

RACE RECORDING FOR MULTITHREADED DETERMINISTIC REPLAY
USING MULTIPROCESSOR HARDWARE

by

MIN XU

徐旻

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Electrical Engineering)

at the

University of Wisconsin–Madison

2006

© Copyright by Min Xu 2006

All Rights Reserved

To Rong

To Cyrus, Jun and my parents

Abstract

Multithreaded deterministic replay has important applications in cyclic debugging, fault tolerance, intrusion analysis and more. *Memory race recording* is a key technology for multithreaded deterministic replay. This dissertation proposes a new race recording algorithm and describes a novel implementation of a race recorder based on multiprocessor cache coherence mechanisms. As a result of the new algorithm and the novel implementation, the new race recorder is significantly more efficient and less expensive than existing memory race recorders. Notably, the recorder simultaneously achieves several desired features:

- **Long recording** by reducing the recorder log size to around one byte per thousand instructions.
- **Always-on recording** by reducing the runtime overhead to less than 2%.
- **Inexpensive recording** by reducing the timestamp memory size (which is different from the log size) to approximately 24 kilobytes per processor.
- **Broad applicability** by supporting programs with data races and by supporting multiprocessor systems with both the Sequential Consistency and the Total Store Order (TSO) memory consistency models.

Our improvements stem from several ideas: (1) **a method of creating artificial dependencies** that allows reduction and compression in the log, yet still allows parallel replay; (2) **a method of approximating timestamps** that

allows significant reduction in the chip area cost; (3) **a method of hardware coherence piggybacking** that enables race recording with extremely low runtime overhead, yet still supports race recording with programs with data races; (4) **a method of order-value-hybrid recording** that supports race recording on multiprocessor systems with the TSO memory consistency model.

We evaluate the recorder with full-system simulation of a Chip MultiProcessing (CMP) system and commercial workloads. Our results support that the recorder can be *always-on* and the log size is around one byte per kilo instructions (55 to 180 KB per (2 gigahertz) processor per second).

This dissertation contributes to the race recording literature with three results. First, we show that a recorder can create artificial dependencies that imply other dependencies and enable a compact log. We develop a heuristic algorithm that creates these dependencies without creating replay deadlocks. This is a significant contribution, because this log reduction problem has been researched extensively and the best existing reduction method is optimal (if the recorder does not create artificial dependencies). Second, we show that a recorder needs to store only a small subset of the timestamps to achieve the most benefits of storing all timestamps for all memory blocks. This is again significant, because this optimization reduces the chip area cost of our hardware recorder and is applicable to software recorders. Finally, we demonstrate that a hardware race recorder can be implemented with little additional hardware and extremely low runtime overhead.

Acknowledgments

Not many graduate students get the luxury of having two advisors. On top of this, my advisors were Professor Mark D. Hill and Professor Rastislav Bodík, who gave me the opportunity to witness and learn from two wise, yet distinctly different, minds.

I am also deeply indebted to my tireless committee: Professor Remzi H. Arpaci-Dusseau, Professor Mikko H. Lipasti, Professor Barton P. Miller, Professor David A. Wood and my advisors. Their guidance has made this dissertation more focused.

I am very fortunate to be a member of the Wisconsin Multifacet research group, co-led by Professor Mark D. Hill and Professor David A. Wood. I thank Dr. Milo M. K. Martin and Dr. Daniel J. Sorin for showing me how they grew into great researchers. I thank my fellow graduate students Dr. Alaa Alameldeen, Bradford Beckmann, Jayaram Bobba, Ross Dickson, Brian Fields, Daniel Gibson, Pacia Harper, Jarrod Lewis, Michael Marty, Carl Mauer, Bhavesh Mehta, Kevin Moore, Michelle Moravan and Luke Yen for creating and sharing an excellent group atmosphere. I am grateful to my officemates Dr. Philip C. Roth, Dr. Valentin Puente-Varona and Andy Phelps for memorable discussions on many topics. I thank Carrie Pritchard and Caitlin Scopel for helping me improve my writing skills.

I enjoyed the time talking with respectful researchers from other groups and areas. An incomplete list of them are Professor Gurindar S. Sohi, Professor James R. Goodman, Professor James E. Smith, Professor Jeffrey F. Naughton, Professor

Eric Bach, Professor Susan Horwitz, Professor Ben Liblit, Professor Marvin Solomon, Dr. David Bacon, Dr. Trey Cain, Dr. Jong-Deok Choi, Dr. Joe Emer, Dr. Peter Hsu, Dr. Konrad Lai, Dr. Kevin Lepak, Dr. Chris J. Newburn, Dr. Nhon Quach, Dr. Ravi Rajwar and Shai Rubin.

Studying in the US is an exciting cultural experience. I thank my fellow Chinese friends for sharing this experience with me. They are Chang, Jichuan; Chen, Lei; Professor Ding, Chen; Fu, Wenyin; Dr. Guo, Hongfei; Dr. Hu, Shiliang; Li, Jian; Professor Li, Tao; Li, Yunpeng; Dr. Luo, Gang; Sha, Tingting; Shi, Yixin; Su, Lixin; Wang, Hao; Dr. Wang, Yuan; Zhang, Su; Professor Zhou, Huiyang and Zhou, Pin.

This work is supported in part by the National Science Foundation (NSF), with grants CCF-0085949, CCR-0093275, CCR-0105721, EIA/CNS-0103670, EIA/CNS-0205286, CNS-0225610, CCR-0243657, CCR-0324878, CCR-0326577. This work has also been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCHC020056. This work is also made possible with donations from IBM, Intel, Microsoft, and Sun Microsystems.

Those who I want to thank the most will not mind that I put them the last. It is their true love that made my life worth living. They — my dear Rong, my dear son Cyrus, my dear Dad and Mom and my dear brother Jun — supported me unconditionally. Although my brother Jun Xu fought me all the time, he taught me how to be more relaxed and have fun. My parents Jianhua Xu and Liping Chen cultivated my interests in engineering and taught me the necessary disciplines. My son Cyrus Zhaoyang Xu is a gift from heaven and gave me tears of joy. My wife Rong He unleashed my full potential with her love, care, encouragement and inspiration, which keep me wanting her more.

Contents

Abstract	i
Acknowledgments	iii
Contents	v
List of Figures	xi
List of Tables	xiii
1 Introduction	1
1.1 Motivation	3
1.1.1 Applications of Deterministic Replay	3
1.1.2 Importance of Multithreaded Software	4
1.1.3 Components of Multithreaded Deterministic Replay	5
1.1.3.1 Checkpointing and Input Recording — Not Our Focus	6
1.1.3.2 Race Recording — The Focus of This Dissertation	7
1.1.3.3 Replaying — Not Our Focus	8
1.2 Problem: Simultaneously Achieving Multiple Features	9
1.2.1 Small Log Size	9
1.2.2 Low Runtime and Memory Overheads	10
1.2.3 Broad Applicability	10
1.3 Solution: a New Algorithm and a New Implementation	11
1.3.1 Reducing Log Size — Chapter 3	11
1.3.2 Reducing Runtime Overhead — Chapter 4	13
1.3.3 Reducing Memory Overhead — Chapter 5	14
1.3.4 Broadening Applicability	14

1.4	Contributions of the Dissertation	16
1.5	Differences with Our Earlier Work	16
1.6	Dissertation Organization	17
2	Background and Related Work	19
2.1	Two Recording Philosophies	19
2.2	Race Recording Algorithms	23
2.2.1	Partial-order Algorithms	24
2.2.2	Total-order Algorithms	27
2.2.3	A Comparison of Partial-order and Total-order Algorithms	30
2.3	Race Recorder Implementations	31
2.3.1	Software Based Implementations	31
2.3.2	Hardware Based Implementations	33
2.4	Hardware Assistance for Debugging	35
3	Reduce the Log Size: RTR Algorithm	39
3.1	Terminology	40
3.1.1	Conflicts	40
3.1.2	Memory Races	41
3.1.3	Race Recorder, Dependence Log and Race Replayer	43
3.2	Unoptimized Recording Algorithm	43
3.2.1	The Algorithm and Its Pseudo Code	44
3.2.2	Brief Overview of the Replay	46
3.3	Netzer's Transitive Reduction (TR) Algorithm	47
3.3.1	Log Size Reduction	48
3.3.2	Pseudo Code	49
3.4	New Regulated Transitive Reduction (RTR) Algorithm	51
3.4.1	Log Size Reduction	51
3.4.2	Replay Correctness and Performance	53
3.4.3	Pseudo Code	55
3.5	Correctness the RTR Recording Algorithm	58

	vii
3.5.1	Successful Replay58
3.5.2	Proof Sketch59
3.6	Extending RTR to Supporting the TSO Memory Model61
3.6.1	TSO model and its Impact on Race Recording62
3.6.1.1	The Impact of LoadM Instructions63
3.6.1.2	The Impact of the LoadB Instructions64
3.6.2	Extending the Unoptimized Algorithm from SC to TSO65
3.6.3	Extending the TR Algorithm from SC to TSO70
3.6.4	Extending the RTR Algorithm from SC to TSO71
3.6.5	Race Recording on PC and Other Relaxed Consistency Models ..74
3.7	Discussion: Generalized RTR75
3.7.1	Generalized RTR75
3.7.2	ROLT as a Special Case of the Generalized RTR78
4	Reduce the Runtime Overhead: Hardware Assistance 81
4.1	A Simplified Multiprocessor System82
4.1.1	The Cache Coherence Protocol83
4.2	Unoptimized Hardware Race Recorder85
4.2.1	Instruction Count (IC) and Timestamps85
4.2.2	Conflict Detection87
4.2.2.1	Requester Detection versus Responder Detection88
4.2.2.2	Word Conflict versus Block Conflict89
4.2.3	Dependence Logging91
4.2.4	Put-it-together: the Hardware Algorithm91
4.3	Hardware TR Recorder92
4.3.1	Pairwise-TR93
4.3.2	Put-it-together: the Hardware Algorithm94
4.4	Hardware RTR Recorder96
4.4.1	The Sliding Windows of IC Stride96
4.4.2	Put-it-together: the Hardware Algorithm96

	viii
4.5 Race Recording on Realistic Protocols	99
4.5.1 Finite Caches	99
4.5.1.1 Missing Timestamps	100
4.5.1.2 Missing Coherence Transitions	100
4.5.2 Implicit or Combined Response	104
4.5.3 Owned and Exclusive States	106
4.5.4 Nonatomic Transactions	107
4.5.5 Multilevel Caches	108
4.6 Miscellaneous Hardware Implementation Issues	108
4.6.1 Send and Receive Observations	108
4.6.2 Out-of-Order Execution and Hardware Prefetching	109
4.6.3 Unordered Interconnect	110
4.6.4 Integer Wrap-around	110
4.7 Reducing the Timestamp Communication Overhead	111
4.8 Open Problems	112
4.8.1 Recorder Virtualization	112
4.8.1.1 Virtualization and Coscheduling Virtual Machines	113
4.8.2 Multilevel Coherence	114
4.8.3 Simultaneous MultiThreading (SMT) and Shared L1 Caches ..	115
5 Reduce the Memory Overhead: Timestamp Approximation	117
5.1 Coupled Timestamp Memory	119
5.2 Decoupled Timestamp Memory	120
5.3 Reducing Memory Overhead	122
5.3.1 Set/LRU Timestamp Approximations	122
5.3.1.1 Timestamp Approximation Without LRU	124
5.3.2 Partial Timestamps	125
5.3.3 Storing Both the Read and the Write Timestamps	126
5.3.4 Partial Tags	126
5.3.5 Timestamp Approximation and Software Recorders	127

5.4	Two-level Timestamp Memory	127
6	Evaluation Methodology	129
6.1	Baseline CMP System	129
6.2	Determinizer: a CMP-based Race Recorder	130
6.3	Simulation Methodology	131
7	Evaluation Results	135
7.1	A Design Space of Determinizer/CMP	135
7.2	Performance of Determinizer/CMP	139
7.2.1	Hardware Cost	139
7.2.2	Log Size	140
7.2.3	Runtime and Bandwidth Overheads	141
7.3	Benefits of RTR and Set/LRU	142
7.4	Scalability of Determinizer/CMP	145
8	A Qualitative Discussion of Race Replay	147
8.1	Sequential Replay	147
8.1.1	Algorithm	147
8.1.2	Implementations and Performance	149
8.2	Parallel Replay	150
8.2.1	Algorithm	150
8.2.2	Implementations and Performance	151
9	Conclusion and Future Work	153
9.1	Future Research Directions	154
9.1.1	Race Recording Algorithms	154
9.1.2	Race Recording Implementations	154
9.1.3	Race Replay	155
9.1.4	Deterministic Replay Applications	156
	Bibliography	157

Tables of the Simulation Results	x
	165

List of Figures

1.1 An example race	7
1.2 Dissertation organization	17
2.1 A taxonomy of recording for deterministic replay	22
2.2 Parallel slices based recording	26
2.3 Reconstruct Of Lamport Timestamp (ROLT)	27
3.1 Notations of multithreaded execution and conflict (dependence).	42
3.2 Unoptimized recording algorithm	44
3.3 Netzer's transitive reduction algorithm	48
3.4 Further reduce the log with stricter dependencies	52
3.5 Compact the log with vectorized dependencies	53
3.6 A replay deadlock due to an overly strict dependence.	54
3.7 An example TSO execution (loadM)	63
3.8 An example TSO execution (loadB)	64
3.9 TR and TSO executions	71
3.10 RTR and TSO executions	72
3.11 Synchronizing Lamport scalar clock	79
4.1 A simplified multiprocessor system	83
4.2 The MSI cache coherence protocol	84
4.3 The IC register and the read/write timestamps	86

4.4	Conflict detection in the MSI protocol	88
4.5	Three cases in word-conflict-versus-block-conflict	90
4.6	Pairwise-TR versus vector-TR	93
4.7	The MTS registers	94
4.8	The sliding window registers	96
4.9	Missing conflicts due to writebacks and nonsilent replacements	101
4.10	Missing conflicts due to the lack of acknowledgements	104
4.11	A compact format of the piggybacked IC and Timestamp	111
5.1	The coupled timestamp memory	119
5.2	The decoupled timestamp memory	121
5.3	The Set/LRU approximation	123
6.1	Determinizer/CMP	131
7.1	Impacts of the TSM size and the number of timestamps per block	136
7.2	Impacts of the width of partial timestamps	137
7.3	Impacts of the TSM associativity and the approximation method	138
7.4	Log size of the selected configuration	140
7.5	Runtime and bandwidth overheads of the selected configuration	141
7.6	RTR versus TR	142
7.7	Set/LRU versus perfect TSM.	143
7.8	Set/LRU versus Current IC.	144
7.9	Scalability in terms of number of processors	145

List of Tables

3-1	The Unoptimized Algorithm	45
3-2	Transitive Reduction (TR) Algorithm	49
3-3	Regulated Transitive Reduction (RTR) Algorithm	56
3-4	The Unoptimized Algorithm on TSO	68
4-1	State and Actions for Processor j with Unoptimized Recording	92
4-2	State and Actions for Processor j with TR	95
4-3	State and Actions for Processor j with RTR	97
6-1	Simulation Parameters	132
6-2	Commercial Workloads	133
7-1	Hardware Cost of the Selected Configuration	139
8-1	Sequential Replay Algorithm	148
8-2	Parallel Replay Algorithm	150
A.1	Varying the Timestamp Memory Size — Apache	165
A.2	Varying the Timestamp Memory Size — OLTP	166
A.3	Varying the Timestamp Memory Size — SPECjbb	167
A.4	Varying the Timestamp Memory Size — Zeus	168
A.5	Varying the Width of Partial Timestamps — Apache	169
A.6	Varying the Width of Partial Timestamps — OLTP	169
A.7	Varying the Width of Partial Timestamps — SPECjbb	170

A.8	Varying the Width of Partial Timestamps — Zeus	170
A.9	Varying the Associativity and the Approximation Method — Apache	171
A.10	Varying the Associativity and the Approximation Method — OLTP	172
A.11	Varying the Associativity and the Approximation Method — SPECjbb	173
A.12	Varying the Associativity and the Approximation Method — SPECjbb	174
A.13	Log Size of the Selected Configuration	174
A.14	Runtime and Aggregated Bandwidth with and without the Recorder	175
A.15	RTR versus TR: the Log Size with Infinite TSM	176
A.16	The Log Size of a Recorder with Current IC Timestamp Approximation	176
A.17	Scalability of the Recorder (From 2-core to 16-core)	177

Chapter 1

Introduction

This dissertation studies *Race Recording*: a technique that *records* nondeterministic interthread instruction orderings in a multithreaded program execution. Race recording is a key enabler for *Multithreaded Deterministic Replay*, which recreates a logically identical multithreaded execution after an interesting execution is observed. Multithreaded deterministic replay has several important present and future applications in cyclic debugging [51, 58], fault tolerance [59, 36, 62], intrusion analysis [17] and more [43]. These applications can boost the productivity of both software developers and end users. However, lack of an effective and inexpensive (more precisely defined in Section 1.2) race recorder has been a critical bottleneck for achieving multithreaded deterministic replay, thus compromising the productivity of software developers and end users of multithreaded programs. The thesis of this dissertation is that effective and inexpensive race recording is possible and we support this thesis by designing and evaluating an effective and inexpensive race recorder on Chip MultiProcessing

(CMP) hardware. We are optimistic that effective and inexpensive race recording can ease the ongoing industrial transition from mostly sequential computing (single-threaded, single-core) to more parallel (multithreaded, multi-core) computing, which have compelling benefits both in performance and power.

We begin this chapter by motivating our study of race recording (Section 1.1). Section 1.1.1 lists applications that depend on, or can benefit from, deterministic replay. Section 1.1.2 shows the importance of future multithreaded hardware and software. These two subsections reveal an urgent need for multithreaded deterministic replay. Section 1.1.3 demonstrates four components of multithreaded deterministic replay. Among the four components, race recording is critical and is the sole focus of this dissertation. Section 1.2 presents multiple desired features for race recording. Existing recorders cannot however achieve these features simultaneously. Section 1.3 offers an overview of our solution, which allows our new recorder to simultaneously achieve the desired features. Section 1.4 presents our contributions to the literature. Finally, we differentiate this dissertation with our earlier published research in Section 1.5 and give a roadmap to the rest of the dissertation in Section 1.6.

Throughout this dissertation, we focus on multithreaded programs for their importance in commercial software. Nevertheless, the key ideas in this dissertation can be extended to other parallel programming models, such as message passing in scientific software.

1.1 Motivation

This section offers our motivations for studying race recording.

1.1.1 Applications of Deterministic Replay

Deterministic replay creates an execution that is *logically equivalent* to an original execution of interest. We say two executions are logically equivalent if they contain the same set of dynamic instructions, each dynamic instruction computes the same result, and the two executions compute the same final values in the memory. Deterministic replay is useful for several current and future applications.

Cyclic Debugging. Deterministic replay is often used in cyclic debugging [51]. Without it, a developer may not be able to effectively debug her program, because bugs may not faithfully reappear even in a controlled debugging environment (*i.e.*, the same program input and the same hardware/software configuration).

Fault Tolerance. Deterministic replay can be used in fault tolerance in two ways: (1) replay-based redundancy and (2) replay-based failure recovery.

Deterministic replay can be used to ensure that we detect hardware faults by detecting the differences between an original execution and a (redundant) replay execution, such as in ExtraVirt [36]. In failure recovery, an identical replay execution is needed to reconstruct the most up-to-date program state, after a catastrophic event occurs [59, 62].

Intrusion Analysis. Deterministic replay can also benefit intrusion analysis, such as in ReVirt [17]. In ReVirt, replays need to be deterministic so that intrusions

can be found and analyzed, similar to the requirements of cyclic debugging.

Data Recovery. In the future, deterministic replay can be used as a data recovery method, because data can be observed and retrieved from a replay. For example, when a word processing program is replayed, a user can recover the text he typed into the editor but never saved.

Predictable and Fast Synchronization. In the future, deterministic replay can be used to enable predictable and fast synchronization in multithreaded real-time kernel code [14]. Real-time codes have strict timing deadlines. Nondeterminism in thread synchronization may contribute to unpredictable performance. If a kernel code has static control flow (*i.e.*, the control flow of the code does not change with respect to the program input data, such as the FFT code studied by Chow *et al.* on the CELL processor [13]), deterministic replay of a recorded execution scheduling can guarantee predictable synchronization. Furthermore, if the data addresses are fixed in the static control flow, explicit (software) synchronization primitives can be replaced by (hardware) deterministic replay. Thus, potentially finer-grain and faster synchronization is attainable. (Strictly speaking, this is not deterministic replay because the replayed execution may compute different data. However, our race recording approach still applies to this application.)

1.1.2 Importance of Multithreaded Software

Recently, hardware vendors have started to offer multithreaded chips, using *Chip-MultiProcessing* (CMP) and *Simultaneous-MultiThreading* (SMT). Multithreaded hardware brings compelling benefits in terms of performance and power

to those programs that can be multithreaded [7].

But, will software vendors adopt multithreaded programming? We and other researchers [7, 66] are optimistic for the following reasons: (1) multithreaded programming can benefit both server and desktop programs and (2) active research (including this dissertation) seeks to make multithreaded programming easier. On servers, commercial workloads have plenty of *Thread Level Parallelism* (TLP) [35] that can be harvested through multithreading. On desktops, although desktop programs have modest TLP [20], multithreading can be useful in several scenarios, such as improve user response time [19], expressing independent control flow [50] or parallelizing multimedia decoding and speech recognition [7]. More importantly, low-latency communication makes multithreading more attractive on CMP than on existing *Symmetrical MultiProcessing* (SMP) and *Cache-Coherent NonUniform Memory Accessing* (CC-NUMA) systems [7].

Therefore, as multithreading spreads, multithreaded deterministic replay will become an urgent need. We now examine how to replay multithreaded programs.

1.1.3 Components of Multithreaded Deterministic Replay

Deterministic replay is divided into two phases: the recording phase and the replay phase. In the recording phase, necessary information about the original execution is recorded. This information is then used in the replay phase to make the replay execution logically equivalent to the original execution. To completely bound the replay execution, three types of information are needed: (1) the pro-

gram's initial state; (2) the program's inputs and (3) interthread instruction orderings (some of which are races). The tasks of recording these three types of information are commonly called checkpointing, input recording and race recording, respectively. We now discuss these three recording tasks and the replay task in more detail.

1.1.3.1 Checkpointing and Input Recording — Not Our Focus

Checkpointing is the task of taking a consistent snapshot of the program's internal state, such as the register and memory values. The snapshot state is called a checkpoint, which can provide the initial state for a replay execution. Checkpointing mechanisms are especially useful, because in many cases we desire to start a replay from the middle of the original execution. Researchers have studied extensively various checkpointing schemes [10, 16, 25, 28, 34, 53, 63, 65]. This dissertation does not focus on this problem.

Input Recording is the task of recording nonreproducible program inputs for deterministic replay. Program inputs are caused by external events, which are not always reproducible. For example, if a program calls the UNIX system call `gettimeofday()`, the return value differs from time to time. If the program execution depends on the return value, the replay can be nondeterministic. On the other hand, some program inputs, like read-only files, are reproducible as long as the same disk drive used in the original execution is available during replay. Researchers have proposed several input recording schemes [9, 15, 46, 73]. This dissertation does not focus on this problem.

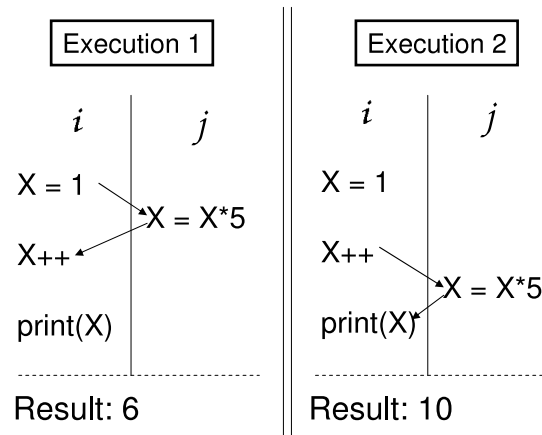


Figure 1.1 An example race. In both execution 1 and 2, the same multithreaded program is run. Although both thread *i* and *j* execute the same dynamic instructions in the same intrathread order (the program order), the interthread order (represented by the arrows) is different, which caused the program to compute different results in execution 1 and 2. Without special considerations, this nondeterminism can cause deterministic replay to fail for multithreaded programs.

1.1.3.2 Race Recording — The Focus of This Dissertation

Race recording is the task of recording interthread orderings of dynamic instructions. Races are the root cause of nondeterminism in multithreaded programs. Two types of race exist: general race (also known as synchronization race) and data race. General race pertains to multithreaded programs intended to be deterministic and causes nondeterministic executions. Data race pertains to nondeterministic programs containing critical sections and causes nonatomic execution of critical sections. General race and data race are more formally defined by Netzer and Miller [48].

Most multithreaded programs contain, at least, general races. Therefore, most multithreaded programs are nondeterministic. For example, Figure 1.1 shows a multithreaded program that contains two threads, *i* and *j*. In two different executions 1 and 2, *i* and *j* “race” to read and write a shared variable *x*. Depending

on which thread wins the “race”, the program computes different results. Unfortunately, the race outcome changes from time to time, depending on the machine configuration and initial state (*e.g.* the cache replacement policy and the initial cache contents). Therefore, races can fail multithreaded deterministic replay. A popular solution is *race recording*, which strives to log a sufficient (but desirably minimal) amount of information of the original execution, so that the replay execution can recreate the same races based on the logged information.

We devote this dissertation to solving the race recording problem, because existing race recorders are unsatisfactory. Race recording is a hard problem. As we demonstrate in Chapter 3, race recording can generate large logs and incur high runtime and memory overheads. Existing race recorders are unsatisfactory because they can be ineffective (generating large logs or having limited applicability) or expensive (high runtime overhead or large memory overhead). Section 1.2 presents the problem of existing recorders in more detail.

1.1.3.3 Replaying — Not Our Focus

After the recording phase, a *replayer* is needed for the deterministic replay. The replayer needs to (1) setup the correct program initial state; (2) deliver correct program input at the correct time; (3) recreate a logically equivalent inter-thread orderings in the multithreaded execution; and (4) support any special features needed during the replay, such as memory inspection for the debugging purpose.

This dissertation does not focus on the replayer, which in itself may warrant

another dissertation. A good recorder should be an important first step: *i.e.*, if a good recorder is made, a good replayer will follow. In Chapter 8, we briefly discuss how our recorder design impacts the replayer design.

1.2 Problem: Simultaneously Achieving Multiple Features

The goal of this dissertation is to develop an effective and inexpensive race recorder. Race recorders should be *effective* by generating minimal log and by not imposing special requirements (*e.g.*, data race freedom or uniprocessor systems) on the recorded executions. Race recorders should also be *inexpensive* by having low runtime overhead and small memory overhead on the recorded executions. Unfortunately, the existing recorders [5, 12, 21, 31, 33, 47, 57, 60] are unable to *simultaneously* meet these requirements. The existing recorders are discussed in more detail in Chapter 2.

1.2.1 Small Log Size

Some existing recorders [5, 31] generate large logs. However, small log size is desirable because large logs are difficult to store, transfer and use. Large logs consume significant network bandwidth when transferred and significant memory and disk storage spaces when stored. The size of the log is roughly determined by (1) frequency of races, which tends to increase with future CMP hardware with fast intercore communication and (2) the length of the recorded execution. In practice, a recorder is often given a fixed log size budget. In other words, the smaller the log size (per unit length of execution), the longer executions the recorder can record.

1.2.2 Low Runtime and Memory Overheads

One of the existing recorders [47] can achieve small log size through log reduction, but the recorder incurs prohibitive runtime overhead and large memory overhead (*e.g.*, hardware cost of a hardware recorder). This recorder is based on software instrumentation and tracing. Because threads often interact frequently through reading and writing the shared memory, this recorder needs to add extra tracing instructions (which incurs significant runtime overhead) and needs to remember a large amount of information of the memory accesses (which incurs significant memory overhead). The overheads prevent this recorder from being used online, *i.e.*, while the program is in production.

1.2.3 Broad Applicability

Some existing recorders [5, 12, 21, 57, 60] can achieve low runtime and small memory overheads, but only at the cost of limiting their applicability. Among these recorders, some assume that the multithreaded programs being recorded are data race free [21, 57], run on only uniprocessor machines [12, 60], or require a bus-based system [5]. These assumptions limit the applications of the race recorders.

Furthermore, none of the existing race recording algorithms handle weak memory consistency models in multiprocessor systems. These algorithms require the underlying multiprocessor system to support *Sequential Consistency* (SC) memory models. However, most existing multiprocessor systems support weaker memory models. As an example, the *Total Store Order* (TSO) memory model,

which relaxes write-to-read orderings to the shared memory, is well defined by the SPARC architecture [70]. To broaden the applicability of race recording, we believe new race recorders should support at least the TSO model, because the popular x86 architecture assumes a memory model that is similar to TSO [32].

1.3 Solution: a New Algorithm and a New Implementation

To the best of our knowledge, none of the existing race recorders meets all these features in Section 1.2 simultaneously (more detail in Chapter 2). The thesis of this dissertation is that it is possible for a new recorder to simultaneously achieve all the features. We develop a new recorder that has small log size, low runtime and small memory overheads, and broad applicability. More specifically, the new recorder uses a new recording algorithm that generates smaller log size than the state-of-the-art recording algorithm. The new recorder relies on a hardware implementation to achieve extremely low runtime overhead. The new recorder employs a new method of timestamp approximation to avoid remembering the large amount of information for memory accesses, thus significantly reducing the memory overheads. The new recorder is able to handle programs that contain data races and supports multiprocessor systems (as opposed to only uniprocessor systems) with either the SC or the TSO memory models. This section overviews the novel ideas that allow the new recorder to achieve these features.

1.3.1 Reducing Log Size — Chapter 3

We propose a new recording algorithm that reduces the log size. To the best

our knowledge, *Transitive Reduction (TR)* [47] generated the smallest log among existing partial-order race recording algorithms. To obtain faithful replay, a partial-order recorder needs to log the order of conflicting instructions (or simply called *conflicts*), which are two instructions that access the same memory location from different threads and at least one thread writes. TR reduces the log size significantly by logging only those conflicts that are not implied by others. TR is optimal if the recorder logs only the conflicts. However, we find a recorder can further reduce the log size if the recorder creates and logs artificial dependencies between instructions that are not conflicting. Based on this observation, we propose a new partial order recording algorithm—*Regulated Transitive Reduction (RTR)*. RTR creates *stricter dependencies*, which are stricter because enforcing them is sufficient but not necessary for replaying the correct order of two conflicting instructions. RTR creates stricter dependencies in a way that the dependencies are *vectorizable*. Vectorized dependencies are reminiscent of vectorized computations—applying the same type of computation to multiple data. Vectorization allows our new recorder to (losslessly) compress the log by applying the same pattern to multiple dependencies. By adding stricter dependencies and vectorizing them, RTR in effect regulates the replay execution (hence the name regulated transitive reduction). We show in Chapter 7 that the RTR algorithm reduces the log size by an average of **28%** over the TR algorithm. The log size growth rate is on average one byte per kilo instructions per thread, or 150 KB/core/second.

1.3.2 Reducing Runtime Overhead — Chapter 4

To reduce the prohibitive runtime overhead of a software race recorder, we developed a hardware-assisted race recorder. Hardware recorders are preferable for several reasons. First, hardware recorders are likely to have much lower overhead. For software recorders, major overhead is not from logging, but from tracing all the shared memory reads and writes. As we show in Chapter 4, the tracing can be done efficiently in hardware. Second, transistors are getting cheaper and software bugs are getting more expensive, thus it makes economic sense to use hardware to assist deterministic replay and debugging in general. Finally, hardware recorders, which are more transparent (*i.e.*, OS-independent) than the software recorders, can be easier to use. The primary drawback of the hardware recorders is their inflexibility after being implemented.

Although hardware-assisted race recorders have been proposed previously [5], our proposal is suitable to a wider range and a newer class of multiprocessor systems. In our proposal, we piggyback race recording onto the hardware cache coherence mechanisms. A significant fraction of functions needed by race recording is already part of the hardware cache coherence mechanisms. Therefore, piggybacking race recording onto the coherence mechanisms incurs little extra hardware cost. In addition, because race recording operations are completely off the critical paths in the cache coherence operations, hardware race recording incurs extremely low runtime overhead. We show that this piggybacking approach incurs little additional hardware in Chapter 4 and less than 2% runtime overhead in Chapter 7.

1.3.3 Reducing Memory Overhead — Chapter 5

We develop a *Set/LRU Timestamp Approximation* algorithm that significantly reduces the memory overhead of our new race recorder. The significant memory overhead is a result of the conflict detection, where (conceptually) each physical memory location needs to carry extra timestamps that represent the Instruction Count (IC) of the dynamic instructions that last read and/or wrote the memory location. The conflicts are then detected and represented by using these timestamps. However, we observe that, because of the transitive reduction, the majority of the conflicts are not logged by the recorder. If a conflict is not logged, then remembering the precise timestamps for the conflict is an overkill and approximated timestamps are enough. The new *Set/LRU* timestamp approximation method allows our recorder to significantly reduce the memory overhead by only remembering timestamps for a small number of memory locations. In Chapter 7, we show that our recorder requires a small 12 KB timestamp memory per processor.

1.3.4 Broadening Applicability

By design, our recorder does not assume data race free programs or uniprocessor systems. Thanks to the hardware assistance, our recorder can afford to trace every memory operation, regardless of whether the operation is a local data operation, a shared data operation, or a synchronization operation. Not only does this design strategy enable the recorder to handle programs with data races, but it also makes the recorder easy to use, because users do not need to annotate pro-

gram synchronization for the recorder. By piggybacking the cache coherence hardware, our recorder handles multithreaded programs running on multiprocessor hardware. In Chapter 4, we show that our race recorder can be implemented on a variety of multiprocessor systems, including but not limited to a CMP system with directory-based cache coherence protocol.

In Chapter 3, we propose an order-value-hybrid method for handling the TSO memory consistency model, in addition to the SC memory model. Existing order-based race recording methods assume each thread is replayed in program order. Modern microprocessors may execute a thread out-of-order, but as long as the system supports SC memory model, the out-of-order execution is logically equivalent to an in-order execution. On systems that support only weak memory models, such as the TSO (x86-like) memory model, an order-based race recorder may record a set of conflicts that are not reproducible if the replay is in-order (Out-of-order replay is non-trivial to implement and makes it harder to apply transitive reduction during the recording. Out-of-order replay is beyond the scope of this dissertation.). We propose an order-value-hybrid recording method that, by recording additional values of some memory read operations, a replayer can use the values to faithfully replay a multithreaded execution, even if the execution violates the SC memory model. For the sake of presentation, we assume the SC model when presenting the basic ideas of our recording algorithm in Chapter 3. In Section 3.6, we extend our recording algorithm to support the TSO model.

1.4 Contributions of the Dissertation

This dissertation contributes to the race recording literature with the following results.

- We show that a recorder can create artificial dependencies that imply other dependencies. We develop a heuristic algorithm that creates these dependencies without creating replay deadlocks. This is a significant contribution, because this log reduction problem has been researched extensively and the best existing reduction method is optimal (if the recorder does not create artificial dependencies).
- We show that a recorder needs to store only a small subset of the timestamps to achieve the similar effects of storing all timestamps for all memory locations. This result is again significant, because it reduces not only the chip area cost of our hardware recorder but also potentially memory overhead of future software recorders.
- We demonstrate that a hardware race recorder can be implemented with little additional hardware and extremely low runtime overhead.
- We propose an order-value-hybrid recording method to handle multiprocessor systems with the TSO memory model.
- We evaluate our race recorder with full-system simulation of a CMP system and commercial workloads.

1.5 Differences with Our Earlier Work

Parts of this dissertation have been published in conferences. In particular,

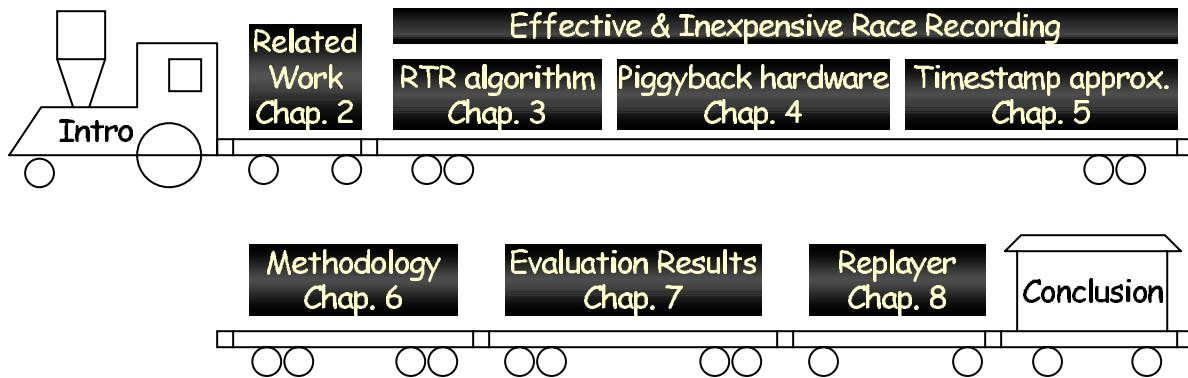


Figure 1.2 Dissertation organization. After this introduction, we present background and related work of race recording. After that, we present details of our effective and inexpensive race recording solutions, which include the RTR (Reduced Transitive Reduction) recording algorithm, the cache coherence hardware assistance and the Set/LRU timestamp approximation. We then evaluate our recorder on a CMP system with commercial workloads. We finally discuss race replayers and offer a conclusion.

the idea of piggybacking race recording (with the Transitive Reduction algorithm) onto the hardware cache coherence mechanisms has been published in ISCA-30th [73]. In that paper, we developed a full-system hardware recorder, the Flight Data Recorder (FDR), which included checkpointing, input recording and race recording. The ideas of the RTR algorithm and the Set/LRU timestamp approximation have been submitted for publication [72]. In the submission, we developed a CMP-based hardware race recorder. The design and evaluation results in this dissertation modestly extend those in the submission.

1.6 Dissertation Organization

The remaining eight chapters of the dissertation are organized in the way pictorially shown in Figure 1.2. Chapter 2 presents background and related work of race recording. Chapter 3 to Chapter 5 dives into our solutions of an effective and inexpensive race recorder. Chapter 6 presents the implementation details of the new race recorder and our evaluation methodology. Chapter 7 presents our evalu-

ation results. Finally, we discuss race replayers in Chapter 8 and offer a conclusion in Chapter 9.

Chapter 2

Background and Related Work

In this chapter, we review the background and related work of race recording. Section 2.1 compares two fundamentally different recording philosophies for deterministic replay. Section 2.2 classifies a variety of existing race recording algorithms with a taxonomy. Section 2.3 compares the pros and cons of different implementation strategies of race recorders. Section 2.4 closes the chapter by surveying recent hardware proposals of race detection and debugging, which share some common characteristics with our hardware race recorder.

2.1 Two Recording Philosophies

From an instruction-set-architecture point of view, deterministic replay of a program execution requires that the same sequences of dynamic instructions of each individual thread are repeated. (In the rest of this dissertation, unless otherwise stated, the term “instruction” refers to “dynamic instruction”.) There are two different philosophies in achieving this goal: *value-oriented* and *order-oriented*.

In value-oriented recording, the input and output values of instructions are viewed as the “essence” of a program execution. By recording these values, deterministic replay can be achieved because we can recreate exactly what happened in the original execution for each dynamic instruction. It is important to note that the output values of an instruction include the value of the Program Counter (PC) register, which determines the next instruction follows the instruction.

In order-oriented recording, ordering between instructions is the “essence” of a program execution—values are merely the results of instructions that execute in certain order, *e.g.* the program order. By recording the ordering between instructions, deterministic replay can be achieved, because we can recreate the values of the program executions by forcing the instructions to execute in the same order.

The two philosophies complement each other in recording for deterministic replay. Value-oriented recording tends to record large amounts of data, because some values are recorded even though they can be regenerated by the execution itself. Order-oriented recording can generate a much smaller log, because both redundant values and redundant orderings are not recorded. On the other hand, value-oriented recording allows flexible *replay scope*—the set of instructions (threads) that have to be replayed together. For example, with value-oriented recording, one can replay each individual thread of a multithreaded execution in isolation. However, order-oriented recording requires all threads to be replayed together. In the *Parallel Program Debugger* (PPD), Miller and Choi used a replay scope called *emulation blocks* (e-blocks) [42] to enable flowback analysis (similar to reverse execution). E-blocks are segments of program code, such as subrou-

tines. The e-block is a good example of balancing the replay scope and the amount of information to be recorded, *i.e.*, using e-blocks enables efficient recording, because the values defined and only used within the e-blocks are not logged.

The nature of the data to be recorded and the desired replay scope affect which recording philosophy should be chosen. For example, value-oriented recording is often used in checkpointing and input recording, because the values often are *not* regenerable by the replayed execution and it is often not desirable to increase the replay scope to include the producers (*e.g.*, another computer on the other end of the internet) of these values. On the other hand, order-oriented recording is likely better for race recording, because the large amounts of data recorded by value-oriented recording is regenerable, and because the desired replay scope includes multiple threads. (In fact, users often need to know which thread generated which value.)

Figure 2.1 shows a taxonomy of the recording methods for deterministic replay, which again are classified into value-oriented and order-oriented. This dissertation focuses on race recording, where the order-oriented method is commonly used.

An order-oriented race recorder can generate either a *partial-order* or a *total-order* of an original program execution for a successful replay. Section 2.2.1 discusses three existing partial-order recorders: Instant Replay [31], *Transitive Reduction* (TR) [47] and Parallel Slices [5]. Among these recorders, Instant Replay and TR form the foundation of our proposed partial-order recording algorithm—*Regulated Transitive Reduction* (RTR)—in Chapter 3.

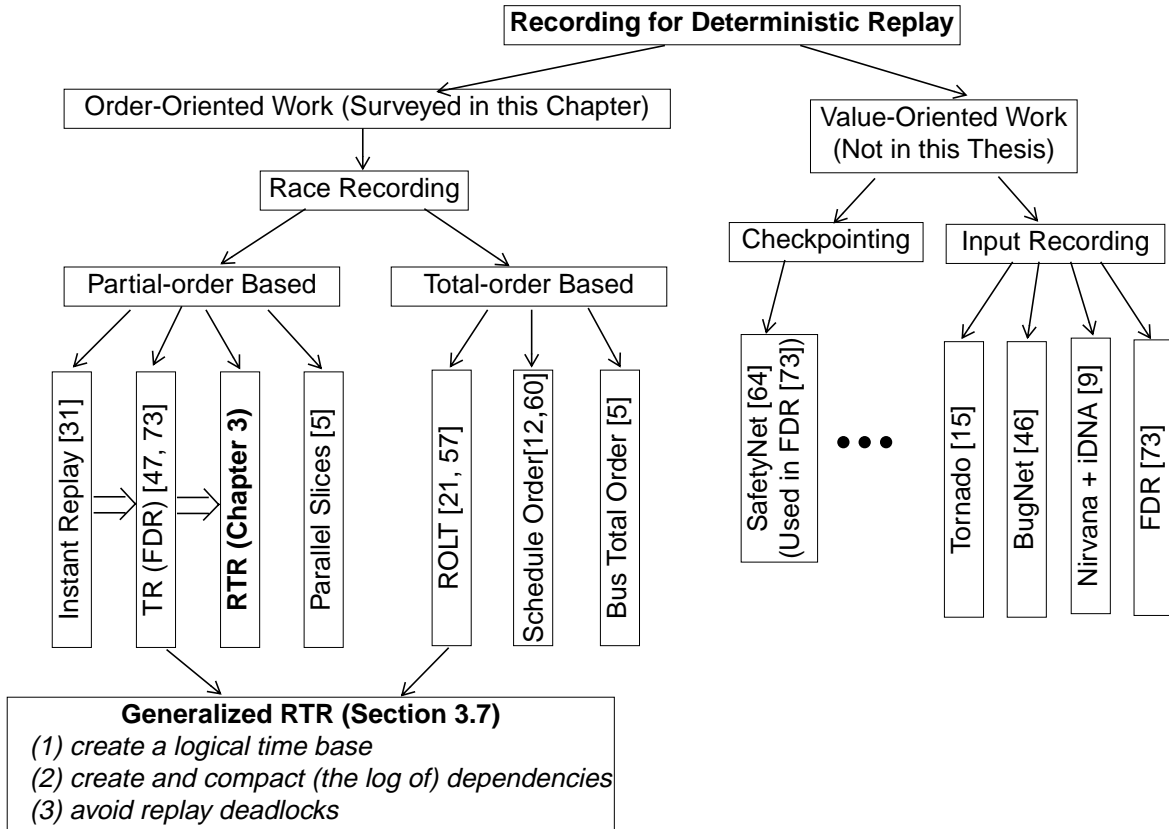


Figure 2.1 A taxonomy of recording for deterministic replay. Recording methods can be divided into value-oriented and order-oriented. This dissertation focuses on (order-oriented) race recording, which can be divided into partial-order based and total-order based. For partial-order based race recording, three existing approaches are discussed in this chapter. Our newly proposed approach is an improved version of the partial-order based recorder. For total-order based race recording, three existing approaches are discussed in this chapter. Finally, a generalized version of our newly proposed approach unifies the two types of order-oriented recording algorithms. Future order-oriented race recorders can be designed within this general framework. For completeness, we show value-oriented recording methods are used in checkpointing and input recording. We offer no further discussion on these recorders.

Section 2.2.2 discusses three types of total-order recorders: *Reconstruction Of Lamport Timestamp* (ROLT) [21, 57], Scheduler Order [12, 60] and Bus Total Order [5]. Unlike the partial-order recording algorithms, the total-order recording algorithms allow only sequential replay. Therefore, except in this chapter, we do not further investigate the total-order algorithms in this dissertation.

In Chapter 3, we present an algorithm framework generalized from RTR. The framework unifies the RTR and the ROLT algorithms by providing three common steps in designing new race recording algorithms.

2.2 Race Recording Algorithms

For single-thread programs, the *program order* defines a total order over all dynamic instructions in the executions, *i.e.*, for any pair of dynamic instructions, one of the two instructions executes before another. Program order is the most intuitive way to execute a single-thread program and all existing machines create an illusion to the programmer that his single-thread program is executed in program order.

On the other hand, for multithreaded programs, program order alone is not enough to define a total order over all dynamic instructions. In particular, given two dynamic instructions from two different threads, it is possible that the two instructions execute in series (a logical order between the two exists), or in parallel (no logical order exists). Therefore, the interthread order is merely a partial-order. To this end, a partial-order recorder can record multithreaded executions.

At the same time, multithreaded programs execute on machines that impose additional constraints on the interthread order. For example, on *Sequential Consistent (SC)* multiprocessor systems, an SC total-order is (implicitly) maintained over all dynamic instructions of the multithreaded execution. This total-order is compatible with the program order of each thread. To this end, a total-order recorder can record multithreaded executions by recording the SC total-

order.

This section surveys both types of the recorders.

2.2.1 Partial-order Algorithms

Instant Replay [31] tackled the partial-order recording problem straightforwardly, *i.e.*, it records *all* interthread ordering. Interthread ordering is more commonly known as conflicts [61], which are classified into (1) *read-after-write (RAW)* conflicts; (2) *write-after-read (WAR)* conflicts and (3) *write-after-write (WAW)* conflicts. Instant Replay augments each shared memory object with a version number. This version number is increased after each write access (during both the recording and the replay phases). Each thread records versioning information in its own log file when accessing the shared memory objects. In this way, all three types of conflicts are recorded by Instant Replay. During replay, Instant Replay ensures that the same version of the data is accessed by all threads.

Transitive Reduction [47], proposed by Netzer, took a slightly different approach to record conflicts. Netzer gave each dynamic instruction within a thread an *Instruction Count (IC)*. The IC numbers all dynamic instructions within a thread execution according to the program order. The conflicts are then detected and represented by pairs of ICs. In this way, conflicts are not much different from the program orders, which can also be represented by pairs of ICs. It is important to note that the recorder does not need to explicitly log the program orders, which are automatically regenerated during the in-order replay. The conflicts and the program orders together form a directed graph, whose number of

arcs can be significantly reduced by transitivity. As a result, a large fraction of the conflicts do not need to be recorded, because the rest of the recorded conflicts and the program orders will transitively recreate these reduced conflicts during replay. This *Transitive Reduction* (TR) technique is proven correct by Netzer by enumeration and induction, which show the replay executions are identical with and without TR.

Both Instant Replay and Transitive Reduction algorithms focus on recording only conflicts. In this dissertation, we propose an improved partial-order recording algorithm, *Regulated Transitive Reduction* (RTR) (Chapter 3). RTR frees itself from recording only the conflicts and creates artificial interthread orders, which we call dependencies. These dependencies can enable more transitive reduction in the log than Netzer's TR algorithm. By creating the dependencies in an optimized way, RTR further reduces the log size by compactly logging the dependencies. We defer the details of RTR until the next chapter.

Parallel Slices [5] algorithm proposed by Bacon and Goldstein is another partial-order recorder. Figure 2.2 shows that parallel slices are groups of dynamic instructions from each thread. Figure 2.2a shows the allowed and disallowed conflicts between slices: a dynamic instruction from a slice can be ordered after another dynamic instruction from a previous slice, but not the same slice or a future slice. These rules restrict the way that we can cut an execution into slices. Figure 2.2b shows a slicing plan for an execution. As one can verify from the example, the slicing plan is not unique. Bacon and Goldstein proposed a runtime heuristic to compute the slices. Figure 2.2c shows that in some cases, Parallel

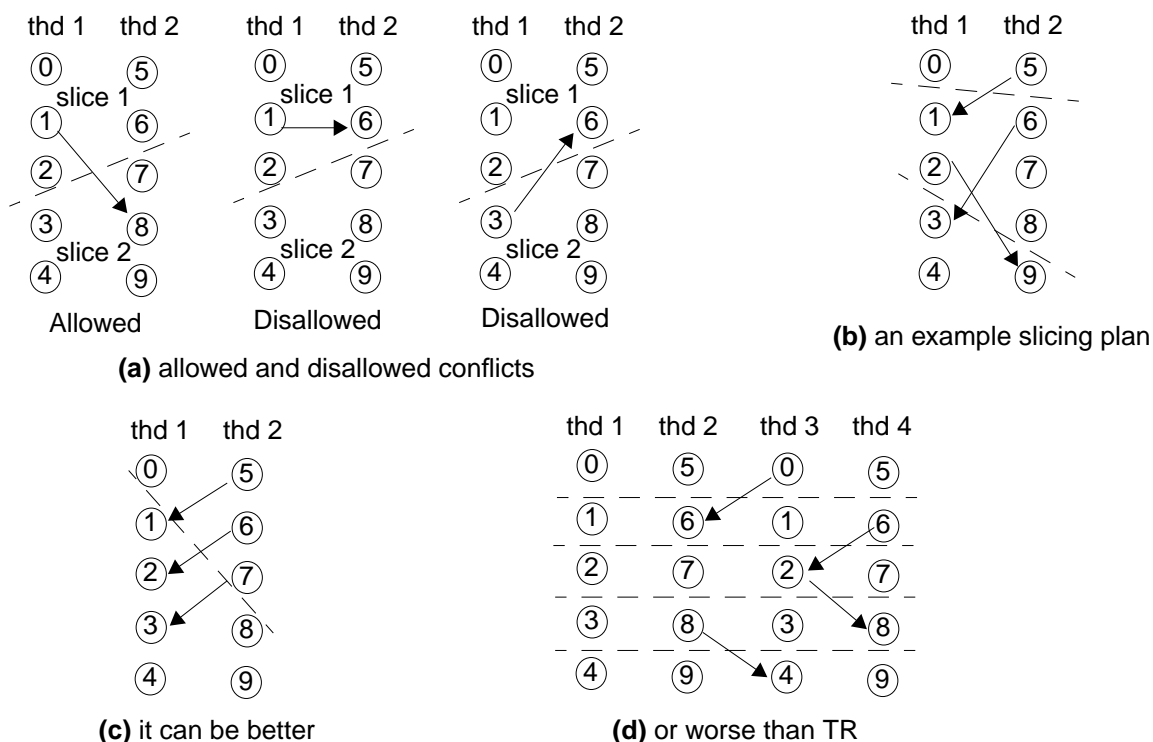


Figure 2.2 Parallel slices based recording. The circles denote dynamic instructions. (a) Allowed and disallowed conflicts between slices. (b) An example slicing plan. (c) In this case, parallel slices can be better than TR. (d) In this case, parallel slices can be worse than TR.

Slice based recording can generate a log that is smaller than TR's log, because it essentially creates a stricter dependence between instruction seven and instruction one to replace the three conflicts, which are recorded by TR in Figure 2.2c. However, in practice, Bacon and Goldstein's heuristic can perform worse than TR, because the parallel slices are forced to group independent instructions across all threads at runtime. Better (perhaps offline) algorithms may exist in finding the optimal slicing plan, but this is beyond the scope of this dissertation.

As we show in Section 3.6, all these partial-order algorithms require the SC memory model.

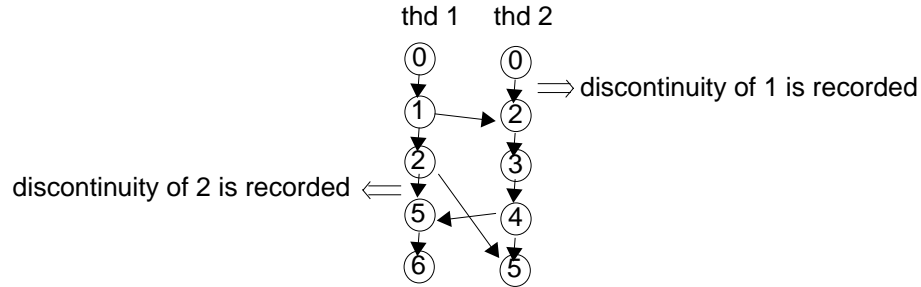


Figure 2.3 Reconstruct Of Lamport Timestamp (ROLT). Dynamic instructions are given Lamport scalar timestamps. Normally, instructions within a thread are assigned with consecutively increasing timestamps. However, when conflicts happen, an instruction gets its timestamp from the maximum timestamp of its preceding instructions. For example, the second instruction of thread 2 gets timestamp 2 (rather than 1), because it is ordered after the second instruction of thread 1, which has timestamp 1. The fifth instruction of thread 2 gets timestamp 5 because it is ordered after the fourth instruction of thread 2, which has timestamp 4. The recorder then records the discontinuity in the timestamps within each thread. Finally, the dynamic instructions that have the same timestamp are ordered by the thread ID. Thus, a total order is established and recorded.

2.2.2 Total-order Algorithms

Reconstruction Of Lamport Timestamp (ROLT) [33] algorithm is a total-order recording algorithm used by RecPlay [57] and JaRec [21]. ROLT uses Lamport Scalar Clock [30], which is a way to give a total-order to *events* in distributed computing. (Events are similar to dynamic instructions in multithreaded execution.) Figure 2.3 shows how ROLT constructs and records a total-order. First, dynamic instructions are given Lamport scalar timestamps. Normally, instructions within a thread are assigned with consecutively increasing timestamps. However, when conflicts happen, an instruction gets its timestamp from the maximum timestamp of its preceding instructions. For example, the second instruction of thread 2 gets timestamp 2 (rather than 1), because it is ordered after the second instruction of thread 1, which has timestamp 1. The fifth instruction of

thread 2 gets timestamp 5 because it is ordered after the fourth instruction of thread 2, which has timestamp 4. The recorder then records the discontinuity in the sequence of timestamps within each thread. Finally, the dynamic instructions that have the same timestamp are ordered by the thread ID. Thus, a total order is established and recorded. This total order is used to deterministically replay the original execution. Because the total order is an SC order, ROLT requires an SC memory model.

It is interesting to note that ROLT shares some common features with the Parallel Slice recording algorithm. In Parallel Slice, if we treat every slice from each individual thread as a “super-dynamic-instruction”, then we can use Lamport timestamps to order them just as it is done in ROLT. The Parallel Slice algorithm can be optimized using ROLT, so that the recorder logs only the slice boundaries and the discontinuity of the Lamport timestamps. After this optimization, one does not need to explicitly record which slices can be executed in parallel, because this information can be obtained from the Lamport timestamps of the slices.

Schedule Order [60] is another total-order recording algorithm used by Russinovich and Cogswell. Assuming multithreaded programs run on a single processor machine, a total-order of all dynamic instructions is created by the decisions made by the thread scheduler. This allows the recorder to record the multithreaded execution by logging the schedule order created by the thread scheduler. As long as the same schedule order is recreated during replay, we can deterministically replay the same multithreaded execution.

DejaVu [12] builds on a similar idea of the schedule order. By tracing so called critical events (including all synchronization operations), DejaVu constructs a logical thread schedule for a multithreaded execution. Consequently, DejaVu is independent of the underlying scheduler implementation of a system. Like Russinovich and Cogswell's recorder, DejaVu is implemented on a uniprocessor. It is possible to use DejaVu on a multiprocessor system, although much greater overhead is expected due to the fact that every read and write to the shared memory is counted as a critical event. The multiprocessor DejaVu algorithm shares some common characteristic with ROLT.

Bus Total Order [5] is a total-order recording algorithm proposed by Bacon and Goldstein. This algorithm exploits the total-order property of a bus-based cache-coherent multiprocessor system. In such a system, the memory functions as if it is attached to only one processor at any given moment. All processors serialize their accesses to the memory. Bacon and Goldstein's recorder works by recording such serial order of memory accesses via snooping the memory bus. In particular, the IC (instruction count) of each bus transaction is sent to the recorder. The recorder then constructs and logs a total order of the multithreaded execution. During replay, the total order is recreated by forcing all threads to serialize in this total order. Bacon and Goldstein's algorithm requires the SC memory model, because the recorded total order needs to be compatible with the program order.

2.2.3 A Comparison of Partial-order and Total-order Algorithms

One of the most important differences between the partial-order and the total-order methods is the replay parallelism allowed by the recording algorithms. For partial-order methods, replay can be done in parallel or sequential mode, as long as the recorded partial-order is somehow recreated. For total-order methods, replay can be performed only in sequential (or in lockstep parallel) mode, which tends to significantly slow down the replay speed. Furthermore, total-order recorders may require special recording conditions to be met, namely uniprocessor or bus-based systems. This can narrow the applicability of total-order recorders.

Nevertheless, total-order recording is interesting for its simplicity and intellectual insights. For example, the ROLT algorithm in Section 2.2.2 is simple, yet generates small logs, because only the (rare) timestamp discontinuities are recorded. In fact, ROLT inspired our new design of the RTR algorithm (Chapter 3) and we will show that ROLT is a special case of a generalized RTR algorithm. Similarly, the Schedule Order based recording algorithm, although applicable to only uniprocessor systems, provokes deeper questions, such as “How to reduce a parallel execution (a partial-order) to a total-order (in fact, the SC order) execution with minimum context switches, so that a race condition can be created with minimal steps?”. Answering these questions should help programmers better understand multithreaded executions.

2.3 Race Recorder Implementations

This section surveys the background on software and hardware implementations of race recorders, as well as hardware assistance of debugging in general.

2.3.1 Software Based Implementations

Most existing race recorders are implemented in software. Software implementations are more flexible than hardware implementations, but software implementations can incur significant runtime and memory overheads to the recorded executions. To overcome the overhead problem, researchers have made different trade-offs.

The lowest runtime overhead in a software recorder is achieved by sacrificing recorder applicability. For example, RecPlay [57] incurs, on average, 2.1% runtime overhead in the recording phase by (1) supporting only data race free executions to be replayed; and (2) using a just-in-time instrumentation mechanism specific to the Solaris [40] operation system to instrument synchronization primitives. By assuming data race free executions, RecPlay is able to avoid monitoring the vast majority of memory accesses and focuses on only recording and replaying synchronization races. Because the execution is data race free, deterministic replay is achieved. However, because data race detection is an important application for race recorders, relying on data race freedom for deterministic replay is a significant limitation to data race detection. In addition, identifying synchronization on-the-fly can be difficult when the source code is unavailable [74].

Similarly, by supporting only the multithreaded executions on uniprocessor

systems, Russinovich and Cogswell's schedule order based recorder [60] incurs, on average, 10% runtime overhead at recording. About half of the overhead is from instrumenting multithreaded executions to keep track of a software instruction counter [41]. Another half of the overhead is due to the extra logging functionality added to the Mach 3.0 OS scheduler. As an increasing number of systems ship with multicore processors, supporting only uniprocessor systems is likely insufficient.

In contrast to the previous two OS specific implementations, JaRec [21] and DeJaVu [12] are two recorders implemented in Java virtual machines. JaRec uses Java Virtual Machine Profiler Interface (JVMPi). When multithreaded Java programs are loaded into the virtual machine, JaRec instruments the synchronization classes through JVMPi. Therefore, no source code changes are needed for the programs to be recorded and replayed. Similar to RecPlay, JaRec also assumes data race free executions. DeJaVu modifies the runtime system of a Java virtual machine to trace the critical events. These critical events are essentially various types of synchronization (`monitor_enter`, `monitor_exit`, `thread_start`, `thread_stop`, `thread_resume`, *etc.*). By tracing these events, DeJaVu computes a logical schedule. In this way, its implementation is decoupled from the physical scheduler of the JVM. This makes DeJaVu more portable than Russinovich and Cogswell's OS scheduler based recorder. Because JaRec and DeJaVu are more general than RecPlay and Russinovich and Cogswell's recorder, they also have higher overheads. JaRec incurs about 80% slowdown on macro-benchmarks. DeJaVu incurs about 70% slowdown on synthetic benchmarks and about 35%

slowdown on SPLASH kernels.

In terms of replay performance, Russinovich and Cogswell's recorder performs the best because it simply needs to enforce the same scheduling plan as recorded. The slowdown is comparable to the recording slowdown, which is around 10%. DeJaVu's replay is also simple because of its logical thread schedule based recording. The replay slowdown is around 40%. RecPlay and JaRec, the two ROLT based recorders, replay much slower than the schedule order based recorders. RecPlay and JaRec incur as much as 570% and 300% slowdown in replay.

Instant Replay [31] and Netzer's transitive reduction based recorder [47] are the most general among the software recorders discussed: (1) they support multithreaded executions that contain data races; and (2) they support multithreaded executions on multiprocessor systems. Unfortunately, the runtime overheads of the recording and replay have not been reported. According to our own experiences on those types of software recorders that trace all shared memory reads and writes, the slowdown is likely to be extremely high. An order of magnitude slowdown is common for commercial workloads, which have intense memory read and write activities.

2.3.2 Hardware Based Implementations

Software recorders often face the dilemma of sacrificing applicability or incurring prohibitive runtime overhead, and thus hardware based recorders have been proposed.

Bacon and Goldstein's race recorder [5] is the first to use hardware assistance.

It is implemented on a bus-based cache-coherent multiprocessor system. The recorder snoops the bus and a subset of the bus transactions are recorded using the total-order and partial-order algorithms discussed in Section 2.2. For the total-order recorder, the system is augmented with a memory page status table, an instruction count table and a log buffer. These components are centralized and are attached to the centralized bus. For the partial-order recorder, two extra components—an event buffer and a schedule queue—are added. Because the logging is done in hardware, the recorder incurs extremely low recording slowdown. Because the total-order recorder does not trace memory accesses on a per block basis, it incurs a small and constant memory overhead. The partial-order recorder, however, incurs larger memory overhead with the event buffer. The replay overhead has not been reported in this case.

The main drawbacks of this recorder are (1) the recorder generates large logs because around 50% of the bus transactions are recorded; and (2) the centralized bus-based multiprocessor systems have poor scalability and poor area/performance/power characteristic [29].

Our earlier work — the *Flight Data Recorder* (FDR) [73] — solves the hardware recording problem from a rather different angle than Bacon and Goldstein’s recorder. FDR is a hardware based race recorder that uses the transitive reduction algorithm [47]. As we will present in greater depth in Chapter 6, FDR is based on a directory-based cache-coherent multiprocessor system. Unlike Bacon and Goldstein’s recorder, FDR records events that are corresponding to only a small fraction of coherence transactions. FDR piggybacks its recording function-

ality onto the hardware cache coherence mechanisms. This piggybacking significantly reduces its hardware cost (memory overhead). FDR is implemented in a distributed fashion, which is more scalable. Finally, FDR's race recorder has been adopted by BugNet [46] as the race recording subsystem.

2.4 Hardware Assistance for Debugging

To close this chapter, we list some other existing work that proposed using hardware to assist debugging. This work is not directly related to our hardware race recorder. But some techniques are similar, such as augmenting hardware caches with extra timestamps and the conflict detection mechanism.

Data Race Detectors. Several hardware based data race detectors have been proposed. Data races are conflicting memory accesses that are not ordered by explicit synchronization in the program. Netzer and Miller [48] formally defined data races and showed that complete and sound data race detection is a NP-hard problem. Nevertheless, dynamic data race detection, which finds data races in a given execution (as opposed to in all possible executions of a program), has drawn great interest in the research community.

Min and Choi [44] proposed to use multiprocessor hardware in dynamic data race detection. They augment cache coherence hardware and give cache blocks logical timestamps that represent synchronization “era” (with respect to `fork/join/barrier` synchronizations). If conflicting data accesses are found within a synchronization era, a data race is found. Due to potential memory overhead, the detector does not support pairwise synchronization (*e.g.*, `mutex` and

semaphore). Furthermore, the detector assumes the compiler places at most one shared object in each cache line. Otherwise, it can miss shared memory accesses, which can introduce false positives and false negatives in the data race detection.

The detector proposed by Richards and Larus [56] does not suffer from this problem because the detector can monitor every shared memory access. This is enabled partially by implementing the detector on a more flexible software-managed *Distributed Shared-Memory* (DSM) system.

More recently, ReEnact [55] aims to provide online debugging—online data race detection and partial repairing—with Thread Level Speculation (TLS) hardware. ReEnact augments the TLS hardware so that the system can roll back longer speculative executions, which allows ReEnact to detect and debug program errors manifested within this speculative execution window. ReEnact supports data race detection by attaching vector timestamps to each cache blocks to capture the causality between so called “epochs”. If memory accesses from different epochs are unordered with respect to any synchronization, a data race is found.

While it provides a rich set of features, the complexity of ReEnact is higher than other hardware assisted debuggers. ReEnact uses multiversion buffers and vector timestamps, which cause considerable runtime and memory overhead. One of the ReEnact authors later proposed CORD [54] to address these complexity and performance issues, in the cost of potentially missing data races (false negatives). CORD uses a bus-based CMP system with private L1/L2 caches. It incurs negligible runtime overhead (on average 0.4%) and 19% cache memory overhead.

CORD sacrifices data race detection accuracy to reduce hardware cost and complexity. CORD strives to report no false positives. CORD uses a small number of timestamps for each cache block to approximate the full access history. This removes the need to have multiversion buffers, but causes the detector to miss data races. Furthermore, CORD uses scalar timestamps, rather than vector timestamps, to capture the causality of memory accesses, which again causes the detector to miss data races.

Finally, the *Serializability Violation Detector* (SVD) [74] is our earlier work that is closely related to data race detection. SVD focuses on detecting concurrency bugs that may cause program errors at runtime. Unlike other data race detectors, which require the programmer to annotate the synchronization code in her programs, SVD automatically infers the dynamic instruction groups (called Computational Units, or CUs) that may need to be executed in a serializable [52] fashion. Then, by detecting when the serializability is violated, SVD can report the violations to the programmer, or trigger automatic rollbacks of the potentially problematic executions.

The SVD algorithm is expensive if implemented in software. In order to use SVD online, hardware assistance is likely required. Future hardware SVD design can draw some ideas from the hardware race recorder in this dissertation. These ideas include augmenting the cache (Chapter 4), piggybacking onto the coherence protocols (Chapter 4) and approximating the timestamps (Chapter 5).

TLS-based iWatcher. As a general debugging framework, *Intelligent Watchpoint* (iWatcher) [76] is developed based on the TLS hardware support. In hard-

ware, iWatcher associates programmer-specified monitoring code with some triggering memory locations. When these memory locations are accessed by a program, the monitoring code is executed using an on-the-fly spawned hardware thread. TLS hardware is then used to guarantee the sequential semantics between the spawned thread and the main thread. The conflict detection mechanisms in the TLS hardware are similar to those used in hardware race recording. The benefit of iWatcher is that it can watch a large number of memory locations simultaneously and potentially perform nontrivial program safety checks, without significantly impacting the speed of the main thread.

In summary, hardware assistance of race recording for deterministic replay and debugging can enhance productivity. As hardware becomes faster and cheaper, adding these “exotic” features to commodity systems may one day become widely accepted. However, hardware assistance is not a panacea. It may speed up a slow algorithm, but it cannot increase the “cleverness” of a given algorithm. In the next chapter, we shall examine how to record memory races with a “clever” algorithm that generates small logs.

Chapter 3

Reduce the Log Size: RTR Algorithm

In the last chapter, we surveyed the background of race recording: the algorithms and the implementations. In this chapter, we present a group of partial-order recording algorithms in much greater detail. Compared with the total-order recording algorithms, partial-order algorithms allow more thread-level parallelism in replay. Unfortunately, an unoptimized partial-order algorithm can generate large logs.

To overcome the problem, this chapter presents a new partial-order recording algorithm that generates reduced logs. The two key ideas for the log reduction are: (1) graph transitive reduction proposed by Netzer [47] and (2) creating stricter and vectorizable dependencies proposed by this dissertation. Evaluation results in Chapter 7 show that our new algorithm reduces the log size by 28% on average over Netzer's algorithm.

Besides the log size problem, the existing partial-order algorithms do not support weak memory models. To mitigate this limitation, we extend the new record-

ing algorithm to supporting the TSO (x86-like) memory model. This extension belongs in this chapter, because it is a modification to the recording algorithm, rather than the recorder implementation.

This chapter is organized as follows: we present the terminology in Section 3.1. We describe a baseline algorithm in Section 3.2. In Section 3.3, we present the details of Netzer’s *Transitive Reduction* (TR) algorithm. In Section 3.4, we present our *Regulated Transitive Reduction* (RTR) algorithm, which is built on top of TR. Section 3.5 provides a correctness proof of the RTR algorithm. Section 3.6 extends RTR to the TSO memory model. Finally, we discuss a generalized RTR algorithm framework in Section 3.7, which provides insights in designing future algorithms to suit different “log size versus parallelism” trade-offs.

To make this dissertation more focused on the recording problem, we delay the discussion of replay until Chapter 8.

3.1 Terminology

3.1.1 Conflicts

Throughout this dissertation, we divide program executions into sequences of dynamic instructions within threads. In a multithreaded execution, two dynamic memory instructions *conflict* if and only if they are executed by different threads, access a common memory location, at least one of them is a write, and satisfy one of the three conditions: (1) one of the write overwrites the value written by the other write, (2) the write overwrites the value read by the read, (3) the read reads

the value written by the write. We call a pair of conflicting dynamic memory instructions simply *a conflict*. To label conflicts, we use a tuple TID:IC to uniquely identify each dynamic instruction. TID is the thread ID of the thread that executes a dynamic instruction. *Dynamic Instruction Count (IC)* is a monotonically increasing number that we give to each dynamic instruction executed by a thread in the program order. A conflict is denoted $i : x \rightarrow j : y$, where \rightarrow denotes the logical order of the two dynamic instructions. We call the first instruction the *source instruction*, the thread that executes the source instruction the *source thread*. Similarly, we call the second instruction/thread the *destination instruction/thread*.

We assume a RISC-like instruction set. If, on the other hand, a CISC instruction set is used, our dynamic instruction count is replaced with a dynamic operation count, where each operation corresponds to an indivisible memory read or write operation. In other words, for unaligned, variable-sized memory operations, we model each B-byte load (store) to address A as an atomic sequence of byte loads (stores) to addresses A, A+1, ..., and A+B-1.

3.1.2 Memory Races

Informally, in a given execution, *memory races* (or simply, *races*) are those conflicts in which the instruction order can reverse in other executions. Races include *general races* and *data races*. Note that some conflicts can change their order from execution to execution, but in a way that the order always follows the order of another general race. Usually, these conflicts are ordered in a way that a pro-

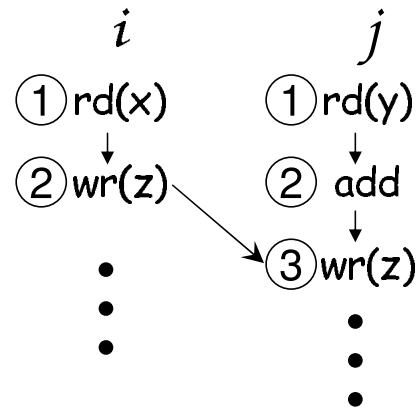


Figure 3.1 Notations of multithreaded execution and conflict (dependence).

Throughout this dissertation, we divide program executions into sequences of dynamic instructions within threads. The notations $rd(x)$ and $wr(x)$ denote read and write to memory location x . Each dynamic instruction is given an instruction count (IC), which is monotonically increasing within a thread and is denoted with numbered circles. The arrows denote conflicts that exist in the original execution, or the dependencies in the replay execution. The direction of arrows determines the order between the two instructions, which are involved in the conflict or the dependence. We call the first and the second instruction/thread of a conflict or a dependence the *source* and the *destination* instruction/thread, respectively.

grammer would expect. Therefore, these conflicts are not considered races. Races are more formally defined by Netzer and Miller [48].

Because all races are conflicts (the converse is not true), it is sufficient to record all conflicts in order to record races. But, why do we need to record races?

Figure 3.1 shows an example conflict ($i : 2 \rightarrow j : 3$), which happens to be a data race. Two threads i and j both write a variable z . The ICs are shown in the numbered circles. The final value of z is determined by which thread “loses” the race. In general, both data accesses and synchronization, such as `mutex_lock`, can race to cause nondeterminism. Imagine multiple threads concurrently, and correctly, insert and search for an integer in a hash table data structure. Depending on the ordering of the insert and search operations, a thread may or may not find the integer it is searching for in the hash table. The nondeterminism caused by

races can prevent multithreaded executions from being faithfully replayed.

3.1.3 Race Recorder, Dependence Log and Race Replayer

To deal with the nondeterminism caused by races, researchers have proposed to use *race recorders* to enable deterministic replay [51]. The goal of race recorders is not to literally record all races in a multithreaded execution. Instead, its goal is to record enough information so that a *race replayer* can use the information to recreate the same races during replay.

A race replayer recreates the same races in a replay execution by forcing the dynamic instructions from different threads to execute in certain orders. The order is called *dependence* in this dissertation. Similarly, the logs generated by a race recorder and used by a race replayer are herein called the dependence logs.

3.2 Unoptimized Recording Algorithm

To ensure a faithful replay, it is sufficient to log every detected conflict as replay dependencies. (Recall all races are conflicts.) An *unoptimized* race recording algorithm can do just that. In fact, Instant Replay [31] essentially employs this algorithm. However, on modern machines with fast interthread communication, conflicts can happen frequently, which causes this unoptimized recording algorithm to suffer from generating large logs.

Although this unoptimized algorithm is not used in our recorders, it serves as a starting point for further refinements. We refine this unoptimized algorithm to obtain the new RTR algorithm using an example in Figure 3.2 to Figure 3.5.

To start the example, Figure 3.2 shows an execution with five conflicts.

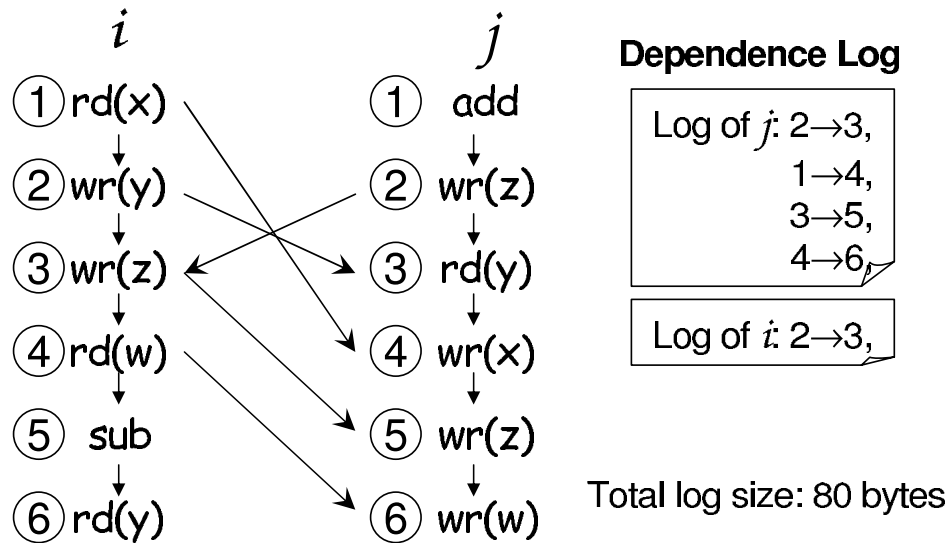


Figure 3.2 Unoptimized recording algorithm. Five conflicts result in five dependencies being logged. The dependence log is 80 bytes if each IC is 64 bits wide and the thread IDs are omitted from the log since there are only two threads.

Assuming each IC is 64-bit wide, the unoptimized algorithm generates two dependence logs (one for each thread) with a total size of 80 bytes. TIDs can be omitted from the logs in this example, which has only two threads. For executions that include more than two threads, TIDs must be included in the logs too.

3.2.1 The Algorithm and Its Pseudo Code

Table 3-1 shows the pseudo code of the unoptimized algorithm. The unoptimized algorithm maintains an integer instruction count for each thread. The IC is also known as a *logical time base*, because it denotes a flow of logical time [30] within each thread. When an IC value is associated with a memory block, the value is called a *scalar logical timestamp*. The unoptimized algorithm maintains a scalar logical timestamp for each memory word, because we assume word-size, word-aligned memory blocks are the minimum memory access granularity. If sub-

Table 3-1 The Unoptimized Algorithm

DATA STRUCTURES timestamp_t := int; // scalar timestamp (IC) readers_t := map { < int thread_i, timestamp_t read_ts_i > , ... } // record a set of readers block_history_t := structure {int writer, timestamp_t w_ts, readers_t readers} // access history	
VARIABLES timestamp_t current_ts; // each thread maintains a current timestamp, initialized to 0 block_history_t the_block; // access history of the block being accessed	
ALGORITHM FOR WRITE EVENTS OF THREAD I current_ts := current_ts + 1 foreach < a_reader, a_ts > in the_block.readers if (i != a_reader) // found a WAR conflict write_log(a_reader:a_ts, i:current_ts) if (i != the_block.writer) // found a WAW conflict write_log(the_block.writer:the_block.w_ts i:current_ts) // update the block's access history clear(the_block.readers) the_block.writer := i the_block.w_ts := current_ts	ALGORITHM FOR READ EVENTS OF THREAD I current_ts := current_ts + 1 if (i != the_block.writer) // found a RAW conflict write_log(the_block.writer:the_block.w_ts i:current_ts) // update the block's access history add(the_block.readers, < i, current_ts >)

word accesses were to be allowed, the logical timestamp needs to be kept at a finer granularity. (In Chapter 4, we show that the access history can be maintained at a coarser granularity, *i.e.*, cache lines.) In general, the logical timestamp is part of the *access history* of each memory block. The access history includes (1) the TID of the most recent writer (the `writer` variable); (2) the logical timestamp when the writer wrote the block (the `w_ts` variable); and (3) a set of <reader TID: reader timestamp> pairs (the `readers` variable) to record the most recent read from each thread after the most recent write from any thread.

The algorithm works by detecting and logging all three types of conflicts: *Write-After-Read* (WAR), *Write-After-Write* (WAW), and *Read-After-Write* (RAW) conflicts. The unoptimized algorithm taps into the execution by tracing all read

and write instructions. The recorder traces all memory locations, because the recorder cannot make the assumptions about which memory locations are private (*i.e.*, not shared by multiple threads). In the SC systems, memory read and write instructions *logically* complete (*i.e.*, commit and access the memory) in the program order. When a write completes, the unoptimized algorithm first increases the `current_ts` to denote the advance of the logical time of the thread. (Alternatively, the logical time can be advanced for every dynamic instruction, instead of only memory instructions. We discuss how to choose the logical timestamp base in Section 3.7.1.) If the instruction is a write to a block `the_block`, the unoptimized algorithm checks `the_block`'s access history to see if there were recent readers. For each reader found in the `readers` set, a WAR conflict is detected and logged. Additionally, a WAW conflict is detected and logged if there was a previous writer found in the access history. Finally, we update `the_block`'s access history by clearing the `readers` set and setting the current thread as the new writer for `the_block`.

Similarly, when a read completes, the unoptimized algorithm again first increases the `current_ts` variable. The unoptimized algorithm then detects and logs a RAW conflicts from the most recent writer to this read. Finally, the unoptimized algorithm adds this reader and the reader thread's current IC as the reader's timestamp to `the_block`'s access history.

3.2.2 Brief Overview of the Replay

After the dependence log is generated in the recording phase, a replayer is

used to recreate a deterministic replay of the original execution. In this dissertation, we assume an in-order replayer, *i.e.*, instructions within a thread execute in the program order during replay, even when the order can be relaxed (*e.g.*, between two independent instructions). For dynamic instructions from different threads, the replayer enforces their logical order according to the dependencies in the dependence log. In this chapter, we do not care how the replayer enforces dependencies. Instead, we assume that all the dependencies in the log are enforced during replay and use this assumption in establishing the correctness argument of our new recording algorithm. We discuss the replayer in more detail in Chapter 8.

Out-of-order replay is not considered in this dissertation, because it has at least three drawbacks: (1) the replayer is complex, because the replayer needs to find independent instructions; (2) it is not clear the replayer can always precompute the IC of an instruction before executing the instruction out-of-order; and (3) the recorder cannot apply transitive reduction assuming the program order will be enforced during the replay. Out-of-order replay can have two advantages in (1) higher replay performance and (2) the potential capability to deal with weak memory consistency models. Further investigation of in-order versus out-of-order replay is beyond the scope of this dissertation.

3.3 Netzer's Transitive Reduction (TR) Algorithm

To reduce the size of the dependence log, we now set out to find a subset of the conflicts to be recorded, while still enabling faithful replay. Logging a subset of

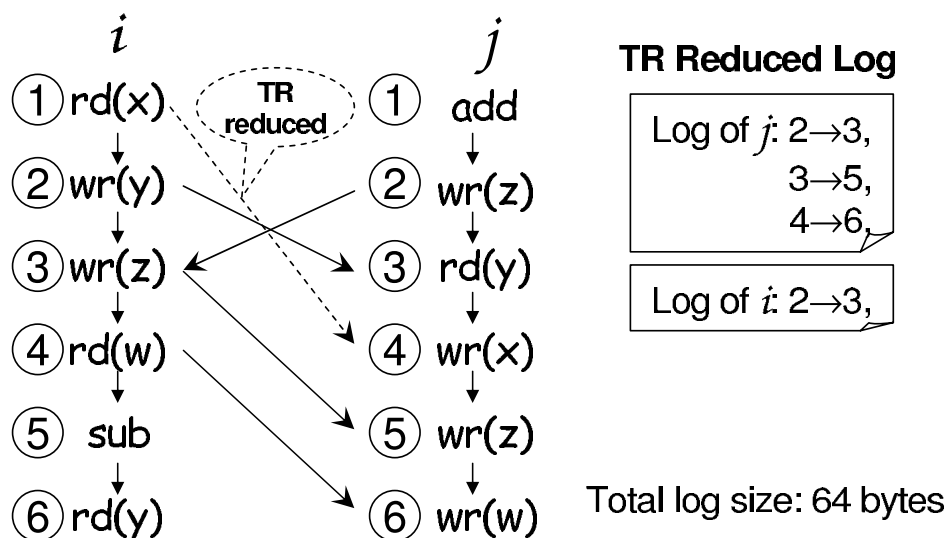


Figure 3.3 Netzer's transitive reduction algorithm. One dependence is reduced by transitivity and the dependence log is 64 bytes.

the conflicts still enables faithful replay because the reduced conflicts are transitively implied by others, as observed by Netzer [47]. Netzer proposed a *Transitive Reduction (TR)* method to reduce the number of conflicts to be recorded.

3.3.1 Log Size Reduction

For example, Figure 3.3 shows the conflict $i:1 \rightarrow j:4$ is implied by the conflict $i:2 \rightarrow j:3$, because $i:1$, $i:2$ and $j:3$, $j:4$ are in program order. Enforcing $i:2 \rightarrow j:3$ during replay will reproduce both $i:2 \rightarrow j:3$ and $i:1 \rightarrow j:4$. Therefore, conflict $i:1 \rightarrow j:4$ can be removed from the log. After this reduction, the total log size is 64 bytes. In practice, Netzer showed that TR reduces the log size significantly (82% to 99.8%) over the unoptimized algorithm [47].

Table 3-2 Transitive Reduction (TR) Algorithm

<p>DATA STRUCTURES</p> <p>timestamp_t := array {int t1, int t2, ... int tn}; // <i>vector timestamp</i></p> <p>readers_t := map {⟨ int thread_i, timestamp_t read_ts_i ⟩, ...} // <i>record a set of readers</i></p> <p>block_history_t := structure {int writer, timestamp_t w_ts, readers_t readers} // <i>access history</i></p>
<p>VARIABLES</p> <p>timestamp_t current_ts; // <i>each thread maintains a current timestamp, initialized to {0, ... 0}</i></p> <p>block_history_t the_block; // <i>access history of the block being accessed</i></p>
<p>ALGORITHM FOR WRITE EVENTS OF THREAD I</p> <p>current_ts[i] := current_ts[i] + 1</p> <p>foreach ⟨ a_reader, a_ts ⟩ in the_block.readers</p> <p> if (i != a_reader && current_ts[a_reader] < a_ts[a_reader])</p> <p> // <i>found an irreducible WAR</i></p> <p> write_log(a_reader:a_ts[a_reader], i:current_ts[i])</p> <p> // update the current timestamp</p> <p> current_ts := merge_vector_timestamp(current_ts, a_ts)</p> <p>if (i != the_block.writer</p> <p>&& current_ts[the_block.writer] < the_block.w_ts[the_block.writer])</p> <p> // <i>found an irreducible WAW</i></p> <p> write_log(the_block.writer:the_block.w_ts[the_block.writer], i:current_ts[i])</p> <p> // update the current timestamp</p> <p> current_ts := merge_vector_timestamp(current_ts, the_block.w_ts)</p> <p> // <i>update the block's access history</i></p> <p> clear(the_block.readers)</p> <p> the_block.writer := i</p> <p> the_block.w_ts := current_ts</p> <p>ALGORITHM FOR READ EVENTS OF THREAD I</p> <p>current_ts[i] := current_ts[i] + 1</p> <p>if (i != the_block.writer</p> <p> && current_ts[the_block.writer] < the_block.w_ts[the_block.writer])</p> <p> // <i>found an irreducible RAW</i></p> <p> write_log(the_block.writer:the_block.w_ts[the_block.writer], i:current_ts[i])</p> <p> // update the current timestamp</p> <p> current_ts[the_block.writer] := the_block.w_ts[the_block.writer]</p> <p> // <i>update the block's access history</i></p> <p> add_and_replace_the_old_one(the_block.readers, ⟨ i, current_ts ⟩)</p>

3.3.2 Pseudo Code

Table 3-2 shows the pseudo code for the TR algorithm. The differences between the TR algorithm and the unoptimized algorithm are highlighted in

bold. Instead of using a scalar timestamp for each thread, the TR algorithm maintains a *vector timestamp* for each thread. Element j of the vector timestamp of thread i represents the maximum IC of thread j that thread i is transitively dependent on, when j does not equal to i . When j equals to i , the element in the vector timestamp is simply the IC of thread i .

Unlike the unoptimized algorithm, the TR algorithm logs conflicts selectively. For example, for a WAR conflict that starts from timestamp `a_ts` and ends at timestamp `current_ts`, TR algorithm first checks whether the conflict is already implied by a previously logged conflict by comparing the corresponding elements of two timestamps: `a_ts[a_reader]` and `current_ts[a_reader]`. When the conflict is not already implied, the conflict is logged, like in the unoptimized algorithm.

Furthermore, regardless of whether the conflict is reducible, we update the `current_ts` with the conflicting timestamp (e.g., `a_ts`) by “merging” the two vector timestamps. During the merge, each element of `current_ts` is updated with the larger element between `current_ts` and `a_ts` for each corresponding element of the vector timestamps. By updating the vector timestamp this way, we propagate the maximum conflict IC from one thread to another, ultimately enabling the transitive reduction.

We point out that propagating vector timestamps is not strictly needed for transitive reduction, but it enables more log size reduction than propagating scalar timestamps. In Chapter 4, we will show that our hardware implementations propagate only scalar timestamps, which sacrifices a few reduction opportunities

in exchange for a cheaper design.

3.4 New Regulated Transitive Reduction (RTR) Algorithm

We find that the log size can be further reduced by introducing *stricter* and *vectorized* dependencies in the dependence log. We call the new reduction method *Regulated Transitive Reduction (RTR)*, because, in the end, it regulates how races are replayed.

The intuition behind the RTR algorithm is to create a group of dependencies that are stored in the log in a compact format. After applying TR, all dependencies from a thread i to a thread j are irreducible. Logging a dependencies from i to j requires two integers: the source IC and the destination IC. The basic idea of RTR is to make the numerical differences (called the *IC stride*) between the source and the destination ICs to be the same for groups of consecutive dependencies. Therefore, logging a group of x dependencies requires only $x+1$ integers (x IC and 1 IC stride), rather than $2x$ integers.

3.4.1 Log Size Reduction

Stricter Dependencies. Figure 3.4 shows how RTR enables more transitive reduction by introducing artificial dependencies in the dependence log. During recording, after the conflict $i:3 \rightarrow j:5$ is detected, RTR writes a stricter dependence $i:4 \rightarrow j:5$ in the dependence log. The dependence is stricter than $i:3 \rightarrow j:5$ because enforcing it during replay is sufficient but not necessary for replaying the conflict $i:3 \rightarrow j:5$. As a result of this stricter dependence, the conflicts $i:3 \rightarrow j:5$

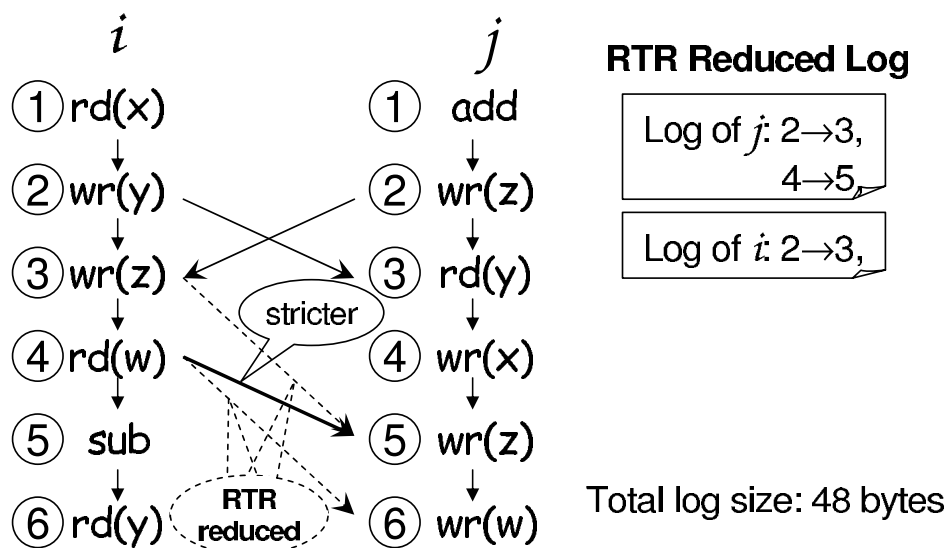


Figure 3.4 Further reduce the log with stricter dependencies. Two dependencies are replaced by one stricter dependence. The dependence log is 48 bytes with three dependencies in total.

and $i:4 \rightarrow j:6$ are both transitively reducible, *i.e.*, not logged. Thus, RTR reduces the log size to 48 bytes, which is not possible if we log only a subset of the original conflicts.

Vectorized Dependencies. But why does RTR decide to introduce $i:4 \rightarrow j:5$ but not $i:5 \rightarrow j:5$? After all, both dependencies are stricter than $i:3 \rightarrow j:5$. RTR introduces $i:4 \rightarrow j:5$ because this dependence and a previous dependence ($i:2 \rightarrow j:3$) form a group of *vectorizable* dependencies. A group of dependencies is vectorizable if and only if all member dependencies have the same IC stride. The IC stride (herein denoted by Δ) of a dependence is the numerical difference between the two ICs of the source and the destination of the dependence. As we show visually in Figure 3.5, $i:4 \rightarrow j:5$ and $i:2 \rightarrow j:3$ both have their IC strides equal to one. By logging vectorized dependencies, RTR can reduce the log size by writing the dependence log with a concise format:

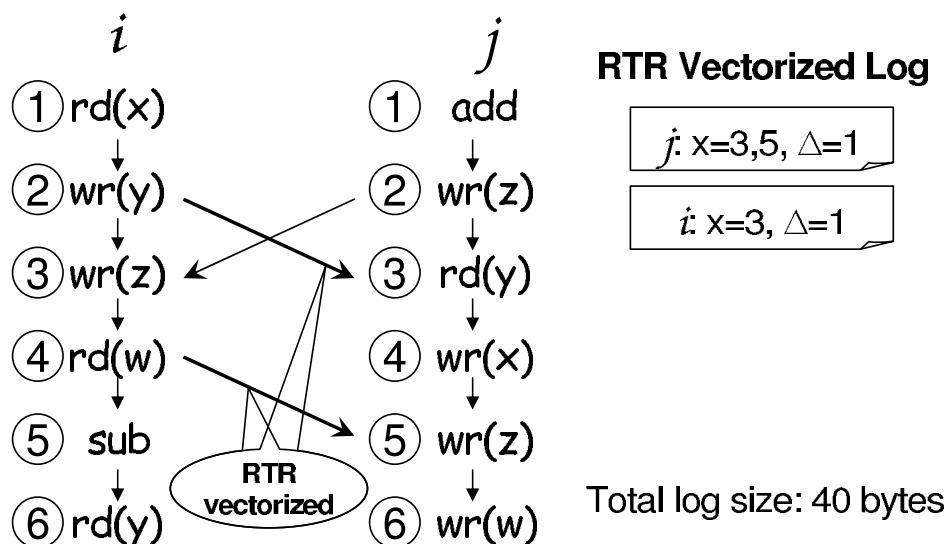


Figure 3.5 Compact the log with vectorized dependencies. Two dependencies of thread j are vectorizable. The compact representation of the dependence log reduces the log size of 40 bytes.

$$\{(x-\Delta) \rightarrow x \mid x=\{3,5\}, \Delta=1\}$$

where $(x-\Delta) \rightarrow x$ serves as a template format of the vectorized dependencies; the variable x is substituted with a set of logged ICs during replay. A more concise format, $\{(x-\Delta) \rightarrow x \mid 3 \leq x \leq 5, \Delta=1\}$, exists. However, this format introduces a different trade-off between the log size and the replay speed. Future work may explore this trade-off.

With both stricter and vectorized dependencies, the total log size is reduced to 40 bytes in our example, a 37.5% reduction over TR.

3.4.2 Replay Correctness and Performance

Replay Correctness. In the TR algorithm, a subset of conflicts is logged in the dependence log, these dependencies directly ensure that replay recreates the same subset of the conflicts. The rest of the conflicts are recreated by transitivity.

The RTR algorithm, on the other hand, introduces stricter dependencies that

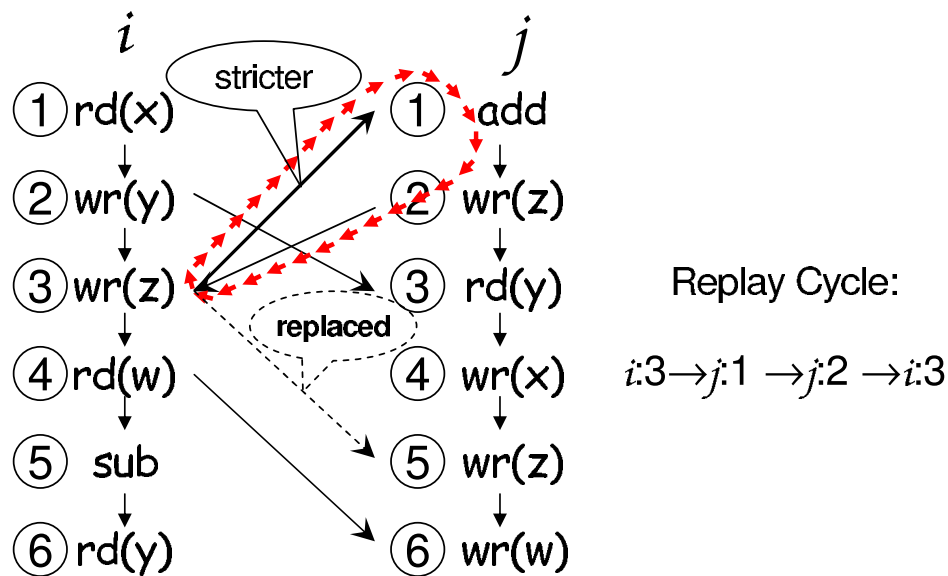


Figure 3.6 A replay deadlock due to an overly strict dependence. The deadlock is caused by a cycle of dependencies. The cycle is introduced by an overly strict dependence.

do not correspond directly to any conflict. To ensure correct replay, a stricter dependence $i:v \rightarrow j:w$ that replaces a conflict $i:x \rightarrow j:y$ must satisfy that (1) v is after or equal to x in thread i ; (2) w is before or equal to y in thread j .

More importantly, the RTR algorithm must not introduce “overly strict” dependences, or it will cause replay deadlocks. For example, in Figure 3.6, let us assume the RTR algorithm introduces $i:3 \rightarrow j:1$ to replace $i:3 \rightarrow j:5$. At a glance, $i:3 \rightarrow j:1$ may appear to be correct because it is indeed stricter than $i:3 \rightarrow j:5$. However, a cycle of dependence, namely $i:3 \rightarrow j:1 \rightarrow j:2 \rightarrow i:3$, is formed after this “overly strict” dependence is added. This cycle will cause replay to deadlock because a replayer cannot decide whether $i:3$ or $j:1$ should be executed first.

To avoid replay deadlocks, the RTR algorithm conservatively ensures that all

stricter dependencies are in a total order. We choose *the SC order* as this total order. This avoids any cycle of dependencies during replay, because all program order dependencies are also in the SC order, which is a topological sort of all dependencies in replay.

We provide a correctness proof of the RTR algorithm in Section 3.5.

Replay Performance. In addition to replay correctness, the RTR algorithm may affect the replay performance, because RTR introduces stricter dependencies that may affect the critical path of the program execution.

More concretely, every dependence, say $i:x \rightarrow j:y$, in the dependence log requires a synchronization check during replay. The synchronization check incurs two types of delay. First, the check itself needs time, mainly for the communication between threads. Second, additional delay to instruction $j:y$ is added if $i:x$ is not yet finished when $j:y$ is ready.

The RTR algorithm affects replay performance in two ways. In one way, introducing stricter dependences can add more delays to replay, because stricter dependences cause an instruction to wait for some extra instructions that it does not conflict with. In the other way, RTR potentially can improve replay performance with fewer synchronization checks. Future research is necessary for a quantitative evaluation of the impact on the replay performance.

3.4.3 Pseudo Code

Table 3-3 shows the pseudo code of the RTR algorithm. Again, the differences

Table 3-3 Regulated Transitive Reduction (RTR) Algorithm

<p>DATA STRUCTURES</p> <p>timestamp_t := array {int t1, int t2... int tn}; // <i>vector timestamp</i></p> <p>readers_t := map {⟨ int thread_i, timestamp_t read_ts_i ⟩, ...} // <i>record a set of readers</i></p> <p>block_history_t := structure {int writer, timestamp_t w_ts, readers_t readers} // <i>access history</i></p> <p>sliding_window_t := structure {int Δ_{min}, int Δ_{max}} // <i>window of vector dep.</i></p>
<p>VARIABLES</p> <p>timestamp_t current_ts; // <i>each thread maintains a current timestamp, initialized to {0, ... 0}</i></p> <p>block_history_t the_block; // <i>access history of the block being accessed</i></p> <p>logable set of deps; // <i>a set of dependences "buffered" before we get the IC stride</i></p> <p>sliding_window_t ws[]; // <i>an array of windows, one for each foreign thread</i></p>
<p>ALGORITHM FOR WRITE EVENTS OF THREAD I</p> <p>current_ts[i] := current_ts[i] + 1</p> <p>foreach ⟨ a_reader, a_ts ⟩ in the_block.readers</p> <p> if (i != a_reader && current_ts[a_reader] < a_ts[a_reader])</p> <p> // <i>found an irreducible WAR</i></p> <p> // compute the new window</p> <p> min := current_ts[i] - reader_thread.current_ts[a_reader]</p> <p> max := current_ts[i] - a_ts[a_reader]</p> <p> if (ws[a_reader] overlaps (min, max))</p> <p> shrink ws[a_reader] according to (min, max)</p> <p> add current_ts[i] to the logable set of dependencies</p> <p> else</p> <p> write_log(logable set of deps, ws[a_reader].Δ_{max}, a_reader, i)</p> <p> ws[a_reader] := (min, max) // make a new window to start over</p> <p> // <i>update the current timestamp</i></p> <p> current_ts := merge_vector_timestamp(current_ts, a_ts)</p> <p> if (WAR conflict logged)</p> <p> current_ts[a_reader] := (current_ts[i] - ws[a_reader].Δ_{max})</p> <p>if (i != the_block.writer && current_ts[the_block.writer] < the_block.w_ts[the_block.writer])</p> <p> // <i>found an irreducible WAW</i></p> <p> // compute the new window</p> <p> min := current_ts[i] - writer_thread.current_ts[the_block.writer]</p> <p> max := current_ts[i] - the_block.w_ts[the_block.writer]</p> <p> if (ws[the_block.writer] overlaps (min, max))</p> <p> shrink ws[the_block.writer] according to (min, max)</p> <p> add current_ts[i] to the logable set of dependencies</p> <p> else</p> <p> write_log(logable set of deps, ws[the_block.writer].Δ_{max}, the_block.writer, i)</p> <p> ws[the_block.writer] := (min, max) // make a new window to start over</p> <p> // <i>update the current timestamp</i></p> <p> current_ts := merge_vector_timestamp(current_ts, the_block.w_ts)</p> <p> if (WAW conflict logged)</p> <p> current_ts[the_block.writer] := (current_ts[i] - ws[the_block.writer].Δ_{max})</p> <p> // <i>update the block's access history</i></p> <p> clear(the_block.readers)</p> <p> the_block.writer := i</p> <p> the_block.w_ts := current_ts</p> <p>ALGORITHM FOR READ EVENTS OF THREAD I</p> <p>(Omitted - similar to the algorithm for the write event)</p>

with the TR algorithm is shown in bold. First, we add two additional per-thread states: (1) a buffer for the group of vectorizable dependencies; (2) a “sliding window” of IC stride for the group of vectorizable dependencies between every pair of threads. For convenience, we have introduced a new variable, `reader_thread`, to denote the thread with TID equals to `a_reader`. As with the TR algorithm, we use timestamps to find irreducible conflicts. Then, instead of logging the irreducible conflict immediately as a dependence, we save the dependence in the buffer and compute the minimum and maximum IC stride for the dependence. The maximum IC stride is straightforward: we compute the difference between the source and destination ICs of the dependence. For the minimum IC stride, we assume the strictest dependence we could introduce is from the most recently completed instruction of the source thread to the destination thread. Therefore, the minimum IC stride is computed from this strictest dependence, which could be later introduced as a concrete dependence.

With multiple dependencies buffered, we compute a common “overlapping” window for the IC stride sliding window. Clearly, the sliding window will shrink as more dependencies are buffered. Eventually, a new dependence will be found not to “overlap” with the sliding window, *e.g.*, the dependence is not vectorizable with the buffered dependencies. In that case, we flush the group of vectorizable dependencies to the log and reset the sliding window for the next group of vectorizable dependencies.

The RTR algorithm is a greedy heuristic in finding vectorizable dependencies. The RTR algorithm requires only one pass in processing the conflicts and is suit-

able for online vectorization, although RTR does not guarantee finding the optimal vectorization to minimized the log size.

3.5 Correctness the RTR Recording Algorithm

In this dissertation, we target *Process-Level Replay*. Informally, a process-level replay is successful if and only if (1) each thread has the same sequence of dynamic instructions in the original and the replay executions and (2) for every load instruction, the replay provides the information about which store instruction generates the value that the load instruction reads from the memory. The next subsection formally defines what we consider a successful replay.

It is interesting to note that process-level replay is a stronger replay requirement than that used in BugNet [46]. In BugNet, the replay requirement can be called *Thread-Level Replay*, which requires the same set of instructions to be replayed for each thread. Thread-level replay does not always provide the information on the ordering relationship between load and store instructions from different threads.

3.5.1 Successful Replay

Definition 1: A *recorded execution* E is a tuple $E = \langle O, \Rightarrow, \rightarrow \rangle$, where O is the set of all instructions of all threads; \Rightarrow is a total order relation defined on O ; \rightarrow is a partial order relation defined on every pair of conflicting memory instructions in O and \rightarrow is compatible with \Rightarrow , i.e., $\forall (a,b) \in O, a \rightarrow b$ implies $a \Rightarrow b$.

Definition 2: A *replay execution* \tilde{E} is a tuple $\tilde{E} = \langle \tilde{O}, \tilde{\Rightarrow}, \tilde{\rightarrow} \rangle$, where \tilde{O} is the set of all instructions of all threads; $\tilde{\Rightarrow}$ is a total order relation defined on \tilde{O} ; $\tilde{\rightarrow}$ is a partial order relation defined on every pair of conflicting memory instructions in \tilde{O} and $\tilde{\rightarrow}$ is compatible with $\tilde{\Rightarrow}$. (In fact, $\tilde{\rightarrow}$ is the set of dependencies logged by a recorder and $\tilde{\rightarrow}$ is used to restrict the replay.)

Definition 3: A *successful replay execution* $\tilde{E} = \langle \tilde{O}, \tilde{\Rightarrow}, \tilde{\rightarrow} \rangle$ of a recorded execution $E = \langle O, \Rightarrow, \rightarrow \rangle$ satisfies: (1) $O = \tilde{O}$; and (2) \rightarrow is compatible with $\tilde{\Rightarrow}$.

3.5.2 Proof Sketch

We now prove the correctness of the RTR algorithm (with respect to the SC memory model) with the following theorem.

Theorem 1: The RTR algorithm (as described in Section 3.4.3) enables successful replays.

Proof sketch:

We first make three observations (without proof) about the RTR recording algorithm in Section 3.4.3.

Observation 1: We assume that both the recorded and the replay executions are performed on sequential consistent machines. (It is important to note that we assume in-order replay.) Therefore, by the definition of SC, for every thread the program order (herein denoted by $\widehat{\rightarrow}$) is compatible with the total orders \Rightarrow and $\tilde{\Rightarrow}$ in the recorded and replay executions. (This is from the definition of the SC memory model.)

Observation 2: For a conflict $x \rightarrow y$ in E , a dependence $a \xrightarrow{\sim} b$ introduced by a recorder satisfies:

1. $((x = a) \vee (x \widehat{\rightarrow} a)) \wedge ((b = y) \vee (b \widehat{\rightarrow} y))$ (Section 3.4.2);
2. $a \Rightarrow b$ (Section 3.4.2).

Observation 3: Logging the vectorizable dependencies in a vectorized format (as opposed to logging them individually) does not change the correctness of the replay.

Given these observations, we prove Theorem 1 in two steps.

Step 1: A replay execution exists.

To show a replay execution $\tilde{E} = \langle \tilde{O}, \tilde{\Rightarrow}, \tilde{\rightarrow} \rangle$ exists, we show the set of dependencies $\tilde{\rightarrow}$, which is logged by a recorder, is a partial order (*i.e.*, acyclic). This is trivial to prove: from the second condition of Observation 2, we know $\forall (a,b) \in \tilde{O}, a \xrightarrow{\sim} b$ implies $a \Rightarrow b$. In other words, \Rightarrow is a topological sort of $\tilde{\rightarrow}$.

Therefore, a replay execution exists.

Step 2: The replay execution is successful.

(1) By the correctness of transitive reduction (proved by Netzer with enumeration and induction [47]), the first condition of Observation 2, and Observation 3, we know $\forall ((x,y) \in O, x \rightarrow y)$ there is at least one dependence $a \xrightarrow{\sim} b$, such that $((x = a) \vee (x \widehat{\rightarrow} a)) \wedge ((b = y) \vee (b \widehat{\rightarrow} y))$.

(2) From Definition 2, we have that $\tilde{\rightarrow}$ is compatible with $\tilde{\Rightarrow}$. Thus, $a \xrightarrow{\sim} b$.

(3) From Observation 1, the replay execution is sequentially consistent. Thus, $x \xrightarrow{\sim} a$ and $b \xrightarrow{\sim} y$.

From (2) and (3), we know $x \xrightarrow{\sim} y$, *i.e.*, \rightarrow is compatible with $\xrightarrow{\sim}$.

(4) Because all conflicts are reproduced, through an induction (omitted), we can show all dynamic load instructions get the same values in the original and the replay executions. Because all dynamic load instructions get the same value, $O = \tilde{O}$.

By Definition 3, the replay execution is successful.

To conclude the proof, we show that a replay execution both exists and is successful. Therefore, the RTR algorithm enables successful replays.

3.6 Extending RTR to Supporting the TSO Memory Model

We now extend our recorder to supporting the Total Store Order (TSO) [70] memory model, which relaxes write-to-read ordering to the shared memory. We focus on TSO because it is well defined [70] and the popular x86 architecture assumes a memory model that is similar to TSO [32]. By supporting TSO, we hope our recorder will support a majority of the existing systems.

We have three goals in extending RTR to supporting TSO.

1. Recording TSO executions without forcing the executions to conform to SC.
2. (At most) Modestly increasing the log size of the recorder.
3. (At most) Modestly increasing the hardware complexity and cost of the recorder.

It is important to note that this section inevitably discusses some issues of the hardware implementation of the race recorder, because the TSO memory model is intimately related to the multiprocessor hardware implementation. When

needed, we use forward references to the next chapter on our recorder’s hardware implementation.

Because our simulation infrastructure supports only the SC memory consistency, we do not quantitatively evaluate the TSO recording method described in this section. Instead, we qualitatively argue that the TSO recording method achieves the three goals above.

3.6.1 TSO model and its Impact on Race Recording

With TSO, a processor can implement a hardware first-in-first-out (FIFO) *write buffer*. Informally, we say a store instruction *commits* by placing its write value in the write buffer. Later, the store instruction is *ordered* when the instruction exits the write buffer and updates the memory. Furthermore, TSO requires a processor’s stores to be *ordered* in *commit* order (*i.e.*, FIFO write buffer). Load instructions from the same processor can *bypass* from the write buffer. A load instruction *commits* after all instructions before the load instruction commits. We say a load instruction is *ordered* when the load instruction commits. Load can return its value either from the memory or the write buffer. We call the two types of load instructions load_M (load-memory) and load_B (load-bypass), respectively. (If a load instruction partially bypasses from the write buffer, we break it into smaller load_M and load_B sub-instructions.)

load_M and load_B instructions can cause our race recording algorithm (as presented so far) to fail under the TSO memory model. More specifically, load_M instructions can cause *Instruction Reordering*. load_B instructions can cause *Store*

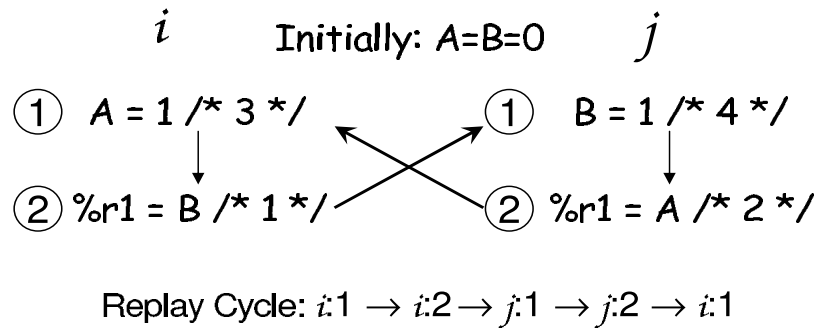


Figure 3.7 An example TSO execution (load_M). In this figure, we show a multithreaded execution with memory values (instead of just $\text{rd}(x)$ and $\text{wr}(x)$). The load_M instructions (“ $\%r1=B$ ” and “ $\%r1=A$ ”) are ordered before the store instructions. The execution causes a replay deadlock if the recorder logs only the interthread dependencies and the replayer replays in-order (in an SC order). The numbers in $/^* */$ denote the machine cycles, when the instructions are ordered (reading from or writing to the memory).

Atomicity violation. Both instruction reordering and store atomicity violation can cause an execution under the TSO memory model to violate the SC semantics required by the SC memory model. Arvind and Maessen proposed a formal framework of defining multiprocessor memory models using instruction reordering and store atomicity [4]. We now use examples to describe the impact of load_M and load_B instruction to race recording in more detail.

3.6.1.1 The Impact of Load_M Instructions

In an SC system, instructions (executed by the same thread) commit and are ordered in the program order. In a TSO system, even though the instructions commit in program order, the write buffer can cause independent load_M and store instructions to be ordered differently from the program order, which can cause a multithreaded execution to violate the strong SC semantics. In this dissertation, we call those multithreaded executions that are allowed by the TSO memory

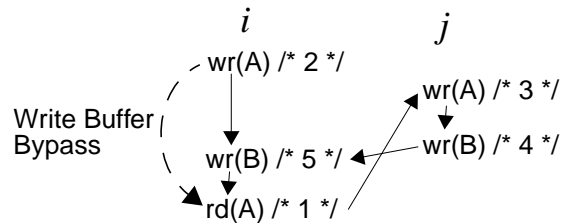


Figure 3.8 An example TSO execution (load_B). In this figure, a load_B instruction $i:\text{rd}(A)$ reads its value from $i:\text{wr}(A)$ through the write buffer during recording. The numbers in /* */ denote the machine cycles when the instructions are ordered. A cycle of dependencies are recorded due to the write buffer bypassing.

model the *TSO executions*.

Figure 3.7 shows an example TSO execution that is not an SC execution. In this execution, thread i first writes memory location A then reads a different memory location B , thread j first writes B then reads A . Let us assume thread i and j run on two different processors. Because of the write buffers, the two load_M instructions are ordered (*i.e.*, commit) before the two store instructions. The numbers in /* */ denote the machine cycles when the instructions are ordered. For this example, our race recorder would log two interthread dependencies $i:2 \rightarrow j:1$ and $j:2 \rightarrow i:1$. During the in-order replay, these dependencies cause a deadlock (*i.e.*, cycle of dependencies). Therefore, simply recording interthread dependencies does not work with the TSO executions.

3.6.1.2 The Impact of the Load_B Instructions

Figure 3.8 shows another TSO execution that is not an SC execution caused by a load_B instruction. The load_B instruction $i:\text{rd}(A)$ reads its value from $i:\text{wr}(A)$ through the write buffer during recording. The numbers in /* */ denote the machine cycles when the instructions are ordered. Although $i:\text{rd}(A)$

is ordered before $i : wr(A)$, our recorder log the dependence $i : rd(A) \rightarrow j : wr(A)$, rather than $i : wr(A) \rightarrow j : wr(A)$, because $i : rd(A)$ is after $i : wr(A)$ in the program order (with a larger IC). Thus, a cycle of dependencies is recorded due to the write buffer bypassing.

3.6.2 Extending the Unoptimized Algorithm from SC to TSO

We now extend the unoptimized recording algorithm (Table 3-1) to handling TSO executions. Our key observation is that $load_M$ and $load_B$ instructions cause replay deadlocks, because these load instructions are ordered differently with respect to the program order (commit). Therefore, we give special treatment to those “problematic” load instructions, in effect, replaying these load instructions by value, rather than by the interthread ordering.

Our solution is an *order-value-hybrid* recording method. In addition to recording dependencies (the orders), our recorder records the value of a $load_M$ or a $load_B$ instruction if the load instruction may violate the SC ordering. The hybrid recording method deals with the $load_M$ and $load_B$ instructions uniformly: the recorder monitors the loaded memory location from the (physical) time O to (physical) time C , where time O is when a $load_M$ instruction becomes ordered or a the store that feeds a $load_B$ instruction becomes ordered; time C is when all preceding instructions (in program order) of the $load_M$ or the $load_B$ are ordered. During this monitored time period, if the loaded memory location is overwritten by another thread, the recorder logs the *value* of the load instruction. The load instruction is called a potential SC-violating load. If the memory location is not overwritten by another

thread, the recorder does not log the load value, because the load instruction is logically ordered in program order, which means the interthread dependencies will ensure the correct replay of the load.

It is important to note that the definition of time \circ is different for load_M and load_B instructions. It is relatively easier to understand why do we start to monitor a load_M instruction after the load_M is ordered. After all, if no other thread overwrites the loaded memory location in the time interval $[0, C]$, the instruction reordering logically did not happen. It is harder to understand why do we choose to start monitoring a load_B after the store (say `store_x`) feeding the load_B becomes ordered, rather than when the load_B is ordered. This is because the atomicity of `store_x` is violated only if an overwriting store from another thread happens in the time interval $[0, C]$.

In hardware, to achieve this value recording, we augment the processor core with additional hardware that monitors the accessed cache line after the value of a load instruction is returned by the cache or a bypassed store value is written into the cache line. Should the cache line be stolen away (written) by another processor before all preceding (in the program order) store instructions (which are delayed by the write buffer) are ordered, our recorder logs the loaded value for the potential SC-violating load instruction and avoid logging the WAR dependence that sources from the potential SC-violating load instruction, which may cause replay deadlocks. The detection circuitry is similar to the misspeculation detection circuitry in the SC systems (*e.g.*, MISP R10000 [75]) that utilize the speculative execution technique [22]. The difference is that our hardware logs the

load values of potential SC violations rather than triggering pipeline recovery.

We now apply this hybrid recording method to the example in Figure 3.7. The hybrid recorder would monitor the memory location A and B , at thread j and thread i respectively, after the two load_M instructions are ordered. Because A is overwritten by thread i before the store instruction $j:1$ is ordered, the recorder would log the load value of $j:2$ ($A=0$). Similarly, for the example in Figure 3.8, the hybrid recorder would monitor the memory location A after $i:\text{wr}(A)$ is ordered. Because A is overwritten before the instructions $i:\text{wr}(B)$ (which precedes $i:\text{rd}(A)$) is ordered, the recorder would log the load value of $i:\text{rd}(A)$.

During replay, the logged values are used to overwrite the (incorrect) values read from the memory. In Figure 3.7, without the WAR dependence $j:2 \rightarrow i:1$, the replay will not deadlock. When it comes time to execute $j:2$, however, the replayer uses the logged value to overwrite the value loaded from the memory. Therefore, in addition to the changes in the recorder, the order-value-hybrid method requires a (small) change in the replayer so that the replayer can overwrite the value of a potential SC-violating load instruction using the value log.

Table 3-4 highlights the changes to extend the unoptimized algorithm to TSO. In the new algorithm, there is no longer a single `current_ts` variable. Rather, we add an `ordering_buf` data structure to keep track of the load and store instructions in the program order. The `ordering_buf` data structure is a FIFO queue, whose elements are pushed from the tail and popped from the head. The head of the buffer keeps the oldest instruction (with the smallest IC) that has not been ordered. The tail of the buffer keeps the youngest instruction (with the larg-

Table 3-4 The Unoptimized Algorithm on TSO

<p>DATA STRUCTURES</p> <p>timestamp_t := int; // scalar timestamp (IC)</p> <p>readers_t := map { < int thread_i, timestamp_t read_ts_i >, ... } // record a set of readers</p> <p>block_history_t := structure {int writer, timestamp_t w_ts, readers_t readers} // access history <i>// tail is the IC of the most recently committed instruction, mode is 0 (load) or 1 (store)</i></p> <p>ld_st_order_buf_t := structure {int tail, int mode[], int addr[], int IC[]}</p>	
<p>VARIABLES</p> <p>timestamp_t current_ts; // each thread maintains a current timestamp, initialized to 0</p> <p>block_history_t the_block; // access history of the block being accessed</p> <p>ld_st_order_buf_t ordering_buf; // a load store ordering buffer</p>	
<p>WHEN A WRITE OF THREAD I IS ORDERED</p> <p>foreach j in all_threads if (j != i) <i>// log values if we steal from j's load</i> if (the_block in j's ordering_buf && may cause an SC-violation) write_value(j, ...) <i>// may log the WAR separately</i> foreach < a_reader, a_ts > in the_block.readers if (i != a_reader) <i>// found a WAR conflict</i> write_log(a_reader:a_ts, i:ordering_buf.HEAD().IC) if (i != the_block.writer) <i>// found a WAW conflict</i> write_log(the_block.writer:the_block.w_ts i:ordering_buf.HEAD().IC) <i>// update the block's access history</i> clear(the_block.readers) the_block.writer := i the_block.w_ts := ordering_buf.HEAD().IC until (IS_A_STORE(ordering_buf.HEAD())) ordering_buf.POP()</p>	<p>WHEN A READ OF THREAD I EXITS ORDERING_BUF</p> <p>if (i != the_block.writer) <i>// found a RAW conflict</i> write_log(the_block.writer:the_block.w_ts i:ordering_buf.HEAD().IC) <i>// update the block's access history</i> if (the load value was not logged) <i>// avoid WAR for the SC-violating loads</i> add(the_block.readers, < i, ordering_buf.HEAD().IC >)</p> <p>WHEN AN INSTRUCTION INSTR COMMITS</p> <p>ordering_buf.tail := ordering_buf.tail + 1 if (IS_MEM_OP(instr)) ordering_buf.PUSH(instr.mode, instr.addr, ordering_buf.tail)</p>

est IC) that has been committed¹. When a store instruction is ordered (exits from the write buffer), the algorithm first checks if the store overwrites any memory location currently monitored by other threads, with the help from the

1. Speculative loads can be inserted in to the `ordering_buf`. In order to do so, the load needs to have an IC and a memory address and `ordering_buf` needs support the insertion. As Table 3-4 shows, the load instructions in the `ordering_buf` do not generate “real” WAR dependencies, but the load value may be logged. In the cases of misspeculation, the logged value may be dropped. For more information about speculative instructions, please make a forward reference to the next chapter.

`ordering_buf`. In the affirmative case, the algorithm logs the loaded value. Then, the algorithm detects WAR and WAW conflicts using the IC of the head instruction of the `ordering_buf`, instead of the `current_ts` variable. It is important to recall that the store instruction is at the head of the `ordering_buf`. Finally, the algorithm pops the `ordering_buf` until the next unordered store instruction is at the head (if any). A load instruction updates the access history of a memory block when the load instruction is pop from the `ordering_buf`. Any RAW conflict caused by this load instruction is also detected and logged at the same time¹.

We now discuss some of the properties of the hybrid recording algorithm.

This order-value-hybrid recording method does not force TSO executions to conform to the SC model. Instead, the new method logs additional values to deal with the potential deadlocks during the in-order replay.

This order-value-hybrid recording method should (at most) slightly increase the log size of the recorder. Several studies have shown that, even under the consistency models that are weaker than TSO, cache lines of load instructions are rarely stolen away before the all preceding instructions are ordered [11, 23]. Therefore, the load values are infrequently logged.

We now reason about the correctness of this new hybrid recording method without considering TR and RTR. In the unoptimized case, the recorder logs all

1. In hardware, the RAW conflict is detected when a coherence transaction happens, *i.e.*, when the load reads the memory (Section 4.2.2). Therefore, in the hardware TSO recorder, the RAW conflict needs to be *buffered* until the load instruction pops out of `ordering_buf`. For more information about conflict detection, please make a forward reference to the next chapter.

conflicts except those WAR conflicts related to the SC-violating loads. The recorder omits a subset of the WAR dependencies during recording. Omitting WAR dependencies does not affect the correctness of the thread that wrote the memory location (the write is still performed in the right order with respect to other writes). It does affect the load value of the reading thread, because removing the WAR dependence causes the load instruction to see a new version of the memory location. However, our value log supplies the correct value to the load instruction. Therefore, all load instructions get the same values in the original and the replay executions. As a result, the hybrid recording method provides a successful replay for both SC and TSO executions.

In the next two subsections, we extend the hybrid recording method to the TR and RTR recording methods. The basic principle is that we suppress the TR and RTR optimizations for the potential SC-violating loads. Therefore, TR and RTR continue to work correctly for the remaining instructions.

3.6.3 Extending the TR Algorithm from SC to TSO

Now, assuming TR is applied, applying TR to the WAR dependencies, which are later omitted, is incorrect. As shown in Figure 3.9, the dependence $i:wr(C) \rightarrow j:wr(C)$ is transitively reduced by another WAR dependence ($i:rd(B) \rightarrow j:wr(B)$). However, if the WAR dependence is omitted in the recording to avoid replay deadlocks, an incorrect replay order can violate the dependence $i:wr(C) \rightarrow j:wr(C)$.

Fortunately, it is easy to solve this problem by not using those WAR dependen-

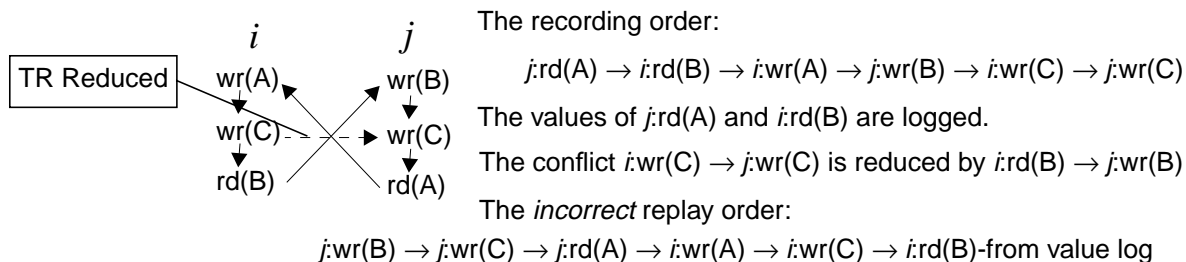


Figure 3.9 TR and TSO executions. Applying TR to TSO executions directly can cause incorrect replay. The dependence $i:wr(C) \rightarrow j:wr(C)$ is transitively reduced by another WAR dependence ($i:rd(B) \rightarrow j:wr(B)$). However, because the WAR dependence is removed in the replay to break the cycle of dependencies, as shown by the incorrect replay order, the dependence $i:wr(C) \rightarrow j:wr(C)$ is not satisfied. Our solution is to avoid using those removable WAR dependencies in transitive reduction in the recording.

cies in transitive reduction in the recording. In Table 3-4, the WAR dependencies of the potential SC-violating loads are detected when the loads are still in `ordering_buf`, not via the access history of memory locations. In Table 3-2, however, TR is achieved via the vector timestamps stored in the access history of memory locations. Therefore, the same reduction algorithm in Table 3-2 can be used in Table 3-4, which will automatically exclude the WAR dependencies of the potential SC-violating loads from being used in TR.

3.6.4 Extending the RTR Algorithm from SC to TSO

We now extend the RTR optimization to support the hybrid recording method. Recall that RTR relies on the SC total order to avoid creating replay deadlocks by overly-strict dependencies. In TSO executions, there is not an SC total order. To extend RTR to TSO execution, we modify RTR so that it creates the stricter dependencies more conservatively to avoid replay deadlocks.

In Table 3-3, RTR assumes the SC memory model. The variable `current_ts` stores the IC of the most recently committed instruction. Under the SC model, all

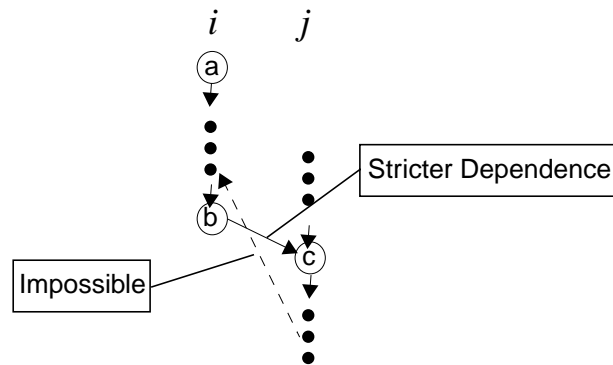


Figure 3.10 RTR and TSO executions. After modifying RTR for TSO execution, RTR creates stricter dependencies within a window $[i:a, i:b]$, where all instructions before $i:b$ in thread *i* have been ordered. This way, there must not be any cycle of dependencies, because such dependence like the one shown in the dashed arrow, is impossible.

memory instructions committed before `current_ts` have already been ordered.

In a TSO system, however, instructions in the write buffer are committed but not yet ordered. When a conflict is detected in TSO executions, if RTR uses the conflicting timestamp (e.g., `a_ts[a_reader]`) and the most recently committed timestamp (e.g., `reader_thread.current_ts[a_reader]`, which corresponds to `reader_thread.order_buf.TAIL().IC.`) to compute the sliding window, this sliding window allows the RTR algorithm to later choose a dependence sourced from a load instruction that is still in the `ordering_buf`. This effect is similar to the effect when a loaded memory location is overwritten, which may result in SC violations and requires the recorder to log the loaded value.

Therefore, directly applying RTR on TSO executions can create replay deadlocks or (substantially) increase the number of load values to be logged in the hybrid recording method. Fortunately, this can be fixed by modifying the RTR algorithm so that it does *not* “expose” the load instructions in the `ordering_buf`

prematurely. In order to do so, we modify the RTR algorithm so that RTR computes the sliding window less aggressively. Instead of using the `reader_thread.current_ts[a_reader]` variable, RTR uses the `reader_thread.ordering_buf.HEAD().IC` in computing the sliding windows. This way, the sliding window computed based on this new variable can not possibly “expose” the load instructions in the `ordering_buf`. This modified RTR algorithm sacrifices some freedom in introducing the stricter dependencies, but avoids logging more load values.

We use Figure 3.10 to argue the correctness of this modification. With this modification, RTR creates stricter dependencies, say from the window $[i:a, i:b]$ to $j:c$, at the time when $j:c$ exits the `ordering_buf` and when the conflict $i:a \rightarrow j:c$ is detected. Instruction $i:b$ is the new value stored in the `ordering_buf.HEAD().IC` variable. We show the correctness by a contradiction. Suppose RTR creates an overly-strict dependence that causes a replay deadlock. This means an instruction (of thread i) before $i:b$ must depend on an instruction (of thread j) after $j:c$. Because all instructions (of thread i) before $i:b$ have already exited the `ordering_buf`, therefore, the instructions they depend on must have been out of the `ordering_buf` of thread j . The instructions before $i:b$ cannot depend on any instruction after $j:c$, because instructions after $j:c$ are either in the `ordering_buf` or not yet executed. Therefore, the modified RTR algorithm cannot create overly-strict dependencies that causes replay deadlock on TSO executions.

We have only shown the correctness on a 2-thread example. For more than two

threads, the key is still showing the absence of cycles. We can always merge all but one thread into an encapsulating “virtual” thread, i.e., in 3-thread case, any of the two threads can be merged into a virtual thread, because there is no cycle in the 2-thread case. Then, the cycle-freedom can be shown with the virtual thread and the remaining thread.

Thus, despite of the lack of the SC total order, our modified RTR algorithm relies on creating stricter dependencies more conservatively to avoid replay deadlocks for the TSO executions.

3.6.5 Race Recording on PC and Other Relaxed Consistency Models

In addition to SC and TSO memory consistency models, some multiprocessor architectures support Processor Consistency (PC) and other relaxed consistency models [1]. PC differs from TSO in that PC allows different processors to observe different write orderings, which violates the store atomicity [4]. Like the load_P instructions of TSO, PC’s store atomicity violations can cause replay deadlocks if a race recorder records only the interthread orderings. Furthermore, other relaxed consistency models that allow write-to-write, read-to-read and read-to-write instruction reordering can cause replay deadlocks, because of more freedom in the instruction recording.

Our order-value-hybrid recording method cannot be directly applied to PC and other relaxed consistency models for two reasons. First, in PC, the detection of a store atomicity violation is potentially expensive, because precise detection requires information from multiple processors. If the detection is made efficient,

however, the problematic loads can be again value-logged to enable successful replay. Second, in other relaxed consistency models, value-logging load instructions is not sufficient in avoiding replay deadlocks, because of write-to-write reordering. We leave supporting PC and other relaxed consistency models as an open problem.

3.7 Discussion: Generalized RTR

To close this chapter on race recording algorithms, we provide a discussion of a generalized RTR algorithm framework. The content presented in this section does not affect understanding for the other parts of this dissertation. Therefore, this section can be skipped by the eager readers.

3.7.1 Generalized RTR

We observe that there are three relatively discrete components in the RTR algorithm:

1. RTR chooses to use dynamic instruction count (IC) as the logical time base. Using IC to distinguish each dynamic instructions is very natural. Nothing needs to be recorded about the ICs, they are automatically recreated during replay.
2. RTR uses a heuristic to introduce stricter and vectorizable dependencies. The goal of these dependencies is to ensure the same set of conflicts are recreated during replay.
3. RTR relies on the SC total order to prevent replay deadlocks from being created by overly-strict dependencies.

Each of these three components can be replaced by some other mechanism. By doing so, a new instance of the RTR algorithm can be created. In fact, these three components lead us to a generalized algorithm framework. Within this framework, a group of RTR algorithms can be developed.

To design a new recording algorithm in this framework, we need to answer three design questions, corresponding to the three components of RTR algorithm above:

1. What is the logical time base?
2. How to introduce stricter dependencies?
3. How to prevent replay deadlocks?

To answer question (1), we recognize IC is not the only possible logical time base. As we will see in the next section, Lamport Scalar Clock [30] can be used as a logical time base as well. In an extreme case, even a synchronized *physical* clock can be used as the time base. However, different logical time bases incur different levels of recording overhead. For example, IC requires nothing to be recorded for the logical time, but Lamport Scalar Clock requires every discontinuity of the clock to be recorded (recall the ROLT algorithm in Chapter 2). Even higher recording overhead is introduced by using the synchronized physical clock, which requires us to record the physical time when each dynamic instruction is executed.

More importantly, how we choose the logical time base affects how we record conflicts. For example, if we use the IC as the logical time base, there is no inherent interthread ordering provided by the IC (IC does provide intrathread order-

ing). Therefore, we are required to record some information about the conflicts in the execution. On the other hand, using *Lamport Scalar Clock* or a synchronized physical clock, the ordering among conflicting instructions are captured by the time base. Therefore, it is not necessary to record extra information about the conflicts to enable faithful replay.

Assuming we have chosen a logical time base that requires conflicts to be recorded, we now have to answer question (2). Using different heuristics to introduce the stricter dependencies affects how much information we have recorded for the conflicts and how much replay parallelism is available during replay. For example, instead of using our heuristic to introduce vectorizable dependencies, we can use another heuristic that always introduces the strictest dependencies allowed by the SC total order. This new heuristic introduces potentially more waiting time during replay, hence allowing less replay parallelisms. But, depending on the program behavior, this heuristic may enable more reduction in the total log size.

To answer question (3), we need to recognize that replay deadlock happens only when there is a cycle of dependencies. Our method of deadlock avoidance in the RTR algorithm is a conservative solution. For example, if more computation cost is allowed, a cycle detection algorithm can be used to detect and avoid cycles during recording. Database systems have long used such algorithms in their concurrency control manager to avoid cycles of dependencies among in-flight transactions.

Finally, the three questions are not independent. An algorithm designer

should carefully choose solutions to each of the questions with an overall consideration of recording runtime and memory overhead, replay parallelism, and recording efficiency (the log size).

3.7.2 ROLT as a Special Case of the Generalized RTR

The RTR algorithm framework provides a unified understanding of partial order recording algorithms in this chapter and the ROLT recording algorithm [33]. In particular, ROLT can be obtained by:

1. Using the Lamport Scalar Clock as the logical time base.
2. Making every stricter dependence with an IC stride equals to one. Because all dependencies have the same IC stride and every dynamic instruction has an incoming dependence, no dependence information needs to be recorded. All the strict dependencies essentially force the partial order executions to be recorded as total order executions.
3. Using the Lamport Scalar Clock to establish a total order to avoid replay deadlocks. This is similar to our method of using the SC total order to avoid replay deadlocks.

Finally, we want to mention that a variation of the Lamport Scalar Clock, which helps reduce the log size of a ROLT race recorder. Figure 3.11 shows the so called “Synchronizing Lamport Scalar Clock” logical time scheme. Normally, when a Lamport clock adjustment is needed, we increase the clock to 1 more than the maximum timestamp of the preceding instructions. This is shown in Figure 3.11 with the conflict from thread 1’s instruction 1 to thread 2’s instruc-

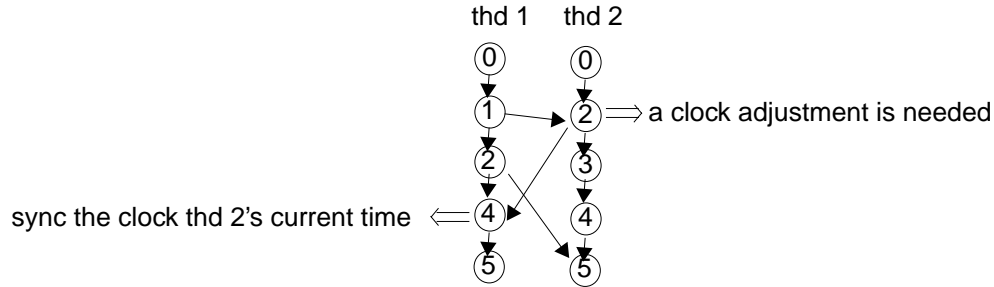


Figure 3.11 Synchronizing Lamport scalar clock. Normally, when a Lamport clock adjustment is needed, we increase the clock to 1 more than the maximum timestamp of the preceding instructions. This is shown in the figure with the conflict from thread 1's instruction 1 to thread 2's instruction 2. However, any positive adjustment is correct. In particular, with Synchronizing Lamport Scalar Clock, we adjust the clock ahead to the timestamp of the instruction *currently* executed by the other thread. For instance, for the conflict between thread 2's instruction 2 to thread 1's instruction 5, we adjust the clock ahead to 4 for thread 1, although normal Lamport clock will only adjust it to 3.

tion 2. However, any positive adjustment to the clock is correct. In particular, with the Synchronizing Lamport Scalar Clock, we adjust the Lamport clock ahead to the timestamp of the instruction currently executed by the other thread. As shown by the conflict between thread 2's instruction 2 to thread 1's instruction 5, we adjust the clock ahead to 4 for thread 1, although the normal Lamport clock only adjusts to 3. Our experiments on commercial workloads show this change in the Lamport Scalar Clock reduced the log size by up to 20% for ROLT based race recorders.

Chapter 4

Reduce the Runtime Overhead: Hardware Assistance

In the last chapter, we presented the RTR log-reduction algorithm, which is a key to reducing the log size, *i.e.* recording long executions with small logs.

However, the RTR algorithm can incur significant overheads, which can be seen from the algorithm pseudo code (Table 3-3). Not only does RTR monitor *every* memory read and write instruction, which is likely to incur prohibitive runtime overhead, but it also requires several timestamps per memory location, which incur significant memory overhead by increasing a program's memory footprint by a significant factor.

In this chapter, we present how to leverage hardware cache coherence mechanisms to perform race recording that reduces the runtime overhead to a negligible level. We present techniques for implementing the RTR algorithm in hardware. Our key observation is that RTR can piggyback onto cache coherence mechanisms with few hardware changes required. In Chapter 7, evaluation results show that a hardware race recorder incurs no more than 2% runtime over-

head.

We will leave solving the memory overhead problem to the next chapter (Chapter 5). In this chapter, we temporarily assume sufficient storage space and communication bandwidth is available for the timestamps (more detail in Section 4.2.1).

We begin this chapter by describing a simplified multiprocessor system with a simple cache coherence protocol (Section 4.1). We use this simplified system as a substrate for hardware race recording. It helps us explain the most important components in hardware race recording without distractions from the less important implementation details. There are three important components in hardware race recording. First, Section 4.2 shows how to piggyback conflict detection onto this coherence protocol. Second, Section 4.3 shows how to perform *transitive reduction* in hardware. Third, Section 4.4 shows how to perform *regulated transitive reduction* in hardware. Section 4.5 extends our hardware recorder to work with more complex cache coherence hardware. Section 4.6 discusses miscellaneous hardware implementation issues for implementing a hardware race recorder. Finally, we close this chapter by discussing the open problems with our hardware race recording (Section 4.8).

4.1 A Simplified Multiprocessor System

To explain our hardware race recorder, we first describe a simple multiprocessor (MP) system. Figure 4.1 shows a 2-way configuration of the system. The system has in-order processor pipelines. Each processor has an infinite single-level

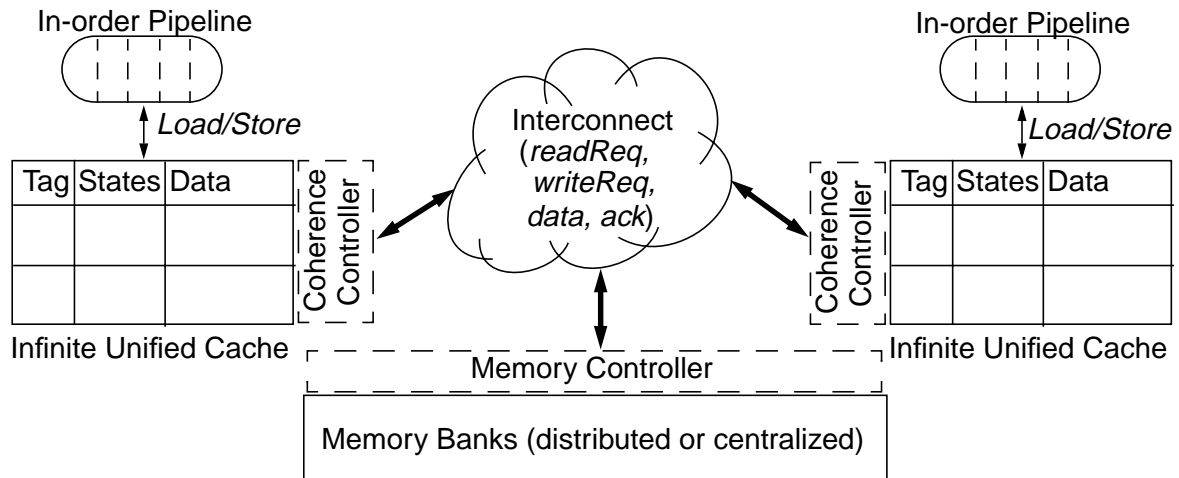


Figure 4.1 A simplified multiprocessor system. In this 2-way MP system, we assume each processor has an in-order pipeline that issues load and store requests to an infinite unified cache. The caches and the memory banks are connected via an interconnection network, where read and write requests and data messages are transmitted with respect to cache block granularity.

unified cache. The memory banks are connected to the processor caches with an arbitrary interconnection network. The system enforces the Sequential Consistency (SC) memory model by performing memory accesses atomically and in program order by issuing one memory access at a time at the commit stage and blocking on cache misses. We remove these simplifications and describe race recording with more realistic processors in Section 4.6.

4.1.1 The Cache Coherence Protocol

To maintain cache coherence, our simple system uses an invalidation-based cache coherence protocol implemented in hardware [67]. (This dissertation does not consider update-based cache coherence protocols.) The protocol consists of three stable states (MODIFIED, SHARED, and INVALID) and no transient states. The INVALID state (or I state) of a cache block signifies that the processor may neither

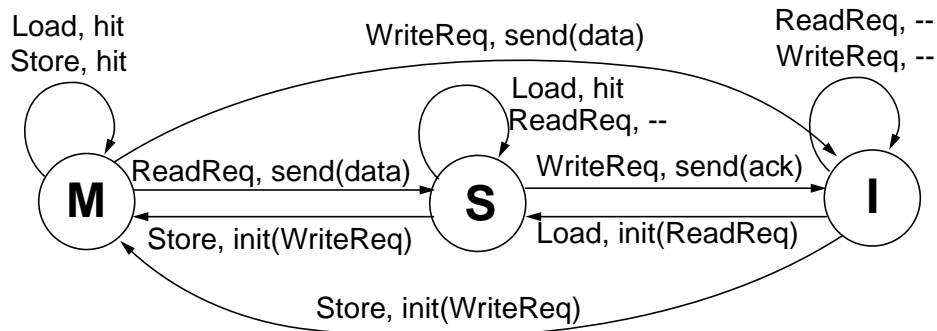


Figure 4.2 The MSI cache coherence protocol. The coherence controllers are driven by events from both the processor and the interconnection network. These events include loads, stores from the local processor, as well as read and write requests from the remote caches. Certain actions are specified for certain combinations of a cache block state and an event. The notion “A, B” represents a pair of corresponding event A and action B, which are associated with a state transition denoted by the arrows. For the actions, the notion “init()” represents initializing a broadcast request for a cache block. The notion “send()” represents sending a response message to the requester. The notions “hit” and “--” represent a cache hit and a null action, respectively.

read nor write the cache block. The SHARED state (or S state) of a cache block signifies that the processor may read but must not write the cache block. The MODIFIED state (the M state) of a cache block signifies that the processor may both read and write the cache block.

Figure 4.2 shows the state transition diagram for this basic MSI protocol. The cache coherence controllers are driven by *events* from both the processor and the interconnection network. These events include *loads* and *stores* from the (local) processor, as well as *read* and *write* requests from the (remote) caches. Based on the coherence state and the incoming event of a cache block, certain *actions* are specified by the protocol. In Figure 4.2, the notion “A, B” represents a pair of corresponding event A and action B, which are associated with a state transition denoted by the arrows. For the actions, the notion “init()” represents initializing a request for a cache block. The notion “send()” represents sending a response mes-

sage to the requester. The notions “hit” and “--” represent a cache hit and a null action, respectively.

This MSI protocol is simple, but unrealistic in the following aspects: (1) the caches are infinite (no cache evictions); (2) use explicit acknowledgement messages for S to I cache state transitions; (3) do not use the OWNED and the EXCLUSIVE states; (4) all coherence transactions are atomic (no transient states); and (5) single level cache hierarchy. In Section 4.5, we systematically remove these idealizations and describe race recording on more realistic protocols.

4.2 Unoptimized Hardware Race Recorder

We now examine how to implement the basic unoptimized race recording algorithm (Table 3-1) in hardware. There are three questions which need to be answered: (1) how to compute IC and store timestamps in hardware; (2) how to detect conflicts in hardware; (3) how to log dependencies in hardware. This section first answers these questions and then presents a hardware algorithm.

4.2.1 Instruction Count (IC) and Timestamps

Figure 4.3 shows the necessary hardware components for the unoptimized hardware race recorder. The addition of the IC register is straightforward: it gives each dynamic instruction a unique number, functioning just like the `current_ts` variable in Table 3-1. (We assume each thread is bound with each processors. This is a limitation of our hardware race recorder. We discuss this in more detail in Section 4.8.)

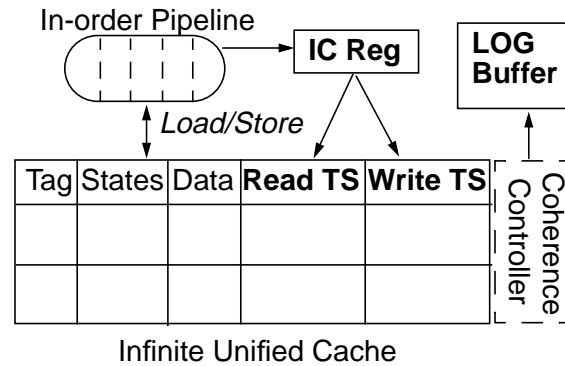


Figure 4.3 The IC register and the read/write timestamps. To compute IC in hardware, we add an IC register at the commit stage of the processor pipeline to uniquely identify each dynamic instruction. To provide speedy accesses to the timestamps, we put read and write timestamps along with the cache blocks themselves. The log buffer temporarily stores the dependences recorded by the recorder.

In Table 3-1, the unoptimized algorithm keeps the reader and writer $\langle \text{ID}, \text{timestamp} \rangle$ pairs in a “centralized” data structure (`block_history_t`). There is one such data structure for every memory block. Naïvely implementing this data structure in hardware results in slow timestamp accesses and is unnecessary. Instead, we distribute the timestamps among processors: distributing reader timestamps among the readers and the writer timestamps with the writers, respectively. In particular, we add two read and write timestamps to each cache block. The read (write) timestamp records the last dynamic instruction that read (wrote) the cache block. The IDs of the readers and the writers are implicitly encoded by which cache contains the timestamps. The timestamps do not move with cache block from cache to cache. Nor do timestamps need to be stored in the memory because we assume infinite caches.

This distributed design has two benefits. First, we do not need to explicitly store the processor ID in the timestamp data structure. Second, processors can

quickly access the timestamps that are stored in the local cache. As we will see shortly, processors never need to directly access the timestamps that are stored in remote caches.

For simplicity, we can merge the read and write timestamps into one so called *last access* timestamp. The last access timestamp signifies which dynamic instruction last read *or* wrote the cache block. Using the last access timestamp to approximate the read and write timestamps is correct but conservative. From now on, we use the last access timestamp to replace the read and write timestamps in the rest of this chapter. In Chapter 5, we explore the impact from this and other timestamp approximation techniques in more detail.

4.2.2 Conflict Detection

After computing the IC and storing the timestamps, the hardware race recorder needs to detect conflicts between dynamic instructions from different processors. Figure 4.4 shows how we achieve this in the MSI coherence protocol.

In the MSI protocol, three state transitions, namely $M \rightarrow S$, $S \rightarrow I$ and $M \rightarrow I$, correspond to three types of conflicts we want to detect: RAW (Only the first RAW between two consecutive writes to a block is detected here. Subsequent RAW conflicts can be detected with additional states that are kept in the cache and the memory. We discuss the modifications to detect the rest of RAW conflicts in more detail in Section 4.5.3.), WAR, and WAW. As shown in the bold arrows in the figure, when those state transitions happen, the responder sends a data message or an acknowledgement message to the requester. Both the sending and the receiv-

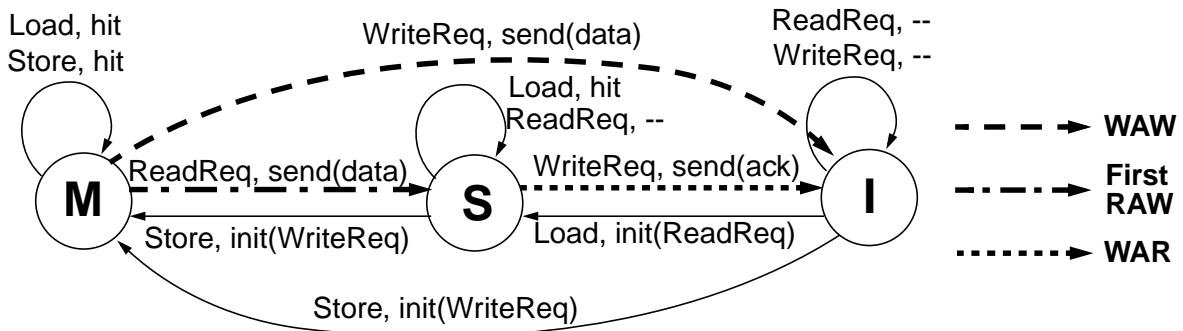


Figure 4.4 Conflict detection in the MSI protocol. The bold arrows represent protocol transitions that indicate conflicts between processors. Both the sender and the receiver processors can observe the conflict. For most of these transitions, explicit response messages are sent between processors involved in the conflicts.

ing ends of these messages can detect the conflict. For example, when the requester i receives the data or the acknowledgement message from a responder j , i detects a conflict from a dynamic instruction of j to i 's next committing instruction $i:IC+1$, which will access the cache block. The problem is that i also needs to know the IC of the dynamic instruction of j . To do that, j looks up the timestamp of the cache block and piggybacks the timestamp when j sends the data or the acknowledgement to i . Similarly, in order to allow the responder j to detect the conflict, the requester i must piggyback the timestamp ($IC+1$) when sending the request to j . We call these two detection strategies *requester detection* and *responder detection*, respectively.

4.2.2.1 Requester Detection versus Responder Detection

Fortunately, the same conflict needs to be detected (and logged) only once. A recorder must choose between requester detection and responder detection. The choice is based on the convenience and efficiency of the piggybacking. For

requester detection, the timestamp is piggybacked on the response messages. These messages are often sent from point to point. Therefore, the bandwidth consumption of the piggybacking is small. Unfortunately, not all coherence protocols include explicit acknowledgements. For responder detection, the timestamp is piggybacked on the request messages, which are either broadcasted on a bus or forwarded to a directory. All coherence protocols should include request messages, but piggybacking timestamps on request messages can consume more bandwidth or incur more design complex (because of the forwarding).

In this dissertation, we choose requester detection, because our baseline coherence protocol has explicit acknowledgements and we want our hardware recorder to be simple and bandwidth efficient. In sections 4.3–4.6, we design other implementation techniques (*e.g.*, TR and RTR) assuming requester detection. We believe these implementation techniques can apply to responder detection with appropriate modifications.

4.2.2.2 *Word Conflict versus Block Conflict*

Thus far, our hardware recorder detects conflicts on cache block granularity, herein called *block conflicts*. In Table 3-1, however, the unoptimized recording algorithm detects conflicts on word granularity, herein called *word conflicts*. Block conflict is a conservative approximation to word conflict, *i.e.*, block conflict may introduce extra dependencies between instructions, but it never misses any dependence otherwise required by word conflict. The correctness of using block conflicts to approximate word conflicts is established through transitivity. A block

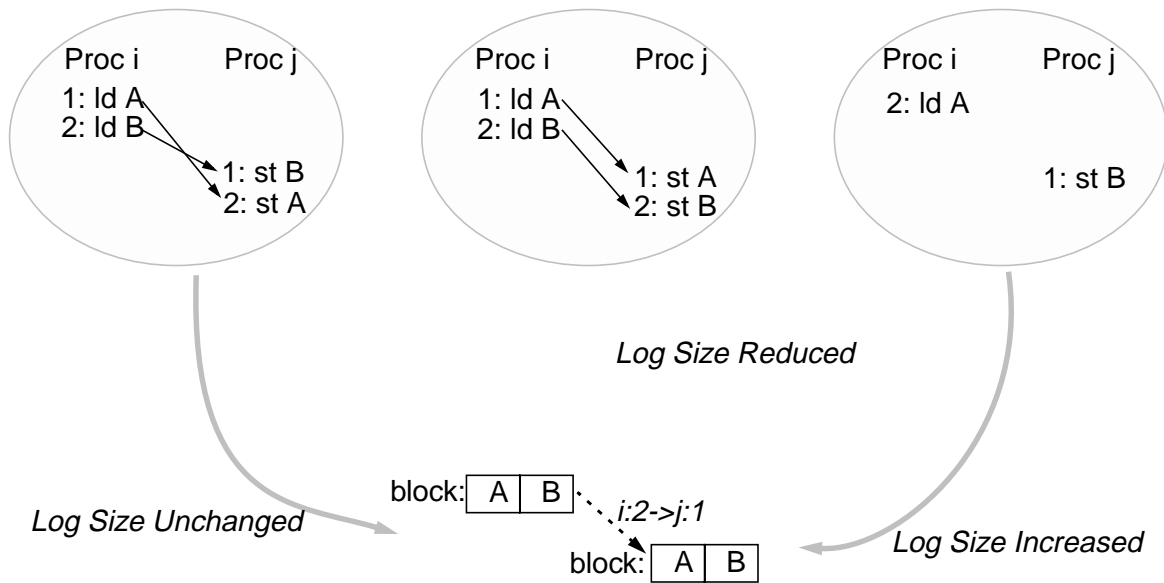


Figure 4.5 Three cases in word-conflict-versus-block-conflict. The correctness of using block conflicts to approximate word conflicts is established through transitivity. The impact to log size from this approximation is mixed. For example, assuming A and B are two words residing in the same cache line. The solid arrows represent word conflict. The dotted arrows represent block conflict. Logging the block conflict can have no impact on, increase or reduce the log size.

conflict dictates the dependence between the last access of a processor to the first access of another processor. Any word conflict associated with the block are implied by the block conflict.

Logging the block conflicts can have no impact on, increase or reduce the log size. Figure 4.5 shows an example for each of the cases: (1) one of the two word conflicts are reduced and remaining word conflict is identical to the block conflict; (2) two word conflicts are replaced by one block conflict; (3) an extra block conflict is logged. In all three cases, the block conflict records a dependence from $i:2$ to $j:1$, which is a sufficient dependence to recreate the original word conflicts. In fact, the block conflict is a stricter dependence to some word conflicts. Its correctness is provided by both being stricter and not overly-strict. For brevity, we omit

the arguments on why block conflicts are not overly-strict.

4.2.3 Dependence Logging

After conflict detection, a hardware recorder needs to store the dependencies into its log. Unlike the unoptimized algorithm in Table 3-1, our hardware recorder cannot directly store the dependence log in a log file. Instead, we add a hardware log buffer at each processor to temporarily store the dependence log.

Figure 4.3 shows the hardware log buffer. The buffer is essentially a FIFO and is written by the coherence controller when a conflict is detected. There are at least two possible methods to empty the buffer. First, the hardware recorder can interrupt the operating system when the buffer is full. The interrupt handler then moves the log data from the buffer to virtual memory (perhaps through a DMA). Second, the recorder can reserve a special region from the physical memory for each processor. The recorder then uses hardware to lazily move the data from the buffer to the physical memory. Once the log data is stored in (virtual or physical) memory, a software handler can save the data to a log file.

The read-latency of the log buffer is not critical. Therefore, the log buffer can be banked and pipelined to meet aggressive cycle-time requirements. We do not further discuss the details of the log buffer, because the log buffer is not a critical component of the hardware race recorder.

4.2.4 Put-it-together: the Hardware Algorithm

Table 4-1 shows the hardware algorithm for unoptimized race recording. We augment the processor core with IC, CTS, and LOG. On each commit of memory

Table 4-1 State and Actions for Processor j with Unoptimized Recording

<p>STATE AT EACH PROCESSOR J</p> <p>IC: Instruction count of last dynamic instruction committed at processor j</p> <p>LOG: Unbounded log buffer at processor j</p> <p>M: total number of memory blocks</p> <p>CTS[M]: Cache Timestamp (TS): CTS[b] is the timestamp of last load or store of block b</p>
<p>ACTIONS AT EACH PROCESSOR J</p> <p>On commit of instruction x { IC++ // After, IC is x's dynamic instruction count if (IS_A_LOAD_OR_STORE(x)) { // b must be cached before x can commit, x.block is the cache block accessed by x CTS[x.block] := IC } }</p> <p>On sending coherence response for block b to processor k { // Conflict begins at processor j's last instruction that accessed b, which is j:CTS[b] ID := j last_access_TS := CTS[b] SEND(ID, last_access_TS, ... data/ack ...) }</p> <p>On receiving coherence response for block b from processor i { // Conflict ends at the next instruction committing at processor j, which is j:(IC+1) (ID, last_access_TS, ... data/ack ...) := RECEIVE() // Log this conflict, which is ID:last_access_TS \rightarrow j:(IC+1) LOG.APPEND(ID, last_access_TS, IC+1) }</p>

instructions, we increase the IC by one and update the corresponding timestamp in the cache. When sending a coherence response to a requester, we look up the timestamp from the cache and piggyback it to the response data or acknowledgment message. Finally, when a coherence response arrives, we log the dependence in the log buffer, based on the piggybacked timestamp.

4.3 Hardware TR Recorder

Clearly, the unoptimized algorithm records one or more dependence(s) for each coherence transaction. For modern MP systems, logging dependencies at this rate can quickly fill a log buffer several MB in size, resulting in large logs

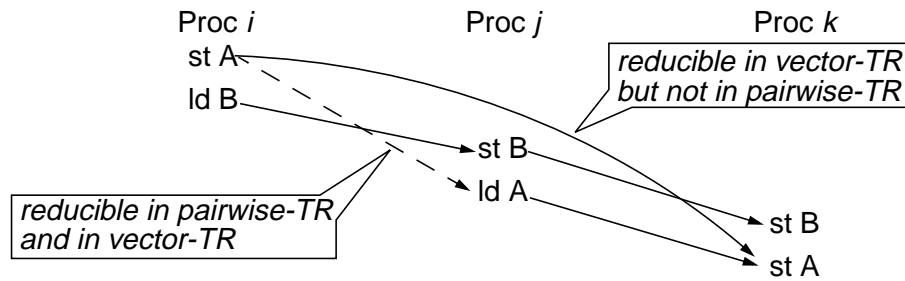


Figure 4.6 Pairwise-TR versus vector-TR. In vector-TR, transitivity works across multiple processors (e.g., from processor i to j then to k). In pairwise-TR, we can discover only the transitivity among every pair of processors. The benefits of pairwise-TR is that only *scalar* timestamps need to be stored and communicated for race recording.

and significant runtime overhead. This section describes a hardware implementation of transitive reduction, which reduces the log significantly.

In Table 3-2, “perfect” transitive reduction needs to utilize vector timestamps. Not only are the vector timestamps used for the IC, they are also recorded for each memory block. Naïvely implementing the vector timestamp based transitive reduction (herein called *vector-TR*) is prohibitively expensive. The costs are mainly two folds: (1) all cache timestamps now need to be an n -ary vector, where n is the total number of processors; (2) the coherence messages now need to carry vector timestamps, which can significantly increase the bandwidth consumption (and possibly cause network congestion).

4.3.1 Pairwise-TR

Our idea is to approximate this vector-TR with *pairwise-TR*, which avoids the two costs detailed above by sacrificing minimal reduction opportunities. Figure 4.6 shows the difference between pairwise-TR and vector-TR. In pairwise-TR, only *scalar* timestamps need to be stored and communicated between processors. In particular, when receiving a coherence response from a remote processor,

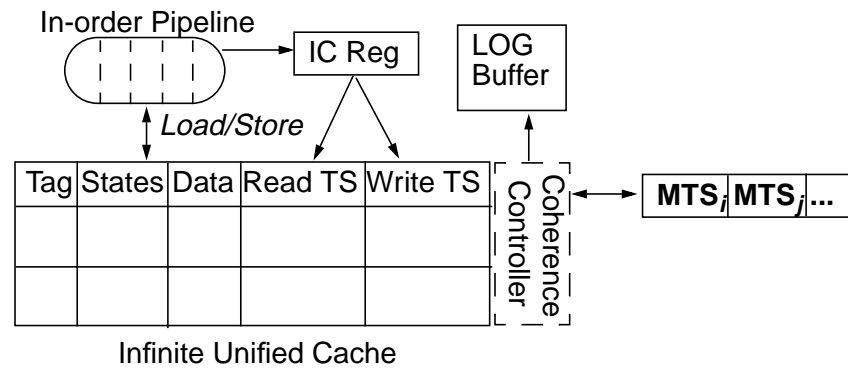


Figure 4.7 The MTS registers. The MTS (Maximum TimeStamps) registers contain the maximum timestamps a processor received from another processor. For a system with N processors, each processor needs a MTS with size $(N-1)$.

the coherence controller records the maximum timestamp received from the remote processor. In the future, by comparing the received timestamps with the maximum timestamp, we can discover transitivity for the conflicts between the pair of processors. Without vector timestamps, however, we cannot discover transitivity across more than two processors.

Figure 4.7 shows how we augment the hardware for pairwise-TR. We add a group of timestamp registers, called *Maximum Timestamp* (MTS) registers. MTS registers contain the maximum timestamps a processor received from another processor. For a system with N processors, each processor needs $(N-1)$ MTS registers. It is important to note that we do not need to expand the cache timestamps into vectors, nor do we need to expand the IC register into a vector.

4.3.2 Put-it-together: the Hardware Algorithm

Table 4-2 shows the hardware algorithm for TR race recording. The differences over the unoptimized algorithm are marked in bold. We add a simple numerical comparison to the action when a coherence response is received. If the

Table 4-2 State and Actions for Processor j with TR

<p>STATE AT EACH PROCESSOR J</p> <p>IC: instruction count of last dynamic instruction committed at processor j</p> <p>LOG: log buffer of processor j</p> <p>M: total number of memory blocks</p> <p>CTS[M]: cache Timestamp (TS): CTS[b] is the timestamp of last load or store of block b</p> <p>N: total number of processors</p> <p>MTS[N]: an array of maximum timestamps received from other processors</p>
<p>ACTIONS AT EACH PROCESSOR J</p> <p>On commit of instruction x {</p> <p style="padding-left: 20px;">IC++ // After, IC is x's dynamic instruction count</p> <p style="padding-left: 20px;">if (IS_A_LOAD_OR_STORE(x)) {</p> <p style="padding-left: 40px;">// b must be cached before x can commit, x.block is the cache block accessed by x</p> <p style="padding-left: 40px;">CTS[x.block] := IC</p> <p style="padding-left: 20px;">}</p> <p>}</p> <p>On sending coherence response for block b to processor k {</p> <p style="padding-left: 20px;">// Conflict begins at processor j's last instruction that accessed b, which is j:CTS[b]</p> <p style="padding-left: 20px;">ID := j</p> <p style="padding-left: 20px;">last_access_TS := CTS[b]</p> <p style="padding-left: 20px;">SEND(ID, last_access_TS, ... data/ack ...)</p> <p>}</p> <p>On receiving coherence response for block b from processor i {</p> <p style="padding-left: 20px;">// Conflict ends at the next instruction committing at processor j, which is j:(IC+1)</p> <p style="padding-left: 20px;">(ID, last_access_TS, ... data/ack ...) := RECEIVE()</p> <p style="padding-left: 20px;">if (last_access_TS > MTS[i]) { // "Pairwise-TR" using the MTS registers</p> <p style="padding-left: 40px;">LOG.APPEND(ID, last_access_TS, IC+1)</p> <p style="padding-left: 40px;">MTS[i] := last_access_TS</p> <p style="padding-left: 20px;">}</p> <p>}</p>

last_access_TS is larger than the current MTS, we detect an irreducible conflict and we log the corresponding dependence. After that, the MTS is updated. After this optimization, only a small fraction of coherence transactions causes a dependence to be logged in the log buffer. Only a small log buffer (e.g., 32KB) is needed to absorb the burstiness of the dependence log.

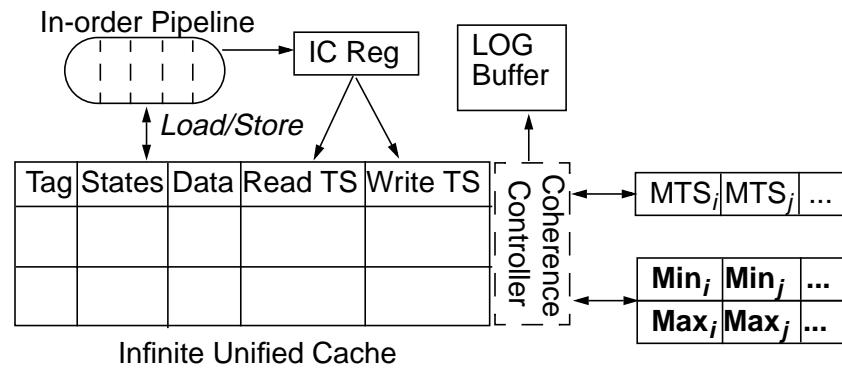


Figure 4.8 The sliding window registers. These registers record the minimal and maximal *IC stride* for the vectorizable dependencies. A pair of registers are read only when an irreducible dependence is logged. The pair of registers are written only when the irreducible dependence has a narrower (or new) sliding window.

4.4 Hardware RTR Recorder

4.4.1 The Sliding Windows of IC Stride

To further reduce the size of the dependence log, we implement the RTR algorithm in hardware as well. Recall in Table 3-3, the RTR algorithm uses a “sliding window” to compute stricter and vectorizable dependencies on-the-fly. The sliding window is represented by the maximum and minimum IC stride of the current group of stricter and vectorizable dependencies. In other words, a window defines a range of stricter and vectorizable dependencies, which can be introduced by the recorder without incurring any potential deadlocks in the replay. Figure 4.8 shows the additional hardware needed for the RTR algorithm. Again, we only incrementally change the hardware from TR to RTR. Each processor has (N-1) sliding windows: each corresponds to a remote processor in the system.

4.4.2 Put-it-together: the Hardware Algorithm

Table 4-3 shows the hardware algorithm for RTR. The changes from the hard-

Table 4-3 State and Actions for Processor j with RTR

<p>STATES AT PROCESSOR J</p> <p>IC: instruction count of processor j</p> <p>LOG: log buffer of processor j</p> <p>CTS[M]: last access timestamps of blocks in j's cache. M is the total number of blocks</p> <p>MTS[N]: a vector of maximum timestamp received. N is the total number of processors.</p> <p>$\Delta_min[N]/\Delta_max[N]$: sliding minimal/maximal IC stride for vectorizable dependences from another processor to j.</p>
<p>ACTIONS AT PROCESSOR J</p> <p>On commit of instruction x {</p> <p style="padding-left: 20px;">IC++ // After, IC is x's dynamic instruction count</p> <p style="padding-left: 20px;">if (IS_A_LOAD_OR_STORE(x)) {</p> <p style="padding-left: 40px;"><i>// b must be cached before x can commit, x.block is the cache block accessed by x</i></p> <p style="padding-left: 40px;">CTS[x.block] := IC</p> <p style="padding-left: 20px;">}</p> <p>}</p> <p>On sending a coherence response for block b to processor k {</p> <p style="padding-left: 20px;"><i>// Send both last access IC and processor current IC</i></p> <p style="padding-left: 20px;">ID := j</p> <p style="padding-left: 20px;">last_access_TS := CTS[b]</p> <p style="padding-left: 20px;">current_processor_IC := IC</p> <p style="padding-left: 20px;">SEND(ID, last_access_TS, current_processor_IC, ... data/ack ...)</p> <p>}</p> <p>On receiving a coherence response for block b from processor i {</p> <p style="padding-left: 20px;"><i>// 1. Drop the conflict if it is transitive reducible</i></p> <p style="padding-left: 20px;"><i>// 2. Introduce a vectorizable stricter dependence if we can</i></p> <p style="padding-left: 20px;"><i>// 3. Otherwise, log the previous groups of conflicts, create a new group</i></p> <p style="padding-left: 20px;">(ID, last_access_TS, current_processor_IC, ... data/ack ...) := RECEIVE()</p> <p style="padding-left: 20px;">min := IC - current_processor_IC</p> <p style="padding-left: 20px;">max := IC - last_access_TS</p> <p style="padding-left: 20px;">if (last_access_TS <= MTS[i]) {</p> <p style="padding-left: 40px;"><i>// This conflict is reduced by "Pairwise-TR", do nothing</i></p> <p style="padding-left: 20px;">} else if (WINDOW_OVERLAPS([min, max], [Δ_min[i], Δ_max[i]])) {</p> <p style="padding-left: 40px;"><i>// vectorizable, adjust the sliding window</i></p> <p style="padding-left: 40px;">Δ_min[i] := MAX(min, Δ_min[i])</p> <p style="padding-left: 40px;">Δ_max[i] := MIN(max, Δ_max[i])</p> <p style="padding-left: 40px;">LOG.APPEND(ID, IC) <i>// ID:?$\rightarrow$$j$:IC is logged, ? is determined in the future</i></p> <p style="padding-left: 40px;">MTS[i] := IC - Δ_max[i] <i>// update MTS for transitive reduction</i></p> <p style="padding-left: 20px;">} else {</p> <p style="padding-left: 40px;"><i>// not vectorizable</i></p> <p style="padding-left: 40px;">LOG.APPEND(Δ_max[i]) <i>// a stride is logged for last group of dependencies</i></p> <p style="padding-left: 40px;"><i>// reset the sliding window, start a new group of vectorizable dependencies</i></p> <p style="padding-left: 40px;">Δ_min[i] := min</p> <p style="padding-left: 40px;">Δ_max[i] := max</p> <p style="padding-left: 40px;">LOG.APPEND(ID, IC) <i>// ID:?$\rightarrow$$j$:IC is logged, ? is determined in the future</i></p> <p style="padding-left: 40px;">MTS[i] := IC - Δ_max[i] <i>// update MTS for transitive reduction</i></p> <p style="padding-left: 20px;">}</p> <p>}</p>

ware TR algorithm is incremental and marked in bold. There are two major differences between the TR and RTR hardware algorithms.

1. When sending a coherence response, a processor sends *both* the last access timestamp (`last_access_TS`) of the block and the processor's current IC (`current_processor_IC`), instead of just one last access timestamp as in the TR algorithm.
2. Whenever a coherence response is received by the controller, the controller first uses the MTS register to perform transitive reduction as before. When an irreducible dependence is detected, the controller computes the maximum and minimum IC strides for this dependence using the timestamp information piggybacked on the coherence response. The two IC strides form a window of stricter and vectorizable dependencies that can be introduced to replace the irreducible dependence. The controller then compares this window with the existing sliding window recorded in the sliding window registers (`Δ_min[]/Δ_max[]`). There are two subcases:

First, if the two windows overlap, the instruction of this irreducible dependence is added to the log and the existing sliding window is updated to the overlapping range of the two windows.

Second, if the two windows do not overlap, an IC stride is computed from the existing sliding window and the IC stride is written to the log to signify the “finalization” of the IC stride of the previous group of stricter and vectorizable dependencies. After the IC stride is written, a new sliding window is installed

based on the IC stride window of the new irreducible dependence.

Finally, in both of these two subcases, the MTS register is updated based on the new sliding window, assuming the least strict dependence is eventually introduced for the irreducible dependence (via $MTS[i] := IC - \Delta_{\max}[i]$).

4.5 Race Recording on Realistic Protocols

Thus far we have described the hardware race recording algorithm assuming a simple system and an idealized coherence protocol. The algorithm is relatively simple, but real hardware implementation is *never* simple. In this section, we describe how we gradually add more “features” to the idealized protocol to make it more realistic. Despite our efforts, we believe the techniques included in this section covers race recording on an important subset of real protocols, *not all* real protocols.

4.5.1 Finite Caches

We first remove the unrealistic assumption of infinite caches. With finite caches, cache lines are eventually evicted from the caches. Recall each cache line stores two types of crucial information for race recording: (1) the last access timestamps and (2) the MSI state that identifies if a processor is the writer, a reader or neither. Cache eviction causes us to lose both these two types of information. Fortunately, there are various approaches to recover and approximate them.

4.5.1.1 Missing Timestamps

Without timestamps we cannot pinpoint the dynamic instructions that accessed a cache block even after a conflict is detected for the block. In general, perfect timestamp memory incurs a significant memory overhead to our recorder. Our solution to the problem of missing timestamps is identical to our solution of reducing memory overhead of the timestamp memory. This solution is described in detail in Chapter 5. The basic idea is to use an imperfect, yet much smaller, timestamp memory to approximate the perfect (and much bigger) timestamp memory.

In this chapter, we temporarily assume the timestamp memory is not affected by the cache evictions. In this way, each processor has enough timestamp memory to remember which instruction last accessed each cache block.

4.5.1.2 Missing Coherence Transitions

Without the MSI state of a cache block, we cannot detect all conflicts. We miss conflicts because cache evictions can remove the three important state transitions ($M \rightarrow I$, $M \rightarrow S$ and $S \rightarrow I$) shown in Figure 4.4. Without these state transitions, we will not detect conflicts. The solution to this problem depends on (1) if the missing information is stored elsewhere (*e.g.*, in a directory); (2) if the missing information is not stored elsewhere, how then can it be approximated? We will now discuss three distinct cases for this problem.

MSI States Stored Elsewhere (Case: Silent Replacement). In some directory protocols, evicting a SHARED cache is silent, *i.e.*, the cache does not notify the directory

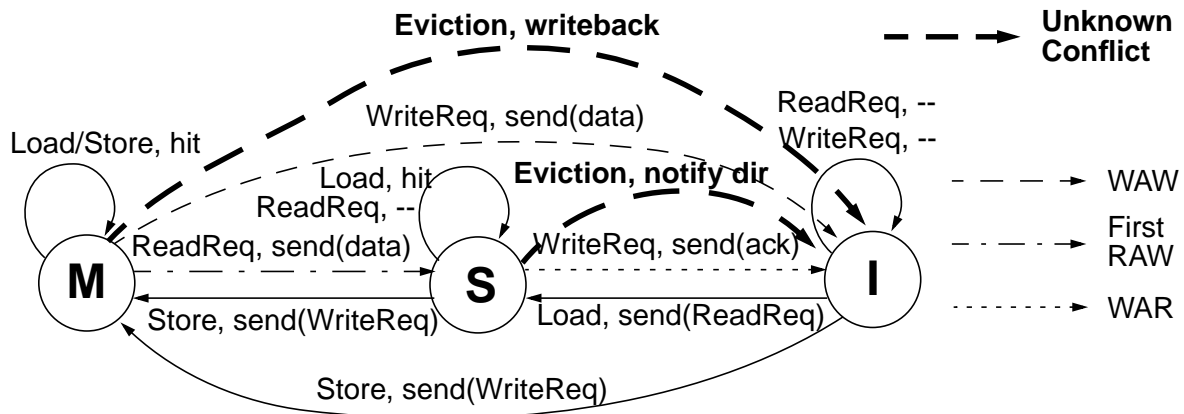


Figure 4.9 Missing conflicts due to writebacks and nonsilent replacements. When these bold state transitions happen due to cache eviction, the conflict remains undetected.

about the change. After the cache eviction, although the cache no longer records the SHARED state, the directory continues to record the same information by keeping the cache on the sharer list. Therefore, in these directory protocols, silent replacements do not cause any problem for race recording because the same conflicts will be revealed by the protocol even after the replacement.

MSI States Stored Elsewhere (Cases: Writeback and Nonsilent Replacement). Unfortunately, in directory protocols, writeback and nonsilent replacement can cause our race recorder to miss conflicts. The problem is that after writebacks and nonsilent replacements, the directory no longer records which cache possesses the cache block in MODIFIED or SHARED state. Figure 4.9 shows the cache state transitions on writebacks and nonsilent replacements. The $M \rightarrow I$ and $S \rightarrow I$ transitions of the writeback and nonsilent replacements do not reveal a conflict, because the potential future conflicting instructions have not been executed.

We solve the problem of nonsilent evictions and writebacks by adding special

“sticky” states in the directory to record the caches that once possessed a block in MODIFIED or SHARED states. The intuitive name “sticky states” came from Moore *et al.* [45]. The sticky states are reset only when a cache block is written again. With these sticky states in the directory, additional invalidation messages are sent to the caches even after they have evicted a cache block. This ensures conflicts are detected after eviction.

The additional invalidation messages and their responses can incur significant runtime overhead, because they slow down the completion of coherence transactions. The extra messages also consume more bandwidth. We fix this problem by requiring the cache to send a block’s timestamp to the directory when the block is evicted from the cache. The directory then saves these timestamps and sends the timestamps directly to a requester without using the extra invalidation messages to query the timestamps from the cache. Recall that we temporarily assume the timestamp memory at the directory is perfect and defer optimizing it until the next chapter.

Adding the sticky states to the directory requires directory entries to be allocated even after a cache block is evicted from all caches in the system. This may significantly increase the directory size. We next present an optimization that allows a directory entry to be deallocated.

MSI States *Not* Stored Elsewhere (Cases: Directory Eviction and Snooping Protocols).

Once a directory entry is evicted, or even worse, in snooping protocols that have no directory, MSI state information is permanently lost. After that, it is

impossible to recover the precise conflicts that may happen on a cache block.

Our solution is to conservatively detect the conflicts at the cost of introducing false conflicts. After a cache block is evicted from a cache, the cache cannot distinguish the block from those cache blocks that were never accessed. Our policy is to treat all cache blocks as if they have been accessed at some point in the past. (The timestamps of these blocks are approximated in the next chapter. In this chapter, since we have perfect timestamp memory, we simply find correct timestamps for the blocks that are indeed accessed and use an initial timestamp, say 1, for those blocks that were never accessed.) In this way, the recorder may introduce false conflicts for the blocks that were never accessed by a cache, but the recorder never misses a true conflict. As we will see in the next chapter, the extra false conflicts are most likely to be infrequent and reducible, which is due to (1) the false conflicts happens only when a directory entry is missing; (2) the timestamp computed for a never-accessed block is likely very “old”. The likely reducible false conflicts do not significantly increase the log size.

Similar to the case of writebacks and nonsilent replacements, we collocate the timestamps of evicted cache blocks with the directory. Therefore, the false conflicts do not incur additional messages to fetch the timestamps of the evicted cache blocks. In this way, detecting false conflicts incurs little coherence transaction latency, hence, low runtime overhead.

As for the snooping protocols, there is no directory, hence no additional timestamp memory is collocated with the directory. We can use a slightly different conflict detection method in snooping protocols, because these protocols usually do

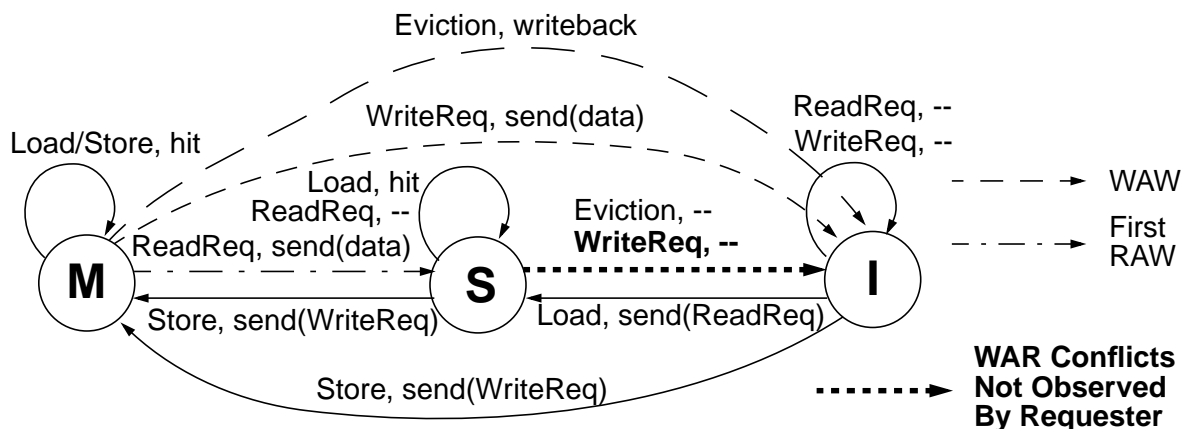


Figure 4.10 Missing conflicts due to the lack of acknowledgements. Snooping protocols often exploit the total order property of interconnection networks. This total order allows snooping protocols to eliminate explicit acknowledgements, which cause a requester to fail to observe the necessary WAR conflicts for race recording. One solution is to use *responder detection* instead of *requester detection*.

not have explicit acknowledgements for S→I state transitions. The next section presents this detection method in more detail, then answers the question on missing MSI states on cache evictions.

4.5.2 Implicit or Combined Response

Our idealized coherence protocol assumes explicit acknowledgements are sent when a cache block transits from the SHARED to the INVALID state (*send(ack)*). This allows both the requester and the responder to observe the WAR conflict indicated by the state transition. However, snooping protocols often use implicit or combined responses, rather than explicit acknowledgements. Implicit and combined responses exploit the total order property of interconnection networks (*e.g.*, buses). This total order property allows snooping protocols to eliminate explicit acknowledgements. Without explicit acknowledgement, a requester cannot observe necessary WAR conflict because it does not know which other cache has

had a SHARED copy of the cache block and has invalidated the block. Figure 4.10 shows this problem.

Many existing coherence protocols do not assume a total order interconnect, and they use explicit acknowledgements. For example, AMD Hammer protocol is a recent example of a broadcasting protocol that does not use a bus interconnect and relies rather on explicit acknowledgements [2, 71]. On CMPs, Kumar *et al.* show that on-chip Shared Bus Fabrics (SBFs) often do not achieve the best area/performance/power trade-off [29]. It remains to be seen whether SBFs will be used in future CMPs beyond eight cores.

It is possible to implement our race recorder with a bus based snooping system that does not have explicit acknowledgements. We can use responder detection instead of requester detection. Recall that both the responder and the requester of a cache block can observe cache block conflicts. Section 4.2.2.1 shows the difference between them is mainly the bandwidth efficiency and design complexity of request-forwarding. For snooping protocols, responder detection requires the requester always piggyback its IC when requesting a cache block. This piggybacked information allows the responder to record conflicts. However, piggybacking the IC on every request may consume precious bus cycles or it may require a widening of the bus width. Both of these options can be more expensive than piggybacking the response messages in directory protocols. We will leave optimizing responder detection in bus interconnects to future research.

Another reason to choose responder detection over requester detection in snooping protocols is to handle cache evictions. (Recall that we have delayed han-

dling this problem from Section 4.5.1.) When cache blocks are evicted in snooping protocols, there are no other places the MSI state information is stored. We again approximate the missing conflicts by conservatively assuming all blocks that are not present in a cache have been accessed some time in the past by the cache. This incurs false conflicts. These false conflicts incur extra response messages (hence extra memory latency) if we use requester detection. They do not incur any extra messages (hence no extra memory latency) in broadcasting protocols using responder detection. Overall, implementing our race recorder on a snooping system can also achieve low runtime overhead, but it can incur more bandwidth overhead than the recorder on a directory system.

4.5.3 OWNED and EXCLUSIVE States

Our idealized coherence protocol has only three stable states: M, S, and I. Many realistic protocols include the OWNED and/or EXCLUSIVE states for better performance. The OWNED state allows read-only access to the block (much like SHARED), but also signifies that the value in the main memory is incoherent or *stale*. The Exclusive state allows read-write access to the block (much like MODIFIED), but also signifies that the value in the main memory is coherent or up-to-date.

In Figure 4.4, we detect only the first RAW conflict in the MSI protocol at the M→S transition. The rest RAW conflicts can be detected by adding the OWNED state. If without the OWNED state, special bits can to be added in the cache or the directory to signify the a SHARED block is most recently in the MODIFIED state.

These special bits effectively record a cache block was in MODIFIED state after the first RAW conflict. In more detail, between any consecutive writes to a block, the block can be read zero or more times. In other words, there can be multiple RAW and WAR conflicts for every WAW conflict. In the MSI protocol, we detect all WAW and WAR conflicts, but only the first RAW conflict is detected on the M→S state transition, between each pair of consecutive writes to a block. In a MOSI protocol, the first RAW conflict is detected when the M→O state transition happens. The rest of RAW conflicts are detected when the owner observes the new ReadReq requests. The necessary timestamps are piggybacked on the data message that the owner sends to the requester. In the MOSI protocol, WAW conflicts are detected when O→I and M→I transitions are observed, instead of just M→I transitions in the MSI protocol.

Our detector does not require the EXCLUSIVE state. But if the E state is used, it will be treated like the M state. In particular, when a block transits from the I to E state, the recorder records both WAR and WAW conflicts by assuming a write will eventually happen to the E state. Similarly, when a block transits from the E to O state or from the E to I state, RAW and WAW conflicts are recorded as well.

4.5.4 Nonatomic Transactions

So far, we have only considered atomic transactions in the simple MSI protocol. Realistic coherence protocols can use nonblocking directory or split bus transactions. In our experiences in implementing the hardware recorder, nonatomic transactions complicate the recorder implementation, just like it complicates the

coherence protocol itself. However, we have not encountered any nonatomic transactions that prevent us from implementing our hardware race recorder. Two issues are generally important in the implementation: the stable state that a nonatomic transaction will eventually reach and the logic ordering (and reordering) of coherence events.

4.5.5 Multilevel Caches

Finally, our simple coherence protocol assumes a unified, single-level cache hierarchy. In realistic protocols, split instruction and data caches and two-level caches are often used. We have implemented our hardware race recorder on such realistic configurations with split I/D, writeback L1 caches and exclusive L1 and L2 caches [73]. Other realistic cache configurations include write-through L1 caches and nonexclusive L1/L2. We will leave supporting these configurations to future research.

4.6 Miscellaneous Hardware Implementation Issues

This section extends our hardware race recorder to handle the “implementation details” of speculative processors, out-of-order interconnect, and integer wrap-around of a realistic system. Note that this realistic system implements the TSO memory model. We use send and receive observations as lemmas for optimizations.

4.6.1 Send and Receive Observations

Send Observation. When a processor j at instruction count Cur_IC sends a coher-

ence response for block b , it may include any instruction count in the interval $[CTS[b], Cur_IC]$.

We should be familiar with the send observation, because it is a direct corollary of the correctness of the RTR algorithm. In the RTR algorithm, the recorder can introduce stricter but not overly-strict dependencies. These dependencies anchor their starting instructions (responder, sender) in the intervals allowed by the send observation.

Receive Observation. When a processor j at instruction count IC receives a coherence response for block b , it may associate the conflict with the instructions in the interval $[IC+1, XIC]$, where $(IC+1)$ is the next instruction the processor wishes to commit and XIC is the IC of the instruction that actually accesses the block.

The receive observation is symmetrical to the send observation. The receive observation introduces stricter but not overly-strict dependencies on the requester (receiver) side.

4.6.2 Out-of-Order Execution and Hardware Prefetching

In a speculative processor, nonblocking caches and out-of-order executions allow coherence requests for blocks that are *not* accessed by the next instruction to commit. In fact, a cache block may be speculatively accessed before its coherence message arrives. A processor implementing SC, however, appears to access blocks when instructions commit [23]; therefore, an early access will always appear to follow the message. As a result, the receive observation continues to hold (*i.e.*, if we associate each coherence response with $IC+1$, the actions in

Table 4-3 continue to operate correctly).

In the TSO memory model, we use the `ordering_buf` data structure to handle the speculative loads. Similar to the algorithm in Table 3-4, a speculative load can be put in the `ordering_buf`. If the load eventually commits but the load value is over-written by another thread, the load value is logged. If the load is squashed, it is treated like a load prefetch, where we use the receive observation.

4.6.3 Unordered Interconnect

Table 4-3 assumes at most one message is transmitted from processor i to j in any given time. With speculative processors, multiple messages from processor i can arrive at processor j out of order. One solution to the problem is to send full ICs (as opposed to changes of IC, as discussed in Section 4.7) in each message. This way, each message contains the full IC for the beginning of a conflict, we associate the ending of the conflict with the current committing instruction. Receive observation ensures the correctness. Nevertheless, message re-ordering can affect the effectiveness of the transitive reduction.

4.6.4 Integer Wrap-around

The hardware recording algorithm in Table 4-3 assumes unbounded integers, while practical hardware must use finite integers. A trivial solution is to use 64-bit integers since they will not overflow for decades. Using 64-bit integers is acceptable for IC, MTS and sliding window registers. However, 64-bit integers can increase the memory overhead of cache timestamps (CTS) significantly. In the next chapter, we systematically present techniques in reducing the memory

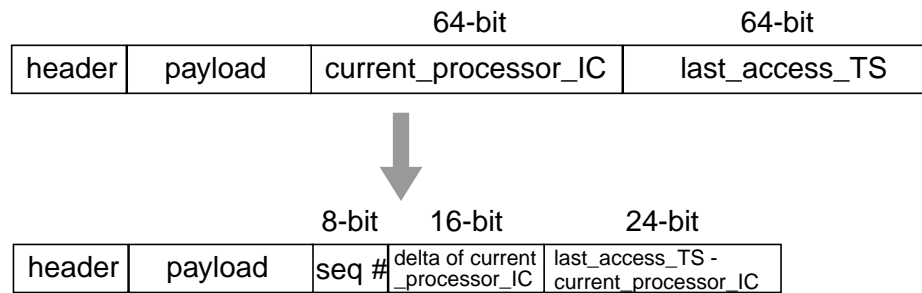


Figure 4.11 A compact format of the piggybacked IC and Timestamp. To lower the extra bandwidth consumption, we do not piggyback the full IC and timestamp in the coherence messages. Without our optimization, the full IC and timestamp are represented by two 64-bit integers, *i.e.*, **16 bytes** long. With our optimization, the IC and timestamp are represented by an 8-bit sequence number, a 16bit delta IC and a 24-bit timestamp-IC differential, *i.e.*, **6 bytes** long (62.5% reduction).

overheads for the cache timestamps. Our solution allows us to achieve the benefit of 64-bit integers, *i.e.*, no wrap-around, without actually storing the 64-bit integers.

4.7 Reducing the Timestamp Communication Overhead

Section 4.3.1 presents the Pairwise-TR technique that significantly reduces the communication overhead for race recording. However, even with pairwise-TR, significant network bandwidth is consumed by sending full 64-bit timestamps. We will now describe an optimization that helps reduce this communication overhead.

In the RTR algorithm (Table 4-3), we piggyback two 64-bit integers onto each coherence response message. Figure 4.11 shows how we break and compact the two 64-bit integers into three smaller integers: an 8-bit sequence number, a 16-bit delta IC (ΔIC) and a timestamp-IC 24-bit differential.

1. The sequence number reestablishes the ordering between out-of-order mes-

sages. The advantage is that we can send changes of IC to reduce the communication overhead. The disadvantage is that we need to buffer the changes of IC and reconstruct the full ICs before we can write the dependence log. With an 8-bit sequence number, we allow a total of 128 in-flight messages between each pair of processors, without worrying about sequence number wrap-around.

2. Δ IC represents the number of dynamic instructions that were committed since the last message was sent from each ordered pair of processors. With 16-bit Δ IC, we allow the consecutive messages between two processors to be at most 2^{16} dynamic memory instructions apart. If two processors do not communicate in either direction for more than 2^{16} instructions, we fall back to full IC in these rare events.
3. The 24-bit timestamp-IC differential allows us to represent the `last_access_TS` with sufficient precision. If a timestamp-IC differential overflows the 24-bit integer, we simply use 2^{24} as the differential. This approximation is allowed by the send observation.

In Chapter 7, we show that this compact representation of IC and timestamps reduces the recorder bandwidth overhead to no more than 10% more than the baseline system without the race recorder.

4.8 Open Problems

4.8.1 Recorder Virtualization

In this dissertation, we assume full-system recording, where each processor

core is treated as a single sequence of instruction streams. Process and thread virtualizations are transparent to the recorder.

In the future, if we need application level recording, the race recorder has to be virtualized, just as the processor resource is virtualized by the OS. Virtualization requires the race recorder to support thread context-switch and migration. This would require the recorder states to be saved and restored on context-switches. The recorder states include IC, MTS registers, sliding window registers, and timestamps. Naïvely saving and restoring these states may not be feasible due to the potentially high overhead. We believe there are optimizations to reduce the overhead. More importantly, virtualization requires the record to handle a variable number of threads, as opposed to a fixed number of processors. We have to change the MTS registers and sliding window register to be dynamically sized, possibly in the virtual memory space. This further increases the design complexity of a virtualizable recorder. We will leave these tasks to future research.

4.8.1.1 Virtualization and Coscheduling Virtual Machines

Nevertheless, our nonvirtualized recorder can be useful in coscheduling virtual machines, such as VMware [69]. A Virtual Machine (VM) is a software program that emulates a hardware system. In a multiprocessor VM, a fixed number of (virtual) processors are emulated and each of these (virtual) processors is emulated by a designated software thread. Coscheduling enables these threads to be scheduled simultaneously to avoid potential waste of processor cycles due to excessive spin-wait.

Our race recorder can support coscheduling virtual machines with minor changes. First, the MTS, sliding windows registers can still have fixed sizes, because coscheduling VMs have limited and a fixed number of virtual processor threads. Second, if the virtual processor threads are affiliated with a designated physical processor, the recorder states do not need to be saved and restored on context-switches. Instead, a way to enable and disable the recorder is necessary to ensure that the recorder is activated only for the virtual processor threads.

In summary, the hardware race recorder proposed in this dissertation is almost ready to be applied to enable deterministic replay for coscheduling virtual machines.

4.8.2 Multilevel Coherence

Another limitation of our hardware recorder is that it supports only single-level coherence protocols. In some scalable SMP systems (*e.g.* Sun Wildfire [24]) and recent CMP systems, multilevel coherence protocols are used to handle cache coherence problems in different domains in the memory hierarchy. For example, core-level and chip-level coherence both need to be guaranteed in a multichip CMP system. It is unclear if multilevel coherence protocols are going to be widely used in the future. Marty *et al.* recently studied the possibility of using a single-level protocol, which may significantly reduce the design complexity of future CMPs [39].

In this dissertation, we assume a single-chip CMP. Therefore, only one level (core-level) cache coherence needs to be considered. We will leave multilevel

coherence support to future research.

4.8.3 Simultaneous MultiThreading (SMT) and Shared L1 Caches

In recently years, *Simultaneous MultiThreading* (SMT) was proposed to allow a processor core to support multiple threads by splitting the execution resources via space or time [18]. Our recorder cannot be directly extended to SMT processors, because threads from the same processor core share the same L1 cache. Sharing L1 cache is different from sharing higher level (*i.e.*, L2) caches, where cache coherence is still needed at the lower level (*i.e.*, L1). In SMT processors, piggybacking conflict detection onto the coherence protocol is no longer sufficient, because threads from the same processor core may conflict without generating any coherence transactions (the threads share the same L1). A potential solution to this problem is to add more metadata in the L1 cache to differentiate the accesses from different threads. The metadata perhaps can include a writer ID and a bit map that represents the reader IDs, similar to the metadata kept in a coherence directory. With the help from the metadata, it is possible to detect conflicts in the shared L1 cache. The hardware has to check for potential conflicts even on cache hits, rather than just on cache misses. The extra checks may consume more power, but should not incur significant performance overhead. We leave exploring this direction of supporting SMT processors to future work.

Chapter 5

Reduce the Memory Overhead: Timestamp Approximation

In the previous chapter, we presented a hardware race recorder that piggybacks itself onto the cache coherence hardware. Hardware assistance can significantly reduce the runtime overhead of race recording (see results in Chapter 7) by avoiding the cost of software tracing for every memory instruction.

However, hardware assistance does not reduce the memory overhead of our recording algorithm. In particular, our RTR algorithm has the memory overhead that is proportional to the total memory footprint of a program. In the previous chapter, we assumed sufficiently large timestamp storage is available to the hardware recorder. We also assumed the timestamp memory is collocated with the data blocks in the caches. These assumptions introduce two important problems of our race recorder. First, large timestamp memory is both costly in chip area and slow in access time. Second, collocating timestamp memory with processor cache introduces higher design complexity that can generate design push-back from the hardware designers [49].

In this chapter, we solve these problems with novel techniques in timestamp approximation. These approximation techniques significantly reduce the memory overhead of the RTR algorithm, while still retaining its small log size and low runtime overhead. The key idea is that only a small number of timestamps need to be stored precisely, and the vast majority of timestamps only help the recorder detect conflicts that are reducible and therefore not logged. For those timestamps, approximated values are sufficient. Compared to a straightforward timestamp memory design, our new timestamp memory has a small size (12 KB per processor) and it does not grow with the program memory size or the cache size. As a result of the reduced size, the timestamp memory can be separated from the caches and accessed quickly. We call our new timestamp memory *decoupled timestamp memory*, which has lower design complexity than the collocating timestamp memory.

We begin this chapter by reviewing a straightforward design of timestamp memory: the coupled timestamp memory (Section 5.1). We then propose decoupled timestamp memory in Section 5.2. The decoupled timestamp memory gives us more flexibility and results in lower design complexity. In Section 5.3, we describe four timestamp approximation techniques that can significantly reduce the memory overhead of race recording. We close this chapter with a performance optimization of the timestamp memory for a subset of the directory protocols (Section 5.4).

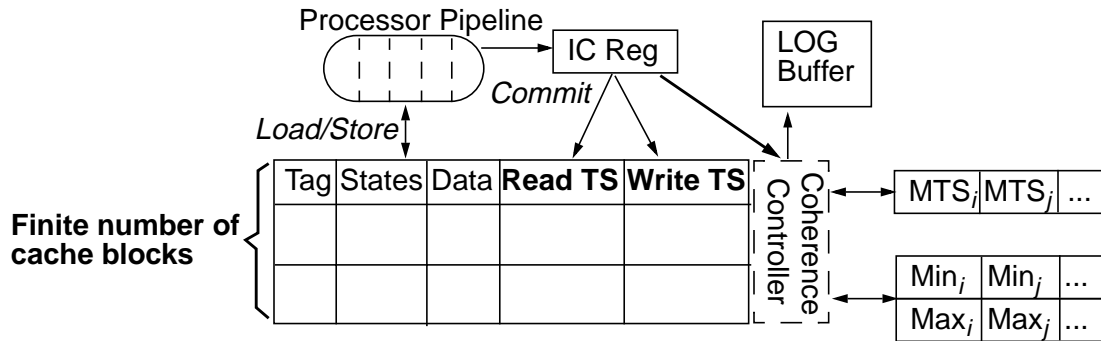


Figure 5.1 The coupled timestamp memory. We design the coupled timestamp memory so that the timestamps are collocated with the cache blocks. Due to the finite size of the cache, hence the timestamp memory, the coherence controller uses the current value of the IC register to approximate the timestamps of the missing blocks.

5.1 Coupled Timestamp Memory

In Chapter 4, we temporarily assume each processor could store and access a sufficiently large timestamp memory. This enabled our recorder to find the last read or write IC for any given memory block. Unfortunately, implementing the timestamp memory in this fashion can incur significant memory overhead, because the size of the timestamp memory will need to be proportional to the total memory footprint of a program.

As a baseline design, we describe *coupled timestamp memory*, whose size is determined by the cache size of the processor. In this design, the timestamp memory and the cache share the same tag array, *i.e.*, the size and associativity of the timestamp memory is coupled with the cache. As shown in Figure 5.1, timestamps are collocated with the cache blocks themselves. With 64-byte cache blocks and one 64-bit timestamp, the coupled timestamp memory incurs 12.5% chip area overhead. (Assuming only one timestamp to represent the last access IC.)

With finite timestamp memories, we must be able to handle the conflicts on

blocks whose timestamps are “missing” from the timestamp memory (due to evictions). We solve this problem by using the send observation, which allows us to approximate a block b 's timestamp using any IC from the interval $[CTS[b], Cur_IC]$, where $CTS[b]$ is the precise timestamp of b and Cur_IC is the current value of the processor's IC register. For the coupled timestamp memory, we use IC to approximate the timestamp of b when $CTS[b]$ is not available. Using IC is a conservative, but a simple, approximation.

5.2 Decoupled Timestamp Memory

In modern microprocessors, caches are critical components. Coupling the timestamp memories with caches increases the design complexity, which may generate push-back from designers [49].

In fact, timestamp memories are simpler than the caches, because (1) they do not need to be heavily multi-ported, as timestamps are written only when memory instructions commit; (2) they do not need to keep any extra state, such as valid and dirty states (The valid state is not needed and timestamps are initiated to zero.); (3) they do not need to support coherence invalidations; and (4) they do not need to implement the same replacement policy as the caches (Section 5.3). For all these reasons, we propose a decoupled timestamp memory that is simpler to design than the coupled timestamp memory.

Figure 5.2 shows the high-level design of the decoupled timestamp memory, which is organized like a set-associative cache. The timestamp memory is indexed by memory block's physical addresses. The write and read ports are con-

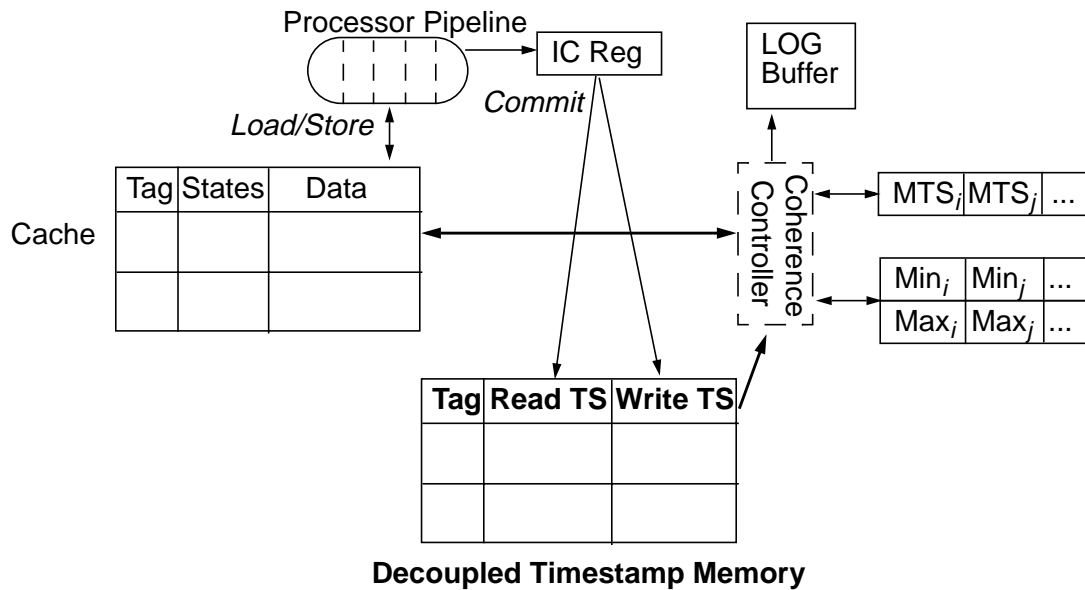


Figure 5.2 The decoupled timestamp memory. Compared to the coupled timestamp memory, the decoupled timestamp memory has an independent tag array. The timestamp memory is accessed by the processor only when memory instructions commit. The coherence controller uses the LRU timestamps from the timestamp memory to approximate the missing timestamps.

nected to the IC register and the coherence controller, respectively. When the processor commits a memory instruction, it writes the IC of the instruction to the timestamp memory. When coherence requests and responses arrive at the cache coherence controller, the controller looks up the corresponding timestamp from the timestamp memory, then either sends the timestamp to the requester or possibly writes the log, depending on decisions made by the hardware RTR algorithm (Table 4-3). Compared to the coupled timestamp memory, the decoupled timestamp memory has an independent tag array.

In addition to the simpler design, the decoupled timestamp memory gives us the flexibility to vary its size and associativity without changing the cache. As we show in Chapter 7, timestamp memory can be sized much smaller than the memory caches, which dramatically reduces the hardware cost when the two are

decoupled.

Instead of using the IC register when a timestamp is missing from the timestamp memory, we use a new timestamp approximation technique in the decoupled timestamp memory. This new technique is called Set/LRU timestamp approximation. Set/LRU is one of the key techniques that helps us reduce the memory overhead of the timestamp memory. We discuss Set/LRU in more detail in Section 5.3.1.

5.3 Reducing Memory Overhead

This section presents four techniques that help reduce the space (chip area) overhead of the timestamp memory. All of the techniques are heuristics in approximating the precise last read and last write timestamps of a memory block. As stated in the send observation, it is correct to use any timestamp in the interval $[CTS[b], Cur_IC]$ in case $CTS[b]$ of block b is not available.

Our key observation is that by the nature of the transitive reduction, the older (smaller) a timestamp, the more likely the conflict that sourced from the timestamp is reducible. We exploit this property by storing the older timestamps with less bits, so long as their approximated values reveal that they are old enough to be reducible. Therefore, we can significantly reduce the memory overhead of race recording without adversely impacting the log size.

5.3.1 Set/LRU Timestamp Approximations

Because our timestamp memory is organized in set-associative fashion, it is natural to use the timestamps stored in the same associative set to approximate

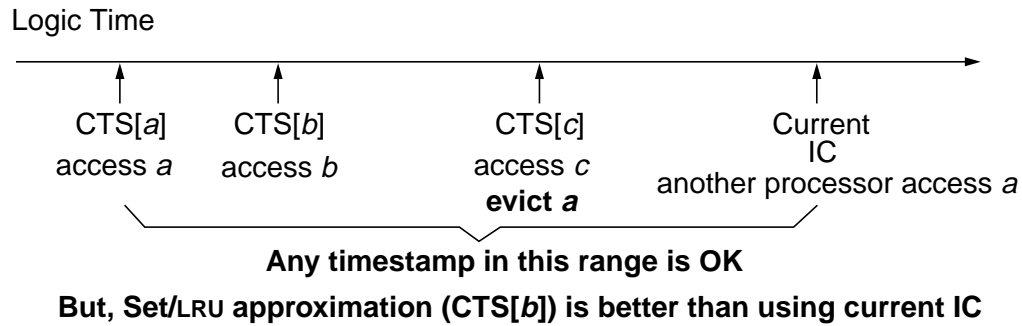


Figure 5.3 The Set/LRU approximation. The logical time flows from left to right. The timestamp memory has an associativity of 2 and is initially empty. At timestamp CTS[a], block *a* is accessed by a processor, then CTS[a] is stored in the processor's timestamp memory. If two other blocks *b* and *c* map to the same set as block *a*, accessing *b* and *c* will evict block *a*'s timestamp from the timestamp memory. Any timestamp approximation from the interval [CTS[a], IC] is acceptable, but CTS[b] is closer than the current IC to the precise timestamp (CTS[a]).

the missing timestamps. We call this **Set/LRU timestamp approximation**, in short **Set/LRU**. As its name implies, **Set/LRU** employs the **Least Recently Used (LRU)** replacement policy in the timestamp memories. When an approximated timestamp of a missing block is needed, **Set/LRU** chooses the timestamp of the LRU block in the same associative set where the missing block maps. Like the approximation technique used in the coupled timestamp memory, **Set/LRU** is guaranteed to use an approximated timestamp in the range [CTS[b], Cur_IC].

Figure 5.3 shows an example of **Set/LRU**. Let's assume we have a two-way set-associative timestamp memory with LRU replacement policy. If three memory blocks, *a*, *b*, and *c*, are mapped to the same set and are accessed by a processor in this order, block *a*'s timestamp that was once stored in the timestamp memory will be evicted when the processor accesses block *c*. Later, when another processor requests block *a*, the responding processor needs to approximate block *a*'s timestamp. In the coupled timestamp memory, we use the current IC as the approxi-

mated timestamp. However, because current IC is the youngest (largest) IC from the responding processor to the request processor, this approximation always introduces an irreducible conflict. With Set/LRU, the responding processor approximates a 's timestamp with b 's timestamp, because it is guaranteed that block b was last accessed after block a was last accessed. This approximated timestamp is closer to the precise timestamp, which makes the resulting conflict more likely to be reducible.

The example above assumes only one timestamp is stored per block. When both last read and last write timestamps are stored in the timestamp memory (see Section 5.3.3), Set/LRU approximates a missing timestamp (read or write) with the maximum of read and write timestamps of the LRU block.

In a coupled timestamp memory, the timestamp of a LRU block is not always older than the timestamp of a block missing from the memory, because coherence invalidations may invalidate a cache block before it is replaced. However, in our decoupled timestamp memory, timestamps are not invalidated. Therefore, the timestamp of a LRU block is always older than the timestamp of a block missing from the memory.

The effectiveness of Set/LRU depends on both the size and the associativity of the timestamp memory. We will explore the design space of the size and the associativity of decoupled timestamp memory in Chapter 7.

5.3.1.1 Timestamp Approximation Without LRU

Realistic hardware sometimes implements pseudo-LRU or other replacement

policies. In these cases, using the timestamp of the Most Recently Used (MRU) cache block to approximate a missing timestamp is correct, although less precise.

Additionally, if the hardware budget permits, a modification can make Set/LRU work with arbitrary replacement policies. We need to add an extra timestamp register per associative set of the timestamp memory. The extra timestamp register records the maximum timestamp that was ever evicted from a set, or invalidated in a set. In this way, we can approximate the missing timestamps with these extra timestamp registers.

5.3.2 Partial Timestamps

Another technique to reduce the hardware cost of the timestamp memory is to store only the least significant bits of the timestamps in the memory. For example, instead of storing timestamp 0x1234 in the timestamp memory, we may store only 0x34. When the partial timestamp is read back, it is concatenated with the most significant bits of the current IC. If the current IC is 0x4321, the concatenation produces a timestamp of 0x4334. Since 0x4334 is larger than the current IC, we use 0x4234 to approximate the precise timestamp (0x1234), because 0x4234 is the largest possible IC that would have produced the partial timestamp of 0x34. If the concatenation result is smaller than the current IC, we simply use it to approximate the precise timestamp. This approximation is again in the range $[CTS[b], Cur_IC]$. We will explore the design space of the width of the partial timestamps in Chapter 7.

5.3.3 Storing Both the Read and the Write Timestamps

In Chapter 4, we assumed only one `last_access_TS` is stored in the timestamp memory. However, this is only an approximation. The single `last_access_TS` approximates the precise last read and last write timestamps with the maximum from the two. Saving only one timestamp per block halves the memory overhead, but increases the possibility that a reducible conflict will be recorded unnecessarily. For instance, if a processor first writes a block, then reads the same block long after the write, future reads from other processors will be misdetected as conflicting with the last read from the first processor. When similar access patterns frequently happen, it is better to store both the last read and last write timestamps.

In Chapter 7, we will present experimental results that can direct us in choosing when to use both timestamps and when to use a single `last_access_TS` for each block.

5.3.4 Partial Tags

Finally, similar to partial timestamps, we can store the LSBs of the tags in the timestamp memory to reduce the memory overhead, especially for highly associative timestamp memories. Partial tags introduce an aliasing problem in both reading and writing the timestamp memory. The problem is essentially similar to the false sharing problem in the memory caches. As we have discussed in Section 4.2.2.2, false sharing does not affect the correctness of race recording. Therefore, using partial tags can reduce the memory overhead without affecting

the correctness of race recording.

Partial tags may or may not significantly impact the recording efficiency. The impact depends on if partial tags cause us to overly approximate a block's precise timestamp. We do not further evaluate the effect of partial tags, because we can already reduce the memory overhead of the timestamp memory significantly with the previous three approximation techniques.

5.3.5 Timestamp Approximation and Software Recorders

The idea of timestamp approximation is not limited to the hardware race recorders. If implemented, the idea can help software race recorders to lower their memory overhead. For example, Kim *et al.* proposed a data structure for implementing stack simulation of highly-associative caches [26]. Similar data structure could be a good fit for implementing software timestamp memory. In particular, the dual data structure consists of a hash-table and a double-linked list to simulate each cache set. For the software timestamp memory, the hash table can provide fast lookup of the timestamps given the block address. The double-linked list can provide LRU order to determine the approximated timestamp when a block is not stored in the timestamp memory. Because of the small size of the timestamp memory (Chapter 7), both the hash table and linked list can be both small and fast.

5.4 Two-level Timestamp Memory

In directory protocols with notifying (rather than silent) cache evictions, it is important to include timestamp memories at both the processors and the direc-

tory. Earlier in this chapter, we assume that there is a single timestamp memory for each processor. When a processor stores to a cache block, coherence messages are needed to obtain the timestamps from those processors that have read or written the block, regardless of whether or not the processor have evicted the block. In a directory protocol with silent eviction, these messages are needed for invalidation anyway. In a directory protocol with notifying cache evictions, these messages can incur extra latency and bandwidth overheads.

To avoid these overheads, we add a timestamp memory for each processor at the directory. The second-level timestamp memory is updated when a block is evicted from a processor's cache. The timestamps themselves are piggybacked onto the directory-notifying or write-back messages.

Chapter 6

Evaluation Methodology

From chapters 3 to 5, we described the techniques for implementing effective and inexpensive hardware race recorders on different flavors of multiprocessor systems. In this chapter, we describe a specific design of a hardware race recorder—*determinizer/CMP*. Determinizer/CMP is different from existing race recorders in four ways: (1) determinizer/CMP implements the RTR recording algorithm; (2) determinizer/CMP uses the Set/LRU timestamp approximation; (3) determinizer/CMP uses decoupled timestamp memories; (4) determinizer/CMP is based on a Chip MultiProcessor (CMP) system.

We begin this chapter by describing the baseline CMP system in Section 6.1. We then describe determinizer/CMP in Section 6.1. Finally, we describe our simulation infrastructure and workloads.

6.1 Baseline CMP System

We implement determinizer/CMP on a single-chip CMP system. Single-chip CMP systems, such as Sun UltraSPARC T1 [27], do not support memory coher-

ence between multiple CMP chips. The complexity of the cache coherence protocols for single-chip CMP systems is more manageable than that of the multichip CMP systems [39]. For determinizer/CMP, a single-chip CMP baseline allows simple design and easy deployment because no off-chip change is required.

To keep the on-chip caches coherent, our baseline CMP system keeps track of sharer and owner information of cache blocks in private L1 caches using a directory located at the shared on-chip L2 cache (similar to Piranha[8]). The L2 cache keeps shadow tags of all cache blocks in the L1 caches. The MOSI coherence protocol implements notifying cache replacement, *i.e.*, the directory at the L2 keeps a precise list of sharers for each L1-cached block. The L2 cache enforces neither inclusion nor exclusion.

We choose a directory protocol with point-to-point on-chip interconnection, because a recent study shows on-chip shared bus fabrics often do not achieve the best area/performance/power trade-off [29]. As we discussed in Chapter 4, implementing our race recorder with a bus-based snooping system is possible, but it will incur more bandwidth overhead.

6.2 Determinizer: a CMP-based Race Recorder

Figure 6.1 shows the high level structures of determinizer/CMP, which consists of a 4-way CMP and additional hardware for the race recorder. We augment each processor core with an instruction count (IC) register. The value of the IC register is used to update the timestamp memory (TSM) when a dynamic instruction commits. The L1 cache coherence controller uses the timestamps stored in the TSM

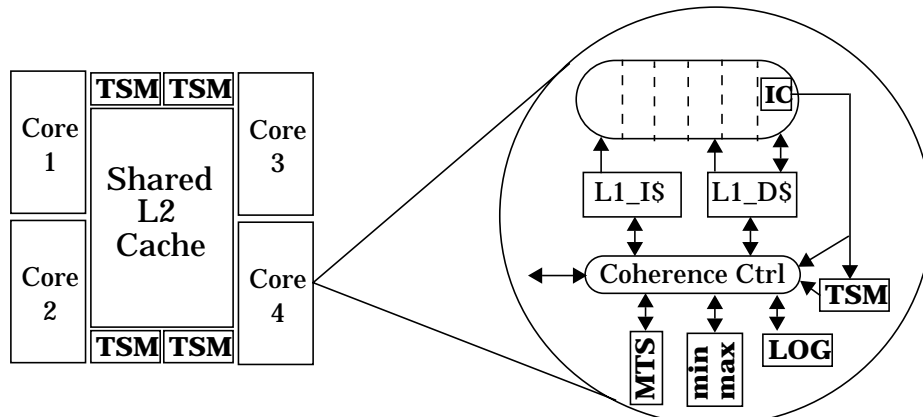


Figure 6.1 Determinizer/CMP. This figure shows a 4-way CMP system with the additional hardware for the race recorder. We augment each processor pipeline with an instruction count register (IC). The value of the IC register is used to update the timestamp memory (TSM) when a dynamic instruction commits. The L1 cache coherence controller uses the timestamps stored in the TSM in detecting and logging dependencies. The recorder implements the RTR recording algorithm with the maximal timestamp registers (MTS) and timestamp window registers (min/max). The RTR algorithm reduces the log size by transitivity and vectorization. Irreducible dependencies are stored in the log buffer (LOG). The TSMs at the L2 are used to avoid extra coherence messages. When a cache block is evicted from the L1 caches or written back to L2, the block's timestamp is sent to the L2 and stored in the L2 TSM. Future conflicts on the block will obtain necessary timestamps directly from the L2 TSM.

in detecting and logging dependencies. The recorder implements the RTR recording algorithm with the maximal timestamp registers (MTS) and timestamp window registers (min/max). The RTR algorithm reduces the log size by transitivity and vectorization. Irreducible dependencies are stored in the log buffer (LOG). The TSMs at the L2 are used to avoid extra coherence messages. When a cache block is evicted from the L1 caches or written back to L2, the block's timestamp is sent to the L2 and stored in the L2 TSM. Future conflicts on the block will obtain necessary timestamps directly from the L2 TSM.

6.3 Simulation Methodology

We use Wisconsin GEMS [38] full system simulation infrastructure to evalu-

Table 6-1 Simulation Parameters

CORES	Four 2 GHz, 1-issue, in-order superscalar cores.
PRIVATE L1 CACHES	Split I & D, each 64 KB 4-way set associative with LRU replacement, 64-byte lines, 1-cycle.
SHARED L2 CACHE	Unified 16 MB, 4-way set associative with LRU replacement, 64-byte lines, 15-cycle hit latency.
MEMORY	4 GB of DRAM, 80ns off-chip access time.
TIMESTAMP MEMORY	Decoupled, parameters vary in the experiments in the next chapter. The L1 and L2 timestamp memories have the same parameters.

ate determinizer/CMP. GEMS obtains the functional behavior of a multiprocessor Sun SPARC V9 platform architecture (used for Sun E6000s) using Virtutech Simics [37]. Simics mimics a SPARC system in sufficient detail to boot an unmodified Solaris 9 operating system.

We obtain the performance estimation of determinizer/CMP by simulating the memory system in detail. For the processor cores, we model a simple in-order 1-way-issue 2 GHz processor with 1 GHz system clock. For the coherent memory system, we model a MOSI directory protocol, as described earlier, with and without determinizer/CMP. The simulator captures all state transitions (including transient states) of the coherence protocol in the cache and memory controllers. We model a hierarchical switch-based interconnection as well as the contention within this interconnection, including contention with and without the recorder's message overhead. Despite SPARC V9 architecture specifying *Total Store Order* (TSO) as the default memory model, our simulator implements *Sequential Consistency* as a correct implementation of TSO. Table 6-1 summarizes the system configuration we simulate.

We exercise determinizer/CMP with four commercial workloads summarized in

Table 6-2 Commercial Workloads

<p>Apache is a static web serving workload. We use Apache 2.0.43, configured to use pthread locks and minimal logging as the web server. We use SURGE [6] to generate web requests. We use a repository of 20,000 files (totalling ~500 MB). We simulate 3200 clients, each with 25 ms think time between requests, and warm up for ~2 million requests before taking measurements for 600 requests for the performance simulations and 100 requests for the sensitivity studies.</p>
<p>Online Transaction Processing (OLTP) models database activities of a wholesale supplier, with many concurrent users performing transactions. Our setup uses TPC-C v3.0 benchmark and IBM's DB2 v7.2 EEE database management system. We use a 5 GB database with 25,000 warehouses stored on eight raw disks and an additional dedicated database log disk. We reduced the number of districts per warehouse, items per warehouse, and customers per district to allow more concurrency provided by a larger number of warehouses. We simulate 128 users, and warm up the database for 100,000 transactions before taking measurements for 200 transactions for the performance simulations and 40 transactions for the sensitivity studies.</p>
<p>SPECjbb is a server-side java benchmark that models a 3-tier system, focusing on the middleware server business logic. We use SUN HotSpot 1.4.0 Server JVM. Our experiments use 1.5 threads and 1.5 warehouses per processor (6 for 4 processors), a data size of ~44 MB, a warm-up interval of 200,000 transactions and a measurement interval of 10,000 transactions for the performance simulations and 5000 transactions for the sensitivity studies.</p>
<p>Zeus is another static web serving workload driven by SURGE. Zeus uses an event-driving server model. Each processor of the system is bound by a Zeus process, which is waiting for web serving event (<i>e.g.</i>, open socket, read file, send file, close socket, <i>etc.</i>). The rest of the configuration is the same as Apache's. We take measurements for 800 requests for the performance simulations and 200 requests for the sensitivity studies.</p>

Table 6-2. We counter the workload variabilities using a pseudo-random perturbation method [3]. For performance simulations, we report the mean results and 95% confidence intervals from 20 randomized runs. We report the average log growth rate of $\text{determinizer}/\text{CMP}$ in MegaBytes/core/second for each processor core. This number times the total number of cores indicates the total off-chip log bandwidth required by the recorder. We compress the log with the LZ77 [77] algorithm, which removes a small amount of packing inefficiencies in the log.

Chapter 7

Evaluation Results

In this chapter, we present a quantitative study on `determinizer/CMP`—our new hardware race recorder. In Section 7.1, we explore a design space of the race recorder by varying its design parameters. Then, in Section 7.2, we choose a representative design point and zoom in on the recorder’s hardware cost, log size, runtime overhead and bandwidth overhead. In Section 7.3, we conduct two experiments to show the benefits of the new RTR recording algorithm and the new Set/LRU timestamp approximation method. Finally, in Section 7.4, we present the scalability of the recorder by showing how the recorder log size scales with respect to the number of processor cores.

7.1 A Design Space of `Determinizer/CMP`

In this section, we vary several design parameters of `determinizer/CMP`.

1. **Algorithm** used by the recorder (TR or RTR);
2. **Timestamp Approximation Method** (Set/LRU or CurrentIC)
3. **Size** of the timestamp memories (2 KB to 2048 KB);

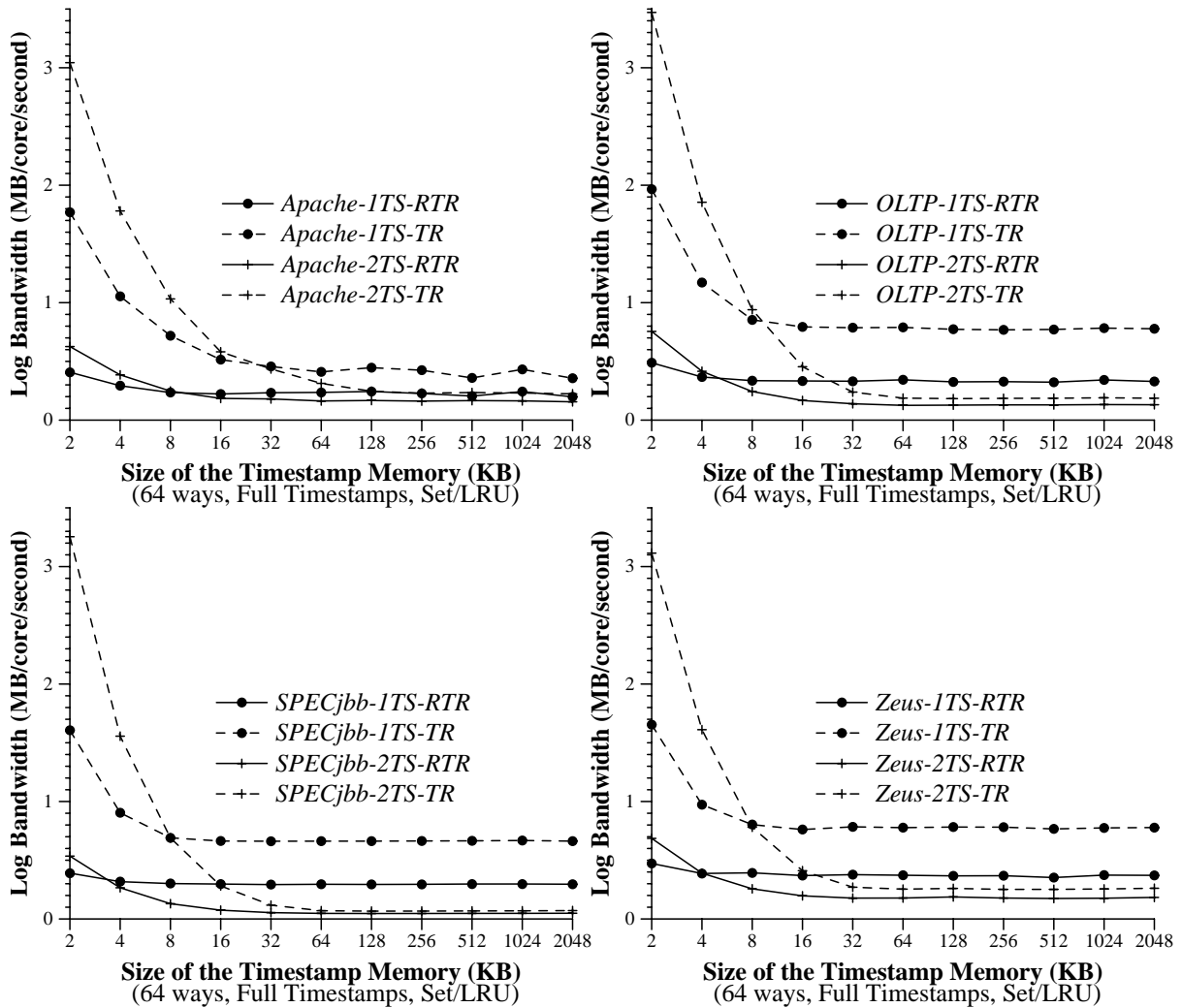


Figure 7.1 Impacts of the TSM size and the number of timestamps per block. We vary the number of timestamps per block from one (1TS) to two (2TS) and vary the size of the timestamp memory from 2 KB to 4096 KB. The raw data of the simulation results are shown in Table A.1 – Table A.4.

4. **Width** of partial timestamps (8-bit to 29-bit);
5. **Number** of timestamps per block (one or two timestamps per block);
6. **Associativity** of the timestamp memories (2-way to 1024-way).

We perform several sensitivity studies to determine the effects of these design parameters. In Figure 7.1, we vary the size and number of timestamps per block for the timestamp memories, while all other parameters are fixed. Both RTR and

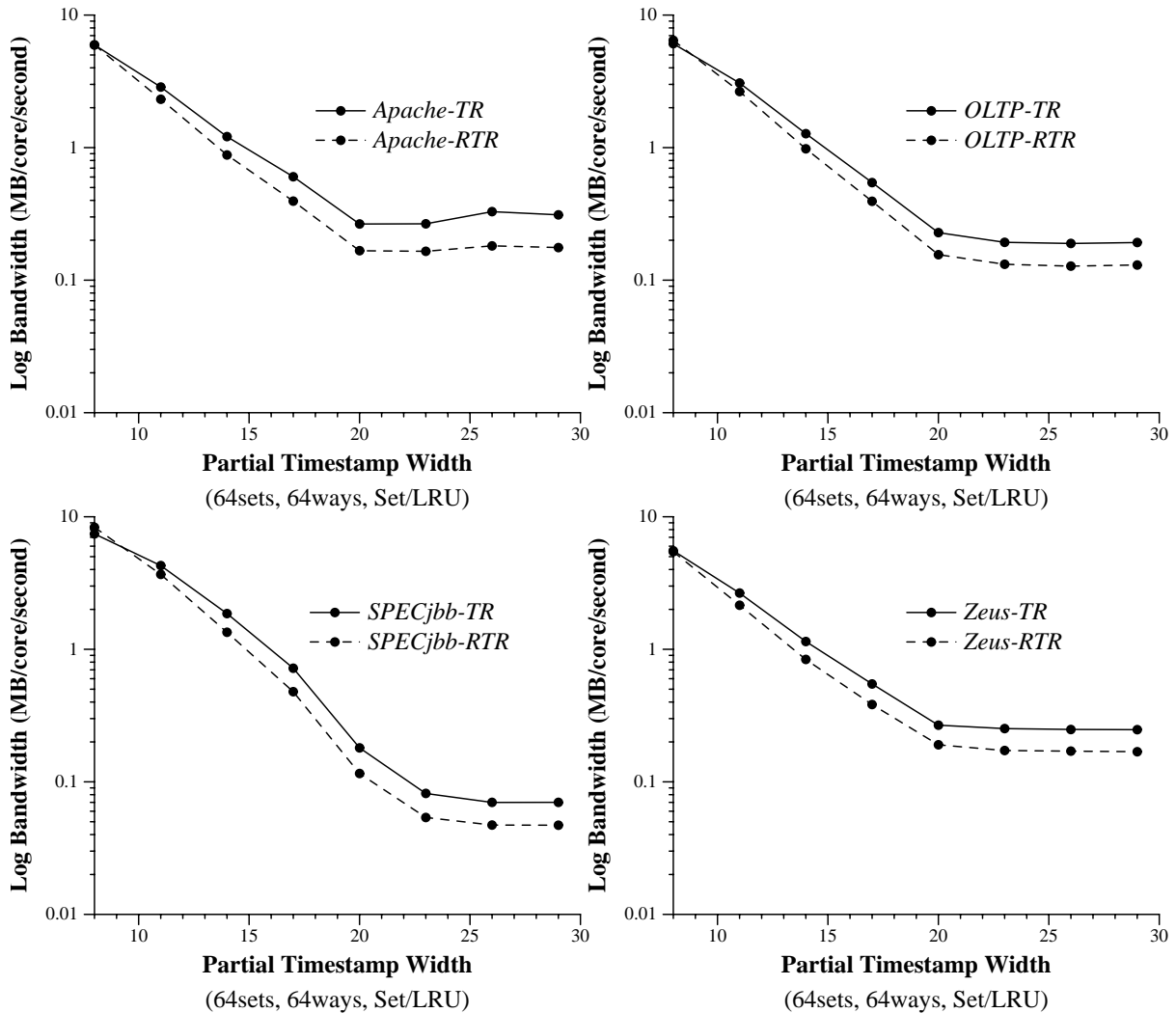


Figure 7.2 Impacts of the width of partial timestamps. We vary the partial timestamp width from 8-bit to 29-bit, with a 3-bit increment in each step. The raw data of the simulation results are shown in Table A.5 – Table A.8.

TR algorithms exhibit similar trends: (1) only small timestamp memories (no more than 64 KB) are needed to achieve small log size; (2) two timestamps per memory block (for both the last read and the last write) should be used after the timestamp memory size is sufficiently large to make the Set/LRU approximation effective.

In Figure 7.2, we vary the number of bits of partial timestamps. Although the

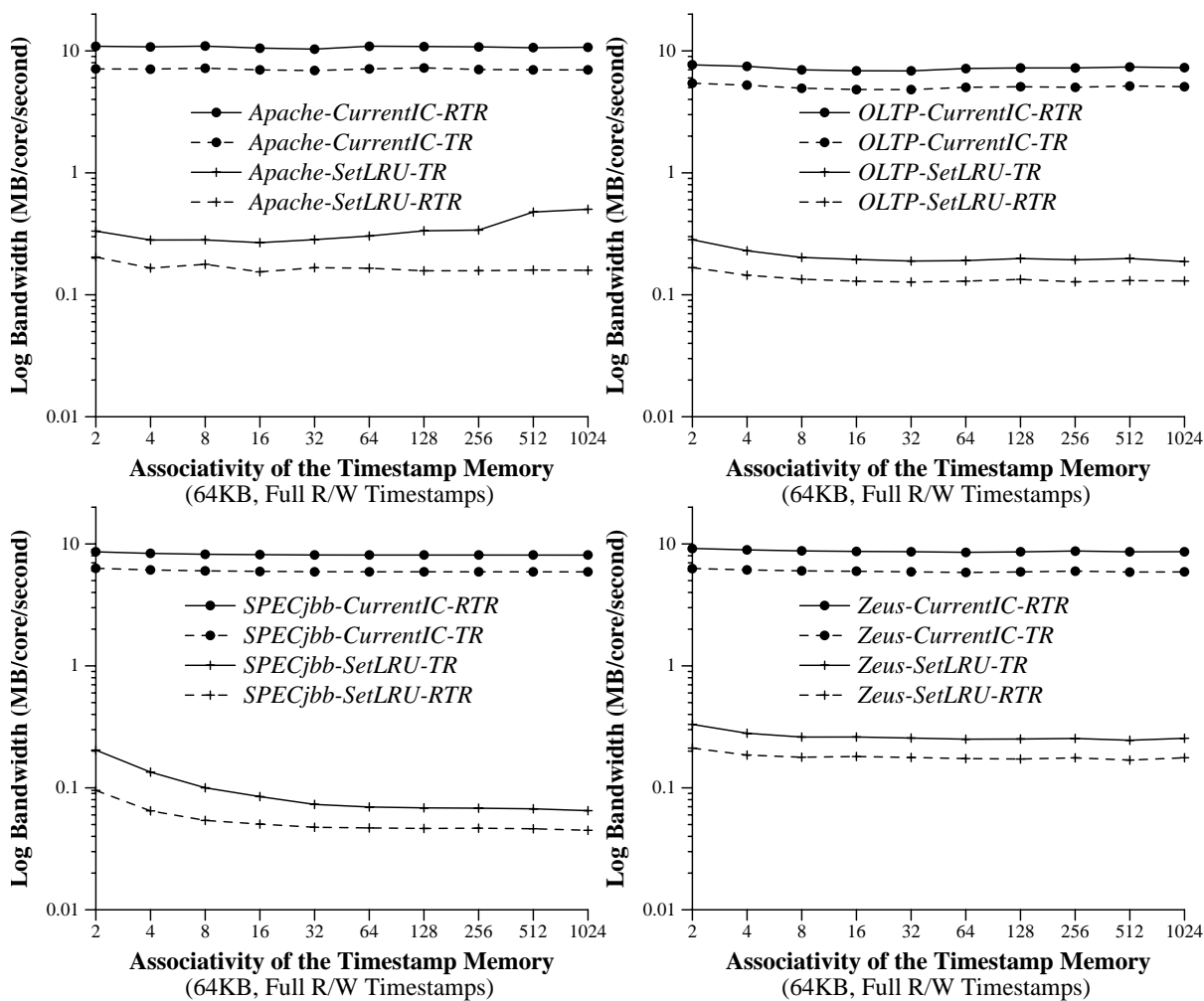


Figure 7.3 Impacts of the TSM associativity and the approximation method. We vary the timestamp memory associativity from 2-way to 1024-way and vary the timestamp approximation between Set/LRU and CurrentIC. The raw data of the simulation results are shown in Table A.9 – Table A.13.

full width of a timestamp is 64-bit, it is enough to store only the least significant 24-bit.

In Figure 7.3, we vary the associativity of the timestamp memories with either the Set/LRU or the CurrentIC timestamp approximation methods. The Set/LRU approximation method significantly outperforms the CurrentIC approximation method. Unlike the Set/LRU method, the CurrentIC approximation method does not benefit from higher associativity of the timestamp memories. In

Table 7-1 Hardware Cost of the Selected Configuration

RECORDER PARAMETERS	DETERMINIZER/CMP
Algorithm (RTR or TR)	RTR
Set/LRU	Yes
Number of Timestamps per Block	2
Partial Timestamp Bits	24-bit
Timestamp Memory Associativity	64-way
Timestamp Memory Size/core	24 KB ^a

a. 12 KB * 2 for both L1 and L2.

the next section we choose 64-way associative timestamp memories because higher associativity has diminishing returns.

From these results, we observe a trade-off between the log size and the cost (size and associativity) of the timestamp memories. This trade-off is useful as recorders may be optimized differently according to different design goals.

7.2 Performance of Determinizer/CMP

Now, we select a specific configuration of determinizer/CMP and report its performance characteristics.

7.2.1 Hardware Cost

Table 7-1 summarizes the design parameters of this selected configuration. Because the baseline CMP system implements a notifying-eviction directory protocol, determinizer/CMP uses the two-level timestamp memory, which uses two (small) timestamp memories per core. Each timestamp memory has 32 sets and 64 ways. With two (read and write) timestamps per block and 24-bit partial timestamps, the total size of the two timestamp memories (without the address tags) is 24 KB per core. Each timestamp memory is 12 KB, which should be fairly

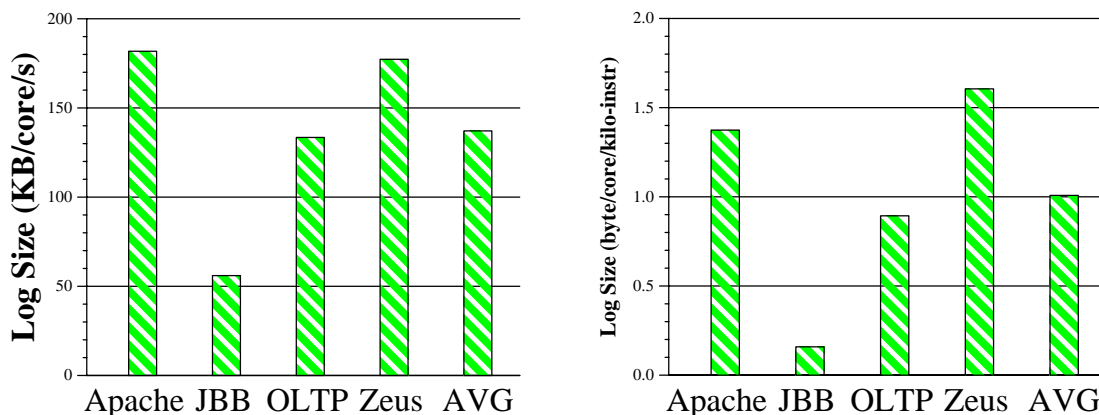


Figure 7.4 Log size of the selected configuration. In the left figure, we show that the recorder generates 50 KB–180 KB log per core per second for commercial server workloads. In the right figure, we show that the recorder generates 0.2–1.6 byte log per core per kilo-instruction. The raw data of the simulation results are shown in Table A.13.

easy to accommodate by the die area budget of current CMP chips. Further reducing the hardware cost is possible at the cost of a larger log size. As we show in Figure 7.1, reducing the timestamp memories to 2 KB increases the log bandwidth to no more than 1 MB/core/s. Even this log bandwidth is still well within the bandwidth capability of the baseline CMP system. Large log size, however, can shorten the length of the recording.

7.2.2 Log Size

Figure 7.4 shows the log size of determinizer/CMP for each commercial workload. The log size ranges from 50 KB/core/s to 180 KB/core/second for the 4-way CMP. On average, the recorder generates about 140 KB/core/second. With a 1 MB log buffer allocated in the main memory, the recorder can record about 2 seconds of the executions. Much longer recording is possible with large disk storage.

In terms of instructions, the log grows about one byte per kilo-instruction on

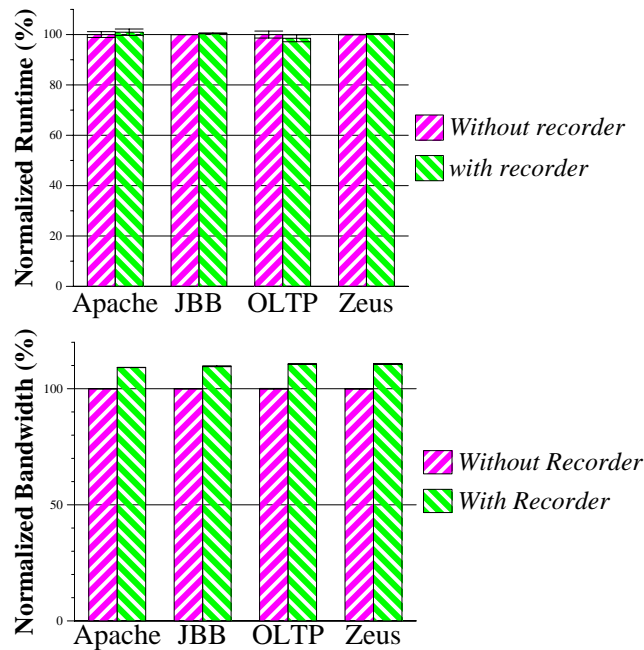


Figure 7.5 Runtime and bandwidth overheads of the selected configuration. The recorder incurs no more than 2% runtime overhead and around 10% network bandwidth overhead. The raw data of the simulation results are shown in Table A.14.

average. This means that for processors with high instruction per cycle (IPC), the log growth rate can be higher, but is still within an acceptable range.

7.2.3 Runtime and Bandwidth Overheads

Determinizer/CMP incurs extremely low runtime overhead and reasonable network bandwidth overhead. Figure 7.5 shows the runtime and bandwidth consumption of each workload, with and without the recorder. The runtime overhead is no more than 2%. The bandwidth overhead is around 10%. These low overheads are the results of two optimizations: (1) the two-level timestamp memory avoids the overheads caused by the extra messages; (2) sending differences (deltas) of timestamps rather than the full timestamps (Section 4.7) reduces the

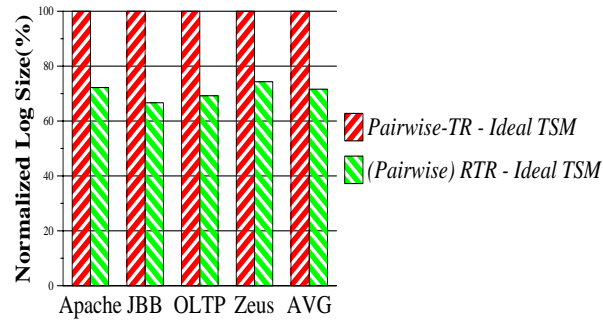


Figure 7.6 RTR versus TR. The log size of the RTR algorithm is around 30% smaller than that of the TR algorithm. The raw data of the simulation results are shown in Table A.15.

bandwidth overhead.

7.3 Benefits of RTR and Set/LRU

The small log size of determinizer/CMP comes from both the new RTR recording algorithm and the Set/LRU timestamp approximation method. We now compare RTR with Netzer’s TR algorithm assuming infinite timestamp memory is available. Then we show the effectiveness of the Set/LRU approximation by comparing it with infinite timestamp memory.

RTR versus TR. To isolate the effects of the RTR algorithm from the Set/LRU approximation, we give the recorder infinite timestamp memories, *i.e.*, no timestamp approximation is used and two timestamps are remembered per block. Figure 7.6 shows the log size reduction from changing the recording algorithm from TR to RTR. For all workloads, RTR has a smaller log size. The average reduction is 28%, which is a result of RTR’s ability to add more dependencies with the same IC stride.

Set/LRU Approximation. Figure 7.7 shows the effectiveness of the Set/LRU times-

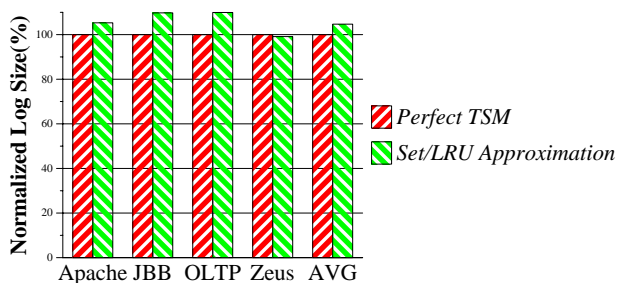


Figure 7.7 Set/LRU versus perfect TSM. The log size of the 12KB timestamp memories with Set/LRU approximation is no more than 10% larger than the log size of infinite timestamp memories. The raw data of the simulation results are shown in Table A.13 and Table A.15.

tamp approximation method. With 12 KB timestamp memories and the approximation, the recorder increases the log size by no more than 10%, compared to the infinite timestamp memories. Note that Set/LRU slightly outperforms perfect timestamp memory for Zeus. This is possible because RTR is a greedy heuristic and precise timestamps do not always help in transitive reduction. In other words, an approximated timestamp may help in transitive reduction.

Additional impacts of Set/LRU can be seen in Figure 7.3. In that figure, we use 64 KB timestamp memories and vary the timestamp approximation method and the timestamp memory associativity. That figure reveals: (1) Set/LRU dramatically reduces the extra log caused by misses in the timestamp memory, because the conflicts are more likely reducible by both TR and RTR; (2) Set/LRU benefits from the associativity of the timestamp memory, because higher associativity enables better approximations; (3) The RTR algorithm works better than TR if it is combined with Set/LRU, because Set/LRU enables more flexibility for RTR to introduce stricter dependencies. RTR performs worse than TR when Set/LRU is

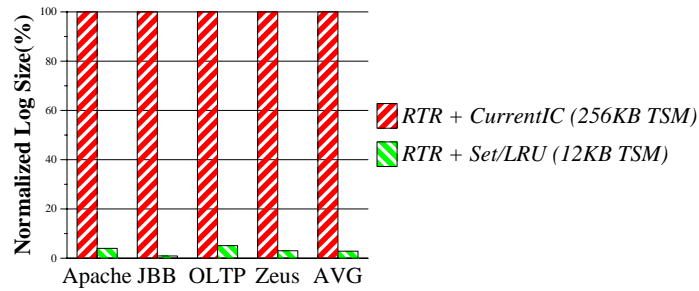


Figure 7.8 Set/LRU versus Current IC. The log size of the 12KB timestamp memories with Set/LRU is 25X smaller than the log size of using Current IC approximation with even larger (256KB) timestamp memories. The raw data of the simulation results are shown in Table A.13 and Table A.16.

not used, because each RTR log entry is variable length, which incurs a delimiter overhead. This overhead is most severe when log entry contains one dependence (RTR often cannot find multiple vectorizable dependencies with the CurrentIC timestamp approximation).

In Figure 7.8, we compared the recorder log size of a 12 KB Set/LRU timestamp memory and the recorder log size of a 256 KB timestamp memory used in our earlier recorder design of the Flight Data Recorder (FDR) [73]. FDR did not use the Set/LRU approximation method, but used the Current IC approximation method. Even with the large timestamp memory, the Current IC approximation method introduced many irreducible dependencies. In the end, with a smaller timestamp memory, Set/LRU is able to reduce the log size by 25 folds compared with the larger timestamp memory with the Current IC approximation method.

The Set/LRU approximation clearly works well in avoiding the extra dependencies that are generated when a timestamp misses from the timestamp memory. There are at least two reasons why Set/LRU works well. First, temporal locality in

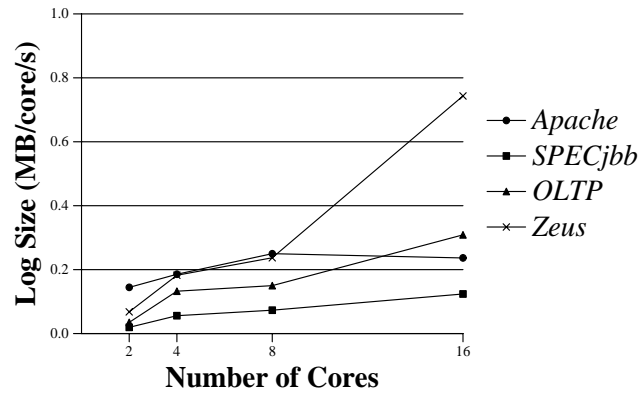


Figure 7.9 Scalability in terms of number of processors. The recorder’s log size scales modestly with respect to the number of cores. The raw data of the simulation results are shown in Table A.17.

program memory reference stream makes the timestamp of the LRU block relatively “old”, because the timestamp is often generated in the distant past. Second, the RTR algorithm create stricter dependencies, which increase the chances for a Set/LRU approximated dependencies to be reduced by transitivity.

7.4 Scalability of Determinizer/CMP

Finally, Figure 7.9 shows the scalability of determinizer/CMP. In this experiment, we increase the number of cores in the CMP system from 2 to 16. The hardware cost of determinizer/CMP increases proportionally with the number of cores, but by a small factor. The log size increases slowly for each processor core, with the exception of Zeus. This shows that the interactions between threads increase modestly as more threads are added to the system. For Zeus, our simulation data (Table A.17) indicates that the workload itself is not scalable with 16 processors.

Chapter 8

A Qualitative Discussion of Race Replay

In this dissertation, we focus on only the recording phase of multithreaded deterministic replay. After the recording phase, however, a *replayer* is needed to recreate the original execution. The replayer must perform *race replay*, where the recorded dependence log is used to enforce that the same races are recreated during the replay.

In this chapter, we briefly overview two potential methods for race replay. First, we discuss race replay on uniprocessor systems with a sequential replayer in Section 8.1. Then, we discuss race replay on multiprocessor systems with a parallel replayer in Section 8.2. The high level bits of the TSO order-value-hybrid replay has been discussed in Section 3.6. Therefore, we omit the details of how to couple the order-based replay with the value-based replay in this chapter.

8.1 Sequential Replay

8.1.1 Algorithm

Table 8-1 shows the high-level algorithm of *sequential replay*. For each thread,

Table 8-1 Sequential Replay Algorithm

DATA STRUCTURES DEP_T := {int destination_IC, int source_tid, int source_IC} THREAD_T := {int tid, int IC, DEP_T next_dep}
VARIABLES THREAD_T T[]; // an array of threads int current_tid; // the executing thread
ALGORITHM <i>// Initially, T.IC equals 0 for all threads (T[0] ... T[n])</i> current_tid := 0; while (false == ALL_THREAD_ENDED()) { // loop until all threads end <i>// Switch to the next thread in round-robin fashion if this thread has ended</i> if (THREAD_ENDED(current_tid)) { current_tid := NEXT_TID(current_tid); } <i>// Execute this thread until it is block by a dependence</i> while (T[T[current_tid].next_dep.source_tid].IC >= T[current_tid].next_dep.source_IC) { T[current_tid].next_dep := READ_NEXT_DEPENDENCE(current_tid); while (T[current_tid].IC < T[current_tid].next_dep.destination_IC) { T[current_tid].IC++; EXECUTION_ONE_INSTRUCTION(current_tid); } } <i>// switch to the source thread</i> current_tid := T[current_tid].next_dep.source_tid; }

the replayer keeps track of the thread's dynamic instruction count (*IC*) and the next dependence that the replayer has to enforce. Each dependence data structure (*DEP_T*) consists of three integers that are read from the thread's dependence log. The three integers are (1) the destination *IC* of this dependence, (2) the source thread, and (3) the source *IC* of the source thread.

The sequential replay works by executing instructions from a thread until the thread is blocked by a dependence. The replayer starts by reading a dependence from the initial thread's dependence log (created during the recording phase). The

replayer performs a check when the thread executes the destination instruction (`destination_IC`). A thread is blocked if and only if the source thread (`source_tid`) has not executed the instruction with the source IC (`source_IC`). If a thread is blocked, the replayer switches the execution context to the source thread and continues executing instructions from the source thread. Otherwise, the replayer reads this thread's next dependence and starts the next iteration of "execution—check for dependence—read next dependence".

8.1.2 Implementations and Performance

There are two implementation strategies for the sequential replay algorithm. First, a replayer can be *interpretation-based*. Because the replay is done instruction-by-instruction, the replayer can act like a functional simulator that interprets each instruction. Interpretation can incur significant runtime overhead. But, it can be easier to switch between threads, or to hide the replayer from the replayed program, *etc.* Given the commercial availability of system-level functional simulators, such as Simics [68], interpretation-based replayers may be relatively easy to implement.

Second, a replayer can employ the *direct execution* technique. By direct execution, the replayer uses the processor to execute the instructions from the replayed programs directly. To regain the control from the replayed program, the replayer may need to interrupt the replayed execution after a predefined number of dynamic instructions. Although direct execution incurs less overhead than interpretation, it can be more expensive in switching the thread context and it can be

Table 8-2 Parallel Replay Algorithm

DATA STRUCTURES DEP_T := {int destination_IC, int source_tid, int source_IC} THREAD_T := {int tid, int IC, DEP_T next_dep}
VARIABLES THREAD_T T[]; // an array of threads
ALGORITHM INDEPENDENTLY RUNNING ON EACH PROCESSOR <i>// Initially, T.IC equals 0 for all threads (T[0] ... T[n])</i> while (THREAD_ENDED()) { // loop until this thread ends <i>// Execute this thread until it is block by a dependence</i> while (T[T[current_tid].next_dep.source_tid].IC >= T[current_tid].next_dep.source_IC) { T[current_tid].next_dep := READ_NEXT_DEPENDENCE(current_tid); while (T[current_tid].IC < T[current_tid].next_dep.destination_IC) { T[current_tid].IC++; EXECUTION_ONE_INSTRUCTION(current_tid); } } <i>// busy wait until the source thread catches up</i> while (T[T[current_tid].next_dep.source_tid].IC < T[current_tid].next_dep.source_IC) {} }

tricky to hide the replayer from the replayed programs (*e.g.*, dealing with the address spaces of the replayer and the replayed program).

In general, sequential replay can suffer from high runtime overhead for two reasons. First, executing multiple threads serially is slow. Second, interpretation or frequent interrupts and context switches of the direct execution is slow. If the replay performance is important, parallel replay is desired.

8.2 Parallel Replay

8.2.1 Algorithm

Table 8-2 shows the high-level algorithm for *parallel replay*. The algorithm differs from the sequential replay algorithm in that it runs each thread on differ-

ent processors and employs busy-waiting instead of context switches. The $T[\text{current_tid}].IC$ variable of a thread is only modified by the thread itself, but it is continuously read by other threads. This access pattern has suboptimal performance in an invalidation-based cache coherence protocol. An update-based coherence policy can be considered for the $T[\text{current_tid}].IC$ variable.

8.2.2 Implementations and Performance

Like the sequential replayer, the parallel replayer can be interpretation-based or direct-execution-based. In parallel replay, the direct-execution-based replayer has a significant performance advantage because there is no need to do context switches. Furthermore, because there are no context switches, the direct-execution-based replayers may be implemented in hardware to avoid frequent interrupts. This is because context switches are harder to perform in hardware.

Like our hardware race recorder, a hardware replayer can piggyback itself onto the cache coherence mechanisms. The value of the $T[\text{current_tid}].IC$ variable can be piggybacked on the cache coherence messages. A potential complication to the hardware replayer is the out-of-order execution. When the processor executes instructions out-of-order, the dependencies may not be correctly enforced and those dependencies that were transitively reduced can be violated. A solution is to perform necessary checks at the in-order commit stage to detect when the in-order semantics are violated.

We speculate that the replay is often slower than the recording, because of the extra waiting time due to the dependencies and the communication latency of the

`T[current_tid].IC` variables. The new RTR recording algorithm in this dissertation can impact the replay performance in both directions. The RTR algorithm creates stricter dependencies, which can increase the wait time during replay. On the other hand, the RTR algorithm logs 30% less dependencies, which reduces the impact of the `T[current_tid].IC` communication latency. (Frequent dependence checks require frequently updating the `T[current_tid].IC` variable.)

Chapter 9

Conclusion and Future Work

This dissertation studies how to perform hardware-based race recording that *simultaneously* achieves small log size, extremely low runtime overhead, low memory overhead (*i.e.*, hardware cost) and broad applicability. We contribute the following new ideas to the race recording literature. We propose and evaluate a new race recording algorithm, Regulated Transitive Reduction (RTR), which concisely records multithreaded executions by creating stricter and vectorizable dependencies between threads. We design and implement a hardware race recorder that piggybacks itself onto the hardware cache coherence mechanisms in multiprocessor systems. We propose and evaluate a Set/LRU timestamp approximation method that significantly reduces the hardware cost of the recorder to a small constant of 12 KB per processor. We propose an order-value-hybrid recording method that enables race recording on multiprocessor systems with the TSO memory model. We are confident that our *effective and inexpensive* race recorder will be a key enabler of deterministic replay for multithreaded programs.

9.1 Future Research Directions

The directions for future research include the following.

9.1.1 Race Recording Algorithms

Some interesting angles to further improve the race recording algorithm are the following. First, is there a more compact format in recording the dependencies? Second, can the replay scope (*e.g.*, instruction, e-block [42], parallel slice [5], single thread [46], and multiple threads/full system [73]) be changed to improve the tradeoff between the log size and the replay flexibility? Finally, can we better understand a multithreaded execution by summarizing the execution in an easily understandable way? For example, can we use an equivalent sequential scheduling of minimum number of threads (as allowed by the control flow graph of the code) and minimum number of context switches to understand a multithreaded execution? Or, can we develop a way to automatically separate the conflicts that do and do not change the results of a program execution?

9.1.2 Race Recording Implementations

There is a great interest in performing race recording entirely in software. The new recording algorithm and the new timestamp approximation method in this dissertation are directly applicable to software race recorders. Future research can focus on how to reduce the runtime overhead without the hardware assistance.

The hardware race recorder in this dissertation does not support processor-thread virtualization. Virtualizing the race recorder is important if we want to

record and replay each individual process instead of the whole system. But, virtualization is not required. With coscheduling virtual machines, the entire virtual machine is recorded and replayed and it is unnecessary to distinguish different threads running on a virtual processor.

Future research is required to extend race recording on SMT systems. The problem with SMT processors is the lack of coherence transactions in the shared L1 cache. Adding extra metadata in the L1 cache may help a race recorder to detect the conflicts without coherence transactions.

Finally, future research is needed to support race recording on those systems that implement a shared memory model that is weaker than TSO. This problem represents an intellectual challenge for existing race recorders and an potential solution is to relax the in-order replay assumption in this dissertation. Out-of-order replay is potentially more complex and makes it harder to apply transitive reduction in recording. However, out-of-order replay can potentially achieve high performance and avoid replay deadlocks caused by weak memory models. Out-of-order replay perhaps maps better to the underlying hardware as well.

9.1.3 Race Replay

While Chapter 8 qualitatively discusses the potential race replay methods, future work should study the replayer implementation and replayer performance in more detail. The future research should answer the question that what level of replay slowdown is caused by the stricter dependencies created by the RTR algorithm? Furthermore, it would be useful to study what level of replay slowdown is

acceptable in different applications (*e.g.*, debugging) of deterministic replay.

9.1.4 Deterministic Replay Applications

In the end, deterministic replay are useful only after it is applied to specific applications, such as cyclic debugging, fault-tolerance, security analysis, data recovery and predictable synchronization. More work needs to be done in providing integrated recording and replay solutions for these applications. With the new flexibility enabled by the virtual machine technology, we anticipate more applications to be developed around deterministic replay. In a way, a virtual machine helps overcome the space constraint in computing, deterministic replay helps overcome the time constraint in computing.

Bibliography

- [1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, December 1996.
- [2] Ardsher Ahmed, Pat Conway, Bill Hughes, and Fred Weber. Amd opteron shared memory mp systems. In *Proceedings of the 14th HotChips Symposium*, August 2002.
- [3] Alaa R. Alameldeen, Milo M. K. Martin, Carl J. Mauer, Kevin E. Moore, Min Xu, Daniel J. Sorin, Mark D. Hill, and David A. Wood. Simulating a \$2m commercial server on a \$2k pc. *IEEE Computer*, 36(2):50–57, February 2003.
- [4] Arvind and Jan-Willem Maessen. Memory model = instruction reordering + store atomicity. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, June 2006.
- [5] David F. Bacon and Seth Copen Goldstein. Hardware-assisted replay of multiprocessor programs. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, published in *ACM SIGPLAN Notices*, pages 194–206, 1991.
- [6] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of the 1998 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 151–160, June 1998.
- [7] Luiz Barroso. The price of performance: An economic case for chip multiprocessing. *ACM Queue*, 3(7), September 2005.
- [8] Luiz Andre Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzky, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 282–293, June 2000.
- [9] Sanjay Bhansali, Wen-Ke Chen, Stuart De Jong, Andrew Edwards, and Milenko Drinic. Framework for instruction-level tracing and analysis of

- programs. In *Second International Conference on Virtual Execution Environments*, 2006.
- [10] Nicholas S. Bowen and Dhiraj K. Pradhan. Processor- and memory-based checkpoint and rollback recovery. *IEEE Computer*, 26(2):22–31, February 1993.
 - [11] Harold W. Cain and Mikko H. Lipasti. Memory ordering: A value-based approach. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004.
 - [12] Jong-Deok Choi and Harini Srinivasan. Deterministic replay of Java multithread applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT-98)*, pages 48–59, August 1998.
 - [13] Alex C. Chow. Personal communication, January 2006.
 - [14] Alex Chunghen Chow, Gordon C. Fossum, and Daniel A. Brokenshire. A programming example: Large fft on the cell broadband engine. In *Proceeding of the 2005 Global Signal Processing Expo (GSPx)*, October 2005.
 - [15] Frank Cornelis, Michiel Ronsse, and Koen De Bosschere. Tornado: A novel input replay tool. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'03)*, pages 23–26, 2003.
 - [16] F. Cristian and F. Jahanian. A timestamp-based checkpointing protocol for long-lived distributed computations. In *Proceedings of IEEE Symposium on Reliable Distributed Systems*, pages 12–20, 1991.
 - [17] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation*, pages 211–224, December 2002.
 - [18] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5):12–18, September/October 1997.
 - [19] Krisztian Flautner, Rich Uhlig, and Trevor Mudge. Thread level parallelism and interactive performance of desktop applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 129–138, November 2000.

- [20] Krisztian Flautner, Rich Uhlig, Steven Reinhardt, and Trevor Mudge. Threadlevel parallelism of desktop applications. In *Workshop on Multi-threaded Execution, Architecture and Compilation, Toulouse, France, January 2000*.
- [21] A. Georges, M. Christiaens, M. Ronsse, and K. De Bosschere. Jarec: a portable record/replay environment for multi-threaded java applications. *Software: Practice and Experience*, 34(6):523–547, 2004.
- [22] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proceedings of the International Conference on Parallel Processing*, volume I, pages 355–364, August 1991.
- [23] Chris Gniady, Babak Falsafi, and T.N. Vijaykumar. Is sc + ilp = rc? In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 162–171, May 1999.
- [24] Erik Hagersten and Michael Koster. Wildfire: A scalable path for smps. In *Proceedings of the Fifth IEEE Symposium on High-Performance Computer Architecture*, pages 172–181, January 1999.
- [25] G. Janakiraman and Yuval Tamir. Coordinated checkpointing-rollback error recovery for distributed shared memory multicomputers. In *Symposium on Reliable Distributed Systems*, pages 42–51, 1994.
- [26] Y. H. Kim, M. D. Hill, and D. A. Wood. Implementing stack simulation for highly-associative memories. In *Proceedings of the 1991 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 212–213, May 1991.
- [27] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparcc processor. *IEEE Micro*, 25(2):21–29, Mar/Apr 2005.
- [28] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.
- [29] Rakesh Kumar, Victor Zyuban, and Dean Tullsen. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005.

- [30] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
- [31] Thomas J. Leblanc and John M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.
- [32] Kevin Lepak. Personal communication, March 2006.
- [33] Luk Levrouw and Koenraad Audenaert. Minimizing the log size for execution replay of shared-memory programs. In *Lecture Notes In Computer Science; Vol. 854, Parallel Processing: CONPAR 94 - VAPP VI, Third Joint International Conference on Vector and Parallel Processing, Linz, Austria, September 6-8, 1994, Proceedings*, pages 76–87, 1994.
- [34] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and migration of unix processes in the condor distributed processing system. Technical Report 1346, Computer Sciences Department, University of Wisconsin–Madison, April 1997.
- [35] Jack L. Lo, Luiz Andre Barroso, Susan J. Eggers, Kouros Gharachorloo, Henry M. Levy, and Sujay S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 39–50, June 1998.
- [36] Dominic Lucchetti, Steven K. Reinhardt, and Peter M. Chen. ExtraVirt: Detecting and recovering from transient processor faults. In *2005 Symposium on Operating System Principles work-in-progress session*, October 2005.
- [37] Peter S. Magnusson et al. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, February 2002.
- [38] Milo M.K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet’s general execution-driven multiprocessor simulator (gems) toolset. *Computer Architecture News*, pages 92–99, September 2005.
- [39] Michael R. Marty, Jesse D. Bingham, Mark D. Hill, Alan J. Hu, Milo M. K. Martin, and David A. Wood. Improving multiple-cmp systems using token coherence. In *Proceedings of the Eleventh IEEE Symposium on High-Performance Computer Architecture*, February 2005.

- [40] Jim Mauro and Richard McDougall. *Solaris Internals: Core Kernel Architecture*. Prentice Hall, 2001.
- [41] John M. Mellor-Crummey and Thomas J. Leblanc. A software instruction counter. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 78–86, April 1989.
- [42] Barton P. Miller and Jong-Deok Choi. A mechanism for efficient debugging of parallel programs. In *Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 135–144, June 1988.
- [43] Mark S. Miller. Uses of determinism. <http://www.erights.org/elang/concurrency/determinism/index.html>.
- [44] Sang Lyul Min and Jong-Deok Choi. An efficient cache-based access anomaly detection scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 235–244, April 1991.
- [45] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. Logtm: Log-based transactional memory. In *Proceedings of the Twelfth IEEE Symposium on High-Performance Computer Architecture*, February 2006.
- [46] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 284–295, June 2005.
- [47] Robert H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging (PADD)*, pages 1–11, 1993.
- [48] Robert H. B. Netzer and Barton P. Miller. What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.
- [49] CJ Newburn. Personal communication, October 2003.
- [50] Robert O’Callahan. Blog: Night of the living threads, December 2005.

- [51] Cherri M. Pancake and Robert H. B. Netzer. A bibliography of parallel debuggers, 1993 edition. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging (PADD)*, pages 169–186, 1993.
- [52] Christos Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, Rockville, Maryland, 1986.
- [53] James S. Plank, Kai Li, and Michael A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, October 1998.
- [54] Milos Prvulovic. Cord: Cost-effective (and nearly overhead-free) order recording and data race detection. In *Proceedings of the Twelfth IEEE Symposium on High-Performance Computer Architecture*, February 2006.
- [55] Milos Prvulovic and Josep Torrellas. Reenact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 110–121, June 2003.
- [56] Brad Richards and James R. Larus. Protocol-based data-race detection. In *SIGMETRICS symposium on Parallel and Distributed Tools*, pages 40–47, 1998.
- [57] Michiel Ronsse and Koen De Bosschere. Non-intrusive on-the-fly data race detection using execution replay. In *Automated and Algorithmic Debugging*, November 2000.
- [58] Michiel Ronsse, Koen De Bosschere, and Jacques Chassin de Kergommeaux. Execution replay and debugging. In *Automated and Algorithmic Debugging*, November 2000.
- [59] Mendel Rosenblum. Virtual is better than real. http://www.vmware.com/vmworld/2005/keynote_rosenblum.pdf.
- [60] Mark Russinovich and Bryce Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *Proceedings of the SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 258–266, June 1990.
- [61] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

- [62] German Shegalov. *Integrated Data, Message, and Process Recovery for Failure Masking in Web Services*. PhD thesis, Saarland University, August 2005.
- [63] Daniel J. Sorin. *Using Lightweight Checkpoint/Recovery to Improve the Availability and Designability of Shared Memory Multiprocessors*. PhD thesis, University of Wisconsin, August 2002.
- [64] Daniel J. Sorin, Milo M. K. Martin, Mark D. Hill, and David A. Wood. Safetynet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 123–134, May 2002.
- [65] Johnny Srouji, Paul Schuster, Maury Bach, and Yulik Kuzmin. A transparent checkpoint facility on NT. In *The 2nd USENIX Windows NT Symposium*, pages 77–86, 1998.
- [66] Herb Sutter and James Larus. Software and the concurrency revolution. *ACM Queue*, 3(7), September 2005.
- [67] Paul Sweazey and Alan Jay Smith. A class of compatible cache consistency protocols and their support by the IEEE Futurebus. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 414–423, June 1986.
- [68] Virtutech AB. Simics full system simulator. <http://www.simics.com/>.
- [69] VMware. <http://www.vmware.com/>.
- [70] David L. Weaver and Tom Germond, editors. *SPARC Architecture Manual (Version 9)*. PTR Prentice Hall, 1994.
- [71] Fred Weber. Amd’s next generation microprocessor architecture, October 2001.
- [72] Min Xu, Rastislav Bodik, and Mark D. Hill. Considerably longer race recording using less timestamp memory. In *Submitted for Publication*.
- [73] Min Xu, Rastislav Bodik, and Mark D. Hill. A “flight data recorder” for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 122–133, June 2003.
- [74] Min Xu, Rastislav Bodik, and Mark D. Hill. A serializability violation detector for shared-memory server programs. In *Proceedings of the SIG-*

PLAN 2005 Conference on Programming Language Design and Implementation, pages 1–14, June 2005.

- [75] Kenneth C. Yeager. The mips r10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.
- [76] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iwatcher: Efficient architectural support for software debugging. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 224, June 2004.
- [77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

Appendix A

Tables of the Simulation Results

Table A.1 Varying the Timestamp Memory Size — Apache

TIMESTAMP MEMORY CONFIGURATION ⁱ						RESULTS				
SETS	WAYS	No. OF TS	BITS	SET / LRU ⁱⁱ	SIZE (KB)	RUNTIME ⁱⁱⁱ (CYCLES)	TR LOG SIZE (BYTES)		RTR LOG SIZE (BYTES)	
							UNCOMP	COMP	UNCOMP ^{iv}	COMP
One Timestamp Per Block										
4	64	1	64	1	2	155705285	1730896	1102106	548371	253866
8	64	1	64	1	4	155103975	971020	653752	350894	181590
16	64	1	64	1	8	152580716	662456	438679	261695	143147
32	64	1	64	1	16	129342010	417272	266200	199775	115214
64	64	1	64	1	32	116720257	348488	212971	182790	108908
128	64	1	64	1	64	164423923	437432	270172	265076	155231
256	64	1	64	1	128	120709319	380220	215678	201726	117700
512	64	1	64	1	256	114948318	329024	195292	173750	104133
1024	64	1	64	1	512	152520141	362812	219101	209378	125124
2048	64	1	64	1	1024	116810757	339976	201826	191424	113512
4096	64	1	64	1	2048	139465480	335748	198847	187460	110701
Two Timestamps Per Block										
2	64	2	64	1	2	131156993	2818688	1596545	824422	328567
4	64	2	64	1	4	129423421	1422336	921952	439610	199449
8	64	2	64	1	8	142509081	825860	588280	273574	140886
16	64	2	64	1	16	129267418	397256	300693	172578	95583
32	64	2	64	1	32	155722014	355876	271474	195658	112319
64	64	2	64	1	64	151597000	245512	189701	165225	98947
128	64	2	64	1	128	117409877	150888	115378	130574	78504
256	64	2	64	1	256	117131779	139780	107562	122952	75852
512	64	2	64	1	512	116417557	143732	109331	128190	77393
1024	64	2	64	1	1024	115505188	138200	106642	123418	75727
2048	64	2	64	1	2048	121596455	142904	108826	122223	75770

- i. In all tables, this configuration is the same for both the L1 and the L2 timestamp memories.
- ii. All tables in this appendix use the following notion: Set/LRU: 1; CurrentIC: 0.
- iii. For each simulation in a table, we run a fixed number of transactions.
- iv. In all tables, the log size is aggregated from all processor cores.

Table A.2 Varying the Timestamp Memory Size — OLTP

TIMESTAMP MEMORY CONFIGURATION						RESULTS				
SETS	WAYS	NO. OF TS	BITS	SET / LRU	SIZE (KB)	RUNTIME (CYCLES)	TR LOG SIZE (BYTES)		RTR LOG SIZE (BYTES)	
							UNCOMP	COMP	UNCOMP	COMP
One Timestamp Per Block										
4	64	1	64	1	2	129022571	1596936	1014809	500410	251878
8	64	1	64	1	4	127768264	1048832	598547	351953	187597
16	64	1	64	1	8	121161875	877220	413864	305198	162874
32	64	1	64	1	16	127814399	946896	405787	327501	170501
64	64	1	64	1	32	123620093	892564	388978	310515	163778
128	64	1	64	1	64	136891246	1014624	432241	350028	187938
256	64	1	64	1	128	148938326	1093084	460591	369656	194062
512	64	1	64	1	256	133503340	952576	410454	333455	174835
1024	64	1	64	1	512	117253537	835116	362140	290987	151496
2048	64	1	64	1	1024	140435692	1026768	440079	360930	192172
4096	64	1	64	1	2048	130171955	959068	405007	333345	171377
Two Timestamps Per Block										
2	64	2	64	1	2	144457398	3170464	2005635	881321	436387
4	64	2	64	1	4	137690908	1431476	1021573	435037	229018
8	64	2	64	1	8	153010625	744060	575314	258266	148902
16	64	2	64	1	16	113199451	254244	205802	121284	75633
32	64	2	64	1	32	115663478	139556	110416	99840	64428
64	64	2	64	1	64	123422735	119312	92491	95691	62056
128	64	2	64	1	128	120424585	114444	88134	94578	61442
256	64	2	64	1	256	132914663	129136	98586	106598	69011
512	64	2	64	1	512	114045627	109884	84998	90150	59145
1024	64	2	64	1	1024	119040425	118456	90504	97019	62983
2048	64	2	64	1	2048	125920146	122248	93937	101405	65980

Table A.3 Varying the Timestamp Memory Size — SPECjbb

TIMESTAMP MEMORY CONFIGURATION						RESULTS				
SETS	WAYS	NO. OF TS	BITS	SET / LRU	SIZE (KB)	RUNTIME (CYCLES)	TR LOG SIZE (BYTES)		RTR LOG SIZE (BYTES)	
							UNCOMP	COMP	UNCOMP	COMP
One Timestamp Per Block										
4	64	1	64	1	2	180622190	1772952	1160423	513542	281790
8	64	1	64	1	4	180551598	1330904	652820	430020	229318
16	64	1	64	1	8	180622967	1245868	499175	416444	217690
32	64	1	64	1	16	180563630	1231732	479095	411185	214615
64	64	1	64	1	32	180613207	1228920	478158	405218	211420
128	64	1	64	1	64	180543621	1231292	478324	408891	213233
256	64	1	64	1	128	180591513	1229724	478554	406919	212274
512	64	1	64	1	256	180610743	1233096	479694	408590	213099
1024	64	1	64	1	512	180597101	1236664	480441	413028	214954
2048	64	1	64	1	1024	180611250	1237252	482537	411387	215048
4096	64	1	64	1	2048	180604851	1229276	478683	410520	213693
Two Timestamps Per Block										
2	64	2	64	1	2	180595905	3210876	2350871	728940	386576
4	64	2	64	1	4	180615752	1393336	1124264	331562	191894
8	64	2	64	1	8	180622289	577500	498402	150058	93959
16	64	2	64	1	16	180611408	225284	203174	81055	53944
32	64	2	64	1	32	180615998	95676	84752	57209	38873
64	64	2	64	1	64	180616139	61864	50676	50462	34202
128	64	2	64	1	128	180569561	59316	48165	49376	33469
256	64	2	64	1	256	180605349	59804	48485	49758	33657
512	64	2	64	1	512	180611775	60716	49336	50422	34105
1024	64	2	64	1	1024	180566969	61536	50049	50975	34359
2048	64	2	64	1	2048	180581766	63120	51432	52063	35033

Table A.4 Varying the Timestamp Memory Size — Zeus

TIMESTAMP MEMORY CONFIGURATION						RESULTS				
SETS	WAYS	NO. OF TS	BITS	SET / LRU	SIZE (KB)	RUNTIME (CYCLES)	TR LOG SIZE (BYTES)		RTR LOG SIZE (BYTES)	
							UNCOMP	COMP	UNCOMP	COMP
One Timestamp Per Block										
4	64	1	64	1	2	106281354	1125152	703185	370328	200770
8	64	1	64	1	4	105017258	733852	408772	280345	162644
16	64	1	64	1	8	105584183	709112	339105	284615	165907
32	64	1	64	1	16	105456154	676172	321122	268118	155781
64	64	1	64	1	32	105292900	716204	330338	272803	158858
128	64	1	64	1	64	104929837	686136	326270	267897	156205
256	64	1	64	1	128	105920493	699576	331513	265659	155419
512	64	1	64	1	256	105329171	702156	329220	265970	155070
1024	64	1	64	1	512	105559202	676220	323626	253000	148894
2048	64	1	64	1	1024	105571127	687516	327353	266708	157574
4096	64	1	64	1	2048	105359127	688172	328010	267874	156464
Two Timestamps Per Block										
2	64	2	64	1	2	105546061	2242528	1314870	660659	289529
4	64	2	64	1	4	105027444	988216	677017	301475	163799
8	64	2	64	1	8	105032765	432796	328129	173646	107763
16	64	2	64	1	16	105217990	224108	172924	127348	82851
32	64	2	64	1	32	105320940	151792	113815	113757	74779
64	64	2	64	1	64	105567545	146036	107189	117258	75254
128	64	2	64	1	128	105302510	148388	109524	121532	78945
256	64	2	64	1	256	105754143	145672	106514	116572	75340
512	64	2	64	1	512	105798026	145324	106655	112156	73711
1024	64	2	64	1	1024	105701883	146872	107939	113569	74719
2048	64	2	64	1	2048	105906982	151928	110695	118530	77679

Table A.5 Varying the Width of Partial Timestamps — Apache

TIMESTAMP MEMORY CONFIGURATION						RESULTS				
SETS	WAYS	NO. OF TS	BITS	SET / LRU	SIZE (KB)	RUNTIME (CYCLES)	TR LOG SIZE (BYTES)		RTR LOG SIZE (BYTES)	
							UNCOMP	COMP	UNCOMP	COMP
64	64	2	8	1	8	121408969	4923160	2890702	5216925	2890728
64	64	2	11	1	11	148833248	2466284	1706281	2511223	1380621
64	64	2	14	1	14	118229889	733712	572100	722093	416475
64	64	2	17	1	17	126428735	374200	304968	337003	199622
64	64	2	20	1	20	117574315	159804	124766	129302	78390
64	64	2	23	1	23	116117682	158960	123634	125974	76703
64	64	2	26	1	26	158567700	273184	208483	193109	115132
64	64	2	29	1	29	165784196	266708	206324	195032	116870
64	64	2	32	1	32	148575514	215724	169097	144671	86250
64	64	2	35	1	35	123806410	178816	138960	134689	79962

Table A.6 Varying the Width of Partial Timestamps — OLTP

TIMESTAMP MEMORY CONFIGURATION						RESULTS				
SETS	WAYS	NO. OF TS	BITS	SET / LRU	SIZE (KB)	RUNTIME (CYCLES)	TR LOG SIZE (BYTES)		RTR LOG SIZE (BYTES)	
							UNCOMP	COMP	UNCOMP	COMP
64	64	2	8	1	8	121401925	4862800	2954108	5619267	3131942
64	64	2	11	1	11	124167209	2136552	1526021	2286149	1316229
64	64	2	14	1	14	143850181	922752	733494	948821	563643
64	64	2	17	1	17	121035668	313752	263970	308950	190444
64	64	2	20	1	20	136807879	154720	124912	131933	85240
64	64	2	23	1	23	119013039	117884	91865	96045	62737
64	64	2	26	1	26	127307389	123984	96329	100134	64962
64	64	2	29	1	29	131830737	131976	101377	106535	68653
64	64	2	32	1	32	138037856	134688	104736	108968	70982
64	64	2	35	1	35	149973138	150844	116143	120572	77438

Table A.7 Varying the Width of Partial Timestamps — SPECjbb

TIMESTAMP MEMORY CONFIGURATION						RESULTS				
SETS	WAYS	NO. OF TS	BITS	SET / LRU	SIZE (KB)	RUNTIME (CYCLES)	TR LOG SIZE (BYTES)		RTR LOG SIZE (BYTES)	
							UNCOMP	COMP	UNCOMP	COMP
64	64	2	8	1	8	180551967	8605372	5365206	10741731	6019243
64	64	2	11	1	11	180609768	4284324	3090950	4608267	2653451
64	64	2	14	1	14	180579118	1650932	1343577	1661355	970649
64	64	2	17	1	17	180612835	584672	520263	577948	345516
64	64	2	20	1	20	180589159	139392	130356	128704	83490
64	64	2	23	1	23	180558398	69952	58996	57721	38903
64	64	2	26	1	26	180607574	61568	50512	50321	34094
64	64	2	29	1	29	180624012	61644	50610	50208	34011
64	64	2	32	1	32	180609083	61500	50355	49757	33797
64	64	2	35	1	35	180618704	61920	50726	49844	33945

Table A.8 Varying the Width of Partial Timestamps — Zeus

TIMESTAMP MEMORY CONFIGURATION						RESULTS				
SETS	WAYS	NO. OF TS	BITS	SET / LRU	SIZE (KB)	RUNTIME (CYCLES)	TR LOG SIZE (BYTES)		RTR LOG SIZE (BYTES)	
							UNCOMP	COMP	UNCOMP	COMP
64	64	2	8	1	8	104645052	4029840	2318975	4203440	2281318
64	64	2	11	1	11	105330668	1617048	1119303	1632186	906390
64	64	2	14	1	14	105232638	624596	481855	615881	352508
64	64	2	17	1	17	106038327	286880	232316	268987	162462
64	64	2	20	1	20	105516083	149704	112945	122595	80277
64	64	2	23	1	23	106015103	144976	106936	111509	73071
64	64	2	26	1	26	105900292	143004	105260	109627	72173
64	64	2	29	1	29	104717100	141172	103640	107467	70748
64	64	2	32	1	32	105196896	142096	105654	110856	73205
64	64	2	35	1	35	105239636	145020	107118	114732	74546

Table A.9 Varying the Associativity and the Approximation Method — Apache

TIMESTAMP MEMORY CONFIGURATION						RESULTS				
SETS	WAYS	NO. OF TS	BITS	SET / LRU	SIZE (KB)	RUNTIME (CYCLES)	TR LOG SIZE (BYTES)		RTR LOG SIZE (BYTES)	
							UNCOMP	COMP	UNCOMP	COMP
CurrentIC Timestamp Approximation										
2048	2	2	64	0	64	137368745	8748876	3906877	15140634	5994371
1024	4	2	64	0	64	135250145	8563372	3835115	13803590	5839764
512	8	2	64	0	64	131228219	8202492	3785256	13945757	5738989
256	16	2	64	0	64	160041722	9820652	4470030	16083779	6748183
128	32	2	64	0	64	171703764	10262832	4736552	17138910	7099884
64	64	2	64	0	64	138904857	8851844	3951286	14702788	6068201
32	128	2	64	0	64	114855359	7077600	3336305	12044050	4990766
16	256	2	64	0	64	146678311	9255112	4126251	15399372	6331054
8	512	2	64	0	64	158956074	9839556	4444104	16247585	6746491
4	1024	2	64	0	64	148845897	9335356	4154720	16304802	6370749
Set/LRU Timestamp Approximation										
2048	2	2	64	1	64	118445565	198932	157725	161161	96542
1024	4	2	64	1	64	148716127	211952	167299	164397	98602
512	8	2	64	1	64	117403889	169224	132305	138706	83492
256	16	2	64	1	64	126308613	171216	135251	129521	77898
128	32	2	64	1	64	140945339	207024	159771	158892	94035
64	64	2	64	1	64	161768400	252812	196306	179816	106877
32	128	2	64	1	64	152521169	267984	204200	162140	95962
16	256	2	64	1	64	124341101	226204	168770	132208	78319
8	512	2	64	1	64	155736030	427888	297184	175448	99439
4	1024	2	64	1	64	122419806	382764	246228	133810	77755

Table A.10 Varying the Associativity and the Approximation Method — OLTP

TIMESTAMP MEMORY CONFIGURATION						RESULTS				
SETS	WAYS	No. OF TS	BITS	SET / LRU	SIZE (KB)	RUNTIME (CYCLES)	TR LOG SIZE (BYTES)		RTR LOG SIZE (BYTES)	
							UNCOMP	COMP	UNCOMP	COMP
CurrentIC Timestamp Approximation										
2048	2	2	64	0	64	124059274	5037844	2695087	8620158	3810588
1024	4	2	64	0	64	124250426	4881436	2598105	8467525	3712048
512	8	2	64	0	64	135662982	5023820	2683411	8627407	3793923
256	16	2	64	0	64	133438309	4848428	2573621	8386933	3666331
128	32	2	64	0	64	152667463	5504488	2937440	9602809	4183424
64	64	2	64	0	64	126448569	4843044	2543487	8333268	3627502
32	128	2	64	0	64	133165816	5125364	2704354	8936374	3859570
16	256	2	64	0	64	122568614	4690484	2466450	8233513	3551780
8	512	2	64	0	64	121783598	4766096	2509704	8109389	3593430
4	1024	2	64	0	64	125178723	4862860	2548559	8340330	3643720
Set/LRU Timestamp Approximation										
2048	2	2	64	1	64	129006184	177404	145923	136006	86185
1024	4	2	64	1	64	117954793	135320	108346	105682	68286
512	8	2	64	1	64	109377172	111764	88573	90239	58710
256	16	2	64	1	64	126116628	125260	98304	100286	65246
128	32	2	64	1	64	124836639	120756	94455	98100	63576
64	64	2	64	1	64	148630520	147012	113367	118519	76927
32	128	2	64	1	64	120373168	124564	95524	99380	64400
16	256	2	64	1	64	124106365	125000	96005	98172	63384
8	512	2	64	1	64	119921574	124584	95115	96307	62688
4	1024	2	64	1	64	105189051	102200	78772	83593	54688

Table A.11 Varying the Associativity and the Approximation Method — SPECjbb

TIMESTAMP MEMORY CONFIGURATION						RESULTS				
SETS	WAYS	NO. OF TS	BITS	SET / LRU	SIZE (KB)	RUNTIME (CYCLES)	TR LOG SIZE (BYTES)		RTR LOG SIZE (BYTES)	
							UNCOMP	COMP	UNCOMP	COMP
CurrentIC Timestamp Approximation										
2048	2	2	64	0	64	180615595	7797192	4557876	13436041	6224055
1024	4	2	64	0	64	180618343	7602584	4420714	13088278	6041800
512	8	2	64	0	64	180605692	7494084	4341114	12883303	5942127
256	16	2	64	0	64	180628721	7446308	4305409	12804962	5899038
128	32	2	64	0	64	180577126	7402756	4274501	12729975	5862079
64	64	2	64	0	64	180618357	7395804	4267550	12724870	5856767
32	128	2	64	0	64	180603503	7395300	4267002	12717471	5855580
16	256	2	64	0	64	180554950	7391236	4264838	12713514	5853081
8	512	2	64	0	64	180595729	7390988	4263763	12713124	5851621
4	1024	2	64	0	64	180619174	7389624	4262401	12711284	5849881
Set/LRU Timestamp Approximation										
2048	2	2	64	1	64	180567800	160288	147375	107555	68794
1024	4	2	64	1	64	180567907	107432	97188	70494	46746
512	8	2	64	1	64	180562183	82840	72293	57941	39054
256	16	2	64	1	64	180607547	72012	61210	53734	36423
128	32	2	64	1	64	180563558	63700	52806	50468	34301
64	64	2	64	1	64	180615662	61444	50314	49957	33895
32	128	2	64	1	64	180621992	60496	49490	49534	33559
16	256	2	64	1	64	180618242	60412	49180	49682	33722
8	512	2	64	1	64	180598757	59740	48616	49272	33358
4	1024	2	64	1	64	180422645	57864	46898	47684	32365

Table A.12 Varying the Associativity and the Approximation Method — SPECjbb

TIMESTAMP MEMORY CONFIGURATION						RESULTS				
SETS	WAYS	No. OF TS	BITS	SET / LRU	SIZE (KB)	RUNTIME (CYCLES)	TR LOG SIZE (BYTES)		RTR LOG SIZE (BYTES)	
							UNCOMP	COMP	UNCOMP	COMP
CurrentIC Timestamp Approximation										
2048	2	2	64	0	64	180615595	7797192	4557876	13436041	6224055
1024	4	2	64	0	64	180618343	7602584	4420714	13088278	6041800
512	8	2	64	0	64	180605692	7494084	4341114	12883303	5942127
256	16	2	64	0	64	180628721	7446308	4305409	12804962	5899038
128	32	2	64	0	64	180577126	7402756	4274501	12729975	5862079
64	64	2	64	0	64	180618357	7395804	4267550	12724870	5856767
32	128	2	64	0	64	180603503	7395300	4267002	12717471	5855580
16	256	2	64	0	64	180554950	7391236	4264838	12713514	5853081
8	512	2	64	0	64	180595729	7390988	4263763	12713124	5851621
4	1024	2	64	0	64	180619174	7389624	4262401	12711284	5849881
Set/LRU Timestamp Approximation										
2048	2	2	64	1	64	180567800	160288	147375	107555	68794
1024	4	2	64	1	64	180567907	107432	97188	70494	46746
512	8	2	64	1	64	180562183	82840	72293	57941	39054
256	16	2	64	1	64	180607547	72012	61210	53734	36423
128	32	2	64	1	64	180563558	63700	52806	50468	34301
64	64	2	64	1	64	180615662	61444	50314	49957	33895
32	128	2	64	1	64	180621992	60496	49490	49534	33559
16	256	2	64	1	64	180618242	60412	49180	49682	33722
8	512	2	64	1	64	180598757	59740	48616	49272	33358
4	1024	2	64	1	64	180422645	57864	46898	47684	32365

Table A.13 Log Size of the Selected Configuration

TIMESTAMP MEMORY CONFIGURATION						RESULTS				
SETS	WAYS	No. OF TS	BITS	SET / LRU	SIZE (KB)	RUNTIME (CYCLES)	TR LOG SIZE (BYTES)		RTR LOG SIZE (BYTES)	
							UNCOMP	COMP	UNCOMP	COMP
Apache										
32	64	2	24	1	12	694987581	1157836	887201	866048	505342
OLTP										
32	64	2	24	1	12	702150698	876240	672551	601249	374830
SPECjbb										
32	64	2	24	1	12	358461366	193660	170105	120543	80289
Zeus										
32	64	2	24	1	12	355362792	516284	384511	392088	251992

Table A.14 Runtime and Aggregated Bandwidth with and without the Recorder

RUN #	APACHE		OLTP		SPECJBB		ZEUS	
	Cycles	B/W (GB/s)	Cycles	B/W (GB/s)	Cycles	B/W (GB/s)	Cycles	B/W (GB/s)
Without the Recorder								
1	681095497	13.40	736512056	12.82	357463063	11.92	355635650	12.81
2	672380574	13.40	727118907	12.89	357263583	11.92	356179284	12.72
3	653447128	13.39	703866144	12.90	357530071	11.92	354475659	12.79
4	634474047	13.39	743305494	12.82	357515515	11.92	355749692	12.79
5	682961360	13.40	697463596	12.85	357291743	11.92	354557106	12.83
6	680101833	13.35	758570334	12.87	357283740	11.92	354791494	12.79
7	692223970	13.43	764252864	12.84	357471237	11.92	354875638	12.80
8	671867057	13.37	768147650	12.82	357323188	11.92	355204709	12.80
9	672357591	13.42	730530882	12.85	357241233	11.92	355042838	12.80
10	673320415	13.37	749762545	12.78	357424787	11.92	354540486	12.83
11	681238659	13.37	693409942	12.84	357293828	11.92	356040636	12.77
12	642271830	13.35	733735540	12.82	357450627	11.92	355446129	12.80
13	688677556	13.39	732856147	12.89	357798224	11.92	354506900	12.82
14	686412448	13.36	762921763	12.91	357237294	11.92	354513379	12.81
15	665917611	13.37	752058387	12.81	357343453	11.92	355011136	12.80
16	701887974	13.38	726876204	12.84	357276002	11.92	355041743	12.84
17	681870948	13.38	735466630	12.88	357283399	11.92	355442791	12.81
18	677529437	13.39	705499311	12.82	357227056	11.92	355042224	12.82
19	653439647	13.38	733205040	12.86	357233957	11.92	354815641	12.80
20	675037282	13.38	733277954	12.83	357505719	11.92	355010125	12.76
With the Recorder								
1	699430602	14.63	727160868	14.18	358774871	13.08	354617200	14.20
2	696860785	14.61	724043937	14.16	358444706	13.09	356609838	14.15
3	673921710	14.60	695449744	14.21	358769855	13.08	356441930	14.17
4	696870691	14.61	758907056	14.20	358445628	13.09	355850685	14.15
5	671262980	14.62	719623392	14.19	358776162	13.08	355664100	14.16
6	687511648	14.64	696172832	14.22	358746672	13.08	355581423	14.19
7	652092810	14.60	725998768	14.21	358720923	13.08	355675387	14.17
8	672376141	14.61	735151562	14.18	363664878	12.91	356483125	14.14
9	678127096	14.62	749267696	14.28	358487849	13.09	356617047	14.13
10	681843712	14.62	696852932	14.21	361370851	13.00	358510774	14.07
11	698813815	14.61	764485080	14.21	358435277	13.08	355071871	14.16
12	692489967	14.60	716717815	14.22	358774637	13.08	358428468	14.12
13	666412470	14.59	704146849	14.20	359135034	13.08	355207676	14.17
14	696603800	14.59	698610953	14.22	358715201	13.08	354565554	14.18
15	667181821	14.60	721637710	14.18	358720403	13.08	355696550	14.16
16	643131824	14.58	702641568	14.21	358491378	13.09	356351380	14.14
17	697485042	14.61	717746866	14.19	358477937	13.09	354958473	14.21
18	700005748	14.61	726599479	14.24	358765143	13.08	355545056	14.17
19	645906221	14.59	732443898	14.26	358485630	13.08	355995526	14.18
20	676228945	14.58	757831211	14.29	358775324	13.08	354742124	14.15

Table A.15 RTR versus TR: the Log Size with Infiniteⁱ TSM

TIMESTAMP MEMORY CONFIGURATION						RESULTS				
SETS	WAYS	No. OF TS	BITS	SET / LRU	SIZE (GB)	RUNTIME (CYCLES)	TR LOG SIZE (BYTES)		RTR LOG SIZE (BYTES)	
							UNCOMP	COMP	UNCOMP	COMP
Apache										
16384	16384	2	64	1	4	671368243	867620	642143	784136	463631
OLTP										
16384	16384	2	64	1	4	723795619	687480	507860	559457	351485
SPECjbb										
16384	16384	2	64	1	4	358710388	136680	109820	111739	73205
Zeus										
16384	16384	2	64	1	4	356508397	474988	343180	392124	255047

i. We use a sufficiently large TSM (4GB) to simulate the infinite TSM, because the large TSMs incur no TSM eviction.

Table A.16 The Log Size of a Recorder with Current IC Timestamp Approximation

TIMESTAMP MEMORY CONFIGURATION						RESULTS				
SETS	WAYS	No. OF TS	BITS	SET / LRU	SIZE (KB)	RUNTIME (CYCLES)	TR LOG SIZE (BYTES)		RTR LOG SIZE (BYTES)	
							UNCOMP	COMP	UNCOMP	COMP
Apache										
16384	4	1	32	0	256	657983997	18146564	8813713	30551175	11972138
OLTP										
16384	4	1	32	0	256	744048552	12169972	6229132	18726168	7788300
SPECjbb										
16384	4	1	32	0	256	358775444	11718000	6593854	19850630	8843554
Zeus										
16384	4	1	32	0	256	355386148	14600276	5883239	23288976	8349381

Table A.17 Scalability of the Recorder (From 2-core to 16-core)

No. Cores	TIMESTAMP MEMORY CONFIGURATION						RESULTS				
	SETS	WAYS	No. OF TS	BITS	SET/ LRU	SIZE (KB)	RUNTIME ¹ (CYCLES)	TR LOG SIZE (BYTES)		RTR LOG SIZE (BYTES)	
								UNCOMP	COMP	UNCOMP	COMP
Apache											
2	32	64	2	24	1	12	2312734104	1330036	916062	1189510	668128
4	32	64	2	24	1	12	672309727	1136928	869913	852426	498887
8	32	64	2	24	1	12	726071014	2803508	2139547	2436896	1451618
16	32	64	2	24	1	12	199783618	1665640	1329470	1230394	756050
OLTP											
2	32	64	2	24	1	12	954536534	196872	154575	106724	66777
4	32	64	2	24	1	12	772450787	967700	740937	658086	409498
8	32	64	2	24	1	12	493785576	1372020	1090367	950916	592225
16	32	64	2	24	1	12	292852033	2618848	2100631	2218651	1446947
SPECjbb											
2	32	64	2	24	1	12	671685085	77596	64452	37999	26163
4	32	64	2	24	1	12	358439283	192100	168563	120045	80126
8	32	64	2	24	1	12	250544714	528752	472536	223689	146508
16	32	64	2	24	1	12	183819676	1806964	1546946	660938	364227
Zeus											
2	32	64	2	24	1	12	577554085	176304	127907	120583	78424
4	32	64	2	24	1	12	356465958	519824	387765	404467	260056
8	32	64	2	24	1	12	260688501	867600	665821	761906	494481
16	32	64	2	24	1	12	347578392	5585212	4184530	6670851	4133622

i. All simulations complete the same number of transactions, even with different number of cores.

