RECORD AND DETERMINISTIC REPLAY OF PARALLEL PROGRAMS ON MULTIPROCESSORS

BY

NIMA HONARMAND

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, 2014

Urbana, Illinois

Doctoral Committee:

Professor Josep Torrellas, Chair and Director of Research
Professor Sarita V. Adve
Professor Vikram S. Adve
Associate Professor Samuel T. King
Dr. Gilles A. Pokam, Intel Labs
Associate Professor Satish Narayanasamy, University of Michigan, Ann Arbor
Abstract

Record and deterministic Replay (RnR) is a primitive with many proposed applications in computer systems, including debugging, security and fault tolerance. RnR is typically a two phase process: in the first phase (record) enough information about an execution is logged which is then used in the second phase (replay) to re-create the execution.

Application-level RnR seeks to record and replay single programs (or sets of programs) in isolation from the rest of the system. In this environment, there are typically two sources of non-determinism that an RnR solution should capture: program inputs (such as the results of the system calls the program makes to the OS or the signals the program receives) and the memory-access interleaving of concurrent threads that result in inter-thread data dependences.

In order to enjoy wide acceptance, RnR solutions should be practical to build and, at the same time, enable a diverse range of use-cases (such as debugging and security analysis). Low recording overhead is a key requirement for many use cases of RnR. While software can often be used to record program inputs with low overhead, it can incur significant overhead to record memory-access interleaving. To reduce this overhead, hardware-assisted RnR techniques have been proposed. The main challenge here is devising hardware mechanisms that are simple enough to be embraced by processor vendors and, at the same time, powerful enough to work for the complex architectures of today. The first part of this thesis is a step in this direction — i.e., building practical and low overhead hardware-assisted RnR systems.

We focus on the hardware-assisted RnR of parallel programs on multiprocessor machines. Firstly, we introduce QuickRec [65], the first physical realization of a hardware-assisted RnR system including new hardware and software. The focus of this project is understanding and evaluating the implementation issues of RnR on a real platform. We demonstrate that RnR can be implemented efficiently on a real multicore Intel Architecture (IA) system. We show that the rate of memory log generation is insignificant, and that the recording hardware has negligible performance overhead, as expected. The evaluations however point
to the software stack as the major source of overhead (incurring an average recording overhead of nearly 13%), an issue that was largely ignored by previous work on hardware-assisted RnR.

We then address the problem of replay speed by introducing **Cyrus** [31], an RnR scheme that can record programs and replay them in parallel without making any changes to the cache coherence protocol and messages. The proposal uses a novel hybrid hardware/software mechanism for recording memory-access interleaving. The hardware component records a raw and incomplete log that is then processed and transformed into a complete log by an on-the-fly software Backend Pass. As the raw log is being generated, this pass transforms it for high replay parallelism. This can also flexibly trade-off replay parallelism for log size. We evaluate Cyrus through full-system simulation including simulated hardware and using the same real software stack that was used in QuickRec.

QuickRec and Cyrus are limited in terms of the memory consistency models they support: Total Store Order (TSO) and Sequential Consistency (SC), respectively. To enable RnR for other architectures whose memory model is more relaxed, we then propose **RelaxReplay** [32]. It is a general hardware-assisted MRR scheme that works for any relaxed-consistency model of current processors and does not require any changes to the underlying coherence protocol and messages. RelaxReplay’s core innovation is a new way of capturing memory access reordering. Each memory instruction goes through a post-completion in-order counting step that detects any reordering, and efficiently records it. The evaluations show that RelaxReplay induces negligible recording overhead and that the average size of the log produced is only 1–4x as large as in existing solutions — still very small compared to the memory bandwidth of modern machines.

After considering the challenges of building practical RnR systems, the next question to be answered is that of their usability. The last part of this thesis investigates the issue of using the RnR technology in program debugging, the most commonly cited use-case of replay. RnR enables deterministic reproduction of hard-to-repeat software bugs. However, simply providing support for repeatedly stumbling on the same bug does not help diagnose it. For bug diagnosis, developers typically augment the program source with debug code — E.g., by creating and operating on new variables, or printing state. Unfortunately, this renders the RnR log inconsistent and makes **Replay Debugging** (i.e., debugging while using an RnR log for replay) dicey at best.

To attack this problem, we propose **rdb** [33], the first scheme for replay debugging that guarantees exact replay in the presence of debug code. **rdb** relies on two mechanisms. The first one is compiler support to
split the instrumented application into two executables: one that is identical to the original program binary, and another that encapsulates all the added debug code. The second mechanism is a runtime infrastructure that replays the application and, without affecting it in any way, invokes the appropriate debug code at the appropriate locations. We describe an implementation of rdb based on LLVM [3] and Pin [53], and show an example of how rdb’s replay debugging helps diagnose a real bug.
To the memory of my first teacher, my loving mother.

To my father, who taught me to be a human before being a scholar.

To my precious wife, Razieh, whose tender love I cherish above all else.
Acknowledgements

I am grateful to many people for supporting and contributing to the content of this thesis:

- My advisor, Josep Torrellas, unwaveringly encouraged, supported, and guided me throughout my Ph.D. research. He spent many hours with me, patiently brain storming and providing much needed discussions to help me polish my thoughts. He also contributed many ideas to the projects.

- Sam King worked closely with Josep and me and was a key driving force behind the QuickRec and Cyrus projects. He provided the initial code of Capo3 and continued to advise me afterwards on all the issues related to QuickRec’s OS aspects and hardware/software interface.

- I was lucky to collaborate with Gilles Pokam, Klaus Danne, Cristiano Pereira, Rolf Kassa, Tim Kranich, Shiliang Hu and Justin Gottschlich from the Intel Corporation on the QuickRec project. They designed and implemented the QuickRec hardware for which I built the software and did the hardware/software integration. This collaborative effort laid the foundation for all other projects in this dissertation.

- My friend, Nathan Dautenhahn, had a major role in getting the Capo3 software up and running on the QuickRec hardware, as well as in implementing its replay tool and running the experiments.

Finally, I would like to thank my Ph.D. committee members who provided me with much valuable feedback in improving this dissertation.
# Table of Contents

## List of Tables
- Chapter 1 Introduction
  - 1.1 Record and Deterministic Replay
    - 1.1.1 Scope of RnR and Definition of Determinism
    - 1.1.2 Capturing Sources of Non-determinism
    - 1.1.3 RnR in This Thesis
    - 1.1.4 Capturing Non-determinism: Hardware-Assisted vs. Software-Only RnR
  - 1.2 A Brief Overview of Existing Related Work
    - 1.2.1 Software-Only RnR Solutions
    - 1.2.2 Hardware-Assisted RnR Solutions
  - 1.3 Thesis Contributions and Outline
    - 1.3.1 First Physical Prototype of Hardware-Assisted RnR
    - 1.3.2 Enabling Fast Replay Through Replay Parallelism
    - 1.3.3 Efficient RnR for Relaxed Memory Models
    - 1.3.4 Replay Debugging

## List of Figures
- Chapter 2 QuickRec: Prototyping an Intel Architecture Extension for Record and Replay of Multithreaded Programs
  - 2.1 Introduction
  - 2.2 QuickIA Recording System
    - 2.2.1 QuickIA Processor Emulation Platform
    - 2.2.2 Recording Interleaving Non-Determinism
    - 2.2.3 Programming Interface
    - 2.2.4 Other Issues
  - 2.3 Capo3 System Software
    - 2.3.1 Comparing Capo3 with Capo
    - 2.3.2 Capo3 Architecture
    - 2.3.3 Virtualizing the QuickRec Hardware
    - 2.3.4 Handling Context Switches
    - 2.3.5 Recording in Kernel Mode
    - 2.3.6 Handling Input/Chunking Interactions
  - 2.4 Prototype Characterization
    - 2.4.1 Experimental Setup
    - 2.4.2 Log Analysis
List of Tables

2.1 QuickIA platform parameters. ................................................................. 18
2.2 Main reasons for terminating chunks. WAB (Write-After-Both) is when a write in one chunk
  hits in both the read and the write set of another chunk. .......................... 23
2.3 Characteristics of the benchmarks. The last column shows the total number of instructions
  executed in the 4-threaded run in billions. Water refers to Water-nsquare. ........ 30
3.1 Parameters of the simulated hardware. .................................................. 70
4.1 Architectural parameters. ...................................................................... 101
# List of Figures

2.1 Photograph of the QuickRec prototype with FPGAs in CPU sockets (a); architecture of the QuickIA processor-emulation platform (b); and architecture overview of the extended Pentium core in QuickRec, where circled numbers identify the main CPU touch points required to enable recording (c).  

2.2 Instruction atomicity violation (IAV) example.  

2.3 IAV counter mechanism.  

2.4 Packet formats in QuickRec.  

2.5 Overall architecture of Capo3. Dashed boxes indicate QuickRec-specific components.  

2.6 Examples of dependences between input events (solid lines) and between chunks (dashed lines).  

2.7 Uncompressed log sizes.  

2.8 Memory bandwidth requirements.  

2.9 Average chunk size.  

2.10 Cumulative distribution of chunk size.  

2.11 Chunk termination reasons.  

2.12 RSW and IAV characterization.  

2.13 Execution time with each recording configuration for four-threaded executions. The bars are normalized to the execution time of native.  

2.14 Total time that the processors spend working on the applications divided into user and system time.  

2.15 Breakdown of the total processor cycles for different thread counts.  

2.16 Breakdown of the normalized overhead of input recording and management. CTU stands for Copy To User.  

2.17 Breakdown of the normalized chunk-management overhead.  

3.1 Difficulties in capturing inter-thread dependences in application-level RnR.  

3.2 Keeping a snoopy cache coherence protocol unmodified requires recording dependences from the dependence source only.  

3.3 Overview of the Cyrus system.  

3.4 Characterizing the types of interprocessor interactions.  

3.5 High-level view of the Cyrus hardware.  

3.6 Recording dependences in Cyrus.  

3.7 Example of execution and resulting Cyrus logs. The table in (b) depicts the initial chunk data dumped by the processors, while the other tables show the results of the different backends, encoding the corresponding DAGs. In the tables, dashes indicate entries corresponding to dependencies to the processor itself. These are never used.
Chapter 1

Introduction

1.1 Record and Deterministic Replay

Many problems in designing and programming reliable computer systems can significantly benefit from the ability to examine a past execution. Being able to deterministically re-create an execution is thus a crucial primitive for these domains. Record and deterministic Replay (RnR) is a primitive that strives to achieve this ability. RnR has broad uses in, at least, program debugging [4, 13, 17, 41, 71, 75, 90, 60], where, for example, a bug can be easily reproduced, intrusion analysis [40, 24, 39, 68], where an intrusion can be traced back to an attacker’s actions, and fault-tolerant, highly-available systems [71, 22], where a backup machine can resume where the primary failed.

RnR is typically a two phase process: in the first phase (record) enough information about an execution is logged which is then use in the second phase (replay) to re-create the execution.

1.1.1 Scope of RnR and Definition of Determinism

RnR may be achieved at different levels of abstraction, from distributed programs (e.g., [7]) to physical machines (e.g. [56]) to virtual machines (e.g., [24, 80]), to OS processes (e.g., [57, 42]) to program components (e.g., [30]). Different solutions have different costs and overheads and, in turn, enable different use-cases. The choice of the level of abstraction, therefore, highly depends on the use-case of RnR that the system designer has in mind and the cost he/she is willing to pay.

Similarly, the definition of “determinism” in the context of replay also depends on the chosen level of abstraction as well as the intended use-case. For example, in some cases, determinism only concerns the observable outputs of a program (e.g. [6, 62]) while, in other cases, it may concern the exact sequence of instructions in the dynamic control flow (e.g., [57, 59]).
1.1.2 Capturing Sources of Non-determinism

In all these cases, however, all sources of non-determinism that can affect the relevant properties of the execution should be captured at record time or inferred at replay time. The resulting logs should be used at replay time to guide the re-execution to ensure deterministic replay. From this point of view, replay solutions can be divided into two broad categories:

- **Search-based** RnR techniques record only a subset of all the non-deterministic events that can affect the execution at record time and then use a search phase to infer the missing pieces (e.g., [6, 62, 36]). Example of such missing information include not recorded program inputs or inter-thread data dependencies. They typically employ such techniques as symbolic execution or constraint solvers in their search phase. These solutions normally rely on a relaxed definition of “determinism” (such as output determinism), since they can never guarantee stronger forms of determinism due to incompleteness of recorded logs, and are sometimes referred to as relaxed RnR methods. In the extreme end of the spectrum, they only record the final state of a program (as a core dump) and try to synthesize an execution that can lead to that final state [89].

- **Strong** RnR techniques that record enough information to be able to do replay without searching. They typically have higher recording overhead than search-based techniques (due to recording larger logs) but provide simpler and faster replay mechanisms.

1.1.3 RnR in This Thesis

*Application-level* RnR seeks to record and replay single programs (or sets of programs) in isolation from the rest of the system. This is what users need most of the time and it is general enough to be useful for many of the RnR use-cases mentioned earlier. Also, compared to recording whole machines, it is more portable. Recreating the whole machine state during replay is often very hard and, to work correctly, needs to deal with many non-portable operating system and hardware issues of the platform. Application-level RnR, in contrast, tends to be more portable and add less overhead. In addition, many of the techniques developed for this RnR style, can be easily applied to RnR of virtual machines (using hypervisors) that is another widely useful flavor of RnR.

In this thesis, we target RnR of parallel programs on shared-memory multi-processor machines. We
consider strong RnR techniques and seek functional determinism: each thread in the replayed execution should follow the same sequence of instruction as in the original run and each instruction’s inputs should be the same as those of its original counterpart.

1.1.4 Capturing Non-determinism: Hardware-Assisted vs. Software-Only RnR

In this environment, there are typically two sources of non-determinism that an RnR solution should consider: program inputs (such as the results of system calls the program makes to the OS or signals the program receives) and the memory-access interleaving of concurrent threads that result in inter-thread data dependence. The process of recording the latter is sometimes called Memory Race Recording (MRR) in the literature.

Most proposed solutions for application-level RnR use some software support to record and replay program inputs, typically relying on the OS, compiler, libraries and/or run-time systems for this purpose. Typically, this is enough for RnR of single-threaded applications. For multi-threaded applications, however, memory-access interleaving also becomes relevant. There are several proposals for RnR of multithreaded programs. On the one hand, there are those that do not require any special hardware, typically relying on the OS or compiler for MRR. Being software-only solutions, these systems are relatively inexpensive to implement but might incur significant slowdown during recording. Other schemes record with the aid of some special hardware module. These systems add negligible overhead during recording, but can be expensive to implement. We call such schemes hardware-assisted.

Low recording overhead is a key requirement for many use cases of RnR, especially those that involve recording of production runs or cannot tolerate the (often systematic) execution perturbations common in software-only schemes (for example, concurrency debugging). Thus, the lower overhead of hardware-assisted schemes makes them a more likely candidate as a general RnR solution. The obvious challenge here is devising hardware mechanisms that are simple-enough to be embraced by processor vendors and, at the same time, powerful enough to work for the complex architectures of today. This thesis is a step in this direction.

---

1 Inter-Process Communication (IPC) via shared memory is a special form of program input that involves shared memory accesses. Our RnR mechanism can record such communications by recording the communicating processes together (See the notion of Replay Sphere in Section 2.3.1)
1.2 A Brief Overview of Existing Related Work

RnR has been the subject of a vast body of research. In this section, we review some of the existing work and what flavor of RnR they support and how. Since this thesis mainly focuses on hardware-assisted RnR which is typically used for memory-race recording, we put more emphasis on how MRR is achieved in existing proposals.

In addition to this brief review, we discuss and compare to our work the more closely related pieces of previous work throughout the thesis after introducing each of our contributions. We postpone the discussion of previous work on replay debugging until Section 5.8 as it requires concept that will be introduced in that section.

1.2.1 Software-Only RnR Solutions

Software-only RnR relies on modified runtime libraries, compilers, operating systems and virtual-machine monitors to capture all or some sources of non-determinism. In addition, relaxed RnR approaches also use search-based and formal techniques to recover the missing pieces of information that are not captured at record time.

Strong Software-Only Solutions

Compiler- and Library-Based Approaches. We can divide these approaches into two categories, based on how they capture shared-memory interactions: value-based and order-based. Value-based techniques record the value returned by shared memory reads. Together with program inputs, this information is enough to enable independent replay of each thread of execution but does not provide much intuition about the interaction between threads as inter-thread data dependences are not explicitly re-created. Order-based techniques record the order in which concurrent threads of execution access shared memory location and enforce the same ordering during replay.

Recap [61] is a solution to enable reverse execution of parallel programs and uses RnR for this purpose. It takes a value-based approach to MRR, where the compiler inserts some code before every read operation that may access shared data. It also uses a special run-time library that handles system calls, signals, and checkpointing. Using this information it can replay each thread of execution independently. Checkpoints are used to provide a consistent snapshot of the execution from where reverse execution can begin.
iDNA \[11\] is a value-based proposal that uses RnR as a means to reduce the log size needed for recording application traces. Instead of recording the trace, it records the memory values read by the executing instructions, program code, the register state after a kernel-to-user transition and the register state after certain special instructions whose effects are time or machine specific (e.g. the RTDSC and CPUID instructions on IA-32). It then uses this information to do a per-thread replay of the application, thereby regenerating the trace. It uses heavy-weight binary translation and instruction emulation platform called Nirvana to record and replay its logs.

LeBlanc and Mellor-Crummey \[46\] propose an order-based approach. They use Reader-Writer locking for shared memory accesses to log an execution. They capture the locking order on shared objects and thus cannot record racy executions that access shared objects without locking.

LEAP \[35\] is an order-based solution for deterministic replay of multi-threaded Java programs. It uses static analysis to identify all potentially shared memory locations in a program, and then, at record time, instruments the code to record the order in which each variable is accessed by concurrent threads. The authors claim that by recording each variable’s access order independently from other variables, LEAP incurs less record-time synchronization and, hence, less recording overhead compared to solutions that record global interleaving information.

CARE \[38\] reduces the overhead of order-based techniques such as LEAP by only logging a fraction of all actual precedence orders between reads and their associated writes (i.e., read-write dependences). It assigns to each thread a value prediction cache that keeps buffered variable values. Each time a read action is executed, the buffered value is compared with the one actually read. Only when the values are different, the corresponding read-write dependence is logged. The assumption is that if two consecutive accesses (to the same memory location) of the same thread share the same value, they can be assumed to have executed without interleaving writes from other threads and thus only the first one needs to be recorded.

Chimera \[47\] uses instrumentation to capture program inputs, thread schedule on each core and synchronization-induced happens-before relationships. This information is enough to replay data-race-free executions. To add support for racy programs, it uses static analysis to identify all accesses that can potentially participate in a race. It then places each such access inside a region protected by a “weak lock”. In addition to the original program synchronizations, Chimera records happens-before relationships due to weak-locks to guarantee deterministic replay in presence of data races.
PinPlay \cite{63} is an order-based approach based on the Pin dynamic instrumentation system. It uses Pin for both recording and replay. It records the order of shared memory accesses using a simulation-based implementation of FDR \cite{85} (discussed in Section 1.2.2). The logger simulates a cache-coherency protocol noting the last reader/writer for address ranges used in the program at a tunable granularity. It uses this information to compute read/write, write/write, and write/read dependences for shared-memory address ranges and records a subset of them (others are implied).

Jockey \cite{72} is a library for record and replay of single-threaded programs. It is a library that is loaded in the same address space as the program under RnR, before loading the program itself. Once the program is loaded, Jockey takes over and instruments the program code to intercept system call invocations. In record mode, intercepted system calls are logged in a file; in replay mode, they are replayed back from the file. It also mechanisms to capture signals and memory-mapped I/O to ensure deterministic replay.

R2 \cite{30} is a annotation-based RnR solution for the Windows platform. It records the results of selected functions in a log and during replay returns the results from the log rather than executing the functions. A Programmer have to choose functions that should be recorded. In doing so, they can trade recording overhead and replay fidelity. The programmer should annotate the chosen functions with some keywords so that R2 can handle calls with side effects and multithreading.

**Operating-System- and Virtual-Machine-Based Approaches.** Some RnR techniques use OS/VM support instead of, or in addition to, compiler and library support. For example, Russinovich and Cogswell \cite{71} propose to modify the OS scheduler to record thread interleaving in a uniprocessor. This will enable RnR of multi-threaded programs on uniprocessor systems. Similarly, DejaVu \cite{19} records the scheduling decisions of a Java Virtual Machine to enable deterministic replay of multi-threaded Java applications on uniprocessors.

Paranoid Android (PA) \cite{68} is an RnR-based decoupled security solution for smartphones. It records applications on a phone and ships the log to a cloud-based machine that replays the execution on a virtual replica of the phone. PA records process inputs at the level of system calls. The authors claim that PA can handle multi-threaded applications on uniprocessor systems by enforcing repeatable scheduling decisions using the `ptrace` \cite{2} mechanism; unfortunately, there is not enough information in the paper to corroborate that.

Bressoud and Schneider \cite{14} and ReVirt \cite{24} use a modified hypervisor to replay single processor
virtual machines. Targeting uniprocessor VMs, they only need to capture inputs to the virtual machine to guarantee RnR. Virtual machine inputs consist of data that is provided to its virtual devices and interrupts that are delivered to the virtual CPU. VMware also provide a similar RnR capability based on the same principle [80].

SMP-ReVirt [25] extend ReVirt to multiprocessor virtual machines by using virtual-memory page-level protections to detect shared-memory communication between virtual CPUs. They implement a concurrent-read, exclusive-write (CREW) protocol between virtual CPUs in a multiprocessor virtual machine. Each read or write operation to shared memory is checked for access before executing. If a virtual CPU attempts a memory operation for which it has insufficient access, the CREW system can capture it and record appropriate dependences between that virtual CPU and the ones that had previously access the same page. Thus, SMP-ReVirt captures shared-memory dependences at page granularity.

Flashback [75] focuses on application-level RnR for debugging, and captures only the interactions between the application being debugged and the operating system, like system calls and signals. It uses OS support to achieve this and only supports single-threaded programs.

Scribe [42] uses a similar virtual-memory-based mechanism mechanisms to capture the interleaving of shared memory accesses among concurrent threads of execution of a single process. Scribe tries to reduce the number of changes in page-access permissions to reduce the number of dependences that need to be recorded and improve the performance. It does so by letting a thread retain its access to a page for a configurable amount of time and stalling other threads if they try to make a conflicting access the page in that period. Although the paper reports reasonable overheads for system applications with low levels of sharing, it is not clear how it will perform for sharing- and synchronization-intensive programs.

These software-based approaches are either inherently designed for uniprocessor executions or suffer significant slowdown when applied to multiprocessor executions. DoublePlay [77] made efforts to make replay on commodity multiprocessors more efficient. It timeslices multiple threads on one processor and then runs multiple time intervals on separate processors. Hence, it only needs to record the order in which threads in each time interval are timesliced on the corresponding processor. This technique easess logging by only requiring the logger to record the order in which the time slices are executed within a time interval. However, DoublePlay uses an additional execution to create checkpoints off which multiple time intervals can be run in parallel. It also needs to use modified binaries (in particular, a modified libc) for efficient
Search-Based Software-Only Solutions

ODR \cite{6} and PRES \cite{62} are probabilistic replay techniques for reproducing concurrency bugs. The idea is to record only a subset of non-deterministic events required for deterministic replay (to reduce the recording overhead) and use a replayer that searches the space of possible executions to reproduce the same application output or bug, respectively. Respec \cite{50} targets online replay scenarios. It records a subset of non-deterministic events and uses the online replay run to provide external determinism. The idea is to retry the execution from the last checkpoint when a divergence happens. Like DoublePlay \cite{77}, it needs to use modified binaries.

CLAP \cite{36} records the branch trace of each thread independently and uses symbolic execution and SMT-based constraint solving \cite{23} to generate a parallel schedule and program inputs for replay. It uses static analysis to identify shared memory accesses. Symbolic execution is used to calculate constraints on program inputs and shared memory values to generate the observed thread paths. Finally, it uses an SMT solver to solve the constraints and infer a consistent ordering of shared memory reads and writes.

Stride \cite{91} tries to improve the speed of off-line searching at the cost of some recording overhead. It combines access-order recording with off-line search to reconstruct inter-thread dependences. Instead of recording exact source and target instructions in each dependence, it records the approximate locations of the instructions. This allows it to relax the requirement of recording the access ordering atomically with performing the accesses themselves and, thus, reduce the recording overhead. It also records the value returned by reads. An off-line search algorithm then infers the exact source and destinations of each dependence using the recorded information. The authors prove that their search algorithm has polynomial time complexity, unlike the exponential worst-case complexity of pure search-based replay schemes.

Zamfir et al. \cite{89} propose “Execution Synthesis”, a debugging technique that skips recording altogether and instead relies on static analysis and symbolic executions to generate a parallel schedule and necessary program inputs that can reproduce the reported bug symptoms. It only “records” a core dump (typically, provided as part of the bug report) and outputs an execution trace that can result in the state captured in the core dump. The generated trace can then be deterministically replayed using their playback infrastructure.
1.2.2 Hardware-Assisted RnR Solutions

Hardware-based solutions usually use hardware to record memory races to reduce the overhead of RnR for multiprocessor executions. Most reuse existing coherence mechanisms for this purpose (BugNet [59] and LRewrap [18] are exceptions and will be discussed shortly.)

FDR [85] and RTR [86] are among the very first race recording techniques proposed. They record dependences between pairs of instructions. This can result in large log sizes. Also, the resulting fine-grain ordering constraints can hurt replay efficiency. While FDR only supports SC, RTR supports TSO by recording the value of loads that may violate SC.

To remedy the large log size and fine-grain ordering constraints of earlier designs, most recent MRR techniques are based on the concept of Chunks of instructions (also called Blocks or Episodes in the literature). The idea is to divide each thread’s execution into a sequence of dynamic groups of instructions. The execution of each group or chunk is logged as the number of instructions it contains. The MRR hardware also records a partial or total order of all of the application’s chunks. For each inter-thread data dependence, the chunk that contains the source of the dependence is ordered before the chunk that contains the destination. Coherence transactions are used to identify chunk boundaries. During replay, each chunk is executed after all of its predecessors (in the recorded order) and before any of its successors. In this manner, all inter-thread dependences are enforced. The replayer has to count the number of instructions executed in a chunk in order to know when its execution is complete.

Strata [58] uses the concept of stratum to record memory interleaving. A stratum creates a time layer across all the logs for the running threads, which separates all the memory operations executed before and after the stratum. This is done by augmenting the coherence messages with a Log Stratum bit, which can be set by the processor initiating the miss or by a processor that provides the data. If the bit is set, all processors record an entry in their logs at the same time, starting a new epoch. No inter-thread dependence can exist within a single epoch; all such dependences cross strata boundaries. As such, at replay time, dependences can be easily enforced by executing one strata at a time.

DeLorean [56] and Capo [57] use the speculative multithreading hardware of BulkSC [15]. The underlying hardware enforces SC while allowing aggressive out-of-order execution of instructions. The execution is recorded by logging the order in which processors commit their chunks.

Rerun [34] is a chunk-based technique for conventional multiprocessors and does MRR for machines
with directory coherence. It uses a Scalar Lamport Clock \cite{43}, piggybacked on coherence transactions, to order chunks of different processors. The paper also includes a proposal to integrate RTR’s solution for TSO recording with their chunk-based scheme. Timetraveler \cite{81} builds on Rerun and reduces its log size. While Rerun terminates a chunk upon the first conflicting coherence transaction, Timetraveler allows the chunk to grow beyond that to reduce the chunk count and, thus, the log size.

Intel MRR \cite{66} and CoreRacer \cite{67} are similar solutions, but for snoopy protocols. Instead of piggybacking timestamps on coherence messages, every time that a chunk commits, the event is broadcasted with a bus transaction. These global communications are used to synchronize timestamps across different processors. Chunk ordering can be easily established according to the recorded chunk timestamps. While the former is limited to SC, the latter supports TSO by recording the number of stores pending in the processor’s write buffer when a chunk terminates. This allows CoreRacer to correctly account for reordered and forwarded loads by simulating the write buffer’s content during replay.

Karma \cite{10} is the first chunk-based RnR technique that explicitly targets replay parallelism without relying on speculative hardware. It is a whole-system (rather than application-level) RnR scheme for directory protocols. It records bidirectional dependences between source and destination chunks and, hence, makes some modifications to the cache coherence messages. By recording the chunk ordering as partially-ordered directed acyclic graph (DAG), as opposed to scalar timestamps used in earlier solutions, it can accommodate parallel replay.

LReplay \cite{18} is a hardware-assisted MRR solution that does not monitor coherence transactions. Instead, it includes a non-trivial centralized component that directly tracks the memory operations performed by all cores. It relies on this hardware to detect inter-processor dependences. It supports TSO using RTR’s approach. Due to its specific recording technique, its replay algorithm is complicated and needs to simulate all instructions.

BugNet \cite{59} records user processes by storing the result of load instructions in a hardware-based dictionary. It does so by recording the content of each fetched cache line upon first access to it. This is enough to handle both input and memory-interleaving non-determinism and allows each thread to be replayed independently. However, BugNet still needs a solution to record inter-thread dependences, for which it uses FDR \cite{85}.

Lee et al. \cite{48,49} build on BugNet but use off-line search to infer inter-thread dependences for SC \cite{49}.
and TSO executions. They use BugNet’s cache-line recording technique and thus can replay each thread independently. They also periodically record some Strata hints to speed-up the off-line search. Using the results of per-thread replays and the hints, inter-thread data dependences can be determined off-line.

Rainbow builds on Strata and uses SC-violation detection hardware to record non-SC executions. When an SC violation is detected, it records some information about delayed and pending instructions that allows it to replay the situation correctly. This scheme needs to augment coherence messages with instruction identifiers. Moreover, to detect SC violations, it requires a central data structure that tracks the coherence transactions of all processors. Due to this centralized data structure, it cannot support distributed-directory coherence protocols. The scheme claims to support general relaxed memory models, although the operation is not described in enough detail.

1.3 Thesis Contributions and Outline

This thesis seeks to assess and improve practicality of hardware-assisted RnR. In particular, we focus on RnR of parallel programs on parallel hardware.

We believe that a practical hardware-assisted RnR solution should possess the following properties in order to improve its chances of being embraced by processor vendors and system software developers:

- It should work with unmodified programs (e.g., no need to recompile programs in-order to make them RnR-able);
- It should have well-defined system architecture with clean separation and minimal interface between hardware and software components;
- Its MRR hardware should be practical to build, i.e., have low design complexity while supporting coherence protocols and memory models of existing processors; and,
- It should accommodate a wide range of use-cases (such as debugging, security and high-availability).

In this work, we consider these issues in a series of projects that build on top of each other. The following paragraphs provide a brief overview of each project and highlight its contributions:
1.3.1 First Physical Prototype of Hardware-Assisted RnR

We begin by introducing QuickRec (Chapter 2), the first physical realization of a hardware-assisted RnR system. The focus of this project is on understanding and evaluating the implementation issues of hardware-assisted RnR on a real platform.

On the hardware side, QuickRec presents an FPGA-based prototype of a multicore Intel Architecture (IA) platform that is enhanced with MRR hardware. On the software side, it presents a full software stack, based on a modified Linux kernel, that is capable of recording program inputs and managing the new MRR hardware.

The main contributions of QuickRec are the following:

1. The implementation of the first IA multicore prototype of RnR for multithreaded programs. We demonstrate that hardware-assisted MRR can be implemented efficiently on a real multicore IA system with modest hardware complexity.

2. A description of several key implementation aspects. Specifically, we show how to efficiently handle the idiosyncrasies of doing MRR on the x86 architecture. We also describe the intricate inter-play between MRR and input recording components as well as the elaborate hardware-software interface required for a working system.

3. An evaluation of the system. We show that the rate of memory log generation is insignificant, given today’s bus and memory bandwidths, and that the recording hardware has negligible performance overhead (as expected). The evaluations however point to the software stack as the major source of overhead (incurring an average recording overhead of nearly 13% in our workloads), an issue that was largely ignored by previous work on hardware-assisted RnR.

1.3.2 Enabling Fast Replay Through Replay Parallelism

Although QuickRec can record parallel programs, the replay is sequential due to its MRR design. This was a compromise to keep the hardware simple — in particular, to avoid changing the cache coherence protocol. This is important since coherence protocols are among the hardest-to-verify components of multi-processor designs and processor vendors are reluctant to embrace solutions involving coherence protocol changes.
Parallel replay, however, can significantly improve replay speed which is a key factor in many use cases of RnR such as online security analysis and fault tolerance.

To address this problem, we propose Cyrus (Chapter 3), an RnR scheme that can record parallel programs and replay them in parallel without making any changes to the cache coherence protocol and messages.

The main contributions of Cyrus are as follows:

1. It is the first hardware-assisted approach for application-level RnR that explicitly targets parallel replay. Moreover, it achieves these goals without requiring any modifications to commodity snoopy cache coherence.

2. It introduces a novel MRR technique based on a hybrid hardware/software design. This hybrid design is the key idea that enables application-level RnR with parallel replay while keeping the hardware extension simple. The hardware component records a raw and incomplete log (due to our recording requirements of only application-level interactions and no cache coherence protocol changes). This log is then processed and transformed into a complete log by an on-the-fly software backend pass. As the raw log is being generated, this pass transforms it for high replay parallelism. This design can also flexibly trade-off replay parallelism for log size.

1.3.3 Efficient RnR for Relaxed Memory Models

QuickRec and Cyrus are limited in terms of the memory consistency models they support: Total Store Order (TSO) and Sequential Consistency (SC), respectively. To enable RnR for other architectures — such as ARM, Power and Tile — whose memory model is more relaxed, we propose RelaxReplay (Chapter 4). It is a general hardware-assisted MRR scheme that only relies on conventional cache coherence with write atomicity (Section 4.3). Therefore, it works for any current relaxed-consistency multiprocessor.

The main contributions and salient characteristics of RelaxReplay are the following:

1. It is the first complete solution for hardware-assisted MRR that works for any relaxed-consistency multiprocessors with write atomicity.

2. It only relies on the write atomicity property of coherence protocols, and not on knowing the detailed specifications of the particular relaxed-consistency model. Such specifications are often high-level
and hard to map to implementation issues.

3. It can be combined with the specific chunk-ordering algorithm of any existing chunk-based MRR proposal. As a result, that proposal, designed for a certain coherence protocol, can now record relaxed-consistency executions.

4. It produces a compact log representation of a relaxed-consistency execution, which also enables efficient deterministic replay with minimal hardware support.

1.3.4 Replay Debugging

After considering the challenges of building practical RnR systems, the next question to be answered is that of their usability. The last part of this dissertation investigates the issue of using the RnR technology for program debugging, arguably its most commonly cited use-case. RnR enables deterministic reproduction of hard-to-repeat software bugs. However, simply providing support for repeatedly stumbling on the same bug does not help diagnose it. For bug diagnosis, developers typically augment the program source with debug code — E.g., by creating and operating on new variables, or printing state. Unfortunately, this renders the RnR log inconsistent and makes Replay Debugging (i.e., debugging while using an RnR log for replay) dicey at best.

To attack this problem, we propose rdb [33], the first scheme for replay debugging that guarantees exact replay in the presence of debug code. rdb relies on two mechanisms. The first one is compiler support to split the instrumented application into two executables: one that is identical to the original program binary, and another that encapsulates all the added debug code. The second mechanism is a runtime infrastructure that replays the application and, without affecting it in any way, invokes the appropriate debug code at the appropriate locations. We describe an implementation of rdb based on LLVM [3] and Pin [53], and show an example of how rdb’s replay debugging helps diagnose a real bug.
Chapter 2

QuickRec: Prototyping an Intel Architecture Extension for Record and Replay of Multithreaded Programs

2.1 Introduction

To record memory access interleaving with low overhead, researchers have proposed several hardware-assisted RnR designs (e.g., [10, 18, 31, 34, 56, 57, 58, 59, 66, 67, 81, 85, 86]). These proposals have outlined RnR systems that have negligible overhead during recording and can operate with very small log sizes. To evaluate these systems, the authors typically implement their techniques in software-based simulators. In addition, they typically run their simulations without an operating system that manages and virtualizes their special hardware. The exceptions are LReplay [18], which extends and simulates the RTL (Register Transfer Level) model of a chip multiprocessor and does not discuss system software issues, and Capo [57] and Cyrus [31], which use an RnR-aware operating system on top of simulated hardware.

Although this evaluation approach helps assess the efficacy of the proposed algorithms, it ignores practical aspects of the design, such as its integration with realistic cache coherence hardware, coping with relaxed memory models, and virtualizing the recording hardware. In addition, promoting RnR solutions into mainstream processors requires a co-design with the system software that controls the hardware, and omitting software effects from the evaluation presents only part of the overall performance picture.

To evaluate the practical implementability of hardware-assisted RnR, we have built QuickRec, the first multicore IA-based prototype of RnR for multithreaded programs. QuickRec is based on QuickIA [82], an Intel emulation platform for rapid prototyping of new IA extensions. QuickRec is composed of a Xeon server platform with FPGA-emulated second-generation Pentium cores, and Capo3, a full software stack for managing the recording hardware from within a modified Linux kernel.

This work focuses on identifying and characterizing RnR-related implementation issues. Specifically,
we describe how QuickRec records the memory access interleaving of threads, and how to integrate this support into a commodity IA multicore. We discuss subtle issues related to capturing the ordering of instructions with multiple memory accesses, and the interaction with the memory consistency model. We also discuss how Capo3 records the inputs to processes, manages the replay logs, and virtualizes the hardware components. We provide data characterizing QuickRec’s recording performance and log parameters. Overall, our evaluation demonstrates that RnR can be practical for real IA multicore systems.

This effort has led to some lessons learned, as well as to some pointers for future research directions. In particular, we find that the main challenge of RnR systems is to take into account the idiosyncrasies of the specific architecture used, such as single instructions producing multiple memory transactions. Further, we find that the software stack has a dominant role in the overall system performance, as it manages the logs. Based on these experiences, we suggest focusing future research on recording input events efficiently, and on replay techniques that are tolerant of the micro-architectural details of the system.

The main contributions of this work are the following:

- The implementation of the first IA multicore prototype of RnR for multithreaded programs. The prototype includes an FPGA design of a Pentium multicore and a Linux-based full software stack.
- A description of several key implementation aspects. Specifically, we show how to efficiently handle x86 instructions that produce multiple memory transactions, and describe the elaborate hardware-software interface required for a working system.
- An evaluation of the system. We show that the rate of memory log generation is insignificant, given today’s bus and memory bandwidths. In addition, the recording hardware has negligible performance overhead. However, the software stack incurs an average recording overhead of nearly 13%, which must be reduced to enable always-on use of RnR.

This chapter is organized as follows: Section 2.2 introduces the QuickRec recording hardware; Section 2.3 describes the Capo3 system software; Section 2.4 characterizes our prototype; Section 2.5 discusses using replay for validation; Section 2.6 outlines related work; Section 2.7 describes lessons learned; and Section 2.8 concludes the chapter.
Figure 2.1: Photograph of the QuickRec prototype with FPGAs in CPU sockets (a); architecture of the QuickIA processor-emulation platform (b); and architecture overview of the extended Pentium core in QuickRec, where circled numbers identify the main CPU touch points required to enable recording (c).

2.2 QuickIA Recording System

The QuickRec recording system prototyped in this work is built on a FPGA processor-emulation platform called QuickIA. This section introduces QuickIA and then describes the changes we added to support RnR. Figure 2.1a shows a picture of the QuickRec recording system testbed.

2.2.1 QuickIA Processor Emulation Platform

The QuickIA processor emulation platform [82] is a dual-socket Xeon server board in which Xeon CPUs are substituted with FPGA modules from XstreamData [84]. Each such FPGA module is composed of two Compute FPGAs and one Bridge FPGA, as shown in Figure 2.1b. Each Compute FPGA implements a second-generation Pentium core with private L1 and L2 caches. The Bridge FPGA implements the interconnect between the two Compute FPGAs and the Intel Front Side Bus (FSB), which connects the two CPU sockets to the Memory Controller Hub (MCH) on the platform. This allows both CPU sockets to be fully cache coherent, with full access to memory and I/O. The QuickIA system implements a MESI coherence protocol with L2 as the point of coherence.

The Pentium cores used in the QuickIA emulation platform are fully synthesizable. Each core features a dual-pipeline in-order CPU with floating-point support. In addition, each core is extended with a set of additional features to reflect the state of the art of modern processors. These changes include L1 cache line size increase to 64 bytes, Memory Type Range Registers, physical address extension, and FSB xAPICs.
The four emulated Pentium cores run at 60MHz. While this clock frequency is low, the memory bandwidth is also low (24MB/s), which means that the ratio between CPU speed and memory bandwidth is similar to that of today’s systems. The QuickIA system includes 8GB of DDR2 memory and basic peripherals (network, graphics card and HDD), and can boot a vanilla SUSE Linux distribution. The basic platform parameters are shown in Table 2.1.

### 2.2.2 Recording Interleaving Non-Determinism

To record the non-determinism of memory access interleaving, the RTL of the synthesizable Pentium core is augmented to capture the order of memory accesses. This support includes mechanisms to break down a thread’s execution into *chunks* (i.e., groups of consecutive dynamic instructions), and then order the chunks across cores. A significant effort was invested in integrating this support into the Pentium core without adding unnecessary complexity. Some of the main challenges we faced include dealing with the IA memory model, and coping with x86 instructions with multiple memory accesses. The extended Pentium core is then synthesized and downloaded into FPGAs to boot up the QuickRec emulation platform. A high-level overview of the extended Pentium core is shown in Figure 2.1c. In the figure, the *Memory Race Recorder* (MRR) box implements the functionality for recording memory access interleaving, while the circled numbers indicate the CPU touch points required to enable it.

### Capturing and Ordering Chunks

The QuickRec recording system implements a mechanism similar to the Intel MRR [66] to divide a thread’s execution into chunks. It adds Bloom filters next to the L1 cache to capture the read and write sets of

<table>
<thead>
<tr>
<th>Table 2.1: QuickIA platform parameters.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
</tr>
<tr>
<td>Clock</td>
</tr>
<tr>
<td>L1 data cache</td>
</tr>
<tr>
<td>L2 cache</td>
</tr>
<tr>
<td>Coherence</td>
</tr>
<tr>
<td>Memory</td>
</tr>
</tbody>
</table>
the memory accesses in a chunk (R-set and W-set in Figure 2.1c). The line addresses of the locations accessed by loads and stores are inserted into their respective set at retirement and at global observation time, respectively. A thread’s chunk is terminated when the hardware observes a memory conflict (i.e., a data dependence) with a remote thread. Conflicts are detected by checking the addresses of incoming snoops against addresses in the read and write sets. When a conflict is detected, a counter (Counter in Figure 2.1c) with the current chunk size is logged into an internal chunk buffer (CBUF in Figure 2.1c), along with a timestamp that provides a total order of chunks across cores. The chunk-size counter counts the number of retired instructions in the chunk. After a chunk is terminated, the read and write sets are cleared, and the chunk-size counter is reset.

In addition to terminating a chunk on a memory conflict, QuickRec can be configured to terminate a chunk when certain system events occur as well, such as an exception or a TLB invalidation. A chunk also terminates when the 20-bit chunk-size counter overflows. Additionally, the addresses of lines evicted from L2 are looked up in the read and write sets and, in case of a hit, the chunk also ends. This is done because the read and write sets would not observe future coherence activity on these evicted lines. Further information on chunk termination is provided in Section 2.2.3.

Figure 2.1c shows the main CPU touch points required to enable the chunking mechanism described above. The first CPU touch point is hooked-up to the external L1 snoop port to allow snoops to be forwarded to the MRR for address lookups. The second and third CPU touch points are hooked-up to the U and V integer execution pipelines of the Pentium core. They provide diverse functionalities, such as forwarding load and store line addresses to the MRR for insertion into the read and write sets, and forwarding the instruction retirement signal to the MRR to advance the chunk-size counter.

One of the complexities we encountered when integrating the chunking mechanism into the Pentium core was keeping updates to the read and write sets within one cycle, so that they can be performed in parallel with a cache access. The problem is that only the lower bits of the addresses are available at the beginning of a cache cycle, as the upper bits (tag bits) are provided usually late in the cycle, after a DTLB access. To preserve a single cycle for the read and write set update, addresses (tag plus set bits) are buffered into a latch stage before they are fed to the Bloom filter logic. To compensate for the delayed update of the read and write sets, these buffers are also looked-up on external snoops, at the cost of additional comparators for each address buffer.
Integration into the IA Memory Model

The IA memory model, very similar to the Total Store Model (TSO), allows a load to retire before a prior store to a different address has committed, hence effectively ordering the load before the prior store in memory. In this situation, using the retired instruction count is not sufficient to guarantee that loads and stores are ordered correctly during replay. This is because, during replay, instructions are executed in program order. Hence, regardless of when the store committed to memory during the recorded execution, the store is evaluated before the load during replay. To address this problem, QuickRec implements a solution similar to the one proposed in CoreRacer [67] to handle TSO. The idea is to track the number of pending stores in the store buffer awaiting commit and, at chunk termination, append the current number to the logged entry. This number is called the Reordered Store Window (RSW) count. The MRR is hooked-up to the memory execution unit to enable this functionality.

Instruction Atomicity Violation

In the x86 ISA, an instruction may perform multiple memory accesses before completing execution. For instance, a split cache line access, which is an access that crosses a cache line boundary, requires more than one load or store operation to complete. In addition, some complex instructions require several memory operations. For example, the increment instruction (INC) performs a load and a store operation. At the micro-architecture level, these instructions are usually broken down into multiple micro-operations or µops. An Instruction Atomicity Violation (IAV) occurs if an event causes the QuickRec recording system to log a chunk in CBUF in the middle of such an instruction execution. An example of such an event is a memory conflict. Because software is usually oblivious of split cache line accesses and µop execution, IAVs make it difficult for software to deterministically reproduce a program execution.

Figure 2.2 shows an example. Thread T0 executes instruction INC A, which increments the value in memory location A. The instruction breaks down into the three µops shown in the figure: a read from A into user-invisible register r\text{tmp}, the increment of r\text{tmp}, and the store of r\text{tmp} into A. At the same time, thread T1 writes A. Suppose that the operations interleave as shown in the time line.

When the store in T1 executes at time t2, a conflict with T0 is detected, since µop\text{01} has read from the same address at t0. Therefore, QuickRec terminates the chunk in T0 and logs an entry in T0’s CBUF. This chunk is ordered before the store in T1. However, since the INC instruction has not yet retired, INC is not
counted as belonging to the logged chunk. Then, when the INC instruction executes µop03 and retires at t3, a conflict with T1 is detected. This causes QuickRec to terminate the chunk in T1 and log an entry in T1’s CBUF that contains the store. The logged chunk is ordered before the currently-executing chunk in T0, which is assumed to include the INC instruction. Consequently, in this naive design, the replay would be incorrect. Indeed, while during recording, µop01 occurred before the store in T1, which in turn occurred before µop03, during replay, the store in T1 will be executed before the whole INC instruction.

This problem occurs because the INC instruction suffers an IAV. Although the instruction has performed some memory transactions during the earlier chunk in T0, since the instruction has not retired when the chunk in T0 is logged, the instruction is counted as belonging to the later chunk in T0.

The QuickRec recording system solves this problem by monitoring the retirement of the multiple memory accesses during the execution of the instruction. Specifically, it uses a dedicated IAV counter to count the number of retired memory transactions for a multi-line or multi-operation instruction (Figure 2.3). The IAV counter is incremented at every retired memory transaction, and is reset when the instruction retires. At chunk termination, if the IAV counter is not zero, the current instruction has not retired, and an IAV has been detected. In this case, QuickRec saves the value of the IAV counter in the log entry of the terminated chunk. Since, during replay, we know exactly the number (and sequence order) of the memory transactions that need to occur in a given instruction, by reading the IAV counter and examining the RSW count (Section 2.2.2), we know how many memory operations of the subsequent instruction need to be performed before completing the current chunk. In our actual implementation, the IAV counter is incremented by 1 for each access in a split cache line reference, and by 2 for any other access. With this design, an odd counter
value indicates that the chunk terminated between the accesses of a split cache line reference.

![IAV counter mechanism diagram]

Figure 2.3: IAV counter mechanism.

Consider again the example of Figure 2.2. When T1 executes the store at time t2 and a conflict is detected in T0, the INC instruction has not yet retired. The IAV counter in T0 is 2, since the only retired access is that of \( \mu \text{op}_01 \). Therefore, an IAV is detected. The QuickRec recording system terminates the chunk in T0 and, as it logs the chunk, appends to it the value of the IAV counter. This log entry conveys to the replayer the information that an IAV has occurred in the chunk and that only the first memory \( \mu \text{op} \) had retired at the time of chunk termination.

Instruction atomicity violation was first introduced in [64] and then described in [67]. The main difference with [67] is that we log the number of retired memory transactions instead of the number of transferred bytes. The advantage of logging memory transactions over transferred bytes is the reduction in the log size.

**Log Management**

CBUF is organized into four entries, where each is as large as a cache line. When a chunk terminates, a 128-bit chunk packet is stored in CBUF. When a CBUF entry is full, it is flushed by hardware to a dedicated memory region called CMEM. To minimize the performance impact, this is done lazily, during idle cycles, by bypassing the caches and writing directly to memory. Occasionally, however, the chunking mechanism must stall the execution pipeline to allow CBUF to drain to CMEM to avoid overflow.

There are two main packet types inserted into CBUF, namely the timestamp packet (TSA) and the chunk
packet. Both are very conservatively sized as 128-bit long. Once a TSA is logged for a thread, subsequent chunk packets for that thread only need to log the timestamp difference (TSD) with respect to the last TSA. The TSA is then logged again when the value in TSD overflows. Note that this also causes a chunk termination. Figure 2.4 shows the format of these two packets. The chunk packet contains the TSD, chunk size (CS), and RSW and IAV counts. It also contains a Reason field, which indicates why the chunk was terminated — e.g., due to a RAW, WAR or WAW conflict, an exception, or a chunk-size overflow. Table 2.2 lists the main reasons for terminating chunks.

<table>
<thead>
<tr>
<th>Type</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAW</td>
<td>RAW conflict between chunks</td>
</tr>
<tr>
<td>WAR</td>
<td>WAR conflict between chunks</td>
</tr>
<tr>
<td>WAW</td>
<td>WAW conflict between chunks</td>
</tr>
<tr>
<td>WAB</td>
<td>Both WAR and WAW conflicts between chunks</td>
</tr>
<tr>
<td>EXCEPT</td>
<td>Exception, interrupt, far call, or far return</td>
</tr>
<tr>
<td>EVICT</td>
<td>Line eviction from L2 that hits the R-set or W-set</td>
</tr>
<tr>
<td>CS_OVERFLOW</td>
<td>Chunk size overflow</td>
</tr>
<tr>
<td>TLBINV</td>
<td>TLB invalidation</td>
</tr>
<tr>
<td>XTC</td>
<td>Explicit chunk termination instruction</td>
</tr>
</tbody>
</table>

Table 2.2: Main reasons for terminating chunks. WAB (Write-After-Both) is when a write in one chunk hits in both the read and the write set of another chunk.
2.2.3 Programming Interface

The QuickRec recording system contains a set of registers to configure and program the hardware. For instance, using these registers, the hardware can be programmed to record memory non-determinism for user-level code only, or for both user- and system-level code. It can also be programmed to terminate a chunk under certain conditions only, such as a specific type of conflict or exception. Privileged software can also specify where in memory the logs are written for each recorded thread. The QuickRec recording system also has a status register that is updated at chunk termination time to capture the state of the machine at that point. Among other information, it captures the reason for the chunk termination. Some of its information is copied to the Reason field of the logged chunk packet. A more detailed discussion of the programming interface, and how the system software uses it to manage the QuickRec hardware is provided in Section 2.3.3.

QuickRec extends the ISA with two new instructions: one that terminates the current chunk (XTC), and one that terminates the current chunk and flushes CBUF to memory (XFC). The use of these two instructions is restricted to privileged software. Examples of their use are discussed in Sections 2.3.4 and 2.3.6.

2.2.4 Other Issues

Because the main purpose of this work is to demonstrate the feasibility of hardware-assisted RnR, this prototype only addresses the issues that are critical to support RnR for the majority of programs. For instance, the prototype only supports Write-Back (WB) memory [37], which constitutes the majority of memory accesses in current programs. Memory accesses to Uncacheable (UC) or Write-Combining (WC) memory are not tracked, and cause the system to terminate a chunk. Chunking is resumed when the next access to WB memory occurs.

In some cases, the IA memory model allows accesses to WB memory to have different ordering semantics than TSO. For instance, in fast string operations, a store to WB memory can be reordered with respect to a prior store. To ensure that QuickRec’s RSW and IAV support work properly, we disable this feature, so that all loads and stores obey TSO semantics.

Although we do not discuss how to extend our mechanisms to support Hyperthreading, the changes required to do so are minimal. In modern IA cores, there already exist mechanisms for detecting conflicts between the different hardware thread contexts sharing the same cache. Therefore, in order to enable RnR on
a Hyperthreaded core, one would only need to replicate certain resources for each hardware thread context (e.g., the read and write sets).

2.3 Capo3 System Software

To manage the QuickRec hardware, we built a software system called Capo3. Capo3 draws inspiration and borrows many of the concepts and principles from Capo\cite{57}, a system designed for hardware-assisted RnR. However, Capo3 must run on real hardware, and as such, we encounter several issues that were abstracted away in Capo due to using simulated hardware. In this section, we compare Capo3 with Capo, describe its architecture, and focus on several of its key aspects.

2.3.1 Comparing Capo3 with Capo

Capo3 uses some of the basic ideas introduced by Capo, including the Replay Sphere and the Replay Sphere Manager (RSM). The Replay Sphere abstraction is the single application (or a group of applications) that should be recorded/replayed in isolation from the rest of the system. The Replay Sphere Manager is a software component that is responsible for correctly capturing non-deterministic input and memory access interleaving.

Capo3 also uses the same basic techniques as Capo to record program inputs, including interactions between the operating system and processes (e.g., system calls and signals), and non-deterministic instructions (i.e., \texttt{rdtsc} and \texttt{cpuid}). Recording these input events guarantees that, during replay, the same data can be injected into the user-mode address space. However, some system calls also affect the kernel-mode data structures of the program. Hence, to ensure that their effects are deterministically recreated during replay, we re-execute these system calls during replay.

To correctly capture kernel state, like in Capo, the RSM enforces a total order of input events during recording. The same total order is enforced during replay. This total order has major performance and correctness implications, as shown in Sections 2.3.6 and 2.4.

Capo3 uses a different software architecture than Capo. Specifically, it places the bulk of the RnR logic in the kernel — whereas Capo used \texttt{ptrace} to capture key events with user-mode logic. Moreover, since Capo3 must virtualize real hardware, its design must support a hardware/software interface to enable context switches, record memory access interleaving when the kernel is running with interrupts enabled,
and manage subtle interactions between QuickRec hardware and Capo3 software.

### 2.3.2 Capo3 Architecture

Capo3 implements the RSM as an extension to the Linux kernel. To record an execution, a driver program initializes a Replay Sphere using the RSM-provided interface. The RSM then logs the input events, sets-up the MRR hardware to log the memory access interleaving, and makes all these logs available to the driver program that is responsible for the persistent storage and management of the logs. Figure 2.5 shows the high-level architecture of the Capo3 software stack.

![Figure 2.5: Overall architecture of Capo3. Dashed boxes indicate QuickRec-specific components.](image)

Our decision to use a kernel-based implementation was driven by the observation that the Linux kernel has well-defined places to enable the kernel to interpose on processes. As a result, Capo3 only requires the kernel to be augmented in a few key places, so it can interpose on all system calls, signals, and memory copies between processes and the kernel. These changes also allow Capo3 to virtualize the QuickRec hardware by saving/restoring QuickRec state upon a context switch. Overall, our kernel-based implementation consists of roughly 3.4K lines of code, where the bulk of the code is dedicated to managing the logs, and is well isolated from the rest of the kernel.

There are four different sources of input non-determinism that the RSM captures: system calls, data copied to user-mode address spaces, signals, and non-deterministic processor instructions. To bind these
recorded events to their corresponding threads, the RSM assigns a unique R-Thread ID to each recorded thread. During replay, each thread is guaranteed to get the same R-Thread ID. These R-Thread IDs are also used to associate chunks recorded by the QuickRec hardware with their corresponding threads.

### 2.3.3 Virtualizing the QuickRec Hardware

To virtualize the QuickRec hardware, the RSM uses the programming interface outlined in Section 2.2.3. The main components of this interface are the seven registers shown in the lower level of Figure 2.5. Specifically, the Chunk Memory Pointer (\texttt{CMEM\_PTR}) points to CMEM, which is the in-memory buffer that contains the logged chunk data. Each thread gets its own CMEM. The Chunk Memory Index (\texttt{CMEM\_IDX}) indicates the location in CMEM where the next CBUF entry is to be written. This register is updated by hardware as CBUF entries are written to memory. The Size Register (\texttt{CMEM\_SZ}) indicates the size of CMEM. The Threshold Register (\texttt{CMEM\_TH}) indicates the threshold at which a CMEM overflow interrupt is generated. The Control Register (\texttt{MRR\_CTL}) enables and disables chunking under certain conditions, while the Status Register (\texttt{MRR\_STATUS}) provides the status of the hardware. These last two registers were described in Section 2.2.3. Finally, the Flags Register (\texttt{MRR\_FLAGS}) controls kernel-mode recording and is discussed later.

It is the RSM's responsibility to manage the CMEM buffers and virtualize these hardware registers so that different threads can use the hardware without having their chunk data mixed-up. In particular, this involves: (i) ensuring that a valid CMEM pointer is configured before recording begins, (ii) allocating a new CMEM buffer when the previous one fills-up, and (iii) writing to CMEM any contents remaining in the CBUF before a thread is pre-empted.

When a CMEM buffer reaches its capacity, Capo3 writes it to a file. Because there may be multiple full CMEM buffers in the system waiting to be written to the file, the RSM serializes this write operation using a work queue handled by a dedicated thread. This work queue provides an effective back-pressure mechanism when the buffer completion rate of the recorded threads exceeds the speed of the thread that empties the queue. Specifically, when the work queue becomes full, the RSM puts the recorded threads to sleep until the work queue can catch up. This mechanism preserves correctness, although it may negatively impact recording performance.
2.3.4 Handling Context Switches

On a context switch, the RSM first executes an XFC instruction to ensure that the current chunk terminates, and that all the residual data in the processor’s CBUF are flushed to CMEM. This is needed to avoid mixing the log of the current thread with the next thread.

Once this has been performed, the RSM saves and restores the values of the registers in the MRR. Specifically, for the current thread, it saves the registers that the hardware may have modified during execution. They are the CMEM_IDX and MRR_FLAGS registers. Then, before the next thread can execute, the RSM restores the thread’s prior CMEM_PTR, CMEM_IDX, CMEM_SZ, CMEM_TH, MRR_CTL, and MRR_FLAGS values, enabling it to correctly resume execution.

2.3.5 Recording in Kernel Mode

Certain parts of the kernel can interact with a process’ address space, creating the potential for the kernel to have races with user-level instructions. The copy_to_user family of functions in the Linux kernel is an example of such code. Hence, in order to record all the memory access orderings that can affect the execution of an application during replay, the QuickRec hardware must also capture the execution of these kernel-level memory accesses.

QuickRec provides a flag that, if set, allows the MRR to record kernel instructions as well as user-mode instructions. Hence, to record sections of the kernel such as copy_to_user(), our initial approach was to set that flag prior to entering copy_to_user() and reset it after returning from copy_to_user(). The problem with this approach is that an asynchronous interrupt (e.g., from a hardware device) or a page fault can occur during the execution of copy_to_user(). In this case, since the flag is still set, QuickRec would incorrectly record the interrupt or page fault handler code.

Our solution to this problem is to have an MRR_FLAGS register, where the least significant bit (LSB) acts as the previously-mentioned flag. On entry to copy_to_user(), we set the LSB, while on returning from it, we reset it. Moreover, the register operates as a shift register. When an exception is taken, the register automatically shifts left with a 0 being inserted into the LSB, which disables recording. Upon returning from the exception handler (as indicated by the iret instruction of x86), the register shifts right, restoring the previous value of the LSB. If the exception has happened in the middle of a copy_to_user(), this design disables recording as soon as the exception is taken, and resumes it as soon as the execution returns to
2.3.6 Handling Input/Chunking Interactions

The RSM component that records the input log and the one that manages the chunking log proceed almost independently from each other, each creating a total order of their events. However, in our initial implementation, we observed a subtle interaction between the two components that resulted in occasional deadlocks.

The problem occurs if a chunk includes instructions from both before and after an input event. In this case, the dependences between chunks and between inputs may intertwine in a way that causes deadlock.

As an example, consider Figure 2.6a where chunks C1 and C2 execute on processors P1 and P2. Suppose that C2 first executes an input event that gets ordered in the input log before an input event in C1. Then, due to a data dependence from P1 to P2, C1 is ordered in the chunking log before C2. We have recorded a cyclic dependence, which makes the resulting logs impossible to replay and, therefore, causes deadlock.

![Figure 2.6: Examples of dependences between input events (solid lines) and between chunks (dashed lines).](image)

To avoid this problem, Capo3 does not let a chunk include instructions from both before and after an input event. Instead, before an input event is recorded, the RSM executes the XTC instruction — therefore terminating the current chunk. With this approach, the situation in Figure 2.6a transforms into the one in Figure 2.6b. In this case, there are four chunks and the cyclic dependence has been eliminated. Both input and chunk dependences are satisfied if we replay the chunks in the C11, C21, C12 and C22 order.

Another issue related to the interaction between the two logs is how the replayer can match the input log entries and the chunk log entries generated by the same thread. Fortunately, this is easy, since the RSM
assigns a unique R-Thread ID to each thread (Section 2.3.2). As the logs are generated, they are augmented with the R-Thread ID of the currently-running thread. In particular, as the RSM writes the CMEM buffers to the log, it attaches the current R-Thread ID to the buffer’s data.

2.4 Prototype Characterization

2.4.1 Experimental Setup

We evaluate the QuickRec system by collecting and analyzing both log data and performance measurements for a set of SPLASH-2 benchmarks (Table 2.3). We execute each benchmark to completion, and show results for a default configuration of 4 threads running on 4 cores. In addition, we also assess the scalability of QuickRec by analyzing runs with 1, 2, 4, and 8 threads. For our experiments, we pin each application thread to a particular core. Thus, in the default case, we assign each thread to its own core and, in the 8-threaded case, we assign two threads to each core. We implement Capo3 as a kernel module in Linux 3.0.8.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Input Size</th>
<th># of Instruc. (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes</td>
<td>nboby 8000</td>
<td>3.4</td>
</tr>
<tr>
<td>FFT</td>
<td>-m 22</td>
<td>3.7</td>
</tr>
<tr>
<td>FMM</td>
<td>-m 30000</td>
<td>5.3</td>
</tr>
<tr>
<td>LU</td>
<td>-n 1024</td>
<td>3.0</td>
</tr>
<tr>
<td>LU-NC</td>
<td>-n 1200</td>
<td>4.7</td>
</tr>
<tr>
<td>Ocean</td>
<td>-n 1026</td>
<td>7.5</td>
</tr>
<tr>
<td>Ocean-NC</td>
<td>-e1e-16</td>
<td>2.2</td>
</tr>
<tr>
<td>Radix</td>
<td>-n 1000000</td>
<td>2.3</td>
</tr>
<tr>
<td>Raytrace</td>
<td>teapot.env</td>
<td>0.3</td>
</tr>
<tr>
<td>Water</td>
<td>1000 molecules</td>
<td>5.4</td>
</tr>
</tbody>
</table>

Table 2.3: Characteristics of the benchmarks. The last column shows the total number of instructions executed in the 4-threaded run in billions. Water refers to Water-nsquare.

2.4.2 Log Analysis

In this section, we analyze the size and bandwidth requirements of the logs generated during the recorded execution. In addition, for the chunk log, we perform a detailed characterization. In all cases, we consider
logs without data compression.

**Log Sizes and Bandwidth**

Figure 2.7 shows the *uncompressed* size of the input and chunk logs for each of the benchmarks and for the average case (AVG). For each benchmark, we show data for 1-, 2-, 4-, and 8-threaded runs. The size is given in bytes per million instructions. From the bars, we see that the average log size produced by QuickRec for 4 threads is 1,224 and 1,235 bytes per million instructions for input logs and for chunk logs, respectively. These are small numbers. However, the Ocean-NC and Raytrace benchmarks generate notably larger logs for 4-8 threads. This effect is mainly due to the increased use of synchronization in the benchmarks, which involves frequent calls to the *futex()* system call. As a result, the input log size increases substantially. Also, since Capo3 terminates the running chunk before recording an input event (Section 2.3.6), the chunk log also grows substantially.

![Figure 2.7: Uncompressed log sizes.](image)

The average log sizes that we measure are in line with sizes reported in previous work. For example, the log sizes reported for Cyrus [31], DeLorean [56], Rerun [34], and LReplay [18] are all within approximately 0.5x–2x of ours. We also note that our numbers correspond to a simple, unoptimized RnR implementation, and can easily be improved. As a simple example, consider the log entry for a chunk in QuickRec (Figure 2.4). Of the 128 bits, in most cases, only 80 bits are used for RnR. The remaining bits are mostly used
for debugging and characterization of the hardware. If we eliminated them, we would get the average log sizes labeled REDUCED-AVG in Figure 2.7. Further log size reductions can be attained with improved bit encoding.

Figure 2.8 shows the memory bandwidth requirements of logging. The figure is organized as the previous one and shows bandwidth in KB per second. From the average bars, we see that the bandwidth for 4 threads is 40 KB/s and 43 KB/s for input and chunk logs, respectively. These numbers, when combined, represent only 0.3% of the 24 MB/s bandwidth available in our prototype (Table 2.1). Hence, the effect of logging on bus and memory contention is very small. If we use the 80-bit chunk entries for the log (bars labeled REDUCED-AVG in Figure 2.8), the bandwidth requirements are slightly lower.

Figure 2.8: Memory bandwidth requirements

To reason about the bandwidth requirements of QuickRec’s logging on modern computers, consider the following. A modern multicore computer cycles at a higher frequency than our prototype, but it also has higher memory bandwidth. To understand the impact of these changes, we recompiled and ran our benchmarks on a dual socket Xeon server with 2.6 GHz E5-2670 processors. We measured the elapsed time (and speedup over our prototype) of the 4-threaded applications and scale the bandwidth numbers accordingly. Assuming the 80-bit log entry per chunk, we obtained an average bandwidth consumption across the benchmarks of 17.9 MB/s (and 61.1 MB/s for Ocean-NC, which is bandwidth-intensive). Given that the E5-2670 processor provides a memory bandwidth of up to 6.4 GB/s per core, the logging accounts
for only 0.07% on average (and 0.23% in Ocean-NC) of the available bandwidth of 4 cores. Based on these estimates, we conclude that the bandwidth usage is negligible and will not have a negative impact on the performance of real systems.

If we compress the logs using gzip’s default DEFLATE algorithm, we attain an average compression ratio of 55% for chunk logs and 88% for input logs. Hence, the average 4-threaded benchmark can be recorded for almost three days before filling up a terabyte disk.

Finally, Figure 2.7 and Figure 2.8 also suggest that both the log sizes and the bandwidth requirements scale reasonably as the number of threads increases from 1 to 8.

**Chunk Characterization**

Figure 2.9 shows the average size of the chunks in terms of retired x86 instructions. Figure 2.10 shows the distribution of chunk sizes for 4-threaded runs. On average, the size of a chunk for 4-threaded runs is 39K. However, Figure 2.10 shows that, while many chunks are large (e.g., more than 80% of the chunks in Barnes, LU, and LU-NC are larger than 10,000), there are many chunks with fewer than 1,000 instructions. For three benchmarks, there is a significant fraction of zero-sized chunks, which mostly result from explicitly terminating a chunk unconditionally at input events. This effect can be avoided by changing Capo3 or the hardware.

![Chunk Size vs Instructions per Chunk](image_url)

**Figure 2.9: Average chunk size.**

Figure 2.11 details the chunk termination reasons, using the categories shown in Table 2.2, except that...
exceptions, chunk-size overflows, and TLB invalidations are grouped together in Other. From the figure, we see that the largest contributor to chunk termination is cache line evictions. In the QuickRec hardware, a chunk must be terminated if a line that is evicted from the L2 hits the read set or the write set in the same core. This is because subsequent snoop requests to that line are not delivered to the MRR; they are filtered out by the L2. Techniques to mitigate this behavior will contribute to reducing the number of chunks.

Conflicts due to WAR, RAW, WAW and WAB are the second most prevalent reason of chunk terminations. Another frequent reason is explicit chunk termination with XTC. This termination reason is common when we have more threads than processors (i.e., in the 8-threaded runs). In this case, there are many context switches which use XTC. This reason is also common if the benchmark has numerous input events, such as signals or system calls, which require explicit use of XTC to obtain a total order of events. For example, this is the case for Raytrace and Ocean-NC, which, as shown in Figure 2.10, have a large number of zero-sized chunks.

To deal with instruction reordering and instruction atomicity violations, QuickRec appends RSW and
IAV information to chunk entries. Figure 2.12 displays the fraction of chunks that are associated to non-zero RSW and/or IAV values. The figure reveals that such chunks are common. For 4-threaded runs, an average of 16% of the chunks are RSW or IAV chunks. In fact, both RSW-only and IAV-only chunks are common. One interesting case is that of Radix, where the fraction of IAV chunks is over 40%. The reason is that Radix has a long-running tight loop with several multi-memory-operation instructions. FFT has many RSW-only chunks, which result from executions where loads and stores are interleaved. Overall, RnR systems must be designed to handle these cases.
2.4.3 Performance Measurements

To measure the overhead of QuickRec’s different components, we ran each benchmark in five different configurations. First, *native* is the normal execution with no recording. Second, in *hw-only*, the MRR hardware is enabled and writes chunk data to main memory, but otherwise no other component of the system is enabled. This configuration measures the overhead of the extra memory traffic generated by the MRR. Third, in *input*, the RSM only logs the sources of input non-determinism described in Section 2.3.2 and the MRR is disabled. Fourth, *chunk* augments the *hw-only* configuration by having the RSM dump the CMEM buffers to a file; no input is recorded. Finally, *combined* is a full recording run where both input and chunk data are processed by the RSM. To reduce the OS-induced noise, each configuration is run five times and the results are averaged. Each run executes with four threads.

Figure 2.13 shows the execution time of each configuration normalized to the execution time of *native*. The figure shows that, in most benchmarks, recording both input and chunk logs only incurs a 2–4% overhead. The main exceptions are *Ocean-NC* and *Raytrace*, which suffer an overhead close to 50%. As indicated in Figure 2.7, these two benchmarks perform substantial synchronization, which involves frequent calls to the *futex*() system call and, often, results in putting threads to sleep. On average across all of the benchmarks, the recording overhead is 13%.

Interestingly, the recording overhead is entirely due to the software stack. Indeed, the hardware overhead, as shown in *hw-only*, is negligible. We also see that the software overhead is primarily due to input logging, rather than chunk logging. Overall, future work should focus on optimizing the software stack and, in particular, input logging — specifically, removing the serialization in the recording of input events.

Figure 2.14 shows the processor time (the time processors spend doing useful work for the applications) separated into user and system time. For each benchmark, we show three bars: one for the recorded application itself (App), one for the driver that reads the input log from memory and writes it to disk (Input), and one for the driver that reads the chunking log from the memory and writes it to disk (Chunking). For each benchmark, the bars are normalized to the processor time of the application.

The figure shows that most of the processor time is spent running the application. On average, the drivers add little overhead. Only the two benchmarks with large logs in Figure 2.7 spend noticeable time in the drivers. Finally, most of processor time in these applications is user time.
Figure 2.13: Execution time with each recording configuration for four-threaded executions. The bars are normalized to the execution time of native.

To understand the sources of overhead in QuickRec, Figure 2.15 breaks down the total processor cycles into four categories. First, App time are the cycles spent executing instructions not resulting from Capo3 overhead. Second, Input overhead (working) are the cycles spent in Capo3 code managing the input events. Third, Input overhead (sleeping) are the cycles spent in Capo3 waiting on synchronization in order to enforce a total order of input events. Finally, Chunking overhead are the cycles spent in Capo3 code managing the chunking log. The figure shows the breakdown for different thread counts. As the figure indicates, for 4- and 8-threaded runs, the main overhead of Capo3 is due to enforcing a total order of input events. We are looking into optimizations and/or alternative designs for this component.

Figures 2.16 and 2.17 present detailed breakdowns of the input and chunking overheads, respectively, for different thread counts. In each figure, the overheads are normalized to the overhead of the 1-threaded execution for the given benchmark.

Figure 2.16 divides the overhead of input recording and management into the contributions of system calls, copy to user (CTU), and other events. In each case, the figure separates working and sleeping overheads. The figure shows that the sleeping overhead resulting from serializing the system calls is by far the largest component for 4- and 8-threaded runs. In particular, FFT’s normalized overhead for 4-
Figure 2.14: Total time that the processors spend working on the applications divided into user and system time.

and 8-threaded runs is high. The reason is that FFT has minimal overhead with 1 thread and has many synchronization-induced futex() calls with 4 or more threads.

Figure 2.17 depicts a similar breakdown for the chunk-management overhead. The overhead is divided into execution of XTC instructions (Chunk term), execution of XFC instructions (CBUF flush), allocation of a new CMEM buffer (Buffer allocation), putting a CMEM buffer in the work queue (To workqueue) and Other. The latter is dominated by the overhead of saving and restoring MRR registers in a context switch. We see that Buffer allocation and Other dominate.

2.5 Validation Using Replay

A critical aspect of the design and implementation of a recording system is to validate it with replay. Replaying recorded logs enables full assurance that the recording system captures the correct and complete information. Therefore, in this section we discuss the replayer from the perspective of its validation of QuickRec.

We implemented the replayer using the Pin [53] binary instrumentation framework. We chose this ap-
Figure 2.15: Breakdown of the total processor cycles for different thread counts.

proach for three reasons. First, user-level binary instrumentation is operating-system independent (similar to PinPlay [63]), which enables replay to occur on a machine that is independent from the QuickRec system. Second, Pin operates at speeds faster than existing instruction-set simulators, while maintaining an acceptable level of observability. Third, using Pin, we can extend the replayer by integrating other analysis tools, such as race detectors [9, 73] and debuggers [52].

### 2.5.1 High-Level Implementation Description

To correctly replay a recorded execution, the replayer requires the executed code (binary and libraries, including self-modified code), and the program inputs and shared-memory access interleaving experienced during the recorded execution. Prior to replay, the static code is extracted from the log files. Self-modified code, which is not present in the log files, is re-generated by the replayed execution. Non-deterministic inputs are made deterministic by injecting the appropriate recorded data into the replayed execution at appropriate execution points. For most system calls (e.g., read()), this operation involves emulating the system call, by: (i) injecting the logged data into the program if there is a logged copy_to_user() entry, and (ii) setting the return values as defined in the input log. However, there are a few system calls, such as thread creation and termination, that are re-executed to recreate the proper kernel state.

Chunk ordering is accomplished by counting instructions as they are replayed, and stopping when the counter reaches the logged chunk size. In addition, the replayer enforces the logged chunk order, based on
the recorded timestamps.

**Chunks with Non-Zero RSW or IAV Counts**

To handle the IA memory model correctly, the replayer needs to take into account the values of the RSW and IAV counts. Specifically, to support TSO, the replayer simulates a thread-local store buffer. On a store operation, the replayer writes the address and value of the store to the local store buffer — instead of committing the store to the global memory. On a load operation, the replayer first checks the local store buffer. If the address is not found, it loads the value from the global memory. Then, at the end of the chunk, the replayer drains the stores from the local store buffer, except for a number equal to the RSW count of the chunk, and commits their values to the global memory. The stores remaining in the local store buffer are committed as part of the next chunk.

To handle non-zero IAV counts, the replayer needs to know the number of memory transactions involved in the execution of each instruction. When the replayer finds a chunk whose IAV is non-zero, after executing the chunk, it emulates the execution of the memory transactions of the first instruction after the chunk, one at a time. The replayer stops when the number of memory transactions is equal to the IAV count. The remaining memory transactions of the instruction are emulated at the beginning of the next chunk.

Figure 2.16: Breakdown of the normalized overhead of input recording and management. CTU stands for Copy To User.
2.5.2 Validating the Complete System

Prior to full-system tests, we developed multiple levels of system validation. We began with RTL simulations to validate the MRR hardware without software, while we used Simics simulations to validate Capo3. Next, we integrated Capo3 with QuickRec and developed tests to independently exercise the recording functionalities of input non-determinism and shared-memory interleaving. Last, we tested the complete system with our benchmarks.

When bugs were found during full-system tests, the major challenge was pinpointing their origin. In QuickRec, bugs can originate from either the replayer, the recording hardware, or the recording software; distinguishing between the three is usually non-trivial. In our experiments, the most common type of bug manifestation was a divergence between the memory state or the control flow of the recorded and replayed executions. There are many reasons why a divergence can occur, and being able to pinpoint the root cause of such a divergence is critical.

The most obvious location to check for divergent executions is where non-deterministic input events are logged. This is because, during recording, Capo3 saves the contents of the processor registers at the entry of system calls. Hence, the replayer can compare the state of the processor registers before a system call to the recorded state. This provides a clear detection point of divergence. Moreover, a system call should result in a chunk termination and, therefore, should be the last instruction of the chunk it belongs to. This provides
another divergence check.

Unfortunately, non-deterministic input events are infrequent and, therefore, insufficient to detect the root cause of most divergences — the source of divergence can be thousands of instructions before the system call. Therefore, a more fine-grained mechanism to detect divergences was needed.

For this purpose, we added a branch-tracing module in the FPGA hardware. It collects the history of branches executed — like the Branch Trace Store of today’s IA processors. With this information, the replayer can compare the control flow of the recorded execution with that of the replayed execution. This is a powerful method to detect divergences, since if either the record or replay system has a bug, then the replayed execution typically leads to a different control flow. Also, with branch traces, the detection point of a divergence tends to be close to its source.

**Hardware Instruction Counting Bug**

With branch tracing, we found one particularly noteworthy hardware bug. In the water benchmark, we found that a system call was not aligned with the end of the chunk during replay, indicating a bug in the system. The replayer was encountering a system call two instructions prior to the expected end of the chunk.

At first, the problem appeared to be a control-flow divergence manifesting as different instruction counts between the log and replayed execution. However, the branch traces revealed no control-flow divergence. Further investigation showed that the hardware was miscounting instructions when handling floating-point exceptions. Without a confirmation from the branch traces regarding no control-flow divergence, it would have been very difficult to pinpoint this bug.

**2.6 Related Work**

In terms of the hardware, QuickRec resembles CoreRacer [67] the most. While the chunking and the instruction reordering are handled similarly, the main differences are on the implementation of instruction atomicity violation, and on the integration of input recording and chunking. LReplay [18] extends a multiprocessor system with a pending period-based mechanism for recording thread interleaving, and uses large CAM structures to deal with instruction reordering. LReplay is evaluated using RTL simulation and does not discuss issues related to system software.

All of these hardware-assisted approaches have only been modeled using simulation, and often without
considering the necessary software support. As such, they have generally ignored practical aspects of RnR systems. The QuickRec system is the first work to evaluate RnR across the entire stack using real hardware.

2.7 Lessons Learned

The main lessons we learned from this effort are:

- Clearly, to maximize the chance that RnR is considered for adoption, it is critical to minimize the number of touch points that it requires on current processor hardware. QuickRec demonstrates that chunk-based recording can be implemented with low-enough implementation complexity and few-enough touch points to make it attractive to processor vendors.
- By far the biggest challenge of implementing RnR is dealing with the idiosyncrasies of the specific architecture used, as they fundamentally permeate many aspects of the hardware and software. Examples of idiosyncrasies are the memory consistency model and the CISC nature of the architecture.
- The design of the deterministic replayer must account for the micro-architectural details of the system, if it is to reproduce the execution exactly. This was altogether neglected by prior replay work. In fact, such micro-architectural details substantially increase the replayer’s complexity, in turn impacting the usage models and potentially the ability to create non-proprietary replay tools.
- A new research direction is to investigate replay techniques that reduce or abstract away the complexity mentioned. Such techniques may hinge on commodity hardware, or may require hardware extensions to enable replay software.
- The design of the recording software stack can considerably impact the hardware design, as well as the overall performance. For instance, to properly record kernel-mode instructions (e.g., copy_to_user() calls), we had to make non-trivial changes to the hardware-software interface (Section 2.3.5). Also, the software stack is responsible for practically all of the QuickRec recording overhead.
- The main performance overhead in QuickRec is in the software layer collecting and managing the input logs. A seemingly unimportant issue such as the serialization of input-event processing has become our most obvious bottleneck. Recording input events very efficiently is an area were further work is needed.
- The performance analysis clearly suggests that, with a slightly-improved software stack, RnR can be used in always-on manner, enabling a potentially-large number of new RnR uses. Additional features may need to be added, such as checkpointing and log compression to reduce log file sizes in long-running programs.
Finally, full-system prototyping is required to understand RnR issues related to architecture idiosyncrasies, hardware-software interaction, and true performance bottlenecks.

2.8 Concluding Remarks

RnR of multithreaded programs on multicores has high potential for several important uses: debugging applications, withstanding machine failures, and improving system security. To make RnR systems practical, this work has contributed in three ways.

First, we presented the implementation of QuickRec, the first multicore IA-based prototype for RnR of multithreaded programs. The prototype includes an FPGA instantiation of a Pentium multicore and a Linux-based full software stack.

Second, we described several key implementation aspects in QuickRec. We showed how to efficiently handle x86 instructions that produce multiple memory transactions, and detailed the elaborate hardware-software interface required for a working system.

Third, we evaluated QuickRec and demonstrated that RnR can be provided efficiently in real IA multicore machines. We showed that the rate of memory log generation is insignificant, given today’s bus and memory bandwidths. Furthermore, the recording hardware had negligible performance overhead. However, the software stack induced an average recording overhead of nearly 13%. Such overhead must come down to ensure always-on use of QuickRec.

Based on this work, we suggest focusing future research on several directions. First, to reduce the software stack overhead, it is important to record input events very efficiently — specifically, in a partially-ordered manner. This will reduce recording overhead, and truly enable always-on RnR.

Second, much emphasis should be placed on the replay aspect of RnR. We need approaches that are tolerant of, and abstract away, the micro-architectural details of the recording platform. Otherwise, proprietary details will stifle the development of replay support. We need creative ways of combining hardware and software support for replay.

Finally, we need to develop and demonstrate many uses of the RnR technology that solve real problems of multicore users. The areas of parallel program development tools and security-checking aids seem particularly ripe for development.
Chapter 3

Cyrus: Unintrusive Application-Level Record-Replay for Replay Parallelism

3.1 Introduction

Different RnR schemes attempt to optimize different metrics. Traditionally, hardware-assisted RnR schemes have attempted to minimize log size requirements. Software-only schemes, instead, have focused on minimizing the overhead of recording — in some cases, even at the cost of potentially having to replay multiple times [6, 62]. Very few schemes have focused on maximizing replay speed — most notably DeLorean/Capo [56, 57], DoublePlay [77], and Karma [10]. All three use parallel replay mechanisms for this purpose.

Each of the three previous systems has shortcomings that could limit its practicality. Specifically, DeLorean/Capo uses transactional record and replay hardware, which requires a redesign of current commodity processor hardware. Karma provides whole-system RnR rather than application-level RnR. As indicated above, this is not what users typically need and, in addition, it is hardly portable. In addition, Karma requires augmenting the cache coherence protocol messages — which we want to avoid. Finally, DoublePlay is a software-assisted scheme, which requires modifying and recompiling the application, marking its synchronizations.

This is unfortunate, given that fast replay is a key enabling property for RnR systems. For example, debugging can be more productive if buggy executions can be quickly replayed to the point of the bug. Similarly, intrusion analysis can benefit from extensive on-the-fly analysis of how the attack is taking place. Finally, in fault tolerance, a backup machine has to quickly catch up with a failed one to provide hot replacement.

To attain effective low-overhead RnR, we believe that, in addition to providing fast parallel replay, the system needs to: (i) support application-level RnR, and (ii) rely on unintrusive hardware design. In particular, it should avoid system-level hardware changes such as any changes to the cache coherence protocol. We believe this is fundamental for acceptance of RnR hardware. Since most multiprocessors today use snoopy
cache coherence, we require our design to be compatible with (and not modify) snoopy protocols.

In this work, we make the following contributions:

- We present the first hardware-assisted approach for unintrusive, application-level RnR that explicitly targets high-speed replay. The approach, called *Cyrus*, requires no modification to commodity snoopy cache coherence.
- *Cyrus* introduces the concept of an on-the-fly software *Backend Pass* during recording which, as the log is being generated, consumes it and transforms it. This pass fixes-up the log, which has incomplete information due to our recording requirements of only application-level interactions and no cache coherence protocol changes. In addition, the backend pass exposes a high degree of parallelism for replay. Finally, as the backend pass produces the final log, it can also flexibly trade-off replay parallelism for log size.
- We modified the Linux kernel to control and virtualize a simulated version of the *Cyrus* hardware. Our results show that *Cyrus* adds negligible recording overhead, even with the backend pass. In addition, for 8-processor runs of SPLASH-2, *Cyrus* attains an average replay parallelism of 5 (in terms of the length of the critical instruction path), and a replay speed that is, on average, only about 50% lower than the recording speed.

The rest of this chapter is organized as follows: Section 3.2 discusses background issues and challenges in RnR; Section 3.3 presents *Cyrus*’ architecture; Section 3.4 describes implementation issues; Sections 3.5 and 3.6 evaluate *Cyrus*; Section 3.7 discusses related work; and Section 3.8 concludes the chapter.

### 3.2 Background and Key Challenges

#### 3.2.1 Background on Deterministic RnR

Deterministic Record-Replay (RnR) consists of monitoring the execution of a multithreaded application on a parallel machine, and then exactly reproducing the execution later on. RnR requires recording all the nondeterministic events that occur during the initial execution. They include the inputs to the execution (e.g., return values from system calls) and the order of the inter-thread communications (i.e., the interleaving of the inter-thread data dependences). Then, during replay, the logged inputs are fed back to the execution at the correct times, and the memory accesses are forced to interleave according to the log.

To accomplish application-level RnR, we leverage previous work, Capo [57], which describes how the
OS virtualizes the RnR structures. A Replay Sphere is the single application (or group of applications) that we want to RnR in isolation from the rest of the system. Each sphere has an Input Log and a Memory Log.

The Memory Log collects the order of the data dependences between threads. To collect such orders, we can use a software-only solution that relies on the runtime or operating system. Alternatively, we can use a hardware-assisted scheme that relies on a special hardware module. This approach has the advantage of recording with negligible performance overhead, even for applications with frequent inter-thread data dependences. Hardware-assisted schemes typically use cache coherence transactions to detect inter-thread dependences.

To reduce the amount of state that needs to be collected in the Memory Log, most of the recent proposals of hardware-assisted schemes [10, 34, 56, 57, 66, 81] log the amount of work that a thread does between communications, rather than the communications themselves. Specifically, each entry in the log records the number of consecutive operations executed by a processor between inter-thread dependences. These groups of instructions or memory accesses are known as Chunks. The entry also has information on what other chunks in the log depend on this one. During replay, inter-thread dependences are enforced by executing these chunks in the proper order.

Most of the proposed chunk-based recording schemes encode the execution in a fairly serial form that can take away much of the parallelism that existed in the original execution. Specifically, some schemes log a total order of chunks [56, 57, 66], while others use Scalar Lamport Clocks (SLC) [43] to order chunks [34, 81]. In a basic design with SLCs, when a processor detects an inter-thread data dependence, it terminates its current chunk and assigns to it a scalar clock value (usually called timestamp). The recording mechanism guarantees that if an instruction in chunk C2 depends on an instruction in chunk C1, then C2 receives a timestamp strictly larger than that of C1. This way of using timestamps creates many unnecessary chunk ordering constraints which hide the original parallelism.

To recover lost parallelism, DeLorean uses speculative execution of chunks in parallel [56]. Karma [10], instead, records the chunk ordering as a directed acyclic graph (DAG) of chunks. In the general case, each chunk has multiple predecessor and successor chunks, and parallel replay is possible.

Karma, however, is a whole-system RnR scheme that has been designed around a directory-based cache coherent system. It augments the coherence messages with timestamps and some new fields. It relies on the availability of explicit invalidation acknowledgements in directory protocols.
3.2.2 Key RnR Challenges

Our goal is to develop a useful and easy-to-implement RnR hardware scheme. Such a scheme needs to: (i) support application-level RnR rather than whole-system RnR, (ii) avoid changes to system-level multiprocessor hardware, especially changes to the snoopy cache coherence protocol, and (iii) enable highly-parallel replay.

In this section, we elaborate on the challenges that these three requirements pose. Our general approach matches current proposals \cite{ref1, ref2, ref3, ref4, ref5, ref6}: we use the cache coherence transactions to detect dependences between threads, and each entry in the log records the number of consecutive instructions executed by a processor between dependences.

**Challenge 1: Application-Level RnR**

The main difficulty in performing application-level RnR is that many of the cache coherence transactions observed in the machine are potentially unrelated to the application being recorded; they are due to the OS or to other applications. Consequently, an application-level RnR scheme has to identify these unrelated transactions and prevent them from inhibiting correct replay. This is unlike a whole-system RnR scheme.

Figure 3.1 shows this problem for a three-processor machine. As we run the application, we may observe coherence transactions between processors executing the application being recorded (e.g., transaction (1)). However, we may also observe transactions between the application being recorded and the OS (transaction (2)), between two OS threads (transaction (3)), and even between the application (or OS) and an application that is *not* being recorded (transactions (4) and (5)).

Some of these communication events are unrelated to the application and can make the log inconsistent and cause replay divergence. They result from a variety of causes. One is interaction between OS and application threads, possibly through common buffers, and between OS threads. Another is the presence of hardware prefetchers, which may move unpredictable data and change its coherence state. Another effect is the processor issuing speculative loads, which access unpredictable data. In addition, the presence of context switches adds further uncertainty: a transaction may move data from a cache where the owner thread has been preempted. Should we record it? Finally, the Bloom filters \cite{ref7} used in many RnR schemes to help detect dependences between threads \cite{ref8, ref9, ref10} compound the problem: the events may be false positive.
dependences due to address aliasing in the signature.

**Challenge 2: Unintrusive Hardware**

The second challenge of an RnR scheme is the need to avoid changes to system-level hardware and, in particular, any changes to the cache coherence protocol. This work focuses on snoopy coherent systems because they are the most commonly used approach today. In this environment, we must not augment the coherence protocol with new messages or even new fields in existing messages. This is because the timing of the messages is an integral part of the protocol design and any timing changes will require protocol re-validation.

A key consequence of this requirement is that the RnR scheme must record inter-thread dependences from the dependence source only. To see why, consider Figures 3.2(a)-(c). In these charts, processor P1 initiates a request, which causes P0 to supply a cached line and/or to invalidate it. These are dependences that need to be recorded in the RnR log. However, snoopy protocols provide incomplete information. Specifically, while the requesting processor (P1) includes its ID in the coherence transaction, the processor at the dependence source (P0) does not supply its ID in Figure 3.2(c) because there is no response message; in addition, P0 may or may not provide its ID in Figures 3.2(a)-(b). This is unlike directory-based protocols, where there are explicit request and response messages that include the sender ID.

Hence, in any dependence, only the RnR module at the source processor (P0) knows about it and logs it; we have to assume that the RnR module at the destination processor (P1) is completely unaware of the dependence. Any replay system must be able to reconstruct the execution from a log with only dependences
of the form shown in Figure 3.2(d) instead of bidirectional successor and predecessor information as in Figure 3.2(e).

**Challenge 3: Replay Parallelism**

Parallelism is fundamental to high-speed replay, which in turn will enable new uses of this technology. To expose maximum parallelism, the log must encode the dependences between chunks across threads. Also, for each dependence, the source and destination chunk boundaries should be *as close as possible* to the dependence’s source and destination references, respectively.

### 3.3 Unintrusive App-Level RnR for Replay Parallelism

#### 3.3.1 Main Idea

We propose a new general approach to address the previous challenges and deliver hardware-unintrusive, application-level RnR for replay parallelism. Our approach is called *Cyrus*. We use the mechanisms of Capo [57] to record input logs (Section 3.4.2). As for the memory log, to support application-level RnR, the hardware judiciously avoids logging certain types of interprocessor interactions. Moreover, to keep the cache coherence protocol unmodified, the hardware logs the dependence only on the source processor. The
result of these two constraints is a log with some dependences that still need to be fixed-up or discarded, and with unidirectional dependence information only.

Consequently, as the log is being dumped into memory, an on-the-fly software Backend Pass consumes it and transforms it into the final log. This backend pass performs three actions: (i) fixes-up and discards some of the dependences to correctly implement application-level replay; (ii) transforms the unidirectional dependences into bi-directional ones for ease of replay; and (iii) produces a log that enables a high degree of parallelism during replay. In addition, the backend pass can flexibly produce a log with the desired tradeoff between degree of replay parallelism and log size.

Figure 3.3 shows the system. We consider this backend pass to be a fundamental enabler of a hardware-assisted RnR scheme that is hardware-unintrusive, supports application-level RnR and allows a maximum (and also settable) degree of replay parallelism. In the following, we show how Cyrus addresses each of the challenges.

![Figure 3.3: Overview of the Cyrus system.](image)

### 3.3.2 Application-Level RnR

To understand which interprocessor dependences need to be recorded for application-level RnR, consider the model of Figure 3.4(a). At any given time, a processor may run a process that is being recorded or one that is not. We call such times $M$ and $N$ for monitored and non-monitored, respectively. During the $M$ time, the processor may run application code ($M_{app}$ time) or OS code on behalf of the application ($M_{OS}$ time).

In this environment, there are several types of interactions between a source and a destination processor. The destination processor ($dst$) is the one that initiates a coherence action and receives a response —
e.g., it misses in its cache or sends an invalidation. The source processor (src) is the one that sends the response. Figure 3.4(c) shows the types of interactions possible, together with the corresponding type of dependence involved, the action taken by Cyrus, and a dependence example from Figure 3.4(b). In the next few paragraphs, we describe each of the interactions and how Cyrus handles them.

We start by describing the interactions where the source is a processor running a monitored process (srcM), as shown in the first group of entries in Figure 3.4(c). If the destination is a processor running monitored application code (srcM_{app} \rightarrow dstM_{app} or srcM_{OS} \rightarrow dstM_{app}), this is a correct dependence within the recorded application — the recorded application misses in the cache or sends an invalidation. Therefore, Cyrus records it in the log.

If the destination is running the OS on behalf of a monitored application (srcM_{app} \rightarrow dstM_{OS} or srcM_{OS} \rightarrow dstM_{OS}), two cases are possible. One is that the OS is accessing data that will later be accessed by the monitored application code (i.e., it is effectively prefetching the data); the other case is
that the OS is accessing data that is unrelated to the monitored program and happens to be in the source processor’s cache. In the first case, we must record this correct dependence that is detected early; in the second case, we must discard it. Since Cyrus does not know which case it is, it conservatively records it in the initial log. Later, the backend pass will find which case it is, and either set the destination of the dependence to be the next $M_{app}$ chunk running on the destination processor (an action called “deferring the dependence”), or discard it. In Figure 3.4(c), we call this action “Record & Defer/Discard”.

Finally, if the destination processor is running a non-monitored process ($srcM \rightarrow dstN$), the action pertains to unrelated data, and does not need to be recorded. However, for ease of implementation as we will see, Cyrus records it as in the previous case.

The next row in Figure 3.4(c) corresponds to a context switch where an $M$ process is preempted. After the preemption, data left in the cache may be requested by other processors. To avoid having to log such interactions, in a context switch, Cyrus conservatively records one dependence from this processor to every other processor in the machine. This is called a serialization because it effectively serializes the last chunk of the current processor prior to the context switch before the current chunk of every other processor. Cyrus records such dependences in the initial log and the backend pass will defer or discard them. Specifically, any such dependence will be discarded if no monitored process ever runs on the destination processor.

The final two rows in Figure 3.4(c) correspond to when the source is a non-monitored process ($srcN$). In this case, a $srcN \rightarrow dstM$ interaction can be correct, early or unrelated, while a $srcN \rightarrow dstN$ interaction is unrelated. Cyrus ignores each of these dependences. This behavior is correct since any such dependence is guaranteed to be superseded by one of the serialization dependences described above.

Overall, as shown in Figure 3.4(c), to ensure correct application-level recording, Cyrus only needs to log events when a processor running the application or OS code of a monitored process is: (i) either the source of a dependence (i.e., at the request of another processor, it provides a line from its cache or invalidates a line from its cache) or (ii) suffers a context switch. Still, we need a later pass to fix or discard certain dependences.

### 3.3.3 Unintrusive Recording Hardware

The Cyrus hardware is shown at a high level in Figure 3.5. Each core has a Race Recording Unit (RRU) associated with the cache controller. For simplicity, we show the RRU for a system with a single-level cache.
hierarchy. In this design, the RRU observes the bus transactions, and is also informed of processor requests and cache evictions.

![Diagram](image)

**Figure 3.5**: High-level view of the Cyrus hardware.

To keep the design unintrusive, we require that it does not change the cache coherence protocol in any way — including, for snoopy schemes, not adding new fields to messages. As explained in Section [3.2.2](#), the implication for snoopy schemes is that, when an interprocessor dependence takes place, only the source processor knows about it and can record it.

Consequently, Cyrus operates as follows. When a processor \(P_0\) executing a chunk \(C_{0,i}\) of a monitored process observes a bus transaction to which its cache needs to respond (by invalidating a line and/or providing a line), the RRU hardware is signaled. The RRU terminates \(C_{0,i}\) and (in a naive design) creates a local log entry composed of: \(C_{0,i}\)’s chunk size (CS) in number of instructions, the ID of the processor that initiated the transaction (the dependence’s destination processor), and the current time. Cyrus counts time as the number of bus transactions so far, which is known by and is consistent across all processors. We call such number the Time Stamp (TS). The destination processor is unaware that a dependence has been recorded.

This information is all that Cyrus needs to log, and requires no modification to the coherence protocol. However, to ease the replay, we will need to have bidirectional dependence information as in Figure [3.2(e)](image). Such information is generated from the initial log by the backend pass and is stored in the final log (Section [3.3.5](#)).
3.3.4 Replay Parallelism

With the naive approach described, the log records an inter-thread dependence between the chunks that are running when the coherence action is detected. This approach enables only limited replay parallelism. For example, consider Figure 3.6(a), where processor $P_0$ writes to variable $x$ in chunk $C_{0,i-n}$ and processor $P_1$ reads $x$ in chunk $C_{1,j}$ at time $T_0$. The figure also shows the log entry. Since the coherence action occurs while $P_0$ is executing chunk $C_{0,i}$, the logged entry implies a dependence and a replay order between chunks $C_{0,i}$ and $C_{1,j}$ as in Figure 3.6(b) — even though the source of the dependence is much earlier, and the destination is deep inside the destination chunk. To extract maximum parallelism, we would like the log to represent the execution as in Figure 3.6(c), where processors $P_0$ and $P_1$ overlap their execution as much as possible.

To approach this ideal capability, Cyrus can be designed to use a small Maximum Chunk Size and to track multiple chunks at a time. The idea is for the RRU to keep information for the most recent $N$ completed local chunks. These completed chunks plus the currently-running chunk form the Dependence-Tracking Window, from which dependence sources are tracked. Each of these chunks (except for the oldest one) has
a read and a write signature register (ReadSig and WriteSig), which hash-encode with a Bloom filter [12] the addresses of the lines read or written by the chunk (Figure 3.6(d)). When the local cache responds to an incoming coherence request, the hardware checks the address of the request against the signatures in reverse order, starting with the ones for the currently-running chunk. When one of the signatures matches the address, we know that the corresponding chunk was the source of the dependence, and record it. This allows us to precisely place the source of the dependence in the right chunk. If none of the signatures matches the address, the oldest of the N completed chunks is assumed to source the dependence. If the currently-running chunk is the source of the dependence, it is terminated. In this case, all the chunks are shifted up, the old one is written to the log, and a new one starts. With this support, the log records the example dependence as in Figure 3.6(e), where the source of the arrow is closer to the source access. This enables more replay parallelism. Karma [10] uses this approach for N=1.

Figure 3.6(d) shows other fields of each entry in the dependence-tracking window, which we will discuss later.

Unfortunately, even this enhanced approach has some shortcomings. To have a large dependence-tracking window, N needs to be high, which means that many pairs of costly signatures are needed. The alternative is to increase the chunk size, therefore needing a lower N. In this case, however, the source of the dependence may be far from the end of the source chunk, and the destination of the dependence may be far from the beginning of the destination chunk. In the worst case, the source and destination references are separated by twice the maximum chunk size.

To address this problem, Cyrus introduces the concept of Chunk Clusters. Chunk clusters use the observations that: (i) to reduce the separation between the beginning of the destination chunk and the destination reference, we need small chunks; and (ii) to reduce the separation between the source reference and the end of the source chunk, we need a large dependence-tracking window which, to be cheap, needs large chunks. Hence, in chunk clusters, we use small chunks and combine them to make them appear as large chunks. In practice, reducing the separation in (i) is more important than in (ii). The reason is that any separation in (i) directly slows down the replay execution relative to the recorded execution.

With chunk clusters, we use a small chunk size, but we group multiple consecutive chunks into a cluster for the purpose of tracking dependence sources. The RRU’s dependence-tracking window contains multiple chunk clusters. Each one has a single ReadSig and WriteSig signature pair that contains the addresses
accessed by all the chunks in the cluster. If the address of an incoming coherence transaction matches the signature, then the source of the dependence is assumed to be the last chunk of the cluster.

Figure 3.6(f) shows the case of four chunks per cluster. When a chunk executes and exhausts its maximum size without recording a dependence, its termination time stamp is stored in TS[i] and the next chunk in the cluster starts. Note that for such chunks, Cyrus does not need to store the size explicitly because it is known to be the maximum chunk size. When a dependence is found in the running cluster’s signatures, the running chunk is assumed to be the source; that chunk is terminated, its time stamp and size (CS) are saved, and the cluster is terminated. All the cluster information is shifted up and a new cluster is started. Future dependence sources found in the signatures of any cluster in the RRU dependence-tracking window, are assigned to the last chunk in that cluster.

With this support, Cyrus provides a large dependence tracking window, and at the same time, reduces the distance between the beginning of the destination chunk and the destination reference. This is seen in Figure 3.6(g). The result is more replay parallelism.
Figure 3.7: Example of execution and resulting Cyrus logs. The table in (b) depicts the initial chunk data dumped by the processors, while the other tables show the results of the different backends, encoding the corresponding DAGs. In the tables, dashes indicate entries corresponding to dependencies to the processor itself. These are never used.
3.3.5 Backend Software Pass

The initial log generated by the recorder has unidirectional dependence information only, and contains some dependences that need to be fixed-up or discarded for application-level RnR. To correct these issues, a backend software pass processes the log, creating a final log that is highly amenable to parallel replay. In addition, the backend pass can format the log for different tradeoffs between replay parallelism and log size.

Transforming the Log

Each entry of the initial log contains the following base information for one chunk: the ID of the CPU that executed the chunk and the chunk’s termination time stamp (TS). In addition, if this is the last chunk of a cluster that sourced dependences, the information also includes the chunk size (CS) in number of instructions, and successor vector (SV). The latter has one entry for each of the other processors in the machine. SV entry $i$ is either null or has the TS when the current cluster sourced a dependence to processor $i$. If the cluster sourced multiple dependences to processor $i$, SV[$i$] has the TS of the earliest one — which is the most conservative.

Figures 3.7(a) and (b) show an example execution with 4 processors and the resulting initial log, respectively. Each row in Figure 3.7(b) indicates a chunk dumped by the corresponding processor. In all of the tables in Figure 3.7, TID is the ID of the thread to which the chunk belongs. It is provided by the OS driver that controls the RRU. The hardware itself is oblivious to the notion of threads (Section 3.4.2).

For simplicity, we assume one chunk per cluster and two clusters. In Figure 3.7(a), we can see that, as soon as a processor sources a dependence for a datum accessed in the current chunk, it terminates the chunk. If the dependent datum has not been accessed in the current chunk but in past chunks, the current chunk is not terminated. For example, at TS=200, CPU1 performs “Wr A”. Since this access does not conflict with chunk C01 of CPU0, C01 is not cut and the dependence is assigned to C00, instead.

In Figure 3.7(a), all the dependences are Correct ones except for the one from CPU0 to CPU3, which is an Early or an Unrelated one. In the figure, CPU3 is initially executing the OS on behalf of the monitored process. The OS accesses variable $B$, creating a dependence with processor 0, which terminates its chunk C00. According to Cyrus’s operation, it has to record the dependence, and rely on the backend pass to either defer it or discard it. Since, as shown in the figure, CPU3 later executes chunk C30 of the monitored application, the backend pass sets the destination of the dependence to be C30 — i.e., defers the dependence.
This is required for correctness, as chunk C30 could next silently access variable B. If, instead, CPU3 never executes any chunk of the monitored application, the backend pass discards the dependence.

Similarly, if the OS preempts a monitored thread (i.e., on a context switch), it uses the programming interface of the RRU (Section 3.4.2) to create Serialization dependences with all other processors; they are eventually deferred or discarded by the backend pass.

As the backend pass processes each entry of the initial log, fixing up and discarding dependences, it also records, in each dependence’s destination chunk, which other chunk is the source. Encoding such bidirectional dependence information will enable parallel replay. Hence, it incrementally builds the dependence DAG that captures all the necessary ordering of chunks for a deterministic replay.

To encode the resulting DAG in the final log, we adopt and generalize the representation used by Karma [10], in which, instead of representing dependences as source-destination chunk pairs, we use a token-based representation. Assume a dependence between chunk C1 of processor P1 to chunk C2 of processor P2. To enforce the dependence during the replay, the log will have C1 send a token to P2 after its execution, and C2 wait for a token from P1 before starting. Both source and destination are processor numbers rather than chunk numbers.

Our baseline backend pass algorithm and the resulting transformed log are called MaxPar because they expose maximum replay parallelism obtainable from the initial log. An entry in the MaxPar log contains the following information for a chunk: the IDs of the CPU and thread that executed it, its size, the Successor Token Vector (STV), and the Predecessor Token Vector (PTV). The STV is a bit vector with as many bits as other processors. Bit i is set if a successor of the chunk is in processor i. The PTV is an array of counters with as many entries as the STV. Entry i counts the number of predecessors that the chunk has in processor i. For our example, the resulting MaxPar log and execution DAG are shown in Figure 3.7(c).

With the MaxPar log, replay will involve processors executing in parallel, synchronizing only on dependences — figuratively passing tokens between them. In the following, we outline the MaxPar algorithm and then consider other algorithms.

MaxPar: Algorithm for Maximum Parallelism

In this discussion, we call a chunk open while it still has unresolved successors or predecessors, and resolved otherwise. After a chunk becomes resolved, the backend can write it to the transformed log as soon as all of
the previous chunks of the same processor are written. After writing, we say the chunk is retired.

Figure 3.8 shows the high-level pseudocode of the algorithm. Each processor is represented by a proxy object. A proxy keeps track of its open chunks in a chronologically ordered list. Also, it keeps a data structure (called waitingList) for chunks of other proxies that, according to their Successor Vectors (SV), have unresolved successors in this proxy.

```
AddBatch(batch, proxy):
  for each chunk c in the batch
    for each valid successor processor s in c.sv
      /* call sp the proxy for processor s */
      add c to sp.waitingList[proxy]
  foreach other proxy op in the system:
    foreach chunk c in proxy.waitingList[op]
      find dep
      /* dep is the chunk in proxy that is the successor of c */
      if (dep is not NULL)
        remove c from proxy.waitingList
        mark this dependence as resolved in c
        update c.STV and dep.PTV
    if enough time has passed since last trimming
      Trim()

Trim():
  for each proxy p in the system
    for each chunk c in p
      if c is old enough and all its predecessors are retired
        write c to the transformed log
        remove c from p

MaxPar():
  while (there are batches)
    batch ← next batch
    AddBatch(batch, proxies[batch.cpu])
```

Figure 3.8: High-level description of the MaxPar algorithm.

The algorithm processes chunks in batches of consecutive chunks from the same processor. When a new chunk is added to proxy $P$, its SV is checked, and for each successor, the chunk is added to the waitingList of the proxy for that successor. Next, since a new batch has been added to $P$, chunks in the waitingList of $P$ are checked to see if their dependences can be resolved. To resolve a dependence that was recorded at time $t$, the open chunks of $P$ are binary-searched to find the first chunk whose time stamp is larger than $t$. This
chunk is the destination of the dependence. The dependence is recorded by setting the appropriate entry in the STV of the source chunk and incrementing the corresponding entry in the PTV of the destination chunk. As dependences are resolved, periodically, a trimming pass is run to retire the resolved chunks from the proxies by writing them to the transformed log. Before writing a chunk to the log, MaxPar tries to merge it with the previous chunk of the same processor, if that does not reduce the recorded parallelism. Specifically, assume that the previous chunk is $C_0$ and the current chunk is $C_1$. If $C_0$ has no successors (other than $C_1$) and $C_1$ has no predecessors (other than $C_0$), merging $C_0$ and $C_1$ will not change the recorded parallelism.

There are some details that are not shown in Figure 3.8. One difficulty is how to tell whether all the predecessors of a given chunk have been seen and resolved. Processors dump their chunk data independently and in batches (not one by one) and it is quite possible that when a chunk is dumped, some of its predecessors are still in their respective processors.

The solution here is to make sure chunk data do not indefinitely reside in processor buffers and will be dumped if they have been around for a preset amount of time, called Maximum Silence Period (or MSP) — e.g., 100000 timestamp units. Consider chunk $C_i$ of proxy $P_i$. With the above guarantee, which we call the Bounded Silence guarantee, if the maximum timestamp of all the chunks dumped so far is larger than $C_i.ts + MSP$, then we know for sure that all the predecessors of $C_i$ have also been dumped and their dependencies have been resolved (please recall that the predecessors of $C_i$ have smaller timestamps than $C_i$ itself). At this time, we can consider $C_i$ to be old enough (Figure 3.8) to be retired.

Another important question concerns Early or Unrelated dependencies (See the table in Figure 3.4). Assume $C_i$ records $P_j$ as a successor but, since $P_j$ is not running any monitored threads, it will not dump any chunks for a long time (or maybe forever). How should the recorded dependence be resolved?

To handle this case, again, we use the Bounded Silence guarantee. If $C_i$ recorded the dependence at time $t$ and $P_j$ dumps no chunks before time $t + MSP$, then it is guaranteed that no monitored chunk existed on $P_j$ at time $t$. Hence, it suffices to attribute the dependence to the next dumped chunk of $P_j$, or it can be safely discarded if $P_j$ never dumps. The token-based representation enables an efficient implementation in this case. Each proxy, $P_j$ in this example, has a vector of counters which count the number of Early dependences from other processors ($P_i$ in this case). Let us call this vector the Early Token Vector. At time $C_i.ts + MSP$, $C_i$ can consider its dependence resolved and send an Early token to $P_j$ by incrementing entry $i$ of the Early Token Vector in $P_j$. When the next chunk of $P_j$ is dumped, the counters in the Early
Token Vector are added to the PTV of that chunk and then they are reset.

The MSP-based techniques described above imply that, at time $C_i + ts + MSP$, $C_i$ is old enough (term used in Figure 3.8) to be retired, since all of its predecessors and successors are guaranteed to have been resolved by that time.

**Trading-off Replay Parallelism for Log Size**

As the backend pass generates the final log, it can transform it in ways that affect the size of the log and the potential for parallelism in its replay. This provides substantial flexibility (Figure 3.9). For example, when RnR is used for on-line intrusion analysis or fault tolerance, it typically requires high-speed replay. In this case, the log format should be such that it enables highly-parallel replay. Such format is MaxPar. On the other hand, when RnR is used for (off-line) software debugging, replay speed is less important. Hence, we likely prefer a format that needs less log space at the expense of offering less parallelism. Such format is called Stitched. This format is also suitable for intrusion analysis or fault tolerance when the application is I/O- or memory-intensive. This is because, in such scenarios, replay is typically faster than recording. Finally, when small log size is paramount, even at the expense of replay speed, the Serial or StSerial formats should be used. In the following, we discuss these formats.

![Figure 3.9: Flexibility of the backend pass.](image-url)
**Stitched: Reduced Parallelism.** The Stitched format uses less log space than MaxPar, but it offers less parallelism for replay. Compared to MaxPar, Stitched merges consecutive chunks of an application thread into a *Stitched* chunk as long as this process does not introduce cycles in the graph.

Accurately detecting cycles on-the-fly can be computationally intensive. There are conservative techniques that can be used instead. One such technique involves using Lamport Scalar Clocks. Specifically, each chunk is assigned a clock that should be strictly larger of those of its predecessors. While merging a sequence of consecutive chunks into a stitched chunk \( SC \), the algorithm watches the clock values of the all the predecessors of the chunks in \( SC \). As long as all of these predecessors have clock values not larger than that of the first chunk in \( SC \), no cycle can be created and we can safely merge the chunks. If, however, one of the predecessors of the next chunk to stitch violates this condition, we stop merging and start a new chunk sequence.

Figure 3.7(d) shows the Stitched execution DAG and log for the example in Figure 3.7(a). Compared to the MaxPar algorithm (Figure 3.7(c)), we have combined chunks into bigger chunks, hence reducing the number of log entries but also decreasing the parallelism of the DAG available to the replayer.

**Serial: Sequential Replay.** When having a very small log is very important, even at the expense of any parallelism in the replay, we use the Serial format. In this case, we create a total order of chunks. This format is generated with a simple topological sort on the dependence DAG. It can be generated either on-the-fly in the backend or off-line after the MaxPar log has been created. Each serial log entry only needs to contain the thread ID and the size of a chunk — the rest of the information is unnecessary.

Figure 3.7(e) shows the Serial execution DAG and log for the example in Figure 3.7(a). Compared to the MaxPar algorithm (Figure 3.7(c)), we have created a total order of chunks, disabling any parallel replay, but substantially reducing the log size.

**StSerial: Stitched Sequential Log.** Finally, we can reduce the log size even more if we apply the Serial algorithm to a DAG generated by Stitcher. The result is called StSerial. Compared to MaxPar, we reduce both the number of log entries and the size of each of them. The replay is also serial. Figure 3.7(f) shows the StSerial execution DAG and log for the example in Figure 3.7(a).
**Advanced Uses.** Other flexible uses of the backend pass are possible. One use is for the backend to dynamically change the format of the log at different phases of a program’s execution. This scenario may be useful when RnR is used in online-replay scenarios (e.g., fault tolerance) where a secondary server follows the execution of a primary one. To reduce the bandwidth required to transfer the log from the primary to the secondary server, the backend may usually use the Stitched format. However, in sections of the program when fast replay is needed (perhaps because the execution becomes compute intensive), the backend may switch to MaxPar.

Once can also think of a transformation to reduce the number of processors in the log. The transformation involves combining the entries from two or more processors into one. In practice, this transformation is unlikely to be useful since (i) it does not change the number of log entries and just reduces the size of PTV and STV, and (ii) the replay can already be done with fewer processors than used for recording, even with an unmodified log; we only need that a replay processor execute chunks from multiple recording ones.

### 3.4 Implementation Issues

#### 3.4.1 Race Recording Unit (RRU) Design

The Cyrus hardware consists of a *Race Recording Unit* (RRU) associated with the cache controller of each processor (Figure 3.5). When a processor is executing a monitored process, if its cache observes a bus transaction that induces a dependence with data previously accessed by the processor, the RRU records that dependence.

Figure 3.10 shows the hardware inside the RRU. It has four components: Tracked Chunk-Cluster Buffer, Chunk-Cluster Buffer, Time Counter, and Eviction Signature. The Time Counter is the global clock, obtained by counting the number of coherence transactions on the bus. It has the same value in all the cores.

The Tracked Chunk-Cluster Buffer (TCCB) implements the dependence-tracking window described in Section 3.3.4. It contains information about several chunk clusters: the currently-running one (*Running* in Figure 3.10) and the N most recently-completed ones. Of these, the earliest one is called *Last Tracked* in Figure 3.10. All chunk clusters in the TCCB except for the Last Tracked one have read and write signature registers. These registers hash-encode and accumulate with a Bloom filter [12] the addresses of all the lines read or written by all the chunks in the corresponding chunk cluster.
To understand how the TCCB works, assume first that there is no cache overflow; we consider cache evictions later. When a request on the bus *hits in a cache*, the cache’s RRU checks the requested address against the signature registers in its TCCB — a bus write is checked against read and write signatures, while a bus read only against write signatures. The checks are performed in order, starting with the signatures of the Running cluster and proceeding to older clusters. The goal is to find which cluster is the latest source of the dependence.

If there is a hit in the signatures of the Running cluster, the current time stamp is saved in the cluster’s Successor Vector (SV) entry for the requesting processor. Moreover, the cluster terminates and the current chunk size is saved in the cluster’s CS field. In addition, the whole TCCB is shifted upward, pushing the contents of the Last-Tracked cluster into the Chunk-Cluster Buffer and a new Running chunk cluster begins.

If, instead, there is a hit in the signatures of an older cluster in the TCCB, we save the current time stamp in that cluster’s SV entry for the requesting processor. Finally, if instead, the request does not hit in any signature, we conservatively assume that the Last Tracked cluster is the source of the dependence. In this case, we save the current time stamp in that cluster’s SV entry for the requesting processor. In all cases, if the corresponding SV entry is already set to a smaller timestamp, we do not update it.

The case when processor P0 writes a variable, then P1 reads it and then P2 reads it, correctly triggers the logging of a dependence P0→P2 (in addition to P0→P1). The reason is that, although the P2 read does not induce any coherence operation on P0’s cache, P0’s cache hits and, as a result, P0’s write signatures are checked. If the P0 write occurred during its Last Tracked cluster, there is no signature, but Cyrus still records a dependence by default. This would be conservative (although correct) if P0 had only read, not written. Fortunately, recording conservative dependences so far in the past is not expected to hurt replay.
parallelism.

If the current chunk in the Running cluster reaches its maximum size without sourcing a dependence, it terminates, saving the current time stamp in the corresponding TS field of the Running cluster. Then, a new chunk starts. If all the chunks of the Running cluster have been exhausted, the cluster terminates, and the whole TCCB is shifted upward. As information on an old cluster is displaced from the tail of the TCCB, it is dumped into the Chunk Cluster Buffer (Figure 3.10). When the Chunk Cluster Buffer is about to fill up, its contents are appended to the tail of an in-memory buffer provided by the operating system (Section 3.4.2).

**Eviction Signature**

Caches suffer line evictions. In the design presented, when a cache evicts a line, its RRU loses the ability to record inter-processor dependences on data from that line. Indeed, future bus transactions on that line would not find the data in the cache and, therefore would not trigger checks in the dependence-tracking window.

To eliminate this problem, when a clean or dirty line is evicted from a cache, Cyrus hash-encodes and accumulates its address into the RRU’s Eviction Signature (ES) (Figure 3.10). Then, when a transaction is observed on the bus, Cyrus checks if the address hits in the cache or in the ES. If it hits in at least one, Cyrus proceeds with checking the chunk clusters in the TCCB.

The ES should be regularly cleared to avoid collecting many addresses that could cause address aliasing. Fortunately, every time that the OS preempts a thread that is being monitored, the local RRU records a Serialization dependence with all other processors in the system (Section 3.3.2). At that point, the ES can be cleared. Also if, at any time, the ES contains too many addresses, Cyrus simply terminates the current chunk, records a dependence from it to all the other processors, and clears the ES.

### 3.4.2 OS Design for RnR

Figure 3.11 shows the overall architecture of our RnR system, where the dashed boxes indicate the Cyrus extensions. We base our design on that of Capo [57]. The Replay Sphere Manager (RSM) is the basic kernel module that controls RnR. We organize the OS extensions according to the sources of non-determinism. Hence, we have two components: one for input non-determinism and one for memory-interleaving non-determinism.
We use a driver program to launch a RnR sphere to perform record or replay. In record mode, the RSM generates the input log, while the RRUs generate the memory-interleaving log. The data transfers proceed as shown with numbers in the figure. As the initial memory interleaving log is generated, the backend pass runs on a dedicated processor and transforms it. In replay mode, the driver reads the input and memory interleaving logs and passes them to the RSM, which consumes them. In addition, the RSM uses performance counters to detect chunk termination, as we will see.

**Input Non-Determinism Module**

This module is similar to Capo’s [57]. There are four different sources of input non-determinism that Cyrus handles: system calls, data copied to/from the user address space, signals, and non-deterministic processor instructions. Unlike Capo, which uses `ptrace`, we have implemented this component as a Linux kernel module to improve the performance and make it easier to integrate with the memory-interleaving module. Since this module uses per-thread data structures, it is easy to support multiple replay spheres simultaneously.
Memory-Interleaving Non-Determinism Module

Using the RRU at Record Time. As the RRU generates the log, it dumps it into an OS-allocated memory buffer called `cmem`. The RRU offers a minimal interface for the OS to manage and virtualize the hardware during recording. This interface contains: (i) a pointer to `cmem` (`cmem_ptr`), and (ii) a threshold register that indicates the point at which `cmem` is about to overflow (`cmem_th`). When `cmem_th` is reached, an interrupt is triggered, and a new `cmem` is allocated.

The OS manages the per-thread `cmem` areas and virtualizes these hardware registers, so that different threads can use the hardware without mixing up their data. In particular, this involves making sure that a valid `cmem_ptr` is configured before recording begins, allocating a fresh `cmem` when the previous one is full, and ensuring that, on a context switch, all the recorded data is dumped into the `cmem` and a Serialization dependence is recorded. This is done by writing to a RRU-specific control register. Also, the OS appends to each `cmem` buffer the ID of the thread to which the chunks in the buffer belong. Thus, the RRU itself does not need to know about threads.

Enforcing the Recorded Interleavings during Replay. The OS is able to recreate the recorded interleavings by allowing each chunk to start its execution only after all of its predecessors have executed. For this, it uses mechanisms to detect chunk termination and to synchronize predecessor/successor chunks.

To detect chunk termination, Cyrus uses performance counters similar to those available in commodity processors. The replaying thread configures the counter so that an interrupt is triggered when the number of instructions executed equals the needed chunk size. Cyrus assumes synchronous and precise interrupts for this, i.e., the interrupt is generated just before the first instruction of the next chunk is executed.

To synchronize predecessor/successor chunks, Cyrus uses a software solution. When a chunk finishes, it should send tokens to its successors, and before the next chunk starts, it should wait for enough tokens from its predecessors. Cyrus implements this in the RSM (i.e., in the OS and without modifying the application code) using software semaphores. There is a semaphore for each \((P_i, P_j)\) pair of different record-time processors. This semaphore represents tokens sent from \(P_i\) to \(P_j\). After a chunk terminates, the OS first sends tokens to the appropriate semaphores for its successors. It then reads the next chunk from the memory-interleaving log and for each recorded predecessor, it uses the appropriate semaphore to wait until enough tokens are received from that predecessor.
### Processor and Memory System Parameters

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Chip multiprocessor</strong></td>
<td>Bus-based with snoopy MESI protocol 8 proc. for application; 1 for backend</td>
</tr>
<tr>
<td><strong>Processor</strong></td>
<td>Single issue, x86 ISA 2GHz clock</td>
</tr>
<tr>
<td><strong>L1 Cache</strong></td>
<td>64KB size, 64B line, 4-way assoc. 1 cycle hit, 2-bit LRU replacement</td>
</tr>
<tr>
<td><strong>L1 Cache Miss Latency</strong></td>
<td>10-cycle round-trip to another L1 100-cycle round-trip to memory</td>
</tr>
</tbody>
</table>

### Cyrus Parameters

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Read &amp; write signature</strong></td>
<td>4×512bits &amp; 4×256bits H3 Bloom filter</td>
</tr>
<tr>
<td><strong>Eviction signature</strong></td>
<td>4×512bits H3 Bloom filter</td>
</tr>
<tr>
<td><strong># Tracked chunk-clusters</strong></td>
<td>2</td>
</tr>
<tr>
<td><strong># Chunks per chunk cluster</strong></td>
<td>16</td>
</tr>
<tr>
<td><strong>Maximum chunk size</strong></td>
<td>4K instructions</td>
</tr>
<tr>
<td><strong>Chunk Cluster Buffer (CCB)</strong></td>
<td>8 entries</td>
</tr>
</tbody>
</table>

Table 3.1: Parameters of the simulated hardware.

### 3.5 Evaluation Setup

For our evaluation, we augmented the Linux 3.0.8 kernel with a Replay Sphere Manager (RSM). The OS drives and virtualizes the Cyrus architecture modeled with the Simics [54] full-system simulator. The OS changes include the input non-determinism module and the memory-interleaving non-determinism module that manage the two logs.

We use Simics to model an x86-based chip multiprocessor with a single level of private caches that are kept coherent using a snoopy-based MESI cache coherence protocol. Table 3.1 shows the parameters of the architecture. Unless explicitly specified, we perform the parallel record and replay runs with our applications running on 8 processors. The backend pass uses one additional processor. The baseline RRU configuration uses chunks with at most 4K instructions. It has 2 tracked chunk clusters (N = 1), and hence we only need one pair of signatures per RRU. We use 16 chunks per cluster. We execute 10 applications from the SPLASH-2 suite, which we run from beginning to end.
3.6 Evaluation

3.6.1 Recording & Backend Overhead

We first examine the initial log size and whether it can become a bottleneck for Cyrus. Figure 3.12 shows the growth in the rate of the initial log generation as the number of processors increases. This is a temporary log and, hence, it is not compressed. The time unit in this figure is 1K cycles of total execution time (i.e., 0.5 $\mu$sec assuming a 2GHz clock). On a system with 8 processors, this means 8K-cycles worth of instruction execution. As seen in the figure, on average, the logging rate grows about linearly with the number of processors. However, a simple calculation shows that even with 8 processors, the average log generation rate is less than 29 MByte/sec. This is far less than the bandwidth of the system bus in current machines (which is typically on the order of several GByte/sec). For this reason, it is not likely that the initial log generation can become a bottleneck.

![Figure 3.12: Initial log size for different numbers of processors, shown in terms of the number of bits generated per 1K cycles of total execution time.](image)

Figure 3.13 examines the overhead of recording with and without the backend. The figure compares the execution time of the benchmarks when the Cyrus hardware is not enabled (NoMemLog), when Cyrus records the memory-interleaving log (MemLog), and when, in addition, the backend pass runs (MemLog+Backend). In all cases, the RSM is recording the input non-determinism log. For each benchmark, the bars are normalized to NoMemLog. The applications execute with 8 processors.
The figure shows that the overhead of recording the memory-interleaving log, either with or without the backend pass, is negligible. The backend pass induces little overhead because it uses a dedicated processor. While this fact increases the system cost, it allows Cyrus’ RnR to be non-intrusive to the hardware.

### 3.6.2 Comparing Different Backend Pass Algorithms

We now compare the different Cyrus’ backend pass algorithms. We compare the available replay parallelism and the log size of the Serial, StSerial, Stitched, and MaxPar formats (Figures 3.14 and 3.15). To estimate the available replay parallelism, we use the Normalized Inverse Critical Path Length (NICPL) of the dependence graph in the log. To measure the NICPL of a benchmark, we start by computing the length of the longest chain of dependences (in terms of number of instructions) in the log. This is the critical path length. Then, we divide the critical path length obtained with a fully-serial log like Serial or StSerial (which is the number of instructions in the benchmark) by the critical path length obtained with a given log. The result is the NICPL of the log. Thus, a higher NICPL value indicates more parallelism in the recorded dependence graph.

Figure 3.14 compares the NICPL values for the Serial, StSerial, Stitched, and MaxPar formats. We can see that, on average, MaxPar and Stitched provide a replay parallelism of 5 and 3, respectively. Most of the applications can benefit considerably from MaxPar and, to a lesser extent, from Stitched.
Figure 3.15 shows the resulting size of the logs. We compressed the logs with bzip2 and report the number of bits used per 1K instructions. We see that, on average, MaxPar and Stitched generate about 2 bits/Kinstruction, while Serial and StSerial generate 1 bit/Kinstruction. Stitched is not capable of considerably reducing the log size over MaxPar’s because, as mentioned in Section 3.3.5, MaxPar already merges many of the recorded chunks while retaining maximum parallelism. On the other hand, StSerial is only slightly more space efficient than Serial. Finally, the figure shows that water_spatial produces large log files. This is because it synchronizes frequently, which creates many small chunks.

Overall, comparing MaxPar to Serial, we conclude that, with a 2x bigger log, MaxPar delivers a 5x higher parallelism. This is likely to be a very good tradeoff in some RnR applications. On the other hand, Stitched is not a desirable design point. With a 2x bigger log than Serial, it only delivers a 3x higher parallelism. StSerial is only slightly better than Serial.

3.6.3 Replay Execution Time

We now compare the replay execution time of the benchmarks under a variety of scenarios. We start with the MaxPar log with different chunk sizes. Figure 3.16 shows the replay execution time for maximum chunk sizes equal to 1K, 4K, 16K, and 64K instructions. In all cases, there are 2 chunk clusters and 64K instructions per cluster. Thus, there are 64, 16, 4, and 1 chunks per cluster, respectively. The plot is
Figure 3.15: Log size in bits per 1K instructions.

normalized to the execution time of recording with 64K-instruction chunks (the recording time for all the other scenarios is practically the same). We can see that, in general, replay execution time is comparable to recording time, even for these communication-intensive benchmarks. On average, with 1K chunks, replay takes only 50% longer than recording, while with 4K chunks, it takes only 60% longer. As we increase the chunk size, replay time increases. This is largely because there is less replay parallelism with big chunks.

Figure 3.16: Replay execution time with the MaxPar log for different chunk sizes.

Figure 3.17 shows how the logs of the different backends (MaxPar, Stitched, StSerial, and Serial) affect
the replay execution time. For this experiment, we use the baseline RRU configuration of Section 3.5.

As usual, the figure shows the replay times normalized to the recording execution time. As expected, the less-parallel logs cause an increase in the replay execution time. On average, we see that with MaxPar and Stitched, it takes about 60% and 100% longer, respectively, to replay than to record. Replaying in StSerial and Serial takes, on average, about 7 and 12 times longer, respectively, than recording. The relative speeds of MaxPar, Stitched, and StSerial largely match the parallelism numbers provided by the NICPLs in Figure 3.14. Serial, however, is much slower. The reason is that, in the current implementation, Serial does not try to merge chunks before writing them to the log, since this may cause replay deadlocks (we omit the discussion of why this is the case in the interest of space). Hence, it has a high overhead passing tokens, which is done with semaphores.

![Figure 3.17: Replay execution time with logs from different backends for a 4K chunk size.](image)

An important feature of application-level RnR is the ability to replay on machines that have different architectures and, in particular, different processor counts than the recording machine. In Figure 3.18 we show the effect of using fewer processors to replay than were used to record. The recording run used 8 processors, while the replay executes a MaxPar log on 8, 4, 2 or 1 processors. In each case, the 8 threads of the application have to be multiplexed over the available number of processors and synchronize by passing tokens around. The figure compares the execution times normalized to the recording time. As shown in the figure, the replay becomes progressively slower, but not exceedingly so. The amount of slowdown is a function of the number of replay processors as well as the parallelism that existed in the original execution.
and was captured in the MaxPar log.

![Graph showing normalized replay time](image)

Figure 3.18: Replay execution time with a lower processor count than during recording.

Finally, Figure 3.19 breaks down the execution time of replay with the MaxPar log. For each benchmark, the bars are normalized to 100 and broken down into: time spent executing user mode instructions (user), time when the OS is executing on behalf of the application, such as servicing system calls (kernel), overhead associated with handling the input log (input log overhead), overhead associated with handling the memory interleaving chunk log (chunk log overhead), time spent waiting for tokens from predecessor chunks (wait for pred), and time that could not be classified as one of the above (other). The latter is mostly application-level load imbalance — e.g., when some application threads are waiting on a barrier while other threads have not yet arrived at that barrier.

The figure shows that the benchmarks exhibit very different behaviors. However, if we focus on the overheads — i.e., the categories other than user and kernel — we see that other and wait for pred are often dominant. The other category typically implies that there is load imbalance, and little can be done. For example, FFT has a long initialization phase that causes load imbalance. The wait for pred category appears in many benchmarks. They suffer considerable slowdowns just because of waiting for tokens. This suggests focusing on recording more parallelism, possibly with more aggressive techniques. Also, utilizing special hardware support (rather than using semaphores and an all-software solution) for token passing during replay may significantly reduce this overhead for some applications.
3.6.4 Dependence-Tracking Window Analysis

In this section, we compare different organizations of the dependence-tracking window, using replay parallelism (i.e., NICPL) as our metric. Figure 3.20 shows the NICPLs for different designs, represented as $I \times J \times K$, where $I$ is the number of chunk clusters (including the last one, with no signatures), $J$ is chunk size, and $K$ is the number of chunks per chunk cluster. Recall that the tracked window size equals $I \times J \times K$.

In the figure, higher bars are better. The figure is normalized to the NICPL of $2 \times 64K \times 1$, which
can track a window of 128K. If we reduce the chunk size to 4K (second bar), we increase the NICPL even though the window is now only 8K. This design is better because small chunks improve parallelism. If we increase the number of clusters to 3 while keeping the chunk size to 4K (third bar), we improve NICPL significantly because the tracked window increases to 12K, but at the cost of an extra set of signatures. Finally, if we keep the clusters to 2 and the chunk size to 4K but use 16 chunks per cluster (last bar), we have nearly the same NICPL. We have a less precise tracking with only 2 clusters, but have a larger tracked window size. Moreover, we need no extra signatures. This is the most competitive design and we use it as default.

3.7 Related Work

Most software-only approaches are either inherently designed for uniprocessor executions or suffer significant slowdown when applied to multiprocessor executions. DoublePlay \[77\] made efforts to make replay on commodity multiprocessors more efficient. It timeslices multiple threads on one processor and then runs multiple time intervals on separate processors. Hence, it only needs to record the order in which threads in each time interval are timesliced on the corresponding processor. This technique eases logging by only requiring the logger to record the order in which the time slices are executed within a time interval. However, DoublePlay uses an additional execution to create checkpoints off which multiple time intervals can be run in parallel. It also needs to use modified binaries (in particular, a modified \texttt{libc}) for efficient execution.

FDR \[85\] and RTR \[86\] are among the very first race recording techniques proposed. They record dependences between pairs of instructions and, thus, can record parallel dependence graphs. However, they are full-system techniques and rely on modified directory protocols. Also, recording dependences between pairs of instructions can produce large logs and increase associated overhead. To reduce this overhead, chunk-based techniques \[34\], \[66\], \[67\], \[18\], \[81\] have been proposed, but they are not designed for parallel replay and require changes to the coherence protocol.

DeLorean \[56\] and Capo \[57\] are chunk-based techniques that they use speculative multithreading hardware to achieve replay parallelism. Strata \[58\] requires that all the processors agree to start a new stratum (or logging epoch) at regular intervals. This is done by augmenting the messages with a Log Stratum bit, which can be set by the processor initiating the miss or by a processor that provides the data. Strata uses a recording approach that requires that all processors record an entry in their logs at the same time, which does
not scale well with the processor count. Each stratum can be replayed in parallel but all processors should sync up at stratum boundaries. Also, it is not clear if the proposed design, due to its global natures, can be used in application-level RnR where multiple simultaneous recording sessions share the same hardware.

Karma [10] is the first chunk-based RnR technique that explicitly targets replay parallelism without relying on speculative hardware. It is a whole-system (rather than application-level) RnR scheme for directory protocols. It records bidirectional dependences between source and destination chunks and, hence, makes some modifications to the cache coherence messages. The design allows two chunks to be tracked, an idea we build on in this work. Karma allows the chunks to grow beyond conflicts, similar to the Stitched logs presented in this work. The paper reports replay speeds within 19%-28% of vanilla runs (i.e., without RnR). The authors make, however, several simplifying assumptions about the mechanisms used for recording non-deterministic input events, and for handling memory logs, which need care in a realistic, OS-aware implementation of RnR. It is also difficult to extrapolate their results to an application-level scheme like Cyrus. It is also unclear how their replay mechanism can be extended to application-only replay, and to cases where the recording and replaying machines have different numbers of processors.

BugNet [59] records user processes by storing the result of load instructions in a hardware-based dictionary. This is enough to handle both input and memory-interleaving non-determinism and allows each thread to be replayed independently. However, BugNet still needs a solution to record inter-thread dependences, for which it uses FDR [85]. Lee et al. [49, 48] augment this technique by using offline symbolic analysis to reconstruct the inter-thread dependences. This technique is mostly suitable for debugging, since the analysis is, in general, a slow process.

### 3.8 Concluding Remarks

While it is conceivable that a single-threaded backend can become a bottleneck as the number of processors increases, we did not find it to be so in our experiments (as evidenced by Figure 3.13). Also, when such a backend becomes a bottleneck, we can parallelize it and allocate more than one processor to it. This should only be necessary for considerably large systems and workloads that actively use many processors and generate many chunks. It should be emphasized that compute- and sharing-intensive applications such as SPLASH2 programs represent the worst case for Cyrus. Less demanding system workloads, such as databases and web servers, that create smaller memory logs, are considerably easier to handle.
Although we motivated Cyrus using bus-based snoopy systems, it can be easily adapted to other coherence schemes. Cyrus uses source-only recording, which is crucial for parallel application-level RnR. Moreover, it uses the number of bus transactions as the time source. The same time source can be used in any coherence scheme where all the processors see all of the requests in the same order (e.g., address-broadcast tree of Sun’s Starfire [16] or the ring-based design of Intel’s SandyBridge processors [87]). Alternatively, in many modern CMPs, there are chip-level-consistent clock sources that can be used as the time stamp (e.g., the uncore clock that synchronizes the on-die interconnect of recent Intel systems).

For directory-based designs, where such time sources are not available, the local time stamp of the requesting processor can be piggybacked on the request message. Then, the sourcing processors can save this time stamp in their successor vectors — hence, implementing source-only recording. This requires some changes in the format of the coherence requests. Fortunately, compared to bus-based designs, such changes are relatively easy to make in directory-based systems.
Chapter 4

RelaxReplay: Record and Replay for Relaxed-Consistency Multiprocessors

4.1 Introduction

The majority of current proposals for hardware-assisted MRR require that the recorded execution obeys Sequential Consistency (SC) [44]. Under SC, memory-access instructions execute in program order, which substantially simplifies what events need to be logged and when. Unfortunately, commercial machines almost universally use more relaxed memory consistency models, allowing loads and stores to reorder. Recording such execution is especially challenging.

There have been a few proposals for MRR under non-SC models. All but one of them require the Total Store Ordering (TSO) memory model [10, 18, 34, 48, 65, 67, 86], which only allows loads to bypass stores. Such proposals either log which stores are bypassed [65, 67], or log the values read by the bypassing loads [10, 18, 34, 86], or use off-line analysis to identify the actual order that occurred [48]. The other proposal, called Rainbow [70], focuses on detecting SC violations as they happen, and recording enough information to replay them. However, this scheme requires a coherence protocol that is centralized and that needs substantial hardware changes. Moreover, the operation of the scheme’s major components is not clearly described in the paper. All these schemes are discussed in detail in Section 4.6. Overall, the long-standing problem of finding a general MRR solution that works for any relaxed-consistency model (such as that of ARM [8], Power [69] or Tile [76] processors) is still open.

This work contributes with the first complete solution for hardware-assisted MRR that works for any relaxed-consistency model of current processors. With the scheme, called RelaxReplay, we can build an RnR system that works for any relaxed-consistency model and any cache coherence protocol. RelaxReplay’s key innovation is a new approach to capture memory access reordering. Specifically, each memory instruction goes through a post-completion in-order counting step that detects any reordering, and efficiently records it in the log. We present two designs, called RelaxReplay_Base and RelaxReplay_Opt, with different emphases.
Several salient characteristics of the RelaxReplay mechanism to capture memory access reordering are:

- It only relies on the write atomicity property of coherence protocols, and not on knowing the detailed specifications of the particular relaxed-consistency model. Such specifications are often high-level and hard to map to implementation issues.
- It can be combined with the specific chunk-ordering algorithm of any existing chunk-based MRR proposal. As a result, that proposal, designed for a certain coherence protocol, can now record relaxed-consistency executions.
- It has modest hardware requirements. Its hardware is local to the processors and requires no change to the cache coherence protocol.
- It produces a compact log representation of a relaxed-consistency execution.
- The resulting log enables efficient deterministic replay with minimal hardware support.

We evaluate RelaxReplay with simulations of an 8-core Release-Consistent (RC) multicore running SPLASH-2 applications. The results show that RelaxReplay induces negligible overhead during recording. In addition, the average size of the log produced is 1–4x the log sizes reported by existing SC- or TSO-based MRR systems. Hence, the bandwidth required to save this log is still a small fraction of the bandwidth provided by current machines. Finally, deterministic replay using this log is efficient: the sequential replay of these 8-processor executions with minimal hardware support takes on average 6.7x as long as the parallel recording.

This chapter is organized as follows: Section 4.2 provides a background; Sections 4.3 and 4.4 present RelaxReplay’s design and implementation, respectively; Section 4.5 evaluates RelaxReplay; Section 4.6 discusses related work; and Section 4.7 concludes the chapter.

### 4.2 Background on Chunk-Based Recording

State-of-the-art proposals for hardware-assisted MRR record each processor’s execution as a series of Chunks (also called Blocks or Episodes) of instructions executed between communications with other processors [10, 31, 34, 56, 57, 65, 66, 67, 81]. The chunks of different processors are ordered in a graph based on inter-processor data dependences. A typical chunk-based recorder provides three main functionalities:
(1) establishes chunk boundaries such that each chunk’s execution appears atomic, (2) establishes a proper order between chunks that captures all data dependences (to ensure correct replay) and has no cycles (to avoid replay deadlocks), and (3) represents chunks in the log in an efficient format.

Chunk boundaries are set at points where the executing processor communicates with other processors. Chunk-based recorders usually keep track of the read and write operations performed by the instructions of the current chunk. Often, the addresses of these operations are hashed in Bloom filters [12] and stored as read and write signatures. At the same time, the hardware checks for cache-coherence transactions that conflict with the read or write set of the current chunk. When one does, we have detected an inter-processor data dependence. Then, in simple designs, the current chunk is terminated and a new chunk starts. There are optimizations that allow chunks to grow beyond the conflicts.

Chunk-based recorders must ensure that the chunk containing the source of a dependence is ordered before the chunk containing the destination of it. For this, some schemes piggyback ordering information on coherence messages (e.g., [34, 81]) or add new messages [66]. Specifically, when an incoming coherence request conflicts with the local chunk, the global order of the local chunk is sent to the requesting processor (or broadcasted to all processors in [66]), so that its chunk orders itself after the local one. Alternatively, other schemes rely on a globally-consistent clock (e.g., [31, 67, 65]) that is available to all processors to establish chunk ordering. In both cases, by replaying the chunks according to their global order, all data dependences will be enforced.

Chunk-based recorders log chunks in a very efficient format. Specifically, a chunk is represented as the number of instructions (or memory operations) performed in the chunk, together with the recorded global ordering of the chunk.

### 4.2.1 Advantages of Chunk-Based Recording

Chunk-based recorders have at least three advantages over non chunked-based ones that have made them popular. Firstly, their operation lends itself to a relatively simpler hardware implementation in the cache hierarchy, while still generating small log sizes. Secondly, they support application-level RnR especially well because their recording hardware can be easily virtualized [57, 31, 65] and shared by multiple independent applications.

A third advantage is that the resulting logs can be efficiently replayed with minimal hardware support,
as in Cyrus (Section 3.4.2). Specifically, all that they need is a counter that counts the number of instructions (or memory-access instructions) executed, and then triggers a synchronous interrupt when the number of instructions in the chunk are exhausted. In this way, instructions can be replayed natively by the hardware rather than being simulated by an instruction simulator. At the same time, a simple software module can enforce the recorded chunk order. This combined hardware/software solution enables efficient native replay. Moreover, with the appropriate design, the resulting logs can be replayed in parallel [10, 31], and deliver fast replay.

4.2.2 Main Limitation: Access Reordering

In its basic form, chunk-based recording (as well as non chunk-based one) relies on the assumption that processors expose their memory operations to the coherence subsystem in program order, providing a sequentially-consistent environment [44]. Hence, any execution that violates SC cannot be captured by these recorders.

Unfortunately, commercial machines almost universally use more relaxed memory models, allowing loads and stores to perform out of program order. For example, to show how aggressive modern processors are, Figure 4.1 shows the fraction of memory-access instructions that are performed out of program order — i.e., with some earlier memory instructions still pending. The details of the experiment are discussed in Section 4.5.1. Of all the memory instructions, on average, 59% are out-of-order loads and 3% are out-of-order stores.

To begin to address this problem, there have been a few proposals for MRR under non-SC models. As indicated in Section 4.1 and discussed in Section 4.6, however, these proposals address only a conservative memory model (TSO), or are otherwise limited. To help popularize RnR, we need to find a general solution for MRR that works for any of the relaxed-consistency models used in current processors (such as ARM [8], Power [69] or Tile [76]). The rest of this chapter presents a solution to this problem that is compatible with the use of chunk-based recording.
4.3 RelaxReplay Design

4.3.1 Concept of Interval

To understand RelaxReplay, we define the concepts of performing and counting a memory-access instruction, and the notion of an Interval. A load instruction performs when the data loaded returns from the memory system and is deposited into a register. Later, the load retires when it reaches the head of the Reorder Buffer (ROB) and has already performed. A store instruction retires when it reaches the head of the ROB and its address and data are available. At this point, the store is deposited into the write buffer. Depending on the memory consistency model, the store can be merged with the memory system right away, or has to wait to do so until all earlier stores have been removed from the write buffer. Merging may trigger a coherence transaction. When the coherence transaction terminates (i.e., when all the necessary replies and acknowledgments have been received), the store has performed. Finally, in RelaxReplay, each retired load and each performed store in the processor goes through an additional logical stage in program order that we call Counting. Counting records the completion of the instruction in program order. Hence, each
An Interval in the execution of a processor is the period of time between two consecutive communications of the processor with other processors. An interval has a Perform Set and a Counting Set. These are the sets of perform and counting events, respectively, that took place in the processor during the interval. The set of perform events in an interval may correspond to memory-access instructions that are not contiguous in program order. This is because, in a relaxed-consistency machine, accesses can perform out of order. This is in contrast to the instructions of a chunk in a conventional chunk-based recorder, which are required to be contiguous. However, the set of counting events in the interval do correspond to consecutive memory-access instructions, since counting is done in program order.

4.3.2 Main Idea in RelaxReplay

In an RnR environment that supports general relaxed consistency models, working with chunks of contiguous instructions, as in conventional chunk-based recorders, is inconvenient. Instead, we propose to use the interval abstraction, which directly corresponds to the work performed between communications. To show the usability of intervals, we make two observations.

Observation 1: In memory-consistency models that support write atomicity, the perform event of a given access can only be placed in a single interval.

The property of write atomicity means that a write operation by a processor can be observed by another processor only if it has been made visible to all other processors, and that writes to the same location are serialized [74]. This property, which is typically enforced by the coherence substrate, is provided by all the popular multiprocessor systems in use today. It implies that the execution of a memory access can be thought of as atomic, and can only be placed in a single interval, namely the one where the access performs. As a result, we can record the execution of a processor as a sequence of intervals, where each access is assigned to the interval where it performs.

Unfortunately, representing an interval as a set of perform events is inefficient. Indeed, since memory instructions are performed out of program order, we would have to record the complete list of such events. To reduce the state we need to log, it is better to record the interval as a set of counting events (which can be efficiently represented as a range of consecutive in-order memory instructions) plus some reorder information. A second observation allows us to keep this additional reorder information that we need to
Observation 2: For the large majority of memory-access instructions, we can logically move the perform event forward in time to coincide with its counting event.

Given a memory-access instruction by a processor \( (P_1) \), we can logically move its perform event forward in time to coincide with its counting event if no other processor \( (P_j) \) has observed the access between the two points in time. \( P_j \) observes the access if it issues a conflicting access to the same (line) address that causes a coherence transaction that reaches \( P_1 \) between the two points. By “moving”, we mean that, as far as the other processors are concerned, the instruction can be assumed to have performed at the point of its counting. Since the access has not yet been observed by any other processor, this assumption will not affect any of the inter-processor dependences and, therefore, is correct. Fortunately, in practice, the large majority of accesses are not observed between the two events.

As an example, Figure 4.2(a) shows a store (ST) and a load instruction (LD) from a processor in program order, and their perform (P) and counting (C) event times. It also shows the time when an external communication occurs and, therefore, the interval terminates. In the figure, the perform events are in order. Figure 4.2(b) shows the case when the perform events are out of order. In both cases, each perform event happens in the same interval as its corresponding counting event and, thus, can be trivially moved to its counting time. Therefore, in both cases, we can concisely represent this interval as including the two accesses in program order.

![Figure 4.2: Examples of a two-instruction pattern with different timings for their perform (P) and counting (C) events.](image)

In Figure 4.2(c), the load has its perform and counting events in two different intervals. In this chapter, we present two version of RelaxReplay, depending on how we deal with this case. In a base design with simpler hardware, called RelaxReplay_Base, the perform event is never moved across intervals to its count-
ing event; in an optimized design with more hardware, called RelaxReplay Opt, the perform event is still moved across intervals to its counting event if none of the coherence transactions received between the two events conflicts with the (line) address of the access.

If RelaxReplay is able to move all the perform events to their counting events, each interval is concisely logged as comprising a certain number of accesses in program order — irrespective of the actual access reordering that occurred during recording due to the relaxed consistency model. Otherwise, the log entry for an interval also includes additional information on what accesses were counted in the interval but were out of order. We will discuss the exact representation later. Since, for the large majority of accesses, RelaxReplay is able to move the perform events to the counting events, the RnR log of intervals is both stored and replayed efficiently.

Overall, RelaxReplay is able to record an execution under any memory consistency model with write atomicity, and store it in a log for efficient deterministic replay. RelaxReplay relies on hardware that tracks the perform and counting events of each memory access and, while watching for conflicting accesses from other processors, tries to combine them before storing a compact representation of intervals in the log.

Note that RelaxReplay’s goal is to record intervals. For a full MRR solution, we also need a mechanism to establish a proper order between intervals of different processors. For this, we can use any of the existing chunk-based recording schemes. Such schemes now use coherence messages and read/write signatures to establish a proper order between intervals rather than chunks.

Next, we describe the architecture that processes perform and counting events, how it handles store-to-load forwarding, and how we replay a RelaxReplay log.

### 4.3.3 Tracking Instruction Events in RelaxReplay

Intuitively, the RelaxReplay architecture requires a longer ROB that keeps each memory-access instruction in the processor beyond retirement, and until it is ready to be counted. At that point, if the instruction’s perform event can be moved to its counting event, the instruction is included in the current interval as an in-order access, and logged as such. Otherwise, the instruction is included in the current interval as a reordered access, with enough state added to the log so that it can be correctly replayed.

In practice, rather than enlarging the ROB, RelaxReplay adds a hardware structure to the processor that works in parallel with the ROB for memory-access instructions. The structure is a circular FIFO called
Tracking Queue (TRAQ) (Figure 4.3). As a memory-access instruction is inserted in the ROB, it is also inserted in the TRAQ. A memory-access instruction is removed from the TRAQ when it is at the head of the TRAQ and is ready to be counted — i.e., for a load, it is performed and retired, and for a store, it is retired and performed. At that point, the instruction is counted and added to the log record for the interval. Note that the TRAQ can contain both non-retired and retired accesses. The ROB-like structure of the TRAQ enables RelaxReplay to handle the squashing of speculative instructions easily, as we explain in Section 4.4.

![Figure 4.3: High-level architecture of RelaxReplay.](image)

RelaxReplay keeps in a register the ID of the interval that is currently being processed at the head of the TRAQ. This ID is a counter called Current Interval Sequence Number (CISN) (Figure 4.3). Every time the processor communicates with another processor, the current interval is terminated, its information is stored in the memory log, the CISN is incremented, and a new interval starts.

The fundamental operation of the RelaxReplay hardware is simple. When a memory-access instruction is performed, the current value of the CISN is copied to the instruction’s TRAQ entry. It is stored in a field called Performance Interval Sequence Number (PISN). When the instruction reaches the TRAQ head and is counted, its PISN is compared to the CISN. At this point, there are several possible outcomes.

First, if the two values are the same, the interval has not changed since the perform event. Hence, RelaxReplay logically assumes that the memory-access instruction performs at the point of counting. In this case, RelaxReplay simply increments the count of consecutive memory-access instructions that have executed in this interval. Such count will be included in the log record for the interval that will be stored to
memory when the interval terminates. An example of this case is shown in Figure 4.4(a), which depicts a load that performs and is counted in interval 10, and whose perform point is logically moved by RelaxReplay to its counting point.

Second, if the PISN and CISN are different, the interval has changed because the processor has communicated between the perform and counting events. In RelaxReplay_Base, we process the access as reordered, as we will see later. In RelaxReplay_Opt, the hardware checks if the access is indeed reordered by comparing its (line) address to the (line) addresses of all the coherence transactions that the processor received since the PISN interval. Such addresses are collected in hardware in a structure called Snoop Table (Figure 4.3). This structure is only present in RelaxReplay_Opt, and is described in detail in Section 4.4.2.

If the comparison shows that no transaction conflicting with that address has been received, then RelaxReplay logically assumes that the memory-access instruction performs at this point, as in the first case. As before, RelaxReplay increments the count of consecutive memory-access instructions that have executed in this interval. An example is shown in Figure 4.4(b), which depicts a load that performs in interval 10 and is counted in 12. Since the processor has received no transaction that conflicts with this address in the meantime, the perform point is logically moved.

However, if the comparison finds that a conflicting transaction has been received, or the machine only supports RelaxReplay_Base, then the hardware records a reordered access. The following sections describe the cases of a reordered load and a reordered store separately.
Reordered Loads

To be able to record reordered loads, RelaxReplay needs to retain the values that loads obtain as they perform, until the loads’ counting time. Such values are stored in the corresponding TRAQ entries, as part of what Figure 4.3 refers to as Other.

When RelaxReplay counts a load and finds that it is reordered, it does not increment the count of consecutive memory-access instructions executed in this interval. Instead, it adds a special type of entry in the log record for the interval. The entry contains the value that was returned by the load as it performed (and was retained in the TRAQ). Later, when the execution is deterministically replayed, the value is read from the log and supplied to the destination register of the load. In this way, the replay of the load in program order can correctly reproduce what happened in the recorded execution out of program order. If, instead, during the replay, the load tried to access the memory system as it replayed, it might read an incorrect value. Note that any consumers of the load, as they are replayed in program order, will obtain the correct value. Xu et al. [86] used this approach of recording the values returned by out-of-order loads in the log for TSO machines.

An example is shown in Figure 4.4(c). A load performs in interval 10, and the processor later receives a coherence event that conflicts with the loaded address. The load is counted in interval 12. RelaxReplay then takes the value read by the load and stores it in the log record for interval 12.

Reordered Stores

To be able to record reordered stores, RelaxReplay needs to retain the values they write and the addresses they write to, until the writes’ counting time. Such values are saved in the TRAQ entries as part of the Other fields.

When RelaxReplay counts a store and declares it reordered, it does not increment the count of consecutive instructions executed in this interval. Instead, it adds another special type of entry in the log record for the interval. The entry contains the address written to, the value written, and the difference between CISN and the value of PISN in the store’s TRAQ entry. We call this difference Offset; it denotes how many intervals ago the store performed.

Before this log can be used for deterministic replay, this entry needs to be extracted from this interval’s record and inserted in the record of an earlier interval — specifically, at the end of the interval that is Offset.
positions earlier, which is the interval when the store performed. In the interval where the store is counted, we leave a dummy entry so that the store is not re-executed there. This “patching” step can be done as an off-line pass or on the fly as the log is read for replay.

After this change is made, the log is ready for replay. The store entry is found in the interval when it was performed, and the log contains the value to store and the address to store to. The store is thus executed, exactly reproducing the conditions in the recorded execution. In the interval where the store was counted, the store instruction is skipped (as indicated by the dummy entry mentioned above).

Figure 4.4(d) shows an example of a store that performs at interval 10, and the processor later receives a conflicting coherence event. The store is counted in interval 12. RelaxReplay then takes the value and address from the TRAQ and, together with an offset of 2, stores them in the log record for interval 12.

Example

To understand the format of the log record for an interval, Figure 4.4(e) shows the more extensive example of an interval that counts 8 memory-access instructions. Of these, \(i_1, i_2, i_4, i_5, i_7,\) and \(i_8\) both perform and are counted in interval 15. However there is a load (\(LD\)) and a store (\(ST\)) that perform in interval 10 and are counted in interval 15. Assume that none of the communications between intervals 10 and 15 conflict with the addresses accessed by \(LD\) or \(ST\).

If we use RelaxReplay_Base, the hardware does not know that there is no conflict and assumes that \(LD\) and \(ST\) are reordered. Hence, as shown in Figure 4.4(e), as it counts \(LD\), it reads the value that \(LD\) loaded and saves it in the log record. As it counts \(ST\), it reads the value that \(ST\) stored and the address it stored to, computes the offset of 5, and saves the three values in the log record.

Figure 4.4(f) shows the resulting log record for interval 15. It contains several entries, which are inserted in order as instructions are counted in order. As \(i_1\) and \(i_2\) are counted, they increment the counter of consecutive instructions that have executed in this interval. As RelaxReplay reaches \(LD\) and finds it reordered, it saves the counter in an entry of type \textit{InorderBlock}, and resets the counter. This means that there is a group of 2 in-order instructions executed. Then, it records an entry of type \textit{ReorderedLoad} with the value of the load. This means that the next instruction in program order is a reordered load. Then, for instructions \(i_3\) and \(i_4\), RelaxReplay records another entry of type \textit{InorderBlock} with size 2. Then, RelaxReplay records an entry of type \textit{ReorderedStore} for \(ST\), with its address, value, and offset. This entry signifies that the next instruction
in program order is a reordered store. Finally, RelaxReplay stores another entry of type \textit{InorderBlock} with size 2 for \(i_7\) and \(i_8\). This information is enough for the deterministic replay of these instructions. As will be seen later, this log format enables efficient replay.

Figure 4.4(g) shows the log for the same interval using RelaxReplay\_Opt. Since RelaxReplay\_Opt discovers that none of the intervening coherence transactions conflicts with the addresses of \textit{LD} or \textit{ST}, it records \textit{LD} and \textit{ST} as in-order accesses. In general, since the number of truly reordered accesses is very small, this log format is often very compact.

More details of the hardware and logging are presented in Section 4.4. In particular, the \textit{InorderBlock} entries count the number of total instructions in order, not just memory-access instructions. This design eases replay.

### 4.3.4 Handling Store-to-Load Forwarding

Modern superscalar processors typically allow store-to-load forwarding, whereby a load gets its value from an older store of the same processor that is pending in the write buffer. Such a load is not serviced off the coherent memory; it obtains its value from the non-coherent write buffer. In this section we show that RelaxReplay correctly records such loads.

Figure 4.5 shows the timing of a forwarding instance, where a load (\textit{LD}) obtains its value from an older store (\textit{ST}). Following RelaxReplay’s operation, \textit{LD} performs as soon as it gets the forwarded data, before \textit{ST} merges with the memory system and performs. Later, \textit{ST} is counted and \textit{LD} is counted.

![Figure 4.5: Timing of store-to-load forwarding.](image)

RelaxReplay seamlessly supports this case. Since \textit{LD} gets its value from \textit{ST}, we can assume it \textit{logically} performs at the same time as \textit{ST}. Thus, in order to correctly record \textit{LD}, we only need to monitor conflicting accesses between \textit{ST}’s perform event and \textit{LD}’s counting event (Period 1 in Figure 4.5). However, this period is properly contained between \textit{LD}’s perform and counting events (Period 2 in Figure 4.5). Thus, if
there is a change of interval (in RelaxReplay_Base) or reception of a conflicting coherence transaction (in RelaxReplay_Opt) in Period 2, we conservatively assume that it happened in Period 1. In this case, the hardware saves in the log the value obtained by $LD$ at its perform point, and the replay system later uses it at the counting point. Otherwise, the hardware correctly moves $LD$’s perform point to its counting point. No change to RelaxReplay is needed.

### 4.3.5 Replaying a RelaxReplay Log

The log generated by RelaxReplay is very compact and enables efficient replay using only minimal hardware support. To replay an execution, we use a module in the OS as in Cyrus (Section 3.4.2). Specifically, during replay, the OS reads the log of intervals and enforces the order of the intervals. As the OS reads the record for an interval, before launching its execution, it waits until all intervals ordered before this interval finish executing. This can be accomplished using software synchronization through condition variables or semaphores. In addition, the OS also injects the application inputs that were recorded in the original execution.

The log record for an interval can have three types of relevant entries: $InorderBlock$, $ReorderedLoad$, and $ReorderedStore$. If an $InorderBlock$ entry is found, the OS configures a hardware counter to generate an interrupt when the number of executed instructions equals the size of the block. This approach, proposed in the Cyrus system, requires a synchronous interrupt from the counter — i.e., the interrupt should be triggered upon (and before) executing the first instruction after the block. When the block is complete, the
interrupt transfers the control back to the OS. This instruction counting mechanism is similar to performance counters available in modern commercial microprocessors. It is the only hardware support needed to replay RelaxReplay logs.

If a ReorderedLoad entry is found, the OS reads the value from the log. It then saves it in the destination register of the load that is part of the architectural context of the application saved in the OS. Recall that the application context was saved upon entering the OS and will be restored before exiting the OS. The OS also advances the program counter, which is also stored as part of the architectural context.

If a ReorderedStore entry is found, the OS reads the address and value from the log and performs the memory update. Recall that we are now in the interval where this store performed, not where the store was counted. This is because this entry was processed earlier by a “patching step” (Section 4.3.3), which moved it from the store’s counting interval to its perform interval. Hence, in the current interval, there is no corresponding store instruction. Therefore, the OS does not advance the program counter. Later, when the OS reaches the interval where the store was counted, the OS will find the corresponding dummy entry. At that point, the OS will take no action beyond advancing the program counter by one.

When all the entries of the interval are processed, the OS uses software synchronization to signal the completion of this interval to its successors. Then, it reads the next interval from the log. Note that the replay process is oblivious to whether the log comes from RelaxReplay_Base or RelaxReplay_Opt; both use the same log format.

4.3.6 Discussion

RelaxReplay can be used to convert any of the existing chunk-based MRR schemes [10, 31, 34, 65, 66, 67, 81] to an MRR solution for relaxed-consistency models. In this case, as shown in Figure 4.7 RelaxReplay uses the chunk-ordering mechanism of the specific MRR scheme to form and order intervals and, then, uses the techniques outlined in this chapter to capture instruction reordering. In doing so, RelaxReplay retains the basic properties of the original chunk-based MRR scheme. For example, if the original scheme admits parallel replay of chunks [10, 31], then the resulting interval-based solution will admit parallel replay of intervals.

We designed RelaxReplay in a way that does not rely on knowing the detailed requirements of a particular relaxed-consistency memory model. RelaxReplay works for any relaxed-consistency model as long
as the coherence substrate supports write atomicity. This is a well-established property of current memory subsystems that is likely to hold in future generations of processors. We chose this approach because the memory models of most commercial processors are ill defined. In addition, new generations of processors may add new instructions to their ISAs that use different memory models than the older instructions. Finally, memory models are often defined in abstract, usually declarative, terms that do not provide much intuition about the implementation techniques used to support them.

### 4.4 Detailed Implementation Issues

#### 4.4.1 Memory Race Recorder

Following previous RnR designs, we place the hardware for recording memory races in a per-processor Memory Race Recorder (MRR) module. This module is shown in the top right part of Figure 4.6(a). It comprises two parts. On the left side, there is the mechanism for creating and ordering intervals, which can reuse any of the designs proposed by existing chunk-based recorders. On the right side, there is the mechanism to track events within intervals, which is the proper RelaxReplay hardware. The inputs to the MRR module are processor signals (instruction dispatch into the ROB, instruction retirement, memory operation performed, and pipeline squash), and memory system signals (coherence transactions).

For the mechanism to create and order intervals, we show a design that follows the QuickRec approach, i.e., records a total order of intervals based on a globally-consistent scalar timestamp. The timestamp associated with each interval is the cycle count of a global clock when the interval was terminated. Intervals are ordered according to their timestamps. The scheme uses a snoopy coherence protocol.

As shown in the figure, the hardware needed is a pair of Bloom filters as the read and write sig-
natures of the current interval, a Global Timestamp counter that counts the number of cycles of a global clock, and a Log Buffer that automatically saves the log records. When memory operations are performed, their line addresses are inserted into the signatures. Snooped coherence transactions are checked against the signatures; if a conflict is detected, the current interval is terminated.

The proper RelaxReplay hardware is on the right side, and records the events within an interval. It comprises the TRAQ, the Current Interval Sequence Number (CISN), the Current InorderBlock Size count, and the Snoop Table. The latter is only needed in RelaxReplay Opt and will be discussed later. The Current InorderBlock Size count is the number of in-order instructions that have so far been counted for the current block; this count is saved in the next InorderBlock entry logged.

Memory-access instructions are inserted into the TRAQ in program order when they are dispatched to the ROB. If the TRAQ is full, instruction dispatch stalls. The TRAQ also receives pipeline flush information from the processor, in order to keep its state consistent with the ROB’s. Specifically, if the ROB is flushed, then the TRAQ is also flushed accordingly. This occurs, e.g., on a branch misprediction. If an individual instruction in the ROB is squashed and replayed, the TRAQ takes no action, since its entry in the TRAQ will be correctly overwritten upon the re-execution of the instruction. This occurs, e.g., when a speculative load is squashed and replayed due to memory consistency requirements.

Figure 4.6(b) shows a TRAQ entry. Each memory-access instruction allocates a TRAQ entry and stores the address accessed, the value read or written, and the PISN. The other two fields in a TRAQ entry are the Snoop Count and the Non-Memory Instruction (NMI) field. The former is only needed in RelaxReplay Opt and will be discussed later. The NMI field enables RelaxReplay to log block sizes (in InorderBlock entries) in number of instructions rather than in number of memory-access instructions. This support may ease replay because processors are more likely to provide interrupt support for number of instructions executed than for number of memory-access instructions executed.

The NMI field works as follows. When a memory-access instruction (M) is dispatched and obtains a TRAQ entry, its NMI field is set to the number of instructions dispatched since the most recent memory-access instruction. Then, when M reaches the TRAQ’s head and is counted, the Current InorderBlock Size count is incremented by the value in the NMI field (plus one if M is not reordered).

The NMI field has a limited number of bits, which is 4 in our implementation. It is possible that more than 15 instructions appear between two consecutive memory-access instructions. In this case, RelaxReplay
allocates a TRAQ entry for each group of 15 such instructions. These TRAQ entries do not correspond to any memory-access instruction, and their NMI field is set to 15.

Figure 4.6(c) shows the format of the different types of entries in the log record of an interval. An InorderBlock entry is recorded for a group of consecutive instructions to be replayed in order. It includes the value of the Current InorderBlock Size count. A ReorderedLoad and ReorderedStore entry is recorded for each reordered load and store, respectively; their fields have been discussed before. An interval may log multiple instances of each of these three entry types. Finally, when an interval ends, an IntervalFrame entry is logged, with the value of CISN to identify the interval. In addition, an IntervalFrame must also contain some ordering information to establish its order among all the recorded intervals. The information required depends on the particular interval-ordering mechanism used. In our case, since we use the QuickRec interval ordering, it suffices to record the current value of the Global Timestamp.

4.4.2 Extension for RelaxReplay Opt

RelaxReplay Opt tracks the coherence transactions that a processor observes between the perform and counting events of a memory-access instruction. If the address of any of them conflicts with the address accessed by the instruction, the latter is declared reordered at counting time. To track transactions, RelaxReplay Opt adds the Snoop Table in the MRR (Figure 4.6(a)) and the Snoop Count in each TRAQ entry (Figure 4.6(b)).

The Snoop Table consists of two arrays of counters (Figure 4.8). When the processor observes a coherence transaction, the transaction’s line address is hashed, using a different function for each array, and the resulting two counters in the arrays are incremented. We use two arrays to reduce false positives caused by aliasing. When a memory-access instruction performs, its line address is hashed, and the corresponding two counters in the Snoop Table are read. The current values of these two counters are then stored in the Snoop Count field of the TRAQ entry. Later, when the instruction is counted, if its PISN is not equal to CISN, the two counters are read again from the Snoop Table. Their current values are compared to the values saved in the Snoop Count field. If none of the counters has changed or only one has (this case is due to aliasing), the instruction is declared in order; otherwise, it is declared reordered. If it is in order, since we are moving the perform event of the instruction to the current interval, we insert the address accessed by the instruction in the read or write signature (for a load or store, respectively), to ensure proper ordering of intervals.
The counters are allowed to wrap around. Moreover, no action is taken when a line is evicted from the cache: there is no danger of missing a coherence transaction because, in a snoopy protocol, all caches see all the transactions.

Overall, although conservative, this design correctly detects all of the true conflicts. The only problem would be if the counter size was so small that between the perform and counting points, a counter could wrap around and reach exactly the same value. To prevent this case, we use sizable structures: two 64-entry arrays of 16-bit counters. This means that the overall Snoop Table size is 256 bytes. In addition, the Snoop Count field in each TRAQ entry is 4 bytes. For the 176-entry TRAQ that we evaluate, the combined size of all the Snoop Count fields is 704 bytes. These are minor costs for RelaxReplay_Opt’s large reduction in log size and increase in replay speed (Section 4.5).

4.4.3 RelaxReplay for Directory Coherence

RelaxReplay’s mechanism to track events within intervals remains unchanged for directory-based coherence, whether centralized or distributed, as long as write atomicity is guaranteed. The mechanism to order intervals may need to change, and we can use any of the proposed chunk-based MRR schemes that work for directories.

One issue that appears in directory-based protocols is that, after a dirty line is evicted from a cache, the cache is no longer able to observe coherence transactions on the line. In this case, the Snoop Table proposed in Section 4.4.2 for RelaxReplay_Opt would lose its ability to observe conflicting transactions. To solve this problem, when a dirty line is evicted, its address is hashed and the two corresponding counters in the Snoop Table are incremented. This ensures that any memory-access instruction that performed an access to that address but has not been counted yet, is conservatively declared reordered. Hence correctness is preserved.
4.4.4 Modest Hardware Complexity

RelaxReplay’s hardware tracks events within intervals. Such hardware is local to the processors, rather than being distributed system-wide. Even within a processor, it leverages the well-understood general structure of the ROB. It does not require any changes to the cache coherence protocol of the machine. Moreover, its operation is independent of the scheme used to order intervals. Consequently, when RelaxReplay is paired with an interval ordering scheme that itself does not require modifications to the coherence protocol, such as QuickRec (Chapter 2) or Cyrus (Chapter 3), RelaxReplay provides a general MRR solution that works without coherence protocol modifications. This fact substantially lowers its hardware complexity. Finally, the resulting log can be replayed using minimal hardware support, which is very similar to existing performance counters.

4.5 Evaluation

4.5.1 Experimental Setup

We evaluate RelaxReplay with a cycle-level execution-driven simulator. We model a multicore with 4, 8 (default), or 16 cores. The cores are 4-issue out-of-order superscalars that use the RC memory model. Cores have a private L1 cache and a shared L2. The interconnection network is a ring that uses a MESI snoopy cache-coherence protocol. Table 4.1 shows the architectural parameters, including those of RelaxReplay. From the table, we can compute the overall size of the per-processor RelaxReplay structures. Specifically, for RelaxReplay_Base, the overall MRR module of Figure 4.6(a) is 2.3KB, of which the TRAQ uses 1.8KB; for RelaxReplay_Opt, the MRR is 3.3KB, of which the TRAQ uses 2.5KB. Since the processor has 2 Ld/St units, we design the TRAQ to be written twice (at perform events) and read twice (at counting events) per cycle. We design the Snoop Table to be read twice (typically at perform events) and written once (on a snoop) per cycle. We run SPLASH-2 codes [83].

The effectiveness of RelaxReplay’s instruction tracking mechanism depends on the average size of the intervals. To a large extent, this size is determined by the maximum interval size chosen by the chunk-based recorder paired with RelaxReplay. Some recorders, such as Karma [10] and Cyrus set the maximum interval size to a small value, in order to increase replay parallelism. Other schemes, such as CoreRacer [67] and QuickRec (Chapter 2), use a very large maximum interval size because they replay sequentially, and large
Processor and Memory System Parameters

<table>
<thead>
<tr>
<th>Multicore</th>
<th>Ring-based with MESI snoopy protocol, 4, 8 (default), or 16 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>4-way out-of-order superscalar @ 2GHz, 176-entry ROB, 2 Ld/St units, 128-entry Ld/St queue</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>Private, 64KB, 4-way associativity, 64-entry MSHR, 32B line, write-back, 2-cycle round-trip</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>Shared, 512KB per core, 16-way associativity, 64-entry MSHR, 32B line, write-back, 12-cycle average round-trip</td>
</tr>
<tr>
<td>Ring</td>
<td>32B wide, 1-cycle hop delay</td>
</tr>
<tr>
<td>Memory</td>
<td>32B bus, 150-cycle round-trip from L2</td>
</tr>
</tbody>
</table>

Table 4.1: Architectural parameters.

Intervals have lower overheads. Thus, to assess the sensitivity of RelaxReplay to the maximum interval size, we evaluate RelaxReplay with two different maximum interval sizes: 4K instructions and infinitely large (INF).

To estimate replay performance, we have written a software module to control the replay according to the algorithm outlined in Section 4.3.5. In a real system, this module would be part of the OS. However, our execution-driven simulation setup does not run OS code. Thus, to measure the overhead of this control software, we link its code with the code of the application. With this setup, we can measure its performance cost. This control software uses the recorded total order of intervals to enforce interval ordering. For this, it uses condition variables. Then, for each interval, it follows the algorithm explained in Section 4.3.5: it executes the interval’s InorderBlocks on the hardware, and emulates the execution of the reordered instructions.

In the following, we first characterize the logs. Then, we evaluate the recording and replay performance. Finally, we analyze RelaxReplay’s scalability with the processor count.

4.5.2 Log Characterization

We start by analyzing how many memory-access instructions are found by RelaxReplay to be reordered. Figure 4.9 shows the number of such instructions as a fraction of all memory-access instructions, for 4K maximum intervals (Chart (a)) and INF maximum intervals (Chart (b)). Each chart shows bars for all the applications and the average. In each case, we have bars for RelaxReplay_Base and RelaxReplay_Opt.
On average, RelaxReplay_Base logs 1.7% and 0.17% of memory instructions as reordered for 4K and INF intervals, respectively. This number is much smaller than the 60% of memory-access instructions that are performed out of program order, as shown in Figure 4.1. This shows that most of the reorders are invisible to other processors. In fact, RelaxReplay_Opt reduces this fraction even more — to a minuscule 0.03% for both 4K and INF intervals. As we will see, this large reduction has a significant impact on the size of the generated logs and the replay speed.

In all cases, loads dominate the reordered instructions. Comparing the 4K and INF results, we see that larger intervals help RelaxReplay_Base reduce the fraction of reordered instructions. However, RelaxReplay_Opt’s effectiveness is independent of the interval size. This is because RelaxReplay_Opt relies on the Snoop Table to detect reordered instructions, rather than on whether perform and counting events are in the same interval.

The number of reordered instructions affects the number and size of the InorderBlock entries in the logs. Recall that an InorderBlock entry corresponds to a set of consecutive in-order instructions. An InorderBlock is terminated by either a reordered access or an interval termination. Hence, if an optimization such as RelaxReplay_Opt reduces the number of reordered accesses, the InorderBlock size increases and the number of InorderBlock entries goes down.
Figure 4.9: Fraction of memory-access instructions found by RelaxReplay to be reordered for 4K (a) and INF (b) intervals.
Figure 4.10 shows the number of InorderBlock entries in the logs for 4K (Chart (a)) and INF (Chart (b)) intervals. The figure is organized as the previous one except that, in each application, the bars are normalized to RelaxReplay_Base. The figure shows that RelaxReplay_Opt’s ability to reduce the number of reordered access results in many fewer InorderBlocks. On average, it only logs 13% and 48% as many InorderBlocks as RelaxReplay_Base for 4K and INF intervals, respectively.

Figure 4.10: Number of InorderBlock entries (IBs), normalized to RelaxReplay_Base, for 4K (a) and INF (b) intervals.
Finally, Figure 4.11 shows the uncompressed log size for the two designs, in bits per 1K instructions for 4K (Chart (a)) and INF (Chart (b)) intervals. We see that RelaxReplay_OPT reduces the log size over RelaxReplay_BASE substantially — the result of its ability to reduce the number of reordered instructions. For 4K intervals, the average log size per 1K instructions goes down from 360 bits in RelaxReplay_BASE to 22 bits in RelaxReplay_OPT; for INF intervals, it goes down from 42 bits to 12 bits. These are substantial reductions in logging needs.

The resulting RelaxReplay_OPT log sizes are 1–4x the log sizes reported for previous chunk-based recorders [18, 31, 34, 56, 66, 67] and are, therefore, comparable to them. This is despite the fact that the previous schemes required the strict SC or TSO models, while RelaxReplay_OPT handles the relaxed RC model. In fact, RelaxReplay_OPT’s logs are quite small compared to the several GB/s of memory bandwidth available in modern machines. Indeed, in our experiments, RelaxReplay_OPT generates on average only 48 MB/s and 25 MB/s of logging state for 4K and INF intervals, respectively, which is a small rate. On the other hand, RelaxReplay_BASE generates on average 840 MB/s and 90 MB/s. Although we consider the former excessive, the latter is small and shows that the simpler RelaxReplay_BASE design is a viable solution if large intervals are acceptable — i.e., when replay parallelism is not required.
Figure 4.11: Uncompressed log size in bits per 1K instructions for 4K (a) and INF (b) intervals.
4.5.3 Characterization of Recording Performance

It can be shown that the execution overhead of recording under RelaxReplay_Opt, or under RelaxReplay_Base with INF intervals is negligible. This is consistent with past proposals for hardware-assisted RnR. To understand why recording overhead is negligible in these cases, consider the two main sources of overhead: memory bus contention as the log is being saved, and stalls due to lack of TRAQ entries. From the previous section, we deduce that the induced memory bus contention is negligible for RelaxReplay_Opt and for RelaxReplay_Base with INF intervals.

To assess the TRAQ stall, Figure 4.12 shows the TRAQ utilization. The figure applies to both RelaxReplay_Opt and RelaxReplay_Base. In Chart (a), we show the average number of TRAQ entries utilized by each application. We see that, in all cases, this number is less than 64. This is a small number compared to the TRAQ's 176 entries. In Chart (b), we show distributions of number of used TRAQ entries for four representative applications. In the figure, each bar corresponds to the fraction of samples where a certain number of entries (grouped in bins of 10) were used. As can be seen, although different applications have different overall shapes, in all cases, most of the time around 80 or fewer entries are used. Hence, TRAQ-induced stall is very rare. It can be shown that it accounts for less than 0.3% of the execution time for RelaxReplay_Opt and for RelaxReplay_Base with INF intervals.
Figure 4.12: TRAQ utilization: average (a) and histograms for four representative applications (b).
4.5.4 Characterization of Replay Performance

Figure 4.13 shows the time it takes to replay the applications with RelaxReplay_Opt logs or RelaxReplay_Base logs, for 4K (Chart (a)) and INF (Chart (b)) intervals. For each application, the times are normalized to the time it takes to record the application, shown as the leftmost bar of each group. Note that, while recording was done in parallel with 8 processors, replay in these experiments is performed sequentially — because the interval-ordering mechanism records a total order of intervals. Moreover, the replay time is broken down into execution of the application (User Cycles) and execution of our control module that emulates the OS (OS Cycles). The latter orders intervals, reads log entries, and emulates reordered instructions.

From the figure, we see that replaying with the RelaxReplay_Opt log is fast. Although the replay is performed sequentially, it takes on average only 8.5x and 6.7x as long as the parallel recording for the 4K and INF intervals, respectively. OS time is about a third to a sixth of the replay time.

Replay with the RelaxReplay_Base log is a bit slower for INF intervals and substantially slower for 4K intervals. Specifically, sequential replay takes on average 8.6x and 26.2x as long as the parallel recording for the INF and 4K intervals, respectively. The slowdown is due to the larger fraction of reordered instructions. There is a substantial fraction of OS cycles, as the OS deals with reordered instructions. User cycles are sometimes higher than in the RelaxReplay_Opt bars because there are more pipeline flushes, as end-of-block interrupts transfer execution to the control module.

Overall, using the RelaxReplay_Opt log or the RelaxReplay_Base log with INF intervals ensures efficient replay. If we combine them with interval ordering schemes that admit parallel replay [10, 31], we expect substantially faster replay.
Figure 4.13: Replay time with Opt or Base logs, normalized to recording time, for 4K (a) and INF (b) intervals.
4.5.5 Scalability Analysis

To analyze RelaxReplay’s scalability with the number of processors, we repeat the experiments with 4- and 16-core machine configurations. Figure 4.14 shows how the fraction of memory-access instructions that RelaxReplay perceives as reordered (Chart (a)) and the log generation rate (Chart (b)) change with 4, 8, and 16 processors ($P_4$, $P_8$, and $P_{16}$) for RelaxReplay_Base and RelaxReplay_Opt. In each figure, the left- and right-hand sides present the results for the 4K and INF configurations, respectively. Each bar is the average of all the applications.

The figures show that both the fraction of reordered instructions and the log size increase with the number of processors. Leaving aside the case of RelaxReplay_Base with 4K intervals, we see that both instruction reordering and log size are still small for up to 16 processors but increase noticeably, although not exponentially.

The reason for the increase is that, with more cores, we have more traffic and, in particular, more coherence traffic. Moreover, in our ring-based snoopy protocol, all processors observe all the traffic. As a result, there is more chance for false positives in the signatures and in the Snoop Table. The former causes additional terminations of intervals, which results in bigger logs and, in RelaxReplay_Base, more reordering; the latter causes RelaxReplay_Opt to count more reordered instructions. As a result, we see fast
increases in both parameters. With directory coherence, we expect lower growth rates, as each core only sees coherence messages for the cache lines it accessed.

The case of RelaxReplay_Base with 4K intervals is less sensitive to the number of cores. The reason is that its behavior is largely determined by the small maximum interval size. Adding more coherence transactions only has a marginal impact in further reducing the interval sizes.

4.6 Related Work

RTR [86] supports MRR for TSO models by recording the value of loads that may violate SC. Similar to our approach, it records a load’s value if there is a conflicting access to its memory location between the time the load is performed and all of its predecessors are performed. Unlike RelaxReplay, it records dependences between pairs of communicating instructions.

DeLorean [56] and Capo [57] are chunk-based schemes that use the speculative multithreading hardware of BulkSC [15]. The underlying hardware enforces SC while allowing aggressive out-of-order execution of instructions. The execution is recorded by logging the order in which processors commit their chunks.

Rerun [34] and Karma [10] are chunk-based techniques for conventional multiprocessors with directory coherence. The papers also include proposals to integrate RTR’s solution for TSO recording with their chunk-based schemes. As such, they can be considered as chunk-based recorders for TSO. However, they only provide high-level discussions about how the integration could be done without providing detailed designs or results.

CoreRacer [67] is a chunk-based recorder for snoopy coherence and supports TSO by recording the number of stores pending in the processor’s write buffer when a chunk terminates. This allows CoreRacer to correctly account for reordered and forwarded loads by simulating the write buffer’s content during replay. Apart from being limited to TSO, replay efficiency may suffer because it requires write buffer simulation. Also, recording the number of pending stores for each chunk is often unnecessary since the resulting re-ordering is rarely visible to other processors. QuickRec (Chapter 2) uses the same basic mechanism to support TSO.

LReplay [18] is a hardware-assisted MRR solution that does not monitor coherence transactions. Instead, it includes a non-trivial centralized component that directly tracks the memory operations performed by all cores. It relies on this hardware to detect inter-processor dependences. It supports TSO using RTR’s
approach. Due to its specific recording technique, its replay algorithm is complicated and needs to simulate all instructions.

Lee et al. [48] builds on BugNet [59] and uses off-line search to infer inter-thread dependences for TSO [48] executions. They do not directly record dependences. Instead, they log data fetched on a cache miss (when it is accessed for the first time). This allows independent replay of each thread. They also periodically record some Strata [58] hints to speed-up the off-line search. Using the results of per-thread replays and the hints, inter-thread data dependences can be determined off-line.

Rainbow [70] builds on the sequentially-consistent Strata and improves it in two ways. Firstly, it uses a centralized hardware structure, called Spectrum History, to reduce the number of recorded strata, and thus, improve the log size and replay speed. Secondly, it shows that the same data structure can aid in detecting potential SC violations in order to record non-SC executions. The idea is to record some information about delayed and pending instructions that allows it to replay the situation correctly when an SC violation happens. Although the paper discusses the hardware structures and algorithms required to implement the first improvement fairly extensively, the second part is only explained vaguely and at a very high level. In particular, detailed record and replay algorithms are only presented for the first contribution; the paper does not explain the mechanisms or hardware structures required to track and communicate the pending and delayed instructions that are central to its second contribution.

It is difficult to provide a detailed comparison between Rainbow and RelaxReplay, especially in terms of the hardware data structures, given Rainbow’s lack of details and the complexity of its proposed SC-violation handling mechanism. However, it is clear that, similar to RelaxReplay, Rainbow requires write atomicity, since each instruction can only be recorded in a single spectrum. Unlike RelaxReplay, however, Rainbow cannot accommodate distributed directory protocols due to its centralized Spectrum History design. Also, unlike RelaxReplay, it needs to augment the coherence protocol messages and/or add new ones (if write-after-read dependences are to be explicitly recorded). In addition, it is unclear how the Spectrum History can be virtualized in order to accommodate application-level RnR (i.e., recording single applications instead of whole machines). Moreover, unlike RelaxReplay that can be applied to any chunk-based MRR scheme, the SC-violation handling mechanism of Rainbow is particular to its Strata-like design and cannot be directly used in conjunction with other MRR schemes.
4.7 Concluding Remarks

This chapter proposed RelaxReplay, the first complete solution for hardware-assisted MRR that works for any relaxed-consistency model with write atomicity. With RelaxReplay, we can build an RnR system for any coherence protocol — whether snoopy, centralized directory or distributed directory. RelaxReplay’s insight is a new way to capture memory-access reordering. Each memory instruction goes through a post-completion in-order counting step that detects any reordering, and efficiently records it. We presented two designs with different hardware needs, log sizes, and replay speeds.

RelaxReplay’s mechanism to capture memory-access reordering does not rely on consistency model specifications. It can be combined with the chunk-ordering algorithm of any existing chunk-based MRR proposal — enabling that proposal to record relaxed-consistency executions. Its hardware is local to the cores and does not change the cache coherence protocol. Finally, it produces a log that is compact and efficient to use for replay with minimal hardware support.

We evaluated RelaxReplay with simulations of an 8-processor RC multicore running SPLASH-2 applications. The results showed that RelaxReplay induces negligible overhead during recording. In addition, the average size of its log was 1–4x the log sizes reported for existing SC- or TSO-based MRR systems, and still a small fraction of the bandwidth of current processors. Finally, deterministic replay is efficient, since the sequential replay of these 8 processors with minimal hardware support took on average only 6.7x as long as the parallel recording.
Chapter 5

Replay Debugging: Leveraging Record and Replay for Program Debugging

5.1 Introduction

One of the main usage models proposed for RnR is program debugging. The motivation is that some software bugs such as data races are often hard to repeat across executions with the same inputs, which makes them hard to debug. Hence, having the ability to deterministically reproduce an execution should help debug them.

However, simply providing support for repeatedly finding the same bug will not help remove it. The process of debugging involves modifying the program, for example by adding code to access program data and code (e.g., reading program variables and invoking program functions), create and operate on new variables, and print state out. We call the process of performing all of these operations while using a log to replay an execution Replay Debugging.

Unfortunately, any of these changes is very likely to distort the program’s code and/or data, forcing the replayed execution to follow different paths than in the original execution encoded in the log. As a result, the log becomes inconsistent and cannot guide the new execution.

In practice, prior work has shown that this scenario still has some value. The relevant system, called DORA [78], can help diagnose bugs or test software patches. However, DORA often faces substantial divergence between the replayed execution and the original one. Importantly, it cannot guarantee deterministic replay. As a result, it is unable to ensure the exact reproduction of non-deterministic events that resulted in a bug. This is especially problematic when dealing with timing-dependent events like data races or atomicity violations.

Our goal, instead, is to be able to always guarantee exact replay during debugging, to quickly diagnose even highly non-deterministic bugs. Other researchers have also pointed out the importance of exact replay for debugging. Zamfir et al [88] argue that an effective replay-based debugging solution should provide
“Debug Determinism” — ability to reproduce the same failure and same root cause as in the recorded execution.

We argue that, to guarantee replay debugging with exact replay, we need two capabilities. One is the ability to generate, out of the instrumented code, an executable that is identical to that of the original application. The second is the ability to replay the execution encoded in the log while invoking the debug code at the appropriate locations in the code.

To attain this goal, we present rdb, an scheme for replay debugging that guarantees exact replay. With rdb, programmers’ debugging experience is very similar to an ordinary bug diagnosis process. They can write debug code to access program variables, invoke program functions, create and use new debug variables and debug functions, set watchpoints, and print state. However, they cannot modify the state or instructions used by the program itself. Under these conditions, rdb uses the log generated by hardware-assisted RnR to guarantee deterministic re-execution.

rdb’s capability is possible thanks to two mechanisms. The first one is a compiler pass that splits the instrumented application into two binaries: one that is identical to the original program binary, and another that encapsulates all the added debug code. The second mechanism is a runtime infrastructure that replays the application and, without affecting it in any way, invokes the appropriate debug code at the appropriate locations. No special hardware is needed beyond the original RnR system.

Overall, the contributions of this work are:

• It presents rdb, the first scheme for replay debugging that guarantees exact replay.

• It describes an open-source implementation of rdb using LLVM [3] for the compiler mechanism and Pin [53] for the runtime mechanism.

• It discusses an example of how rdb’s replay debugging is used to diagnose a real bug.

The rest of this chapter is organized as follows: Section 5.2 gives a background; Section 5.3 discusses how to use RnR for replay debugging; Sections 5.4-5.5 describe rdb; Section 5.6 presents an example of replay debugging; Section 5.7 outlines limitations; Section 5.8 covers related work; and Section 5.9 concludes the chapter.
5.2 Background

5.2.1 Assumed RnR Environment

As our baseline, we assume a hardware-assisted RnR environment like QuickRec. As mentioned in Chapter 2, QuickRec uses OS support to record program inputs and special hardware implemented with FPGAs for MRR. Recorded program inputs include system calls, data copied to application buffers by the OS as a result of system calls, signals, and results of some non-deterministic processor instructions. Memory interleaving is captured as a log of totally-ordered chunks. QuickRec’s handling of program inputs is typical of most existing application-level RnR systems (e.g., [75, 42, 57]) and its chunk-based MRR is similar in basic operation to most recent proposals for hardware-assisted RnR.

QuickRec’s replay tool is based on Intel’s Pin [53] binary instrumentation infrastructure. It takes the application binary together with the recorded input and memory logs. As the program replays, it is able to inject the recorded inputs at appropriate points. In addition, it counts the instructions of each chunk as it executes, and enforces the recorded size and ordering of each chunk. Figure 5.1 illustrates the high-level structure of the system.

![Figure 5.1: High-level organization of the assumed RnR system.](image)

5.2.2 Debugging Can Break Replay

To diagnose the root cause of a bug, programmers typically employ a process that involves the use of a debugger (e.g., gdb [1]), as well as writing some debug code. To do an effective job, programmers should
be able to write code to perform at least the following tasks:

- Inspect program state, including registers, variables, and memory content of the program.
- Calculate expressions based on such state. This can involve calling subroutines from the program being debugged.
- Present the inspection results, e.g., using print statements.
- Create and keep debug state in the form of local, global or heap-allocated data structures used only for debugging.
- Set breakpoints and watchpoints to trigger some debugging activity when certain conditions become true.

Such a debugging process almost always involves distorting the code and/or data state of the program. Unfortunately, RnR mechanisms are very sensitive to such distortion. As a result, if we try to use the RnR log created by the original execution to replay the distorted program, we will observe replay divergence from the log.

Specifically, any changes in the code or data layout can potentially affect the control flow of a thread, changing the number and type of instructions executed. In hardware-assisted RnR, it causes the chunk boundaries to be placed at wrong instructions during replay, causing potentially-incorrect chunk orderings. This, in turn, can violate the recorded inter-thread data dependences.

In addition, code or data changes can cause replay divergence even in single-threaded programs, where memory access interleaving is not a concern. For example, code changes may result in a different set of system calls than was recorded, or system calls that are invoked with different operands. As another example, programs often use the value of pointers (i.e., addresses of variables) to construct data structures such as sets or maps of objects. If the pointer values change, the internal layout in such data structures will change. When the program traverses these data structures, changes in pointer values can result in different traversal orders and executions.

Viennot et al. [78] consider the problem of replaying modified programs in the context of a software-only RnR engine called Scribe [42] (as opposed to our focus on hardware-assisted RnR). Scribe uses OS support to record the non-deterministic events of an execution. Their proposed “mutable replay” system,
called DORA, then uses a search-based technique to find and compare different ways of augmenting (or modifying) the recorded log in order to have it guide the execution of the modified code. When the search fails, DORA switches from replaying to recording in order to continue the execution. The trade-off made in such a system, thus, is that it gives up on the guarantee of exact replay, in order to gain more flexibility by supporting a range of program modifications.

We take a different approach here. We aim to provide guaranteed deterministic replay using an RnR log, while allowing programmers to perform the debugging tasks mentioned earlier.

5.3 Using RnR Support for Debugging

We call Replay Debugging the process of debugging a program while replaying its execution using a previously-generated RnR log. To quickly diagnose non-deterministic bugs, we are interested in the ability to always guarantee exact replay during replay debugging. A requirement for this capability is that the debugging process should not distort the program’s code or data in any way. If this requirement is not satisfied, the RnR log becomes obsolete and cannot be used. Unfortunately, many of the features needed in an effective debugging process are at odds with such a requirement. In this section, we discuss four usability features that we believe are needed for effective debugging. For each, we outline the challenge it presents to our target environment, and how a system that we propose, called rdb, addresses the challenge.

5.3.1 Inline Debug Code in the Program Code

Programmers typically inline debug code in the program code, as if it were part of the main program. For example, in C/C++ programs, debug code is often enclosed between #ifdef and #endif pre-processor macros, so that it is included in the compilation of a debug version of the program and is excluded otherwise (Figure 5.2(a)). This approach enables writing complex debug logic while allowing easy access to the program’s state and code.

Challenge. Since the inlined debug code is compiled together with the main program’s code, it changes the program’s code and data structures, and renders the RnR log obsolete.
int a = 1;
#ifdef DEBUG
 printf("a is %d", a);
#endif

int a = 1;
#ifdef DEBUG
 rdb_begin
 printf("a is %d", a);
 rdb_end
#endif

void main() {
#ifdef DEBUG
 rdb_begin
 printf("J\nLVG\nD\n")
 rdb_end
#endif
}

Final main program code:
int g() { ... }
void main() {
#ifdef DEBUG
 rdb_begin
 printf("g() is %d", g());
 rdb_end
#endif
}

Extracted debug function:
void _rdb_func(int arg0) {
 printf("J\nLVG\nD\n")
}

Solution. To address this challenge, rdb uses a compiler pass to extract the debug code from the program code. Programmers can write inlined debug code, but they need to enclose it within special rdb markers so that the compiler can identify the enclosed code as debug code. Figure 5.2(b) shows the code surrounded by the rdb_begin and rdb_end macros understood by the compiler.

The step of extracting the debug code should take place in the compiler front-end at the level of the Abstract Syntax Tree (AST) — before any transformation or code generation is done. From this point on, the compiler will compile two different bodies of code: (1) the main program code, which is exactly the same code that was used for generating the binary of the recorded program, and (2) the debug code.

The extracted debug code is transformed before being compiled. This is because it references variables, memory locations and functions of the program that are not available to it after the extraction step. For example, in Figure 5.2(b), the reference to variable a will not be resolvable after the debug code is extracted. Hence, the compiler transforms each group of debug instructions into a debug function that receives, as its formal arguments, those variables of the program code that are accessed by the debug code (Figure 5.2(c)).

In addition, the compiler front-end needs to leave some markers in the main program code to convey to the back-end the location in the program where the debug code should be executed, as well as the variables it will access. The back-end will use these markers to generate extra files with information about the debug
functions and their arguments. This information will be used to invoke the extracted debug code.

### 5.3.2 Access Program Code & Data from the Debug Code

The debug code needs to be able to read arbitrary variables and memory locations of the main program. It also needs to be able to invoke subroutines in the program — e.g., to evaluate the value of an expression or to traverse program data structures that might in turn call other subroutines. To provide this capability, the debug code should run in the same virtual address space as the main program.

**Challenge.** Allowing the debug code to use the address space of the program to contain its code and state results in some memory ranges not being available to the main program. If the main program tries to access a location in these ranges, it can result in replay divergence.

**Solution.** \( rdb \) places the debug code and state in those parts of the program address space that are not going to be used by the main program. This is feasible, since the RnR log contains enough information to allow \( rdb \) to identify the memory ranges not used by the main program.

### 5.3.3 Output the Results of the Debug Code

Inspecting the program’s state is not very useful if the inspection results cannot be conveyed back to the developer. For example, in C programs, programmers often use \texttt{printf}.

**Challenge.** Since the debug code is running in the same address space as the program, it could call the program’s \texttt{printf}. However, the call will change the contents of data structures internal to the runtime library (\texttt{libc} in this case) — which are part of the main program state. This change will cause a replay divergence.

**Solution.** \( rdb \) provides the debug code with its own instance of the runtime libraries (e.g., \texttt{libc} and \texttt{libstdc++} for C/C++ programs). In the code generation phase, the compiler treats calls to such subroutines by the debug code differently than calls to the subroutines of the main program. For example, consider Figure 5.2(d). The debug code contains two function calls, namely \texttt{printf()} and \texttt{g()}. The compiler identifies \texttt{printf()} as a member of the runtime library and not as an input to the debug code. Later, when
the debug code is linked with its own libc, this function will be resolved to the printf in that instance of libc.

On the other hand, function \( g() \) comes from the main program. Like any other piece of main program accessed by the debug code, it will be passed to the debug code as an input. Specifically, when the debug function gets called, the location of \( g() \) will be passed as an argument to the function. Hence, the debug code calls the program’s instance of \( g() \). Figure 5.2(e) shows the resulting main program and extracted debug code.

### 5.3.4 Keep State in the Debug Code

When programmers debug code with complex data structures, they often need to keep some shadow state for debugging purposes. This is usually done by allocating some heap objects that outlive the piece of debug code creating them. They are accessed in the future by some other part of the debug code. In addition, these objects may need to include references to objects belonging to the main program.

**Challenge.** Debug code cannot allocate its dynamic objects in the same heap as the main program. This would change the program’s state and potentially result in replay divergence.

**Solution.** rdb provides the debug code with its own instance of the runtime library. Hence, the debug code will automatically use the heap that belongs to this runtime library as it invokes memory allocation routines (e.g., malloc()). Recall from Section 5.3.2 that rdb ensures that the addresses used by the main and debug codes do not interfere. However, since debug code lives in the same virtual address space as the main code, debug objects can easily contain references to objects belonging to the main program.

### 5.4 Basic Design of Replay Debugging with rdb

We argue that, to guarantee replay debugging with exact replay, we need two capabilities. One is the ability to generate, out of the code instrumented with debug statements, an executable that is identical to that of the original application. The second is the ability to replay the program encoded in the log while invoking the debug code at the appropriate locations in the code. In this section, we describe how rdb attains these two abilities. Before this, we discuss the structure of the debug code. In our discussion, we assume that rdb
5.4.1 Structure of the Debug Code

To replay-debug a program with rdb, a developer writes snippets of debug code inlined in the program code. The inlined debug code should be a single-entry, single-exit region in the control flow graph of the program. This is needed to ensure that the compiler can easily extract the debug code from the program. We call every such piece of debug code a Debug Region. Each debug region is enclosed between rdb_begin and rdb_end markers to help the compiler identify it.

The code in a debug region can freely access any object (variable, function or constant) that is accessible by the main program code as long as it does not write, directly or indirectly, to the memory owned by the main program. A debug region can also have locally-declared variables that are only visible in that debug region, and freely use functions provided by its private instance of runtime libraries. Figure 5.3 shows an example of a debug region with a for loop, a locally-declared variable, and a printf statement.

```
if (...) {
    N = ... /* program code */
    x = ... /* program code */
    rdb_begin
    int i;
    for (i = 0; i < N; i++) {
        printf("x[%d]=%d", i, x[i]);
    }
    rdb_end
    } else { ... /* program code */ }
```

Figure 5.3: Example of a debug region.

In addition, the developer can also write new functions to call from the debug region, and declare and use new global variables that do not exist in the original code. These function and global variable declarations are not in a debug region. We explain how rdb supports them in Section 5.5.1.

5.4.2 Generating the Executable for Replay Debugging

After the developer has augmented the program source with debug regions, the first step is to generate an executable of the application that, while identical to the original application in both code and data, can also
invoke the debug code. The idea in rdb is to force the compilation process to generate two binary files from the program source files. One is identical to the binary of the original program with no debug code; the other encapsulates all the extracted debug code.

To this end, the compiler takes each source file and generates two object files, one with the main program code, and the other with the extracted debug code. After all the files have been processed, the two sets of object files are linked separately to generate two different binaries.

In the following, we describe the operation in detail. We describe it in the context of the Clang/LLVM compilation flow [45], which is outlined in Figure 5.5. This tool set takes C/C++ source files and, in the front-end (leftmost box), translates them to unoptimized LLVM Intermediate Representation (IR). The output is then taken by the LLVM optimizer (central box), which generates optimized LLVM IR. For simplicity, the current implementation of rdb operates under the assumption that the code is compiled without optimization (i.e., with the -O0 command line option). Section 5.7.2 discusses the extensions needed to handle optimized code. Finally, the output of the central box is taken by LLVM CodeGen backend (rightmost box), which translates it into x86 machine code. rdb augments the last two boxes.

Figure 5.4: Compiling an example program for replay debugging: C program containing debug code (a); resulting LLVM IR generated by the Clang front-end (b); extracted debug module (c); resulting main code containing rdb markers (d); function descriptors (e); and argument descriptors (f).
To aid the presentation, we use the simple C program in Figure 5.4(a) as a running example. The original program reads a character from the standard input. The debug code then prints it to the standard output. Figure 5.6(a) shows the rdb compilation flow, which we will describe in steps.

We use the Clang front-end to translate the program source to its equivalent LLVM IR. After translation, the code in a valid debug region retains its shape as a single-entry, single-exit region enclosed between begin and end markers. Figure 5.4(b) shows the resulting LLVM IR. Following the LLVM convention, names that start with % are virtual registers, while those that start with @ are global objects. For simplicity, we show an abridged version of the LLVM code that, although not complete, captures the essence of the generated IR. rdb_begin and rdb_end are replaced by calls to two dummy functions that will be removed later. We are now ready to perform the two compilation steps for rdb: code extraction and machine code generation.

**Step 1: Code Extraction.**

This step is performed inside the LLVM IR optimizer. It is shown in Step 1 of Figure 5.6(a). This step, called Extractor, extracts the debug code from the input LLVM IR code, and generates two modules. One is the extracted debug code; the other is the resulting main code. The Extractor runs before any further processing of the input LLVM IR code, so that the next compilation steps are guaranteed to operate on the same LLVM IR as in the original code.
For each debug region, the Extractor generates one debug function, which contains the LLVM code of that region. Any variable or function that belongs to the main code and is accessed in the debug region becomes an argument to the debug function. We call such variables *Debug Arguments*. The Extractor replaces all the references to a debug argument in the body of the debug function with references to the corresponding argument.

Figure 5.4(c) shows the debug module extracted from the example program. It contains one debug function. The debug region accesses three objects that are not defined in the region: variable `c`, function `printf()`, and constant string `.str`. Variable `c` is an input to the function. The `printf()` function comes from the debug code’s libc. Finally, `.str` is a constant that would not have existed if it was not used in the debug function. Hence, it should only be part of the debug code. Thus, the single argument of the function is the address of `c` from the main code when the function is invoked.

The resulting main code is the same as the original program code, except for some markers that are added by the Extractor to establish the necessary relation between the main code and the debug code. There are two type of markers: (1) *Location* markers, which mark the points in the control flow of the main code where the debug functions should be invoked, and (2) *Argument* markers, which mark the variables that are referenced in the debug region and thus have to be passed as arguments to the corresponding debug function. These markers are represented as LLVM intrinsics, which are calls to built-in functions of the compiler (`llvm.rdb.location()` and `llvm.rdb.arg()`, respectively). They will be processed in Step 2. Figure 5.4(d) shows the resulting main code, where the whole debug region has been replaced by intrinsic calls.

The arguments of the markers are used to identify the correct debug code. Specifically, each debug region is assigned a unique integer ID by the Extractor. This ID is passed as the first argument to the corresponding location and argument markers in the main code. In Figure 5.4(d), this is ID 1. To relate these IDs to the debug function names, the Extractor generates a Function Descriptor file that associates an ID to each generated debug function name (Figure 5.4(e)). Using this information, the replay execution will identify the debug function that has to be invoked at a given marked location. In addition, the argument marker (`llvm.rdb.arg()`) for a variable takes two additional arguments: the position of the variable in the argument list of the debug function, and the variable. In Figure 5.4(d), the position is 0 because variable `c` is the only argument of the debug function.
Step 2: Machine Code Generation.

The second \texttt{rdb}-specific compilation step is performed in the CodeGen pass. It is shown in Step 2 of Figure 5.6(a). This step takes the extracted debug and main modules and translates them to machine code. The debug module does not need any special treatment from CodeGen, since it is normal LLVM code. The main module, however, contains the markers that need to be handled. In this step, we need to ensure that the markers do not change the code generation process relative to the original code. It is at this step that the location of the debug arguments in the main code is determined. Generating a location that is different from a variable’s location in the original code will result in an inconsistent execution during replay.

CodeGen removes the argument markers early on — before any code generation activity such as instruction selection or register allocation takes place. In this manner, \texttt{rdb} can guarantee that the machine code generated is the same as for the original code. During the code generation, however, these debug variables are tracked, such that we can know what location has been assigned to each of them. After the machine code is finalized, CodeGen outputs an Argument Descriptor file, which has a descriptor for each debug argument. The descriptor for an argument includes the ID of the function to which the argument belongs, the position of the variable in the argument list of that function, and some information about the class of the variable. The latter allows the replay execution to find the location of the variable in the main program when invoking the debug function.

There are three classes of variables that CodeGen tracks: (1) register-allocated variables, (2) stack-allocated variables, and (3) global variables or functions. For register-allocated variables, the descriptor contains the register name. Stack-allocated variables are described by a \((\text{register}, \text{offset})\) pair; \text{register} is usually one of the stack pointer or frame pointer registers, and \text{offset} is an immediate value to add to \text{register}. Global variables and functions are described as a \((\text{symbol}, \text{offset})\) pair. The desired location is calculated by adding \text{offset} to the location of \text{symbol} in the address space; the latter is found by looking up \text{symbol} in the symbol table of the program.

Figure 5.4(f) shows the argument descriptor file for the example. The first row corresponds to variable \(c\). It belongs to function \_rdb\_func\_1 (ID is 1), it is the function’s first argument (Position is 0), and it is found in the stack at offset -20 from the stack pointer (Info is (\text{SP},-20)).

Finally, location markers, which indicate main-code locations at which debug functions should be invoked, are translated to labels in the code. These labels do not affect the code generation process in any
way. At the end, they become symbols in the symbol table of the generated machine code. The name of the symbol contains the ID of the corresponding debug function as a suffix. This way, the replay execution knows which debug function to call when the execution flow reaches that location.

### 5.4.3 Executing the Debug Code while Replaying

After rdb has generated the main and debug binary modules described above, the second mechanism needed for replay debugging is the ability to replay the execution encoded in the logs while invoking the debug code at the appropriate locations in the code. For this, we need an infrastructure with three functionalities.

First, we need to set up a virtual address space that is shared by the main program and the debug code. However, each of the two needs to have its own instance of the runtime libraries, and use different memory ranges for their code and data (stack, static data, and heap).

Second, the infrastructure needs to replay the application using the recorded input and memory access interleaving logs, injecting inputs and enforcing access interleavings as recorded.

Finally, it should provide the ability to invoke the appropriate debug function with appropriate arguments, without affecting the deterministic replay, when the execution flow of the application reaches a marked location. The required steps involve pausing the replay, setting up a stack frame for the debug function without affecting the main program, transferring the control to the debug code (i.e., invoking the function), and returning the control back to the main code when the debug function completes.

Figure 5.7(a) shows a high-level view of the infrastructure. It contains a replay tool that reads a log, controls application execution, and invokes the debug functions. We build our replay infrastructure augment the Pin-based replay mechanism used in QuickRec. The reason is that, as we discuss next, Pin already provides some of the features needed. In Section 5.5.6 we discuss an alternative replay infrastructure.

**Replay Debugging Using Pin: Rdbtool.**

Pin provides much of the required functionality described. The address space of an application that runs under Pin consists of three parts (Figure 5.7(b)): (1) the application, (2) the Pin infrastructure, and (3) a Pintool, which is a shared library. The Pintool can use Pin’s API to monitor and control the execution of the application. Internally, Pin uses binary instrumentation to implement this. When Pin is invoked, it loads
the Pintool and provides it with a copy of the runtime libraries **libc** and **stdlibc++**. Then, it lets the Pintool analyze the instructions in the application’s code and instrument them. In QuickRec, this Pintool provides **replay** functionality. In **rdb**, we further extend it to provide **replay debugging** functionality, and call it **Rdbtool**.

To replay, we need the application binary, the Rdbtool binary, the libraries of both binaries, and the RnR input and memory logs. The memory log is only required for RnR of multi-threaded workloads, and is in the form of a set of totally-ordered chunks. Prior to starting Pin, we analyze the input log to identify all the memory ranges that are going to be used by the application. This can be done by examining the input log entries for **mmap()** and **brk()** system calls. We then make sure that Pin and the Rdbtool do not use these ranges, to ensure correct replay.

The Rdbtool keeps the debug code and data, and will ensure that the debug code executes when needed. As shown in Figure 5.6(b), the Rdbtool binary is built by compiling together: (1) the code of the core Rdbtool logic (i.e., baseline replay functionality as in QuickRec, plus the invocation of debug functions when execution reaches debug markers), (2) the object files with the extracted debug functions, (3) files with other, non-inlined debug code (explained in Section 5.5.1), and (4) the function and argument descriptor files generated by the modified compiler.

The Rdbtool controls the RnR input and memory logs during replay. To inject application inputs, the Rdbtool instruments system calls, so that it can emulate their results according to the RnR input log. Most
system calls are emulated by injecting their results into the application. Some system calls, however, need to be re-executed to create the appropriate kernel-level state for the application — e.g., memory mapping and thread creation system calls. As for the RnR memory access interleaving log, as chunks replay, a counter counts the instructions executed. When the counter reaches the logged chunk size, the thread’s execution is paused and it looks for the next chunk in the log to execute.

Most importantly, the Rdbtool manages the replay debugging. When the Rdbtool is loaded by Pin, it first searches the symbol table of the main program for symbols that mark code locations at which debug functions should be called. When it instruments the application’s binary, it instruments these code locations to set breakpoints. When execution hits one of these breakpoints, the Rdbtool pauses the replay, and uses the information in the descriptor files to find the address and arguments of the corresponding debug function to call. Then, it calls the function. Note that this function call takes place on the Rdbtool’s stack, rather than on the application’s stack, to avoid changing the application’s memory. Once the debug function completes, the Rdbtool transfers execution to the main program.

### 5.5 Advanced Issues

We now describe several advanced issues in the rdb design. The last two are discussed here for completeness but have not been implemented in the current system.

#### 5.5.1 Debug-Only Functions and Global Variables

In the process of debugging, developers often need to define global objects (variables or functions) for use in the debug code. The definitions of such objects can only be included in the debug binary; including them in the main binary would result in a program that is different from the original program. To ensure this, rdb requires that the developer writes the definitions of such global objects in source files that are linked with the extracted debug code, to form the Rdbtool binary. Such files are shown as the box labeled Non-inlined Debug Code in Figure 5.6(b).

When a global object is accessed in a debug region, the Extractor pass needs to know whether it belongs to the main code or it is a debug-only object. A reference to a debug-only global object is not changed, and is resolved at link time when the support file containing the definition of the object is linked-in. A reference to a global object belonging to the main code is turned into a debug function argument.
In our current implementation, the Extractor makes the decision based on the name of the object. All debug-only global object names are required to have a particular prefix. A more elegant solution would involve using C/C++ attributes for this purpose — e.g., each debug-only global object could be marked with a C/C++ attribute named \texttt{rdb} to make it easy to identify.

### 5.5.2 Event-Driven Debugging

Developers often like to invoke debug code when a certain event happens in the main application — rather than when execution reaches a marked location. This is called event-driven debugging, and is supported in \texttt{rdb} with a certain API. Developers can use this API to associate call-back functions with events, rather than marking the application code. The Rdbtool then adds instrumentation to the application code to detect the occurrence of the events. When an event happens, the associated call-back function is invoked.

There are several events that the developer can ask \texttt{rdb} to monitor. One is the occurrence of a system call. The associated call-back is invoked before or after a system call executes. For example, in some programs, buffer overflow or under-synchronized buffer accesses can result in gibberish program output. By asking \texttt{rdb} to monitor \texttt{write()} system calls, one can identify the code responsible for the bug.

Another event is the call of a function. The associated call-back is invoked when an arbitrary function is called. This is especially useful to monitor library calls in a program.

Finally, another event is reading or writing a certain memory location. This corresponds to the popular “watchpoint” functionality. In this case, the associated call-back function is invoked before or after the program execution accesses the location. This functionality is useful to diagnose bugs such as segmentation faults or buffer overflows. Currently, this functionality is implemented in \texttt{rdb} by monitoring each memory access, and comparing the accessed address to the watched address, and invoking the call-back function if they match. A future implementation will involve using the watchpoint registers provided by the x86 processor hardware.

### 5.5.3 Protecting Against Writes to Main-Program Memory

The debug code should not write directly or indirectly to memory regions of the main program. To enforce this, the Rdbtool can optionally change the access protection of main-program memory regions to read-only prior to invoking a debug function. It then restores the original protections after executing the debug code.
This comes at a performance cost, but detects debug code that violates the read-only-access requirement.

### 5.5.4 Using gdb with Replay Debugging

Pin can be connected to gdb, giving gdb full control over the execution of the application running under Pin. This way, gdb can be used to debug the application as if the debugger was directly attached to the application. This feature is independent of rdb, so it is possible to use gdb even during replay. However, only a subset of gdb features are safe to use in this fashion — namely, those that do not modify the application’s memory content (code or data), such as reading the application’s memory or setting breakpoints.

More complex debugging logic has to be implemented as rdb debug code, to avoid affecting the application state. Examples of such debug logic include adding local, global and dynamic objects in the debug code, adding and executing new code, including if statements, while loops, and function calls/definitions, or creating shadow data structures. The support of such complex debug code is one of the main features that distinguishes rdb from merely using gdb in conjunction with a replay tool (e.g., gdb plus QuickRec).

Still, in an rdb-based debugging scenario, using gdb can be particularly useful for debugging tasks that are not easy to do using inlined debug code, such as back-tracing an application’s stack or single-stepping through the execution.

### 5.5.5 Replay Debugging with Partial Logs

In long-running recordings, the recorded log size can grow very large. To reduce storage requirements, periodic snapshots of the application state could be taken. In this case, when a snapshot is taken, the recorded log up to that point would be purged. Thus, in this environment, the execution would be recorded as an application snapshot plus a partial log that records the rest of the execution.

For rdb to perform replay debugging in such an environment, it would first have to initialize the state of the application using the snapshot; then it could replay the events in the partial log. Since the program being replayed is exactly the same (in terms of both code and data content) as the recorded program, rdb would work correctly after restoring the snapshot.
5.5.6 Replay Debugging without Pin

Section 5.4.3 described how rdb uses Pin to support the second mechanism needed for replay debugging: executing the debug code while replaying the main program. In reality, rdb can be built on top of other replay infrastructures. One of them is replay using OS functionality, as exemplified by Cyrus (Chapter 3) and Scribe [42].

In this case, the role of the Replay Tool in Figure 5.7(a) is played by a modified OS kernel. The OS creates a process, loads into memory the code of the application to be replayed and the libraries it uses, and then starts replaying the application. The OS injects the recorded inputs from the input log (e.g., when the program makes system calls), and enforces the memory access interleavings from the memory log, using mechanisms explained in Cyrus and Scribe [42].

To support replay debugging in this environment, the OS also needs to load the binary for the debug code, and link it with a separate instance of the run-time library. The OS can easily make sure that the application and the debug-code binaries use distinct address ranges. To mark debug locations, the OS can use either hardware or software breakpoints. When a breakpoint is hit, control transfers to the OS. The OS can then calculate the address and arguments of the corresponding debug function using the information in the descriptor files. Then, it sets up a dummy stack in an unused part of the address space, sets the program’s PC to point to the first instruction of the debug function, and transfers control back to user mode. In this way, when the program resumes execution, it will execute the debug function. After the function terminates, the OS transfers control back to the main program.

This technique to invoke user-mode code by the kernel is the same mechanism used in Linux, for example, to invoke signal handlers. These handlers are functions defined in user-mode code that are executed when the kernel receives signals destined for the process.

5.6 An Example of Replay Debugging

To illustrate replay debugging with rdb, we examine a bug in the GNU bc program version 1.06 [28] that crashes the program due to a segmentation fault [29]. This bug is also included in the BugBench bug-benchmark suite [51]. While this bug is not timing-dependent or multithreaded, we examine it because it illustrates many of rdb’s capabilities.
bc is a popular numeric processing program that takes as input a program with C-like syntax and executes it. bc works by first translating its input program to an internal byte-code format (translation phase) and then executing the byte-code (execution phase). In this section, “instruction” refers to a byte-code instruction. Instructions read their operands from an “operand stack” and push the result back on the stack.

We assume that a user was running the bc program on a machine equipped with RnR hardware when the crash happened, and that he/she gave us the resulting RnR log. We now consider the replay debugging process in steps.

![Figure 5.8: Example using rdb for replay debugging: program with a debug region.](image)

**Step 1: Replay to Find Out the Crash Point.** The first step is to find out where the crash happens in the program code. For this, we attach gdb to the replayer (Section 5.5.4) and replay the execution. When the program crashes, we use gdb to analyze its stack frames at the crash point. We conclude that the crash
happens while executing the byte-code (the execution phase). Specifically, it happens in the BRANCH case of a switch statement inside a while loop that processes each instruction `inst` that gets executed (Figure 5.8).

<table>
<thead>
<tr>
<th>Local variable(s) defined in debug code:</th>
</tr>
</thead>
<tbody>
<tr>
<td>depth, temp</td>
</tr>
<tr>
<td>Function(s) defined in debug code:</td>
</tr>
<tr>
<td>_rdb_out_char</td>
</tr>
<tr>
<td>Function(s) defined in run-time library:</td>
</tr>
<tr>
<td>printf</td>
</tr>
<tr>
<td>Variable(s) defined in main code:</td>
</tr>
<tr>
<td>ex_stack, inst</td>
</tr>
<tr>
<td>Function(s) defined in the main code:</td>
</tr>
<tr>
<td>bc_out_num</td>
</tr>
</tbody>
</table>

Figure 5.9: Objects accessed in the debug region.

**Step 2: Replay Again to Print State at the Crash Point.** The next step is to find out why the crash happens. For this, we write a debug region before the crash point that prints `inst` and the contents of the operand stack when `inst` executes. Figure 5.8 shows the debug code inside the `rdb_begin` and `rdb_end` markers. Figure 5.9 shows the objects accessed in the debug code. They include local variables defined in the debug code (depth and temp), functions defined in the debug code (\_rdb\_out\_char, but the definition is not shown), functions from the run-time library of the debug code (printf), variables defined in the main code (ex\_stack and inst), and functions from the main code (bc\_out\_num).

The main program includes the bc\_out\_num function that is used to write numbers to the output. Internally, it first computes the characters that need to be put out and then, instead of directly writing them to the output, it passes each character to a pretty-printing function which is passed to bc\_out\_num as an argument. This second function does the actual output. In the debug code, when we call bc\_out\_num, we cannot pass to it any of the pretty-printing functions defined in the main program, since they eventually call functions from the libc of the main code. This would result in replay failure. Instead, we define an equivalent function in the debug code, called \_rdb\_out\_char, and pass it as an argument to bc\_out\_num.

**Step 3: Identify that the Problem Is Somewhere Else.** Based on the data printed by the debug code, we find that the program crashes because a BRANCH finds an empty operand stack while it expects to find the branch condition on the stack. Now we know that the actual problem is in the translation phase — the
instructions preceding the BRANCH do not produce correct operand stack state. Consequently, we need to examine the input program of bc and find the portion of it that generates the instructions before the branch.

**Step 4: Replay Again to Print State at the New Point.** To obtain the input code that generates the instructions before the branch, we add a new debug region in the translation code. The region prints the input program from bc’s internal data structures before it is translated. After this, we replay the program again and print the code.

**Step 5: Diagnose the Bug.** We compare the output of Step 2 (instructions and stack content) to the output of Step 4 (input program) to find the bug. The input program contains a for loop whose condition is empty. This is equivalent to a true condition, which means that the body of the loop should be executed. Unfortunately, for this pattern, the buggy translator fails to include an instruction that pushes the constant true on the stack, and the subsequent branch instruction crashes.

This example has shown several rdb features, such as: (1) the combined use of gdb and rdb, (2) three deterministic replays of the program with expanding debug instrumentation, and (3) debug regions that use many different types of objects. The replays would be deterministic even for timing-dependent, multithreaded bugs.

### 5.7 Current Limitations and Potential Solutions

We now discuss the main limitations of the current rdb design.

**5.7.1 Adding/Removing Code in the Main Program**

Since rdb targets replay debugging with guaranteed exact replay, it cannot tolerate changes to the main program that are not extracted into the debug code binary. In a debugging process, after rdb has helped diagnose the bug, the code will be patched to fix the defect. Patching involves adding and/or removing code in the program. After the code is patched, rdb cannot be used for replay debugging the resulting program using the original log. This is a fundamental limitation of replay debugging with guaranteed exact replay.

---

1Alternatively, we could ask the user who gave us the log to also provide the input program. However, we can easily regenerate it ourselves.
5.7.2 Supporting Compiler Optimizations

A limitation of the current implementation of rdb is the assumption that compiler optimizations have been disabled. Disabling optimizations results in that the generated LLVM IR and machine code are in direct correspondence with the high-level program. This makes the debug code extraction and code generation processes of Section 5.4 easier to implement.

The difficulty with compiler optimizations is that, because they are applied after the debug code has been extracted from the program, the compiler performs them without being aware of the debug code. Hence, the compiler may optimize away some of the state that the debug code will attempt to access. In general, the compiler may perform optimizations that are invalid in the presence of the debug code.

Figure 5.10(a) shows an example. In this program, character c is read from the input and variable a is set (gray box in the figure). However, a is not used in the main code — it is only used in the debug code. After we extract the debug region, a Dead Code Elimination (DCE) pass will remove the statement in the gray box from the main code as dead code. The DCE optimization has to be performed because it was also performed in the original program recorded in the log. However, this optimization causes the debug code executed during replay debugging to fail.

Figure 5.10: Optimization example: program before (a) and after (b) automatic debug code insertion.

To assess the applicability of rdb to optimized code, we analyzed the optimizations that LLVM performs in its -O2 and -O3 optimization levels. In the next several paragraphs, we present a brief discussion of some of our findings. However, as of the time of this writing, the problem of supporting compiler optimizations in conjunction with rdb is still widely open.
First, we find that some optimizations (e.g., Common Subexpression Elimination (CSE), loop unrolling and inlining) do not affect the validity of the debug code. The reason is that these optimizations do not optimize away the main program state accessed by the debug code. Hence, the compiler can safely repeat such optimizations in the presence of rdb markers.

However, there are other optimizations where the compiler has to optimize based on the knowledge of both the main and debug codes. In some cases, the compiler can automatically generate extra debug code after performing an optimization on the main code. This extra debug code undoes some of the effects of the optimization for the debug code. As an example, consider the code in Figure 5.10(b). After having removed the statement in the gray box in Figure 5.10(a) from the main program, the compiler adds a new debug region (gray box in Figure 5.10(b)) that calculates a in the debug code for later use. With this change, the compiler can optimize the main code exactly the way it optimized the original code, while keeping the debug code valid.

Sometimes, it may be hard or even impossible to undo optimization effects in the above fashion — in particular, when complex pointer-aliasing relations exist between the debug code and the main code. In these cases, the compiler can generate an error message to let the programmer know about the problem. Then, the programmer has a chance to change the debug code to avoid the problem.

In either case, the main difficulty in implementing the approaches presented above is tracking the effect of optimizations on debug code without requiring major re-engineering of existing compiler transformations.

### 5.7.3 Cross-Region Data Sharing

In the current implementation of rdb, it is not possible to define a debug-only local variable in one debug region and use it in another debug region in the same function. This is because we convert each debug region into a separate function. Thus, all of the cross-region data sharing has to happen through debug-only global variables (Section 5.5.1).

This inconvenience can be easily relieved by adding compiler support for automatically converting such local variables to global variables. Using stack-like data structures, it is also possible to support situations (such as recursive function calls) in which multiple instances of the same static debug-only local variable are simultaneously alive. We leave the details of the design to future work.

\[\text{For simplicity, this example presents the changes in C code. In practice, the extra code will be inserted at the level of the LLVM IR or machine code.}\]
5.8 Related Work

In general, debugging involves bug reproduction, diagnosis and fixing. While there are many proposals for using RnR to reproduce bugs (e.g., [6, 20, 21, 26, 30, 41, 46, 59, 63, 72, 75]), very few have tackled the issues of bug diagnosis and fixing.

Some RnR proposals [20, 39, 63, 79] allow limited execution inspection capabilities. Aftersight [20], IntroVirt [39] and Simics Hindsight [79] allow programmers to write code that inspects the state of the program under replay. They all record and replay virtual machines rather than individual applications. As a result, the kind of inspection code they support is different in nature than debug code that can be inlined with main code. In addition, Aftersight and Hindsight keep the debug state in a separate address space than the program being debugged, and IntroVirt does not allow debug code to keep state. Hence, neither provides all of the usability features mentioned in Section 5.3. PinPlay [63] uses Pin [53] for both record and replay. Similar to rdb, it uses Pin’s gdb-connection feature (Section 5.5.4) to let the debugger control and inspect the application’s execution. However, to avoid replay divergence, it can only use the limited set of features explained in Section 5.5.4.

DORA [78] specifically targets bug diagnosis and patch testing using RnR. Its underlying RnR system, SCRIBE [42], uses a modified Linux kernel to record program inputs as well as inter-thread data dependences. Given the logs recorded by SCRIBE, DORA then uses a search-based algorithm to allow “mutable replay” of modified programs, as explained in Section 5.2.2. DORA does not guarantee deterministic replay, and hence, cannot ensure exact debug-time reproduction of non-deterministic events that resulted in a bug. This is a major limitation and affects its usability as a replay debugging tool.

In addition, to reduce the recording overhead, SCRIBE’s approach to recording memory-access interleavings systematically perturbs a program’s shared-memory accesses. Moreover, DORA’s replay of memory interleavings in multithreaded programs relies on SCRIBE’s particular style of recording them. This design choice negatively affects DORA’s usefulness for capturing, reproducing and debugging concurrency-related bugs such as data races and atomicity violations.
5.9 Concluding Remarks

While hardware-assisted RnR has been proposed as a primitive for debugging hard-to-repeat software bugs, simply providing support for repeatedly stumbling on the same bug does not help diagnose it. For bug diagnosis, developers need to modify the code — e.g., by creating and operating on new variables or printing state. Unfortunately, this renders the RnR log inconsistent.

This chapter introduced \( \text{rdb} \), the first scheme for replay debugging that guarantees exact replay. With \( \text{rdb} \), the user interface is the same as in an ordinary bug diagnosis session: the user can read program variables, invoke program functions, create new variables and functions, set watchpoints, and print state. \( \text{rdb} \) uses the log generated by hardware-assisted RnR to always guarantee deterministic re-execution. \( \text{rdb} \)'s operation is possible thanks to two mechanisms. The first one is a compiler mechanism that splits the instrumented application into two binaries: one that is identical to the original program binary, and another that encapsulates all the added debug code. The second mechanism is a runtime one that replays the application and, without affecting it in any way, invokes the appropriate debug code at the appropriate locations. This chapter described an implementation of \( \text{rdb} \) using LLVM and Pin, and discussed an example of how \( \text{rdb} \)'s replay debugging is used to diagnose a real bug.
Chapter 6

Conclusion

Many problems in designing and programming reliable computer systems can significantly benefit from the ability to examine a past execution. Record and Deterministic Replay (RnR) is a powerful primitive that allows one to do just that. It is typically a two phase process: in the first phase (record) enough information about an execution is logged which is then use in the second phase (replay) to re-create the execution.

RnR has been shown to enable diverse use-cases in areas of computer science and engineering such as program debugging, computer systems security and high-availability scenarios. As such, it has interested researchers from different disciplines including compilers, software engineering, operating systems and computer architecture.

Proliferation of multiprocessor shared-memory computers has arisen much interest in enabling RnR for these systems. The major difficulty facing us in this context is capturing the non-deterministic effect of shared-memory communication. As discussed in Chapter 1, hardware-assisted memory race recording (MRR) is a promising approach to capture this source of non-determinism efficiently using special hardware support.

In this thesis, we investigated RnR with hardware-assisted MRR from two different angles: (i) building practical RnR solutions, and (ii) using them to enable effective program debugging as one of the most widely cited use-cases of RnR.

For (i), we presented QuickRec, Cyrus and RelaxReplay. QuickRec was mostly a prototyping effort to investigate the sources of complexity and performance issues in building a hardware-assisted RnR system. Its implementation involved modifying the Linux kernel and a multi-processor Intel Pentium-based system (for recording) and an Intel Pin-based system (for replaying programs). QuickRec proves that it is possible to build a hardware-assisted RnR system with relatively low design complexity and that it can indeed achieve low recording overhead. Cyrus and RelaxReplay are novel hardware designs to improve QuickRec’s hardware to, respectively, enable faster replay (through replay parallelism) and support processors
with memory models more relaxed than Intel’s.

For (ii), we presented rdb—a solution for effective replay-based debugging in the context of existing RnR systems. We showed that plain RnR only enables bug reproduction and is hardly enough for bug diagnosis. We then presented a system combining novel compiler techniques and replay mechanisms to enable effective bug diagnosis while guaranteeing deterministic replay.

There are many directions in which this work can be extended. Below, we name a few:

1. As mentioned in Section 5.7, debugging optimized programs using an rdb-like system is still an open problem.

2. Debugging is but one use-case of RnR. Much research is needed to understand how one can effectively use RnR in other areas such as security and high-availability. An RnR solution suitable for these use-cases will certainly have different requirements than one that can only accommodate debugging. In addition, it is very likely that, as is the case for debugging, plain RnR would not be sufficient for enabling these use-cases and it would need to be augmented with other mechanisms in order to be useful.

3. MRR hardware is likely to have many applications other than RnR. It is a powerful, and low overhead, monitoring hardware and can provide much information about the shared-memory behavior of a program as it executes. It is conceivable that this new source of information could enable novel profiling, performance debugging and dynamic optimization techniques. As of today, this is a widely open research area.

4. Current hardware-assisted MRR techniques mostly assume a cache-coherent memory substrate. Proliferation of non-coherent or locally-coherent-globally-incoherent memory systems being popularized by heterogeneous on-chip systems necessitates devising new MRR techniques that could support such designs.
References


[34] Derek R. Hower and Mark D. Hill. Rerun: Exploiting Episodes for Lightweight Memory Race Recording. ISCA, June 2008.


[38] Yanyan Jiang, Tianxiao Gu, Chang Xu, Xiaoxing Ma, and Jian Lu. CARE: Cache Guided Deterministic Replay for Concurrent Java Programs. ICSE, June 2014.


[70] Xuehai Qian, He Huang, Benjamin Sahelices, and Depei Qian. Rainbow: Efficient Memory Dependence Recording with High Replay Parallelism for Relaxed Memory Model. HPCA, February 2013.


