

DETECTING AND MITIGATING CONCURRENCY BUGS

A Dissertation

Presented to the Faculty of the Graduate School
of Cornell University

in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by

Ruirui Huang

May 2013

© 2013 Ruirui Huang
ALL RIGHTS RESERVED

DETECTING AND MITIGATING CONCURRENCY BUGS

Ruirui Huang, Ph.D.

Cornell University 2013

As computing hardware moves to multi-core systems, future software needs to be parallelized in order to benefit from increasing computing resources. However, writing a correct parallel program is notoriously difficult, partly because of non-determinism in concurrent program executions. Because thread executions can be interleaved in many ways, a parallel program may produce a non-deterministic outcome even for identical program inputs if threads are not properly synchronized. Such a non-deterministic behavior, if not intentional, is often referred to as a concurrency bug. In this research, a solution is presented for efficient concurrency bug detection and mitigation.

As data races are widely used as a way to identify potential concurrency bugs, this research presents an efficient hardware architecture, named RaceSMM, that enables run-time data race detection with high coverage (99%) and minimal performance overhead (4.8% on average). The proposed hardware mechanism is based on the happens-before vector clock algorithm, which is known for its accuracy yet considered to be expensive due to a large amount of meta-data. As the main optimization, the proposed architecture in this research decouples meta-data storage from regular caches so that expensive meta-data is only selectively stored for memory locations that are accessed by multiple threads within a relatively short period where most data races happen.

While data races can detect a broad range of concurrency bugs where conflicting memory accesses are not controlled at all, recent studies show that many

concurrency bugs are not detectable by data races. Hence, this research introduces a new heuristic for non-race concurrency bug detection, named order-sensitive critical sections, which extends the intuition in data races to capture non-race bugs in practice. The order-sensitive critical sections are defined as a pair of critical sections that can lead to non-deterministic shared memory state depending on the order in which they execute. This research presents a runtime algorithm, named OSCS, that uses the notion of order-sensitive critical sections to detect real-world concurrency bugs. Experiments show OSCS provides a good bug coverage (90%) for both non-race atomicity and ordering violations, with a small number of false positives. Additionally, OSCS can be supported in hardware efficiently while maintaining a high detection coverage (84%) and causing a negligible performance overhead.

Finally, to provide a solution to further mitigate concurrency bugs post detection, an efficient deterministic replay scheme for multithreaded programs is introduced based on a concept of commutative critical sections. The commutative critical sections are critical sections in a multithreaded program that can be executed in any order while producing a consistent program output. In this research, we propose a deterministic replay scheme, named CommuteReplay, which allows the commutative critical sections to execute without enforcing an explicit deterministic order between the commutative critical sections to allow a noticeably better replay performance (reducing more than 20% of the overall performance overhead on average).

BIOGRAPHICAL SKETCH

Ruirui Huang attended the University of Waterloo as an ECE undergraduate student from the year 2002 to 2007. Right after his graduation from the University of Waterloo, he attended the Cornell University as a MS/PhD graduate student in the ECE department. Through his years at Cornell, he worked with his advisor, Professor G. Edward Suh, through various topics in the computer architecture field, including hardware support for computing reliability, security, and availability domains.

This thesis is dedicated to my parents, my wife Wanjuan, and my son
Alexander (Xiaoyou).

ACKNOWLEDGEMENTS

In the fall of 2007, I started my doctoral journey as someone who just completed his bachelor degree. I was not yet married to my wife, and my lovely son was not yet born. In the following 6 years, I have become a husband, a father, and now a doctor with my PhD degree. It was a long and exciting journey, and probably one of the most challenging ones in my first 30 years of life.

In the past 6 years, I have shared the best and not-so-best moments of my graduate school journey with many people. It has been a great privilege to be a graduate student in the Computer System Laboratory (CSL), which is a part of the Department of Electrical and Computer Engineering at the Cornell University. The CSL members and the ECE members at Cornell will always remain dear to me.

I would like to express the deepest appreciation and gratitude to my advisor and committee chair, Professor G. Edward Suh. He has supported me throughout my graduate school career, and provided the vision, encouragement, help, support, and advise necessary for me to finish the doctoral program and complete my dissertation. More importantly, he has also provided me with guidance and insight in life, and helped me to overcome challenges both in and out of school. I also want to thank Ed for giving me the freedom to pursue research projects that I am truly passionate about, and for serving as a role model to me both as a member of academia and a member of society.

Special thanks to my committee members, Professor David H. Albonesi and Professor Huseyin Topaloglu, for their unconditional support and great guidance. Their advice and suggestions throughout the past several years have helped me to become a better student and researcher. I owe them my heartfelt appreciation.

Members of my research group also deserve my sincerest thanks. I simply could not complete all of my works without their awesome help and assistance. Special thanks to (in chronological order) Dan Deng, KK Yu, Erik Halberg, and Andrew Ferraiuolo, who I have the privilege to collaborate with. Also, special thanks to Dan Lo who has contributed greatly in numerous discussions which ultimately lead to one of the key ideas in my thesis. I would also like to thank the rest of the group members, especially Yao Wang, for their friendship and assistance throughout the years.

In addition, I would like to thank Dr. Major B. Bhadauria and Dr. Karan Singh, who I have met when I first started my PhD program at Cornell. As senior students, they graciously showed me the ins and outs of the PhD program, and encouraged me to finish the doctoral journey since the very beginning. Special thanks also go to my fellow Cornell graduate students Saugata Ghose, Robert Karmazin, Janani Mukundan, Jonathan Tse, and many more for their encouragement and friendship which make me enjoy my time at Cornell that much more. I also want to thank all my friends in Canada, US, and China. Their friendship and support have made this journey that much easier.

I wish to thank my parents, Ruihua Huang and Shuxiang Li. Their unconditional love and support have gotten me where I am today, and I truly love them. I would also like to thank my extended family members in China, for the support and love that they have provided all my life. My wife, Wanjuan Lin, whose love, support, and encouragement have allowed me to finish this journey. She simply has my heart, and I thank her for allowing me to keep her heart as well. My lovely son, Alexander (Xiaoyou), whose smile and calling me "Daddy" simply make my world full of sunshine each and every day.

To all my friends and family, I hope this work makes you proud.

TABLE OF CONTENTS

Biographical Sketch	iii
Dedication	iv
Acknowledgements	v
Table of Contents	vii
List of Tables	x
List of Figures	xi
1 Introduction	1
1.1 Assumptions	3
1.2 Data Race Detection	5
1.3 Non-Race Concurrency Bug Detection	7
1.4 Deterministic Replay Execution	10
1.5 Organization	12
2 Data Race Detector	13
2.1 Introduction	13
2.2 Data Race Detection Overview	14
2.2.1 Data Races	14
2.2.2 Baseline Race Detection Algorithm	16
2.2.3 Challenges for Efficient HW Support	19
2.3 HW-Assisted Race Detection	21
2.3.1 Selective Bookkeeping	21
2.3.2 Distributed Scalar Clocks	24
2.3.3 Architecture Support	26
2.3.4 AHB Optimizations	33
2.4 Evaluation	35
2.4.1 Evaluation Setup	35
2.4.2 Race Detection Capability	36
2.4.3 Performance Overhead	43
2.4.4 Comparison to Related Schemes	45
3 Non-Race Concurrency Bug Detector	48
3.1 Introduction	48
3.2 Order-Sensitive Critical Sections	49
3.2.1 Intuition in Bug Detection Through Data Races	49
3.2.2 Non-Determinism in Critical Sections	51
3.2.3 Detection Heuristic	55
3.2.4 Limitations	58
3.3 Detection Algorithm	60
3.3.1 Overview	60
3.3.2 Detailed Algorithm	63
3.3.3 Debug Information on Detection	67

3.3.4	Optimizations	67
3.4	Hardware Implementation	68
3.4.1	Checker Module	70
3.4.2	Access History Buffer	72
3.4.3	Vector Clocks	73
3.5	Evaluation	74
3.5.1	Evaluation Setup	74
3.5.2	Bug Detection Capability	77
3.5.3	False Positives	80
3.5.4	Software Performance Overhead	83
3.5.5	Memory Space Overhead	83
3.5.6	Hardware Evaluation	85
3.5.7	Comparison to Other Schemes	87
4	Efficient Deterministic Replay Execution Through Commutative Critical Sections	90
4.1	Introduction	90
4.2	Commutative Critical Section	91
4.2.1	Overview	91
4.2.2	Definition	94
4.2.3	Terminology	95
4.2.4	Commutative CS Checking Scheme	100
4.3	Deterministic Replay Scheme with Commutative CS	110
4.3.1	Overview	110
4.3.2	Identifying Commutative CS	112
4.3.3	Baseline Record and Replay Scheme	114
4.3.4	Record and Replay Using Commutative CS	117
4.3.5	Logging Optimization	123
4.3.6	External Determinism Guarantee	123
4.4	Evaluation	126
4.4.1	Evaluation Setup	127
4.4.2	Commutative CS Count	129
4.4.3	Log Size Study	131
4.4.4	Performance Impact	132
4.4.5	Limitations	138
5	Related Work	142
5.1	Concurrency Bug Detection	142
5.1.1	Data Race Detection	142
5.1.2	Hardware-Based Race Detection	144
5.1.3	Concurrency Bug Detection (Beyond Data Races)	146
5.2	Deterministic Replay and Execution	147
5.2.1	Deterministic Replay	147
5.2.2	Deterministic Execution	150

5.2.3	Deterministic Programming Languages	152
6	Conclusion	154
	Bibliography	156

LIST OF TABLES

2.1	Percentage of shared locations in memory within various access window sizes. (P) - PARSEC, (S) - SPLASH2.	23
2.2	Baseline architecture parameters.	36
2.3	Detection capabilities.	37
2.4	Race injection study results. 50 races were randomly injected into the benchmarks to measure the detection capability. (P) - PARSEC, (S) - SPLASH2.	38
2.5	Comparison of HW Assisted Data Race Detection Schemes. . . .	45
3.1	Baseline architecture parameters.	75
3.2	Non-race concurrency bugs tested. (A - Atomicity violation, O - Order violation, M - Multi-variable bug).	76
3.3	Bug injection study. Potential non-race bugs are injected to measure the detection coverage. (P) - PARSEC, (S) - SPLASH2, (A) - Atomicity violation, (O) - Order violation.	77
3.4	The number of false positives (redun-wr: redundant writes). . . .	80
3.5	SW performance overhead. The slowdowns are normalized to the execution times of Pin instrumentation.	82
3.6	AHB Size Study. (P) - PARSEC, (S) - SPLASH2, (A) - Atomicity violation, (O) - Order violation.	87
3.7	Comparisons to other non-race bug detection schemes. (AV - Atomicity Violation; OV - Ordering Violation; MV - Multivariable Violation)	88
4.1	The execution environments used in our evaluation.	129
4.2	Counts of the total and commutative critical sections. (P) - PARSEC, (S) - SPLASH2.	130
4.3	Log sizes and compression ratio in CommuteReplay. (P) - PARSEC, (S) - SPLASH2.	131
4.4	Pin and instrumentation-only slowdowns for 2core_4thread configuration. (P) - PARSEC, (S) - SPLASH2.	133
4.5	Pin and instrumentation-only slowdowns for 4core_4thread configuration. (P) - PARSEC, (S) - SPLASH2.	135

LIST OF FIGURES

1.1	A high level overview of a parallel programming development cycle, which consists of three stages (i.e. square blocks) and the associated approaches in dealing with concurrency bugs at each stage (i.e. ovals).	2
2.1	A data race bug in MySQL due to a missing critical section. The example is obtained from a previous study [42].	15
2.2	The vector clock meta-data required for baseline race detection algorithm (RaceVC). Each element in a vector clock records a time stamp (TS) for the associated thread.	16
2.3	RaceVC: Baseline data race detection algorithm.	17
2.4	Flow charts for operations done on memory accesses. The dark block is the additional step that we have added to selectively bookkeeping shared locations	22
2.5	RaceSMM uses only scalar variables (TS and TID) for each memory location per core.	25
2.6	Architecture overview for RaceSMM.	27
2.7	Checker: keeps current thread ID (TID), a thread vector clock (ThreadVClock), and detection/bookkeeping logic.	28
2.8	AHB: a history table that saves PrevReadTS, PrevWriteTS, and PrevWriteTID for the most recent read/write to shared memory locations.	30
2.9	The impact of caches and AHB sizes on the detection capability of RaceSMM. We injected 50 races to each configuration. We reduced L1, L2, L3 and AHB sizes to 1/2 and 1/4 of the baseline configurations (32KB, 256KB, 8MB, 1024-entries).	41
2.10	The impact of L2/L3 cache sizes on the detection capability of RaceSMM. We injected 50 races to each configuration. We reduced L2 and L3 cache sizes separately to 1/2 and 1/4 of their baseline configurations (256KB L2, 8MB L3).	42
2.11	Performance overhead normalized to native execution.	43
3.1	A data race example (from MySQL [42]).	50
3.2	Types of critical section pairs.	51
3.3	Examples for different types of critical section pairs.	52
3.4	An ordering violation bug example (KB3) [76].	57
3.5	A false positive example from Ocean.	58
3.6	OSCS operations on every memory access.	60
3.7	Meta-data required for our algorithm.	61
3.8	OSCS: detailed algorithm for operations on each synchronization operations.	62
3.9	OSCS: detailed algorithm for operations on each memory operations.	63

3.10	Architecture overview for OSCS.	69
3.11	Checker: keeps current thread ID (TID), a thread vector clock (ThreadVClk), the most recent critical section information that a thread is currently in (CSList) and detection/bookkeeping logic.	70
3.12	AHB: a history table that saves previous accesses information for two most recent read and write accesses to shared memory locations within critical sections	72
3.13	A false positive example from Radix.	80
3.14	A false positive example from Pbzip2.	81
3.15	Performance overhead normalized to native execution.	84
4.1	An abstract example from the Ocean (Splash2) benchmark in which a global max is computed from multiple local max values.	92
4.2	The critical section in Thread 1 is recorded to execute before the critical section in Thread 2. On a replay through a traditional deterministic replay scheme, the execution of the critical section in Thread 1 is stalled due to a different execution environment, which results in the overall execution stalling as the execution of the critical section in Thread 2 needs to wait for Thread 1's critical section to finish its execution.	93
4.3	The $\langle CS \rangle$ can be described as a function $f()$, which takes input set X and produces output set Y . To simplify our discussion, we categorize inputs and outputs into three categories and use input vectors $X_{LI}/X_{GSI}/X_{GIO}$ and output vectors $Y_{LI}/Y_{GSO}/Y_{GIO}$ to represent the input elements.	97
4.4	Two instances of $f()$ are concatenated together to form $F()$. Note that any intermediate output values of Y_{GIO,CS_1} from CS_1 in Thread 1 are subsequently used by CS_2 in Thread 2 as input values, and hence are not part of outputs from $F()$. Also note that only one set of X_{GSI} and one set of Y_{GSO} are shown for $F()$ since both CS_1 and CS_2 share the same set of X_{GSI} and Y_{GSO}	99
4.5	Two orders of execution between CS_1 and CS_2 . The difference here is the order of the variable values fed into $F()$ in each execution order. The output values can also be different between the two orders.	104
4.6	An example, $\langle CS_{ocean} \rangle$, from the Ocean (Splash2) benchmark. $\langle CS_{ocean} \rangle$ has a global max that is computed from multiple local max values.	106
4.7	Two $\langle CS_{ocean} \rangle$ instances from the Ocean benchmark, CS_{ocean1} and CS_{ocean2} , are shown with different execution orders. In each $\langle CS_{ocean} \rangle$ instance, a MAX operation is performed with the two inputs, and the output of the MAX operation is the only output of the $\langle CS_{ocean} \rangle$ instance.	107

4.8	An example, $\langle CS_{ws} \rangle$, from the Water-Spatial (Splash2) benchmark. $\langle CS_{ws} \rangle$ has a local thread ID that is assigned from a rolling global thread counter.	108
4.9	Two $\langle CS_{ws} \rangle$ instances from the Water-Spatial benchmark, CS_{ws1} and CS_{ws2} , are shown with different execution orders. In each $\langle CS_{ws} \rangle$ instance, the value of <i>global_thread_index</i> is first assigned to <i>ProcID</i> , and <i>global_thread_index</i> is then incremented by 1. Both <i>ProcID</i> and <i>global_thread_index</i> are outputs of a $\langle CS_{ws} \rangle$ instance.	109
4.10	Once a critical section is identified as commutative, wrapper functions of <i>lock_CCS()</i> and <i>unlock_CCS()</i> are used instead of the regular <i>lock()</i> and <i>unlock()</i> functions to indicate commutativity.	113
4.11	In cases where multiple commutative critical sections share a mutex object, a <i>Cnum</i> is also used as an identifier parameter to wrapper functions of <i>lock_CCS()</i> and <i>unlock_CCS()</i>	114
4.12	An example from Ocean, where there are two instances of a critical section, one in each thread. The clock of the mutex object is incremented on lock/unlock operations, and the clock value is recorded in a log file along with the thread ID (TID) of each mutex operation. The log file is read on a replay to enforce the recorded execution order.	115
4.13	The BaseReplay operations on a replay for each synchronization operation.	116
4.14	An example from Ocean, where there are two instances of a commutative critical section, one in each thread. In CommuteReplay, the clock value remains the same for the mutex operations in both threads since the mutex operations belong to the same commutative critical section. To record, a CommuteRecord entry is stored in the log file, which contains the clock value, and the number of mutex operations in each thread that share the same clock value (i.e. consecutive mutex operations that belong to the same commutative critical section).	118
4.15	The CommuteReplay operations on a replay for each mutex synchronization operation.	120
4.16	There are two commutative critical sections that use the same mutex object. Two instances of each commutative critical section are shown. In the BaseReplay scheme, the clock of mutex object <i>l</i> is updated on each mutex operation. In the CommuteReplay scheme, the clock value is only updated when the previous operation is from another critical section.	122
4.17	The execution time of BaseReplay and CommuteReplay in Configuration 1. The execution time is normalized to the instrumentation-only execution time. The BaseReplay's overhead is broken down to two main categories.	134

4.18	The execution time of BaseReplay and CommuteReplay in Configuration 2. The execution time is normalized to the instrumentation-only execution time. The BaseReplay's overhead is broken down to two main categories.	136
4.19	The execution time of BaseReplay and CommuteReplay in Configuration 3. The execution time is normalized to the instrumentation-only execution time. The BaseReplay's overhead is broken down to two main categories.	139

CHAPTER 1

INTRODUCTION

As computing hardware moves to multi-core and many-core systems, future software needs to be parallelized in order to benefit from increasing computing resources. However, writing a correct parallel program is notoriously difficult, partly because of non-determinism in concurrent program executions. Because thread executions can be interleaved in many ways, a parallel program may produce a non-deterministic outcome even for identical program inputs if threads are not properly synchronized. Such a non-deterministic behavior, if not intentional, is often referred to as a concurrency bug.

In general, there are three high level stages in a parallel programming development cycle as shown in Figure 1.1. Namely, a parallel program is firstly coded by its programmers (developers); then tested to ensure the correct operation of the program; finally the program is deployed in the production environment. Given that writing a completely bug-free parallel program is known to be notoriously difficult, we assume that a number of concurrency bugs would exist in the program after the coding stage.

A concurrency bug can be detected at either the testing or deployment stage. In theory, all concurrency bugs can be identified at the testing stage if all possible thread interleaving patterns can be tested for all possible inputs during testing. Unfortunately, such an exhaustive testing is infeasible in practice. Therefore, run-time detection of concurrency bugs is desirable during both the testing and deployment stages because a deployed program is highly likely to contain concurrency bugs that were not covered during the testing stage of the development cycle. Previous studies have also shown that there is a noticeable number

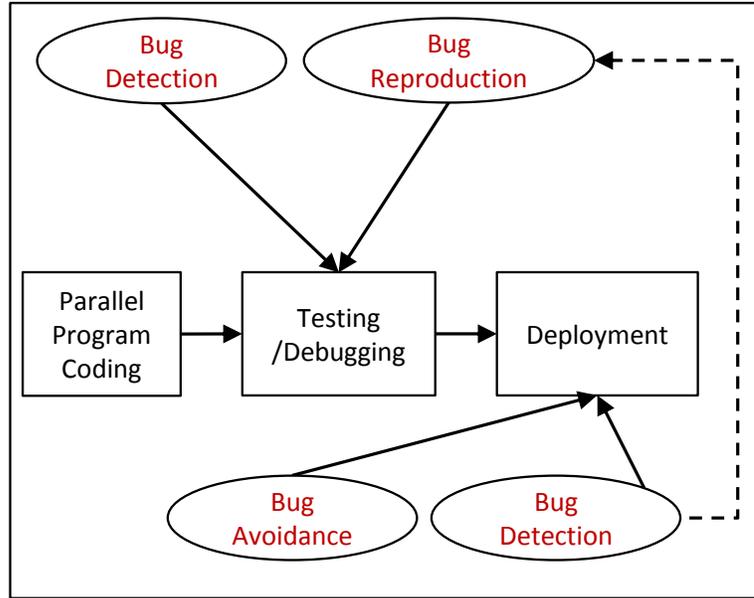


Figure 1.1: A high level overview of a parallel programming development cycle, which consists of three stages (i.e. square blocks) and the associated approaches in dealing with concurrency bugs at each stage (i.e. ovals).

of concurrency bugs in today’s multithreaded applications [42, 43, 61].

Once a concurrency bug is detected, it is desirable to have a deterministic way to reproduce the buggy interleaving for further debugging and testing. Deterministic replay has been shown to be able to simplify the debugging and testing process of multithreaded programs [7, 28, 30, 35] in both the testing and deployment stages. Furthermore, an alternative bug avoidance approach can be used after the program has been deployed with potential concurrency bugs. Deterministic replay can be combined with minor perturbations to avoid concurrency bugs in the program [63].

Overall, the goal of this thesis is to provide a solution for efficient concurrency bug detection and mitigation. There are three major components in the

thesis. First, this research investigates an efficient hardware architecture that addresses challenges in supporting accurate run-time data race detection. Second, we further extend the intuition behind traditional data races into critical sections, and introduce a new notion named *order-sensitive critical sections* to detect common non-race concurrency bugs. An efficient hardware architecture extension that supports the detection of order-sensitive critical sections is also discussed. Last, we investigate an efficient deterministic replay scheme, which allows either bug avoidance or a much simpler debugging and testing environment for multithreaded programs.

1.1 Assumptions

In this thesis, we make a couple of assumptions that are common across many concurrency bug detection and mitigation schemes, namely a shared memory programming model and identification of synchronization operations.

The thesis considers programs that are written under the shared memory programming model. Except for creating a thread and waiting for a termination, threads communicate through accesses to shared memory locations. In other words, a concurrency bug manifests as an unintended divergence in shared memory access orders in multiple program runs.

We also assume that synchronization operations that control interleaving patterns among threads can be explicitly identified. Programmers often rely on a library such as Pthreads to implement synchronization operations. In such cases, synchronization operations can be easily identified from the library calls. In this thesis, our prototype implementation detects synchronization in this

way. If a programmer uses custom synchronization primitives, our approach assumes that such primitives can be either marked explicitly by the programmer or automatically identified. For example, previous studies show that primitives such as spinlocks can be automatically detected [70, 73].

In general, parallel programs rely on two types of synchronization primitives to control thread interleaving. Primitives such as barriers and wait-signal pairs explicitly enforce a predetermined ordering among threads. In essence, the synchronization makes the thread interleaving deterministic. In this research, we refer to these primitives as *ordering* synchronization operations. On the other hand, primitives such as mutex and semaphores provide mutually exclusive code regions, often called *critical sections*, without enforcing a particular execution order. Thus, such critical sections can execute in a non-deterministic order in each run. We refer to such primitives as *mutex* synchronization operations.

In this thesis, we describe synchronization operations using *release* and *acquire* instead of individual synchronization operations. While there exist many types of synchronization primitives, they can fundamentally be considered as acquiring and releasing tokens. For example, mutual exclusion requires for each thread to acquire a token (lock) before entering a critical section and releases a token after the critical section. Similarly, barrier synchronization can be realized by having each thread release its token after reaching a barrier and wait for acquiring tokens from all other threads before proceeding. In the rest of this thesis, we refer to synchronization tokens as *synchronization objects*.

1.2 Data Race Detection

In this thesis, we propose to look into architectural optimizations that would enable us to use vector clocks [26, 71] efficiently, thus enabling accurate data race detection.

Data race detection is widely used as a way to identify potential concurrency bugs due to unsynchronized memory accesses. In general, a data race refers to conflicting (same location, at least one write) memory accesses from multiple threads that are not synchronized. Even though data races cannot detect all concurrency bugs, they provide a general condition to identify a broad range of bugs without application-specific knowledge. This research presents an efficient algorithm and hardware architecture that enable run-time data race detection with both high coverage and minimal performance overhead. The proposed technique enables parallel programs to be continuously monitored for races even in production systems, which are extremely sensitive to run-time overheads.

Because checking data races purely in software can introduce substantial runtime overheads, several hardware-assisted techniques have been proposed [18, 52, 55, 61, 62, 79]. However, existing hardware techniques either still show noticeable performance overheads or trade off detection coverage or scalability for lower overheads. For example, precise data race detection algorithms often depend on vector clocks [26, 71] to capture the happens-before relations [36] between memory accesses. While effective in accurately detecting data races, efficient and scalable hardware support for vector clocks is challenging because the size of vector clocks need to scale with the number of threads. For example,

an early hardware vector clock scheme [62] could only support a small number of threads. The state-of-the-art vector clock scheme [18] provides good scalability with a comprehensive detection coverage, but still reports a significant performance overhead at run-time (80% on average). Alternatively, a scheme based on scalar clocks was shown to have low overheads, but also a noticeably lower detection coverage of 77% [61].

This thesis proposes a data race detection scheme, named `RaceSMM`, with a set of optimizations to selectively manage meta-data, which enable accurate race detection based on the happens-before relations in hardware with minimal performance overheads and without noticeably sacrificing detection capability and scalability. The main optimization comes from the observation that only a small fraction of memory locations are accessed by multiple threads within a relatively short period where most data races happen. As a result, we found that storing meta-data only for those shared locations can greatly reduce the overheads with minimal impacts on coverage. While selectively maintaining vector clocks for statically shared memory locations has been proposed recently [18], we found that limiting the bookkeeping to locations that are *dynamically shared within a small window* is critical in achieving low overheads.

The proposed race detector only requires minor hardware changes with a small amount of state: a 13-KB buffer per core and a 1-bit tag per data cache block. Experimental results show that our optimization of selectively bookkeeping meta-data do not significantly impact the data race detection capability. In our experiments, the optimized detection scheme still detected all 13 real-world data race bugs that we tested, and detected more than 99% of hundreds of data races that we injected to multithreaded programs. Moreover, the experimental

results show that the proposed scheme has minimal impact on performance, with a 4.8% slowdown on average.

In essence, the proposed `RaceSMM` scheme represents a new trade-off point between performance and coverage that was not possible before, making a deployment of continuous race detection in production systems feasible.

1.3 Non-Race Concurrency Bug Detection

While data races can detect a broad range of concurrency bugs where conflicting memory accesses are not controlled at all, recent studies show that many concurrency bugs are not data races [42]. Programmers can remove data races by placing shared memory accesses within critical sections. However, because the critical sections can still execute in an arbitrary order, program outputs may still be non-deterministic even without data races.

In this research, we propose a new concurrency bug detection scheme that is designed to detect common non-race bugs by extending the intuition behind traditional data races into critical sections. Data races are commonly considered to be potential concurrency bugs because the conflicting accesses can execute in an arbitrary order, resulting in different memory state among multiple runs. Similarly, we observe that ordering of certain critical sections may change from run to run, introducing non-determinism in memory state. We call such critical sections as *order-sensitive critical sections*.

This notion of *order-sensitive critical sections* provides a new condition for detecting potential concurrency bugs. However, unlike traditional data races,

there are many cases when critical sections do not introduce non-determinism even if they are not explicitly ordered. Therefore, the question is whether one can effectively distinguish order-sensitive critical sections, which are likely to indicate a bug, from other legitimate uses of critical sections. We solve this challenge by studying how programmers typically use critical sections without introducing non-determinism, and developing a set of heuristic conditions to filter out such legitimate cases including explicit ordering, data parallel operations, redundant writes, and commutative operations.

This research presents a run-time algorithm, named OSCS , that uses the notion of order-sensitive critical sections to detect real-world concurrency bugs. The algorithm relies on vector clocks similar to the ones [26, 71] used for data race detection, to keep track of ordering restrictions, and adds extensions to filter out critical sections whose results do not depend on the execution order.

In practice, we found that this new approach could detect a broad range of non-race concurrency bugs with minimal false positives. In our experiments, the OSCS detection scheme flagged all 9 real-world non-race bugs that we could find and test, including atomicity violation, ordering violation, and multi-variable bugs. The scheme also detected most randomly injected atomicity and ordering violation bugs in PARSEC and SPLASH benchmarks. Moreover, experiments on Apache, Aget, Pbzip2, MySQL, Mozilla, SPLASH2, and PARSEC suggest that this new approach only introduces a small number of false positives.

As an additional optimization, this research also shows that the amount of meta-data can be significantly reduced by using scalar timestamps instead of vector clocks and keeping them only for shared memory locations. This opti-

mization has a minimal impact on detection capability as long as vector clocks are maintained for critical ordering constraints. This idea of only keeping scalar timestamps is similar to a recent software optimization (FastTrack) [27] and improves scalability. More importantly, the reduction in meta-data enables an efficient hardware implementation of the `OSCS` algorithm with a small amount of additional state: a 9-KB buffer per core and a 1-bit tag per data cache block. Experimental results show that the hardware `OSCS` implementation can still detect all 9 real-world non-race bugs tested and more than 84% of the injected non-race bugs. Moreover, the experimental results show that the hardware supported scheme has negligible impact on performance, with a 0.23% slowdown on average. We show that the `OSCS` algorithm can be supported in hardware with limited bookkeeping for very low performance overhead and still have a high detection coverage.

While there have been a number of efforts to detect non-race concurrency bugs, the proposed approach represents a unique contribution. In particular, the `OSCS` scheme can detect multiple types of bugs including atomicity violation, ordering violation, and multi-variable bugs rather than focusing on a single type. Also, `OSCS` does not require training or application-specific information as it relies on a bug condition that is common across programs. However, the proposed technique is still a heuristic and there is no guarantee on bug detection coverage. In this sense, the proposed scheme complements an existing body of work in non-race bug detection.

The main contributions of the `OSCS` scheme lies in introducing the notion of order-sensitive critical sections and the effective heuristics to detect them. We present a run-time `OSCS` detection algorithm that can be used to effectively

detect non-race concurrency bugs with low false positives and false negatives, and demonstrate that the OSCS algorithm can be implemented in hardware efficiently.

1.4 Deterministic Replay Execution

Finally, we propose an efficient deterministic replay scheme by allowing critical sections that are “commutative” to execute in parallel without enforcing a deterministic order on a replay. An efficient deterministic replay scheme helps to further mitigate the concurrency bugs post detection.

In order to maintain a simple testing and debugging environment, application developers desire deterministic behavior for multithreaded applications. Namely, a multithreaded application is deterministic if it would always produce the same output for a given input. While such deterministic behavior can be easily achieved in single thread sequential programs, it is not guaranteed in multithreaded programs. The nondeterministic nature of parallel programs makes debugging, testing, and maintaining multithreaded programs much more difficult than sequential programs.

Deterministic replay can be defined as the ability to replay multithreaded applications in a deterministic fashion based on a previous recording of all non-deterministic events. The deterministic replay capability has been shown to be able to simplify the debugging and testing process of multithreaded programs [7, 28, 30, 35]. Recently, researchers have proposed ways to replay multithreaded applications in a deterministic fashion [2, 39, 40, 59, 72]. However, current deterministic replay schemes incur a significant performance penalty.

For example, Respec, one of the state-of-the-art schemes, incurs more than a 55% performance overhead on average [40]. One of the main sources of the performance overhead for deterministic replay execution is stalling threads while waiting on synchronization operations to match the previously recorded order. Another related main source of the performance overhead is the logging operations needed for checking the previous recorded order of synchronization operations.

In this research, we present the concept of *commutative critical sections*. At a high level, the commutative critical sections can be described as critical sections in a multithreaded program that can be executed in any order while producing a consistent program output. We propose to allow the commutative critical sections to execute without enforcing an explicit deterministic order among them while still maintaining the determinism of always producing the same output from the same input on a replay. We note that this property of always producing deterministic output is called *external determinism* and it has been shown to be valuable in debugging and testing for the multithreaded programs [2, 40].

More specifically, we present a definition for commutative critical sections and an approach to systematically identify such commutative critical sections. By identifying critical sections as *commutative*, we can largely eliminate the overhead of enforcing a deterministic ordering between the execution of commutative critical sections on a replay. We also introduce a deterministic replay scheme, named `CommuteReplay`, which guarantees external determinism, to demonstrate how the concept of commutative critical sections can be used to reduce the performance overhead of deterministic replay in practice. Overall, evaluation results suggest that the proposed `CommuteReplay` scheme elimi-

nates most, if not all, of the replay overhead in stalling or logging for a deterministic ordering between critical sections. Hence, our proposed scheme allows for more efficient execution of deterministic replay.

1.5 Organization

The rest of the thesis is organized as follows. Chapter 2 presents an efficient run-time data race detection scheme, named `RaceSMM`, that provides high detect coverage with minimal overhead. Chapter 3 introduces a non-race concurrency bug detection scheme, named `OSCS`, that leverages the notion of order-sensitive critical sections. An efficient hardware architecture extension that supports the detection of non-race concurrency bugs through `OSCS` is also discussed. Chapter 4 proposes an efficient deterministic replay scheme, named `CommuterPlay`, that leverages the concept of commutative critical sections to reduce the overhead in replaying a recorded multithreaded program execution. Finally, Chapter 5 discusses related work, and Chapter 6 concludes the thesis.

CHAPTER 2

DATA RACE DETECTOR

2.1 Introduction

In this chapter, we present an efficient hardware architecture, named *RaceSMM*, that enables run-time data race detection with high coverage and near-zero performance overhead. The proposed hardware mechanism is based on the happens-before vector clock algorithm, which is known for its accuracy yet considered to be expensive due to a large amount of meta-data. The main optimization in our proposed scheme comes from the observation that only a small fraction of memory locations are accessed by multiple threads within a relatively short period where most data races happen. As a result, we found that storing meta-data only for those shared locations can greatly reduce the overheads with minimal impacts on coverage. Hence, to provide an efficient support for data race detection, we introduce an architectural optimization that decouples meta-data storage from regular caches so that expensive meta-data is only selectively stored for a small number of shared memory locations. While selectively maintaining vector clocks for statically shared memory locations has been proposed in a recent scheme [18], we found that limiting the bookkeeping to locations that are *dynamically shared within a small window* is critical in achieving low overheads. Furthermore, this architecture only adds a small amount of on-chip resources for race detection: a 13-KB buffer per core and a 1-bit tag per data cache block. Experiments show that the proposed scheme provides a high detection coverage (over 99%) with low performance overhead (4.8% on average). Our scheme detected all 13 real-world data race bugs that were tested as well as

hundreds of randomly injected data races.

The rest of the chapter is organized as follows. Section 2.2 presents a traditional (baseline) data race detection scheme based on vector clocks. Then, Section 2.3 describes how an accurate race detection can be efficiently realized in hardware with selective meta-data management in hardware along with hardware architectural optimizations. Section 2.4 evaluates the proposed race detection mechanism in terms of the effectiveness and overheads.

2.2 Data Race Detection Overview

While there are multiple approaches to detect data races, checking happens-before relations [36] is generally considered as the most accurate technique in identifying data races. In this section, we provide an overview of data race detection based on happens-before relations, including the assumptions and intuitions behind the approach. We also describe a state-of-the-art race detection algorithm that uses vector clocks [26, 71] to capture the happens-before relations. This algorithm will be used as a baseline in this chapter.

2.2.1 Data Races

Data race is defined as two conflicting memory accesses execute without any synchronization operation between them. Here, we define *conflicting accesses* as accesses from different threads to the same memory location, which include at least one write.

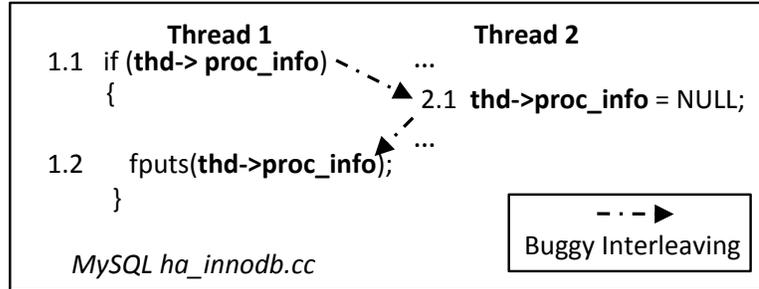


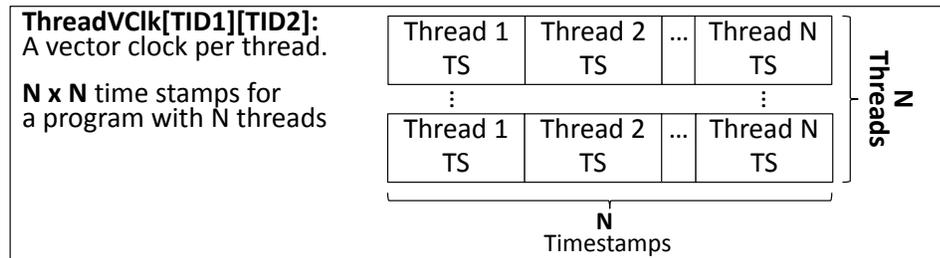
Figure 2.1: A data race bug in MySQL due to a missing critical section. The example is obtained from a previous study [42].

At run-time, data races can be accurately detected by checking if a pair of conflicting accesses are ordered by happens-before relations. In this chapter, we use the term *happens-before relation* to refer to an ordering between two events, in particular synchronization operations [36]. In other words, if a program is data race free, then every pair of conflicting accesses should be ordered by happens-before relations between synchronization operations [55].

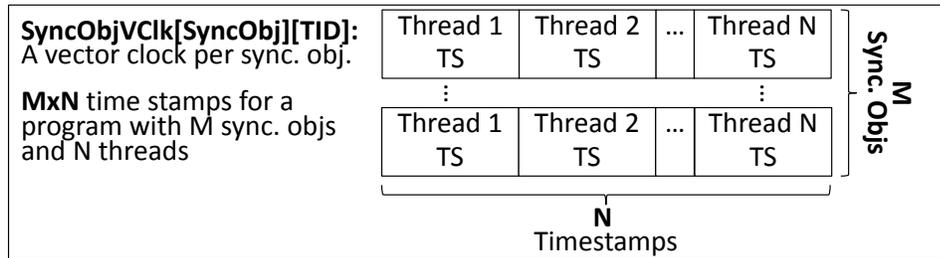
As an example, Figure 2.1 shows a data race in MySQL. In this example, none of the accesses to the shared pointer `thd->proc_info` is protected by synchronization. As a result, these accesses can execute in an arbitrary order, and potentially result in a fault if the pointer is set to be NULL by 2.1 between 1.1 and 1.2. In this example, there are two pairs of conflicting accesses, namely 1.1-2.1 and 1.2-2.1. Data races can be detected as both conflicting access pairs are not ordered by happens-before relations. To fix the bug, both 1.1 and 1.2 need to be protected by a mutex lock, and 2.1 needs to be protected by the same lock to ensure an atomic execution.

2.2.2 Baseline Race Detection Algorithm

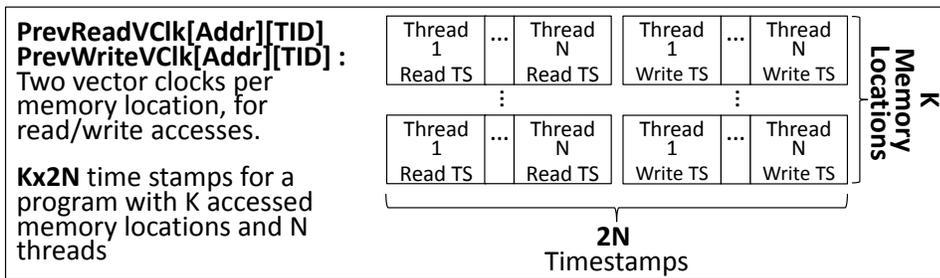
Here, we discuss a race detection algorithm based on vector clocks [26, 71]. We call this algorithm *RaceVC*, and use it as a baseline in the rest of this chapter. Overall, *RaceVC* first identifies conflicting memory accesses, and checks if the conflicting accesses are ordered by happens before relations using vector clocks.



(a) Thread Vector Clock



(b) Synchronization Object Vector Clock



(c) Memory Location Vector Clock

Figure 2.2: The vector clock meta-data required for baseline race detection algorithm (*RaceVC*). Each element in a vector clock records a time stamp (TS) for the associated thread.

As shown in Figure 2.2, there are several vector clocks needed for the traditional vector clock based *RaceVC* scheme. In the scheme, each thread is

RaceVC Algorithm

Functions:

`detect_conflict_access(TID, Addr, Type, ThreadVClk[TID][TID])`

1. Check the most recent write in each thread:

(a) For all valid threadID $i \neq \text{TID}$, call

`check_order(TID, i, PrevWriteVClk[Addr][i]).`

2. Check the most recent read in each thread:

(a) If $(\text{Type} == \text{Read})$, skip Step 2.

(b) For all valid threadID $i \neq \text{TID}$, call

`check_order(TID, i, PrevReadVClk[Addr][i]).`

3. Update the history for the memory location

(a) If $(\text{Type} == \text{Read})$,

`PrevReadVClk[Addr][TID] = ThreadVClk[TID][TID].`

(b) Otherwise,

`PrevWriteVClk[Addr][TID] = ThreadVClk[TID][TID].`

`check_order(TID, PrevTID, PrevTimeStamp)`

1. Check if the memory accesses can be re-orderd:

If `ThreadVClk[TID][PrevTID] ≤ PrevTimeStamp`,
report a data race.

`update_release(TID, SyncObj)`

1. `ThreadVClk[TID][TID]++;`

2. For each element in the vector clock, `SyncObjVClk[SyncObj][i] = MAX(ThreadVClk[TID][i], SyncObjVClk[SyncObj][i]).`

`update_acquire(TID, SyncObj)`

1. `ThreadVClk[TID][TID]++;`

2. For each element in the vector clock, `ThreadVClk[TID][i] = MAX(ThreadVClk[TID][i], SyncObjVClk[SyncObj][i]).`

Figure 2.3: RaceVC: Baseline data race detection algorithm.

uniquely identified by a thread ID (TID). Vector clocks are used to encode the access history and happens-before relations among conflicting memory accesses and synchronization operations.

For a parallel program with N threads, each thread maintains a vector clock with N elements, as shown in Figure 2.2(a). Conceptually, elements in `ThreadVClk` encode the ordering constraint (i.e. happens-before relations) between two threads. For example, `ThreadVClk[i][j]` indicates the earliest

that a memory access from Thread i can be executed in terms of Thread j 's local time without violating the happens-before relations of synchronization operations. $\text{ThreadVClk}[i][i]$ represents Thread i 's local clock that is incremented on each synchronization operation within that thread.

The algorithm also maintains a vector clock for each synchronization object as shown in Figure 2.2(b). SyncObjVClk is used to encode the ordering constraints from each synchronization operation. On a release operation, the SyncObjVClk is updated from the ThreadVClk of the thread that performs the release (take the later timestamp for each element). The SyncObjVClk represents the earliest that the following acquire operation can happen in each thread's local time. On an acquire operation, a ThreadVClk is updated with the corresponding SyncObjVClk .

The algorithm uses PrevReadVClk and PrevWriteVClk , as illustrated in Figure 2.2(c), to record timestamps for the most recent read and write accesses from each thread to each memory location. The access timestamps are recorded based on each thread's local clock (i.e. $\text{ThreadVClk}[i][i]$ for thread i). If the vector clocks are properly maintained, one can check if the current memory access from Thread i and a previous access from Thread j are ordered by happens-before relations by comparing Thread i 's vector clock value $\text{ThreadVClk}[i][j]$ with the timestamp of the previous access from Thread j . If the timestamp of the previous access is greater or equal to $\text{ThreadVClk}[i][j]$, a data race is detected.

Figure 2.3 shows the detailed RaceVC algorithm. On a memory access, the algorithm first detects conflicting memory accesses, i.e. read-after-write, write-after-read, and write-after-write from other remote threads to the same

memory location (`detect_conflict_access()`). Then, the algorithm determines if the conflicting access pair indicates a data race by checking whether the accesses are ordered by happens-before relations (`check_order()`). Lastly, the algorithm updates the associated memory location's vector clock based on each thread's local clock. On a synchronization *release* or *acquire* operation, `update_release()` or `update_acquire()` is called respectively to update vector clocks to encode the happens-before relations, and to increment the calling thread's local clock.

2.2.3 Challenges for Efficient HW Support

The main challenge in hardware support for data race detection lies in managing meta-data efficiently without significantly sacrificing scalability or detection coverage. A large amount of meta-data could result in large hardware structures or noticeable interference with regular program execution. On the other hand, reducing the amount of meta-data may limit the maximum number of threads that hardware can support or result in undetected races. In this context, traditional happens-before detection schemes based on vector clocks, such as *RaceVC*, are particularly challenging to support in hardware because they require vector clocks, whose size increases linearly with the number of threads, and for each memory location.

Specifically, as shown in Figure 2.2, *RaceVC* requires vector clocks for each thread, each synchronization object, and each memory location. The dominating portion of meta-data overhead comes from vector clocks for each memory locations. This is because the number of accessed memory locations is typically

significantly larger than the number of threads, and the number of synchronization objects in a multithreaded program. Quite often, the size of vector clocks for threads and synchronization objects is negligible when compared to the size of memory locations' vector clocks. Therefore, the main challenge here is to efficiently manage meta-data for memory locations.

A recent algorithm, named FastTrack [27], showed that storing access history for a single last write per address, instead of a vector of writes from each thread, is enough to provide a comprehensive data race detection coverage. However, we note that even with a single clock for each write, the size of meta-data still increases linearly with the number of memory locations as we still need vector clocks for read operations.

For the simplicity of presentation, we use vector clocks for both reads and writes in the RaceVC algorithm. A simple modification in RaceVC can be made to accommodate such optimization, namely we would only check the globally most recent write in Step 1 of the `detect_conflict_access()` function.

In order to manage the overheads, previous proposals for happens-before data race detection in hardware store meta-data at a coarse granularity, often one or two for each cache block [61, 62]. Also, these designs integrate the meta-data into data caches, introducing additional storage for each cache block. Unfortunately, such integrated designs trade off flexibility and coverage for lower overheads. Ideally, the hardware support should have low overheads while allowing fine-grained bookkeeping to maintain high detection coverage.

2.3 HW-Assisted Race Detection

In this section, we describe an optimized algorithm, named `RaceSMM`, and a hardware architecture which enables an efficient realization of accurate race detection based on happens-before relations. The proposed optimizations are based on the new insight that it is sufficient to maintain meta-data for a small number of recently shared memory locations. The design also decouples meta-data storage from caches and uses scalar meta-data instead of vector clocks to make the hardware scalable to a large number of threads. Overall, hardware-assisted `RaceSMM` provides efficient and high coverage race detection, enabling it to be applied to production systems.

2.3.1 Selective Bookkeeping

The main optimization in our architecture design comes from the insight that the bookkeeping for race detection is only necessary for “shared” memory locations. Here, we use the term “shared” to refer to locations with conflicting accesses within a certain time period. Such shared memory locations are a fraction of the entire memory space, especially for a relatively small window where most data races happen. Previous studies [42, 46] also made an observation that real-world race bugs typically manifest within a short window. Therefore, most real-world data races can be detected by maintaining meta-data for “shared” memory locations, which have conflicting accesses within a certain time period. Such shared memory locations are a fraction of the entire memory space, especially for a relatively small window where most data races happen.

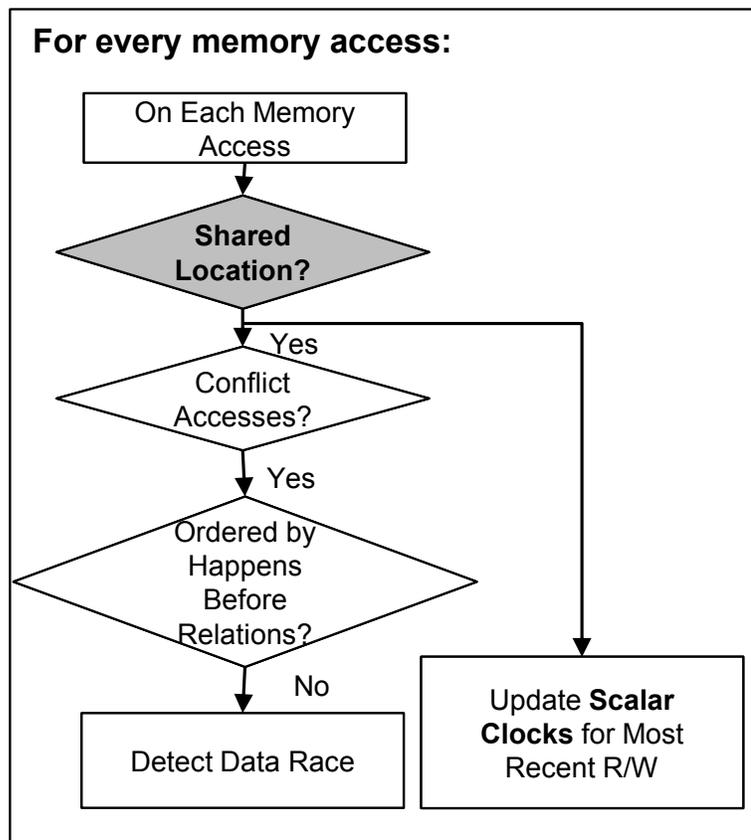


Figure 2.4: Flow charts for operations done on memory accesses. The dark block is the additional step that we have added to selectively bookkeeping shared locations

Table 2.1 shows the ratio of shared locations for various window sizes. Here, we define the window size using the total number of memory accesses (reads+writes) from all threads. The ratio is calculated by using the number of unique locations with conflicting accesses per window divided by the total number of unique locations accessed per window.

For PARSEC and SPLASH2 benchmarks, less than 0.6% of memory locations have conflicting accesses that happen within a window of 100,000 memory accesses. Recent studies [17, 31] also noted that a significant percentage of the

Table 2.1: Percentage of shared locations in memory within various access window sizes. (P) - PARSEC, (S) - SPLASH2.

	Window Size			
	1,000 Accesses	10,000 Accesses	100,000 Accesses	Entire Execution
Blackscholes(P)	0.000023%	0.00014%	0.00028%	40.27%
Bodytrack(P)	0.0030%	0.0059%	0.02%	69.79%
Fluidanimate(P)	0.0016%	0.014%	0.12%	26.70%
LU(S)	0.00021%	0.0030%	0.12%	99.28%
Ocean(S)	0.0020%	0.015%	0.11%	1.52%
Radix(S)	0.0023%	0.29%	0.60%	72.38%
Swaptions(P)	0.0012%	0.017%	0.22%	33.11%
Water-nsquare(S)	0.00013%	0.0040%	0.08%	42.53%
Water-spacial(S)	0.000087%	0.00079%	0.023%	57.43%
Geomean	0.00049%	0.0058%	0.05%	34.24%

accessed memory blocks are only accessed locally by one thread, even in parallel applications. Therefore, keeping meta-data such as timestamps and thread IDs (TIDs) for all memory locations is extremely wasteful. Instead, in our design, we decouple the detection of shared locations and the rest of bookkeeping so that most meta-data are stored only for memory locations with conflicting accesses.

The proposed design dynamically detects shared memory locations by augmenting each data cache block with a 1-bit tag, which indicates whether the block is shared or not, leveraging cache coherence events. The rest of the bookkeeping and detection are only performed for those locations that are marked as shared. Figure 2.4 illustrates this selective bookkeeping algorithm.

While RADISH [18] also discusses reducing meta-data by using a static anal-

ysis to identify memory locations that are never shared for the entire program execution, we found that limiting bookkeeping only to dynamically shared locations *within a time period* is critical to achieve low overheads. As shown in Table 2.1, shared locations over the entire program execution is a significant fraction of the memory space, several orders of magnitude higher than the one for a short period.

Based on the intuition that most data races happen within a relatively small window, we propose to selectively bookkeeping meta-data for a small fraction of memory space by dynamically detecting shared locations for on-chip data. As we will demonstrate in our evaluation section, such dynamic detection of shared locations and selective bookkeeping allows a much more efficient architecture while maintaining high detection coverage.

To support the dynamic detection of shared locations, we augment each block in data caches with a 1-bit tag, which indicates whether the block is shared or not. The rest of the bookkeeping and detection are decoupled from the detection of shared locations, and are only performed for those locations that are marked as shared.

2.3.2 Distributed Scalar Clocks

Even with the selective bookkeeping, the vector clocks to track recent reads and writes, whose size increases linearly with the number of threads, poses a significant challenge in building a scalable hardware-based race detector. While a previous work has shown that keeping information on only one write per location is sufficient to have a complete coverage [27], maintaining a vector clock

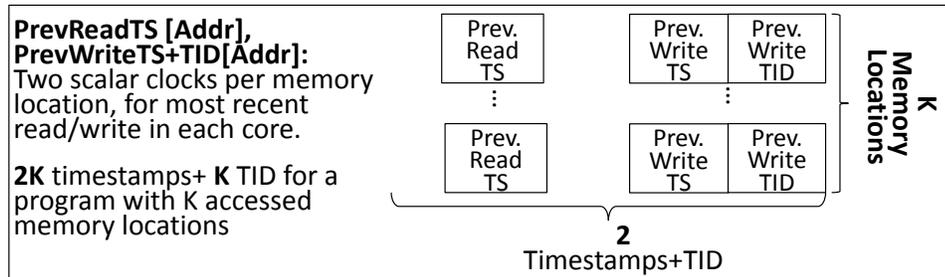


Figure 2.5: RaceSMM uses only scalar variables (TS and TID) for each memory location per core.

per location for reads still pose a scalability challenge.

To address the challenge, RaceSMM stores scalar timestamps for writes and distributed scalar timestamps for reads for each memory location while using vector clocks for synchronization objects. The insight is that the read vector clocks can be maintained distributively across multiple cores so that only one scalar timestamp for read is stored in each core’s meta-data buffer. Effectively, each core can keep a scalar timestamp for the most recent read access from the local thread and a scalar timestamp for the globally most recent write access for each memory location.

As shown in Figure 2.5, in each core, RaceSMM only keeps track of timestamps (PrevReadTS/PrevWriteTS) and the write TID (PrevWriteTID) of the most recent read and write for each memory location. As each core keeps timestamps for reads for a local thread, we do not need to keep the read TID. Compared to RaceVC, the meta-data for each memory location no longer grows linearly with the number of threads. It is now of a constant ratio to a program’s memory footprint. In the case that multiple threads are executed on a same core, all of the metadata shown in Figure 2.5 needs to be treated as a part of thread

state and flashed by an operating system on a context switch.

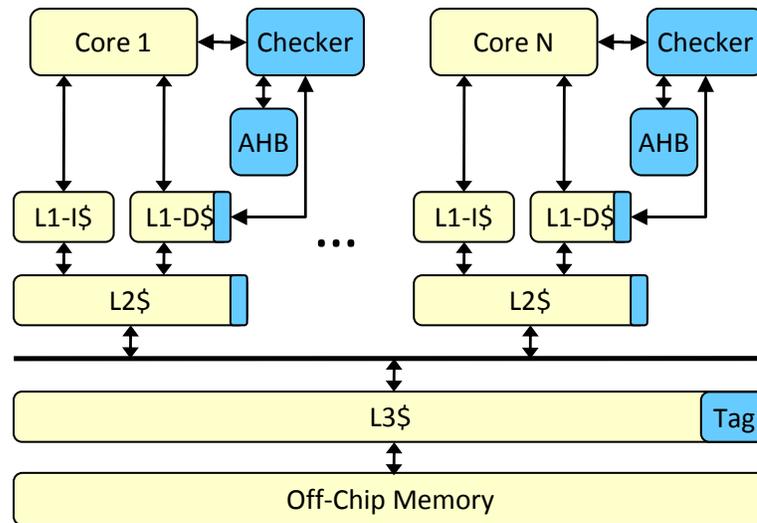
In order to accurately capture the happens-before relations of synchronization objects, and detect data races effectively, `RaceSMM` uses the same metadata structures and bookkeeping operations as `RaceVC` for `ThreadVClk` and `SyncObjVClk`.

2.3.3 Architecture Support

Figure 2.6(a) shows the high-level block diagram for the proposed architecture support for data race detection. In the figure, the blue (dark) blocks indicate the new hardware components needed to support `RaceSMM`. Figure 2.6(b) illustrates the high level operations of each new hardware component for `RaceSMM`, and they are discussed in detail here. The overall detection operations of our architecture closely follow the `RaceVC` algorithm, with the addition of using scalar timestamps in each core and selective bookkeeping for shared locations.

Extension for Shared Location Detection

In our architecture, each block in a data cache has a 1-bit tag, which indicates whether the block is shared. The shared bit is set when a cache coherence event indicates that multiple cores access the same cache block with at least one write. More specifically, the shared bit is set when there is a downgrade request that changes the cache block to either shared or invalid state. For example, in a MESI protocol, the shared bit is set on the following requests: $M \rightarrow S$, $M \rightarrow I$, $E \rightarrow S$, $E \rightarrow I$, $S \rightarrow I$, etc. This shared bit follows the data on-chip; a shared bit



(a) A block diagram for the overall architecture. Blue (dark) blocks are additional hardware support needed for RaceSMM.

Shared Location Detection: 1-bit shared tag for L1/2/3 cache block, set on L1/2 cache coherence downgrade event, propagate to L3.

Access History Buffer (AHB): Meta-data for most recent read/write to a shared location. Including PrevTIDs and PrevTSs.

Checker Module: Keeps active TID and ThreadVClk. On shared access from L1, records access to AHB, checks for data races. Bookkeeping ThreadVClk and SyncObjVClk on sync. operations.

Sync. Object Vector Clocks (not shown in the block diagram): SyncObjVClks are stored and accessed through existing memory hierarchy.

(b) Key components for the RaceSMM architecture

Figure 2.6: Architecture overview for RaceSMM.

is written back to a lower-level cache, and the bit is read with data on a cache miss. However, the shared bit is cleared when a cache block is evicted to or read from off-chip memory. Effectively, this mechanism detects memory blocks that are *shared* within a time window, while the block exists in multiple private

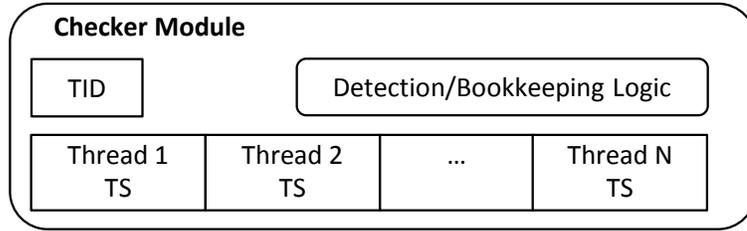


Figure 2.7: Checker: keeps current thread ID (TID), a thread vector clock (ThreadVClk), and detection/bookkeeping logic.

(L1/L2) caches, and keeps this history while the block is on-chip. While this design can only detect shared locations within a short period, our experimental results show that the on-chip bookkeeping is sufficient for virtually all concurrency bugs tested. We further study the impact of cache sizes on the detection capability in our evaluation section.

We believe that the 1-bit tag is sufficient under the assumption that each core runs one thread with infrequent context switches or thread migrations. If not, the 1-bit tag may not detect locations that are shared by multiple threads on one core or incorrectly identify a block as shared when a single thread moves from one core to another. However, note that the inaccuracy can only lead to false negatives, but not false positives. This is because detection can only be made on locations that are marked “shared”. To be more accurate, a thread ID can be added to the 1-bit tag in order to identify which thread each access comes from. The overhead would be still quite low as only one thread ID needs to be maintained per cache block.

Checker Module

In the proposed architecture, a checker module maintains per-thread state and performs most of the bookkeeping and checking operations at each core. As shown in Figure 2.7, for the active thread on the core, the checker keeps a thread ID (TID), and a thread vector clock (ThreadVClk).

On a memory access from the core, the checker module uses the shared bit in an L1 data cache to determine if the access is to a “shared” block. If so, the checker module records the access into the access history buffer (AHB), and examines whether there is a data race between the most recent read/write accesses and the current access.

As vector clocks for recent reads are distributed across each core’s AHB, our scheme requires read access history from remote AHBs to check data races on a local write. On a write, the local write and the local thread’s ThreadVClk is broadcasted to all remote checkers. Each remote checker then checks if the broadcasted write and the most recent read from its own thread are ordered by happens-before relations by comparing the broadcasted ThreadVClk with PrevReadTS from its own AHB (i.e. `check_order()`). Additionally, the PrevWriteTS is compared with ThreadVClk on both local read and write accesses. Since we only keep a single PrevWriteTS for the globally most recent write for a memory location in each AHB, the checker module compares ThreadVClk with PrevWriteTS locally to check if the local access and the globally most recent write access are ordered by happens-before relations.

The access to local AHB can be done in parallel to local L1 accesses, and the ThreadVClk is kept locally within the checker module. Hence, the overhead of

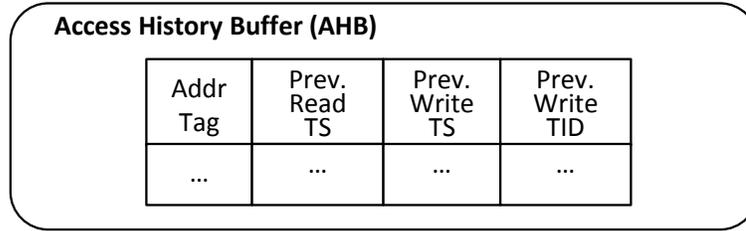


Figure 2.8: AHB: a history table that saves PrevReadTS, PrevWriteTS, and PrevWriteTID for the most recent read/write to shared memory locations.

our detection mechanism is kept minimal on a read access. On a write access, however, broadcasting the write access and `ThreadVClk` can incur overhead. Fortunately, the broadcasting is only needed on write accesses to shared locations. In all of the benchmark programs that we have tested, write accesses to shared locations only account for 0.5% (3% worst case) of all memory accesses. Hence, the broadcasting overhead on shared write accesses is minimal.

The checker module also coordinates with software layers through new instructions. The architecture provides two additional instructions to indicate a synchronization operation, one for *acquire* and the other for *release*. These instructions also convey the address of the vector clock for the corresponding synchronization object (`SyncObjVClk`). The vector clock for synchronization object is accessed and/or updated by the checker module on *acquire* and *release* operations through normal memory hierarchy.

Access History Buffer (AHB)

The access history buffer (AHB) records information on the most recent read and write to a shared location that can be used to detect conflicting accesses and

to check for data races. As shown in Figure 2.8, the AHB serves as a history table that saves `PrevReadTS`, `PrevWriteTS`, and `PrevWriteTID` for recently accessed shared locations. On a memory access to a shared location, the checker module records a thread ID (only for write accesses), a timestamp (TS), along with the memory address tag into the AHB.

The AHB is kept coherent by a cache coherence protocol similar to other on-chip caches. We note that only the access history of a most recent write needs to be kept coherent, and the access history of a most recent read is only updated and accessed locally. We implemented the AHB coherence protocol separate from the main data cache, however, we note that the AHB coherence operations can be piggybacked on existing data cache coherence protocol as well.

As the AHB has a limited capacity, it works like a cache and only keeps the history of recently accessed shared memory locations. However, there is no backup hierarchy for the AHB. If an entry is evicted from an AHB, the information is simply thrown away. A miss to the AHB creates a new entry. While this design implies that we cannot detect conflicting accesses with a long distance in between, our experimental results suggest that an AHB with 1024 entries are sufficient for virtually all races tested. Moreover, a miss to the AHB can only lead to potential false negatives, but not false positives as we would not make a detection on a miss. We will further examine the impact of AHB size on the detection capability in our evaluation section.

We note that the AHB can store an access history *per byte* because only accesses to shared memory locations are recorded. On the other hand, traditional designs that combine meta-data into the main cache often had to store information on a cache block granularity to keep overheads acceptable.

Vector Clocks

Our proposed architecture uses two types of vector clocks: `ThreadVClk` and `SyncObjVClk`. For the `ThreadVClk`, our architecture uses dedicated storage in each checker module for an active thread on the core. We found that the `ThreadVClk` needs to be close to the checker because it is used in each `check_order()` operation. The `ThreadVClk` needs to be treated as a part of thread state and managed by an operating system on a context switch. On the other hand, `SyncObjVClk` data are stored and accessed through the existing memory hierarchy. For each synchronization object, software allocates space for a vector clock in its memory space and passes the location using the instructions that indicate synchronization operations. Hence, on `update_release()` and `update_acquire()`, the `SyncObjVClk` is accessed and/or updated through the existing memory hierarchy. We note that our architecture needs to access `SyncObjVClk` only on synchronization operations, which happen infrequently. `SyncObjVClk` accesses have minimal impact on performance as we will demonstrate in our evaluation section.

Hardware counters have a limited number of bits. As a result, the clock that each thread uses to represent its local time may overflow after many synchronization events. Fortunately, our experiments show that synchronization operations are rather infrequent and the thread clocks only increment slowly. In fact, we did not see any overflow for PARSEC and SPLASH2 benchmarks with 16-bit counters. Given that overflows are infrequent, our architecture handles them in a relatively slow but straightforward fashion instead of adding complex hardware. Upon detecting an overflow in its local clock, a checker raises an exception to an operating system, which in turn interrupts other cores that

run other threads from the same program. Then, the operating system clears all clocks, and marks all AHB entries to be invalid in each core. In order to allow an operating system to clear vector clocks for synchronization objects, an application allocates them in separate pages that are known to the operating system.

2.3.4 AHB Optimizations

There are a couple of simple yet effective architectural optimizations for AHB in our design. The overall goal of these optimizations is to improve our detection coverage and/or further reduce the overheads.

AHB with Flexible Granularity

We note that the AHB can store an access history *per byte* to avoid false positive data race detection due to false sharing. However, typically only a very small fraction of memory accesses are done on a byte granularity, while most of the memory accesses are to variables sized at a word granularity. Therefore, we implemented our AHB with flexible granularity. For each AHB entry, there is a single-bit flag which indicates if granularity is per byte or per word. An additional 4-bits are also appended to each entry to mark which bytes in a word are valid for the AHB entry. For example, if only a byte is accessed within a word, the single-bit flag of the AHB entry would indicate a byte granularity, and only 1-bit out of the appended 4-bits would be marked valid for the AHB entry. In this case, the AHB entry would only contain the access history information for the byte that is accessed. Overall, AHB with flexible granularity improves our

detection coverage by keeping more access histories if most variables used by the program are accessed at the word granularity, while still maintaining the per-byte detection granularity capability to avoid false positives.

AHB Prefetching

At the time that a cache line is loaded into a private cache, we can prefetch all of the missing AHB entries associated with the cache line from remote AHBs. If no data race is detected by checking all of the AHB entries prefetched when a cache line is loaded, then no additional check is needed unless the cache state is downgraded. For example, if a cache line is loaded and remains in S/E state, any remote writes would have caused it to be invalidated. Otherwise, no remote write to the cache line has happened and thus no additional data race could have occurred. Similarly, if a cache line is loaded and remains in M state, no additional remote accesses would have happened unless the M state is downgraded. In all of the programs that we encountered, cache lines are often accessed multiple times before being evicted from the private cache. Hence, by prefetching and checking for data races preemptively, we can reduce the performance overhead by avoiding any additional detection and AHB coherence operations.

2.4 Evaluation

2.4.1 Evaluation Setup

Our infrastructure is built on the Pin binary instrumentation framework [47]. To evaluate the detection capabilities, our tool implements both the baseline `RaceVC` and `RaceSMM` algorithms by intercepting memory accesses and `Pthread` calls. To evaluate the performance overheads in supporting `RaceSMM`, we implemented a typical memory hierarchy with bookkeeping structures in a Pin tool, and also added a timing model with a processing core that runs 1 instruction per cycle, L1/L2/L3 caches, and a memory interface.

Table 2.2 summarizes the baseline architecture parameters. We model a multi-core processor with 4 cores, 64KB L1 and 256KB L2 private caches per core, and an 8MB shared L3 cache. We also model the MESI cache coherence protocol.

For the additional bookkeeping, we model the AHB with an 8-bit thread ID and a 16-bit timestamp per read/write access. Each entry records information for one read and one write. There are 1024 entries in each AHB in the baseline architecture, which results in a 13KB AHB buffer per core.

To evaluate the detection capability, we use two types of benchmarks, namely kernel bugs (KB) and real programs. The kernel bugs are created based on real-world application race bugs (MySQL, Apache, and Mozilla) from previous studies [42, 43, 76]. The kernel bugs use 2 threads to reproduce the original bugs. We also use three data race bugs from two large real-world server applications (Apache and MySQL). We use 30 threads for Apache and 10 threads for

Table 2.2: Baseline architecture parameters.

Component	Parameters
Core	4 2-GHz in-order single-issue cores
Caches	L1 I/D (private, inclusive): 32KB/32KB 4-ways 3 cycles Latency L2 (private): 256KB, 4-ways 15 cycles latency L3 (shared): 8MB, 8-ways 40 cycles latency
Coherence Protocol	MESI
DRAM	4GB 50ns Latency
Meta-data	8-bit thread IDs, 16-bit clocks
AHB	1024-entries, 8-way, 13KB, 3 cycles latency

MySQL. For a further study on coverage, we also perform random race injections to benchmarks from SPLASH2 [15] and PARSEC [8]. The race injection is performed by randomly selecting a critical section and ignoring the critical section for the entire program execution. The SPLASH2 and PARSEC benchmarks are run using 4 threads with the default input size for SPLASH2 benchmarks and `simmedium` input size for PARSEC.

2.4.2 Race Detection Capability

We compare the race detection capabilities of two schemes: `RaceVC` and `RaceSMM`. `RaceVC` is implemented only in software, as we only need to use its detection capability as a baseline. The `RaceVC` algorithm is really not a suitable candidate for a hardware implementation due to the high overhead and scalability concerns. `RaceSMM` is implemented in both software (`RaceSMM-SW`)

Table 2.3: Detection capabilities.

	RaceVC SW	RaceSMM SW	RaceSMM HW
Apache	Yes	Yes	Yes
MySQL-1	Yes	Yes	Yes
MySQL-2	Yes	Yes	Yes
KB1(MySQL)	Yes	Yes	Yes
KB2(MySQL)	Yes	Yes	Yes
KB3(MySQL)	Yes	Yes	Yes
KB4(Apache)	Yes	Yes	Yes
KB5(Mozilla)	Yes	Yes	Yes
KB6(Mozilla)	Yes	Yes	Yes
KB7(Mozilla)	Yes	Yes	Yes
KB8(Mozilla)	Yes	Yes	Yes
KB9(Mozilla)	Yes	Yes	Yes
KB10(Mozilla)	Yes	Yes	Yes

and hardware (RaceSMM-HW). RaceSMM-SW keeps a vector clock for the most recent reads and only a scalar timestamp for the most recent write for each byte in memory, whereas RaceSMM-HW relies on caches and AHB, both with limited capacities, for bookkeeping. We note that as suggested in previous study [27] and concurred in our evaluation study, RaceSMM-SW always have the same detection coverage as the RaceVC. Hence, for the rest of this section, we only present results from RaceVC and RaceSMM-HW. We also use RaceSMM to denote RaceSMM-HW implementation in the rest of the chapter.

Table 2.4: Race injection study results. 50 races were randomly injected into the benchmarks to measure the detection capability. (P) - PARSEC, (S) - SPLASH2.

	RaceVC	RaceSMM (SW/HW)
Blackscholes (P)	50	50
Bodytrack (P)	50	50
Fluidanimate (P)	50	49
LU (S)	50	50
Ocean (S)	50	49
Radix (S)	50	50
Swaptions (P)	50	50
Water-Nsquared (S)	50	49
Water-Spatial (S)	50	50
Total	450/450	447/450

Detection Coverage

Table 2.3 shows detection results for real-world data race bugs. The results show that both `RaceVC` and `RaceSMM` detect all races, indicating our hardware approach do not significantly affect detection coverage.

In theory, `RaceVC` may be able to detect data races that `RaceSMM` cannot. This is because the hardware implementation of `RaceSMM` relies on caches to detect shared locations and the AHB to keep information of recent accesses. Because the caches and the AHB have limited capacities, relevant information may be evicted and lost, causing potential false negatives. As shown in Table 2.4, `RaceSMM` did not detect 3 data races in our race injection study whereas `RaceVC` detected all.

In theory, `RaceVC` may be able to detect data races that `RaceSMM` cannot.

This is because the hardware implementation of `RaceSMM` relies on caches to detect shared locations and the AHB to keep information of recent accesses. Because the caches and the AHB have limited capacities, relevant information may be evicted and lost, causing potential false negatives. Table 2.4 shows the detection coverage of `RaceVC` and `RaceSMM` for the injected races. Here, `RaceSMM` did not detect 3 data races in our race injection study whereas `RaceVC` detected all.

Overall, results for both real-world race bugs in Table 2.3 and injected races in Table 2.4 suggest that `RaceSMM` have a comparable detection coverage to `RaceVC`. `RaceSMM` detected more than 99% of the injected races. Previous studies [42, 46] also observed that real-world race bugs typically manifest within a short window. This explains why the hardware implementation (i.e. with limited bookkeeping) shows comparable detection coverage to the software implementation.

`RaceSMM` signals a race detection when two conflicting (same location, at least one write) memory accesses are not ordered by happens-before relations. This is precisely the definition of a data race and hence there is no false positive in `RaceSMM`. While it is possible to have false negatives due to limited bookkeeping, `RaceSMM` would never flag a pair of conflicting accesses when a data race does not exist. In our study, both the software and hardware implementations of `RaceSMM` report no false positives for Apache, MySQL, SPLASH2, and PARSEC applications.

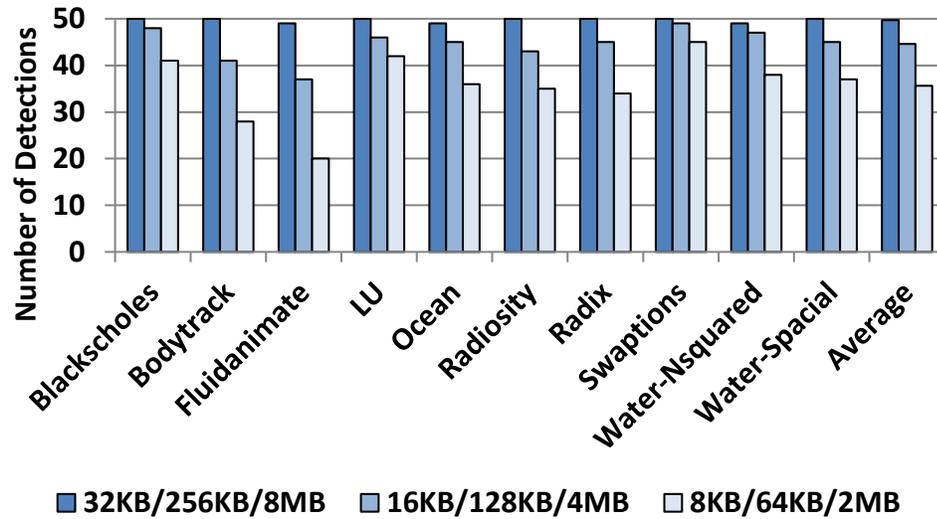
We have also studied the impact of context switches on the detection coverage of our proposed scheme. Each core’s AHB was periodically cleared to mimic the effect of a context switch in a core. Overall, the impact of context switches

on detection coverage is negligible. For example, if a context switch occurs every 1ms on each core, our proposed architecture would still detect virtually all data races tested (coverage remains at 99%). In our experiments, the 1024-entry AHBs are filled within 0.01ms on average and 0.04ms in the worst case, which is a fraction of a typical time quanta. Also, a data race is often detected many times over an execution; so missing one dynamic instance does not necessarily lead to a lower static coverage.

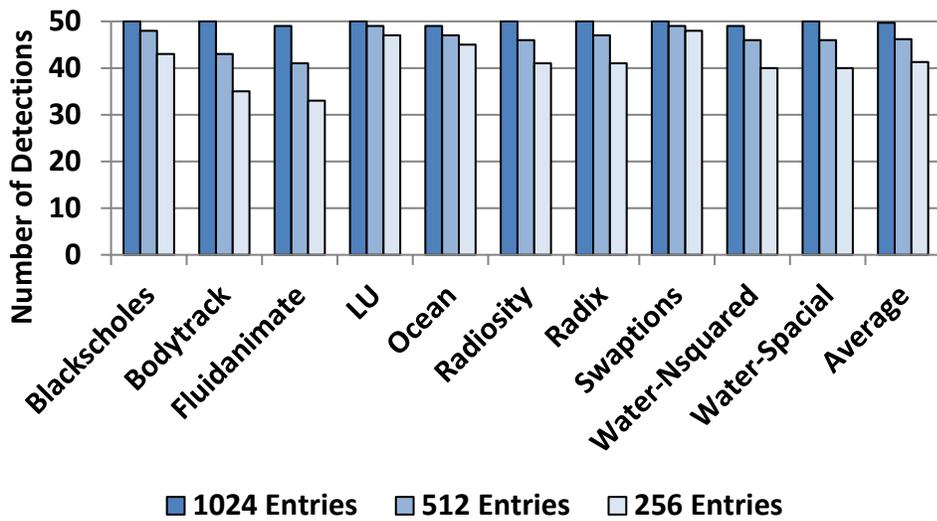
Cache and AHB Size Analysis

The hardware-based `RaceSMM` scheme relies on caches and the AHB for book-keeping. Therefore, the cache and AHB sizes directly affect the detection capability. The race injection study in Figure 2.9(a) shows the impact of reducing cache sizes on the detection coverage. Here, the L1, L2, and L3 caches are reduced to 1/2 and 1/4 of the baseline while keeping the AHB at the baseline size. As expected, the detection rate decreases as the cache sizes decrease. Similarly, Figure 2.9(b) shows the impact of reducing the AHB size. The coverage decreases as the AHB size decreases because smaller AHBs can only keep history for less memory locations. The exact impact of reduced cache and AHB sizes, however, depends on application characteristics. For example, memory intensive benchmarks such as `Fluidanimate` are more sensitive than others. Overall, the experiments indicate that our scheme needs a private (L2) cache of 128-KB, a last level cache (L3) of 4-MB, and an AHB with 512 entries in order to provide good coverage (around 90%).

We also varied individual L2/L3 cache sizes and evaluated their effects on detection capability. We note that L1 cache is implemented to be inclusive, and

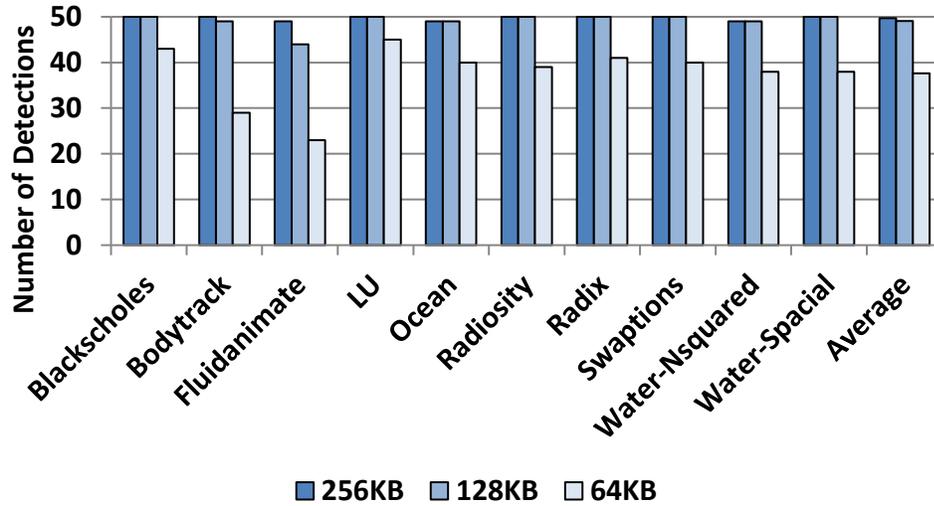


(a) Caches size study.

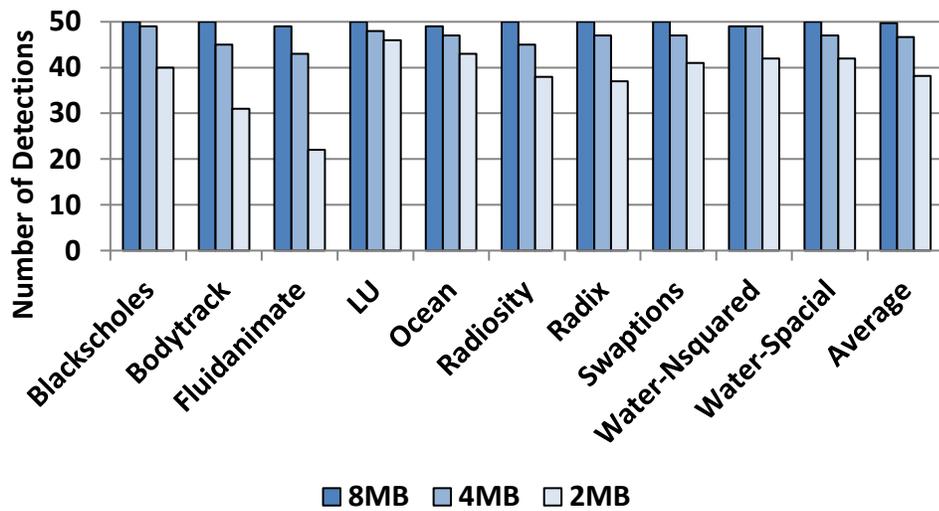


(b) AHB size study.

Figure 2.9: The impact of caches and AHB sizes on the detection capability of RaceSMM. We injected 50 races to each configuration. We reduced L1, L2, L3 and AHB sizes to 1/2 and 1/4 of the baseline configurations (32KB, 256KB, 8MB, 1024-entries).



(a) L2 cache size study.



(b) L3 cache size study.

Figure 2.10: The impact of L2/L3 cache sizes on the detection capability of RaceSMM. We injected 50 races to each configuration. We reduced L2 and L3 cache sizes separately to 1/2 and 1/4 of their baseline configurations (256KB L2, 8MB L3).

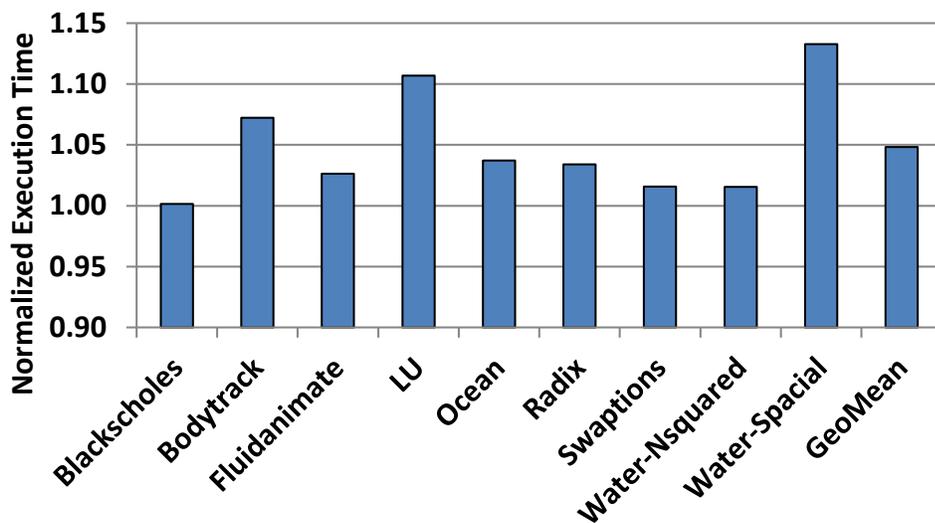


Figure 2.11: Performance overhead normalized to native execution.

hence its size does not impact race detection capability. Figure 2.10 shows the detection coverage of `RaceSMM` when the L2 and L3 cache sizes are reduced separately. Overall, we found that both L2 and L3 cache sizes have comparable impacts on detection capability, though for different underlying reasons. As we set the shared bits in the L2 cache on cache coherence downgrade requests, the L2 cache size impacts the probability of a shared location being detected. On the other hand, the L3 cache stores the shared bits while the shared cache blocks remain on-chip, and its size has a direct impact on how long a shared location's shared bit remains set.

2.4.3 Performance Overhead

Currently, the software implementation of `RaceSMM` incurs a 10-20X performance overhead compared to the plain `Pin` execution of the programs. The overhead excludes the standard overhead from `Pin`, which has 12X overhead

on average. The current software implementations are not optimized, and we expect the overheads to be lower with further optimization.

Figure 2.11 shows the normalized execution time for `RaceSMM`. The performance overheads are low at 4.8% on average. In the worst case, the overhead is 12.5% for `Water-Spatial`. Because the architecture mostly uses dedicated on-chip structures such as 1-bit cache tags and the AHB for bookkeeping, the only major source of performance overhead comes from communications between AHBs for keeping write history coherent.

Additionally, the dynamic power consumption of AHB based on CACTI [50] is estimated to be 56mW on average. We note that the current communication overhead is somewhat conservative as we modeled the communication network bus at only 1 GHz. Additionally we have modeled 32-bit bus width instead of the more aggressive setups. We believe that the overhead will be significantly lower with a further optimized communication network implementation.

Another possible source of overhead comes from accessing vector clocks for synchronization objects through the normal memory hierarchy. However, the number of vector clock accesses is negligible when compared to the number of regular data accesses. The vector clocks only introduce 0.08% additional accesses for `Fluidanimate` in the worst case, and only 0.003% on average. Hence, the overall performance impact due to additional vector clock accesses is negligible. The average L1 cache miss rate only increases 0.03% compared to the baseline.

Counter overflows may also introduce performance overheads by requiring timestamps and vector clocks to reset. However, we have never encountered

Table 2.5: Comparison of HW Assisted Data Race Detection Schemes.

	ReEnact [62]	CORD [61]	RADISH [18]	RaceSMM
Scalability	No	Yes	Yes	Yes
Detection Coverage	High-Very High	77%	100% (Complete)	Over 99%
False Positive	Yes	No	No	No
Hardware Overhead	High	High	Low	Low
Average Performance Overhead	5.8%	0.4%	80%	4.8%
Worst Case Performance Overhead	14.7%	3%	200%	12.5%

any overflow during experiments. In the worst case in `Fluidanimate`, we observed the maximum timestamp value of 47,832 after 287,748,471 memory accesses when the benchmark finishes. In all other benchmarks, the timestamp values never exceeded 10,000 after the entire execution. Hence, we believe that the possible performance overhead introduced by timestamp overflows is negligible.

2.4.4 Comparison to Related Schemes

The proposed `RaceSMM` scheme is most closely related to other hardware supported data race detection techniques based on happens-before relations. As shown in Table 2.5, we compare the characteristics of three other hardware assisted data race detection schemes with ours.

`ReEnact` [62] provides hardware support for logical vector clocks for cache lines. Due to its high hardware complexity and the use of vector clocks for cache lines, `ReEnact` suffers from noticeable performance overhead, and poor

scalability for more than a few threads. While ReEnact provides high detection coverage (note, no quantitative detection coverage is available from [62]), it also suffers from false positives due to false sharing of vector clocks within a cache line. The hardware overhead for ReEnact is also high as it requires specialized registers for vector clock storage in each cache, and the register size increases linearly as the number of threads grows.

CORD [61] avoids the overheads and poor scalability issue of vector clocks by keeping four scalar timestamps per cache line, at the expense of lower detection coverage (77%). CORD has a high hardware overhead as it requires on-chip state equal to 19% of cache capacity. In comparison, RaceSMM only requires 13-KB on-chip buffer space with 1-bit tag per cache line. Overall, CORD provides a scalable data race detection scheme with negligible performance overhead and no false positive at the cost of lower detection coverage and high hardware overhead.

RADISH [18] proposes a hardware and software hybrid race detection scheme that uses a vector clock based race detection approach similar to Fast-Track [27]. While it provides good scalability and a comprehensive detection coverage, it still incurs a significant performance overhead at run-time compare with the proposed scheme.

RaceSMM shows a 4.8% average slowdown while RADISH reports an 80% average slowdown in execution. In particular, two benchmarks were common used to evaluate RaceSMM and RADISH. The performance overheads for `Fluidanimate` are 3% in RaceSMM vs. 72% in RADISH. The overheads for `Blacksholes` are 0.2% in RaceSMM vs. 5% in RADISH.

Comparing with RADISH, the performance improvement in RaceSMM comes from that it only maintains meta-data for *dynamically shared locations within a small window* compared to statically shared locations in RADISH. Also, RaceSMM only stores meta-data in AHBs whereas RADISH keeps them as data in caches and memory, requiring extra transfers in a memory hierarchy.

Overall, RaceSMM provides a scalable scheme with a high detection coverage with no false positives, and requires low hardware overhead through a separate on-chip only buffer (AHB). RaceSMM is the only scalable scheme that provides high detection coverage, low performance overhead, while incurring no false positive of the four schemes compared.

CHAPTER 3

NON-RACE CONCURRENCY BUG DETECTOR

3.1 Introduction

In this chapter, we introduce a new heuristic condition for non-race concurrency bugs, named order-sensitive critical sections, and propose a run-time bug detection scheme based on the condition. The order-sensitive critical sections are defined as a pair of critical sections that can lead to non-deterministic shared memory state depending on the order in which they execute. Here, a run-time non-race bug detection algorithm, named `OSCS`, that detects the order-sensitive critical sections is presented. Experiments show that the proposed scheme provides a very good bug coverage for both non-race atomicity and ordering violations, with a small number of false positives. For example, the scheme, when implemented in software, detected all 9 real-world non-race bugs that were tested as well as over 90% of the injected non-race bugs in our experiments.

Additionally, the proposed non-race bug detector can be supported in hardware while only requiring minor hardware changes with a small amount of state - a 9-KB buffer per core and a 1-bit tag per data cache block. Experiment results show that the hardware-supported `OSCS` scheme can still detect all 9 real-world bugs that were tested, as well as more than 84% of the injected non-race bugs in our experiments. Moreover, the hardware supported scheme has negligible impact on performance, with a 0.23% slowdown on average. Overall, we show that the `OSCS` algorithm can be supported in hardware with limited bookkeeping and very low performance overhead while still maintaining a high detection coverage.

The rest of the chapter is organized as follows. Section 3.2 presents the idea of detecting non-race concurrency bugs based on order-sensitive critical sections. Then, Section 3.3 describes how this idea can be realized as a detection algorithm, and Section 3.4 discusses how the algorithm can be implemented with hardware support. Section 3.5 evaluates the proposed scheme.

3.2 Order-Sensitive Critical Sections

This section introduces a new heuristic for detecting non-race concurrency bugs along with examples. We first discuss how traditional data races capture concurrency bugs. Then, we extend the idea behind data races into non-race bugs and propose a new heuristic.

3.2.1 Intuition in Bug Detection Through Data Races

Informally, concurrency bugs can be considered as mistakes in synchronization that allow an unintended thread interleaving pattern, which results in inconsistent program outcomes from run to run even for identical inputs. Even though programmers may intentionally allow non-deterministic outputs in favor of reduced overheads when precise outputs are not important as in statistics counters, such non-deterministic outputs are infrequent in practice. In this sense, a non-deterministic output often indicates a concurrency bug.

Because exhaustively testing outputs for non-determinism is infeasible in practice, concurrency bug detection approaches often check for improper synchronization patterns, which lead to non-deterministic memory state and likely

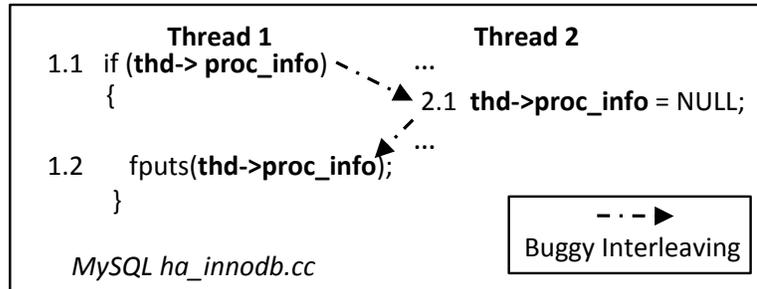


Figure 3.1: A data race example (from MySQL [42]).

non-deterministic program outputs. For example, data races capture a large class of concurrency bugs where synchronization operations are missing between a pair of conflicting memory accesses. Here, conflicting accesses are defined as ones from different threads to the same memory location, with at least one write. The data races imply that conflicting accesses can be executed in an arbitrary order, resulting in an non-deterministic shared memory state.

Figure 3.1 shows a data race in MySQL that results in an atomicity violation bug. In this example, none of the accesses to the shared pointer `thd->proc_info` is protected by synchronization. These accesses can be freely reordered, resulting in a fault if the pointer is set to be NULL by 2.1 between 1.1 and 1.2. To prevent the data race, both 1.1-1.2 and 2.1 need to be protected by a critical section to ensure an atomic execution.

Data races can be detected at run-time by checking if two conflicting memory accesses can be re-ordered while maintaining the same happens-before relationships among *all* synchronization operations. Re-ordering a pair of memory accesses without changing the order of synchronization operations is possible only when there is no synchronization between the accesses, either directly or indirectly.

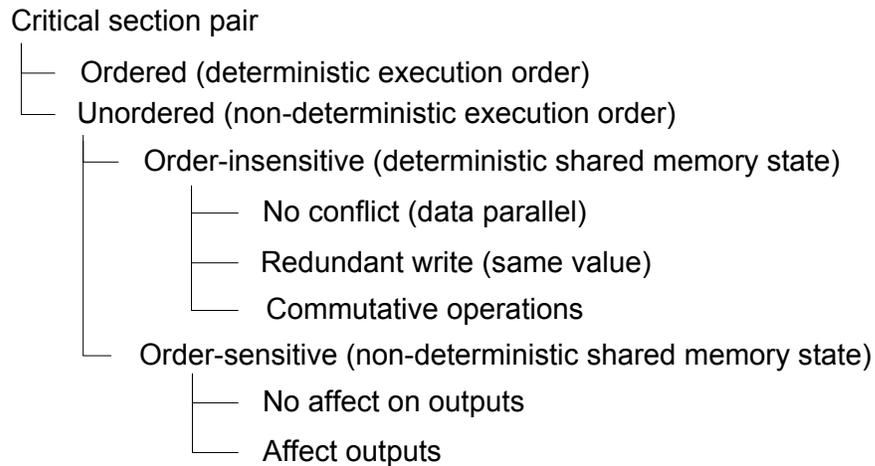


Figure 3.2: Types of critical section pairs.

3.2.2 Non-Determinism in Critical Sections

While data races can detect a broad range of concurrency bugs where conflicting memory accesses are not controlled at all, a recent study shows that many concurrency bugs do not fall into data races [42]. Programmers can remove data races by placing shared memory accesses within critical sections. However, because the critical sections can still execute in an arbitrary order, program outputs may still be non-deterministic even without data races. For example, while the data race shown in Figure 3.1 can be fixed by protecting both *1.1-1.2* and *2.1* with critical sections, the program output remains to be non-deterministic as *2.1* can be executed either before or after *1.1-1.2*.

In this research, we aim to develop a general technique to detect a broad range of non-race concurrency bugs. To achieve this goal, we extend the intuition behind data races, where non-deterministic shared memory state from the lack of synchronization is used as an indicator for a concurrency bug, to critical

Thread 1	Thread 2
1.1 Lock (l);	
1.2 clear(table , 0, size);	
1.3 table->empty = true;	
1.4 UnLock (l);	
...	...
1.5 barrier();	2.1 barrier();
...	2.2 Lock (l);
	2.3 table[index++] = obj;
	2.4 table->size++ ;
	2.5 table->empty = false;
	2.6 UnLock (l);

(a) Ordered critical section pairs.

Thread 1	Thread 2
1.1 local_sum = compute();	2.1 local_sum = compute();
1.2 Lock (l);	2.2 Lock (l);
1.3 global_sum += local_sum ;	2.3 global_sum += local_sum ;
1.4 UnLock (l);	2.4 UnLock (l);

(b) Commutative critical section pairs.

Thread 1	Thread 2
1.1 void LoadScript (nsSpt* aspt) {	
1.2 Lock (l);	
1.3 gCurrentScript = aspt;	
1.4 LaunchLoad (aspt);	
1.5 UnLock (l); }	
	...
	2.1 Lock (l);
	2.2 gCurrentScript = NULL;
	2.3 UnLock (l);
	...
1.6 void OnLoadComplete () {	
/* call back function of	
LaunchLoad */	
1.7 Lock (l);	
1.8 gCurrentScript->compile ();	
1.9 UnLock (l); }	

Mozilla nsXULDocument.cpp

- - - ►
 Buggy Interleaving

(c) Order-sensitive critical section pairs. Non-race atomicity violation bug (Mozilla-1) [43].

Figure 3.3: Examples for different types of critical section pairs.

sections. In essence, we look at a pair of critical sections with conflicting memory accesses and use it as an indicator for a concurrency bug if the two critical sections can lead to non-deterministic shared memory state depending on the order in which they execute. We call such critical section pairs as *order-sensitive critical sections*.

To understand when critical sections lead to non-deterministic shared memory state, let us consider how critical sections are typically used to produce deterministic state even when the mutex synchronization does not enforce any specific ordering between critical sections. Figure 3.2 categorizes common relationships between a pair of critical sections.

A critical section pair can first be categorized based on whether the two critical sections can truly run in an arbitrary order. Even though the mutex synchronization does not enforce any ordering, programmers may use an additional ordering synchronization operation to ensure a certain order between critical sections. In this case, the critical sections execute in a deterministic order. For example, Figure 3.3(a) shows an *ordered* critical section pair from a simple database table manipulation example. In the example, one thread (Thread 1) initializes the table to an empty state before reaching a barrier, and another thread (Thread 2) can add entries to the table only after the barrier. Therefore, the critical section in Thread 1 is ordered to execute before the critical section in Thread 2 by the barrier ordering synchronization operation.

An unordered critical section pair can either be order-insensitive or order-sensitive. The order-insensitive pairs imply that the shared memory state after the two critical sections will be identical no matter which order they run. There are a few common cases where the ordering of critical sections does not affect

the program state. First, critical sections may work on disjoint sets of memory locations without any conflicting accesses. Such a case is common for a data-parallel part of a program. Second, a conflicting write may be redundant, resulting in the same value in memory no matter which order the two critical sections execute. For example, a program may initialize a variable in multiple threads to a single value. Finally, the operations in the critical sections may be commutative. For example, Figure 3.3(b) shows an example where two threads add a partial sum to the global sum. Because an addition is commutative, the result is identical no matter which order the two threads run.

If the execution order of two critical sections affect the resulting shared memory state, the critical section pair is *order-sensitive*. In this case, the shared memory state can be non-deterministic even for identical inputs. If the inconsistency in memory affects a program output, such order-sensitive critical sections represent concurrency bugs. For example, Figure 3.3(c) shows an atomicity violation example from Mozilla [43]. In this case, each memory access to the shared variable `gCurrentScript` is protected by a lock. However, the value of `gCurrentScript` depends on the order in which the two threads' critical sections execute. The program can crash when the thread interleaving follows `1.3 - 2.2 - 1.9`; `gCurrentScript` will be `NULL` for `1.9`.

In this work, we propose to use the order-sensitive critical sections as indicators for bugs. In that sense, we refer to our detection scheme as `OSCS` (Order-Sensitive Critical Sections). Note that we only consider *shared* memory locations, which are used by multiple threads, to determine order-sensitivity. We do not use non-determinism in thread local state as an indicator for bugs. This is because non-deterministic shared memory state is much more likely to result in

non-deterministic program outputs compared to non-deterministic local state. For example, in a case where multiple consumers work on a set of data in parallel, the thread local state will depend on which data that a particular thread works on and can easily be different from run to run even when outputs are deterministic.

3.2.3 Detection Heuristic

Here, we discuss a heuristic approach to detect order-sensitive critical sections in practice. Instead of precisely detecting all order-sensitive critical sections, the goal is to develop a simple heuristic that detects most order-sensitive critical sections with *minimal false positives*. Also, we want the heuristic to be simple enough for run-time checks. Just like data race detectors are often used even when they cannot detect all concurrency bugs, we believe a detection scheme will be useful if it covers a broad range of bugs with low false positives.

While there can be many ways to detect order-sensitive critical sections, our approach defines the order-critical sections indirectly as critical sections that are neither ordered nor order-insensitive. Then, the approach relies on a set of heuristics to filter out common patterns for ordered and order-insensitive critical sections as shown below. The detailed run-time detection algorithm is explained in the next section.

Ordered critical sections: The scheme keeps track of restrictions from all ordering synchronization operations. If there exists a direct or indirect ordering restriction between two critical sections, the pair is considered to be ordered.

No conflict (order-insensitive): If there is no conflicting accesses between two critical sections, the pair is order-insensitive. This condition can be checked by keeping track of previous accesses for each memory location.

Redundant writes (order-insensitive): If a conflicting write does not change the value in memory, the write operation is redundant. For example, both critical sections may write the same value. This condition can be checked by comparing the memory value before and after a conflicting write.

Commutative operations (order-insensitive): Precisely checking if two critical sections are commutative requires understanding of detailed semantics of operations as well as data dependence. Instead, we use a simple heuristic based on a memory access pattern to conservatively filter out commutative operations. Intuitively, in order for an operation on a shared memory location to be commutative, each thread needs to first read the current value before updating it, often atomically. For example, adding a partial sum to a global sum in Figure 3.3(b) requires reading the current global sum and writing the updated sum within a critical section.

Based on this intuition, our heuristic considers a pair of critical sections to be commutative if both contain a read followed by a write for each memory location with conflicting accesses between two critical sections. We call the sequence of a read followed by a write to the conflicting location a *read-write* sequence.

To understand how the proposed checks can detect bugs, let us consider the atomicity violation example in Figure 3.3(c). In this example, the critical sections in Thread 1 (1.2 to 1.5) and Thread 2 (2.1 to 2.3) are unordered and have conflicting writes to `gCurrentScript`. Also, each critical section only

contains one write instead of a read-write sequence, which indicates that the critical sections are not commutative. Therefore, the two critical sections will be flagged as order-sensitive.

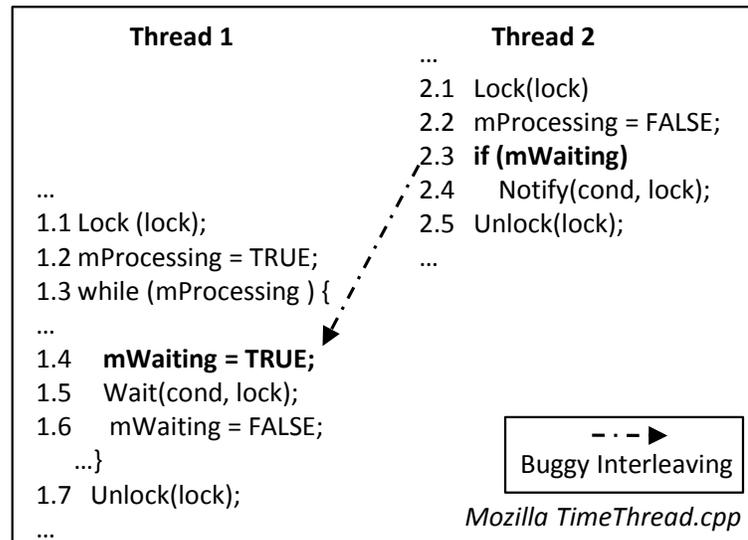


Figure 3.4: An ordering violation bug example (KB3) [76].

As another example, Figure 3.4 shows an ordering violation example from Mozilla. The program hangs if the thread interleaving follows 2.3 - 1.4 because `mWaiting` would be `false` at 2.3 and Thread 1 waits at 1.5 for a notification that will never be sent. The critical section pair in this example will be flagged as order-sensitive because they are not ordered, and have conflicting accesses to `mProcessing` and `mWaiting` without a read-write sequence in both critical sections.

3.2.4 Limitations

The proposed detection scheme, *OSCS*, is a heuristic for detecting non-race concurrency bugs. In that sense, *OSCS* can have both false negatives and false positives, like any other heuristic methods. For example, data race detectors cannot detect all concurrency bugs (false negatives) and may also incorrectly identify programmer intended races for bugs (false positives). Our experiments, however, indicate that *OSCS* can detect a broad range of bugs with a small number of false positives.

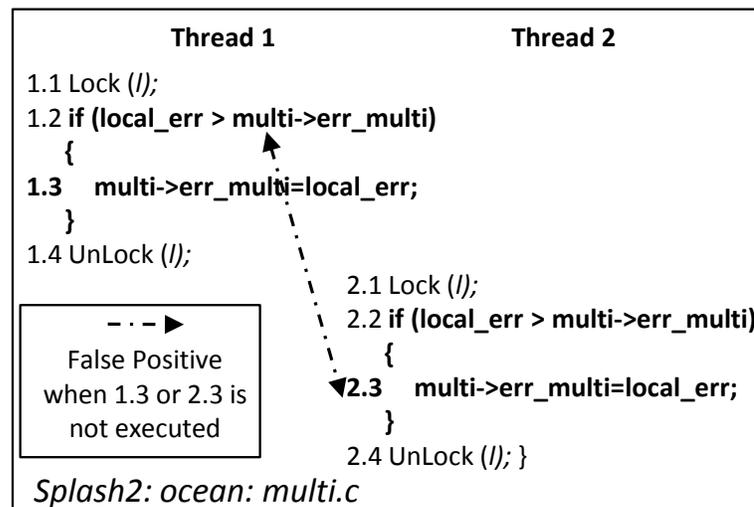


Figure 3.5: A false positive example from Ocean.

From the false positive perspective, *OSCS* relies on the assumption that non-deterministic shared memory state indicates a bug. However, if the non-deterministic shared memory state does not lead to non-deterministic outputs or the outputs are intentionally allowed to be non-deterministic, our approach can lead to a false positive. The *OSCS* scheme also uses a heuristic to detect commutative critical sections where *read-write* sequences in both critical sections are considered as an indicator for a commutative operation. This heuristic can lead

to false positives when it is not enough to detect all commutative operations as shown in Figure 3.5. In this example, each thread is executing the same piece of code, which updates the global maximum error value. A false positive happens when either Thread 1 or Thread 2's `local_err` is less than global variable. This results in one of the critical sections with only a read, while the other critical section has a read followed by a write to the global variable.

There are a couple of approximations in the proposed heuristics that can lead to false negatives (missed bugs). First, we do not detect non-deterministic thread local state as a bug because they are difficult to check and could lead to false positives. Second, the heuristic to detect commutative critical sections can be conservative and incorrectly mark non-commutative ones as commutative. In addition to these approximations in our heuristics, `OSCS` is designed to detect bugs in accesses within critical sections, and does not detect data race bugs or bugs from misplaced ordering synchronization operations.

Note, however, that false positives and negatives are common for any heuristic method. An important question is how well the heuristic works in practice. Our tests on real-world application programs including Apache, MySQL, Aget, Pbzzip2, PARSEC, and SPLASH2 indicates that there are only a small number of false positives. The experiments on real-world and injected bugs show that the `OSCS` scheme is quite effective in detecting non-race bugs; the scheme detected all real-world bugs and over 90% of injected bugs.

3.3 Detection Algorithm

This section presents the run-time algorithm for detecting order-sensitive critical sections. We first present a high-level overview of the algorithm and the meta-data, followed by the detailed algorithm and optimizations.

To be general, we describe the algorithm using *release* and *acquire* instead of individual synchronization operations. Recall that synchronization primitives can be considered as acquiring and releasing tokens, which we refer to as *synchronization objects*. For example, mutual exclusion requires each thread to acquire a token (lock) before entering a critical section and to release the token after the critical section. Similarly, barriers can be realized by having each thread to release a token upon reaching the barrier and waiting to acquire tokens from all other threads before proceeding.

3.3.1 Overview

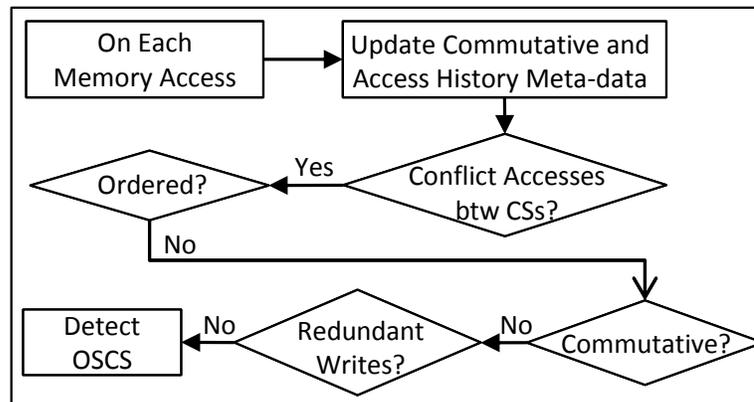


Figure 3.6: OSCS operations on every memory access.

In a high-level, the OSCS algorithm detects a potential concurrency bug by

Meta-data:

1. CSList [TID]:
A set of critical section(s) that a thread is currently in.
2. PrevReadCSList [Addr] [TID] /
PrevWriteCSList [Addr] [TID]:
A set of critical section(s) that a thread was in when the most recent memory accesses happened in that thread.
3. ThreadVC1k [TID1] [TID2]:
A vector clock per thread. A timestamp is stored in each element.
4. PrevReadVC1k [Addr] [TID] /
PrevWriteVC1k [Addr] [TID]:
Time stamp per address per thread for most recent read/write.
5. OrderObjVC1k [OrderObj] [TID]:
A vector clock per ordering synchronization object.
A timestamp is stored in each element.
6. CSRWFlag [Addr] [TID]:
A single bit flag per memory address per thread that indicates whether a read-write sequence exists within a critical section.
7. DetectionList [TID]:
A list of potentially buggy memory access pairs in each thread.

Figure 3.7: Meta-data required for our algorithm.

detecting two critical sections with conflicting memory accesses and checking if the pair is order-sensitive from the perspective of that shared location.

Figure 3.6 shows an overview of the OSCS algorithm on each memory access. The algorithm first checks if there is a previous access to the memory location from another thread, which forms a conflicting access pair with the current access. Then, the algorithm checks if both conflicting accesses are protected by a critical section with the same lock. These checks identify a critical section pair with conflicting memory accesses, which are not data races.

Once a critical section pair is identified, the algorithm checks if the pair is order-sensitive by filtering out ordered and order-insensitive ones. First, the algorithm checks if the critical sections are ordered by keeping track of constraints from all ordering synchronization operations. Then, the algorithm

```

Functions on synchronization operations:
update_acquire(TID, SyncObj)
1. If (type(SyncObj) == OrderObj),
   (a) ThreadVClk[TID][TID]++;
   (b) For each i, update the vector clock: ThreadVClk[TID][i] =
       MAX(ThreadVClk[TID][i], OrderObjVClk[SyncObj][i]).
2. If (type(SyncObj) == MutexObj),
   Add SyncObj to CSList[TID].

update_release(TID, SyncObj)
1. If (type(SyncObj) == OrderObj),
   (a) ThreadVClk[TID][TID]++;
   (b) For each i, update the vector clock: OrderObjVClk[TID][i] =
       MAX(ThreadVClk[TID][i], OrderObjVClk[SyncObj][i])
2. If (type(SyncObj) == MutexObj),
   (a) Remove SyncObj from CSList[TID].
   (b) If (CSList[TID] is empty),
       For each access pair in DetectionList[TID],
       If (!(CSRWFlag[Addr][PrevTID]) || !(CSRWFlag[Addr][TID])),
       Report a bug.

```

Figure 3.8: OSCS: detailed algorithm for operations on each synchronization operations.

checks whether the critical section pair is order-insensitive due to commutative operations or redundant writes. If the critical section pair is neither ordered or order-insensitive, the algorithm logs it as a potential concurrency bug.

In order to perform the necessary checks described above, the algorithm maintains a set of meta-data on each memory accesses as well as each synchronization *acquire* or *release* operation. Figure 3.7 shows a summary of meta-data variables that are used in OSCS. Here, *TID* represents a thread ID, *Addr* represents a byte address for each memory location, and *OrderObj* represents each ordering synchronization object.

```

Functions on each memory access:
OSCS_detectCS(TID, Addr, Type, ThreadVClk[TID][TID]):
1. Check the most recent write in each thread:
  (a) For all valid threadID i ≠ TID,
    (a.1) If (CSList[TID] ∩ PrevWriteCSList[Addr][i])
      Call check_order(TID, i,
        PrevWriteVClk[Addr][i], Addr).
2. Check the most recent read in each thread:
  (a) If (Type == Read), skip Step 3.
  (b) For all valid threadID i ≠ TID,
    (b.1) If (CSList[TID] ∩ PrevReadCSList[Addr][i])
      Call check_order(TID, i,
        PrevReadVClk[Addr][i], Addr).
3. Update the access history for the memory location
  (a) If (Type == Read),
    PrevReadVClk[Addr][TID] = ThreadVClk[TID][TID].
    PrevReadCSList[Addr][TID] = CSList[TID].
  (b) If (Type == Write),
    PrevWriteVClk[Addr][TID] = ThreadVClk[TID][TID].
    PrevWriteCSList[Addr][TID] = CSList[TID].

check_order(TID, PrevTID, PrevTimeStamp, Addr)
1. If (ThreadVClk[TID][PrevTID] ≤ PrevTimeStamp)
  Call check_commutative(TID, PrevTID, Addr).

update_commutative_flag(TID, Addr, Type):
1. If (Type == Write && CSList[TID] ∩ PrevReadCSList[Addr][TID]),
  CSRWFlag[Addr][TID] = True.
  Else, CSRWFlag[Addr][TID] = False.
2. If (Type == Read && !(CSList[TID] ∩ PrevReadCSList[Addr][TID])),
  CSRWFlag[Addr][TID] = False.

check_commutative(TID, PrevTID, Addr)
1. If (!(CSRWFlag[Addr][PrevTID])),
  Report a bug.
2. If (CSRWFlag[Addr][PrevTID] && !(CSRWFlag[Addr][TID])),
  Add an entry to DetectionList[TID].

```

Figure 3.9: OSCS: detailed algorithm for operations on each memory operations.

3.3.2 Detailed Algorithm

This subsection describes the details of the necessary meta-data and operations for each step in the algorithm. Figure 3.8 shows the pseudo-code for operations

on each synchronization operation and Figure 3.9 shows the operations on each memory access.

Critical Sections with Conflicting Accesses

The algorithm detects a critical section pair with conflicting accesses in two steps: detect conflicting accesses, and check if they are within critical sections. To detect conflicting accesses, the algorithm maintains timestamps for the most recent read and write from each thread to each memory location, `PrevReadVClk[Addr][TID]` and `PrevWriteVClk[Addr][TID]`. The timestamps are from each thread's local clock. On each memory access, the algorithm updates the meta-data and checks if there is a conflicting access by looking up previous reads and writes from other threads.

To identify corresponding critical sections for conflicting accesses, the algorithm uses `CSList[TID]`, `PrevReadCSList[Addr][TID]` and `PrevWriteCSList[Addr][TID]`. `CSList` shows the critical sections that each thread is currently running by recording the mutex object (`MutexSyncObj`) and a starting timestamp for each critical section. `PrevReadCSList` and `PrevWriteCSList` record the critical sections for the most recent read and write accesses from each thread to each memory location. On conflicting accesses, the algorithm checks if they are both protected by a critical section with the same lock by comparing the mutex object stored in `CSList` and `PrevReadCSList/PrevWriteCSList`.

The algorithm updates `CSList` on each acquire or release operation by calling `update_acquire()` and `update_release()` in Figure 3.8. Updates for

the rest of the meta-data and checks are performed on each memory access by calling `OSCS_detectCS`, as shown in Figure 3.9.

Ordered Critical Sections

Once a candidate critical section pair is identified, the algorithm checks if the pair is constrained to run in a deterministic order. For this purpose, the algorithm uses vector clocks for each thread (`ThreadVClk[TID1][TID2]`) and ordering synchronization object (`OrderObjVClk[OrderObj][TID]`) to encode ordering constraints between two threads. `ThreadVClk[i][j]` shows the earliest that a memory access from Thread i can be executed in terms of Thread j 's local time without violating the ordering constraint from synchronization. The `OrderObjVClk` represents the earliest that the following acquire operation can happen in each thread's local time for each ordering synchronization object. `ThreadVClk[i][i]` represents Thread i 's local clock, which is incremented on each ordering synchronization operation.

On an *acquire* operation by Thread i for ordering synchronization, the `ThreadVClk[i]` is updated with `OrderObjVClk[OrderObj]` by taking the larger timestamp for each element. On a *release* operation for ordering synchronization, `OrderObjVClk` is updated with `ThreadVClk`. The operations are shown in `acquire_update()` and `release_update()`.

The algorithm uses the vector clocks to check if conflicting accesses from two critical sections are ordered (shown in `check_order()`). The current memory access from Thread TID and a previous access from Thread $PrevTID$ are not explicitly ordered if `PrevVClk[Addr[PrevTID]]` is greater or equal to

ThreadVClk[TID][PrevTID].

Commutative Critical Sections

To detect commutative critical sections, the algorithm uses a 1-bit flag (`CSRWFlag[Addr][TID]`) to indicate whether a *read-write* sequence within a critical section has happened to each memory location from each thread. On a write, `CSRWFlag` is set when the most recent local read was also within the same critical section, and cleared otherwise. The algorithm also clears `CSRWFlag` on the first read within a critical section or reads outside critical sections. These updates are shown in `update_commutative_flag()`.

In our heuristic, a critical section pair is considered to be commutative if *both* critical sections contain a read-write sequence for each conflicting memory location. The algorithm checks this property by calling `check_commutative()` on detecting unordered conflicting accesses from two critical sections. If `CSRWFlag` is not set for the earlier conflicting access, the algorithm can immediately determine that critical sections are non-commutative. However, if the previous access has `CSRWFlag` set while the current access does not, the conflicting access pair is added to `DetectionList` and checked again when the current thread exits the critical section (shown in `release_update()`).

Redundant Writes

The algorithm can also check memory values before and after a conflicting write to see if the write is redundant. We discuss its impact on false positives in the evaluation section. For simplicity, this check is not shown in the pseudo-code.

3.3.3 Debug Information on Detection

The `OSCS` scheme can pinpoint the critical sections and conflicting memory accesses for a potential bug. Our implementation reports the program counters and memory addresses for conflicting accesses, the associated thread IDs, and the time-stamped mutex object for a potential bug. to help debugging. More debugging information can be added if necessary.

3.3.4 Optimizations

For simplicity, we first described the `OSCS` algorithm using vector variables for each memory location. However, the per-thread, per-byte meta-data variables result in a significant memory overhead, which increases linearly with the number of threads. To be more practical, here we propose an optimized algorithm, named `OSCS-Opt`. We will refer to the original algorithm as `OSCS-Base`.

To reduce overhead and improve scalability, `OSCS-Opt` uses scalar variables in place of vector variables for bookkeeping. Specifically, for each location, the algorithm only maintains the history of the two most recent reads and two most recent writes from different threads rather than one read and one write per thread. The algorithm needs the most recent read and write from the current thread within a critical section to update `CSRWFlag`, and at least one recent read/write from another thread to detect conflicting accesses. The scalar variables do not increase with the number of threads.

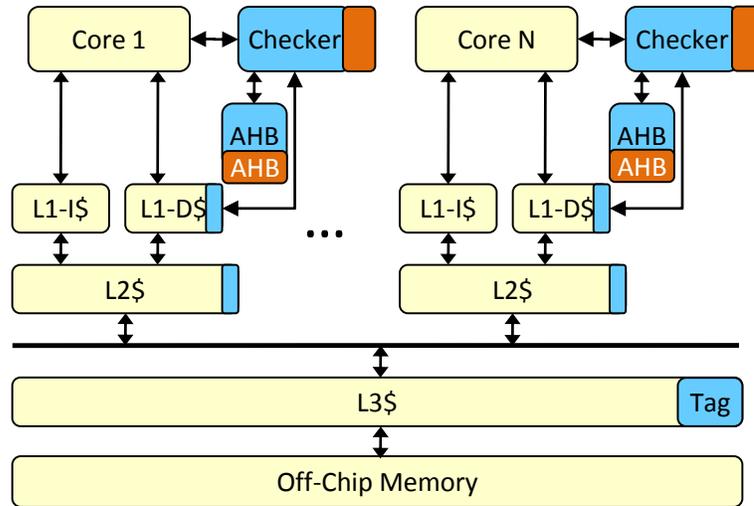
`OSCS-Opt` further reduces the amount of meta-data on previous accesses by keeping the history only for accesses within critical sections to shared memory

locations that have conflicting accesses, exploiting that detection only uses those accesses. Our test shows that shared memory locations are only a small fraction of the entire memory space. Recent studies [17, 31] also made an observation that a significant percentage of memory blocks are only accessed locally by one thread. Our evaluation results indicate that these optimizations significantly reduce the space overhead without a noticeable impact on detection capability.

3.4 Hardware Implementation

This section introduces a hardware implementation of the `OSCS` algorithm, named `OSCS-HW`. Figure 3.10(a) shows the high-level block diagram for the architecture where the blue (dark) blocks indicate the new components that are needed to support `OSCS`. The architecture is based on the `OSCS-Opt` algorithm, which uses scalar meta-data for each shared memory location, but further limits the meta-data for efficiency. Specifically, `OSCS-HW` detects shared locations only through on-chip caches by tagging cache blocks and maintains per-location meta-data only in an on-chip buffer, named `AHB`. The checker module performs bookkeeping and check operations at each core and maintains per-thread meta-data.

We note that the our hardware design in `OSCS-HW` consists the same components as in `RaceSMM` (shown as the blue (dark) blocks). The only notable differences in `OSCS-HW` are the additional meta-data in the `AHB` block and the additional logic in the checker block (shown as the orange (light) blocks). Figure 3.10(b) shows the key components of `OSCS-HW`, which are discussed in detail in the rest of this section.



(a) A block diagram for the overall architecture. Blue (dark) blocks are additional hardware support needed for OPCS (same as in RaceSMM). The orange (light) AHB block indicates the additional meta-data AHB needs to store for OPCS. The orange (light) block in checker indicates the additional operations that need to be performed by the checker module in order to support OPCS.

Shared Location Detection: 1-bit shared tag for L1/2/3 cache block, set on L1/2 cache coherence downgrade event, propagate to L3.

Access History Buffer (AHB): Meta-data for two most recent read/write to a shared location. Including PrevTIDs, PrevTSs, and the CSRWFlags, PrevCSLists needed for OPCS.

Checker Module: Keeps active TID, ThreadVClk and CSList. Within a critical section, on shared access from L1, records access to AHB, checks for bugs. Bookkeeping ThreadVClk and SyncObjVClk on ordering sync. operations.

Order Sync. Object Vector Clocks and Detection List (not shown in the block diagram): OrderObjVClks and DetectionList are stored and accessed through existing memory hierarchy.

(b) Key components for the OPCS architecture

Figure 3.10: Architecture overview for OPCS.

In our architecture, each block in data caches has a 1-bit tag, which indicates whether the block is shared, same as in RaceSMM. Recall that the shared bit is

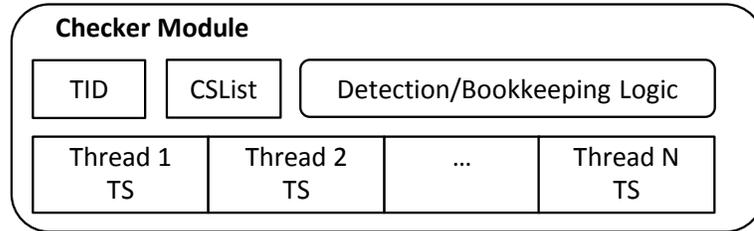


Figure 3.11: Checker: keeps current thread ID (TID), a thread vector clock (ThreadVClk), the most recent critical section information that a thread is currently in (CSList) and detection/bookkeeping logic.

set when a cache coherence event indicates that multiple cores access the same cache block with at least one write. More specifically, the shared bit is set when there is a downgrade request that changes the cache block to either shared or invalid state. For example, in a MESI protocol, the shared bit is set on the following requests: $M \rightarrow S$, $M \rightarrow I$, $E \rightarrow S$, $E \rightarrow I$, $S \rightarrow I$, etc.

3.4.1 Checker Module

The checker module for `OSCS-HW` maintains a per-thread state and performs the bookkeeping and checking operations at each core. As shown in Figure 3.11, for the active thread on the core, the checker keeps a thread ID (TID), a thread vector clock (ThreadVClk), and the critical section that a thread is currently in (CSList).

We note that in order to support nested critical sections, we may need to store more than one critical sections the `CSList`. However, we have found that it is very rare to have nested critical sections in practice. In fact, we have not encountered any real-world or randomly injected bugs which involve nested

critical sections in our evaluation. Moreover, storing the most recent critical section ID and its timestamp can still detect bugs within nested critical sections, as long as the conflicting accesses are within critical sections that are protected by the same mutex object. We also note that storing the most recent critical section information can only lead to potential false negatives as two conflicting accesses will never trigger a detection unless they share the same critical section ID. Therefore, in `OSCS-HW`, we have made the design choice of only storing the most recent critical section ID and its timestamp on chip. However, a simple extension can be made by adding additional on-chip storage or store the critical section information in lower caches to provide complete support to nested critical sections.

On each memory access, the checker module checks if the memory location is “shared” and within a critical section by looking at a cache tag and `CSList`. If so, the checker module stores the access history in an on-chip buffer (AHB). As we only detect non-race concurrency bugs in `OSCS-HW`, bookkeeping of shared accesses within critical sections is sufficient. The checker module also detects conflicting accesses within critical sections using the history in the AHB and the `CSList`, and performs checks for bug detection on those critical sections.

For ordering synchronization operation, the checker module coordinates with a software layer through new instructions for bookkeeping. The architecture provides two additional instructions, one for *acquire* and the other for *release*, conveying the address of the vector clock for the synchronization object. The vector clock is accessed and/or updated by the checker module through normal memory hierarchy.

Access History Buffer (AHB)					
Addr Tag	Prev. Reads TS+TID	Prev. Reads CSList	Prev. Writes TS+TID	Prev. Writes CSList	CSRW Flag bit
...

Figure 3.12: AHB: a history table that saves previous accesses information for two most recent read and write accesses to shared memory locations within critical sections

3.4.2 Access History Buffer

As shown in Figure 3.12, the access history buffer (AHB) in `OSCS-HW` keeps the meta-data for two recent read and write accesses, including the previous TIDs and the previous timestamps for those accesses. Additionally, each AHB entry also keeps the most recent critical section information and the `CSRWFlag` for each of the previous accesses.

The AHB is kept coherent by a cache coherence protocol similar to other on-chip caches. We implemented the AHB coherence protocol separate from the main data cache, however, we note that as the total number of accesses to AHB is very small as discussed earlier, the AHB coherence operations can be done using the existing main cache coherence structure with minimal overhead as well.

Note that unlike the AHB used in `RaceSMM`, we decide to save both the read and write access histories globally here. This is because accesses to AHB only happen rarely in practice, and the associated overheads, especially the overhead for keeping AHB coherent, are kept at minimum. Hence, we decide to use current implementation of AHB for its simplicity.

Similar to the AHB used in `RaceSMM`, the AHB in `OSCS-HW` has a limited capacity and there is no backup hierarchy for the AHB. If an entry is evicted from an AHB, the information is simply thrown away. A miss to the AHB creates a new entry. While this design implies that we cannot detect conflicting accesses with a long distance in between, our experimental results suggest that an AHB with 256 entries are sufficient for virtually all concurrency bugs tested. Moreover, a miss to the AHB can only lead to potential false negatives, but not false positives as we would not make a detection on a miss. We will further examine the impact of AHB size on the detection capability in our evaluation section.

Additionally, our AHB implementation can store an access history per byte with the flexible granularity optimization discussed earlier in `RaceSMM`. Namely, for each AHB entry, there is a single-bit flag that indicates if the granularity is per byte or per word. An additional 4-bits are also appended to each entry to mark which bytes in a word are associated with the AHB entry.

3.4.3 Vector Clocks

Instead of keeping a vector clock per synchronization object, as in `RaceSMM`, the `OSCS` scheme only requires a vector clock per ordering synchronization object. Therefore, the overall space and access overheads for vector clocks are insignificant in practice as the total number of ordering synchronization objects are generally very small in applications. Moreover, as we now only increment the vector clock elements on ordering synchronization operations, the hardware counters for the vector clocks would encounter an overflow very infrequently. In fact, we did not encounter any overflow cases in our evaluation.

Given that overflows are infrequent, our architecture handles them in a relatively slow but straightforward fashion instead of adding complex hardware, same as `RaceSMM`. Upon detecting an overflow in its local clock, a checker raises an exception to an operating system, which in turn interrupts other cores that run other threads from the same program. Then, the operating system clears all clocks, and marks all AHB entries to be invalid in each core. In order to allow an operating system to clear vector clocks for synchronization objects, an application allocates them in separate pages that are known to the operating system.

3.5 Evaluation

This section presents evaluation results for the proposed `OSCS` scheme in both software and hardware implementations. We first study the scheme’s bug detection capability, then discuss the overheads.

3.5.1 Evaluation Setup

For the evaluation, we implemented three schemes, `OSCS-Base`, `OSCS-Opt`, and `OSCS-HW`, using the Pin binary instrumentation framework [47]. Our Pin tool implements the `OSCS` algorithms by intercepting memory accesses and Pthread calls.

To evaluate the architectural support, we implemented a typical memory hierarchy with bookkeeping structures in a Pin tool, and added a timing model with a processing core that runs 1 instruction per cycle, L1/L2/L3 caches, and

Table 3.1: Baseline architecture parameters.

Component	Parameters
Core	4 2-GHz in-order single-issue cores
Caches	L1 I/D (private, inclusive): 32KB/32KB 4-ways 3 cycles Latency L2 (private): 256KB, 4-ways 15 cycles latency L3 (shared): 8MB, 8-ways 40 cycles latency
Coh. protocol	MESI
DRAM	4GB 50ns Latency
Meta-data	8-bit thread IDs, 16-bit clocks
AHB	256-entries, 8-way, 9KB, 3 cycles Latency

a memory interface. Table 2.2 summarizes the baseline architecture parameters for `OSCS-HW`. For the `OSCS` bookkeeping, we model a 1-bit shared bit per cache block and a 256-entry AHB. An AHB entry records an 8-bit thread ID and a 16-bit timestamp per access, for two reads and two writes, and a 16-bit critical section ID and a 16-bit timestamp for the corresponding critical section.

To evaluate the bug detection capability, we used both real-world bugs and injected bugs. For real-world bugs, we first used five non-race bugs on two large real-world applications (MySQL and Mozilla). We also created kernel bugs (KB), which use two threads to reproduce real-world bugs (MySQL and Mozilla) that are reported in previous studies [42, 43, 76]. The list includes all real-world non-race bugs that we could find and reproduce. For a further study on detection, we also tested non-race bugs that are intentionally injected to programs from the SPLASH2 [15] and PARSEC [8] benchmark suites. The following subsection describes the details of the bug injection study.

Table 3.2: Non-race concurrency bugs tested. (A - Atomicity violation, O - Order violation, M - Multi-variable bug).

Bugs	Description
MySQL1-A	Log status can be read intermediately from a remote thread while its two update operations are not within the same critical section.
MySQL2-AM	The log and table operations are not updated within the same critical section. The operation and its log may not match.
Mozilla1-A	NULL can be set in between script-set and script-compile.
Mozilla2-AM	Variable is being cleared in between initialization and setting the empty flag to false, resulting a mismatch.
Mozilla3-O	Loop break flag is set too early, resulting in an infinite loop.
KB1-A (MySQL)	After an update, the variable can be set back to zero intermediately by a remote thread.
KB2-AM (MySQL)	An updated variable and its logged updated value may not match due to remote intermediate update operations
KB3-O (Mozilla)	A waiting flag is checked before it's set, program hangs.
KB4-AM (Mozilla)	Two related variables are separately protected by different locks, and not updated atomically together.

For a false positive study, we ran the SPLASH2 (4 threads, default input size) and PARSEC (4 threads, *simmedium* input size) benchmarks. We also tested full deployed instances of Apache (30 threads), Mozilla (8 threads), MySQL (10 threads), Aget (8 threads), and Pbzip2 (8 threads). The workloads for them were created to mimic real uses.

Table 3.3: Bug injection study. Potential non-race bugs are injected to measure the detection coverage. (P) - PARSEC, (S) - SPLASH2, (A) - Atomicity violation, (O) - Order violation.

	Unique Cases			
	Injected	OSCS-Base Detected	OSCS-Opt Detected	OSCS-HW Detected
Bodytrack (P)	3(A)	2(A)	2(A)	2 (A)
Fluidanimate (P)	5(A)	5(A)	5(A)	5(A)
LU (S)	8(A) 1(O)	6(A) 0(O)	6(A) 0(O)	6(A) 0(O)
Ocean (S)	14(A) 2(O)	12(A) 2(O)	12(A) 2(O)	11(A) 2(O)
Radix (S)	4(A)	4(A)	4(A)	4(A)
Water-nquare (S)	16(A) 1(O)	16(A) 1(O)	15(A) 1(O)	14(A) 1(O)
Water-spacial (S)	14(A)	14(A)	14(A)	12(A)
Total	68	62	61	57
Detection %	—	91%	90%	84%

3.5.2 Bug Detection Capability

Table 3.2 shows the real-world bugs that we tested. `OSCS-Base`, `OSCS-Opt` and `OSCS-HW` detect all real-world bugs in the table after a single run of each application, showing that the notion of order-sensitive critical sections are effective in practice. While our algorithm is not designed for multi-variable bugs, the results show that some multi-variable bugs can be detected from non-determinism in a single variable.

In order to further study the detection capability, we injected atomicity and ordering violations into benchmarks. For atomicity violation, we first created 5 bugs by manually selecting and breaking up a critical section into two or more: 2 in `Ocean`, 2 in `Water-Nsquared`, 1 in `LU`. For ordering violation, we manually

removed a barrier or a wait instruction in a benchmark, creating 4 bugs: 2 in Ocean, 1 in Water-Nsquared, 1 in LU. Except for one injected ordering bug in LU, the OSCS algorithms successfully detected all manually injected bugs after a single run. The manually injected bugs were checked to ensure that they indeed change the program outcome or crash the program.

To obtain more test cases, we performed an automatic injection of atomicity violation bugs using Pin. Here, the tool broke up one critical section into two by injecting an unlock/lock pair between shared memory accesses. The approach constructed a number of unique bug cases by sweeping all possible options. Note that the automatically injected bugs may not truly affect the program outcome; we could not check each case due to the large number.

Table 3.3 shows the detection results for both manually and automatically injected bugs combined. The results show that the proposed scheme can detect most of injected bugs; OSCS-Base detected 91%, OSCS-Opt detected 90%, and OSCS-HW detected 84%. Most of the injected bugs were consistently detected after a single run. Some bugs were only detected on certain program runs: 12 for OSCS-base, 15 for OSCS-Opt, and 18 for OSCS-HW. More than a half of these bugs were detected greater than two third of time in tens of executions, and others were detected about one third of time. The rest of the injected bugs were consistently detected after a single run.

OSCS-Base can detect bugs that OSCS-Opt cannot because OSCS-Base keeps track of the most recent read and write per thread whereas OSCS-Opt only uses two reads and two writes to detect conflicting accesses. Similarly, OSCS-HW has a lower coverage than the software schemes due to its limited bookkeeping space in hardware. However, results for both real-world bugs and

injected bugs suggest that the proposed optimizations have a minimal impact on the bug coverage in practice. This is because most concurrency bugs involve memory accesses that are close to each other. For example, a previous study reported that typical atomicity violations happens within 750 accesses [46].

There are a few possible reasons why some of the injections are not detected by our scheme. First, if both split critical sections, after breaking up one, still have a read-write sequence for all conflicting shared variables, our algorithm will not detect such a split. Second, because our scheme only checks a conflict with the most recent read and write from each thread, a bug could be masked by another read or write within a thread if remote accesses do not happen soon. Finally, while the automatically injected bugs break up a critical section, there is no guarantee that they will result in a real bug with non-deterministic outputs. In that sense, the detection ratio can be seen as a conservative measure.

In the experiments, we found that most critical sections in real-world programs are rather small only consisting of a few shared memory accesses. Hence, if a critical section is mistakenly broken up, it is likely that at least one critical section will not have a read-write sequence. Real-world bugs that we studied also support this observation. Most real-world bugs involve only a single access to a particular variable within a critical section.

While the proposed algorithm cannot detect all non-race bugs, the above results suggest that the OSCS scheme is indeed effective in practice. Overall, the scheme detected all tested real-world bugs, and over 90% of injected atomicity and ordering violations in software implementations, and over 84% in hardware implementation.

Table 3.4: The number of false positives (redun-wr: redundant writes).

	Apache	Bodytrack	Ocean	Pbzip2	Radix	Others
OSCS-Base	4 (1 redun-wr)	3	1	6 (2 redun-wr)	3	0
OSCS-Opt	4 (1 redun-wr)	3	1	6 (2 redun-wr)	3	0
OSCS-HW	4 (1 redun-wr)	3	1	6 (2 redun-wr)	3	0

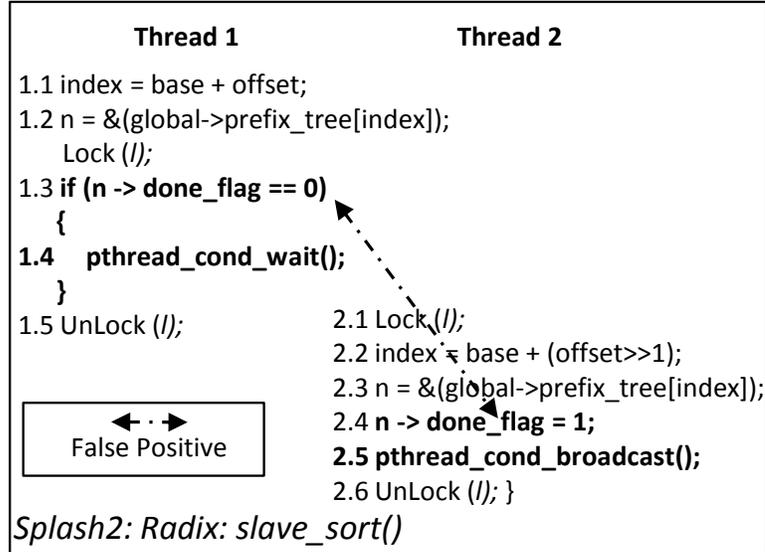


Figure 3.13: A false positive example from Radix.

3.5.3 False Positives

Table 3.4 shows false positives for our detection scheme. As shown, all three implementations have 17 false positives: 4 for Apache, 3 for Bodytrack, 1 for Ocean, 6 for Pbzip2 and 3 for Radix. There was no false positive for other programs including Aget, MySQL, Mozilla, PARSEC, and SPLASH2 benchmarks. Out of these false positives, 1 case in Apache and 2 cases in Pbzip2 were redundant writes that can be avoided by checking values for writes. We note that the reported numbers represent the total false positives over tens of program runs.

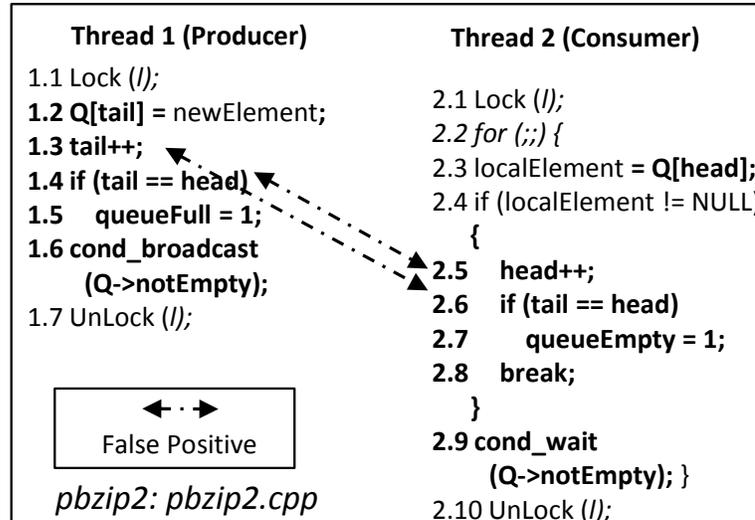


Figure 3.14: A false positive example from Pbzip2.

Four false positives from PbZip2 and all false positives from Apache, Body-track, and Radix are quite similar as they come from a synchronization operation that uses a `pthread_cond_wait()`. Figure 3.13 shows an example from Radix. Here, the `done_flag` of a node from the global `prefix_tree` is set in Thread 2 while Thread 1 is waiting for the broadcast signal. In effect, the shared variable in this case is used as a flag variable. These false positives are similar to spin flags that can cause false positives in data race detectors and can be avoided if the ordering synchronization of the flag variable can be identified explicitly.

Other false positives come from the use of heuristic in identifying commutative critical sections. For example, the false positive in Ocean happens because a MAX function only updates a global variable when it is less than the local value (discussed earlier in Figure 3.5). Two false positives from Pbzip2 involve head and tail pointers of a consumer-producer FIFO queue as shown in Figure 3.14. Here, the head and tail pointers are read by both the producer and consumer, but incremented by only one. In this case, high-level programmer knowledge is

Table 3.5: SW performance overhead. The slowdowns are normalized to the execution times of Pin instrumentation.

	Pin Slowdowns	OSCS-Base Slowdowns Compare to Pin	OSCS-Opt Slowdowns Compare to Pin
Bodytrack (P)	11.25x	25.43x	11.75x
Fluidanimate (P)	12.16x	31.46x	22.32x
LU (S)	14.10x	32.85x	22.78x
Ocean (S)	13.43x	34.53x	23.65x
Radix (S)	21.11x	32.15x	22.77x
Water-Nsquared (S)	12.52x	31.63x	21.18x
Water-Spatial (S)	13.93x	31.48x	20.79x
Geomean	13.80x	31.24x	21.52x

needed to understand that non-determinism in the head and tail pointers only affect timing but not program state.

Overall, the false positive study shows that the proposed algorithm indeed has false positives, but only has a small number of them in practice. Given that the proposed heuristic can detect a broad range of non-race bugs, we believe that the small number of false positives are acceptable and the proposed algorithm can serve as an effective tool for detecting concurrency bug. In fact, the number of false positives in our experiments were comparable to the number of false positives (data races that are not bugs) from happens-before data race detectors.

3.5.4 Software Performance Overhead

Table 3.5 shows the performance of our software implementations. For our benchmarks, the Pin framework without any instrumentation incurs a slowdown of 13.80x on average. In addition to the overhead of Pin itself, our detection algorithms show an average slowdown of 31.24x for `OSCS-Base` and 21.52x for `OSCS-Opt`. `OSCS-Opt` is noticeably faster thanks to its smaller memory footprint and the fact that it only checks up to two most recent accesses in detecting conflicting accesses. The performance overhead is comparable to a race detector based on vector clocks that we implemented in Pin as a comparison. Also, we believe that the software implementations can be further optimized.

3.5.5 Memory Space Overhead

The memory space overhead of the software schemes mainly comes from the per-address per-thread meta-data such as `PrevCSList` and `PrevVClk`. Specifically, our scheme maintains a 32-bit critical section ID (16-bit mutex object ID and 16-bit timestamp for `lock()`), and a 16-bit timestamp for the most recent read and write to each location, which represents a 12-byte overhead per thread per byte for a byte-addressable system. For `OSCS-Base` with 8 threads, the space overhead is 96 times of the original program's memory footprint. On the other hand, `OSCS-Opt` has a much lower memory space usage thanks to the optimizations that only keeps the history of only two recent accesses for shared memory locations. We found that the memory space usage is roughly 2-3x. `OSCS-HW` has no per-address overhead because the meta-data is only kept in

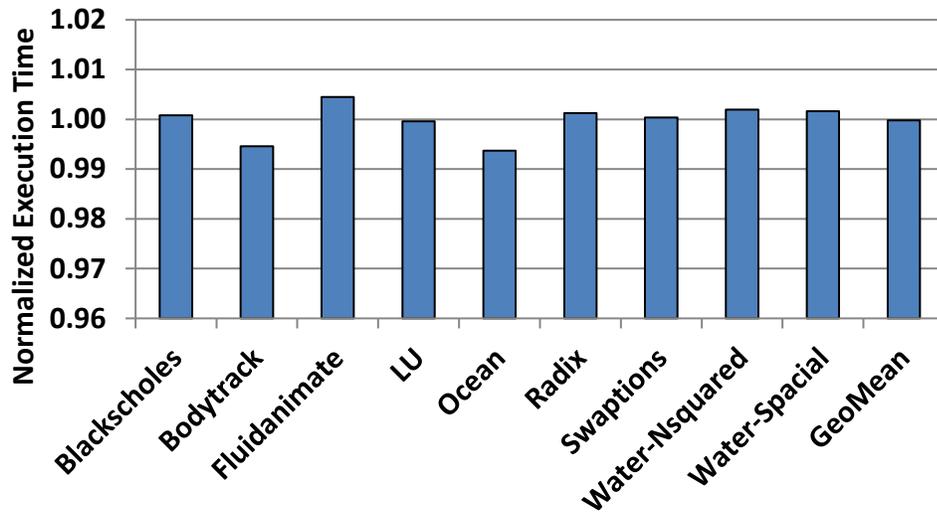


Figure 3.15: Performance overhead normalized to native execution.

the AHB.

In addition to the per-address meta-data, all three OCS schemes require a vector clock for each thread and each ordering synchronization object. Each vector clock has N_{thread} timestamps where N_{thread} is the number of threads. However, the space overhead of these vector clocks is rather small because the number of threads and the number of ordering synchronization objects are small. In our experiments, the number of ordering objects never exceeded 10. Even with 100 threads and 100 ordering synchronization objects, the vector clocks only require 2MB for 16-bit timestamps.

3.5.6 Hardware Evaluation

Hardware Performance Overhead

Figure 2.11 shows the normalized execution time for `OSCS-HW`. The performance overhead is negligible on average. In the worst case, the overhead is 0.5% for `Fluidanimate`. Because the architecture mostly uses dedicated on-chip structures such as 1-bit cache tags and the AHB for bookkeeping, there are only two main sources of overhead for our implementation: the communication among AHBs for coherency and vector clock accesses through the normal memory hierarchy for ordering synchronization objects. However, because the AHB is only accessed on a shared memory access within a critical section, we found that AHB accesses rather infrequent (1.7% in worst case for `Fluidanimate` and 0.23% on average). The dynamic power consumption of AHB based on CACTI [50] is estimated to be 9mW on average. Similarly, the number of vector clock accesses are negligible when compared to the number of regular data accesses (0.07% in the worst case for `Fluidanimate` and only 0.003% on average). While counter overflows may also introduce performance overhead by requiring timestamps and vector clocks to reset, the timestamp values never exceeded a few hundreds after the entire execution for our benchmarks. As a result, the overall performance impact is minimal for `OSCS-HW`.

We note that in certain cases (`Bodytrack` and `LU`), the experiments show a tiny speedup for `OSCS-HW`. This is because additional vector clock accesses change the cache access pattern, which resulted in a slightly lower cache miss-rate. The speedup is quite small and less than 0.5%.

Cache and AHB Size Analyses

Similar to the `RaceSMM-HW` scheme, `OSCS-HW` also relies on data caches to store the “share bit” for each cache block. Hence, variation in cache sizes would have the same impact on detection capability of `OSCS-HW` as it has on `RaceSMM-HW`. We experimented on reducing the cache sizes to 1/2 (i.e. 16KB L1, 128KB L2, 4MB L3) and 1/4 (i.e. 8KB L1, 64KB L2, 2MB L3) of the baseline configuration. As expected, the detection rate decreases as the cache sizes decrease. We found that the detection coverage dropped to 76% and 60% respectively. This finding is inline with our finding in the evaluation of `RaceSMM-HW`. On the other hand, increasing the cache sizes did not show any improvement in detection coverage.

We note that accesses to AHB in `OSCS-HW` scheme is much less frequent than in `RaceSMM-HW` as we only keep previous access histories if a shared location is accessed *and* if the access happens within a critical section. Therefore, `OSCS-HW` requires less entries in the AHB. We tested the AHB size up to 2048 entries, and found no change in detection capability beyond an AHB size of 256 entries. We have further reduced the AHB size to 128 and 64 entries to evaluated the impact of AHB size on the detection coverage of our scheme. As shown in Table 3.6, with 128 entries, the detection coverage is reduced to 78%, and with 64 entries, the detection coverage is reduced to 72%, respectively. As expected, a smaller AHB reduces the detection coverage as it would negatively affect the bookkeeping duration of previous memory accesses, thus reducing the likelihood of a detection in our scheme.

Table 3.6: AHB Size Study. (P) - PARSEC, (S) - SPLASH2, (A) - Atomicity violation, (O) - Order violation.

	Injected Bugs			
	Total	256 Entries Detected	128 Entries Detected	64 Entries Detected
Bodytrack (P)	3(A)	2(A)	2(A)	1 (A)
Fluidanimate (P)	5(A)	5(A)	5(A)	5(A)
LU (S)	8(A) 1(O)	6(A) 0(O)	5(A) 0(O)	5(A) 0(O)
Ocean (S)	14(A) 2(O)	11(A) 2(O)	11(A) 2(O)	11(A) 2(O)
Radix (S)	4(A)	4(A)	4(A)	4(A)
Water-nquare (S)	16(A) 1(O)	14(A) 1(O)	13(A) 1(O)	12(A) 0(O)
Water-spacial (S)	14(A)	12(A)	10(A)	9(A)
Total	68	57	53	49
Detection %	—	84%	78%	72%

3.5.7 Comparison to Other Schemes

Table 2.5 compares the characteristics of the `OSCS` scheme with three state-of-the-art non-race bug detection schemes. All four schemes are based on heuristics and as a result have false positives and negatives.

SVD [75] and AVIO [43] both detect atomicity violations by approximating intended atomic regions from common program behaviors. AVIO has a high coverage in detecting atomicity violations, but requires training runs. On the other hand, SVD has a lower coverage, only detecting 1 out of 6 bugs studied in AVIO [43]. SVD and AVIO cannot detect ordering violations or bugs with multiple variables. The `OSCS` scheme detects all 6 out of 6 bugs studied by AVIO, without training, including a multi-variable bug that AVIO could not detect.

Table 3.7: Comparisons to other non-race bug detection schemes. (AV - Atomicity Violation; OV - Ordering Violation; MV - Multivariable Violation)

	SVD [75]	AVIO [43]	Bugaboo [44]	OSCS
Training Runs	No	Yes Required	Yes for low false positives	No
Detect AV	Yes	Yes	Yes	Yes
Detect OV	No	No	Yes	Yes
Detect MV	No	No	Yes	Yes*
Detection coverage	Low-Med	High	High	90% (SW) / 84% (HW)
False Negative	Yes	Yes	Yes	Yes
False Positive	Yes	Yes	Yes	Yes
SW Slowdowns	65X	25X	15X-5025X	300X
HW Slowdowns	N/A	Negligible	Negligible	Negligible

Bugaboo [44] provides a coverage for atomicity and ordering violations, and can detect both single and multi-variable bugs. However, it requires multiple training runs with explicit labeling of correct and incorrect runs to lower false positives. Even then, Bugaboo reports higher false positives than OS CS (1 true bug from 8 detections).

The software implementation slowdowns of the four schemes are comparable, especially considering the slowdowns of Bugaboo and OS CS include the overhead of the Pin framework. The hardware implementation slowdowns of the schemes are also comparable (SVD does not have a HW implementation) as there are negligible slowdowns in all schemes. However, the hardware slowdowns of AVIO and Bugaboo do not include the time required for training runs, which is not required in OS CS. Overall, compared to the state-of-the-art, the pro-

posed OCS scheme provides high detection coverage for both atomicity and ordering violations with low false positives without requiring program-specific training runs

CHAPTER 4
EFFICIENT DETERMINISTIC REPLAY EXECUTION THROUGH
COMMUTATIVE CRITICAL SECTIONS

4.1 Introduction

In this chapter, we introduce an efficient deterministic replay scheme for multithreaded programs based on a concept of *commutative critical sections*. At a high level, the commutative critical sections can be described as critical sections in a multithreaded program that can be executed in any order while producing a consistent program output. Here, we introduce a definition for commutative critical sections and an approach to systematically identify such commutative critical sections.

We further propose a deterministic replay scheme, named `CommuteReplay`, which allows the commutative critical sections to execute without enforcing an explicit deterministic order between the commutative critical sections on a replay. The proposed scheme maintains a deterministic replay property of always producing the same output from the same input on a replay (i.e. external determinism), which has been shown to be valuable in debugging and testing for multithreaded programs [2, 40]. In all three of the hardware configurations tested, evaluation results suggested that the proposed scheme eliminates most, if not all, of the replay overhead in stalling or logging for a deterministic ordering between critical sections. The proposed scheme reduced more than 20% of the overall performance overhead on average. Overall, our proposed scheme allows a more efficient deterministic replay execution.

The rest of the chapter is organized as follows. Section 4.2 presents a formal definition and an approach to identify for the commutative critical sections. Then, Section 4.3 describes how a deterministic replay scheme can be implemented to allow an efficient replay execution by leveraging the concept of the commutative critical sections. Section 4.4 evaluates the proposed deterministic replay scheme in terms of the performance overheads.

4.2 Commutative Critical Section

In this section, we first present an overview of the commutative critical sections followed by a formal definition and the terminologies that are used to describe the commutative critical sections. We then discuss a methodical way to check for such commutative critical sections. Finally, we demonstrate examples of identifying both commutative and non-commutative critical sections.

4.2.1 Overview

At a high level, commutative critical sections can be viewed as critical sections in a multithreaded program that can be executed in any order while still maintaining a consistent and deterministic program output by the end of the program's execution. For example, as shown in Figure 4.1, a critical section is executed in each thread to calculate a *global_max* from multiple *local_max* values. In this example, the value of the *global_max* variable is used as the program output by the end of the execution. Here, there are two critical sections that are being executed by Thread 1 and 2, namely CS_1 and CS_2 . The value of *global_max* remains

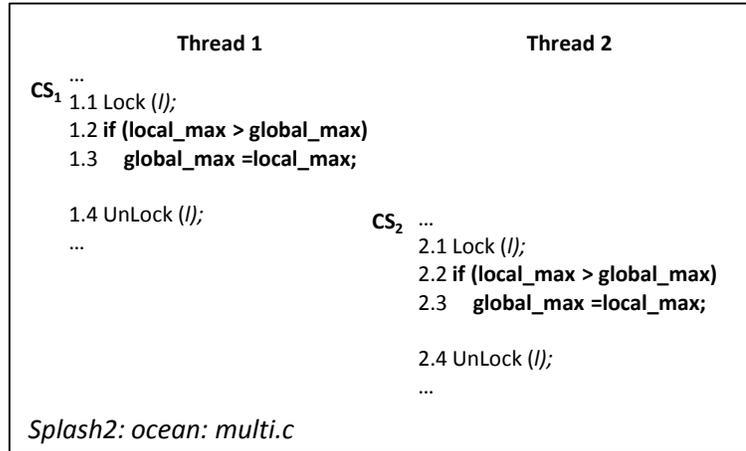


Figure 4.1: An abstract example from the Ocean (Splash2) benchmark in which a global max is computed from multiple local max values.

consistent regardless of the execution order between CS_1 and CS_2 . Hence, CS_1 and CS_2 are conceptually “commutative” with respect to the output value of *global_max*. Therefore, it is possible to allow certain critical sections (such as the ones in Figure 4.1), which are commutative with respect to the program output, to run in any order while maintaining the same output from the same input on a replay. Recall that the characteristic of always producing deterministic output is called *external determinism* and it has been shown to be valuable in debugging and testing multithreaded programs [2, 40]. In the rest of this chapter, unless otherwise specified, we will use the term *deterministic replay* to refer to a replay mechanism that guarantees external determinism.

Figure 4.2 illustrates the potential overhead that can be reduced by allowing the commutative critical sections to execute in any order on a replay. In a traditional deterministic replay scheme, such as the one described in Respec [40], the ordering between critical sections are strictly enforced based on a specific recorded order. However, it is possible for Thread 1’s execution to be slower

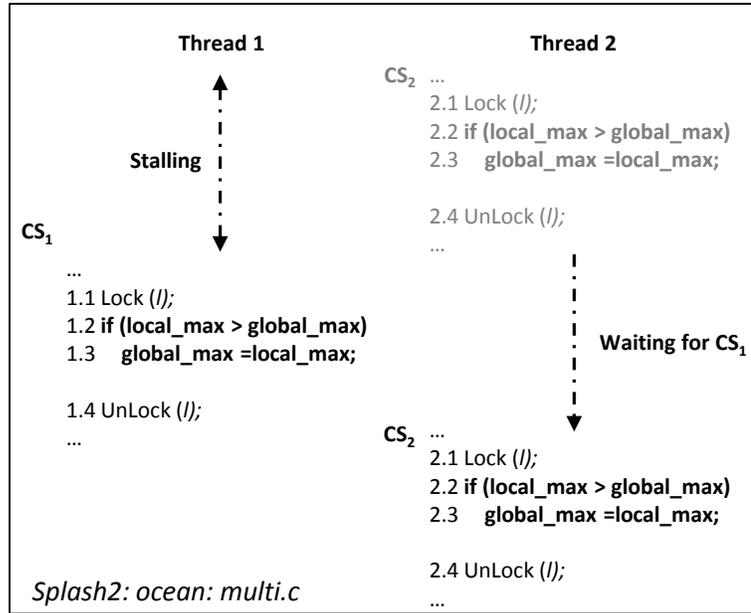


Figure 4.2: The critical section in Thread 1 is recorded to execute before the critical section in Thread 2. On a replay through a traditional deterministic replay scheme, the execution of the critical section in Thread 1 is stalled due to a different execution environment, which results in the overall execution stalling as the execution of the critical section in Thread 2 needs to wait for Thread 1’s critical section to finish its execution.

than Thread 2’s due to a changed computing environment on a replay and thus causing unnecessary delays in the overall execution. For example, it is possible that Thread 1 is executed on a slower core than Thread 2’s on a replay. In such a case, Thread 2’s execution would need to stall upon reaching CS_2 until CS_1 finishes execution. On the other hand, if the critical sections are identified to be commutative and thus are allowed to be executed in any order, CS_2 can execute before CS_1 to avoid the stalling shown in the diagram.

In the rest of this section, we discuss the definition and the identification of commutative critical sections.

4.2.2 Definition

Let $\langle CS \rangle$ denotes a *static critical section* which we define as a segment of code, which starts with a lock acquire operation and ends with a lock release operation. The $\langle CS \rangle$ is a set of operations which maps a program state S to a new program state S' .

We further use CS_i to denote a *dynamic instance of a critical section* which we define as the set of operations executed by a specific instance i of $\langle CS \rangle$.

With this notation, we hereby define a notion of commutativity for critical sections:

Definition 1 A static critical section $\langle CS \rangle$ is defined as commutative if for any number of dynamic instances of $\langle CS \rangle$ (CS_1, \dots, CS_n) and for all possible starting system state S , the system state S' after applying these dynamic instances is the same for all possible execution orders.

Definition 2 For dynamic critical section instances CS_1 and CS_2 , if $CS_1(CS_2(S)) = CS_2(CS_1(S))$ for all possible system state S , then CS_1 and CS_2 are *commutative*.

Note, in a multithreaded execution, the execution order within each thread remains sequential while the inter-thread execution order is nondeterministic. Therefore, only the dynamic instances of $\langle CS \rangle$ from different threads can have a nondeterministic execution order.

Theorem 1 If any two dynamic instances CS_i and CS_j of $\langle CS \rangle$ are commutative where CS_i and CS_j are from two different threads, then $\langle CS \rangle$ is commutative.

Proof 1 *Since any pair of dynamic instances CS_i and CS_j are commutative ($\forall i, j \in \{1, \dots, m\}, i \neq j$), we can consider a subset of dynamic instances of $\langle CS \rangle$, say the first k dynamic instances CS_1, \dots, CS_k without loss of generality.*

For any given system state S , $CS_1(CS_2(\dots CS_k(S))) = S'$. Here, we can achieve any other execution order permutation of $(1, \dots, k)$ by performing consecutive pairwise swaps of CS_i and CS_j (as long as the pair is from different threads) until the desired reordering is achieved without affecting the output system state S' . This is true for any number or choice of dynamic instances of $\langle CS \rangle$.

Hence, $\langle CS \rangle$ is commutative by definition. ■

We note that the above definition of commutative critical sections is conservative and only covers a subset of all potential commutative critical sections. Namely, we only focus on a single static critical section and determine whether the dynamic instances of such critical section are commutative. However, our evaluation results demonstrate that in practice this definition works well as it identifies nearly all commutative critical sections that could cause unnecessary execution delays in a traditional replay scheme.

4.2.3 Terminology

In order to discuss commutative critical sections and a method to identify them, we first introduce the terminologies and notations used in the following discussion.

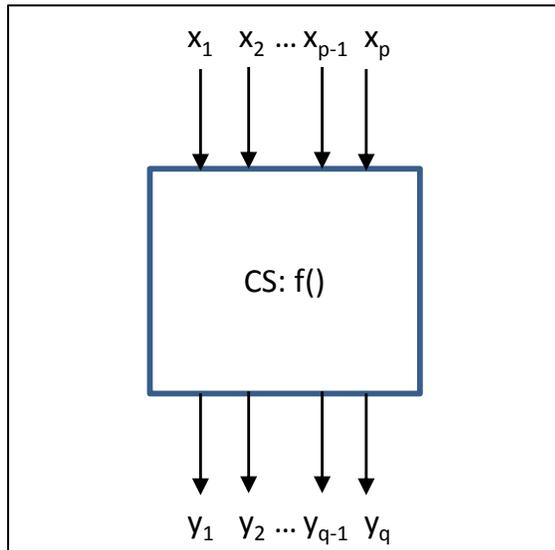
We introduce the *input set*, X , as the set of state elements that can be read by a $\langle CS \rangle$, and the *output set*, Y , as the set of state elements that can be modified

by a $\langle CS \rangle$. We hereby consider the X and Y to be variables that can be read and/or written during the execution of a $\langle CS \rangle$ in the rest of our discussion. In particular, we have X as the set of input variables, and Y as the set of output variables. For example, in Figure 4.1, both *local_max* and *global_max* are input variables as they are read in the critical section, and *global_max* is an output variable as it can be written in the critical section. We note that we only consider a variable to be in Y if the newly written value remains *live* after the execution of $\langle CS \rangle$. Here, we define a variable to be *live* if the modified value in a variable is read before the variable is written again after the execution of $\langle CS \rangle$.

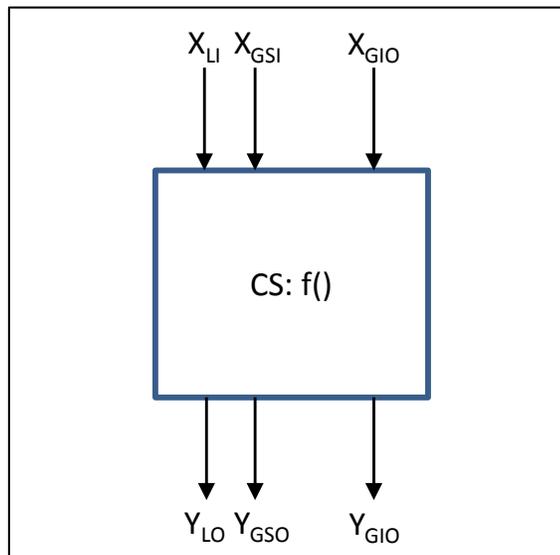
With the input and output elements identified, we are able to describe a $\langle CS \rangle$ as a function, named $f()$, which takes in the inputs from X and produces outputs belonging to Y . Conceptually, the function $f()$ can be described using a data-flow diagram to represent the code segment $\langle CS \rangle$, where input elements flow into $f()$ and output elements flow out of $f()$. To discuss the $f()$ in the most general sense, we say $\{x_1, x_2, \dots, x_p\} \in X$ are the input variables to $f()$, and $\{y_1, y_2, \dots, y_q\} \in Y$ are the output variables from $f()$, as shown in Figure 4.3(a).

For each variable in X and Y , we can categorize them as either *local* or *global* variables by determining whether they can be shared among instances of $\langle CS \rangle$ between threads. Namely, X_{LI} and Y_{LO} are used to denote local input and output variables, and X_{GI} and Y_{GO} are used to denote global input and output variables. For example, in Figure 4.1, *local_max* is a local variable and *global_max* is a global variable. Note that we have following:

$$\begin{aligned} X &= X_{LI} \cup X_{GI} \\ Y &= Y_{LO} \cup Y_{GO} \end{aligned} \tag{4.1}$$



(a) $f()$ with inputs in set X and outputs in set Y



(b) $f()$ with input vectors of $x_{Li}/x_{GSI}/x_{GIO}$ and output vectors $y_{Li}/y_{GSO}/y_{GIO}$

Figure 4.3: The $\langle CS \rangle$ can be described as a function $f()$, which takes input set X and produces output set Y . To simplify our discussion, we categorize inputs and outputs into three categories and use input vectors $x_{Li}/x_{GSI}/x_{GIO}$ and output vectors $y_{Li}/y_{GSO}/y_{GIO}$ to represent the input elements.

From sets X_{GI} and Y_{GO} , we further define X_{GSI} (Global Strict Input) to be the set of global variables that are only read by a $\langle CS \rangle$, Y_{GSO} (Global Strict Output) to be the set of global variables that are only written by a $\langle CS \rangle$, and X_{GIO} and Y_{GIO} (Global Input and Output) to be the set of global variables that are both read and written by a $\langle CS \rangle$. We note that X_{GIO} and Y_{GIO} are equivalent sets, and are both used for the purpose of simplifying the rest of the discussion here. For example, in Figure 4.1, *local_max* is in X_{LI} and *global_max* is in X_{GIO} . Thus we have:

$$\begin{aligned} X_{GSI} &= X_{GI} \setminus Y_{GO} \text{ and } X_{GIO} = Y_{GIO} = X_{GI} \cap Y_{GO} \\ X_{GI} &= X_{GSI} \cup X_{GIO} \text{ and } X_{GO} = X_{GSO} \cup X_{GIO} \end{aligned} \quad (4.2)$$

Figure 4.3(b) shows that the input and output elements from X and Y can be further categorized as the follows:

$$\begin{aligned} \{x_1, x_2, \dots, x_k\} &\in X_{LI} \\ \{x_{k+1}, x_{k+2}, \dots, x_l\} &\in X_{GSI} \\ \{x_{l+1}, x_{l+2}, \dots, x_p\} &\in X_{GIO} \\ \{y_1, y_2, \dots, y_m\} &\in Y_{LO} \\ \{y_{m+1}, y_{m+2}, \dots, y_n\} &\in Y_{GSO} \\ \{y_{n+1}, y_{n+2}, \dots, y_q\} &\in Y_{GIO} \end{aligned} \quad (4.3)$$

Hence, we can use X_{LI}, X_{GSI}, X_{GIO} to represent the input variables, and Y_{LI}, Y_{GSO}, Y_{GIO} to represent output variables, respectively. The function $f()$ can be described as:

$$(Y_{LO}, Y_{GSO}, Y_{GIO}) = f(X_{LI}, X_{GSI}, X_{GIO}) \quad (4.4)$$

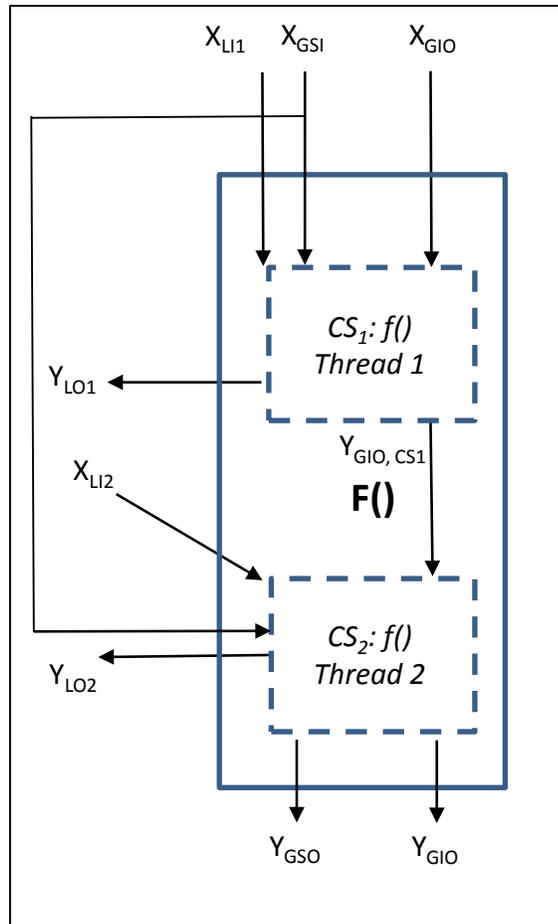


Figure 4.4: Two instances of $f()$ are concatenated together to form $F()$. Note that any intermediate output values of $Y_{GIO,CS1}$ from CS_1 in Thread 1 are subsequently used by CS_2 in Thread 2 as input values, and hence are not part of outputs from $F()$. Also note that only one set of X_{GSI} and one set of Y_{GSO} are shown for $F()$ since both CS_1 and CS_2 share the same set of X_{GSI} and Y_{GSO} .

Recall that we have defined a commutative critical section using multiple dynamic instances of $\langle CS \rangle$ previously. Figure 4.4 shows that it is possible to concatenate two $f()$ functions and represent two critical section instances, say CS_1 and CS_2 , that are executed in two different threads as a $F()$ function.

Since CS_1 and CS_2 are dynamic instances of the same $\langle CS \rangle$ code segment,

the two instances have the same $f()$ functions as shown in Figure 4.3. It implies that the sets of variables that are used in both CS_1 and CS_2 have the same number of elements, and the two critical section instances also share the memory locations for elements in their global variable sets. For example, for a global array that is used in both CS_1 and CS_2 , while it is possible that the two instances each access a separate part of the array (i.e. CS_1 accesses all odd elements, and CS_2 accesses all even elements), all elements of the array are still considered to be in the global variable set for both instances. Furthermore, we note that only one set of X_{GSI} and one set of Y_{GSO} are shown for $F()$ since both CS_1 and CS_2 share the same set of X_{GSI} and Y_{GSO} . Moreover, we note that any intermediate output values of Y_{GIO} from CS_1 (i.e. Y_{GIO,CS_1} in Figure 4.4) are subsequently used by CS_2 as input values, and hence are not a part of outputs from $F()$. Overall, the $F()$ shown here can be described as the following:

$$(Y_{LO1}, Y_{LO2}, Y_{GSO}, Y_{GIO}) = F(X_{LI1}, X_{GSI}, X_{GIO}, X_{LI2}) \quad (4.5)$$

In the rest of this section, we will use $F()$ to describe the operations done by two concatenated $\langle CS \rangle$ instances.

4.2.4 Commutative CS Checking Scheme

Here, we discuss the identification of a commutative critical section. We first discuss our assumptions followed by the identification method in detail.

In order to identify a commutative critical section, we need to use Theorem 1

which requires us to examine whether any two $\langle CS \rangle$ instances from two different threads are commutative. In the rest of the discussion, we use CS_1 and CS_2 to describe our checking mechanism, where CS_1 and CS_2 are two arbitrary dynamic instances of a critical section $\langle CS \rangle$ that are executed in two different threads.

Assumptions

The following assumptions are required by our checking method to provide a systematic way of identifying the commutative critical section. Under the assumptions discussed here, we can limit our checking mechanism to only examining the code segment of a critical section $\langle CS \rangle$ to determine whether the $\langle CS \rangle$ is commutative, without the need to consider other program operations outside of the $\langle CS \rangle$.

1. Other than CS_1 and CS_2 , the rest of the program is executed in a deterministic fashion. In particular, all memory accesses (other than the accesses within $\langle CS \rangle$) are executed deterministically.
2. During the execution of CS_1 and CS_2 , there is no conflicting access to $x, \forall x \in X$ and $y, \forall y \in Y$ outside of CS_1 and CS_2 . Recall we define conflicting accesses as accesses from different threads to the same memory location, which include at least one write.
3. There is no memory aliasing between variables used in $\langle CS \rangle$.
4. There is no user visible output produced within $\langle CS \rangle$.

Under Assumption 1, all memory accesses are executed deterministically outside the execution of CS_1 and CS_2 . We can thereby assume any input vari-

able (in X) used by $F()$ has a deterministic value regardless of the execution order of CS_1 and CS_2 before the execution of $F()$. This also implies that if $\langle CS \rangle$ is commutative, then the values of all output variables would remain deterministic after the execution of $F()$ and the entire program execution would remain deterministic.

However, the values of variables used by $F()$ can still be accessed by other parts of the program during the execution of CS_1 and CS_2 . Hence, by only examining CS_1 and CS_2 in a different execution order, we cannot provide any guarantee of the determinism of the entire program execution unless we can isolate the execution of CS_1 and CS_2 from the rest of the program's execution. Assumption 2 provides such an isolation guarantee since we assume there is no other conflicting accesses from the rest of the program execution other than those within CS_1 and CS_2 . By combining Assumption 1 and Assumption 2, we can deduce that the value of each input variable used by $F()$, unless changed within CS_1 and CS_2 , would remain the same both before and during the execution of $F()$ regardless of the execution order of CS_1 and CS_2 .

So far in our discussion, we implicitly assume that we can distinguish between different input and output variables in $\langle CS \rangle$. However, in the case of memory aliasing, it is possible to have multiple variables occupying the same location, and such aliasing can only be identified by examining the overall program's memory mapping. Under Assumption 3, it is then possible to distinctively identify input and output variables by only examining the $\langle CS \rangle$ code region.

Finally, we assume that there is no user visible output (such as system out calls) during the execution of the instances of $\langle CS \rangle$ in Assumption 4. If user

visible outputs are being produced within the critical section, then the order of the outputs, as well as the output values could become non-deterministic during the execution of CS_1 and CS_2 as we allow multiple execution orders. Under this assumption, we can only focus on the program state after the execution of CS_1 and CS_2 .

Algorithm

With the aforementioned assumptions in mind, a checking algorithm for identifying commutative critical sections is discussed here.

For any two arbitrary dynamic instances of $\langle CS \rangle$, denoted as CS_1 and CS_2 , we consider two execution orders between them. Namely, let *order 1* be $CS_1 \rightarrow CS_2$ and *order 2* be $CS_2 \rightarrow CS_1$ as shown in Figure 4.5. In particular, Figure 4.5 shows that the difference between the two execution orders is the order of variable values fed into $F()$.

Recall that by combining Assumption 1 and Assumption 2, we can deduce that the value of each input variable used by $F()$, unless changed within CS_1 and CS_2 , would remain the same both before and during the execution of $F()$ regardless of the execution order of CS_1 and CS_2 . More specifically, the value of X_{GIO} in order 1 is the same as the value of X_{GIO} in order 2, and X_{LI} and X_{GSI} to each $\langle CS \rangle$ instance has the same value between the two execution orders. We also note that since variables in X_{GSI} are only used as inputs and are shared between CS_1 and CS_2 , the same values of X_{GSI} variables are used in both $\langle CS \rangle$

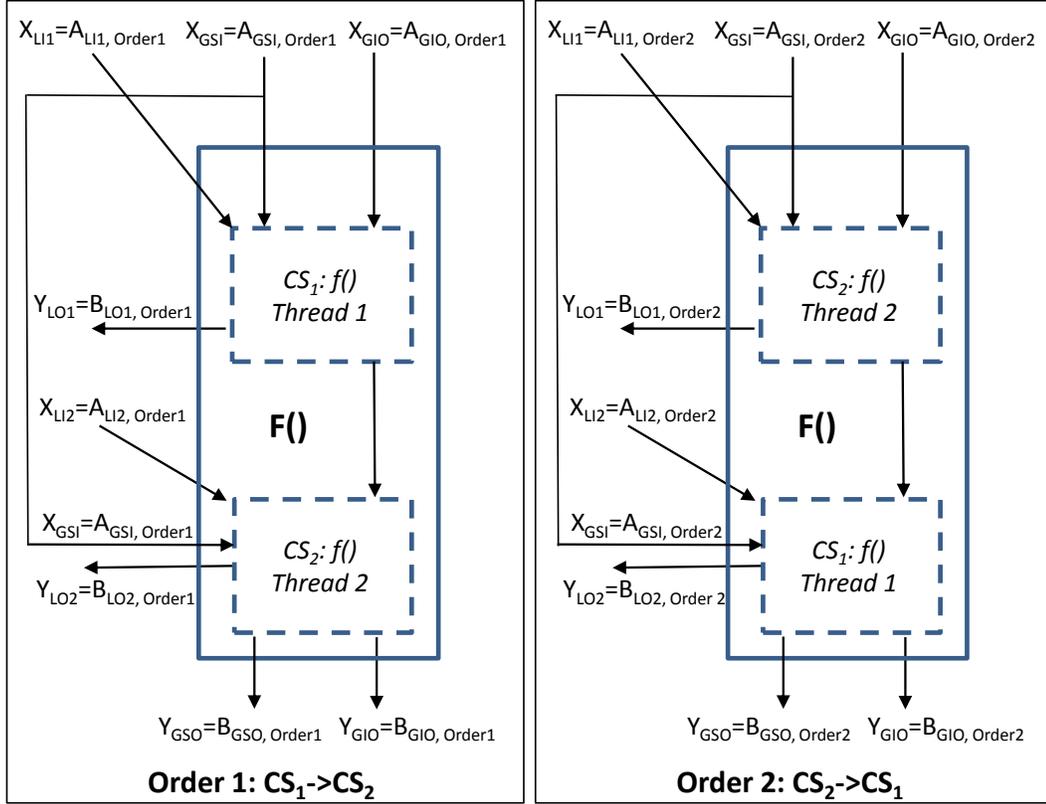


Figure 4.5: Two orders of execution between CS_1 and CS_2 . The difference here is the order of the variable values fed into $F()$ in each execution order. The output values can also be different between the two orders.

instances. In Figure 4.5, we use A to denote input values, and thus we have:

$$\begin{aligned}
 A_{LI1,Order1} &= A_{LI2,Order2} = A_{LI,T1} \\
 A_{LI2,Order1} &= A_{LI1,Order2} = A_{LI,T2} \\
 A_{GSI,Order1} &= A_{GSI,Order2} = A_{GSI} \\
 A_{GIO,Order1} &= A_{GIO,Order2} = A_{GIO}
 \end{aligned} \tag{4.6}$$

As shown in Figure 4.5, the only difference between the two execution orders is the order of the local input values used by the $F()$ function. Here, we use A to denote input values, and B to denote output values, and the $F()$ functions for

the two execution orders are as follows:

$$\begin{aligned}
 \text{Order1} : (B_{LO1,Order1}, B_{LO2,Order1}, B_{GSO,Order1}, B_{GIO,Order1}) = \\
 F(A_{LI,T1}, A_{GSI}, A_{GIO}, A_{LI,T2}) \\
 \text{Order2} : (B_{LO1,Order2}, B_{LO2,Order2}, B_{GSO,Order2}, B_{GIO,Order2}) = \\
 F(A_{LI,T2}, A_{GSI}, A_{GIO}, A_{LI,T1})
 \end{aligned} \tag{4.7}$$

Using Equation 4.7, the $\langle CS \rangle$ is commutative when the outputs from both execution orders are the same for any input values, namely:

$$\begin{aligned}
 B_{LO1,Order1} &= B_{LO2,Order2} \\
 B_{LO2,Order1} &= B_{LO1,Order2} \\
 B_{GSO,Order1} &= B_{GSO,Order2} \\
 B_{GIO,Order1} &= B_{GIO,Order2}
 \end{aligned} \tag{4.8}$$

In other words, the CS_1 and CS_2 are commutative if and only if the local output values from each $\langle CS \rangle$ instance are the same and the global output values at the end of the execution of the two critical section instances are the same for any input values between the two execution orders.

The checking of commutative critical sections can be handled automatically by using existing symbolic execution techniques. Symbolic execution is an analysis technique that evaluates a computer program through symbolic values (instead of actual numeric values) used in the program [34]. For example, a recently proposed symbolic execution technique, named KLEE [14], is able to check the equivalence of two functions in a computer program by analyzing

the execution paths and comparing the symbolic output values of the functions given the symbolic input values to both functions. Essentially, given the different symbolic input values to the two $F()$ functions as shown in Equation 4.7, a commutative critical section can be identified by checking the equivalence of the symbolic output values between the two $F()$ functions as shown in Equation 4.8.

Checking Examples

Here, we demonstrate the checking algorithm discussed earlier using critical section code segments from real world programs.

```

1.1 Lock (l);
1.2 if (local_err > multi->err_multi)
    {
1.3   multi->err_multi=local_err;
    }
1.4 UnLock (l);
Splash2: ocean: multi.c
```

Figure 4.6: An example, $\langle CS_{ocean} \rangle$, from the Ocean (Splash2) benchmark. $\langle CS_{ocean} \rangle$ has a global max that is computed from multiple local max values.

Figure 4.6 shows an example, $\langle CS_{ocean} \rangle$, from *Ocean* where a global max value (i.e. $multi \rightarrow err_multi$) is computed by comparing with multiple local max values in each $\langle CS_{ocean} \rangle$ instance. In this example, $(local_err, multi \rightarrow err_multi) \in X$ and $(multi \rightarrow err_multi) \in Y$.

Let us consider CS_{ocean1} and CS_{ocean2} , two arbitrary instances of the $\langle CS_{ocean} \rangle$ in Figure 4.6 that are executed in different threads. Figure 4.7 shows CS_{ocean1} and CS_{ocean2} in two different execution orders. Recall that under our as-

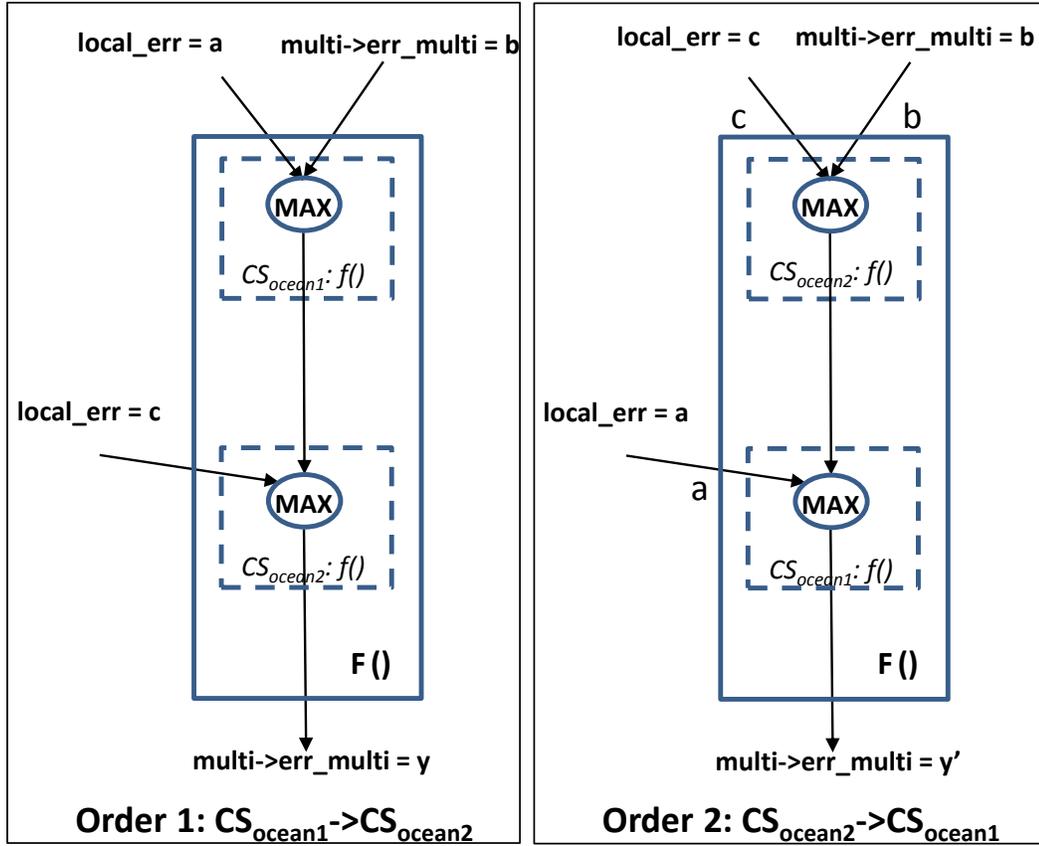


Figure 4.7: Two $\langle CS_{ocean} \rangle$ instances from the Ocean benchmark, CS_{ocean1} and CS_{ocean2} , are shown with different execution orders. In each $\langle CS_{ocean} \rangle$ instance, a MAX operation is performed with the two inputs, and the output of the MAX operation is the only output of the $\langle CS_{ocean} \rangle$ instance.

sumptions, the value of $local_err$ to CS_{ocean1} and CS_{ocean2} are the same between the two execution orders, and we use a and c to denote the value of $local_err$ to CS_{ocean1} and CS_{ocean2} . Also, the values of $multi \rightarrow err_multi$ before the execution of $F()$ functions are the same in both orders, and it is denoted as b in the figure. The $F()$ functions in both execution orders are as follows:

$$Order1 : (y) = F(a, b, c)$$

$$Order2 : (y') = F(c, b, a)$$

In this case, $y = \text{MAX}(c, \text{MAX}(a, b))$ and $y' = \text{MAX}(a, \text{MAX}(c, b))$, where $y = y' = \text{MAX}(a, b, c)$. Hence we can conclude that the $\langle CS_{ocean} \rangle$ in Figure 4.6 is commutative.

```

1.1 lock(l);
1.2 ProcID = global_thread_index++;
1.3 unlock(l);
Splash2: water-spatial: water.c
```

Figure 4.8: An example, $\langle CS_{ws} \rangle$, from the Water-Spatial (Splash2) benchmark. $\langle CS_{ws} \rangle$ has a local thread ID that is assigned from a rolling global thread counter.

Figure 4.8 shows another example, $\langle CS_{ws} \rangle$, from *Water-Spatial* where a local thread ID variable (i.e. *ProcID*) is assigned by a rolling global thread counter (i.e. *global_thread_index*). In this example, $(\text{global_thread_index}) \in X$ and $(\text{ProcID}, \text{global_thread_index}) \in Y$.

Again, let us consider CS_{ws1} and CS_{ws2} , two arbitrary instances of the $\langle CS_{ws} \rangle$ in Figure 4.8 that are executed in different threads. Figure 4.9 shows CS_{ws1} and CS_{ws2} in two different execution orders. Recall that under our assumptions, the value of *global_thread_index* before the execution of $F()$ functions is the same between the two execution orders, and we use a to denote the value of *global_thread_index*. The outputs of the $F()$ function in both execution orders are as follows:

$$\text{Order1} : (b, c, y) = F(a)$$

$$\text{Order2} : (c', b', y') = F(a)$$

In this case, however, $b \neq b'$ and $c \neq c'$ as $b = a$; $c = a + 1$; $b' = a + 1$; $c' = a$. Hence, the $\langle CS_{ws} \rangle$ in Figure 4.8 is not commutative. We note that although the input

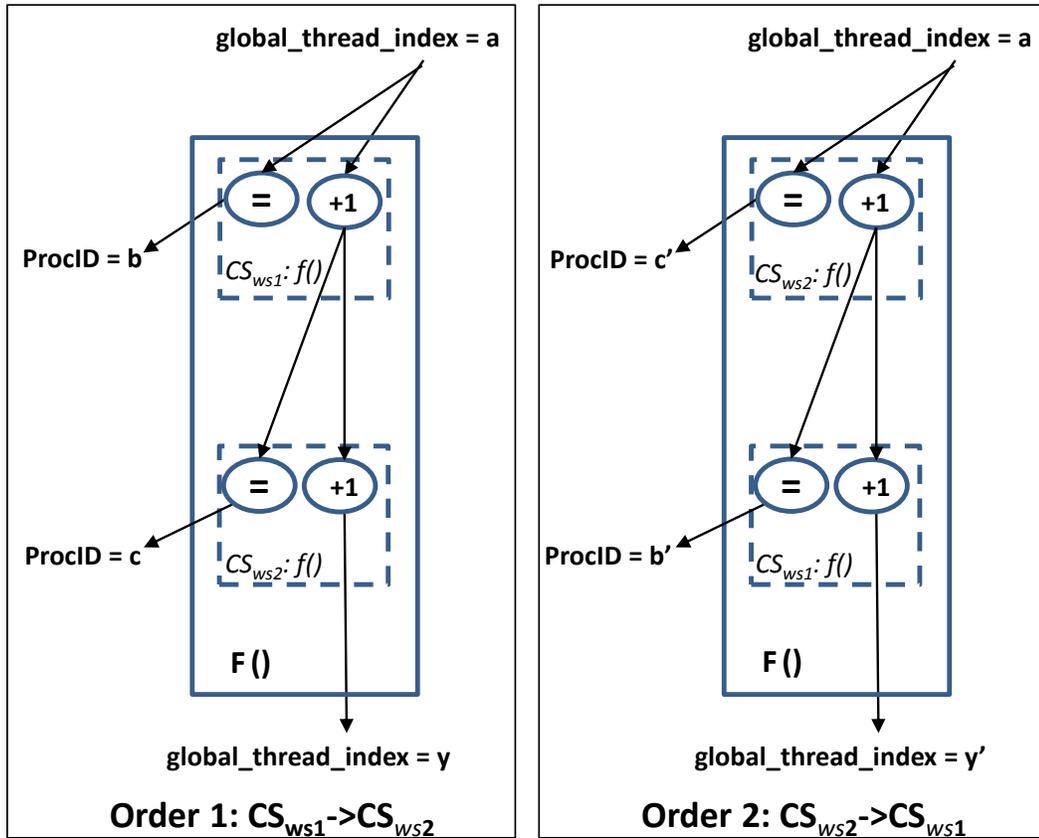


Figure 4.9: Two $\langle CS_{ws} \rangle$ instances from the Water-Spatial benchmark, CS_{ws1} and CS_{ws2} , are shown with different execution orders. In each $\langle CS_{ws} \rangle$ instance, the value of $global_thread_index$ is first assigned to $ProcID$, and $global_thread_index$ is then incremented by 1. Both $ProcID$ and $global_thread_index$ are outputs of a $\langle CS_{ws} \rangle$ instance.

value to $F()$ is exactly the same in both orders, the local output variable $ProcID$ from each $\langle CS_{ws} \rangle$ instance has different values in different execution orders and thus the $\langle CS_{ws} \rangle$ is not commutative.

4.3 Deterministic Replay Scheme with Commutative CS

In this section, we discuss an offline deterministic replay scheme that leverages the concept of the commutative critical sections by allowing commutative critical sections to flexibly execute without enforcing a deterministic order on a replay.

Here, an *offline* replay refers to a replay execution that is performed after the original execution of a program. Moreover, an offline replay scheme allows more than one replay execution based on the same recorded execution. An offline replay scheme is often used for testing and debugging [25, 65]. On the other hand, an *online* replay refers to a replay execution that is performed in parallel with the original execution of a program. An online replay scheme is often used for fault toleration and bug avoidance [40, 63]. While we focus our discussion on an offline deterministic replay scheme in this chapter, we believe the principle of allowing commutative critical sections to execute with more flexibility on a replay can be extended and adapted to an online deterministic replay scheme as well.

4.3.1 Overview

In the rest of this section, we discuss a deterministic replay scheme that aims to achieve *external determinism*. A program's execution is externally deterministic if the program always produces the same user visible output and final program state (i.e. memory and register contents) for a given input from run to run. Here, we define *visible output* as program values sent to devices such as a

screen, network, or disk, etc. In other words, external determinism guarantees that anything that is user observable (e.g. from an outside user’s point of view) is deterministic. There are several proposals for deterministic replay that use the external determinism approach [2, 40], as it allows more efficient implementations and it is often sufficient for testing/debugging purposes as most bugs are externally visible. For example, vast majority of software errors, including assertion violations, crashes, core dumps, and corrupted output data, generally result in externally visible failures.

The main goal of our replay scheme is to demonstrate how the commutative critical sections can be treated with more flexibility during a replay execution to allow better performance. Here, our discussion focuses on how a deterministic replay execution can be achieved for a data race free program. Essentially, our replay scheme offers the same deterministic guarantee as Respec [40] where the program’s replay execution is externally deterministic up to the first data race occurrence.

We note that there are a number of data race detection schemes [18, 27] that offer a complete data race detection coverage. Our `RaceSMM` data race detection scheme also offers high coverage and efficiency as well. Hence, a reasonably tested program can be assumed to be either completely or almost data race free in practice. Moreover, handling occasional data races can be achieved by either recording and enforcing a certain memory access order of the “racy” accesses [33], or having a fall-back mechanism that rollbacks and retries if a data race happened on a replay [2, 40]. Overall, we consider handling data races during a deterministic replay to be orthogonal to the main goal of our research, which is to demonstrate how the commutative critical sections can be treated with more

flexibility during a replay execution to allow a better overall performance.

Furthermore, existing techniques that handle data races during a replay can be added to our replay scheme as an extension. Since data races only happen occasionally in well tested programs, we believe our findings in the deterministic replay of data race free programs would still be indicative in general.

4.3.2 Identifying Commutative CS

Recall that the checking of commutative critical sections can be handled automatically by extending existing symbolic execution techniques, such as KLEE [14]. In the previous section, it is shown that for any two arbitrary instances of a critical section, two functions (one for each execution order) can be formed with different symbolic input values and the critical section is commutative if both functions produce equivalent symbolic output values. A symbolic execution technique can be used to automatically check the equivalence of the two functions by analyzing their execution paths and comparing their symbolic output values.

Additionally, recall that under Assumption 3 and Assumption 4 from Section 4.2.4, a commutative critical section cannot produce any user visible output or have any variable with memory aliasing. In a program, both memory aliasing and system calls that would produce user visible output can be automatically identified through a static analysis at the compilation time. For example, the LLVM compiler framework [37] provides alias analysis and can also identify any system output calls.

```
1.1 Lock_CCS (!);
1.2 if (local_err > multi->err_multi)
    {
1.3  multi->err_multi=local_err;
    }
1.4 UnLock_CCS (!);
```

Splash2: ocean: multi.c

Figure 4.10: Once a critical section is identified as commutative, wrapper functions of *lock_CCS()* and *unlock_CCS()* are used instead of the regular *lock()* and *unlock()* functions to indicate commutativity.

Since the focus of our work is to demonstrate the effectiveness of using commutative critical sections to reduce the performance overhead on a deterministic replay, we did not implement an automatic scheme for identifying commutative critical sections. However, as a part of our future work, we believe an automated identification scheme can be developed by extending previous symbolic execution techniques with program static analysis.

Currently, the identification of commutative critical sections is handled through human inspection (i.e. by programmer or tester, etc.) following the checking algorithm discussed in the previous section. For each tested program, we identified commutative critical sections in a similar fashion as the *Ocean* and *Water-Spatial* examples illustrated earlier.

In our current implementation, we use wrapper functions to replace the lock and unlock function calls to indicate that a critical section is commutative, as shown in Figure 4.10.

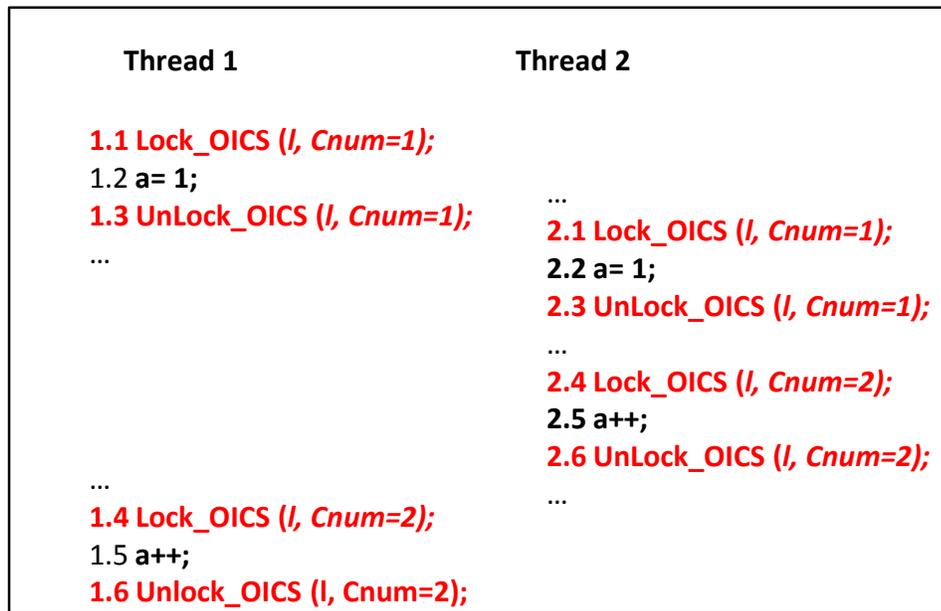


Figure 4.11: In cases where multiple commutative critical sections share a mutex object, a *Cnum* is also used as an identifier parameter to wrapper functions of *lock_CCS()* and *unlock_CCS()*.

It is also possible to have multiple commutative critical sections that share a mutex object (i.e. using a same lock variable). Therefore, we further extend the wrapper functions to include a *Cnum* as an identifier to each commutative critical section. Figure 4.11 shows two commutative critical sections in each thread, and they are distinguished by a different *Cnum* value.

4.3.3 Baseline Record and Replay Scheme

Here, we describe a baseline deterministic replay scheme, named *BaseReplay*, that provides an offline deterministic replay execution. We note that *BaseReplay* is implemented largely based on *Respec* [40], and provides the same external determinism for data race free programs. In particular, our

BaseReplay scheme records the execution order of synchronization operations that use the same synchronization object, and enforces the recorded execution order of all synchronization operations for each synchronization object on a replay.

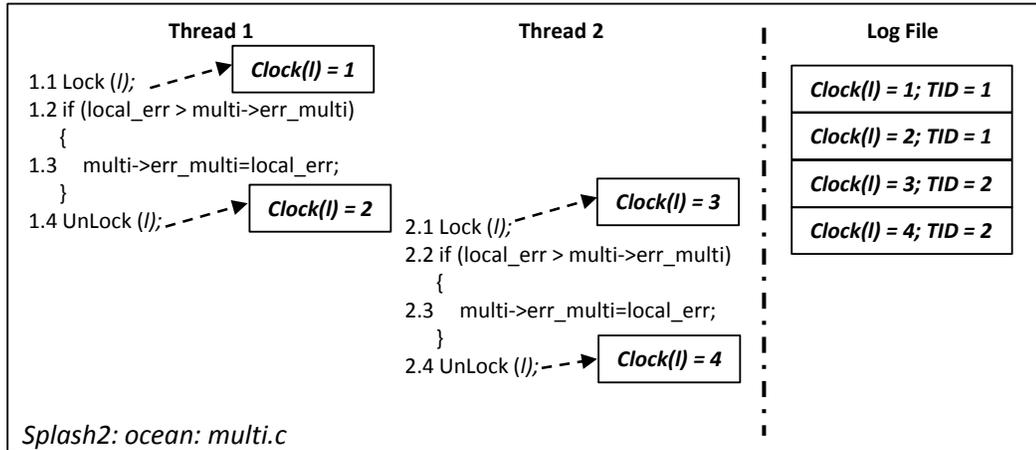


Figure 4.12: An example from Ocean, where there are two instances of a critical section, one in each thread. The clock of the mutex object is incremented on lock/unlock operations, and the clock value is recorded in a log file along with the thread ID (TID) of each mutex operation. The log file is read on a replay to enforce the recorded execution order.

In BaseReplay, conceptually, we maintain a log file and a clock for each synchronization object (i.e. $Clock[SyncObj]$). The clock and log file are used to record the execution order between synchronization operations that use the same synchronization object. To record an execution, on each synchronization operation, the clock value of the synchronization object used is atomically incremented and stored along with the operation's thread ID (TID) to a log file. Figure 4.12 shows an example of how the clock values and the TIDs are stored to a log file for two critical sections in *Ocean*. The log file is later used to achieve a deterministic replay based on the recorded execution order of the synchronization operations.

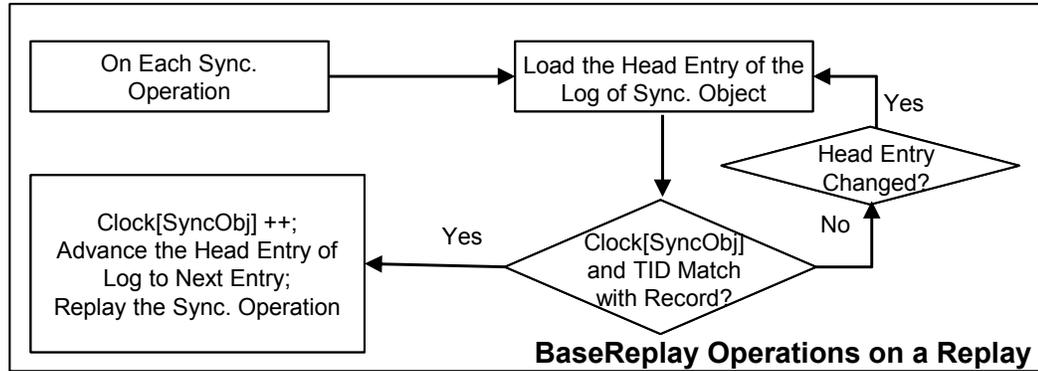


Figure 4.13: The BaseReplay operations on a replay for each synchronization operation.

Figure 4.13 shows a flowchart of operations done in `BaseReplay` on a replay. Here, a clock for each synchronization object (i.e. $Clock[SyncObj]$) is also maintained on a replay execution, and the clock values from the replay execution is compared with the recorded clock values from the log file to ensure a deterministic replay. In particular, when a replayed thread reaches a synchronization operation, it reads the head entry of the used synchronization object's log file, and stalls until the contents of the head entry of the log file match the TID of the replayed thread and the clock value of the used synchronization object. It then increments the clock value by 1, advances the head of the recorded log to the next entry, and replays the synchronization operation. In other words, a synchronization operation is stalled until all other operations that use the same synchronization object and with smaller recorded clock values are executed.

Moreover, similar to previous deterministic replay schemes [40, 65], we record the input and output of each system call, and emulate the same system calls by using the recorded values so that each system call on a replay has identical effects as those from recorded execution. We also record the execution

order of system call executions and force the replayed program to execute the calls in the same order.

Overall, assuming all synchronization operations are identified and the program is data race free, the `BaseReplay` scheme would provide external determinism between the recorded and replayed program executions.

4.3.4 Record and Replay Using Commutative CS

In order to show how to leverage the concept of commutative critical sections to reduce overheads on a replay, we propose the following record and replay scheme, named `CommuteReplay`.

We note that the only difference between `CommuteReplay` and `BaseReplay` is that we now need to handle the critical sections differently in order to allow the commutative critical sections to be executed with more flexibility on a replay. Therefore, `CommuteReplay` only changes the operations on recording and replaying of the mutex synchronization operations while leaving the rest of the scheme same as `BaseReplay`. In the rest of this subsection, we focus our discussion on the changes in the recording and replaying operations of mutex synchronization operations in `CommuteReplay`.

In `CommuteReplay`, to record an execution, we only increment the clock value of a mutex object on an acquire (i.e. `lock()`) operation if the previous release (i.e. `unlock()`) operation does not belong to the same commutative critical section. This allows consecutive instances of a commutative critical section to be reordered with the same clock value during a replay execution. Furthermore,

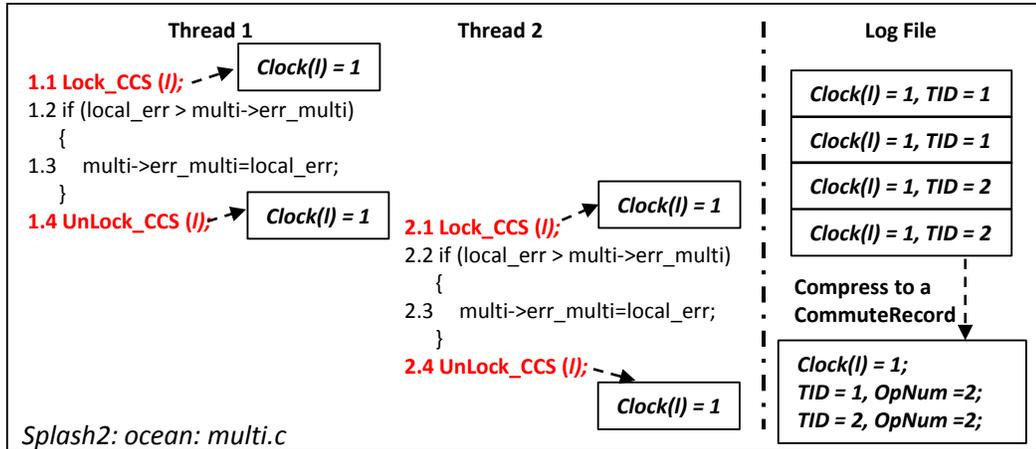


Figure 4.14: An example from Ocean, where there are two instances of a commutative critical section, one in each thread. In CommuteReplay, the clock value remains the same for the mutex operations in both threads since the mutex operations belong to the same commutative critical section. To record, a CommuteRecord entry is stored in the log file, which contains the clock value, and the number of mutex operations in each thread that share the same clock value (i.e. consecutive mutex operations that belong to the same commutative critical section).

we do not increment the clock of a mutex object on a release operation for a commutative critical section. For those mutex synchronization operations from non-commutative critical sections, their clock values are incremented on each mutex operation, as in BaseReplay.

Figure 4.14 shows how we record mutex operations that belong to a commutative critical section. In this example, the clock value remains at 1 as all mutex operations belong to the same commutative critical section, and there are 4 operations recorded. In CommuteReplay, instead of recording consecutive mutex operations from the same commutative critical section in separate log entries, we store a compressed *CommuteRecord* entry for all consecutive operations. The

clock value, and the number of mutex operations (i.e. *OpNum*) sharing the clock value in each thread is stored in a `CommuteRecord` entry.

There are two main advantages of storing a compressed `CommuteRecord` entry for consecutive mutex operations from the same commutative critical section. First, a compressed record entry allows `CommuteReplay` to easily fetch the total number of consecutive mutex operations that need to be executed in each thread before the clock value of the mutex object can be incremented on a replay. Second, the log file in `CommuteReplay` is highly compressed in many cases, which leads to a much lower logging overhead than `BaseReplay`, as we will demonstrate in our evaluation section.

To generate a `CommuteRecord` entry, `CommuteReplay` maintains a counter per mutex object per thread (i.e. *Counter[MutexObj][TID]*). On a mutex operation that is a part of a commutative critical section, the counter is incremented instead of recording the mutex operation in a log file right away. When the clock value of the mutex object is incremented (i.e. no more consecutive instances of a commutative critical section exist to be reordered), the value in the counter is then stored to a `CommuteRecord` entry in the log file of the mutex object used.

To record a mutex operation that is not a part of a commutative critical section, we record the clock value and the TID of the executed thread in the log file of the mutex object used, as in `BaseReplay`.

In `CommuteReplay`, the replay logic remains largely the same as `BaseReplay`, with the exception of the replay mechanism for the mutex synchronization operations. Figure 4.15 shows the operations in `CommuteReplay` on a mutex operation.

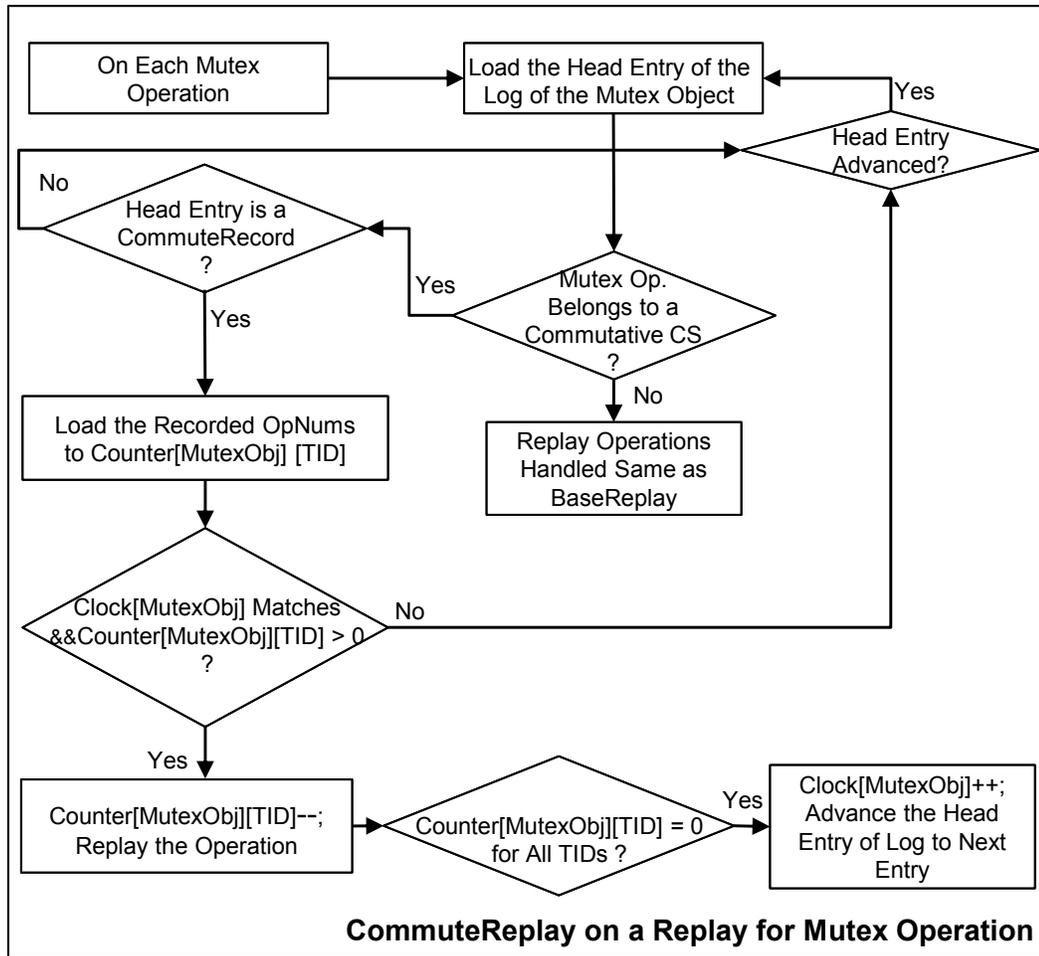


Figure 4.15: The CommuteReplay operations on a replay for each mutex synchronization operation.

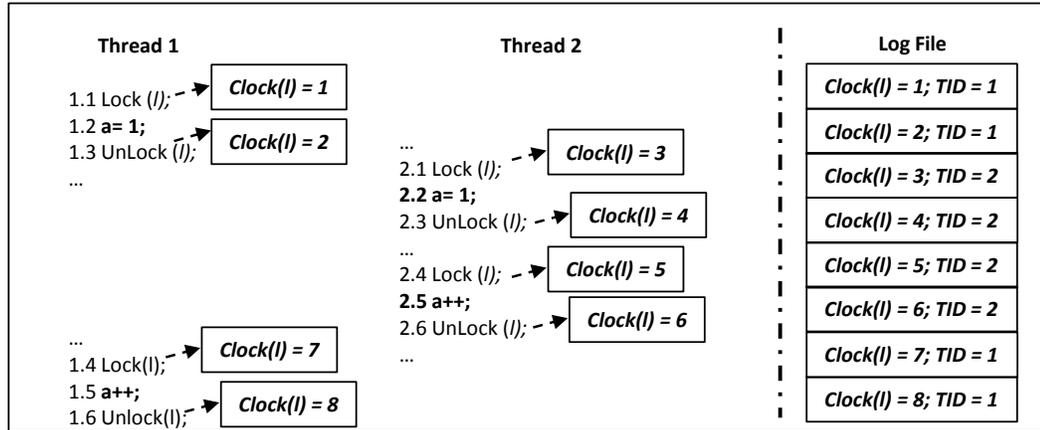
For a mutex operation that does not belong to a commutative critical section, we check if the mutex object's clock value and the TID match the head entry of the recorded log of the mutex object used, as in `BaseReplay`. If so, we replay the mutex operation, increment the mutex object's clock value by 1, and advance the head of the log file to its next entry. Otherwise, the mutex operation is stalled until both the clock value and the TID match the head entry of the recorded log of the mutex object used. We note that if the head entry of the recorded log is a `CommuteRecord` entry, then the non-commutative mutex operation should also

be stalled.

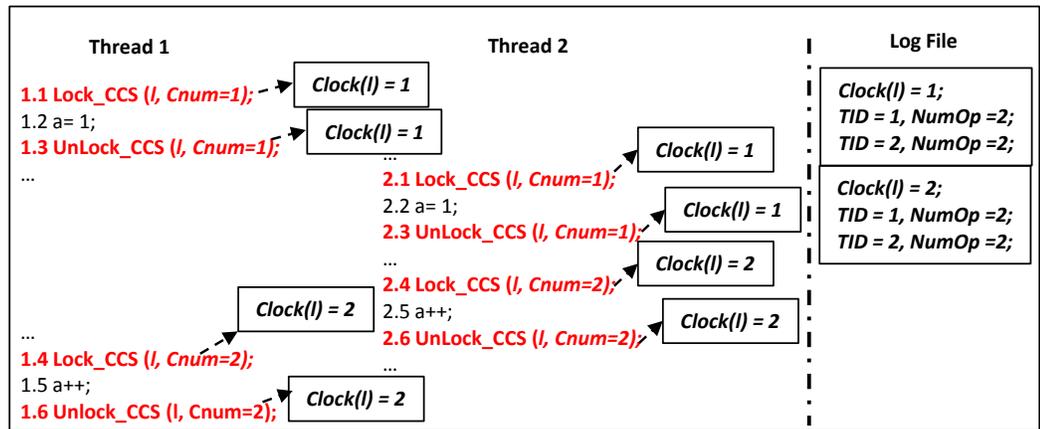
On the other hand, to replay a mutex operation that belongs to a commutative critical section, we first check if the head entry of the recorded log is a `CommuteRecord` entry. If not, we stall until the head entry becomes a `CommuteRecord` entry. If so, we load each `OpNum` value from the `CommuteRecord` entry to a counter that is maintained per mutex object per thread (i.e. $Counter[MutexObj][TID]$). If the clock value matches the recorded value and $Counter[MutexObj][TID]$ is greater than zero for the local thread, we replay the mutex operation and reduce the counter by 1. Once a `CommuteRecord` entry is loaded, we will only increment the clock value and advance to the next entry in the recorded log when all the recorded consecutive instances of a commutative critical section are executed on a replay (i.e. $Counter[MutexObj][TID] = 0$ for all TIDs)

Overall, as `CommuteReplay` allows more than one mutex operation to have the same clock value, the consecutive instances of a commutative critical section can be replayed in different orders to allow better flexibility and lower performance overheads.

Recall that the clock value of a mutex object is incremented only on an acquire (i.e. `lock()`) operation if the previous release (i.e. `unlock()`) operation does not belong to the same commutative critical section. Figure 4.16 shows how the clock value is updated in both `BaseReplay` and `CommuteReplay` when there are more than one critical section in a program's execution. In this example, there are two commutative critical sections that use the same mutex object. In `CommuteReplay`, the clock value is only updated at 2.4 when the lock operation and the previous unlock operation do not belong to the same commutative



(a) Clock updates in BaseReplay Scheme



(b) Clock updates in CommuteReplay Scheme

Figure 4.16: There are two commutative critical sections that use the same mutex object. Two instances of each commutative critical section are shown. In the BaseReplay scheme, the clock of mutex object l is updated on each mutex operation. In the CommuteReplay scheme, the clock value is only updated when the previous operation is from another critical section.

critical section. Therefore, on replay we would allow 2.1-2.3 to execute before 1.1-1.3, though 1.4-1.6 and 2.4-2.6 cannot be executed before 2.1-2.3 and 1.1-1.3. The clock value would also increment if the mutex synchronization operations are not from a commutative critical section, and only reordering between consecutive instances of a commutative critical section is allowed on a replay in CommuteReplay.

4.3.5 Logging Optimization

In `CommuteReplay`, we record consecutive instances of a commutative critical section in a compressed `CommuteRecord` log entry. Specifically, the number of mutex operations (i.e. *OpNum*) sharing a clock value in each thread is stored in a `CommuteRecord` entry. To further reduce the logging overhead, a simple optimization can be done to not store the *OpNum* for a thread if the number is zero. On a replay, if there is no information recorded for a particular thread in the `CommuteRecord` entry, the associated counter value would be initialized as zero by default.

Furthermore, in the case when a clock value of a mutex object is never incremented, we can simply indicate that no ordering is needed on replay for that mutex object. Such a scenario would occur if a mutex object is only used for one critical section, and the critical section is commutative.

It is also possible to further compress the logging overhead by not storing the clock values in both `BaseReplay` and `CommuteReplay`. As we maintain a separate log file for each synchronization object, the recorded clock value for an entry is always one more than the previous entry's clock value. Therefore, the clock value of any entry in a recorded log can be easily computed on a replay.

4.3.6 External Determinism Guarantee

Here, we show that `CommuteReplay` provides the same external determinism guarantee as the `BaseReplay` scheme.

First, we discuss whether all assumptions from Section 4.2.4 are met in

`CommuteReplay`. Otherwise, the identified commutative critical sections may not have all of the commutative properties that we have discussed earlier, and we cannot deduce the external determinism in `CommuteReplay` by using the commutative properties of a commutative critical section.

Recall that the following assumptions are required for the identification of commutative critical sections.

1. Other than CS_1 and CS_2 , the rest of the program is executed in a deterministic fashion. In particular, all memory accesses (other than the accesses within $\langle CS \rangle$) are executed deterministically.
2. During the execution of CS_1 and CS_2 , there is no conflicting access to $x, \forall x \in X$ and $y, \forall y \in Y$ outside of CS_1 and CS_2 . Recall we define conflicting accesses as accesses from different threads to the same memory location, which include at least one write.
3. There is no memory aliasing between variables used in $\langle CS \rangle$.
4. There is no user visible output produced within $\langle CS \rangle$.

`CommuteReplay` uses the same record and replay mechanism as the `BaseReplay` scheme for the rest of the program executions other than those within commutative critical sections and therefore satisfies Assumption 1.

In a data race free program, conflicting accesses to a variable used in a commutative critical section must either be from another critical section that uses the same mutex object, or the conflicting accesses are explicitly ordered by ordering synchronization operations. In the latter case, `CommuteReplay` handles the replay of the ordering synchronization operations in the same way as

`BaseReplay`, and guarantees that the replay of the ordering synchronizations is deterministic. In the former case, such as the example shown in Figure 4.16(b), `CommuteReplay` increments the clock value if two consecutive mutex operations are from different critical section code regions. In other words, only consecutive instances of a commutative critical section can share a clock value and be reordered during a replay.

Essentially, we only need to ensure the commutative properties between consecutive instances of a commutative critical section that are not interleaved by any other critical section that uses the same mutex object or any ordering synchronization operation. Hence, Assumption 2 is satisfied because `CommuteReplay` only allows reordering of consecutive instances of a commutative critical section when there is no conflicting access from outside of the commutative critical section. Additionally, Assumptions 3 and 4 are also checked as the necessary conditions when identifying commutative critical sections.

Second, we reason how `CommuteReplay` provides the external determinism, which requires both the final program state and the user visible output to be deterministic on a replay. In `CommuteReplay`, a replay execution of a program may consist of multiple code regions of consecutive instances of a commutative critical section. In the rest of the discussion here, we refer to such code region as a *commute code region*.

As `CommuteReplay` handles the replay for the rest of the program execution deterministically in the same way as `BaseReplay`, the program state before the first commute code region is deterministic on a replay. By definition, for any given input program state, any number of consecutive instances of a commutative critical section can be reordered and the program state would re-

main deterministic after all instances are executed. Hence, given the input to the first commute code region remains deterministic, the program state after the commute code region is also deterministic in `CommuteReplay`.

We can extend the analysis for the first commute code region to the rest of the program on a replay, and deduce that the program state before and after each commute code region remains deterministic. Therefore, the final state of the program would remain deterministic since all code regions would always receive a deterministic input program state and produce a deterministic output program state.

Additionally, we recall that no user visible output produced is a necessary condition for a commutative critical section. Hence, any user visible output would be produced deterministically outside of the commute code regions on a replay, as in `BaseReplay`.

Overall, `CommuteReplay` allows consecutive instances of a same commutative critical section to be reordered on a replay while still providing the external determinism guarantee.

4.4 Evaluation

This section presents evaluation results for the `CommuteReplay` scheme. We first study the sources of overheads from the `BaseReplay` scheme, and further evaluate the speed up that `CommuteReplay` can provide during a replay. We provide evaluation results for several execution environments.

We note that `CommuteReplay` records essentially the same amount of in-

formation as `BaseReplay`, with an optimization to compress records for consecutive mutex synchronization operations of a commutative critical section. Overall, the recording overheads in both `BaseReplay` and `CommuteReplay` can be best compared to an offline scheme such as `RecPlay` [65] as we record roughly the same amount of data in this case. Particularly, if the recording operations are implemented by modifying the Linux kernel and the GNU glibc library as shown in previous approaches [40, 65], the average overhead for the recording operations has been shown to be small enough (2.1% for `RecPlay`) to keep recording switched on all the time, even in deployed systems.

`CommuteReplay` only differs from `BaseReplay` during a replay execution when there is at least one commutative critical section in a program. The focus of our evaluation is whether `CommuteReplay` can treat the commutative critical sections within the program with more flexibility during a replay execution to allow a better replay performance.

4.4.1 Evaluation Setup

We have implemented a software based offline record and replay infrastructure using the Pin binary instrumentation framework [47]. Our Pin tool implements both the `BaseReplay` and `CommuteReplay` schemes by intercepting Pthread and system calls.

As our schemes are designed to be used for offline replays, we also need to consider different execution environments for a replay system in our evaluation. This is because an offline replay scheme allows the recorded execution of a program to be replayed after the original execution of a program has com-

pleted. For an offline replay, there is no guarantee that the replay system would have the same hardware as the system used for the recorded execution. Indeed, for software debugging and verification purposes, it is likely that the development systems are not as powerful as the real world deployed systems where the execution logs are recorded. For example, consider a web server program that records logs during an execution when a bug is observed. The logs are then sent to the program developer for further analysis. In this case, it is likely that the developer's system does not have as many processors as a deployed web server cluster. Moreover, it is possible that several systems, each with different configurations, are used to replay the same recorded execution for debugging and testing purposes. Finally, it is also possible that the record and/or replay executions are done on heterogeneous systems. The heterogeneity of a system can be from its architectural design due to resource sharing with other unrelated processes (i.e. on a time shared system), or due to distributed systems where each node could have different computing resources.

We have evaluated our proposed schemes with three different configurations, namely *2core_4thread*, *4core_4thread* and *4heterocore_4thread* as shown in Table 4.1. As there are many possible configurations in practice, we evaluated the three configurations to show that `CommuTeReplay` can achieve better replay performance in general. In our evaluation, all three configurations above used the same set of recorded logs that was generated under the *4core_4thread* configuration. The recorded logs are saved in text format on the hard drive of the *4core_4thread* system, and later copied to the other two systems. We note that while the system used for *2core_4thread* setup has less memory than the rest, it is not an issue as none of the benchmarks used in our evaluations requires more than 2GB of system memory.

Table 4.1: The execution environments used in our evaluation.

Replay Configuration	Description
1) 2core_4thread	2.13GHz Dual Core (Core 2 Duo) with HT off 4GB Memory, Ubuntu 11.10 2 cores for 4 threaded execution
2) 4core_4thread	1.86GHz 8 Core (Xeon), 8GB Memory Red Hat Enterprise 5.9 Only 4 cores are used for 4 threaded execution
3) 4heterocore_4thread	1.86GHz 8 Core (Xeon), 8GB Memory Red Hat Enterprise 5.9 Only 4 cores are used for 4 threaded execution, and 2 cores are slowed to about 1/4 of its original speed by instrumentation.

In order to evaluate the performance impact of `CommuteReplay`, We use benchmarks that have at least one commutative critical section, as the replay performance of `CommuteReplay` would be exactly the same as `BaseReplay` otherwise. Out of 9 benchmarks from SPLASH2 (4 threads, default input size) and PARSEC (4 threads, `simmedium` input size) benchmark suites, we find 7 benchmarks to have at least one critical section, and 5 benchmarks to have at least one commutative critical section. The 5 benchmarks with commutative critical sections are evaluated here.

4.4.2 Commutative CS Count

Table 4.2 shows the static number of the total and commutative critical sections, and the number of the total and commutative critical section instances that are dynamically executed in the benchmarks used in our evaluation. Here, a dy-

Table 4.2: Counts of the total and commutative critical sections. (P) - PAR-SEC, (S) - SPLASH2.

	Static CS in Source Code		Dynamic CS Executed	
	Total	Commutative	Total	Commutative
Fluidanimate (P)	5	4	182091	177720
Ocean (S)	4	3	834	828
Radix (S)	6	5	32	26
Water-Spacial (S)	8	4	79	68
Water-Nsquare (S)	8	7	6229	6136
Geomean:	–	73.8% of total CS	–	92.2% of total CS

dynamic instance of a critical section is counted as commutative when it shares a clock value with at least one more dynamic instance of the same critical section. In other words, a dynamic instance of a critical section is counted as commutative if it belongs to a code region where consecutive instances of a commutative critical section are executed. Overall, we found that the vast majority (92.2%) of executed critical section instances are commutative and can be executed more flexibly on a replay in `CommuterPlay`. We note that the static commutative critical sections are executed much more often than non-commutative ones in all of the benchmarks tested. Our evaluation results concur with this finding and show that most of the stalling on a replay are indeed caused by stalling on mutex synchronization operations, and such stalling can be largely eliminated by allowing commutative critical sections to reorder during a replay execution.

Table 4.3: Log sizes and compression ratio in CommuteReplay. (P) - PARSEC, (S) - SPLASH2.

	Log Size (BaseReplay)	Log Size (CommuteReplay)	Log Size Comparison (% to BaseReplay)
Fluidanimate (P)	2.9MB	79KB	2.7%
Ocean (S)	11.5KB	0.6KB	5.2%
Radix (S)	420B	56B	13.3%
Water-Spacial (S)	854B	126B	14.7%
Water-Nsquared (S)	84KB	8KB	9.5%
Geomean:	–	–	7.7%

4.4.3 Log Size Study

Table 4.3 shows the recorded log size for each benchmark used in our evaluation under both `BaseReplay` and `CommuteReplay`. We note that in all of the benchmarks tested, the number of ordering synchronization operations (if there is any) is always much smaller than the number of mutex synchronization operations. Hence, the overall log size can be reduced significantly if we can reduce the logging overhead for the mutex operations. Recall that in `CommuteReplay`, the records of consecutive mutex operations from a commutative critical sections are compressed to a `CommuteRecord` log entry. Since most of the mutex operations belong to consecutive commutative critical section instances (see Table 4.2), their log records can therefore be compressed to reduce the overall log size. Overall, in `CommuteReplay`, the log size is reduced to 7.7% of its original size on average. Our evaluation results also concur that `CommuteReplay` is able to eliminate most of the performance overhead caused by loading and comparing the logged records on a replay.

4.4.4 Performance Impact

We now discuss the performance impact of the `CommuteReplay` scheme in each configuration separately. In each case, we first discuss the performance overheads introduced by the Pin framework, and then discuss the actual performance overheads introduced by the replay scheme.

Configuration 1: 2core_4thread

In this configuration, we map a 4 threaded application to a 2 core machine on a replay. There are several sources of performance overheads for our replay schemes. As we implement the proposed scheme using the Pin binary instrumentation framework, there is a performance overhead due to running native Pin on top of a program even without any Pintool. Additionally, our replay schemes are implemented as a Pintool, which instrument an application and further perform the necessary operations within the instrumentation calls to replay the program. In order to distinguish the overheads based on their underlying causes, we measured the Pin slowdown, the slowdown due to instrumentation calls within a Pintool only (i.e. no replay operations are performed), and the slowdown caused by the replay operations. The first two types of slowdowns are shown in Table 4.4. Overall, the average slowdown due to Pin is 12.03x, where the instrumentation calls of our Pintool add another 3.14x slowdown on average.

We note that the slowdowns shown in Table 4.4 can be mostly alleviated if we implement the replay scheme by modifying the Linux kernel and the GNU glibc library in a similar fashion as several previous approaches have done

Table 4.4: Pin and instrumentation-only slowdowns for 2core_4thread configuration. (P) - PARSEC, (S) - SPLASH2.

	Pin Slowdowns	Instrumentation-Only Slowdowns Compare to Pin
Fluidanimate (P)	10.54x	5.31x
Ocean (S)	6.71x	1.50x
Radix (S)	23.63x	1.39x
Water-Spacial (S)	10.46x	4.76x
Water-Nsquared(S)	8.84x	2.72x
Average	12.03x	3.14x

[40, 65]. Hence, the performance overhead due to the actual replay operations within instrumentation calls is a better indication of how our replay schemes would fair in a real world situation. In the rest of this section, we refer to such overhead as *replay-only overhead*.

As shown in Figure 4.17(a), `CommuteReplay` can reduce the replay-only overhead to a minimum when compared with the `BaseReplay` scheme. In general, we can identify the replay-only overhead into two subcategories, namely *waiting overhead* and *logging overhead*, as shown in Figure 4.17(b). The replay schemes would incur waiting overhead when one or more threads stall on synchronization operations while waiting for other synchronization operations with a smaller clock value to finish first. The logging overhead is incurred due to the loading and checking of the recorded log entries during a replay execution. Overall, `CommuteReplay` eliminates all visible replay-only overhead. In all applications tested, most, if not all, of the waiting overheads are caused by stalling on commutative critical sections in `BaseReplay`. Hence, our `CommuteReplay` scheme is able to eliminate virtually all waiting overheads by allowing different

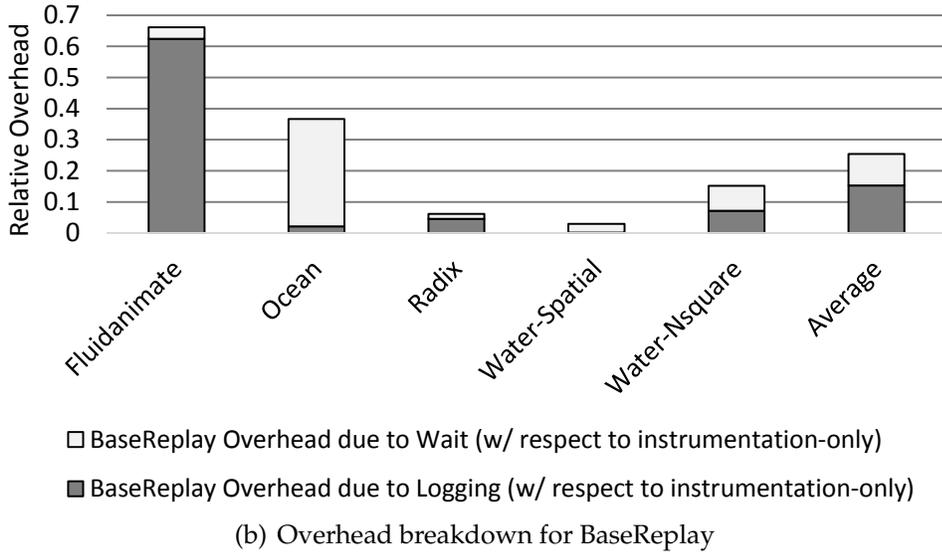
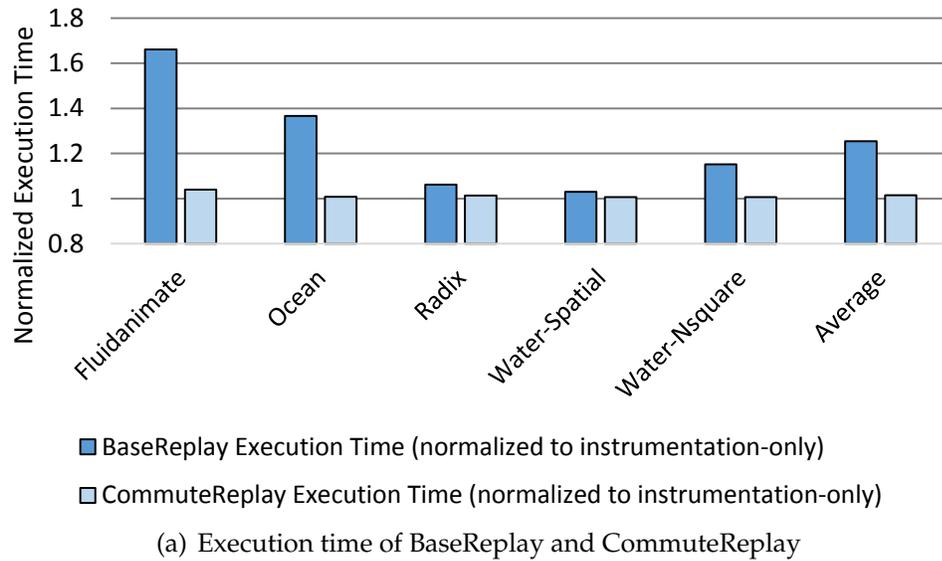


Figure 4.17: The execution time of BaseReplay and CommuteReplay in Configuration 1. The execution time is normalized to the instrumentation-only execution time. The BaseReplay’s overhead is broken down to two main categories.

instances of a commutative critical section to execute in different orders than the recorded one. Recall Table 4.3 shows that the log size in CommuteReplay is an order of magnitude smaller than in BaseReplay. Hence, we are able to eliminate most of the logging overheads by leveraging the logging optimization and compression techniques in CommuteReplay.

Table 4.5: Pin and instrumentation-only slowdowns for 4core_4thread configuration. (P) - PARSEC, (S) - SPLASH2.

	Pin Slowdowns	Instrumentation-Only Slowdowns Compare to Pin
Fluidanimate (P)	13.89x	8.58x
Ocean (S)	12.54x	1.69x
Radix (S)	27.33x	1.41x
Water-Spacial (S)	12.9x	8.45x
Water-Nsquared (S)	10.17x	4.40x
Average	15.37x	4.91x

Configuration 2: 4core_4thread

In this configuration, we map a 4 threaded application to a 4 core machine on a replay. Similar to our previous discussion, we first study the Pin slowdowns and the instrumentation-only slowdowns, then move on to examine the replay-only overhead of the replay schemes. We note that as the execution environment changes, the overall execution speed and scheduling can be different and therefore each application’s behavior could also change here.

As shown in Table 4.5, the Pin slowdowns are similar to what we had seen in Configuration 1. However, the instrumentation-only slowdowns are somewhat higher here (4.91x vs. 3.14x). There are a number of factors that contribute to such a change, such as different hardware with different Linux distributions which likely change how programs are executed in terms of scheduling and the overall execution time. Overall, both slowdowns shown here are within the same order of magnitude as those in Configuration 1.

Under Configuration 2, we note that the replay-only overhead for several

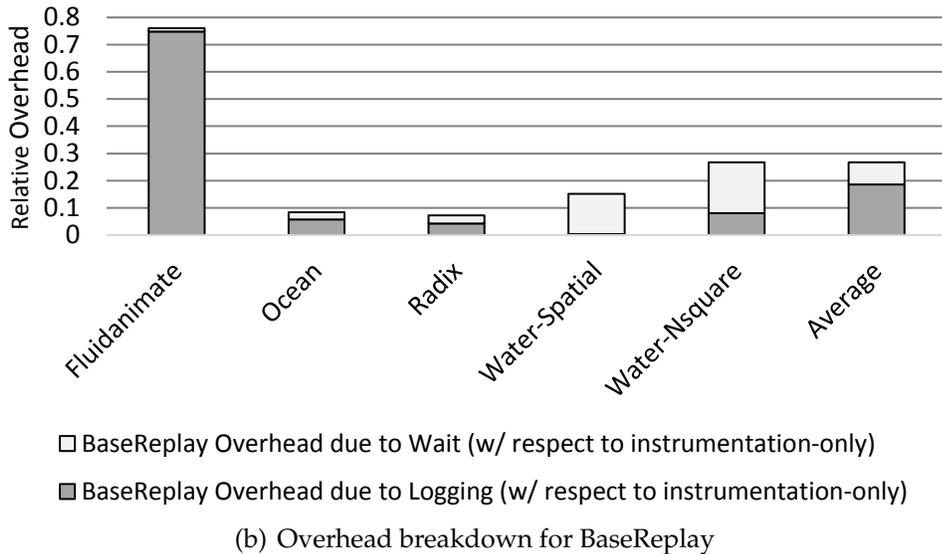
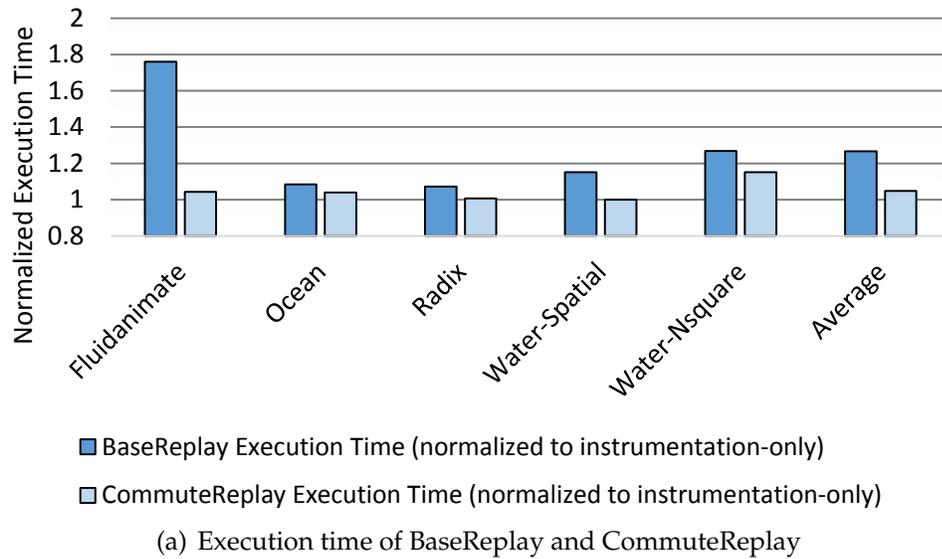


Figure 4.18: The execution time of BaseReplay and CommuteReplay in Configuration 2. The execution time is normalized to the instrumentation-only execution time. The BaseReplay’s overhead is broken down to two main categories.

benchmarks are different than those in Configuration 1. Most notably, *Ocean* has a much lower waiting overhead than before. As shown in Figure 4.6, *Ocean* has a commutative critical section that computes a global max value from locally computed local max values. As the computations for the local max value

are mostly identical in all threads, when all 4 threads in *Ocean* are executed in parallel on 4 homogeneous cores, every threads reaches the commutative critical section at virtually the same time. Therefore, only very little waiting time is incurred even in `BaseReplay`. However, under the setup of Configuration 1, waiting overhead is incurred when a thread stalls while waiting for another thread to become active (i.e. context switch into a core) and computes the local max value before it reaches the critical section.

Water-Spatial and *Water-Nsquared* show a somewhat higher waiting overhead under configuration 2. The increased waiting overhead is likely due to different scheduling policies between configuration 1 and 2. For example, a particular thread A could stall longer if it is scheduled to run ahead of thread B, while it must wait for thread B to reach a certain synchronization operation first.

Overall, `CommuteReplay` is able to eliminate most of the replay-only overhead under Configuration 2 as well. We note that there is a certain amount of waiting overhead (about 14%) remain for *Water-Nsquared* in `CommuteReplay`. This is because the waiting overhead is caused by other synchronization operations rather than those within a commutative critical section.

Configuration 3: 4heterocore_4thread

For Configuration 3, we add additional delays to 2 of the 4 cores in our instrumentation codes in order to mimic a heterogeneous setup. Namely, we add empty spin loops to the instrumentation codes before committing instructions for the cores that executes threads with odd numbered thread IDs. The goal here is to demonstrate how `CommuteReplay` behaves in a general heteroge-

neous setting. We believe the results shown here can be indicative at large.

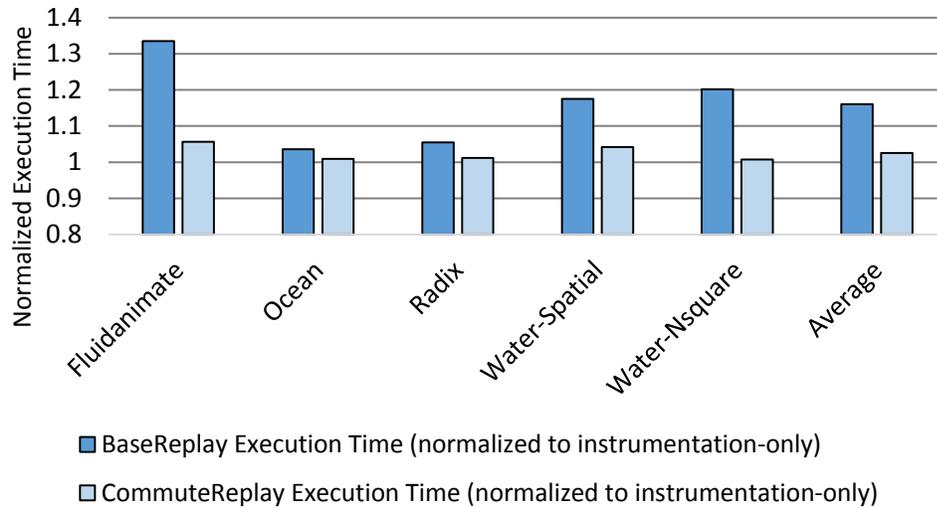
Since we have implemented the heterogeneous configuration through instrumentation, it is inherently not possible to measure the Pin and instrumentation-only slowdowns as it is not possible to measure native and Pin only execution time without the instrumented delays for our heterogeneous setup. Hence, we focus on evaluating the performance of `CommuteReplay` with respect to `BaseReplay` only. We do expect the Pin and instrumentation-only slowdowns to be within the same order of magnitude of the numbers shown for the previous two configurations.

As shown in Figure 4.19, the `CommuteReplay` scheme is able to eliminate most replay-only overheads across all applications. Again, *Water-Spatial* and *Water-Nsquared* show a somewhat higher waiting overhead under Configuration 3, and it is likely due to a changed overall execution ordering as now there are 2 cores executing significantly slower than the others. Hence, it is possible for threads on the faster cores to stall more often to wait for the synchronization operations to finish on the slower cores.

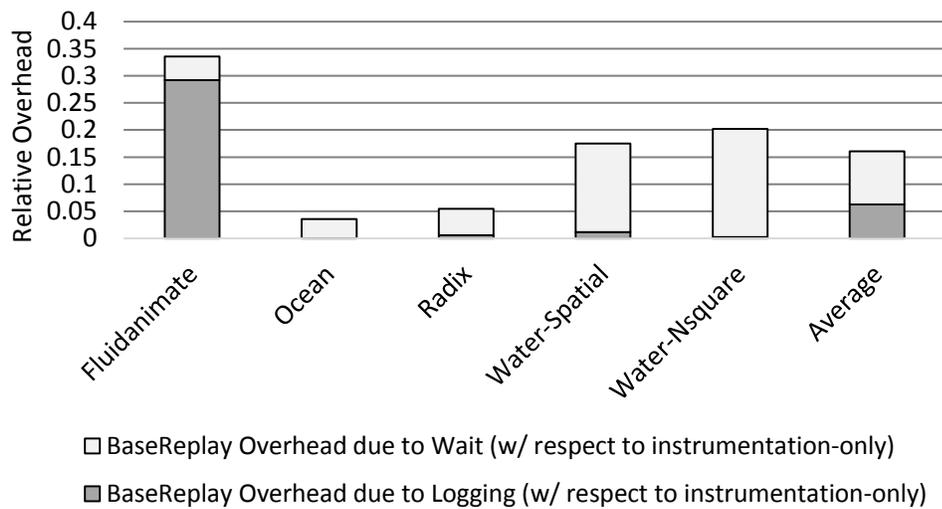
4.4.5 Limitations

Overall, in all three configurations, the proposed `CommuteReplay` scheme is able to eliminate most of the replay-only overhead for all of the applications tested, and provide significant speedups on multiple occasions. However, there are some limitations to our proposed scheme.

First, the replay-only overhead is not the only source of overhead in a replay



(a) Execution time of BaseReplay and CommuteReplay



(b) Overhead breakdown for BaseReplay

Figure 4.19: The execution time of BaseReplay and CommuteReplay in Configuration 3. The execution time is normalized to the instrumentation-only execution time. The BaseReplay's overhead is broken down to two main categories.

scheme. We anticipate the overhead in addition to the replay-only overhead to be anywhere between 10%-150% as suggested in previous works [2, 40, 65]. This is because although the majority of the Pin and instrumentation-only slowdowns can be eliminated if we implement our scheme by modifying the Linux kernel and the GNU glibc library as shown in previous approaches [40, 65], the overhead of such an implementation is not negligible. For example, additional operations need to be added to the necessary library calls to handle the ordering of synchronization operations (i.e. memory comparison, etc.). The kernel also needs to be modified so we can deterministically handle the necessary system calls, etc.

Second, `CommuteReplay` can only provide potential speedup to programs with at least one commutative critical section. Furthermore, it works best when there are a large number of commutative critical sections being executed, while the number of other synchronization operations and system calls are kept at a minimum. This is because other synchronization operations and system calls could interleave the consecutive instances of a commutative critical section and thus prevent those instances from being executed with flexible orderings during a replay. While the number of other synchronization operations and system calls is very small compared to the number of commutative critical section instances in all of the benchmarks that we have tested, there can be programs that do not have a large number of commutative critical instances to benefit from `CommuteReplay`.

Overall, the usefulness of our proposed scheme is bounded by whether a program contains commutative critical sections, and whether we can identify the commutative critical sections in a program in a timely fashion with accu-

racy. Fortunately, within our study, we have found that a large portion of the programs that we have encountered have at least one commutative critical section. We have also precisely defined a checking mechanism that can be used to systematically identify the commutative critical sections in a program accurately. Hence, we believe that a large range of existing programs in the field can benefit from our `CommutateReplay` scheme.

CHAPTER 5

RELATED WORK

This chapter summarizes the existing related work on data race and non-race concurrency bug detection schemes. Additionally, we also discuss previous work on deterministic execution for concurrent programs.

5.1 Concurrency Bug Detection

This research presents two schemes, named `RaceSMM` and `OSCS`, for detecting both data race and non-race concurrency bugs efficiently. Here, we discuss closely related works to both the proposed `RaceSMM` and `OSCS` schemes.

5.1.1 Data Race Detection

At a high-level, data race detection techniques can be categorized into static and dynamic approaches. Static race detection schemes such as `RacerX` [24] use static analysis with heuristics and statistical ranking to detect possible data races. However, static approaches are generally conservative without run-time information, resulting in false positives, and usually require source code, which may not be available in practice.

Dynamic data race detection techniques fall into two main classes, namely lockset based and happens-before based. The lockset approach, such as `Eraser` [66], checks whether each shared variable is protected by at least one lock. The overhead of the lockset approach can be quite low with hardware support such as `HARD` [79]. While the lockset approaches are simple and generally effective,

they inherently rely on heuristics and can result in a large number of false positives. The happens-before approach checks whether two memory accesses are explicitly synchronized [36]. There are many previous proposals that fall into the happens-before category, including RecPlay [65], Light64 [55], ReEnact [62], CORD [61], FastTrack [27], RADISH [18] and others. In general, the happens-before approaches are more accurate than the lockset approaches but often have higher overheads. Researchers have also investigated hybrid approaches in order to reduce the overhead of happens-before algorithms while maintaining a low false positive rate [56, 23, 60].

The proposed RaceSMM architecture can be considered as an extension of the happens-before approach to detect races at run-time. However, our hardware architecture shows that the happens-before race detection approach can be realized in hardware with minimal performance overheads and with minimal impact on detection coverage.

In general, data race detection schemes are heuristic approaches in detecting potential concurrency bugs. They are heuristic because not every data race will result in a concurrency bug, and vice versa. The proposed OSCS detection scheme extends the intuition of conflicting memory accesses (i.e. data races) into critical sections, namely order-sensitive critical sections. In OSCS however, we focus specifically on non-data race bugs. OSCS detects bugs that cannot be detected by any data race scheme.

5.1.2 Hardware-Based Race Detection

Many of the above data race detection and concurrency bug detection schemes present possible hardware support mechanisms to reduce overheads. Here, we discuss the data race detection techniques that are most closely related to the proposed technique in more detail. In this context, ReEnact [62], CORD [61], and RADISH [18] are the most related detection schemes to our work.

In particular, ReEnact [62] provides hardware support for logical vector clocks for cache lines. Due to its high hardware complexity and the use of vector clocks for cache lines, ReEnact suffers from noticeable performance overhead, and poor scalability for more than a few threads.

CORD [61] avoids the overheads and poor scalability issues of vector clocks by keeping four scalar timestamps per cache line, at the expense of lower detection coverage (77%). CORD also has a high hardware overhead as it integrates meta-data into each cache line, thus pays space overheads on every cache line. Overall, CORD provides a scalable data race detection scheme with negligible performance overhead and no false positive at the cost of lower detection coverage and high hardware overhead.

RADISH [18] proposes a hardware and software hybrid race detection scheme that uses a vector clock based race detection approach similar to FastTrack [27]. While it provides good scalability and a comprehensive detection coverage through a hardware and software assisted scheme, it still incurs a significant performance overhead at run-time for some applications (up to 2x for certain benchmarks, 80% on average). RaceSMM shows that decoupling of the detection of dynamically shared memory locations within a small window and

the rest of the bookkeeping can significantly reduce hardware and performance overheads with minimal impact on coverage.

Overall, `RaceSMM` provides a scalable scheme with a high detection coverage and no false positives, and requires a low hardware overhead. Only a separate on-chip only buffer (AHB) is needed. `RaceSMM` is the only vector clock based scalable scheme that provides high detection coverage and low performance overhead while incurring no false positives.

As an alternative to the happens-before approach, researchers have also presented simple hardware support for race detection relying on other heuristics. For example, `HARD` [79] uses lockset and `SigRace` [52] uses hash signatures from Bloom filters to detect possible data races. These approaches enable reasonable race detection capability with minimal hardware additions. However, generally they trade off accuracy and coverage for simplicity. In `RaceSMM`, it is demonstrated that accurate happens-before race detection can also be realized with relatively simple hardware support.

We have also shown that the `OSCS` algorithm can be implemented in hardware with a similar approach as the `RaceSMM` scheme that we have also proposed in this research. In this context, the implementation of hardware supported `OSCS` scheme is also similar to `ReEnact` [62], `CORD` [61], and `RADISH` [18]. However, the `OSCS` scheme is specifically targeted on non-race bug detection, while maintaining similar advantages (i.e. low overhead, scalable, etc.) in hardware implementation as `RaceSMM` over the other related schemes.

5.1.3 Concurrency Bug Detection (Beyond Data Races)

Recently, there have been significant efforts to detect concurrency bugs using symptoms beyond data races. One popular approach is to detect and/or tolerate bugs based on common program behaviors. AVIO [43] and SVD [75] approximate intended atomic regions using common behaviors. Atom-Aid [46] tries to dynamically avoid atomicity violation bugs by creating atomic blocks at run time. MUVI [41] and ColorSafe [45] target to detect concurrency bugs that involve multiple variables. Bugaboo [44] detects anomalies in communication graphs. Alternatively, researchers have also found that concurrency bugs can be identified from their consequences such as memory errors [78] or other system failure patterns [77]. Traditionally, a program is often tested by running with many possible interleaving patterns and checking results [13, 51, 58]. This approach can detect any concurrency bug if a buggy interleaving is tried, yet can only test one case at a time.

This research presents a new approach to detect non-race bugs by introducing the concept of order-sensitive critical sections as a new heuristic. This approach can be applied to programs without learning application-specific or bug-specific behaviors, and can often identify potential bugs before they happen at run-time. Moreover, the proposed scheme can detect both atomicity and ordering violations as the heuristic used in OSCS identifies the non-determinism in shared memory state, which can be caused by any type of violations. However, this technique is still a heuristic and there is no guarantee on bug detection coverage. In this sense, the proposed scheme complements an existing body of work in non-race bug detection.

5.2 Deterministic Replay and Execution

This research introduces the concept of *commutative critical sections* and demonstrates that it is possible to largely eliminate the overhead of enforcing a deterministic ordering between the dynamic instances of such commutative critical sections on a replay. Our work in identifying commutative critical sections complements an existing body of work in deterministic replay. Future work can also extend the use of commutative critical sections for an efficient deterministic execution system. Here, we discuss the closely related works in the field.

5.2.1 Deterministic Replay

There have been a number of works done to record and replay a program's execution deterministically in the past. Traditionally, recording non-deterministic inputs, such as system interrupts, DMA, I/O, etc. is sufficient to ensure deterministic replaying for single-threaded applications. For example, there are previous works that instrument either the operating system [11, 12, 25, 67] or the virtual machine [1, 21, 68] to monitor and record non-deterministic events.

Multithreaded applications can be recorded and deterministically replayed on uniprocessor machines, such as in ReVirt [21] and DejaVu [1]. However, multiprocessor replay remains to be a challenging task due to the added difficulties of efficiently recording non-deterministic shared memory access interleaving. There are various previous works, such as one of the earliest systems named InstantReplay [38] and more recent works such as iDNA [7], PinSel [53], and SMP-ReVirt [22], that are implemented to record the order in which different

threads access a shared memory location and subsequently replay the recorded accesses. While the earlier works can achieve deterministic replay on a multiprocessor system, they often incur prohibitive performance overhead due to the cost of monitoring and recording memory accesses.

There are also approaches that use custom hardware support to reduce the performance cost of recording the execution order of all shared memory accesses. Bacon and Goldstein [3] first proposed to monitor and log all cache coherence messages instead of ordering between shared memory accesses as the memory order can be deduced from the coherence messages. Other hardware supported systems have taken various approaches in recording the ordering of shared memory accesses. For example, FDR [74] records the shared memory accesses through a custom cache coherence mechanism in a similar fashion as in Bacon and Goldstein [3]. BugNet [54] records the register file contents instead in order to enable deterministic replaying of a program's execution. Rerun [33] records the ordering between atomic episodes (i.e. blocks) of thread executions using Lamport Clocks [36] instead of recording individual memory access ordering. DeLorean [48] records only the commit order of atomic execution blocks that are similarly used in transactional memory or speculative multithreading systems to reduce the recording and replaying overhead. Capo [49] proposes a software-hardware interface with a similar hardware system used in DeLorean to provide both the efficiency of a hardware assisted replay system with the added flexibility of a software-based scheme for different recording/replaying needs in practice. Although hardware schemes can drastically reduce the performance overhead of deterministic replay, they suffer the cost of implementing Hence, an efficient software based deterministic replay scheme is still highly desirable.

To overcome the high overhead of recording the ordering of all shared memory accesses in software, there are several proposals which record the ordering of synchronization operations instead. Earlier works, such as RecPlay [40] and JaRec [29], instrument and record the ordering of synchronization operations to ensure deterministic replay for a data race free program (or deterministic replay until the first data race). However, a replay tool that can handle data races is more desirable in a general testing environment.

More recent state-of-the-art solutions provide support for deterministic replay even in programs with data races. For example, ODR [2] and PRES [59] both record partial execution information and use offline searching to test multiple thread interleavings between shared memory accesses to find a potential execution order that satisfies the determinism requirement on replay. Respec [40] provides an online replay scheme, which achieves external determinism, by recording the total order of all synchronization operations that use the same synchronization object. Respec also detects and subsequently replays past any data race in a program through redundant execution and check pointing. DoublePlay [72] also takes the approach of redundant execution to detect data races during recording through uniparallelism (i.e. executes a program with both thread-level parallelism and epoch-level parallelism). Chimera [39] detects any potential data race in a program through a sound lockset based static race detector, and enforces a weak lock mechanism around any potential conflicting memory accesses. It then provides a deterministic replay by logging the ordering between synchronization operations instead of the ordering between all shared memory accesses.

While we have implemented our deterministic replay scheme largely based

on Respec, the concept of commutative critical sections can be extended to all of the above state-of-the-art schemes to potentially achieve *external determinism* in replaying multithreaded programs. This is because all of the aforementioned replay schemes record and enforce the non-deterministic ordering of synchronization objects instead of shared memory accesses. Hence, all of the state-of-the-art replay schemes can benefit from the extra flexibility allowed in replaying mutex synchronization operations that enclose the commutative critical sections. The logging overhead of the aforementioned replay schemes can also be potentially reduced in a similar fashion as we have shown in this research, since the schemes all record the ordering of mutex synchronization operations in a similar fashion.

5.2.2 Deterministic Execution

Recent works have proposed enforcing deterministic execution by ensuring that the thread interleaving observed is always the same for any given input of a multithreaded program [4, 5, 6, 19, 20, 32, 57], without the need for recording the order of shared memory accesses and/or synchronization operations. Similar to the deterministic replay approach, a deterministic execution approach can reproduce a multithreaded execution deterministically, and can be used for debugging, testing, and tolerating bugs in a multithreaded program.

Deterministic execution can be supported in software, and existing schemes generally trade off between support coverage and overhead. Kendo [57] provides deterministic execution for data race free programs with low overhead by enforcing a deterministic order between synchronization operations. CoreDet

[4] supports deterministic execution through a compiler and runtime infrastructure for imposing a deterministic order between shared memory accesses and synchronization operations. While CoreDet can support arbitrary multi-threaded programs, even for the ones with data races, it incurs high overhead and is not suitable to be used in production environments. The dOS [5] operating system provides deterministic execution by tracking the ownership of memory locations through page tables. While dOS supports unmodified program binaries, its overhead is still significant. Grace [6] provides efficient deterministic execution by restricting its support to the class of programs that uses fork-join parallelism. Specifically, it executes each thread in a fork region atomically and commits each thread's execution in a deterministic order.

There are also recent works done using custom hardware to reduce the overhead of deterministic execution. DMP [19] supports deterministic execution efficiently through two approaches. It tracks the data-ownership and uses periodic barriers to ensure that all shared memory accesses happen deterministically; alternatively, it leverages support for transactional memory to speculatively execute code regions by assuming no shared memory access between threads, and roll-back and performs re-execute if shared memory accesses happen. RCDC [20] proposes a deterministic multiprocessor architecture that trades relaxed memory consistency model for low overhead. Similarly, Calvin [32] proposes a deterministic execution architecture that exploits the flexibility of a memory consistency model that is weaker than sequential consistency to improve performance overhead and scalability. Specifically, it supports the Total Store Order (TSO) memory model.

Overall, the notion of *commutative critical sections* can be leveraged for a more

efficient deterministic execution support. The instances of commutative critical sections can be executed with a flexible ordering while still guaranteeing external determinism. Future work can exploit the flexibility of executing commutative critical sections during a deterministic execution in designing an efficient deterministic execution solution.

5.2.3 Deterministic Programming Languages

There is a large body of works on deterministic parallel programming languages in the field. Deterministic parallel programming languages provide an explicit deterministic programming model, thus make the program execution deterministic by default. For example, StreamIt [69] provides determinism for applications that can use streaming semantics by communicating between threads through explicitly declared streams. There are data-parallel functional languages [9, 16] that allow the programmers to use sequential semantics and thus yield deterministic parallel programs.

On the other hand, there are also implicit programming languages such as Jade [64]. When using Jade, programmers augment sequentially written programs with information on how shared memory data should be accessed during a parallel execution. The concurrency accesses are then extracted without violating the original sequential program semantics. Deterministic Parallel Java (DPJ) [10] is another example where a set of Java extensions are developed to allow programmers to control exactly when non-deterministic behaviors are allowed in a program.

We note that while using deterministic languages is a viable solution to the

non-determinism problem in multithreaded programs, most of the current parallel programs still use mainstream non-deterministic languages such as C, C++, or Java. In the foreseeable future, the use of non-deterministic languages to write parallel programs will likely remain the same. Therefore, our research on detecting and mitigating concurrency bugs in programs written with non-deterministic languages, such as C/C++, remains highly relevant not only today, but also in the future of computing.

CHAPTER 6

CONCLUSION

This thesis introduces efficient and effective schemes for data race detection, non-race bug detection, and deterministic replay in multithreaded programs.

First, the thesis proposes `RaceSMM`, an efficient hardware architecture that addresses challenges in supporting accurate run-time data race detection. We show that the proposed scheme enables run-time data race detection with high coverage and minimal performance overhead through an efficient hardware architecture. In particular, we propose an architectural optimization that decouples meta-data storage from regular caches so that expensive meta-data is only selectively stored for dynamically shared memory locations within a small window. Experimental results show that the hardware assisted race detection scheme provides high detection coverage (over 99%) with no false positives, while maintaining minimal overheads. Overall, `RaceSMM` provides an attractive way to detect data races at run-time.

Second, the thesis introduces the notion of order-sensitive critical sections, which captures critical sections that introduce non-determinism during executions, and presents a concurrency bug detection approach that covers non-race bugs. An optimized version of our algorithm is also introduced, which only requires scalar variables for each shared memory address instead of using vectors for all memory addresses. Experimental results show that a broad range of non-race concurrency bugs can be detected by both the baseline and the optimized algorithms. In particular, in our experiments, our algorithms detected all 9 real-world bugs and over 90% of all injected bugs with only a small number of false positives in practice. The algorithm can also be accelerated with hardware

support to have minimal performance overhead. The hardware implementation can still detect all 9 real-world bugs tested and above 84% of all injected bugs. Overall, we demonstrate that the proposed `OSCS` scheme is indeed an effective heuristic for non-race concurrency bug detection, and it can be implemented in hardware effectively and efficiently.

Last, the thesis investigates an efficient deterministic replay scheme, named `CommuteReplay`, which leverages the concept of commutative critical sections, and guarantees external determinism on a replay. Experimental results show that the proposed `CommuteReplay` scheme eliminates most, if not all, of the replay overhead in stalling or logging for a deterministic ordering between critical sections. Overall, the proposed scheme allows a more efficient deterministic replay execution for multithreaded programs.

We believe our work provides attractive ways in dealing with concurrency bugs during the parallel programming development cycle, especially for cases where a run-time solution with low overhead is needed. Overall, this thesis presents novel approaches and an overall solution in detecting and mitigating concurrency bugs efficiently.

BIBLIOGRAPHY

- [1] Bowen Alpern, Jong-Deok Choi, Ton Ngo, Manu Sridharan, and John M. Vlissides. A perturbation-free replay platform for cross-optimized multithreaded applications. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, 2001.
- [2] Gautam Altekar and Ion Stoica. ODR: output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 2009.
- [3] David F. Bacon and Seth Copen Goldstein. Hardware-assisted replay of multiprocessor programs. In *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*.
- [4] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. CoreDet: a compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.
- [5] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. Deterministic process groups in dOS. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, 2010.
- [6] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: safe multithreaded programming for C/C++. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, 2009.
- [7] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, 2006.
- [8] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. Technical Report TR-811-08, Princeton University, January 2008.
- [9] Guy Blelloch and Parallel Ram Model. NESL: A nested data-parallel language (version 3.1). Technical report, Carnegie Mellon University, Pittsburgh, PA.

- [10] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel Java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, 2009.
- [11] Bob Boothe. Efficient algorithms for bidirectional debugging. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, 2000.
- [12] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.
- [13] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.
- [14] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI'08*, 2008.
- [15] CAPSL. The modified SPLASH-2. <http://www.capsl.udel.edu/splash/>, July 2007.
- [16] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel Haskell: a status report. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, 2007.
- [17] Blas A. Cuesta, Alberto Ros, María E. Gómez, Antonio Robles, and José F. Duato. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, 2011.
- [18] Joseph Deviet, Benjamin P. Wood, Karin Strauss, Luis Ceze, Dan Grossman, and Shaz Qadeer. RADISH: always-on sound and complete race detection in software and hardware. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, 2012.

- [19] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: deterministic shared memory multiprocessing. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.
- [20] Joseph Devietti, Jacob Nelson, Tom Bergan, Luis Ceze, and Dan Grossman. RCDC: a relaxed consistency deterministic computer. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.
- [21] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.
- [22] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2008.
- [23] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: a race and transaction-aware Java runtime. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.
- [24] Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.
- [25] Stuart I. Feldman and Channing B. Brown. IGOR: a system for program debugging via reversible execution. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, 1988.
- [26] Colin Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24:28–33, August 1991.
- [27] Cormac Flanagan and Stephen N. Freund. FastTrack: efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.
- [28] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications. In *Proceedings of the Annual conference on USENIX '06 Annual Technical Conference*, 2006.

- [29] 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, May 2004.
- [30] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. R2: an application-level kernel for record and replay. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*.
- [31] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Reactive NUCA: near-optimal block placement and replication in distributed caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009.
- [32] Derek R. Hower, Polina Dudnik, Mark D. Hill, and David A. Wood. Calvin: Deterministic or not? Free will to choose. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, 2011.
- [33] Derek R. Hower and Mark D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, 2008.
- [34] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
- [35] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2005.
- [36] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, July 1978.
- [37] Chris Lattner and Vikram Adve. LLVM: a compilation framework for life-long program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, 2004.
- [38] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, 36(4):471–482, 1987.
- [39] Dongyoon Lee, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Chimera: hybrid program analysis for determinism. In *Proceedings of the*

33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, 2012.

- [40] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. Respec: efficient online multiprocessor replay via speculation and external determinism. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.
- [41] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, 2007.
- [42] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.
- [43] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.
- [44] Brandon Lucia and Luis Ceze. Finding concurrency bugs with context-aware communication graphs. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.
- [45] Brandon Lucia, Luis Ceze, and Karin Strauss. ColorSafe: architectural support for debugging and dynamically avoiding multi-variable atomicity violations. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, 2010.
- [46] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. Atom-Aid: detecting and surviving atomicity violations. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, 2008.
- [47] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 Conference on Programming Language Design and Implementation International*, 2005.

- [48] Pablo Montesinos, Luis Ceze, and Josep Torrellas. DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, 2008.
- [49] Pablo Montesinos, Matthew Hicks, Samuel T. King, and Josep Torrellas. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.
- [50] Naveen Muralimanohar and Rajeev Balasubramonian. CACTI 5.3: A tool to understand large caches.
- [51] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008.
- [52] Abdullah Muzahid, Dario Suárez, Shanxiang Qi, and Josep Torrellas. SigRace: signature-based data race detection. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009.
- [53] Satish Narayanasamy, Cristiano Pereira, Harish Patil, Robert Cohn, and Brad Calder. Automatic logging of operating system effects to guide application-level architecture simulation. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*.
- [54] 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*.
- [55] Adrian Nistor, Darko Marinov, and Josep Torrellas. Light64: lightweight hardware support for data race detection during systematic testing of parallel programs. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.
- [56] Robert O’Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2003.
- [57] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient

- deterministic multithreading in software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.
- [58] Soyeon Park, Shan Lu, and Yuanyuan Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.
- [59] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 2009.
- [60] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2003.
- [61] Milos Prvulovic. CORD: cost-effective (and nearly overhead-free) order-recording and data race detection. In *Proceedings of the 12th Annual International Symposium on High-Performance Computer Architecture*, 2006.
- [62] 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*, 2003.
- [63] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, 2005.
- [64] Martin C. Rinard and Monica S. Lam. The design, implementation, and evaluation of Jade. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(3):483–545, May 1998.
- [65] Michiel Ronsse and Koen De Bosschere. RecPlay: a fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17:133–152, May 1999.
- [66] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15:391–411, November 1997.

- [67] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2004.
- [68] John Steven, Pravir Chandra, Bob Fleck, and Andy Podgurski. jRapture: A capture/replay tool for observation-based testing. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2000.
- [69] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, 2002.
- [70] Chen Tian, Vijay Nagarajan, Rajiv Gupta, and Sriraman Tallam. Dynamic recognition of synchronization operations for improved data race detection. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, 2008.
- [71] Céline Valot. Characterizing the accuracy of distributed timestamps. In *Proceedings of the 1993 ACM/ONR Workshop on Parallel and Distributed Debugging*, 1993.
- [72] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. DoublePlay: parallelizing sequential logging and replay. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.
- [73] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad hoc synchronization considered harmful. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, 2010.
- [74] 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*, 2003.
- [75] Min Xu, Rastislav Bodík, and Mark D. Hill. A serializability violation detector for shared-memory server programs. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

- [76] Jie Yu and Satish Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009.
- [77] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. ConSeq: detecting concurrency bugs through sequential errors. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.
- [78] Wei Zhang, Chong Sun, and Shan Lu. ConMem: detecting severe concurrency bugs through an effect-oriented approach. In *Proceedings of the 15th Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.
- [79] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. HARD: hardware-assisted lockset-based race detection. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007.