the standard chunk sizes in instructions.

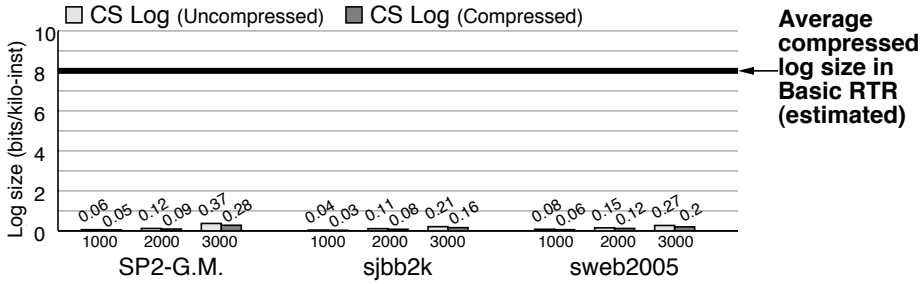



**Figure 8.2:** Size of the CS Log in *PicoLog*. Recall that *PicoLog* has no PI Log. The numbers under the bars are the standard chunk sizes in instructions.

all cases — even without compression. Our preferred 1,000-inst. *PicoLog* configuration needs a compressed log with an average of only 0.05 bits per kilo-instruction. To put this in perspective, it implies that, if we assume an IPC of 1, the combined effect of all eight 5-GHz processors is to produce a log of only about 20GB per day. This is a very small log. It is 0.6% of the estimated size needed by the compressed Memory Ordering Log in Basic RTR. Since the CS Log entries are due to chunk truncation caused by attempted cache overflow, we see that such an event is rare.

Figure 8.3 shows *Order&Size*'s PI Log and CS Logs sizes. We can see that this execution mode requires larger PI Log and CS Logs, sometimes comparable to Basic RTR log sizes. Our preferred 2,000-inst. compressed configuration uses, on average, 3.7 bits per kilo-instruction. This is 46% of the estimated space needed by the compressed Memory Ordering Log in Basic RTR.

So far, we have roughly compared the per-processor log size of two schemes: De-

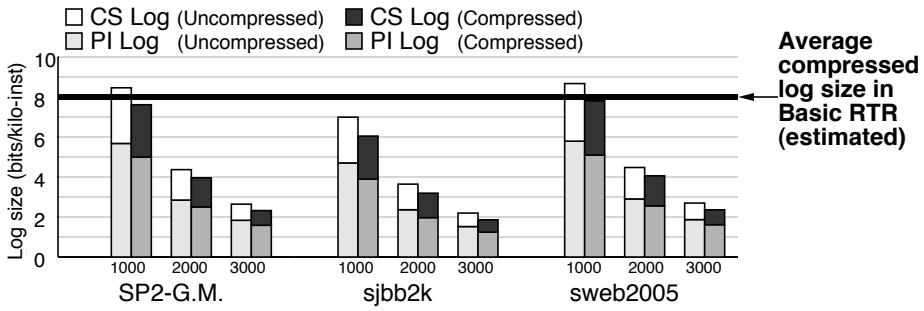



**Figure 8.3:** Size of the PI Log and CS Logs in *Order&Size*. The numbers under the bars are the maximum chunk sizes in instructions.

Lorean's 8-processor runs and Basic RTR's 4-processor runs. To compare to Strata, we can use the fact that both the Strata [41] and RTR [65] papers quantitatively compare their schemes' log sizes to FDR's. Alternatively, we can use the numbers in the Strata paper — which measure different applications than we do and are again for only 4-processor runs. In this case, the Strata paper claims a compressed log size of 2.2KB per million memory operations for the 4 processors combined. DeLorean needs 364B and 13.7B per million memory operations in *OrderOnly* and *PicoLog*, respectively. This is 16% and 0.6%, respectively, of the space Strata claims to need. However, if, to speed-up Strata's replay, we also add WAR dependences in Strata's log, Strata's log size increases by 25% [41]. In addition, since the size of a Strata log entry is proportional to the number of processors, Strata's log size may increase substantially for 8-processor runs.

### 8.1.1 Stratifying the PI Log

Figure 8.4 compares the size of the PI Log in 2000-inst. *OrderOnly* without and with stratification. We consider three Stratified PI Log designs, which differ in the maximum number of committed chunks allowed per processor per stratum, namely 1, 3, or 7. The bars are normalized to the non-stratified design. We can see that stratifying the PI Log while allowing 1 or 3 committed chunks per processor per stratum saves log space. In the case of 1 chunk per processor per stratum, the PI Log size decreases by an average

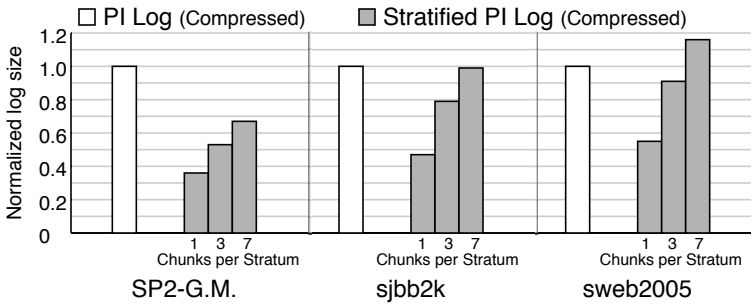



**Figure 8.4:** Size of the PI Log in *OrderOnly* without and with stratification. The numbers under the bars are the maximum number of chunks per processor per stratum.

of 54%. This results in an average total *OrderOnly* log size of about 0.6 bits per kilo-instruction, or 7.5% of the estimated Basic RTR log size. Allowing 7 chunks per processor per stratum results in wasted space and larger logs in SPECweb2005.

## 8.2 DeLorean's Performance

Figure 8.5 compares the performance of *RC* and *SC* to that of the initial execution under each of the three DeLorean modes plus the Stratified *OrderOnly* with one chunk. For comparison purposes, we also show the performance of a BulkSC environment. All bars are normalized to the performance of *RC*.

The figure shows that the average performance of *Order&Size* and *OrderOnly* is only 2-3% lower than that of *RC*. Moreover, some of this reduction is the result of running under BulkSC (which causes some chunk squashes), as can be seen by comparing to the *BulkSC* bar. Consequently, we conclude that DeLorean's logging support causes negligible slowdown. The figure also shows that Stratified *OrderOnly* delivers a performance similar to *OrderOnly*. Stratification, therefore, has negligible performance impact.

The figure also shows that *PicoLog* has a lower performance — on average, execution proceeds at 86% of *RC*'s speed. This is still faster than *SC*, which averages 79% of *RC*. As we will see in Section 8.3, *PicoLog*'s lower performance is less caused by load

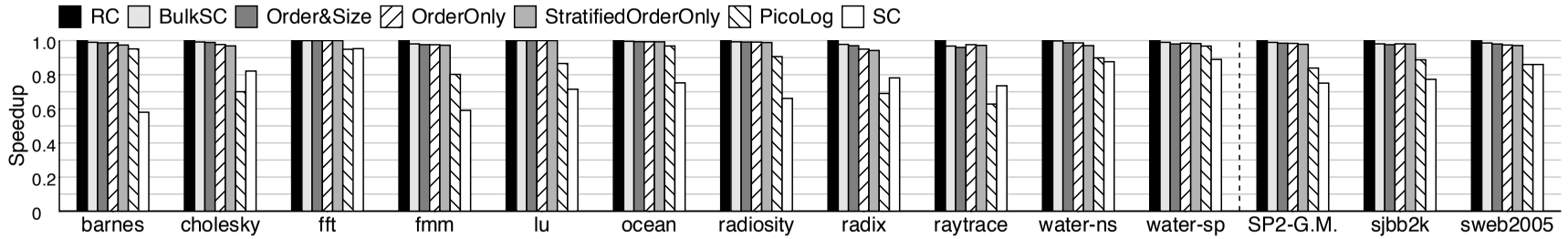



**Figure 8.5:** Performance during initial execution normalized to *RC*.

imbalance due to round-robin commit ordering than to chunk squashes. This problem especially affects `raytrace`. Overall, it can be argued that a performance 14% lower than *RC* is a modest price to pay for a deterministic replay system that only logs 0.05 bits per kilo-instruction — or 20GB per day for the combined eight 5-GHz processors (Section 8.1).

Given that FDR, Strata, and Base RTR have also been shown to have negligible recording overhead, we estimate their performance with the *SC* bar — which is a fairly aggressive implementation of sequential consistency. It is seen in the figure that all DeLorean execution modes on average outperform *SC*, typically substantially. This is because, through chunk-based execution, DeLorean allows for very aggressive reordering and overlapping of accesses.

If we estimate the performance of Advanced RTR to be that of the machine supporting TSO, we can compare Advanced RTR to DeLorean. TSO's performance is similar to that of Processor Consistency (PC). Since our infrastructure does not model TSO or PC, we simply note that previous work showed that PC's performance is significantly lower than RC's [25, 49] — hence significantly lower than at least that of *OrderOnly* and *Order&Size*. Quantitative comparisons are not possible due to the use of different applications.

### 8.2.1 Performance During Replay

We use our replay simulator to estimate the performance of DeLorean's replay. Since replay will likely occur under a virtualized environment, we penalize the replay speed by disabling parallel commit and increasing the commit arbitration latency in the arbiter from 30 to 50 cycles. Moreover, in our simulator, we add random delays to the replay execution to ensure that the timings are different from the initial execution. Specifically, we take the PI Log from the initial execution and use it in 5 different replay runs. In each run,

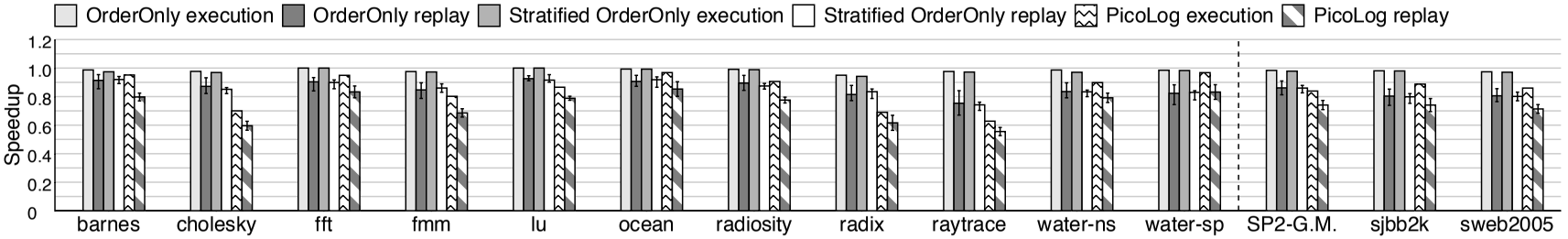



**Figure 8.6:** Performance of several environments during initial execution and replay. All bars are normalized to *RC*.

we add from 10 to 300-cycle stalls before a randomly-selected 30% of the commit operations. We also change the delay of 1.5% randomly-selected cache hits to that of cache misses and the same number of cache misses to cache hits. Finally, we report the average performance of the 5 runs.

Figure 8.6 compares the performance of *OrderOnly*, Stratified *OrderOnly* with one chunk, and *PicoLog* during initial execution and replay. All bars are normalized to *RC*. From the figure, we see that, on average, both *OrderOnly* and Stratified *OrderOnly* replay at 82% of *RC*'s speed, while *PicoLog* replays at 72% of *RC*. Several factors contribute to the lower performance of replay, namely the penalties added, the stall of processors waiting to commit, and two effects due to keeping several completed but uncommitted chunks: additional squashes and cache overflows. However, we believe that, at these speeds, deterministic replay opens up new possibilities in concurrency debugging.

Stratifying *OrderOnly* with one chunk does not appear to hurt replay performance. Overall, Stratifying *OrderOnly* has reduced the log size by half, at some hardware cost, without noticeably impacting the speed of execution recording or replay.

## 8.3 Characterizing Picolog

We perform a sensitivity analysis to determine how *PicoLog*'s performance changes with (i) the number of processors in the CMP, (ii) the standard chunk size in committed instructions, and (iii) the maximum number of chunks that a processor may be executing and are not yet committed. We called the latter the number of simultaneous chunks per processor in Table 7.2. Figure 8.7 shows the resulting performance of *PicoLog* relative to *RC* for the same number of processors. Because our infrastructure does not support the commercial applications on 16 processors, the data in the figure corresponds to SPLASH-2 only.

Increasing the number of processors reduces *PicoLog*'s relative performance. For

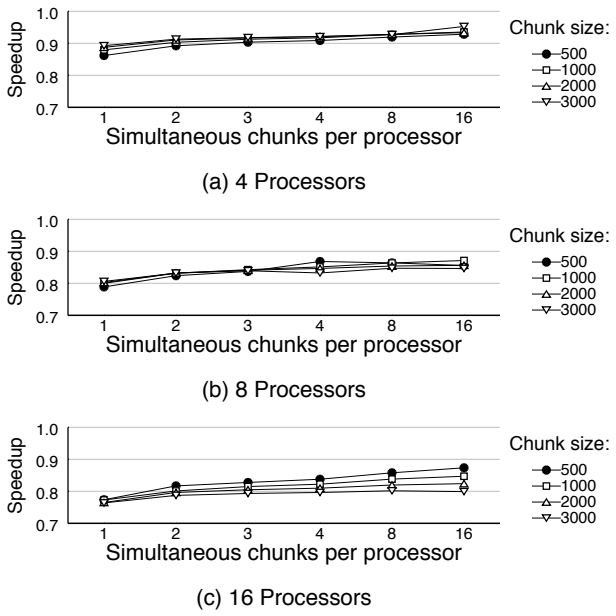



**Figure 8.7:** *PicoLog* performance relative to *RC*.

example, with one chunk per processor and 1000-inst. chunks, the performance drops from 87% for 4 processors to 77% for 16. This is because there are more squashes and because it takes longer for a given processor to get its turn to commit.

The latter can be partly mitigated by increasing the number of simultaneous chunks per processor. These additional chunks keep the processor busy while the chunk waits for its turn to commit. However, the figure shows that we quickly get diminishing returns. The more chunks we add, the higher the chance for chunk collisions and attempted cache overflows. In our baseline design (Table 7.2), we used two simultaneous chunks per processor.

Finally, larger chunk sizes have little effect for 4- or 8-processor systems, but hurt performance for 16-processor systems. Large chunks with many processors tend to induce more conflicts.

Table 8.1 characterizes *PicoLog* for 8 processors. The *Parallel Commit* columns show data on the commit process. The *Ready Procs* column shows the average number of processors with fully-executed, ready-to-commit chunks at a given time. On average,

| Appl. | Parallel Commit | | Commit Token Passing | | | | |
|---|---|---|---|---|---|---|---|
| | Ready Procs (Avg) | Actual Commit (Avg) | Proc Ready (%) | Wait Token (Cyc) | Wait Cplete (Cyc) | Token Rndtrip (Cyc) | Stall Cycles (%) |
| barnes | 4.0 | 2.4 | 80.4 | 499 | 230 | 661 | 4.9 |
| cholesky | 5.3 | 3.0 | 84.7 | 750 | 431 | 793 | 29.4 |
| fft | 3.5 | 2.3 | 77.4 | 411 | 478 | 889 | 2.5 |
| fmm | 5.1 | 3.0 | 84.0 | 739 | 386 | 788 | 20.4 |
| lu | 3.9 | 2.3 | 79.5 | 487 | 207 | 757 | 5.4 |
| ocean | 3.9 | 2.5 | 78.4 | 1067 | 760 | 1601 | 4.2 |
| radiosity | 4.9 | 2.9 | 82.7 | 670 | 403 | 758 | 9.3 |
| radix | 2.5 | 2.3 | 65.6 | 524 | 1119 | 3262 | 0.3 |
| raytrace | 4.6 | 2.5 | 78.4 | 1290 | 691 | 1462 | 34.0 |
| water-ns | 4.4 | 2.6 | 80.9 | 541 | 249 | 659 | 9.4 |
| water-sp | 4.6 | 2.6 | 82.1 | 489 | 203 | 575 | 2.3 |
| SP2-G.M. | 4.2 | 2.6 | 79.3 | 638 | 403 | 956 | 6.0 |
| sjbb2k | 5.1 | 3.0 | 77.5 | 1634 | 694 | 1841 | 7.2 |
| sweb2005 | 5.2 | 2.9 | 83.7 | 1002 | 612 | 1346 | 8.7 |

**Table 8.1:** Characterizing *PicoLog*.

there are 4.2-5.2 such processors. However, not all of them can commit. Indeed, while chunk commits may overlap if there are no conflicts, they are initiated in a round-robin manner. Consequently, if processor *i* is not ready to commit, *i+1* cannot commit. The *Actual Commit* column shows the average number of chunks that end up committing at the same time. The average number is 2.6-3.0.

The *Commit Token Passing* columns characterize how the "commit token" is passed around processors. *Proc Ready* is the percentage of time that a processor is ready to commit when it acquires the commit token. On average, it is 77-84%. For those processors that are ready, the *Wait for Token* column is the number of cycles elapsed from when they completed the chunk until they acquire the token; for those processors that are not ready, the *Wait for Complete* column is the number of cycles elapsed from when they receive the token until they complete the chunk. Both of these two numbers must be smaller than the *Token Roundtrip*, which is the number of cycles it takes for the token to circulate through all processors once. Such number is about 600-3,300 cycles. Finally, *Stall Cycles* shows

the fraction of cycles that processors stall because they have completed two simultaneous chunks and not received the token. On average, this number is 6-9%.

Table 8.1 explains the low performance of some codes in Figure 8.5. For example, consider `raytrace` and `radix`. In `raytrace`, it can be shown that the squashes are concentrated on a few processors, which slow down the passing of the token for everyone. As a result, processors complete the chunk before receiving the token (the *Wait for Token* cycles are 1,290) and stall often (the fraction of stall cycles is 34.0%). In `radix`, it can be shown that squashes are spread over many processors. As a result, processors receive the token before chunk completion (the *Wait for Complete* cycles are 1,119) and stall little (the fraction of stall cycles is 0.3%).

Finally, DeLorean induces more network traffic than *RC* because of signature traffic chunk squashes. It can be shown that the traffic in *Order&Size* and *OrderOnly* is practically the same as in a plain BulkSC system which, in turn, is on average 9% higher in bytes than in *RC* [13]. In *PicoLog*, due to the higher squash frequency, the total network traffic is on average 17% higher than in *OrderOnly*.

# CHAPTER 9

# CapoOne Evaluation

## 9.1 Log Size

Figure 9.1 shows the size of CapoOne's logs, namely the Memory Ordering Log (including the RI Log and the CS Logs) and the Sphere Input Log, measured in bits per committed kilo-instruction. Although not seen in the figure, the contribution of the CS 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.

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 Memory Ordering 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 a larger Sphere Input Log because they execute more system calls.

Overall, we find that the size of the Memory Ordering Log is comparable to the size of the PI Log and CS Logs reported in Section 8.1. 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.

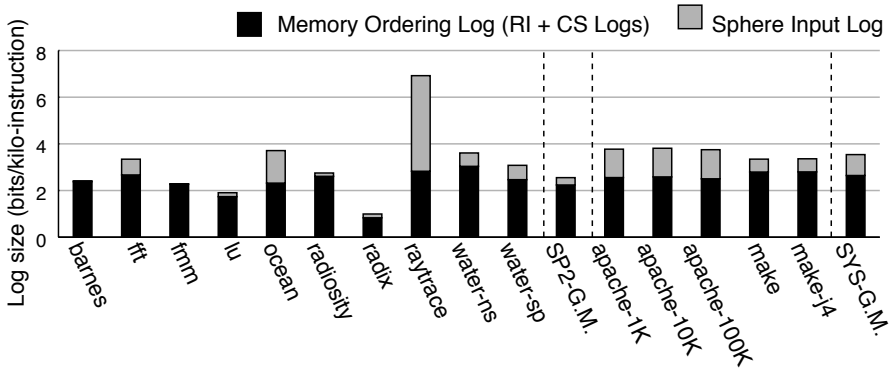



**Figure 9.1:** CapoOne's log size in bits per kilo-instruction.

## 9.2 Hardware Characterization

Table 9.1 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. Other exceptions occur rarely and, as indicated in Section 6.4.3.1, interrupts squash chunks.

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 9.1 should match those with long Memory Ordering Logs in Figure 9.1. However, the correlation is not that clear due to the effect of log compression.

## 9.3 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 be-

| 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 9.1:** CapoOne's hardware characterization.

cause 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 [5, 28, 41, 64, 65].

We first consider a single sphere in the machine. Figure 9.2 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 9.2 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

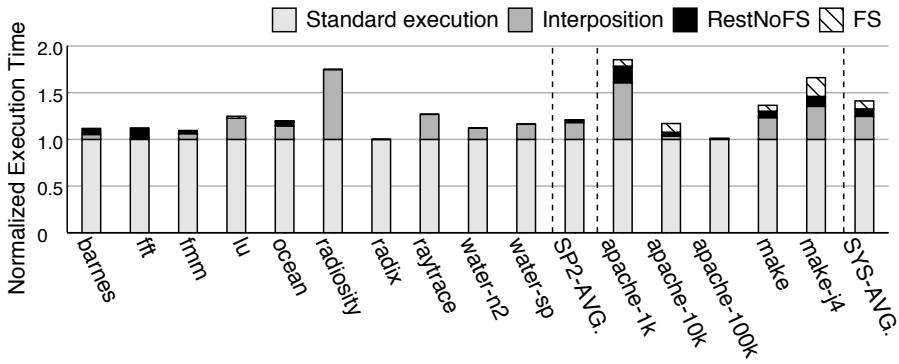



**Figure 9.2:** Execution time overhead of CapoOne during recording for a single replay sphere in the machine.

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 9.3 shows the resulting normalized execution times. For each application, we show three pairs of bars, where each pair corresponds to one of

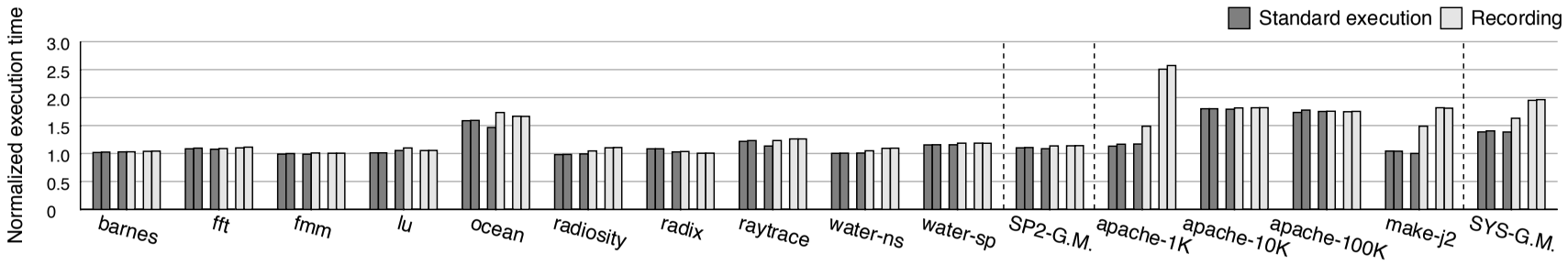



**Figure 9.3:** Execution time overhead of CapoOne during recording when two replay spheres share the machine.

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 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 9.2, 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.

## 9.4 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 9.4 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 R-thread 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 re-executed. 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

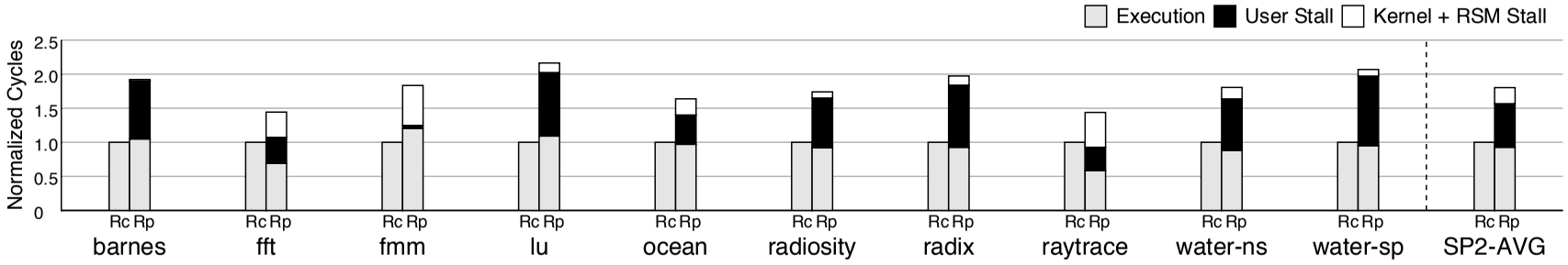



**Figure 9.4:** Normalized number of cycles taken by the SPLASH-2 applications during recording (*Rc*) and replay (*Rp*).

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

**Table 9.2:** Replay performance of the web server applications.

*Real-Commodity-HW* environment and, therefore, measure their performance in elapsed time. Table 9.2 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* [21], 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.

# CHAPTER 10

# Future Work

There are many possible avenues for extensions to the work presented in this thesis. First, on the hardware side, DeLorean and other hardware-based replay systems require important modifications on the processor and/or the memory hierarchies. This scares hardware manufactures because they perceive such changes as too intrusive. Another key issue is that current hardware-based deterministic replay systems offer little support for consistency models other than sequential consistency. However, manufacturers such as AMD, IBM, Intel and Sun use more relaxed memory consistency models because of performance reasons. Thus, an important future line of research will be to explore lightweight hardware mechanisms for more relaxed consistency models, such as processor consistency —found in most desktop and server processors sold today. We believe that such a scheme will have an important impact on industry.

Second, there are important questions that remain to be answered on the software side. For example, the operating system scheduler needs to be aware of the fact that a thread being replayed and currently scheduled in the processor might not be making any progress because it is waiting for another thread to commit. Exposing such knowledge to the OS is still an open problem. We also believe that software can play a very important role on reducing the complexity of deterministic replay hardware. Much of the complexity found in current hardware-only proposals is due to the fact that hardware is oblivious to the properties of the software being recorded and replayed. Hence, I think it is possible to leverage the richness of managed execution to build a hardware-assisted replay system for managed code. In this model, the runtime system will give the replay hardware hints

about the software running on it, hopefully helping reduce complexity.

Finally, we need to expand the horizon of usability for deterministic replay. A very promising line of research is looking for new applications for deterministic replay in the reliability and security domains. For example, CapoOne can help increase the reliability of multiprocessor servers. Instead of having the software just write to disk the logs of a replay sphere, the logs could be sent to a different machine which, in turn, would use them to replay the first one's execution, but this time running extra security checks to ensure that the system has not been compromised.

# CHAPTER 11

# Conclusions

With the shift to multicore hardware, parallel programming must become the norm. Unfortunately, most current programmers find parallelism very challenging. Thus, programmers would benefit greatly from tools and techniques that could help them debugging parallel applications. One such technique is deterministic replay of execution. Recording and deterministically replaying execution gives computer users the ability to travel backward in time, recreating past states and events in the computer.

Current software-based deterministic replay systems are flexible but they perform slowly on (or do not work with) multiprocessors. Hardware-based schemes can record multiprocessor execution much more efficiently than software schemes, but they still fall short in some areas, namely performance and size of their Memory Ordering Log.

This thesis has proposed DeLorean, a novel scheme for hardware-based deterministic replay where processors execute groups of instructions atomically. DeLorean has two fundamental advantages over current hardware-based schemes. First, it records at the speed of the most aggressive memory consistency models used today — and also replays at high speed. This makes it useful for production-run debugging. Second, it summarizes the execution interleaving into a truly small log.

DeLorean's execution modes offer a trade-off between performance and log size. In *OrderOnly*, DeLorean records at the speed of RC execution and replays at 82% of RC speed. In contrast, most other schemes record only at the speed of SC execution and provide no details on replay speed. RTR presents an algorithm for recording TSO executions but does not evaluate its impact on execution speed or log size. Moreover, *OrderOnly*

only needs 1.3 bits of compressed memory-ordering log per kilo-instruction and, with stratification, only 0.6 bits. We estimate the latter to be 7.5% of the log size needed by Basic RTR.

In *PicoLog* mode, DeLorean reduces the memory-ordering log to 0.05 bits kilo-instruction, which we estimate is 0.6% of the log size in Basic RTR. In this mode, we estimate that the total memory-ordering log of an 8-processor 5-GHz machine is only about 20GB per day. Recording speed decreases to 86% of RC execution speed — still higher than typical SC speed.

Another area where hardware-based replay systems —including DeLorean— fall short is practicality. 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. Because of this, hardware-based schemes cannon replace yet software-based systems.

To solve this problem, this thesis 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.

Our evaluation of 4-processor executions showed that CapoOne largely records with the efficiency of hardware-only schemes while still maintaining the flexibility of software-only schemes. 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.

Although CapoOne overheads are modest, we believe that there is lots of room for improvement. This thesis identified the two largest sources of overhead. First, a user-level, ptrace-based RSM implementation and second, excessive stalling when the R-thread that must commit next is not scheduled on any processor. Both of them can be fixed rather easily. In fact, the second implementation of the Capo interface is already being developed in our group; we call it *CapoTwo*. Even though it still is in an early stage, CapoTwo has shown incredible potential.

Overall, the first half of this thesis showed that hardware can record and replay multiprocessor execution efficiently and for long periods of time. The second half, in turn, described how to make hardware-based replay systems practical for the user. Therefore, we believe that deterministic replay of multiprocessor systems is a powerful tool for debugging, security and fault tolerance.

# REFERENCES


[1] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *Western Reseach Laboratory-Compaq. Research Report 95/7*, September 1995.

[2] H. Agrawal, R. A. DeMillo, and E. H. Spafford. An Execution-Backtracking Approach to Debugging. *IEEE Software*, 8(3):21–26, May 1991.

[3] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 423–434, San Diego, California, United States, November 2003.

[4] Apache Software Foundation. Apache HTTP Server. http://www.apache.org.

[5] D. F. Bacon and S. C. Goldstein. Hardware-Assisted Replay of Multiprocessor Programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 194–206, Santa Cruz, California, United States, August 1991.

[6] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for Instruction-level Tracing and Analysis of Program Executions. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, pages 154–163, Ottawa, Ontario, Canada, June 2006.

[7] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 11(7):422–426, July 1970.

[8] B. Boothe. Efficient Algorithms for Bidirectional Debugging. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 299–310, Vancouver, British Columbia, Canada, June 2000.

[9] T. C. Bressoud and F. B. Schneider. Hypervisor-Based Fault-Tolerance. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 1–11, Copper Mountain, Colorado, United States, December 1995.

[10] Canonical. Ltd. Ubuntu Linux. http://www.ubuntu.com.

[11] L. Ceze, K. Strauss, J. Tuck, J. Renau, and J. Torrellas. CAVA: Using Checkpoint-Assisted VAlue Prediction to Hide L2 Misses. *ACM Transactions on Architecture and Code Optimization*, pages 182–208, June 2006.

[12] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Proceedings of the 33rd International Symposium on Computer Architecture*, pages 227–238, Washington DC, United States, June 2006.

[13] L. Ceze, J. M. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 278–289, San Diego, California, United States, June 2007.

[14] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A Scalable, Non-Blocking Approach to Transactional Memory. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, pages 97–108, February 2007.

[15] S.-K. Chen, W. K. Fuchs, and J.-Y. Chung. Reversible Debugging Using Program Instrumentation. *IEEE Transactions on Software Engineering*, 27(8):715–727, August 2001.

[16] J. D. Choi and H. Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 48–59, Welches, Oregon, United States, August 1998.

[17] M. Christiaens, J.-D. Choi, M. Ronsse, and K. D. Bosschere. Record/Replay in the Presence of Benign Data Races. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1200–1206, Las Vegas, Nevada, United States, June 2002.

[18] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent Control with "Readers" and "Writers". *Communications of the ACM*, 14(10):667–668, October 1971.

[19] A. Cristal, D. Ortega, J. Llosa, and M. Valero. Out-of-Order Commit Processors. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, pages 48–60, Madrid, Spain, February 2004.

[20] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High Availability Via Asynchronous Virtual Machine Replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174, San Francisco, California, United States, February 2008.

[21] 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 5th Symposium on Operating Systems Design and Implementation*, pages 211–224, Boston, Massachusetts, United States, December 2002.

[22] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution Replay of Multiprocessor Virtual Machines. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 121–130, Seattle, Washington, United States, March 2008.

[23] S. I. Feldman and C. B. Brown. IGOR: A System for Program Debugging Via Reversible Execution. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 112–123, Madison, Wisconsin, United States, November 1988.

[24] A. Forin. Debugging of Heterogeneous Parallel Systems. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 130–140, Madison, Wisconsin, United States, November 1988.

[25] K. Gharachorloo, A. Gupta, and J. Hennessy. Hiding Memory Latency Using Dynamic Scheduling in Shared-Memory Multiprocessors. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 22–33, Gold Coast, Queensland, Australia, May 1992.

[26] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *Proceedings of the 31st International Symposium on Computer Architecture*, pages 102–113, Munchen, Germany, June 2004.

[27] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 19–31, San Francisco, California, United States, January 1994.

[28] 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*, pages 265–276, Beijing, China, June 2008.

[29] Intel Corporation. Intel 64 and IA-32 Architectures Software Developers Manual, Volume 3B, February 2009. http://www.intel.com/products/processor/manuals/.

[30] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting Past and Present Intrusions Through Vulnerability-Specific Predicates. In *Proceedings of the 20th Symposium on Operating Systems Principles*, pages 91–104, Brighton, United Kingdom, October 2005.

[31] S. T. King and P. M. Chen. Backtracking Intrusions. In *Proceedings of the 18th Symposium on Operating Systems Principles*, pages 223–236, Bolton Landing, New York, United States, October 2003.

[32] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging Operating Systems with Time-Traveling Virtual Machines. In *USENIX Technical Conference*, pages 1–15, Anaheim, California, United States, April 2005.

[33] M. Kirman, N. Kirman, and J. F. Martinez. Cherry-MP: Correctly Integrating Checkpointed Early Resource Recycling in Chip Multiprocessors. In *Proceedings of the 38th International Symposium on Microarchitecture*, pages 245 – 256, Barcelona, Spain, November 2005.

[34] N. Kirman, M. Kirman, M. Chaudhuri, and J. F. Martinez. Checkpointed Early Load Retirement. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture*, pages 16 – 27, San Francisco, California, United States, February 2005.

[35] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[36] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, 36(4):471–482, April 1987.

[37] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. SIMICS: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, 2002.

[38] J. Martínez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pages 3 – 14, Istambul, Turkey, November 2002.

[39] M. Musuvathi, S. Qadeer, T. Ball, and G. Basler. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation*, pages 122–133, San Francisco, California, United States, December 2008.

[40] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture*, pages 129 – 140, Anaheim, California, United States, February 2003.

[41] S. Narayanasamy, C. Pereira, and B. Calder. Recording Shared Memory Dependencies Using Strata. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 229–240, San Jose, California, United States, October 2006.

[42] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 284–295, Los Alamitos, California, United States, June 2005.

[43] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 22–31, San Diego, California, United States, June 2007.

[44] R. H. B. Netzer. Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs. In *Proceedings of the 1993 ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 1–11, San Diego, California, United States, May 1993.

[45] D. Z. Pan and M. A. Linton. Supporting Reverse Execution for Parallel Programs. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 124–129, Madison, Wisconsin, United States, January 1988.

[46] M. Prvulovic. Cost-effective (and nearly overhead-free) Order-Recording and Data Race detection. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 232–243, Austin, Texas,United States, February 2006.

[47] M. Prvulovic and J. Torrellas. ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 110–121, San Diego, California, United States, June 2003.

[48] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 97–108, Anchorage, Alaska, United States, May 2002.

[49] P. Ranganathan, V. S. Pai, and S. V. Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap Between Memory Consistency Models. In *Proceedings of the 9th Symposium on Parallel Algorithms and Architectures*, pages 199–210, June 1997.

[50] R. Rashid, R. Baron, A. Forin, D. Golub, M. Jones, D. Orr, and R. Sanzi. Mach: a Foundation for Open Systems. In *Proceedings of the Second Workshop on Workstation Operating Systems*, pages 109–113, Pacific Grove, California, United States, September 1989.

[51] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, K. Strauss, S. Sarangi, P. Sack, and P. Montesinos. SESC Simulator, January 2005. http://sesc.sourceforge.net.

[52] M. Ronsse and K. De Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM Transactions on Computer Systems*, 17(2):133–152, 1999.

[53] M. Russinovich and B. Cogswell. Replay for Concurrent Non-Deterministic Shared-Memory Applications. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, pages 258–266, Philadelphia, Pennsylvania, United States, 1996.

[54] 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 *Proceedings of the 17th Symposium on Operating Systems Principles*, pages 110–123, Charleston, South Carolina, United States, December 1999.

[55] J. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, California, United States, 2005.

[56] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: Improving the Availability of Shared-Memory Multiprocessors with Global Checkpoint/Recovery. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 123–134, Anchorage, Alaska, United States, May 2002.

[57] 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*, pages 29–44, 2004.

[58] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual Flow Pipelines. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117 – 119, Boston, Massachusetts, United States, October 2004.

[59] Sun Microsystems. UltraSPARC Architecture, June 2008.

[60] J. Tuck, W. Ahn, L. Ceze, and J. Torrellas. SoftSig: Software-exposed Hardware Signatures for Code Analysis and Optimization. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145–156, Seattle, Washington, United States, February 2008.

[61] E. Vallejo, M. Galluzzi, A. Cristal, F. Vallejo, R. Beivide, P. Stenstrom, J. E. Smith, and M. Valero. Implementing Kilo-Instruction Multiprocessors. In *Proceedings of the International Conference on Pervasive Systems*, pages 325–336, Santorini, Greece, July 2005.

[62] VMWare, Inc. Protecting Mission-Critical Workloads with VMware Fault Tolerance, February 2009. http://www.vmware.com/files/pdf/resources/ft_virtualization_wp.pdf.

[63] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for Store-Wait-Free Multiprocessors. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, California, United States, June 2007.

[64] M. Xu, R. Bodik, and M. D. Hill. A "Flight Data Recorder" for Enabling Full-System Multiprocessor Deterministic Replay. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 122–135, San Diego, California, United States, June 2003.

[65] M. Xu, R. Bodik, and M. D. Hill. A Regulated Transitive Reduction (RTR) for Longer Memory Race Recording. In *"International Conference on Architectural Support for Programming Language and Operating Systems"*, pages 49–60, San Jose, California, United States, October 2006.

[66] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. ReTrace: Collecting Execution Trace with Virtual Machine Deterministic Replay. In *Third Annual Workshop on Modeling, Benchmarking and Simulation*, San Diego, California, United States, June 2007.

[67] M. V. Zelkowitz. Reversible Execution. *Communications of the ACM*, 16(9):566, September 1973.

[68] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337– 343, May 1977.

# AUTHOR'S BIOGRAPHY

Pablo Montesinos Ortego was born in León, Spain. He was attracted to computers since he was seven years old, when his father bought the family's first computer and taught him how to program in Logo and Pascal. Pablo received his BS degree in Computer Engineering from the Universidad de León, where he did operating system work. He worked as a R&D programmer for Ydilo Advanced Voice Solutions in Madrid, Spain. He then moved back to León, where he worked as a Lecturer in the Universidad de León. He decided to continue his studies in Urbana-Champaign where he received his M.S. and Ph.D. degrees in Computer Science from the University of Illinois. His graduate research has been focused on computer architecture. He developed solutions for enhancing the programmability of multiprocessor systems and designed multiprocessor systems that can travel back in time. Pablo has co-authored over 15 research papers and has received awards or his research in computer architecture. He has also held a La Caixa Foundation Fellowship during part of his graduate studies. After receiving his PhD he became a Staff Engineer at the Multicore Research Group at Samsung Information Systems America.