Dynamic Analysis of Embedded Software

by

Young Wn Song

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved August 2015 by the
Graduate Supervisory Committee:

Yann-Hang Lee, Chair
Aviral Shrivastava
Georgios Fainekos
Joohyung Lee

ARIZONA STATE UNIVERSITY

December 2015

ABSTRACT

Most embedded applications are constructed with multiple threads to handle concurrent events. For optimization and debugging of the programs, dynamic program analysis is widely used to collect execution information while the program is running. Unfortunately, the non-deterministic behavior of multithreaded embedded software makes the dynamic analysis difficult. In addition, instrumentation overhead for gathering execution information may change the execution of a program, and lead to distorted analysis results, i.e., probe effect. This thesis presents a framework that tackles the non-determinism and probe effect incurred in dynamic analysis of embedded software. The thesis largely consists of three parts. First of all, we discusses a deterministic replay framework to provide reproducible execution. Once a program execution is recorded, software instrumentation can be safely applied during replay without probe effect. Second, a discussion of probe effect is presented and a simulation-based analysis is proposed to detect execution changes of a program caused by instrumentation overhead. The simulation-based analysis examines if the recording instrumentation changes the original program execution. Lastly, the thesis discusses data race detection algorithms that help to remove data races for correctness of the replay and the simulation-based analysis. The focus is to make the detection efficient for C/C++ programs, and to increase scalability of the detection on multi-core machines.

DEDICATION

To my parents for their love and support.

To my wife, Jeyeon, for enduring this long journey with me.

To my sweet girls, Angela and Minji.

ACKNOWLEDGMENTS

First of all, I would like to thank my advisor, Dr. Yann-Hang Lee for his encouragement and support, and for guiding me with his knowledge and experience. It took almost 6 years (that's such a long time!) to complete my Ph.D study. This dissertation would not be possible without his trust and great patience.

Secondly, I would like to thank my committee members, Dr. Aviral Shrivastava, Dr. Georgios Fainekos, and Dr. Joohyung Lee for suggestions and insightful comments. I sincerely appreciate their time and guidance to broaden the scope of this dissertation.

This work was supported in part by the NSF I/UCRC Center for Embedded Systems, and from NSF grant #0856090.

TABLE OF CONTENTS

Page

LIST OF TABLES . . . viii

LIST OF FIGURES . . . x

CHAPTER

1 INTRODUCTION . . . 1

1.1 Motivation . . . 1

1.2 Contributions . . . 3

1.2.1 Reproducible Execution . . . 3

1.2.2 Probe Effect Analysis . . . 4

1.2.3 Data Race Detection . . . 4

1.3 Overview . . . 4

2 BACKGROUND . . . 6

2.1 Happens-before Relation . . . 6

2.2 Logical Clock . . . 7

2.3 Vector Clock . . . 8

3 RECORD AND REPLAY . . . 11

3.1 Introduction . . . 11

3.2 Dynamic Analysis with Execution Replay . . . 14

3.3 Record and Replay . . . 16

3.3.1 Record and Replay Operations . . . 17

3.3.2 Handling Data Races . . . 19

3.3.3 Debugging Support . . . 21

3.4 Execution Time Estimation . . . 22

3.4.1 Execution Time Estimation at Program Level . . . 23

3.4.2 Overhead Measurements . . . 24

CHAPTER Page

3.4.3 Execution Time Estimation at Thread Level 26

3.5 Evaluation 27

3.5.1 Overheads of Record and Replay Operations 28

3.5.2 Execution Time Estimation 29

3.5.3 Accuracy of Analysis in Replay Execution 30

3.6 Related Work 33

3.7 Chapter Conclusions 35

4 PROBE EFFECT ANALYSIS 36

4.1 Introduction 36

4.2 Multi-threaded Program Execution 39

4.3 Model of Multi-threaded Program Execution 41

4.4 Simulated Program Execution 44

4.5 Analysis of Simulation Results 47

4.6 Implementation 49

4.6.1 Execution Environment 49

4.6.2 Execution Trace and Measurements 49

4.6.3 Simulation Analysis Algorithm 50

4.7 Evaluation 52

4.8 Related Work 58

4.9 Chapter Conclusions 61

5 DATA RACE DETECTION FOR C/C++ PROGRAMS 62

5.1 Introduction 62

5.2 Discussion of Data Race Detection 66

5.2.1 What is a (Data) Race? 66

5.2.2 Static Race Detection 70
5.2.3 Using Model Checking Techniques 73
5.3 Vector Clock Based Race Detectors 77
5.3.1 DJIT+ 77
5.3.2 FastTrack 80
5.4 Dynamic Granularity Algorithm 81
5.4.1 Vector Clock State Machine 82
5.4.2 Dynamic Granularity 85
5.5 Implementation 85
5.5.1 Instrumentation 86
5.5.2 Data Structures 87
5.6 Evaluation 88
5.6.1 Performance and Detection Precision 89
5.6.2 Analysis of State Machine 94
5.6.3 Case Studies 95
5.7 Related Work 97
5.7.1 Hybrid Race Detectors 97
5.7.2 Sampling/HW-assisted Approaches 98
5.7.3 Data Race Detection for C/C++ Programs 99
5.7.4 Classifying/Surviving Data Races 99
5.7.5 Automatically Fixing Data Races 100
5.8 Chapter Conclusions 101
6 DATA RACE DETECTION ON MULTI-CORE SYSTEMS 102
6.1 Introduction 102

CHAPTER Page

6.2 Overhead and Scalability of FastTrack 105
6.3 Parallel FastTrack Detector 107
6.4 Implementation 110
6.4.1 Instrumentation and Optimization 110
6.4.2 Parallel FastTrack 111
6.5 Evaluation 112
6.5.1 Analysis of Race Detection Execution 113
6.5.2 Performance and Scalability 117
6.6 Related Work 122
6.7 Chapter Conclusions 123
7 CONCLUSIONS 124
7.1 Summary 124
7.2 Future Work 125
REFERENCES 127

LIST OF TABLES

Table Page

1.1 QT Application with Mouse Inputs 2

1.2 POSIX Message Queue Application 2

3.1 Record and Replay Overheads 29

3.2 Revisit of the QT Application with Recording Operations 29

3.3 Revisit of the MQ Application with Recording Operations 29

3.4 Execution Time Estimation without Replay Operation 30

3.5 Data Race Detection with FastTrack 31

3.6 Flat Profiling Comparison with Callgrind 32

3.7 Cache Simulation Results from PIN Cache 32

4.1 CPU Time for the Dining Philosophers 53

4.2 CPU Time for the Sleeping Barber 54

5.1 Races Detected by a Static Detector, RELAY 70

5.2 Races Detected by a Static Detector, LOCKSMITH 71

5.3 Overall Experimental Results 90

5.4 Memory Overhead of FastTrack Detection with Different Granularities 91

5.5 Maximum Number of Vector Clocks Present 92

5.6 Measures of Same Epoch Accesses 93

5.7 Comparisons of State Machines with Different Configurations 94

5.8 Performance Comparisons with Valgrind DRD and Intel Inspector XE 95

6.1 Number of Accesses Filtered and Checked in the FastTrack Detection 113

6.2 Overheads of the FastTrack Detector 114

6.3 Comparison of CPU Core Utilization 116

6.4 Performance Comparison for the Parallel FastTrack Detection 118

Table Page

6.5 Speedups with Additional Cores 120

6.6 Maximal Memory Usages of the Detections 121

LIST OF FIGURES

Figure Page

1.1 Dissertation Overview 5

2.1 An Example of the Happens-before Relation and Logical Clocks 7

2.2 An Example of Vector Clocks 8

3.1 Dynamic Analyses with Execution Replay 15

3.2 Examples of Replay Minima/Maxima 21

3.3 Debugging Support in Eclipse 22

3.4 Displaying Events and Threads in Eclipse 22

3.5 Calculation of Blocking Time of Events by the Replay Scheduling 25

3.6 Decomposition of the Replay Overhead 31

4.1 An Example of Global/Local Clocks and Event Executions 43

4.2 An Example of Partial Order Graph G 44

4.3 Execution Time for the Simulation 45

4.4 Sub-routines for the Simulation Algorithm 51

4.5 Simulation Algorithm 52

4.6 An Example Execution with the Same Partial Order 55

4.7 An Example Execution with Different Partial Orders 55

4.8 An Example Execution with Different Execution Paths 56

4.9 Partial Order Graphs from Instrumented and Simulated Executions . . . 57

5.1 An Example of Atomicity Violation without Data Races 67

5.2 Classification of Data Races 68

5.3 Examples of Feasible/Apparent Races 69

5.4 A Benign Race Example 69

5.5 Filtering out False Alarms from Static Race Detection 72

Figure Page

5.6 An Example for Predictive Analysis Technique 75

5.7 Example Executions of DJIT+ and FastTrack 78

5.8 Idea of Dynamic Granularity 82

5.9 Vector Clock State Machine for Each Read or Write Location 83

5.10 Instrumentation Code for Memory Read 86

5.11 Indexing Data Structures 87

6.1 A High Level View of the FastTrack Detection 105

6.2 An Overview of Parallel FastTrack 108

6.3 CPI Measures of the Race Detection Programs 115

6.4 Scaling Factors of Race Detectors 119

6.5 Performance Comparison with/without the Hash Filter 120

Chapter 1

INTRODUCTION

1.1 Motivation

Dynamic program analysis is widely used to collect execution information while programs are running. The approach is widely used to aid optimization and debugging of the programs. However, the non-deterministic behavior of embedded software and probe effect [22] make it a challenging task as embedded applications are often constructed with multiple threads and I/O operations.

In Tables 1.1 and 1.2, the probe effect is illustrated in the execution of two embedded programs. The two programs are based on the class projects of Embedded Systems Programming [4] in Arizona State University. The first program is a QT [91] application which draws lines following the inputs of mouse movement. The program consists of three threads. The first thread receives mouse movement packets and sends them to a POSIX message queue (MQ). The second thread receives the input packets from the MQ and draws lines on a display device. The last thread performs a line detection algorithm with the received mouse inputs. We collected mouse movement at a normal sampling rate of 300 inputs per second and then fed the inputs to the application with variant speeds. If the first thread was delayed and was not ready to receive an input, we counted it as a missed input. The program is instrumented using two dynamic analysis tools, i.e., the cache simulator using PIN [29] and our implementation of the FastTrack data race detector [19]. The workload of the program is very light as it only spends less than 10% of CPU time. However, the instrumented execution may miss up to 45% of the inputs. The impact of probe effect caused by the

inputs/sec	Native execution	PIN Cache	Race detector
150	0.0%	16.8%	0.3%
300	0.0%	36.1%	1.2%
450	0.0%	45.5%	1.9%

Table 1.1: QT Application with Mouse Inputs (% of inputs missed out of 4445 mouse movement inputs)

Queue Length	Native execution	PIN Cache	Race detector
5	1.3/7.5	8.3/191.9	5.5/56.3
10	0.5/8	2.5/146.8	2.4/37.9

Table 1.2: POSIX Message Queue Application (# of Queue full/# of Queue empty)

instrumentation is obvious since analysis results may be misleading when the input data are missed.

The second program shown in Table 1.2 is a MQ (Message Queue) test program with six threads and two MQs. The two sender threads send items to the first MQ and the two router threads receive the items from the first MQ and send them to the second MQ with timestamps. Finally, two receiver threads receive the items from the second MQ. We used asynchronous functions for queue operations and, if a queue is empty or full, the thread sleeps a fixed time interval and retries. We count the numbers of occurrences that the queues become empty or full as a way of measuring different program behaviors. In this program, there is no external environment affecting the program execution, but the execution is determined by order of thread executions on the shared MQs. As the results show, instrumentation overhead from the tools has changed the relative ordering of thread operations on the shared MQs which, in turn, leads to different status of the message queues.

The other concern, followed by the data shown in Tables 1.1 and 1.2, is that it will be very hard to know if there exists any probe effect from the execution of instrumented program. If we take any amount of measurement to examine probe effect, the measurement itself would incur instrumentation overhead that can lead

to execution divergence. Even if we know that there were some changes in program execution, we still would not be able to know how the changes affect the results of the analysis.

1.2 Contributions

The goals of this thesis are to provide (1) a reproducible execution environment such that software instrumentation can be applied without any changes of program execution, and (2) analysis frameworks for probe effect and data race detections which are essential for building the reproducible execution environment.

1.2.1 Reproducible Execution

To provide a reproducible execution environment, we consider a record and replay framework. First a program execution is recorded. Second, we apply software instrumentation on replayed execution. If the replay guarantees the same execution as the record, dynamic analyses with the instrumentation can be safely performed without worrying about execution changes of the program. In the record and replay framework, the key is to have minimal recording overhead. Since the recording operation itself is instrumenting the program, the recording overhead can change the program execution.

To provide minimal recording overhead, we model a program execution as a partial order of synchronization and I/O events, i.e., the happens-before relation [39]. In other words, two executions of a program are same if and only if the partial orders from the two executions are identical. In our record and replay, we record a partial order of synchronization and I/O events, and we guarantee the same execution as the recorded one by enforcing the same partial order during the replay.

1.2.2 Probe Effect Analysis

It is perceivable that any recording operation should incur some instrumentation overhead since the execution of the recording itself would have caused perturbation to the original execution. We provide a simulation-based analysis for embedded software to detect any variations of event ordering caused by instrumentation overhead. The simulation-based analysis examines if the recording instrumentation changes the original program execution.

1.2.3 Data Race Detection

In our program execution model, we only consider synchronization and I/O events. However, the deterministic replay and the simulation-based analysis might not succeed in the presence of data races. For this end, this thesis describes the data race detection algorithms that help to remove data races. The algorithms are designed for efficient data race detection for C/C++ programs on multi-core machines. The race detector has low runtime and memory overheads by enabling large detection granularity, and we have further improved the performance and increased scalability on multi-core machines by decoupling data race detection from application execution.

1.3 Overview

Figure 1.1 illustrates an overview of this thesis. In the following chapter, we present an overview of background on which this thesis is built.

Chapter 3 describes our deterministic replay. We present the construction of our replay followed by the effectiveness of dynamic analysis of embedded software using the replay.

Chapter 4 discusses the probe effect analysis for multithreaded embedded software.

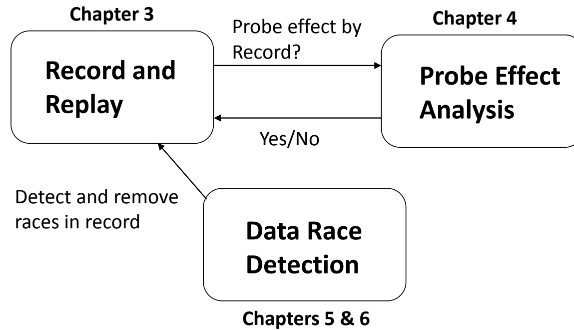


Figure 1.1: Dissertation Overview. The program under test is executed with recording operations. A recorded execution is reproduced and analyzed in replayed executions. Probe effect that might be induced by the recording is checked before replay. Data races are detected and removed when the replay fails.

The simulation-based analysis can detect any changes of program execution based on our modeling of multithreaded program execution.

Chapter 5 describes a data race detection algorithm for C/C++ programs. Based on the FastTrack algorithm [19], we build an efficient data race detector for C/C++ programs by utilizing large detection granularity during runtime.

Chapter 6 discusses how a data race detection can be parallelized by decoupling race detection from application execution. We show the effectiveness of our approach by parallelizing the FastTrack detector [19].

Chapter 7 concludes the thesis. We summarize our contributions and discuss our future work.

Chapter 2

BACKGROUND

2.1 Happens-before Relation

The happens-before relation [39] over the set of events in a program execution, denoted "$\rightarrow$", is the smallest relation satisfying,

- **Program order:** If a and b are in the same thread and a occurs before b, then $a \rightarrow b$.
- **Synchronization [1] :** If a is a release operation of synchronization object (e.g., unlock) and b is the subsequent acquiring operation on the same object (e.g., lock), then $a \rightarrow b$.
- **Transitivity:** If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

The happens-before relation implies a partial ordering of events. Two operations, a and b, are concurrent if they are not ordered by the happens-before relation, i.e., $a \nrightarrow b$ and $b \nrightarrow a$. An example of the happens-before relation in a program execution is shown in Figure 2.1.

An event, as a sequence of instructions that a program executes, defines a particular action (e.g., system call, memory read/write). We consider only a particular set of events for the underling system. In the deterministic replay (Chapter 3) and the probe effect analysis (Chapter 4), program execution is represented by a set of synchronization and I/O events. For the data race detection (Chapter 5 & 6), the

[1] Communication primitives such as message send and receive can be similarly defined. In this thesis, we discuss thread interactions only in terms of synchronization for clarity.

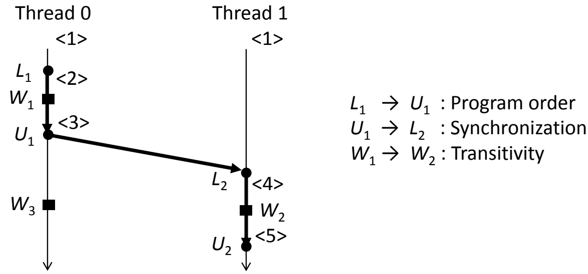


Figure 2.1: An Example of the Happens-before Relation and Logical Clocks in a Program Execution. The events L_1 and L_2, and U_1 and U_2 are lock and unlock operations for the same lock object, respectively. The events W_1 to W_3 are memory write operations. The numbers on the right of each thread are logical clocks updated for the event executions.

execution of a program consists of memory read and write events as well as synchronization and I/O events.

2.2 Logical Clock

In this section, we discuss the logical clock [39] which is the building block for vector clock defined in Section 2.3. We define a clock C_i for each thread i, and C_i is a function which assigns a number $C_i(a)$ to an event a in that thread. The clock C_i is logical rather than a physical clock since we do not assume any physical timing mechanism. For the correctness of the system, we need to ensure that if an event a happens-before another event b, then the logical clock values for events a and b should represent the relation as in a physical clock system. The condition can be stated as follows,

Clock Condition. For any events a and b:

$$\text{if } a \rightarrow b, \text{ then } C(a) < C(b)$$

To satisfy the *Clock Condition*, the logical clock C_i is updated with the following update rule:

- **UR1**. Each thread i increments C_i on every synchronization event.

- **UR2**. (1) If a is a release operation of a synchronization object s (e.g., unlock(s))

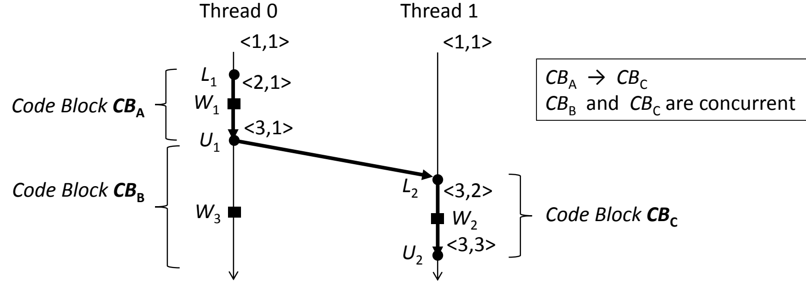


Figure 2.2: An Example of Vector Clocks in a Program Execution. For the same execution as in Figure 2.1, we illustrate vector clocks updates for each thread.

by thread i, then the logical clock of the synchronization object s, C_s, is set to the timestamp of a, i.e., C_s=$C_i(a)$ (2) On the subsequent acquiring operation on the same object s (e.g., lock(s)) by thread j, thread j sets C_j as the maximum of its present value and C_s.

In Figure 2.1, we show an example of happens-before relations with logical clock updates. In the example, since W_1 happens-before W_2, $C(W_1) = 2 < C(W_2) = 4$. Note that the converse of the *Clock Condition* does not hold. Hence, for a given program execution with logical clocks for events, we are not able to find out the partial ordering of events in the execution. In the following section, we show how we can decide if two events are ordered by the happens-before relation given a collection of logical clock information .

2.3 Vector Clock

A vector clock is [14] is an array of logical clocks for all threads. Each thread i has a vector clock T_i and the vector clock is indexed with a thread *id*. For instance, in Figure 2.2, $T_1[0]$ gives the logical clock of thread 0 in thread 1's vector clock, and after event L_2, $T_1[0] = 3$. The execution of a thread is logically divided into code blocks by synchronization operations.

The vector clock for each thread is updated similarly as in the logical clock update

in Section 2.2.

- **VUR1**. On a synchronization operation [2] in thread i, the vector clock entry for the thread is incremented, i.e., $T_i[i]$++.

- **VUR2**. Each synchronization object also maintains a vector clock to convey synchronization information from a releasing thread to the subsequent acquiring thread. At a release operation of object s by thread i, the vector clock for the object s is updated to the element-wise maximum of vector clocks of thread i and object s. Upon the subsequent acquire operation of the object s by thread j, the vector clock for thread j is updated as the element-wise maximum of vector clocks of thread j and object s.

With **VUR1** and **VUR2**, each element of a vector clock contains synchronization information for the corresponding thread. For instance, $T_i[j]$ is the current logical clock for thread j that has been observed by thread i. If there has not been any synchronization from thread j to thread i either directly or transitively, $T_i[j]$ will keep the initialization value.

With the vector clocks in a program execution, we can decide if two events are ordered by the happens-before relation. Consider two vector clocks V_a and V_b for events a and b, respectively. We define "$\sqsubset$" as ordering vector clocks in a element-wise manner such that $V_a \sqsubset V_b$ if for all thread indexes i, the element of V_a is smaller than the element of V_b, i.e., $\forall i : V_a[i] < V_b[i]$. Then, the happens-before relation between a and b can be decided as follows:

$$a{\rightarrow}b, \text{ if } V_a \sqsubset V_b$$

[2] In a vector clock based race detection (Chapter 5 & 6), a thread's vector clock is incremented only for release operations. For the consistent discussion with Section 2.2, we consider all synchronization operations.

As examples, consider the code blocks [3] CB_A, CB_B, and CB_C in Figure 2.2. Let vector clocks of all events in CB_A, CB_B, and CB_C be V_{CB_A}, V_{CB_B}, and V_{CB_C}, respectively. Since $V_{CB_A} =< 2, 1 > \sqsubset V_{CB_C} =< 3, 2 >$, and $W_1 \in CB_A$ and $W_2 \in CB_C$, $W_1 \rightarrow W_2$. Also consider W_3 and W_2. Since $V_{CB_B} =< 3, 1 > \not\sqsubset V_{CB_C} =< 3, 2 >$, and $W_3 \in CB_B$ and $W_2 \in CB_C$, W_3 and W_2 are concurrent.

[3] We discuss memory read/write events with being in code blocks for the sake of simplicity. The detailed descriptions are presented in Chapter 5.

Chapter 3

RECORD AND REPLAY

For program optimization and debugging, dynamic analysis tools, e.g., profiler, debugger, are widely used. To gather execution information, software instrumentation is often employed for its portability and convenience. Unfortunately, instrumentation overhead may change the execution of a program and lead to distorted analysis results, i.e., probe effect. Even without software instrumentation and with the same input data, consecutive runs may result in different executions. In embedded software which usually consists of multiple threads and external inputs, program executions are determined by the timing of external inputs and the order of thread executions. Hence, probe effect/non-deterministic program execution incurred in an analysis of embedded software will be more prominent than in desktop software. This chapter presents a reliable dynamic analysis method for embedded software using deterministic replay. The idea is to record thread executions and I/O operations with minimal record overhead and to apply dynamic analysis tools in replayed execution. In this chapter, we describe our record/replay framework [43, 83], based on Lamport's happens-before relation [39], and show that how dynamic analyses can be managed in the replay execution as if the program execution is deterministic regardless of any instrumentation.

3.1 Introduction

Dynamic program analysis is widely used to collect execution information while programs are running. The approach is widely applied to aid optimization and debugging of the programs. For the collection and analysis of program execution, software

instrumentation is often employed for its portability and convenience. For instance, dynamic binary instrumentation tools such as Intel PIN [29] and Valgrind [61] are most commonly used since they do not require source code and recompilation of program. However, instrumentation overhead from the tools is very high no matter how trivial the analysis is.

The instrumentation overhead can perturb the execution of a program and lead to different execution paths, and consequently misrepresent analysis results. This is so called the probe effect [22]. One example is the delayed executions of input events caused by the instrumentation overhead. Consequently, arriving external inputs may be lost or the deadlines for real-time applications may be missed. The instrumentation overhead may also lead to different order of thread operations on a shared resource and produce different execution results.

In the cyclic debugging process, breakpoints are set in the program and the application is re-run. Thus, the cause of the failure can be observed. As long as each execution is deterministic and repeatable, the cause of the observed failure can be reproduced and identified. However, the program execution can be non-deterministic and non-reproducible as input events can be delayed or O/S internal states (e.g., run-queue state) can be changed due to the pauses of the execution.

In embedded systems, applications run with multiple threads interacting with each other as they share resources, and the execution of threads is often triggered by external inputs. Hence, embedded software will be sensitive to probe effect caused by instrumentation overhead as the timing of input events and the order of thread executions can be easily deformed. On the other hand, probe effect and non-deterministic program execution are less concerns in desktop applications which usually perform predefined work load (e.g., file input) with fixed loop counts. Even if there were execution changes due to the uses of analysis tools, analysis results would be amortized

with repeated executions of the same code, e.g., consistent data races and call-path analysis results. In Tables 1.1 and 1.2, we have shown examples of probe effect caused by instrumentation overhead; the instrumentation overhead has changed the program executions such that the program missed input data and the interaction of threads has been deformed.

To make dynamic analysis tools as non-intrusive as possible, hardware-based dynamic analyses can be used. Intel Vtune Amplifier XE [28] exploits on-chip hardware for application profiling. ARM RealView Profiler [5] supports non-intrusive profiling using dedicated hardware or simulator. However, they are specific for performance profiling and do not support the analyses that require significant processing capabilities. Sampling based analyses [9, 21, 28, 50, 108] can also be used, but the analysis accuracy decreases when sampling rate is reduced to limit any measurement overhead.

For debugging multithreaded programs, one may claim that the existing debugging tools [2, 20, 75] can be used. Although the tools include the functions of setting up breakpoint, single stepping and monitoring at thread level, they do not ensure the reproduction of execution behavior that is required in cyclic debugging. Hence, the developers are responsible for feeding external signals synchronously, and he/she need to manually schedule threads to reproduce the bugs.

Researchers have proposed deterministic replays [7, 24, 41, 42, 52, 68, 76, 77, 97, 104] to reproduce program execution. That is, developers may record a failed run of a program and analyze the failed execution during replay. However, in those works the overhead during recording will be too high such that the recorded execution itself may change the program execution, and none of them have discussed the capabilities and functionalities that replay phase can provide to users.

In this chapter, we present a dynamic program analysis for embedded software using deterministic replay. The idea is to record thread executions and I/O events

with a minimal recording overhead, and the program execution is analyzed with analysis tool/debugger during replayed executions. For this end we have developed a deterministic replay framework [43, 83]. The design goal is to minimize recording overhead to prevent probe effect during recording and to have minimal disturbance on analyzed program executions during replaying. The contributions of this chapter are:

1. We present a dynamic analysis method that makes analyses of a program feasible and faithful, and expedites the debugging and optimization process for embedded programs.

2. We present a record/replay framework that can be incorporated with dynamic analysis tools.

3. We present an overhead analysis of replay which can be incorporated in thread profilers for accurate measurements of thread execution time.

The rest of chapter is organized as follows. In the following section, the dynamic analysis method with deterministic replay is described. Section 3.3 presents the design and implementation of the deterministic replay, and Section 3.4 describes the overhead analysis of replay. In Section 3.5, we present the performance evaluation of the record and replay, and the accuracy of dynamic analysis with the deterministic replay. A concise survey of related work is described in Section 3.6 and we conclude this chapter in Section 3.7.

3.2 Dynamic Analysis with Execution Replay

For dynamic analysis of a program, instrumentation overhead is not avoidable whatever optimization techniques are used. During a cyclic debugging, pausing a thread execution is inevitable to stepping through the thread. The overhead/pausing

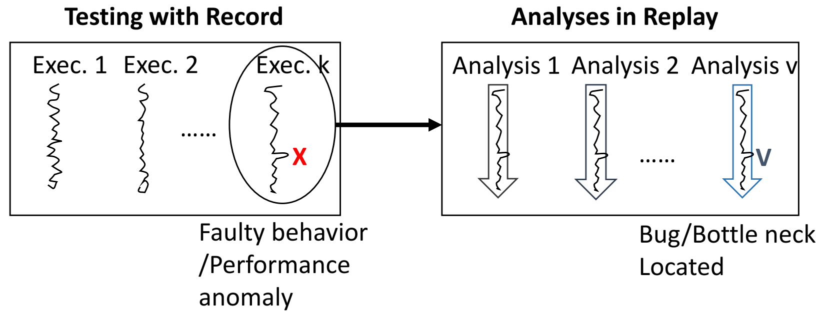


Figure 3.1: Dynamic Analyses with Execution Replay.

can result in different execution behavior. It is possible to repeat the same analysis again and again hoping that eventually we see the true program behavior without the overhead. This repeated running is not even feasible if the program execution depends on the timing of external inputs. Consider the idea of using a record/replay framework for dynamic analysis as shown in Figure 3.1. First, an execution of a program is recorded. If the overhead of recording is insignificant to avoid probe effect by the recording overhead itself, we can assume that the recorded execution is as same as the original program execution. Second, we apply dynamic analyses on the replayed execution which has the same thread executions and I/O events as the recorded one. Thus, the analyzed program will be executed as if there is no execution change by instrumentation. In addition, we can pause and resume any thread during debugging without worrying about execution changes.

Moreover, analysis of program can be expedited with reproducible execution. When we find an unexpected execution during testing of a program, the first step will be to locate the cause of the problem. We may need to run various analysis tools to locate the cause. This will be very time consuming and multiple trials may be needed since it can be hard to reproduce the execution given the significant overhead of the analysis tools. Instead, we record the program execution during the test run. As the execution is reproducible in replayed execution, analysis runs can be performed in the same program execution.

With a deterministic replay, execution times of a thread can be precisely calculated. In *thread profiling*, execution time of each function is measured for each thread. Since there can be probe effect caused by the measurements, researchers have proposed several execution time estimation algorithms [49, 103] that recover the real execution time without the measurement overheads. In the approaches, the real execution time can be recovered with the considerations of three factors: 1) thread local measurement overhead, 2) event reordering, and 3) execution time difference due to the event reordering. As an example of the factors 2) and 3), consider the take and give operations performed on a semaphore. Assume that in the real execution, the semaphore is given first and is taken afterward. Hence there is no blocking. In the instrumented execution, if the semaphore give operation had started late due to the delay of instrumentation overhead, then the thread taking the semaphore would have been blocked. The estimation algorithms [49, 103] can account for the blocking time and then reorder the events. However, if there is a change of program execution path, then there will be no way to recover the real execution.

To avoid the above-mentioned problem, thread profiling can be applied in a replayed execution. Note that the execution with the profiling is deterministic as the event reordering is handled by the replay scheduling. The overhead compensation for the reordering events is no longer needed. Therefore, as long as we can identify the overhead caused by the replay mechanism, the total overhead from the thread profiling tool on a replayed execution is simply the sum of the replay overhead and the thread local measurement overhead from the profiler.

3.3 Record and Replay

In this section, we describe our record/replay framework. The framework is designed to have minimal record and replay overheads. To enable execution replay,

we consider the happens-before relation [39] among events which are execution instances of synchronization or IO function calls. The record/replay operations are implemented in the wrapper functions for events. The happens-before relation over a set of events in a program's execution is logged during recorded execution and is used to guide the thread execution sequence in execution replay.

In the deterministic framework, a data race detector is included as an analysis tool and for managing execution replay in the presence of data races. It also provides an approach for recovering real execution time without measurement overhead of thread profilers (presented in Section 3.4).

3.3.1 Record and Replay Operations

During recording operation, happens-before relations for all events in a program execution are traced and saved into a log file, and the same happens-before relations are to be enforced during execution replay.

A general approach [76] to obtain the happens-before relation is to use Lamport clock [39]. In the approach, a Lamport clock is maintained for each thread and the clock is incremented and synchronized by the happens-before relation as explained in Section 2.2. A timestamp (Lamport clock value) is attached to each recorded event. During the subsequent replay operations, the corresponding event is delayed until all other events having smaller clock values are executed. For instance, consider the execution of event b and let C be a function that returns Lamport clock value for an event. For any other event a, $C(a) < C(b)$ implies that $a \rightarrow b$ and event a has already been executed, or events a and b are concurrent. Therefore, it is safe to execute event b without violation of the happens-before relation. This can enforce a stronger condition than necessary for replaying the partially ordered events as we are also considering events that are concurrent. In our approach, we use a global sequence

number to order the events traced in recording operation. This sequence represents a total order of events and is used to index the set of events during execution replay.

To identify the happens-before ordering, the event log of an event consists of the two sequence numbers for the event itself and the event immediately before it, plus thread id, the function type and arguments for the event. For instance, for event b, let a be the event happened before b immediately. For the execution of event b, the sequence numbers of both events a and b are recorded in the event log of event b. For an input event, the received input is saved into the log file as well. All logging operations are performed in a dedicated logging thread to avoid possible jitters caused by file operations.

In a subsequent replay operation, an event table is constructed from the recorded log. The event table contains a list of events for each thread. Using the sequence numbers in the record log, events are indexed to represent the happens-before relations. Thus, based on the table, the next event for each thread that is eligible for execution can be identified and the same happens-before relations as the recorded ones are used to schedule thread execution.

In our initial version of replay [43], the replay scheduling is performed inside GDB. That is, the GDB thread module is modified to schedule thread executions according to the recorded partial order of events. The GDB command "*set scheduler-locking on*" is used to lock the Linux scheduler and a breakpoint is set at each event for checking happened-before relations. The GDB command "*thread thread-num*" is used for switching to a thread numbered "*thread-num*". This approach can be efficient in debugging of a program as the replay scheduling and the debugging capabilities are in the same software module. However, the uses of replay are limited only for debugging of a program. Hence, in our latest version [83] the replay scheduling is performed inside replay wrapper functions. Thus, the replay operation is implemented purely in

application level, and the uses of replay is as general as possible.

In our latest version of replay, a replay wrapper function for each event (e.g., pthread_mutex_lock()) consists of three parts. Firstly, a thread looks up the event table for the events happened before its current event. If any of the events that should be happened before are not executed, the thread is suspended waiting for a conditional variable. In the second part, when a thread can proceed with its current event, the thread carries out the original function. If the function is for an input event, the thread reads input value from the log file instead of actual reading from an I/O device. Lastly, the event for this wrapper function is marked as executed and the thread wakes up (signal) other threads waiting for the execution of this event. Note that the total order of events based on the sequence numbering is used only for indexing events and the replay operation follows the same partial order as in the recorded execution.

3.3.2 Handling Data Races

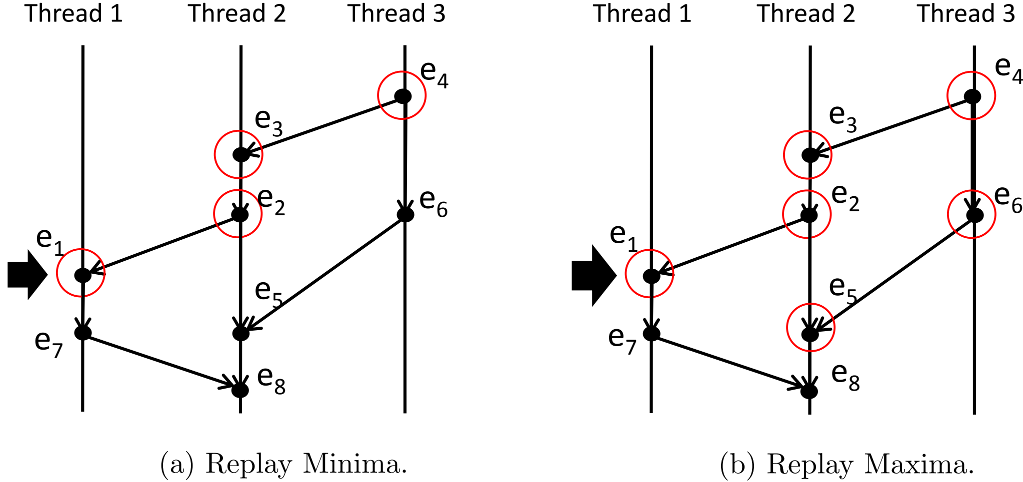
For efficient record and replay operations, only synchronization and I/O events are considered. However, one drawback is that a replay may not be correct in the presence of data races. A data race occurs when a shared variable is accessed by two different threads that are not ordered by any happens-before relation and at least one of the accesses is a write. Assume that, as an example, in the recorded execution there is a data race on a variable that is used to decide alternative execution paths. The presence of the data race implies that the order of accessing the variable is not recorded. Hence, if an alternate accessing order from the record one is chosen in replayed execution, the replayed program may take a different execution path, and the replay is not reproducing the same execution as the record anymore.

We use a similar approach as RecPlay [76] to handle the presence of data races

in replaying operations. RecPlay records an execution of a program as if there is no data race and any occurrences of data races are checked during replay operation using a data race detection tool. If data races are found, the races are removed and record and replay operations are repeated. The approach is correct since a replayed execution is correct up to the point where the first race occurs as shown in [10]. However, a data race detector incurs a substantial runtime overhead. Note that not every data race causes execution path changes. Hence, it may not be practical to fix every data race during replaying operations.

Instead of replaying a race-free program, our replay detects the occurrence of an unexpected event. The detection of an unexpected event is done in the replay wrapper by comparing the current events with the events in the record log. The detection of an unexpected event stops the replay with the location of the unexpected event. Then, the race can be detected with a data race detector and fixed. After fixing the race that results in different execution path, the replay can be safely used with various analysis tools including a data race detector for detecting races that cause errors other than a change of execution paths.

We have implemented a data race detector for C/C++ programs based on FastTrack [19] and the detector is modified to be integrated with our deterministic replay. The FastTrack algorithm detects data races based on happens-before relations in a program execution. However, the replay scheduler invokes additional locking operations which appear as extra happens-before relations for the FastTrack detector. As a consequence, some data races may not be detected in the replayed execution. To correct the missed detections, we alter the FastTrack approach to discount the synchronization operation introduced by the replay scheduler.



(a) Replay Minima. (b) Replay Maxima.

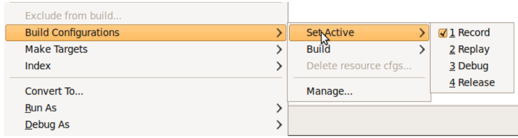
Figure 3.2: Examples of Replay Minima/Maxima. (a): replay minima on event e_1. Only events e_2, e_3, and e_4 are executed since they are the minimum set of events to reach $e1$. (b): replay maxima on event e_1. All events except e_7 and e_8 are executed since events e_7 and e_8 can be executed only after event e_1 is executed.

3.3.3 Debugging Support

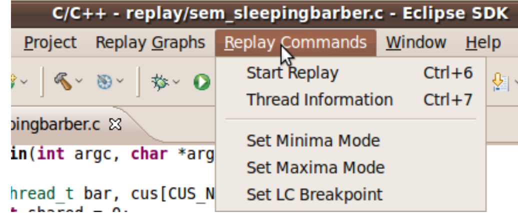
During a replay, the execution follows the same partial order graph of events as in the recorded execution. Hence, given the partial order graph, there can be many different orders of event executions. We provide two debugging modes for more controllable executions of threads:

1. ***Replay Minima***: For a selected event in a thread, the events of other threads will be executed only if they are happened before the selected event. That is, only the minimum amount of executions to reach the selected events are executed.

2. ***Replay Maxima***: For a selected event in a thread, other thread can proceed until there is a happens-before dependency on the selected event. That is, the maximum amount of executions before the selected events are executed to see how far the impact of the selected event can stretch.



(a) Compiling for Record/Replay



(b) Menu of Replay Commands

Figure 3.3: Debugging Support in Eclipse.

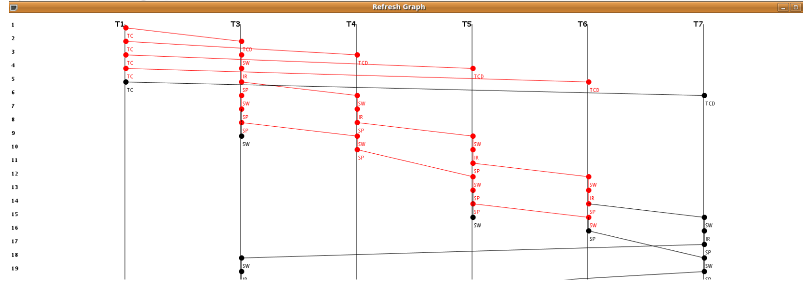


Figure 3.4: Displaying Events and Threads in Eclipse. The executed events are colored in red. The line between two events represents the happens-before relation.

Examples of replay minima/maxima are shown in Figure 3.2. The two debugging modes are incorporated into GDB by adding two GDB commands, "*set replay_minima*" and "*set replay_maxima*". GDB communicates with the record/replay library through log files for the control of each event.

We provide a graphical environment in the form of Eclipse [88] plug-ins. The Eclipse plug-ins includes interfaces to automate compiling process for record/replay and to support the two debugging modes. The Eclipse environment also displays threads and events during a debugging session. Figures 3.3 and 3.4 shows the Eclipse supports for debugging.

3.4 Execution Time Estimation

In this section, we present an approximation approach to estimate execution time that can account for the overheads of replay operation and thread profilers. First, we present the algorithm that can estimate the real execution time without replay at

program level. Second, the algorithm is refined to estimate the real execution time at thread level without the overheads of profilers and replay operation. The replay overhead is approximated based on the assumptions that 1) threads run on multiple cores, 2) events are evenly distributed to each core, 3) the record overheads for all event types are same, and 4) the number of worker threads is greater than or equal to the number of cores.

3.4.1 Execution Time Estimation at Program Level

The estimated execution time of a program execution, $C_{estimate}$, can be calculated by subtracting the replay overhead O_{replay} from the replayed execution time C_{replay}, such that,

$$C_{estimate} = C_{replay} - O_{replay} \tag{3.1}$$

The replay overhead O_{replay} is the sum of 1) replay initialization time C_{init}, 2) extra execution time (CPU time), C_e, spent in replay wrapper functions, and 3) blocking time, B_u, by the replay scheduling that leads to extra delay of replay execution, such that,

$$O_{replay} = C_{init} + C_e + B_u \tag{3.2}$$

The replay initialization is processing the log file and preparing the event table before the replayed program runs. Hence it runs on a single core.

The extra execution time C_e is the sum of overheads for two different cases of event executions. The replay has two different program paths for an event execution depending on whether the event can proceed without blocking or not. For instance, consider two events a and b with $a \rightarrow b$. The execution of event b can be *delayed* until event a is executed, or can be executed with *no-delay* if event a has already been executed. The two possible ways of event executions contribute to various amount of overheads, thus we account them differently. Let n_{nd} and n_d be the numbers of events

of *no-delay* and *delayed* executions, respectively. Let c_{nd} and c_d be the execution time for a *no-delay* and the execution time for a *delayed* event, respectively. Then, C_e is expressed as,

$$C_e = n_{nd} * c_{nd} + n_d * c_d \tag{3.3}$$

The replay overhead should also include extra blocking time (other than extra execution time) by the replay scheduling. Threads can be blocked by the replay scheduling in two ways: 1) the global locking used by the replay scheduling blocks threads, and (2) if a thread tries to execute an event and the partial ordering becomes different from the record as the result of the event execution, then the reply scheduler blocks the thread (delayed event). The blocking of threads leads to additional execution time only when the number of ready threads becomes less than the number of cores, i.e., when the cores are underutilized. Let n_g be the number of occurrences that threads are blocked by the global locking and let b_g be the average delay caused by each global locking. Also, let b_d be the total execution delay caused by the delayed events due to the replay scheduling at the end of a replay execution (i.e., the sum of blocking times of events). Then, B_u is expressed as,

$$B_u = n_g * b_g + b_d \tag{3.4}$$

Combining Equations 3.3 and 3.4 into Equation 3.2 gives,

$$O_{replay} = C_{init} + n_{nd} * c_{nd} + n_d * c_d + n_g * b_g + b_d \tag{3.5}$$

3.4.2 Overhead Measurements

The replay initialization time C_{init}, and the counter values, n_{nd}, n_d, and n_g, can be measured during a replay execution. The blocking delay, b_d, is calculated using the algorithm in Figure 3.5 based on the utilization of the cores. On the other hand, the per event overheads, c_{nd}, c_d, and b_g, are hard to measure online in the execution on

```
n = number of runnable threads
M = number of cores

//At start of every delayed event
if ( n==M ) //start measurement
   ts = get_timestamp();
else if (n < M)
   tmp =  get_timestamp();
   bd += (tmp- ts )*((M-n)/M);
   ts = tmp;
n--;

//At end of every delayed event
if ( n < M )
   tmp =  get_timestamp();
   bd += (tmp- ts )*((M-n)/M);
   ts = tmp;
n++;
```

```
//At start of event delay in thread T
ts-T = get_timestamp();

//At end of event delay in T
te-T = get_timestamp();
bT += (te-T - ts-T);

int blocking_overhead(thread T)
{
    if T is not blocking from replay
        return bT;
    else
        return \
          (bT + get_timestamp()-ts-T);
}
```

Figure 3.5: Calculation of Blocking Time of Events by the Replay Scheduling. (Left): the time measurement in which the cores are underutilized due to the delayed events by the replay scheduling. (Right): the measurement of blocking overhead for a thread T.

multi-core systems. To avoid the online measurement, we assume they can be viewed as constants for a given system and can be estimated offline using measurement programs.

The measurement program for c_{nd} consists of multiple threads which invoke their own locks. Thus, threads are independent of each other and there is no overhead from delayed events. Hence, $n_d * c_d = 0$ and $b_d = 0$. To avoid any idle CPU core caused by the global locking, extra threads running busy loops and with no synchronization and IO event are added. Hence, the blocking overhead from the global locking becomes zero ($n_g * b_g = 0$) since all cores are utilized fully. The execution time without replay, C_{m1}, is measured and we can assume that $C_{m1} = C_{estimate}$. Then, with Equations 3.1 and 3.5, c_{nd} can be calculated using the following equation:

$$C_{m1} = C_{replay} - (C_{init} + n_{nd} * c_{nd}) \tag{3.6}$$

A similar program is used for measuring c_d, where a lock is shared among all

threads. Hence, $n_d * c_d \neq 0$. Using the extra threads with busy loops, the program keeps $n_g * b_g = 0$ and $b_d = 0$. Assuming that the execution time without replay, C_{m2}, is equal to $C_{estimate}$, c_d can be calculated using the following equation:

$$C_{m2} = C_{replay} - (C_{init} + n_{nd} * c_{nd} + n_d * c_d) \tag{3.7}$$

The remaining constant b_g is calculated using a similar measurement program for measuring c_d but without the extra threads and with busy loops.

3.4.3 Execution Time Estimation at Thread Level

In this subsection, we present how we can estimate per-thread replay overhead based on the estimation algorithm presented in the previous Subsections 3.4.1 and 3.4.2, and show how the overhead from a thread profiler can be calculated based on the per-thread overhead estimation. In thread profiling, the execution times of threads' activities such as function executions can be measured and analyzed for each thread. As described in Section 3.2, the measurement overhead from the thread profiler consists of 1) thread local overhead and 2) execution time differences due to event ordering. When the profiler runs in a replayed execution, the latter overhead is contained in the replay overhead for delayed events ($n_d * c_d + b_d$). Therefore, the total overhead from the profiling on a replay execution is simply the sum of the replay overhead and the thread local overhead of the profiler. To estimate the real execution time without the overheads, we need a per-thread measurement at a given time instant. For instance, if a function in thread T is invoked at t_s and finishes at t_e, then the execution time of the function is measured as $t_e - t_s$. Let $C_{estimate-T}$ be a function that returns the estimated execution time of thread T up to a given time instance. Then, the real execution time of the function can be estimated as, $C_{estimate-T}(t_e) - C_{estimate-T}(t_s)$.

We can start the measurements after the initialization of a replay execution, thus

$C_{init} = 0$. All counter values (n_{nd}, n_d, and n_g in Equation 3.5) are maintained for each thread. Note that all per-event measurements (c_{nd}, c_d, and b_g in Equation 3.5) are measured for concurrent executions on multiple cores. Hence, the per-event measurements for each thread are approximated considering the number of cores in the system. Let M be the number of cores in the system. Then, the per-event measurements for each thread, denoted as c'_{nd}, c'_d, and b'_g, can be approximated as the product of M and c_{nd}, c_d, and b_g, respectively. The blocking delay, b_d, is replaced with accumulated blocking time for each thread and it can be calculated as shown in Figure 3.5. Then, the replay overhead in thread T at a given instant t can be represented as,

$$O_{replay-T}(t) = n_{nd-T}(t) * c'_{nd} + n_{d-T}(t) * c'_d + n_{g-T}(t) * b'_g + b_T(t) \tag{3.8}$$

Let the thread local overhead from the profiler in thread T up to a given instant t be $O_{profile-T}(t)$. Then, the estimated execution time for thread T at time t can be expressed as,

$$C_{estimate-T}(t) = t - (O_{replay-T}(t) + O_{profile-T}(t)) \tag{3.9}$$

3.5 Evaluation

In this section, we show the effectiveness of the analysis approach in replay execution through several benchmark experiments. First, we show the overheads of the record and replay operations. Second, we present the evaluation results of the execution time estimation algorithm. Lastly, the accuracy of dynamic analyses using the record and replay is presented. All experiments were performed on an Intel Core Duo processor running Ubuntu 12.04 with kernel version 3.2.0.

The two programs shown in Section 1.1 are used to illustrate the effect of the minimized probe effect in record phase. All other experiments were performed with

11 benchmark programs for desktop computing to reveal the efficiency and accuracy of the dynamic analysis methods performed in the replay phase. Out of the 11 programs, 8 programs are from the PARSEC-2.1 benchmark suite [8] and 3 programs are from popular multithreaded applications: *FFmpeg* [89], a multimedia encoder/decoder; *pbzip2* [30], a parallel version of bzip2; and *hmmsearch* [15], a sequence search tool in bioinformatics.

3.5.1 Overheads of Record and Replay Operations

Table 3.1 shows the overheads of the record and replay operations in our deterministic replay. "Number of events/sec" column shows the number of recorded events per second in the execution of each benchmark program. The column shows a general idea of how much the overheads will be in the record/replay executions as the record/replay operations are performed in the wrapper functions of the events. The overheads of record and replay operations are 1.46% and 2.78% on geometric mean, respectively. The results suggest that the record/replay will be suitable for dynamic program analysis of embedded software. The record operation would not have intrusion on the execution, and analysis results during replay will have insignificant amount of interferences (e.g., extra instructions counted from replay operation for instruction counting profiling) due to replay operation. One exceptional case is *fluidanimate* which incurs noticeable record/replay overheads due to a large number of events in the execution.

Tables 3.2 and 3.3 are the revisits of Tables 1.1 and 1.2 with the additional measures collected in record phase. For fair comparisons for the QT application, the QT libraries are not instrumented/analyzed for all analyses including our record operation, as there is no event to record in the QT libraries. The results in both tables suggest that the recording operation does not have intrusion on the executions (i.e.,

	Benchmark Program	Base time (sec)	Record (sec)	Replay (sec)	Number of events/sec	Record Overhead	Replay Overhead
PARSEC	facesim	6.050	6.054	6.055	2,200.5	0.07%	0.08%
	ferret	5.027	5.073	5.161	2,066.2	0.92%	2.67%
	fluidanimate	2.054	3.620	4.887	2,163,267.3	76.24%	137.93%
	raytrace	9.823	9.843	9.813	9.8	0.20%	-0.10%
	x264	2.196	2.288	2.323	17,487.2	4.19%	5.78%
	canneal	6.643	6.674	6.677	1.5	0.47%	0.51%
	dedup	7.750	8.208	8.711	75,623.9	5.91%	12.40%
	streamcluster	3.781	4.071	4.092	38,417.1	7.67%	8.23%
	ffmpeg	3.053	3.052	3.157	3,560.4	-0.03%	3.41%
	pbzip2	5.297	5.396	5.381	622.2	1.87%	1.59%
	hmmsearch	26.550	26.624	26.699	3,644.7	0.28%	0.56%
Geometric mean						1.46%	2.78%
Average						9.78%	17.31%

Table 3.1: Record and Replay Overheads

inputs/sec	Native execution	Record	PIN Cache	FastTrack
150	0.0%	0.0%	16.8%	0.3%
300	0.0%	0.0%	36.1%	1.2%
450	0.0%	0.0%	45.5%	1.9%

Table 3.2: Revisit of the QT Application (Table 1.1) with Recording Operations.

no probe effect).

3.5.2 Execution Time Estimation

Based on the overhead analyses in Section 3.4, the replay overhead for each benchmark can be measured and calculated with Equation 3.5 and the real execution time without the replay overhead can be estimated using Equation 3.1. Table 3.4 shows the estimated execution times and errors. Column 2 and 3 list the measured times

Queue Length	Native execution	Record	PIN Cache	FastTrack
5	1.3/7.5	1.3/8.1	8.3/191.9	5.5/56.3
10	0.5/8	0/7.1	2.5/146.8	2.4/37.9

Table 3.3: Revisit of the MQ Application (Table 1.2) with Recording Operations

Benchmark Program	Base time (sec)	Replay (sec)	Estimation (sec)	Error
facesim	6.050	6.055	6.041	-0.2%
ferret	5.027	5.161	5.122	1.9%
fluidanimate	2.054	4.887	2.144	4.4%
raytrace	9.823	9.813	9.812	-0.1%
x264	2.196	2.323	2.247	2.3%
canneal	6.643	6.677	6.677	0.5%
dedup	7.750	8.711	7.838	1.1%
streamcluster	3.781	4.092	3.859	2.1%
ffmpeg	3.053	3.157	3.052	0.0%
pbzip2	5.297	5.381	5.378	1.5%
hmmsearch	26.550	26.699	26.595	0.2%
Average				1.24%

Table 3.4: Execution Time Estimation without Replay Operation.

of the application execution without and with the replay, respectively. Column 4 show the estimated execution times with Equation 3.1. Column 5 lists the estimation error for each program, i.e., the difference between columns 2 and 4. On average, the estimation error is only 1.24%.

The replay overhead is classified into the 5 categories as shown in Equation 3.5. In Figure 3.6, the relative overhead in the 5 categories is illustrated for the four benchmarks that have more than 5% of replay overhead: *fluidanimate*, *x264*, *dedup*, and *streamcluster*. In the chart, the items are correspondent to the five categories of Equation 3.5 in the order. Two applications, *dedup* and *streamcluster*, show relatively more percentages of blocking overhead caused by delayed events (around 50%) than the other two applications. This is because there have been more per-event disturbances caused by the replay scheduling than the other two applications.

3.5.3 Accuracy of Analysis in Replay Execution

In this section, we present experimental results to show the accuracy of dynamic analysis with replay execution. Once we have a recorded execution which is as same as the original execution, the analysis result in replay execution will be as close

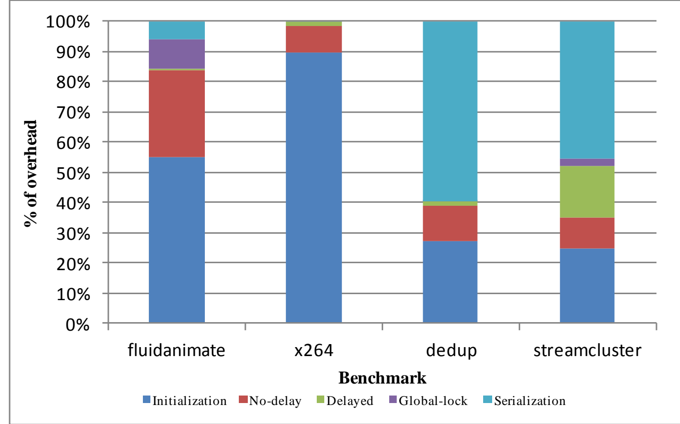


Figure 3.6: Decomposition of the replay overhead is shown only for benchmarks that have more than 5% replay overhead

Benchmark Program	Slowdown		Number of races detected
	Without replay	With replay	
facesim	140	139	8907
ferret	86	82	2
fluidanimate	85	116	1
raytrace	27	27	13
x264	74	77	1300
canneal	12	12	0
dedup	151	148	0
streamcluster	244	251	1053
ffmpeg	120	120	1
pbzip2	68	69	0
hmmsearch	83	86	1
Average	99	103	

Table 3.5: Data Race Detection with FastTrack.

as the analysis without any instrumentation. Since it will be very hard to know what the analysis results will be without any instrumentation [1] , it will be difficult to compare results using real embedded software (i.e., non-deterministic program with I/O events). Instead of using embedded software, we use the 11 benchmark programs (which are for desktop computing) and assume that the programs show no (or negligible) probe effect from the instrumentation. Therefore, in the experiments we consider the execution without replay as the original execution.

[1] We present the idea of revealing the original execution without any instrumentation in Chapter 4

Benchmark Program	Slowdown		# of different function entries
	Without replay	With replay	
facesim	37	37	0
ferret	50	50	0
fluidanimate	59	127	7
raytrace	59	59	0
x264	77	79	0
canneal	25	24	0
dedup	41	43	0
streamcluster	58	52	4
ffmpeg	66	66	0
pbzip2	51	50	1
hmmsearch	28	29	0
Average	50	56	

Table 3.6: Flat Profiling Comparison with Callgrind.

Benchmark Program	Slowdown		Cache miss rate - Without replay			Cache miss rate - With replay		
	Without replay	With replay	L1-Instruction	L1-Data	L2-Unified	L1-Instruction	L1-Data	L2-Unified
facesim	472	462	0.00%	5.92%	12.02%	0.00%	5.92%	12.07%
ferret	336	341	0.02%	4.73%	16.89%	0.02%	4.74%	16.14%
fluidanimate	398	731	0.00%	**1.15%**	17.13%	0.00%	**1.58%**	18.72%
raytrace	395	387	0.00%	2.77%	2.23%	0.00%	2.77%	2.23%
x264	411	424	1.34%	4.07%	**8.10%**	1.32%	4.07%	**9.62%**
canneal	92	91	0.00%	4.06%	43.46%	0.00%	4.08%	43.12%
dedup	286	301	0.00%	4.06%	7.25%	0.01%	3.93%	7.66%
streamcluster	395	454	0.00%	6.55%	49.14%	0.00%	6.42%	48.95%
ffmpeg	478	478	0.01%	4.52%	11.56%	0.01%	4.61%	11.26%
pbzip2	377	381	0.00%	4.03%	13.54%	0.00%	4.03%	14.40%
hmmsearch	498	475	0.00%	0.78%	7.14%	0.00%	0.78%	7.10%
Average	376	411						

Table 3.7: Cache Simulation Results from PIN Cache.

Table 3.5 compares analysis results of the FastTrack race detector with and without replay operation. As shown in the table, the two approaches locate the same data races. For *facesim*, there was a data race that may result in different execution paths. Our replay has detected the unexpected event diverging from the recorded one, and stops the replay execution. Then, the race is detected with the FastTrack detector and fixed with correct synchronization. The data in the table shows the detection result after fixing the race for *facesim*.

Table 3.6 compares the flat profiling results from the Callgrind profiler [93]. The

flat profiler lists the functions in decreasing order that have fetched most instructions. For each benchmark program, we compare the top 10 function entries from normal program execution and replay execution, and the number of different function entries is shown in the last column of Table 3.6. There are three cases showing different function entries. For *streamcluster*, the function entries are different from the 6th entries due to the functions invoked for replay execution. However, from the 3rd function in the list, the functions have used less than 1% of total instructions. Similarly, the 10th function entries are different in *pbzip2* which fetches a negligible percentage of instructions. For *fluidanimate*, it shows 7 different function entries since the benchmark program has high replay overhead with the replay library functions included in the result. However, after removing the replay library functions, the same function entries are shown in the same order.

Table 3.7 shows the comparisons of PIN Cache simulator running in normal program execution and in replay execution. As can be seen from the comparisons, the differences are negligible. There are only two measures of cache miss rates with a more than 10% discrepancy between normal program execution and replay execution.

3.6 Related Work

Instant Replay [41] is one of the earliest works that support deterministic replay on a multiprocessor. It records and replays a partial order of shared memory accesses. Microsoft's iDNA [7] also trace every shared access to enable deterministic replay. However, monitoring every shared memory access is too expensive (over 10x slowdown).

Due to the high overhead of instrumenting all shared memory accesses, researchers have proposed replay systems that reduce the scope of programs that can be replayed [42, 76, 77, 97]. RecPlay [76, 77], which is most similar to our system, logs only syn-

chronization operations. As in our system, this approach only ensures a deterministic replay up until the first race in the program. However, the replay overhead is too high (around 2x) for uses with dynamic analyses.

As an effort to avoid probe effect, several hardware-base dynamic analysis tools have been proposed. To detect data races, CORD [72] keeps timestamps for shared data that are presented in on-chip caches. With simplified but realistic timestamp schemes, the overhead can be negligible. DAProf [59] uses separate caches for profiling of loop executions. In the approach, short backward branches are identified and profiling results are stored in the cache. Both approaches are non-intrusive with acceptable accuracies of analysis results. However, the requirement of extra hardware mechanisms may make the approaches impractical.

Moreno et.al [53] has proposed a non-intrusive debugging approach for deployed embedded systems. In the approach, the power consumption of a system is traced and matched to sections of code blocks. Thus, a faulty behavior in the execution of the code blocks can be identified only with an inexpensive sound card for measuring power consumption and a standard PC for the power trace analysis.

Respec [42] logs systems calls as well as low-level synchronization operations. At the end of every code segment, a failed replay due to data races is checked by examining any deviation in system call output or program state. When a failed replay is detected, Respect rollbacks the execution and retry the failed execution. Respec supports a deterministic replay even in the presence of data races. However, the recording overhead is still high for embedded software (>50%) and it requires modifications on O/S and shared libraries which may change the original program execution.

DoublePlay [97] provides an efficient deterministic replay system by exploiting spare cores. An execution of a thread is time-sliced into program intervals (epochs)

and the same epoch of threads is executed on the same core in a pipelined manner. It also support checkpoint (rollback and retry) and requires speculative execution. However, it requires at least 2x as many cores for efficient record and replay, and as in Respec significant modifications on O/S and shared libraries are required.

3.7 Chapter Conclusions

In this chapter, we have presented how we should carry out dynamic analysis for embedded software. First, we record a program execution with record operation to avoid execution changes caused by instrumentation overhead (i.e., probe effect) and non-deterministic program execution. Then, dynamic analyses are performed in replayed executions without worrying about execution changes. For the analysis approach, we have proposed the deterministic replay framework. The record operation has low overhead to prevent possible probe effect caused by the recording operation itself, and the deterministic replay has low overhead during reply to minimize execution disturbances on the analysis results.

The experimental results show that our approach is viable. However, the approach has to be supplemented in two ways. First of all, even if the record overhead is minimal, it is still possible that the overhead may have changed the program execution. In Chapter 4, we propose a simulation-based approach to decide whether any instrumentation (including our recording operation) has changed the program execution or not. Secondly, we need to have efficient ways to detect data races in a program. To make the record/replay efficient, we only consider synchronization and I/O events. However, if there are data races in the execution that deviate the execution path, our replay might fail. For this end, in Chapter 5 and 6 we propose efficient data race detection for C/C++ programs on multi-core machines.

Chapter 4

PROBE EFFECT ANALYSIS

Software instrumentation has been a convenient and portable approach for debugging or profiling of program execution. Unfortunately, instrumentation may change the temporal behavior of multi-threaded program execution and result in different ordering of thread operations, which is called probe effect. While the approaches to reduce instrumentation overhead, to enable reproducible execution, and to enforce deterministic threading have been studied, no research has yet answered if an instrumented execution has the same behavior as the program execution without any instrumentation and how the execution gets changed if there were any. In this chapter, we present a simulation-based analysis [85] to detect the changes of execution event ordering that are induced by instrumentation operations. The execution model of a program is constructed from the trace of instrumented program execution and is used in a simulation analysis where instrumentation overhead is removed. As a consequence, we can infer the ordering of events in the original program execution and verify the existence of probe effect resulted from instrumentation.

4.1 Introduction

In real-time embedded systems, application tasks usually run in concurrent threads. Threads may interrelate with each other as they share resource and data. They also interact with the external environment to receive sensor data and to control actuators. While the threads are running, any instrumentation to observe program execution behavior will introduce extra overhead to the execution. Instrumentation overhead, no matter how small it is, may intrude and change the execution behavior of the pro-

gram and, consequently, introduce probe effect [22, 38]. Hence the observed behavior through instrumentation is not guaranteed to represent the original program behavior.

Instrumentation operations can perturb program execution in two ways [38]. First, the occurrence of an execution event is delayed by the amount of time spent on running instrumented code. This can change the timing of interacting with other threads and external environment (e.g., reading an input). Second, due to the changes of the timing of invoking guarded resources and critical sections, scheduling decision can be different. This, in turn, can lead to the variations in the sequential order of accessing shared resources. Therefore, the timing perturbation by instrumented code can result in a different happened-before ordering of events [39] and possibly a different program execution path from the original program. The other related issue is that we may not be able to know whether there is any change on program execution paths caused by the timing perturbation.

To obtain a proper observation of multi-threaded program execution through instrumentation, it is important to know the effect of timing perturbation. However, unless we adopt hardware-based monitoring, it might not be possible to know the exact execution behavior of a program. It may be argued that a comparison of computation results from instrumented and un-instrumented programs can reveal any different behavior of program execution. Note that some benign program behavior may not affect the final computation results, for instance, a branch decision can be caused by different conditions in a compound conditional expression. In addition, for embedded systems, it may be tricky to manage identical external inputs arriving at the precise instants of the execution. Another approximation is to measure the overhead of instrumentation, calculate the execution time by removing the overhead, and infer the real execution. In a single thread program, this approach would be feasible. However, in a multi-threaded program it is extremely difficult to consider all thread

interactions when thread execution time is changed. Moreover, it is problematic to take into account kernel states (e.g., run-queue state) which may affect scheduling decision.

There have been several approaches to recover the performance of parallel programs by compensating the instrumentation overhead [49, 103], but they do not examine the ordering of the program execution. On the other hand, deterministic replays [7, 24, 41–43, 52, 68, 76, 77, 83, 97, 104] provide reproducible execution that guarantees the same execution ordering as the one observed in a recording operation. It is perceivable that any recording operation should incur some instrumentation overhead since the execution of the recording itself would have caused perturbation to the original execution. Nonetheless no research on deterministic replay examines the issue of any changes to the original program execution behavior caused by the recording operation.

In this chapter, we present a simulation-based analysis for embedded software to detect any variations of event ordering caused by instrumentation overhead. It is assumed that application tasks are performed by concurrent threads in a priority-based preemptive scheduling system. We also assume that the program is data race free. The analysis starts with an instrumented program (e.g., for dynamic program analysis) from which we want to analyze the impact of instrumentation overhead to the program execution. We add trace codes into the instrumented program to obtain traces for the simulation-based analysis including thread interaction events and the overhead of instrumentation code. All the other activities that can affect a program execution such as O/S events and external inputs are also traced. From the traces of the instrumented program execution, we construct a simulation of the program execution where the instrumentation overhead is removed. Timings of thread events is calculated which decides the ordering of events in the simulated run. Then,

the ordering information of the original program execution with no instrumentation overhead is projected. The contributions of this chapter can be summarized as follows:

1. We provide a novel way of detecting execution changes of a program caused by instrumentation overhead.
2. Timing deciding program execution (i.e., ordering of events) is accurately simulated with the consideration of all factors affecting the program execution, including kernel activities, external inputs, as well as thread execution times.
3. We provide an analysis framework for inferring the original program execution based on simulation results.

The rest of chapter is organized as follows. In Section 4.2, we discuss execution of multi-threaded program, and based on the discussion we present the modeling of multi-threaded program execution in Section 4.3. In Section 4.4, we present the simulation design to reveal the original program execution without instrumentation overhead. Section 4.5 provide the analysis framework for inferring the original program execution based on simulation results. The implementation details are explained in Section 4.6, and Section 4.7 presents the experimental results. A concise survey of related work is presented in Section 4.8, and we conclude this chapter in Section 4.9.

4.2 Multi-threaded Program Execution

Multi-threaded program execution can consist of a set of thread interaction events. Such an event, as a sequence of instructions that the program executes, defines a particular action (e.g. a system call) to interact with other threads, internal and external environment. Examples include synchronization, communication operations, and IO read/write calls. The events can be totally ordered by the timestamps at which

the events take place. A partial order can also be defined among the events based on logical dependencies, i.e., happened-before relation [39].

Let's consider multiple runs of a program and the ordering of interaction events from each run. If the execution events happen at different instants, the happened-before ordering of the events may be different from one run to the other. This may lead to a change of program execution behavior. For instance, in the sleeping barber problem, customers may be served in different orders when they arrive at different instants in separate runs. On the other hand, a particular customer may find out the waiting room is full and miss the service in one run. In the other run, if the barber cuts hair quickly, the customer can find a seat in the waiting room and receive the service eventually. In this case, the execution path of at least one thread is changed which results in a different timestamp-based ordering of events at thread level.

It is a well-known principle used in deterministic replays [43, 76, 77, 83] that, for two runs of a data race free program, if we supply the same input data and have the same happened-before ordering of events, then the two runs must result in the same behavior. Conversely, if we observe two distinguished runs of a program, then either inputs and/or the happened-before ordering of one run must be different from those of the other run.

During an execution of a program, a happened-before ordering of program events can be dependent upon the external inputs it receives, including input data value and the timing that new data arrives. If we have the same inputs, the happened-before ordering of the program events is decided by the execution instants of threads events on shared resources and communication. For example, when two threads compete for a resource, the happened-before ordering of the locking events will be decided by the instants that the two threads issue their resource requests. The choice on who is going to take the resource first may also depend upon the kernel's scheduling policy, e.g.,

priority based or FIFO, as well as the kernel's internal states, including run-queue state and other tasks running in the system including interrupt handlers. So, when a program is instrumented, we can expect that more CPU time is spent to execute instrumentation code and the execution time of each thread becomes longer. This may have a ripple effect on the instants that program events may take place, and the happened-before ordering of the events.

4.3 Model of Multi-threaded Program Execution

To model multi-threaded program execution, we assume there are n concurrent threads, thread T_i for $i = 1, , n$, in an embedded program. The threads are data race and exception free and are scheduled preemptively according to their priorities. The system state is the collection of thread local states and a shared global state. The interactions among the threads and with the external environment are done through the operations on the global state, which are represented by interaction events. An interaction event (abbreviated as event), e, can be a lock/unlock, semaphore give/take, message send/receive, or input/output operation that is performed when a thread invokes an event function f. The following notation is used to indicate that an event e is generated during the execution of the event function f:

$$Ex : f \rightarrow e$$

Apparently, the resulting event of an invocation of f depends upon the local state of the calling thread, as well as the shared global state. For instance, a non-blocking read from a device can succeed or fail depending on the availability of input data when the function is invoked.

There are two important incidents during the execution of an event function f by thread T. The two timing incidents define a partial ordering of events for the program execution.

- "T ***enters*** f" occurs when the processing begins to take place globally. The entrance gives an important timing information since it decides a logical order between events, as well as the possible resultant event to be generated by the function. For instance, when two threads request a semaphore concurrently, the moments that they enter *sem_wait* function provide an order of the requests and can determine which thread can take the semaphore successfully and the consequent global state.

- "e ***happens***" symbolizes that the result of execution, as event e, is posted and is available to subsequent execution. Obviously, the invocation of event functions by a thread form a sequence and an event happened previously can causally affect any subsequent events [39].

To include OS and device activities in the model, a system thread, T_0, is added. The events that occur in T_0 consist of interrupts, OS scheduling, and the arrivals and updates of device input data. Thus, an interrupt event of T_0 may set a semaphore and wake up a waiting application thread. Also, an application thread can read in the latest sensor data if the read operation happens after a data update event in T_0.

To define the occurrence instants, we adopt two types of clock. Clock $\boldsymbol{C}$ indicates the CPU cycles used globally. Starting from 0, it is advanced for all activities consuming CPU cycles, including thread executions, OS activities such as interrupt handler and scheduler, and idle process. In addition, $\boldsymbol{C_i}$ is the thread local clock for thread T_i and is advanced only during the execution of T_i. Hence $\boldsymbol{C_i}$ represents the CPU time spent on the execution of T_i. With the clocks, we define functions that return the clock values:

- $\boldsymbol{ht}$ is a function that returns the $\boldsymbol{C}$ clock values for event function f and event e. That is, $\boldsymbol{ht}(f)$ and $\boldsymbol{ht}(e)$ return the $\boldsymbol{C}$ clock values when f is invoked and e

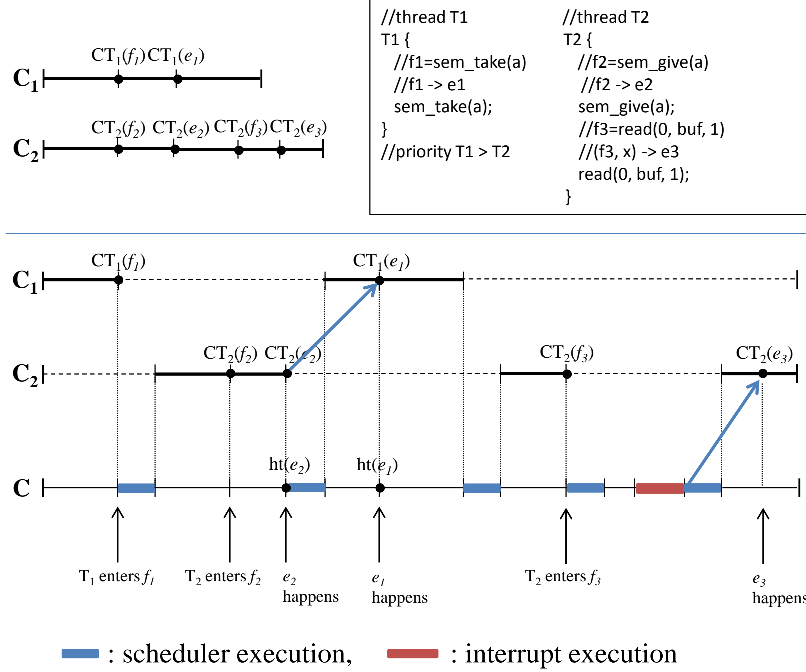


Figure 4.1: An Example of Global/Local Clocks and Event Executions for Two Interacting Threads.

happens, respectively.

- $\boldsymbol{CT_i}(f)$ and $\boldsymbol{CT_i}(e)$ return the $\boldsymbol{C_i}$ clock values when T_i enters f_i and when e_i happens as a result of f_i's execution, respectively. In other words, function $\boldsymbol{CT_i}$ returns accumulated CPU time spent by thread T_i until the instants of entrance of function f or execution of event e.

Based on $\boldsymbol{CT_i}(f)$ and $\boldsymbol{CT_i}(e)$, we can compute $\boldsymbol{ct}(f_{i,k})$ which is thread T_i's execution time between the instants that the $(k-1)^{th}$ event $e_{i,k-1}$ happens and that the subsequent function invocation $f_{i,k}$ is entered. Similarly, $\boldsymbol{ct}(e_{i,k})$ can be obtained as the processing time from entering $f_{i,k}$ to posting $e_{i,k}$. In Figure 4.1, an example execution of the interacting events performed by threads T_1 and T_2 and event timestamps are depicted. In addition to the thread local clocks for T_1 and T_2, thread events are aligned with the global clock and the kernel scheduling and interrupt service events are included.

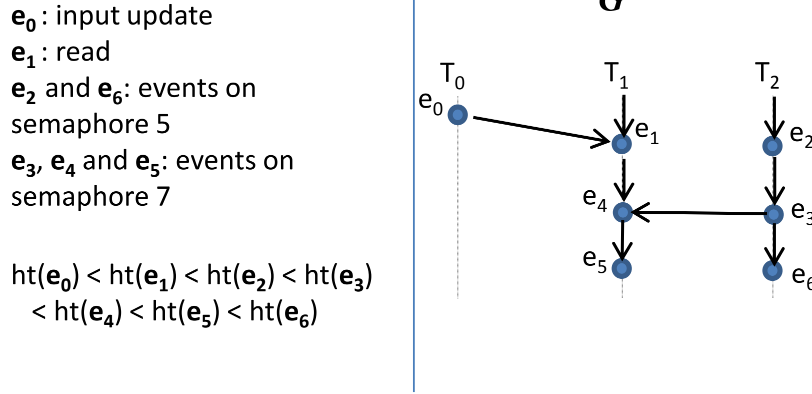


Figure 4.2: An Example of Partial Order Graph G with Seven Events from Program Execution.

For the events generated from the executions of the program threads ($T_1 ... T_n$) and the system thread (T_0), an event graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ can be constructed where $\boldsymbol{V}$ is a set of events and edge $(e_a, e_b) \in \boldsymbol{E}$ if $e_a \in \boldsymbol{V}$ and $e_b \in \boldsymbol{V}$ and e_b is logically dependent on e_a. Basically, $\boldsymbol{G}$ is a partially ordered graph representing the happened-before relation among events [39]. An example of event graphs is shown in Figure 4.2. In the following sections, we use $\boldsymbol{G^I} = (\boldsymbol{V^I}, \boldsymbol{E^I})$ and $\boldsymbol{G^U} = (\boldsymbol{V^U}, \boldsymbol{E^U})$ to represent the event graphs of the instrumented program $P^{\boldsymbol{I}}$ and the original program without the instrumentation P^U, respectively. To determine whether there is a probe effect caused by instrumentation, it is assume that the initial states and the external events, including interrupts and the arrivals of input data, are identical in the execution of P^I and P^U. We will then need to compare $\boldsymbol{G^I}$ and $\boldsymbol{G^U}$ or at least find a way to check whether $\boldsymbol{G^I}$ differs from $\boldsymbol{G^U}$.

4.4 Simulated Program Execution

We consider executions of instrumented and un-instrumented programs with the same input. In the instrumented program, extra code is inserted to record program execution behavior that we are interested in. For instance, additional instrumentation

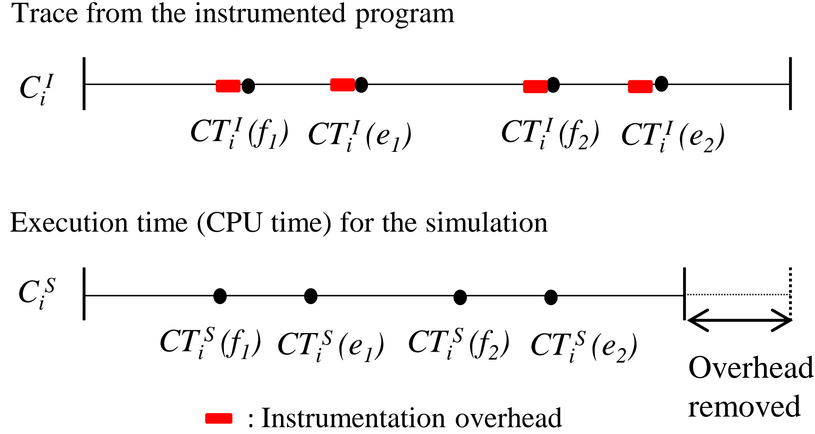


Figure 4.3: (top): execution time from the instrumented program execution. (bottom): From the trace for the instrumented program execution (top figure), instrumentation overhead is removed for the simulation.

is added to obtain the trace of thread interaction events, thread execution time, and the overhead of instrumentation code. Thus, the event graphs $\boldsymbol{G^I}$ can be constructed from the observed event trace. However, $\boldsymbol{G^U}$ of P^U is not observable directly.

With the event functions and the events collected from the execution of P^I, an event-driven simulation can be conducted to analyze the execution behavior of P^U. Since there is no instrumentation in P^U, any instrumentation overhead should be removed from the thread execution time of P^I. As shown in Figure 4.3, the execution time to be considered in the simulation analysis, $\boldsymbol{CT_i}^S(f)$ and $\boldsymbol{CT_i}^S(e)$, can be obtained from the measured $\boldsymbol{CT_i}^I(f)$ and $\boldsymbol{CT_i}^I(e)$ of the instrumented program P^I. As a consequence, if P^U invokes an identical event function and results in the same event as in the execution of P^I, the event may occur at an early instant. The change in execution time can also vary the relative order of program and system events. For instance, an event e of P^I that is invoked after an interrupt may happen before the same interrupt in P^U. This alteration may have a ripple effect. For instance, if the interrupt service routine signals a ready status, the running thread that invokes a function call to read the status in P^U would not see the ready status immediately and may be blocked until the interrupt arrives. The example suggests that, in the

simulation analysis, kernel resource and scheduling operations must be incorporated to determine the event ordering.

The goal of the simulation is to generate an event graph $\boldsymbol{G^S}$ to represent the un-instrumented execution of all threads and system events observed in the execution of P^I. In the simulation, thread T_i will execute a sequence of function invocations and events ($f_{i,k}$ and $e_{i,k}$) in timestamp ordering that are collected from P^I. The inter-event execution times of $\boldsymbol{ct}^S(f_{i,k})$ and $\boldsymbol{ct}^S(e_{i,k})$, are calculated by subtracting any instrumentation overhead from $\boldsymbol{ct}^I(f_{i,k})$ and $\boldsymbol{ct}^I(e_{i,k})$, respectively. The simulation is done with a global clock $\boldsymbol{C^S}$, and assume that the current $\boldsymbol{C^S}$ value is denoted as *cur_time*. At each simulation epoch, the ready thread with the highest priority is chosen as the running thread. The running thread T_i can proceed to perform its next activity $a_{i,k}$, where $a_{i,k}$ is either $f_{i,k}$ or $e_{i,k}$, if there is no system event in [*cur_time*, *cur_time*+$\boldsymbol{ct}^S(a_{i,k})$]. The global clock $\boldsymbol{C^S}$ is then advanced to the next simulation epoch, i.e., $\boldsymbol{C^S} = cur_time + \boldsymbol{ct}^S(a_{i,k})$. Otherwise, the running thread is preempted by the arriving interrupt irq_j at $cur_time = \boldsymbol{ht}(irq_j)$. Note that we have recorded the interrupt happen time, $\boldsymbol{ht}(irq_j)$, from the instrumented program execution and the interrupt is being supplied at the same time as in the instrumented program execution. After the un-instrumented execution time of irq_j, T_i may continue its execution or a context switch may occur if there is a thread of higher priority waked up by the interrupt.

As the simulation proceeds from one event to the other, the graph $\boldsymbol{G^S}$ is constructed by adding edges for logical dependencies among threads and system events. For instance, a happened-before relation is added from a message send event to a subsequent message receive event. Similarly, an interrupt is happened before the thread's sem_wait event which gets completed due to a sem_post event issued by the interrupt. In Section 4.6, the simulation algorithm implemented for execution in

VxWorks is described.

4.5 Analysis of Simulation Results

Once $\boldsymbol{G^S}$ is constructed, we will be interested in the execution behavior of P^U that may be inferred based on $\boldsymbol{G^S}$. Note that $\boldsymbol{G^S}$ does not always represent $\boldsymbol{G^U}$, since if the event ordering of P^U is different from that of P^I (due to the instrumentation overhead), then the execution paths of P^U and P^I might be different. Thus, there might be an event e such that $e \in \boldsymbol{V^I}$ but $e \notin \boldsymbol{V^U}$. Given that the simulation is based on the events in P^I, we have $e \in \boldsymbol{V^S}$. This leads to $\boldsymbol{G^U} \neq \boldsymbol{G^S}$. However, a positive result can be established in the following theorem when $\boldsymbol{G^I}$ and $\boldsymbol{G^S}$ are equivalent.

Theorem 1: $\boldsymbol{G^U} = \boldsymbol{G^I}$ if and only if $\boldsymbol{G^S} = \boldsymbol{G^I}$.

Proof: to show the "if" part, let's assume $\boldsymbol{G^U} \neq \boldsymbol{G^I}$. Then, there should be at lease one thread that generates different events or experiences different happened-before relations in the executions of P^U and P^I. Let $e_i^I(k)$ be the first such event of P^I (in terms of the global clock $\boldsymbol{C^I}$) that $e_i^I(k) \neq e_i^U(k)$, where $e_i^X(k)$ represents the kth event performed by thread T_i of program P^X. All events of P^I that are prior to $e_i^I(k)$ have the identical happened-before relations as their equivalent events in P^U. For thread T_i, it must have executed the same first $(k-1)$ functions and generated the same $(k-1)$ events, i.e., $e_i^I(l) = e_i^U(l)$ for $l = 1, \ldots, k-1$. Thus, it should use the same function in its kth invocation, i.e. $f_i^I(k) = f_i^U(k)$.

To have $e_i^I(k) \neq e_i^U(k)$, the global states of P^I and P^U that are used in the processing of $f_i^I(k)$ and $f_i^U(k)$, should be different. This suggests that at least one extra update is inserted before the invocations of $f_i^U(k)$ in the execution of P^U. Let this update event be performed by a different thread T_j and denoted as $e_j^U(k'), k \neq k'$. As the instrumentation overhead is removed in P^U, this update event is brought

forward and occurs before the invocations of $f_i^U(k)$ in the execution of P^U, even if $e_j^I(k')$ occurs after the invocations of $f_i^I(k)$ in the execution of P^I. This change in event ordering should be observed in the simulation analysis, i.e., $e_j^S(k')$ occurs before the invocations of $f_i^S(k)$ since the same instrumentation overhead is deducted and the program behavior has not been changed before T_i makes its kth invocation. The addition of the happened-before dependency, $e_j^S(k') \rightarrow e_i^S(k)$, in $\boldsymbol{G^S}$ results in $\boldsymbol{G^S} \neq \boldsymbol{G^I}$.

For the "only if" part, the equivalence of $\boldsymbol{G^U}$ and $\boldsymbol{G^I}$ indicates that thread T_i would invoke the same event function $f_i^I(k)$ and generate the same event $e_i^I(k)$ as in the execution of P^I, even if the instrumentation overhead is removed. Thus, the simulated execution of $f_i^I(k)$ will generate the same event $e_i^I(k)$ which has the same happened-before relations with preceding events. This implies $\boldsymbol{G^S} = \boldsymbol{G^I}$. ■

The theorem implies that, if the logical order of thread events built in the simulation analysis is as same as the one in the execution of the instrumented program P^I, then $\boldsymbol{G^I}$ is the true representation of $\boldsymbol{G^U}$. On the other hand, if $\boldsymbol{G^S} \neq \boldsymbol{G^I}$, the simulation failed in a sense that when the partial order begins to be different, the execution path may also have changed too. This suggests that the instrumented program may have started to take a different execution path. Since the simulation uses the same execution path as the instrumented program, the simulated execution is no longer a representation of the un-instrumented program. However, we can find out when and how the execution changes. Let e_d be the very first event of $\boldsymbol{G^S}$ that causes a different partial order from $\boldsymbol{G^I}$. Then, the partial graph $\boldsymbol{G^S}$ for all events priori to e_d is the true representation of the same events in the execution of P^U. A follow-up investigation on the trace can be considered to find out how the instrumentation changes the program execution.

4.6 Implementation

4.6.1 Execution Environment

To implement the proposed approach of detecting probe effect, an execution environment is set up on a single core of a 1.6 GHz Intel Atom processor running VxWorks 6.8 [102]. The VxWorks' priority-based preemptive scheduler is configured. Two IRQs are available during the execution, a 60Hz timer IRQ and a PS2 keyboard IRQ. The queuing mechanism for tasks blocked on a semaphore is based on task priority.

We consider a simplified system in which three kinds of tasks and system activities can affect the timings of event occurrences and must be traced in the execution of instrumented program: application tasks, interrupt handlers, and the scheduler. All application tasks and kernel operations are run in a flat memory space and there is no page-fault exception. We also assume there are no exceptions from the running applications. The priorities of application tasks are set to be higher than the priorities of any other system background tasks. Thus, the execution of background tasks will not affect the analysis of probe effect. The trace data are sent to the host at the end of the application execution to avoid any activities including file IO during application execution.

4.6.2 Execution Trace and Measurements

To trace the invocation of event functions and the occurrence of events, we adopt the instrumentation mechanism of the record of our deterministic replay framework introduced in Chapter 3. Since the record framework already supports the wrappers for tracing event execution, we only added 1) the measurement code for CPU time spent for each thread and 2) the overhead measurement for the instrumentation code. The existing interrupt handlers for timer and keyboard are instrumented to collect

the timestamp when an interrupt arrives and to measure the execution time of interrupt handler. In addition, the keyboard interrupt handler is customized to directly communicate with the keyboard driver. For our Atom-based target processor, x86's RDTSC (Read Time-Stamp Counter) instruction is used to collect timestamps.

Task execution time is measured in scheduler hook routines that are invoked for every context switch. For each task, we keep the timestamp that the task is switched in and when the task is switched out. The difference between the current timestamp and the switched-in timestamp is accumulated to the task CPU time. The execution time of scheduling operation and context switching is measured offline using two tasks, task 1 and task 2, where task 2 has a higher priority than task 1. First, we let task 2 be blocked on a semaphore and when task 1 posts the semaphore, task 2 becomes running. Then, we remove task 2 and just run the sem_post operation by task 1. The intervals from the invocation of sem_post to the completion of the call are measured for the two cases. The difference is considered as the measured execution time of scheduling operation and context switch.

4.6.3 Simulation Analysis Algorithm

Using the execution trace from the instrumented program execution, the simulation is performed as shown in Figures 4.4 and 4.5. It is governed by the invocation to event functions and the occurrence of events and interrupts. The simulation maintains a global clock $\boldsymbol{C}$ which is advanced when the running task, event function, or interrupt service routing is executed. Note that thread T_0 is used to represent system's external activities (e.g., interrupt and input data change events) and runs concurrently with the scheduled application threads.

When a thread is scheduled to run, the global clock $\boldsymbol{C}$ is adjusted to the instant of the next interrupt or the execution time of the subsequent event function call is added

```
1:    enter_func(func: f, thread: T, event e)
2:        if f is resource release function
3:            return
4:        if resource not available
5:            if f is synchronous function
6:                set T to pending
7:            else
8:                mark e as "fail to acquire the resource"
9:        else
10:           mark e as "succeed to acquire the resource"
11:
12:   execute_event(event: e, thread: T)
13:       if e is resource releasing event
14:           the resource is released
15:           if any tasks pending for the resource
16:               select a task and set it to ready
17:       e_L = event that e logically depends on
18:       V^S = V^S ∪ e
19:       E^S = E^S ∪ (e_L, e)
```

Figure 4.4: Sub-routines for the Simulation Algorithm. The selection of a task in execute_event() is based on the system property of the instrumented program, e.g., FIFO, priority-based.

to $\boldsymbol{C}$, whichever comes first. When the global clock $\boldsymbol{C}$ is equal to the arrival time of an interrupt, the time spent on the interrupt is added to $\boldsymbol{C}$. If there is a thread pending for the arrival of the interrupt, its state is changed to *ready* once the interrupt is processed. Then, the highest priority ready thread is scheduled for execution. If an event function invocation takes place, the resource required by the function is evaluated. The call can lead a return with error, a blocked thread, or the execution of event function. A simulated event happens when an event function is completed. The system state may be updated (e.g., a message is dequeued) and blocked tasks may be waked up as the consequence of the happened event. Whenever needed, the scheduler's execution time is added to $\boldsymbol{C}$ to simulate the scheduling operation. When there is no ready thread, the idle process is simulated by advancing $\boldsymbol{C}$ to the next interrupt. Note that, interrupts are accepted during task execution and are delay if

```
1:   do_simulation()
2:     C_cur = 0           // the current global clock
3:     Ci = 0  for all Ti    // Ci is the thread clock for Ti
4:     Tr = the highest priority ready thread
5:     next_act = the earliest action of T0 and Tr
6:     while (true) {
7:         Δ=the instant of cur_act – C_cur
8:         cur_act = next_act
9:         C_cur is advanced to the instant of cur_act
10:        if (cur_act == ISR completion)
11:            set any tasks pending for the ISR to ready
12:        else {
13:            if (Tr ≠ null), Cr is advanced by Δ
14:            if (cur_act == IRQ arrival at T0)
15:                next_act=ISR completion
16:            else if (cur_act==input data change)
17:                mark data input event
18:            else if (cur_act==function f invocation)
19:                enter_func(f, Tr, e) // Ex:f→e
20:            else if (cur_act==event e completion)
21:                execute_event(e, Tr)
22:        }
23:        if any change in task state
24:            Tr = the highest priority ready thread
25:        if (cur_act ≠ IRQ arrival at T0)
26:            next_act = the earliest action of T0 and Tr
27:        if all events are executed
28:            break
29:    }
```

Figure 4.5: Simulation Algorithm

they arrive during the execution of event functions or scheduling operation.

4.7 Evaluation

We used two multi-threaded programs, the dining philosophers and the sleeping barber, from the LTP benchmark suite [90] in the experiments. The dining philosophers program has 5 philosopher threads (P1 to P5) with decreasing priorities from philosopher 1 to philosopher 5. Each philosopher is looping from thinking, picking up

Thread	CPU time (cycles)	CPU time without overhead(cycles)	Overhead
P1	29,964,934	20,042,918	49.5%
P2	29,073,286	20,029,304	45.1%
P3	161,459,044	153,060,446	5.4%
P4	161,626,702	153,232,244	5.4%
P5	160,931,184	153,228,482	5.0%
Average			8.6%

Table 4.1: CPU Time for the Dining Philosophers

forks, and to eating. The thinking and eating activities of philosophers 1 and 2 are implemented with blocking reads for keyboard input. On the other hand, the thinking and eating activities for philosophers 3, 4, and 5 are replaced with a simulated computation. When a keyboard pressing interrupt occurs, a philosopher (1 or 2) will be waked up and may preempt another running philosopher. Thus, philosophers will be differently interleaved depending on the timing of keyboard inputs (e.g., different order of philosophers' eating). In the sleeping barber program, the waiting room has three available chairs and there are one barber thread with the highest priority and 5 customer threads (C1 to C5) with decreasing priorities from customers 1 to 5. The barber is looping from sleeping if there is no waiting customer, and to serving a customer. A customer waits in the waiting room and gets a haircut if a chair is available. He leaves without a haircut if all the three chairs are occupied. The barber's sleep is waked up by a keyboard interrupt. As a consequence, customers 4 and 5 (with lower priorities than customers 1 to 3) may not get haircuts depending on how fast the barber is waked up by keyboard interrupt.

Both programs in the experiments have relatively short execution times considering manually injected keyboard operations. A simulated computation is inserted between event functions to ensure the speed of program execution is comparative to the rate of manual keyboard entry. As we add wrappers to event functions and interrupts, instrumentation overhead is added to the program execution. In addition,

Thread	CPU time (cycles)	CPU time without overhead(cycles)	Overhead
Barber	7,965,912	5,024,050	58.5%
C1	10,065,480	9,544,202	5.4%
C2	10,046,790	9,524,506	5.4%
C3	2,568,782	2,047,280	25.4%
C4	1,378,142	1,029,454	33.8%
C5	1,615,038	1,231,966	31.0%
Average			18.4%

Table 4.2: CPU Time for the Sleeping Barber

extra simulated computation is inserted in thread execution to represent dynamic analysis or profiling overheads. These instrumentation overheads are removed in the following simulation analysis. Tables 4.1 and 4.2 show the execution times and overheads from the average of 5 executions of each benchmark program. In Table 4.1, the CPU times of philosophers 3, 4, and 5 are greater than those of philosophers 1 and 2 as simulated computations are used for the thinking and eating activities instead of blocking reads. In Table 4.2, customers 1 and 2 spend more CPU times than other customers as we inserted additional simulated computations before entering the waiting room to delay the entrances of customers 4 and 5. Thus, customers 4 and 5 would not arrive too early while the first three customers are waiting on the three chairs, and have to leave. There are some differences in the CPU times spent by customers 3, 4, and 5 as customer 3 always gets a haircut while customers 4 and 5 may get haircuts depending on input timing.

Figures 4.6, 4.7, and 4.8 show experimental results illustrating probe effect. Figures 4.6 and 4.7 are the results of two different executions of the dining philosophers program with different inputs and Figure 4.8 is based on an execution of the sleeping barber program. In each result, we compare the event orderings from the instrumented and simulated program executions. We only illustrate specific time frames with particular threads that show some variations in event ordering. The horizontal

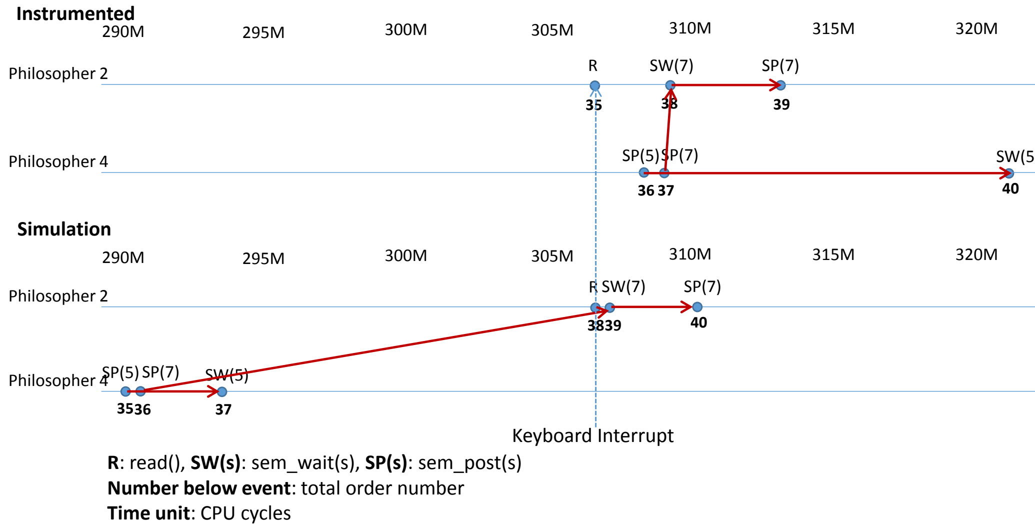


Figure 4.6: An example execution of dinning philosophers program where $G^I = G^S$ but some events are with different timestamp ordering

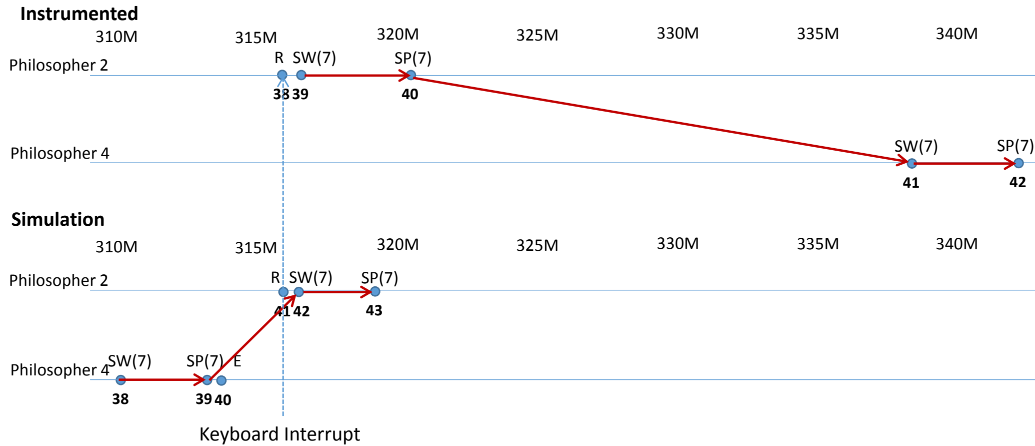


Figure 4.7: An example execution of dinning philosophers program in which $G^I \neq G^S$ (i.e., the partial orderings are different after the 38th event)

lines show the timestamps of event occurrences and arrow lines indicate the logical dependencies between events. The number below each event denotes the event sequence number in timestamp (total) ordering. Figure 4.6 depicts a case when $G^I = G^S$ but with different timestamp orderings. The case appears when philosopher 2 is waiting for a keyboard input and, as soon as the input becomes available, it preempts philosopher 4. In the simulated execution, since thread execution time is reduced due to the removal of the instrumentation, the events of philosopher 4 happen while philosopher

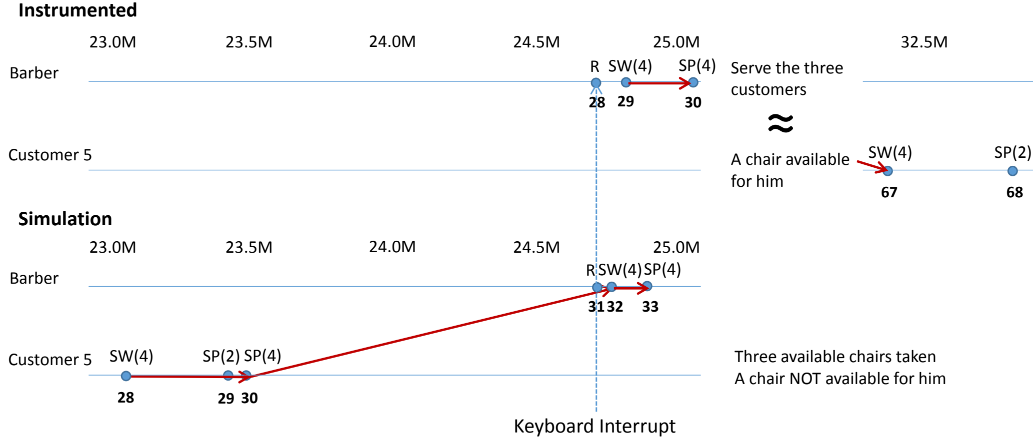
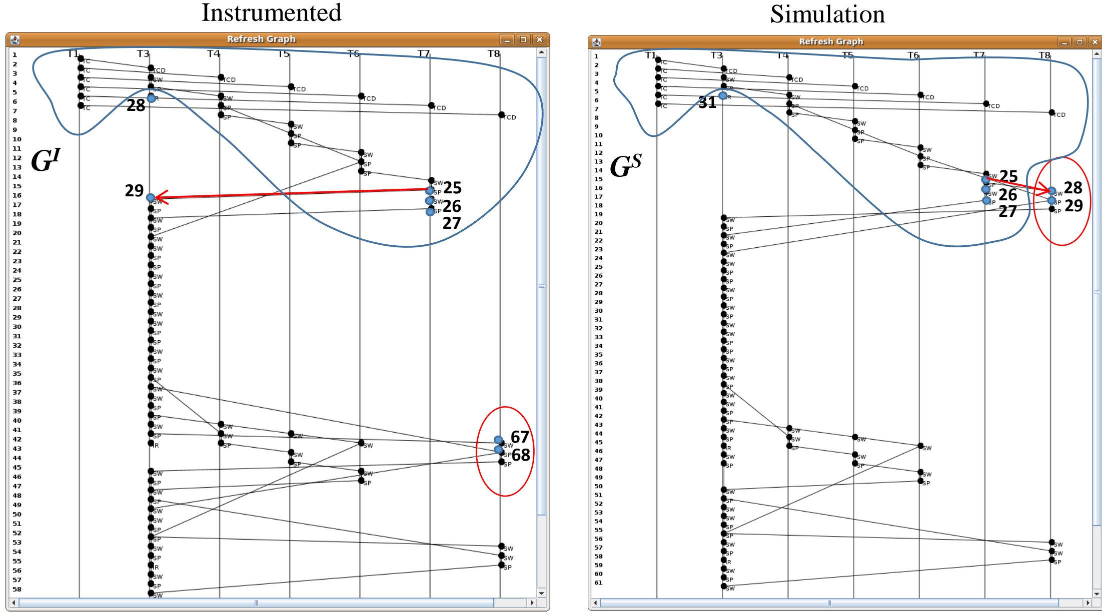


Figure 4.8: An example execution of sleeping barber program where $\boldsymbol{G^I} \neq \boldsymbol{G^S}$ after the 28$^{\text{th}}$ event. In the simulation, customer 5 does not get haircut while he does in the instrumented program

2 is still blocked. As a result, the timestamp ordering of events in the simulation differs from that of the instrumented program. However, the partial ordering of events still remains same, i.e., $\boldsymbol{G^I} = \boldsymbol{G^S}$. Hence, the simulation is the true representation of the original program execution, and since $\boldsymbol{G^U} = \boldsymbol{G^S} = \boldsymbol{G^I}$, the instrumentation overhead has not caused any probe effect on the execution.

Figure 4.7 shows a case where the logical (partial) ordering is altered due to the instrumentation overhead. The timestamp ordering of events in the simulation is changed in a similar way as in Figure 4.6. Events of philosopher 4 in the simulated program execution occurred earlier than in the instrumented program execution while philosopher 2 was still blocking for input. Since $\boldsymbol{G^I} \neq \boldsymbol{G^S}$, the simulation is no longer a representation of the original program. We notice that the partial order graphs, $\boldsymbol{G^I}$ and $\boldsymbol{G^S}$, are identical until the 37$^{\text{th}}$ event. Thus, for each thread, the next event function to be invoked immediately after the 37$^{\text{th}}$ events should be identical in the instrumented and simulated program executions. After the 37$^{\text{th}}$ event, the difference in the logical ordering of events is triggered by the order of the sem_wait(7) events invoked by philosopher 2 and philosopher 4. We manually inspected the source code



(a) Instrumented Program Execution. (b) Simulated Program Execution.

Figure 4.9: Examples of the partial order graphs from the executions of sleeping barber program.

and found out that the change in the logical ordering did not result in a change of execution path. Hence, any dynamic program analysis based on the execution path is still valid. However, in general it will be a very challenging task to find a change of execution path by manual inspection of source code.

Figure 4.8 demonstrates a case for a change of execution path due to instrumentation overhead in sleeping barber program. The result shows that $\boldsymbol{G^I} = \boldsymbol{G^S}$ up to the 27th event. However, in the 28th event, the sem_wait(4) is invoked much earlier in the simulation than in the instrumented program. The resultant partial order graphs are drawn in Figure 4.9. In each partial order graph, the second line shows events for barber thread and the last line shows events for customer 5 thread. In the simulation, the semaphore given at the 25th event is taken by customer 5 while in the instrumented program execution the semaphore is taken by barber thread. In fact, in the instrumented execution, customer 5 arrives after the barber is waked up by a

keyboard input and begins to cut hair. Hence, a chair becomes available for the arriving customer 5. On the other hand, in the simulated execution customer 5 arrives before the keyboard input, suggesting the barber is still in sleep mode. The customer should leave as he cannot find any available chair in the waiting room leading to a different execution path. Since the simulation uses the same observed event from the instrumented program, the simulated execution may not be correct after the 28th event.

4.8 Related Work

Malony et al. presented the instrumentation uncertainty principle [49] suggesting that the accuracy of execution performance is degraded as the degree of instrumentation increases. In the approach, performance perturbation models were proposed to calculate the true performance from instrumented parallel programs. The models were further refined in [103]. In the models, perturbations trigger a change in event execution time and event ordering is represented by time-based and event-based perturbation models. In the time-based model, the thread events are independent while the event-based model considers the dependency between events for recovering the true performance. The dependency considered is performance degradation as arrival time and resource state change. However, the approach assumes the program execution is fixed no matter there is any instrumentation overhead or not. Hence, it does not consider how the program behavior may differ from the un-instrumented program.

When instrumentation perturbation causes different thread interleaving, we will be concerned with the potential problems of data race and execution non-determinism. Data races can result in arbitrary failures and do not help increase scalability [1]. Several efficient dynamic data race detection algorithms [19, 70, 76, 79, 84, 86, 105]

have been proposed and race detection tools [27, 94, 95] are widely used in practice. In general, the approaches are based on the monitoring of read/write operations to shared variables among concurrent threads. However, the delay caused by monitoring operations and any possible probe effect have not been addressed. Deterministic multi-threading techniques provide deterministic event ordering for parallel program execution [6, 11, 12, 45, 46, 51, 65]. In Kendo [65], a thread's progress is represented with a logical clock. It is a thread's turn to take a lock when its logical clock is the global minimum. In [45] and [46], thread's shared memory is isolated from other threads during a parallel phase. During a serial phase, the memory updates to shared variables are applied and locks are taken in a deterministic order. Regardless of their overheads, the approaches don't consider any external input events and time-based operations, and cannot be applicable to embedded software.

It may be argued that instrumentation can be done during program replay if a reproducible execution can be constructed. Instant Replay [41] is one of the earliest works that allows cyclic debugging for parallel programs by tracing and replaying relative order of events for each shared object in the program. RecPlay [76, 77], based on Lamport's happened-before relations [39] records and replays synchronization operations. In the approach, data races can be detected by checking all shared memory references so that the program is free of data race before record/replay operations. To reduce overhead of record and replay, speculative execution and external deterministic replay are used in Respec [42] that is capable of online replaying on multiprocessor systems even with data races. Using speculative execution, the recording process can continue to execute speculatively instead of being blocked until the corresponding replay finishes.

Most profiling tools adopt instrumentation approaches. There have been research efforts to reduce profiling overhead caused by instrumentation. Froyd et al. proposed

a call-path profiler based on stack sampling [21]. The profiler, called *csprof*, provides an efficient way of constructing the calling context tree without instrumenting every procedure's call. Zhuang et al. introduced the adaptive busting approach [108] to build calling context tree with a reduced overhead while preserving profiling accuracy. In their approach, unnecessary profiling is avoided by disabling redundant stack-walking with a history-based predictor. The profiling overhead has been further alleviated by taking advantage of multi-core systems. In shadow profiling [54], shadow processes are periodically created for running instrumented code while the original process is running on a different core with minimal overhead. PiPA [106] exploits parallelism by forming a pipeline to collect and process profiles. Application execution and profiling operation are divided into stages that are pipelined and performed in multiple cores. Kim et al. proposed a scalable data-dependence profiling [37] to reduce runtime and memory overheads. In the approach, memory references are stored as compressed formats, and pipelining and data level parallelism are used to reduce the overheads in the data-dependence profiling. Time-aware instrumentation approaches [3, 16] have been proposed to minimize violation of timing constraints due to instrumentation overhead. DIME [3] monitors instrumentation time and limits the program instrumentation to a given time budget in a time period. A static approach [16] inserts instrumentation code only where the instrumentation can preserve the worst-case execution time.

Although there have been research works on the overhead calculation for precise performance measurement and the reduction of instrumentation overhead for reproducible execution and profiling, no work has been proposed to reveal the possibility of execution deviation caused by instrumentation overhead. In this chapter, the focused analysis is to verify if the recorded or observed execution is a true representation of the original program execution and if any instrumentation may alter the event or-

dering of multi-thread embedded programs as to cause any changes of the intended program behavior.

4.9 Chapter Conclusions

Often the most important metric in dynamic analysis of multi-threaded programs is the overhead of instrumentation since researchers are aware of the potential probe effect caused by the overhead. However, to the best of our knowledge, no research has proposed a way to detect any changes in program execution when the programs are instrumented. In this chapter, we model the execution of multi-threaded program according to the happened-before ordering of global events. Using the trace of event function invocations and O/S activities, a simulation-based analysis is presented to detect if the partial order of events is altered by instrumentation. The experiments of two simple applications running on VxWorks demonstrate how instrumentation overhead can lead to changes in the timestamp ordering and in the partial ordering of the event executions.

In the thread execution model of this chapter and Chapter 3, a program execution is represented with a partial order of synchronization and I/O event executions. The execution model makes the analysis of program efficient. However, presence of data races in a program execution can make the program analysis incorrect. That is, recording only synchronization events would not enough to capture the execution of memory accesses contained in the data races. When an execution of program is deviated from another, it will be hard to decide if the deviation was caused by instrumentation overhead or data races. Hence, in Chapters 5 and 6 we propose efficient data race detection algorithms that help to remove data races.

Chapter 5

DATA RACE DETECTION FOR C/C++ PROGRAMS

In previous chapters, thread executions are modeled as a partial order of synchronization and I/O event executions. The approach makes the dynamic program analyses (i.e., record/replay, probe effect analysis) feasible and efficient. However, the program execution model may not work correctly in the presence of data races. In this chapter, we discuss an efficient data race detection for C/C++ programs with minimal false alarms. To detect races precisely without false alarms, vector clock based race detectors can be applied if the overheads in time and space can be contained. This is indeed the case for the applications developed in object-oriented programming language where objects can be used as detection units. On the other hand, embedded applications, often written in C/C++, necessitate the use of fine-grained detection approaches that lead to significant execution overhead. In this chapter, we present a dynamic granularity algorithm [84] for vector clock based data race detectors. The algorithm exploits the fact that neighboring memory locations tend to be accessed together and can share the same vector clock archiving dynamic granularity of detection. Our experimental results on benchmarks and comparisons with two commercial race detection tools show that the proposed dynamic granularity approach is very viable.

5.1 Introduction

Most embedded applications are constructed with multiple threads to handle concurrent events. With the prevalence of multi-core architectures, applications can be programmed with multiple threads that run in parallel to take advantage of on-

chip multiple CPU cores and to improve program performance. In a multi-threaded program, concurrent accesses to shared resource and data structures need to be synchronized to guarantee the correctness of the program. Unfortunately, the use of synchronization primitives and mutex locking operations in multi-threaded programs can be problematic and results in subtle concurrency errors. Data race condition, one of the most pernicious concurrency bugs, has caused many incidences, including the Therac-25 medical radiation device [44], the 2003 Northeast Blackout [92], and the Nasdaq's Facebook glitch [33].

A data race occurs when a memory location is accessed concurrently by two different threads and at least one of the accesses is a write. Data races are hard to reproduce, find, and fix since a data race may only occur in a particular execution of the program and the race does not necessarily always cause observable errors in the program execution.

Over the past few years, several techniques have been developed to detect data races. Static analysis techniques [17, 34, 58, 71, 98] consider all execution paths for possible data races providing a better detection coverage than dynamic techniques but they suffer from excessive number of false alarms. On the other hand, dynamic techniques detect data races when execution paths are exercised and they largely fall into two categories: LockSet algorithms [79, 105] and happens-before algorithms [19, 70, 76, 77, 94]. In Eraser's LockSet algorithm [79], data races are reported when shared variable accesses violate a specified locking discipline, i.e., the variable is not protected by the same lock consistently. For a given execution path, checking a lock discipline enables Eraser to detect potential data races as well as ones that actually happened in the program execution. Eraser may report many false alarms which hinder developers' focus on fixing real problems. Eraser may also report false alarms since lock operations are not the only way to synchronize threads and a violation of

lock discipline does not necessarily imply a data race.

Happens-before detectors are based on Lamport's happens-before relation [39] and do not report false alarms as the approach only checks any happens-before relation that actually occurs during the given execution paths. Based on the execution of a multi-threaded program, a partial ordering of memory and synchronization operations can be defined. A pair of accesses to the same variable is concurrent when neither of the accesses happens before the other. The happens-before relation is realized by the use of vector clock [14] and there are largely two happens-before detection methods based on how shared accesses are represented and compared for the detection.

In the first method [76, 77, 94], a segment is defined as a code block between two successive synchronization operations and shared memory accesses are collected in a bitmap for each segment. In each thread a vector clock is collected to uncover any concurrent segments of running threads. If two concurrent segments contain common shared memory accesses, the accesses are reported as data races. This method may incur a significant overhead in time despite of several optimization techniques such as *clock snooping* and *merging segments* [76, 77] as the detection requires set operations with the collected shared memory accesses.

In the second method [19, 70], other than a vector clock for each thread, each shared variable has two vector clocks for reads and writes which record access history of the variable by every thread. When a thread reads from a shared variable, the write vector clock of the shared variable is compared with the vector clock of the thread. A write-read data race is reported if the vector clock comparison reveals that the write and the read are not ordered by the happens-before relation. A similar protocol can be performed for a write access. Having vector clocks for each shared variable can result in huge memory consumption. However, for applications developed in object-oriented languages, the memory requirement may be tolerable as large detection units

such as field or object can be used.

Data race detection for embedded applications which are mostly written in C/C++ needs to consider fine granularity of data access (e.g., byte). It would seem to be a better choice to use the first happens-before detection method since having a vector clock for every shared read or write byte may make the detection infeasible. On the other hand, as shown in FastTrack [19], the second method can reduce the space and time overheads of vector clocks from $O(n)$ (where n is the number of threads) to nearly $O(1)$ with no loss in detection precision. While it may become feasible to detect races in C/C++ programs, the overheads using the FastTrack detector with fine granularity is still high for C/C++ programs, as illustrated in our experimental results.

In this chapter, we present a dynamic granularity algorithm for vector clock based race detection. The detection granularity starts from byte and is dynamically adjusted as shared memory locations are accessed. A large detection granularity is adopted when neighboring bytes have the same vector clock. Thus, instead of multiple copies, a single copy of vector clock is shared among these neighboring bytes. Sharing the same vector clock among neighboring memory locations become feasible since (1) neighboring memory locations belonging to array or *struct* tend to be accessed together, (2) data structures are often accessed together during initialization even if they are separately protected afterward, and (3) some groups of shared memory locations are accessed only for one code block.

In the algorithm, a state machine is associated with a vector clock for read or write of a memory location and the state can be *Init*, *Shared*, *Private*, or *Race*. To minimize analysis overhead, the sharing decision for each read or write location is made at most twice for the lifetime of the location. Peak memory consumption is further reduced by temporarily sharing vector clock at *Init* state. In addition, the

possibility of false alarms caused by sharing vector clock is minimized as new sharing decision is made after *Init* state.

We have developed a race detector based on the FastTrack algorithm for C/C++ program and the dynamic granularity algorithm is implemented on top of the FastTrack implementation. Our experimental results on several benchmark programs show that the race detector using our dynamic granularity provides 43% speedup and 60% less memory over the FastTrack detector with byte granularity. Also we provide case studies on two popular data race detectors: Valgrind DRD [61, 94] and Intel Inspector XE [27]. Our dynamic granularity detector is about 2.2x and 1.4x faster than Valgrind DRD and Inspector XE, and consumes about 2.8x less memory than Inspector XE.

The rest of the chapter is organized as follows. In the following section, we discuss the notion of data races in more detail, and a brief survey of static and model-based approaches is described. In Section 5.3, a brief review of vector clock based race detectors is presented. Section 5.4 presents the proposed dynamic granularity algorithm. The implementation of data race detector using dynamic granularity is explained in Section 5.5. Section 5.6 shows evaluation results of our dynamic granularity as well as comparisons with Valgrind DRD and Intel Inspector XE. A concise survey of related work is described in Section 5.7 and, in Section 5.8 we conclude this chapter.

5.2 Discussion of Data Race Detection

5.2.1 What is a (Data) Race?

In this section, we clarify the concepts and terms used in the literature of data race detection. We often use the terms, *"data races"* and *"races"*, interchangeably. According to the definition in [63], a *race* occurs when two different threads access

```
//Thread 1, deposit "amount"
cur_balance = read_balance();
new_balance = cur_balance + amount;

//Thread 2, withdraw "amount"
cur_balance = read_balance();
new_balance = cur_balance - amount;
```

(a)

```
//Thread 1, deposit "amount"
Lock(a);
cur_balance = read_balance();
Unlock(a);
Lock(a);
new_balance = cur_balance + amount;
Unlock(a);

//Thread 2, withdraw "amount"
Lock(a);
cur_balance = read_balance();
Unlock(a);
Lock(a);
new_balance = cur_balance - amount;
Unlock(a);
```

(b)

```
//Thread 1, deposit "amount"
Lock(a);
cur_balance = read_balance();
new_balance = cur_balance + amount;
Unlock(a);

//Thread 2, withdraw "amount"
Lock(a);
cur_balance = read_balance();
new_balance = cur_balance - amount;
Unlock(a);
```

(c)

Figure 5.1: An Example of Atomicity Violation without Data Races. (a): Data race and atomicity violation. (b): No data race but with atomicity violation. (c): No data race and no atomicity violation.

shared memory concurrently. A race can cause a program to behave in unintended ways by the programmer. A *"data race"* is a race that can cause non-atomic execution of critical section. However, these definitions can confuse audiences since an atomicity violation can occur without data races (described in the next paragraph). Hence, in this thesis "a race" refers to "a data race" unless explicitly stated otherwise.

"An atomicity violation" is another type of concurrency bug closely related to data race. An atomicity violation occurs when two different threads access one or more of shared variables in a critical section concurrently. An example of atomicity violation is shown in Figure 5.1. Atomicity violation and data race are closely related to each other as both violations occur when threads access shared variables concurrently. However, free of data races does not necessarily imply no atomicity violation. In the example of Figure 5.1b, all the shared variables are protected by the same lock without data races. Yet, accessing the global variables for the back account is not atomic. Thus, the atomicity violation might cause incorrect updates of the variables. For the detailed discussion of atomicity violation, refer to [18, 26, 32, 56, 67].

The problem of locating *all real* races in a program is computationally hard. Races can be classified into several categories [63, 70]. *Feasible races* are races that

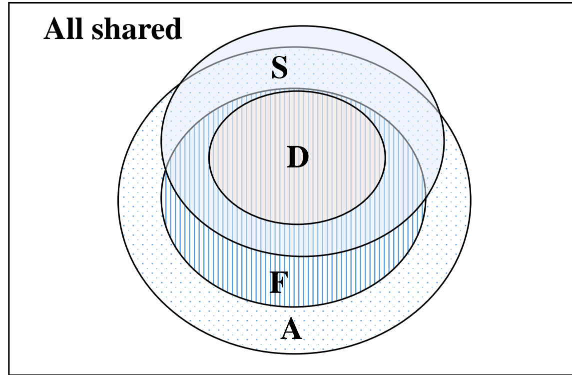


Figure 5.2: Classification of Data Races. The set of all shared variables in a program is represented by region **All shared**. **A**: the set of apparent data races. **F**: the set of feasible data races. **D**: the set of data races that can be detected by a dynamic happens-before approach. **S**: the set of data races that can be detected by a static approach or a LockSet detector.

can appear in a feasible execution of program. Hence, the ideal goal of a race detector is to locate all feasible races in a program. Unfortunately, the problem of locating all feasible races in a program is NP-hard [62]. *Apparent races* are approximations to feasible races based on explicit synchronizations. That is, a race can be located when a shared memory access is not protected by explicit lock operations. However, exhaustively locating all apparent races is also NP-hard [62].

In Figure 5.2, we show the classification of races. Vector clock based detectors locate a subset of feasible races since the detection is based on happens-before relations that actually occur in a program execution. In Lockset based or static approaches, the detection is based on approximations of lock-unlock pairs locating a subset of apparent races. Figure 5.3 show examples of feasible/apparent race. In an execution of the program in Figure 5.3a, there is a feasible race on variable X. Since the variable is not inside lock-unlock pairs, the race will be detected by a Lockset/static detector. However, a vector clock detector will not report a race on variable X since the accesses are ordered by the happens-before relation in the particular execution of the program. Figure 5.3b shows an example of an apparent race but not feasible.

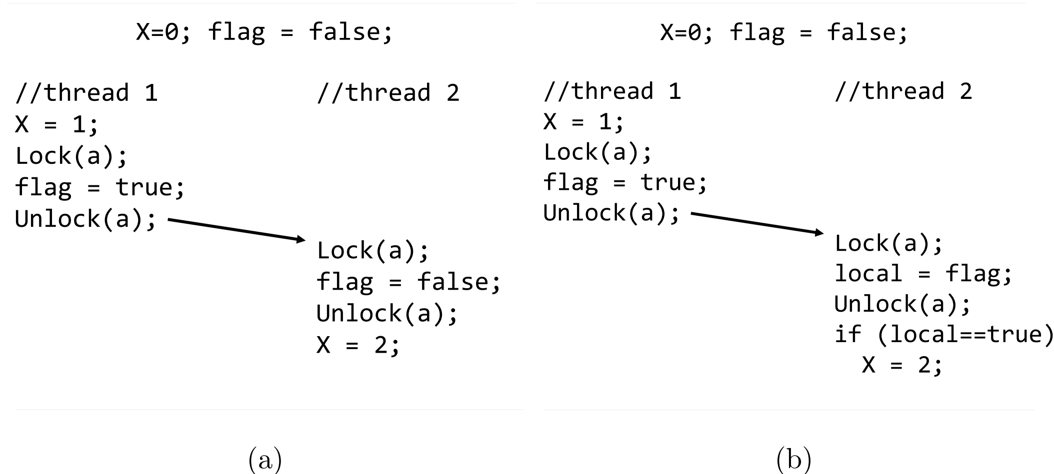


Figure 5.3: Examples of Feasible/Apparent Races. (a): A feasible race on variable X. A vector clock detector does not report a data race, but a Lockset/static detector will report a data race. (b): An apparent race but not feasible. A Lockset/static detector will report a false alarm, but a vector clock detector will not detect it as a data race.

```
static inline int stealTasksSpecialized( TaskQ *myT, TaskQ *srcT) {
    TaskQList *srcShared =  ( TaskQList *)&srcT->q.shared;

    volatile int *srcCount = &srcShared->count;
    if ( *srcCount == 0)    {return 0;} // Quick Check. Unprotected.
```

Figure 5.4: A Benign Race Example.

The accesses on variable X will be reported as a race by a Lockset/static detector as any of the accesses is not inside a lock-unlock pair. A vector clock detector will not report it as a data race as any execution that makes the accesses on X concurrent is not feasible.

Feasible races are real races that can appear in a program execution. However, not all feasible races malfunction the program. A benign race is a feasible race that does not malfunction the program execution, and it can be categorized [60] as follows,

1. ***User supplied synchronization idiom***: programmers may synchronize threads without using conventional synchronization (e.g., pthread_lock, semaphore). For instance, a programmer may use a busy waiting loop to wait for a condition, e.g., { *while(!flag)* {} *do_something();* }.

Program	Static	True	False alarms	% of false alarms
Apache	118	8	110	93%
SQLite	88	3	85	97%
Memcached	7	1	6	86%
Fmm	176	58	118	67%
Barnes	166	16	150	90%
Ocean	115	3	112	97%
Pbzip2	65	9	56	86%
Knot	157	2	155	99%
Agt	256	4	252	98%
Pfscan	17	2	15	88%
Average				90%

Table 5.1: Races Detected by a Static Detector, RELAY [36]

2. ***Double checks***: a programmer may insert a check for a global condition without locking to improve the performance. The program is correct since the global condition is checked again with locking in the critical section, e.g., *if (condition)* { *lock(); {if (condition)* ... } }.

3. ***Subtle race but valid***: there can be a case that a data race between read and write operations does not harm the program execution. For instance, consider the example in Figure 5.4 which is a data race found in *facesim* program of the PARSEC benchmark suite [8]. In the example, the consumer of a queue reads the length of the queue without synchronization. Thus there is a data race. The consequence is that the consumer may return even if the queue is not actually empty. However, it does not cause any problem on the program execution, but the consumer just could waits longer.

5.2.2 Static Race Detection

Static race detectors analyze source core of a program without any execution information. As the static algorithms do not rely on any particular execution, they often have better detection coverages than dynamic race detectors. On the other hand, lack of execution information (e.g., memory addresses) makes the static detection

Program	Static	True	False alarms	% of false alarms
aget	62	31	31	50%
ctrace	10	2	8	80%
engine	7	0	7	100%
knot	12	8	4	33%
pfscan	6	0	6	100%
sntprc	46	1	45	98%
3c501	15	4	11	73%
eql	35	0	35	100%
hp100	14	8	6	43%
plip	42	11	31	74%
sis900	6	0	6	100%
slip	3	0	3	100%
sundance	5	1	4	80%
synclink	139	0	139	100%
wavelan	10	1	9	90%
Average				81%

Table 5.2: Races Detected by a Static Detector, LOCKSMITH [71]

imprecise. The first and foremost problem is to match what memory location a given operation is affected by. *lvalue* can be passed by a function as a parameter and the value can be changed by pointer arithmetic. Various analysis techniques such as calling-context-sensitive-analysis and pointer-alias-analysis are required. However, those analysis techniques are often imprecise and computationally expensive. Also, it will be hard to decide whether two lock operations at different accesses are for the same lock object or not. Thus, static race detectors produce an excessive number of false alarms. Tables 5.1 and 5.2 shows data races detected by RELAY [98] and LOCKSMITH [71], respectively. The true data races (feasible races) in Table 5.1 are verified by a dynamic race validation approach [36]. True data races in Table 5.2 are verified by manual code inspection by the author of the paper.

False positives produced by static detectors can be filtered out with various methods. RELAY [98] filters out false alarms with several static analyses. However, the filtering itself can be imprecise removing some of feasible races. Some of filtering analyses are described as follows,

1. ***Object allocation and initialization***: consider the code snippet in Fig-

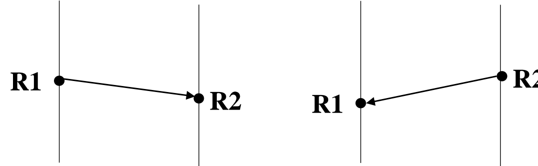
```
Func_a()
{
  Obj = malloc(…);       //allocate shared object
  Obj -> field1 = SOME; //initialize it
  Lock(&m);      //enter critical section
  Add_objt(obj); //now it begin to share
  …
}
```

Filter out because it is allocated in the same thread

(a)

If both orders can be observed, it is a feasible race

(R1, R2)
: a pair of accesses reported by a static race detector



(b)

Figure 5.5: Filtering out False Alarms from Static Race Detection. (a): RELAY [98] filters out a false alarm when it is allocated in the same thread. (b): Dynamic testing for validating feasible races. Using schedule steering, it tries to observe both orders of accesses.

ure 5.5a. The thread allocates and initializes an object, and passes the object to the critical section. A static detector will report a race for the access during the initialization (Obj→field1=SOME) since the initialization is not performed inside locking. As an approximation, RELAY filters out races for objects that are allocated inside the thread within which the race is reported.

2. ***Un-sharing***: due to the field and arithmetic insensitivity of the alias analysis, a static detector reports many false alarms. For instance, when a pointer represents a large number of fields, reported races within the field are most likely false alarms. RELAY filters out those races by considering insensitive aliasing analysis and the number of represented nodes by a pointer. Note that this is also an approximation and can remove feasible races.

In dynamic testing approaches [36, 80], a race reported by an imprecise detector is verified by scheduling thread executions. Figure 5.5b shows the basic idea of the

approaches. Given a pair of accesses reported by a race detector, the approaches try to expose both possible orders. If both orders can be exercised, we can conclude that the pair is a feasible race. To control the order of accesses, thread executions are steered. That is, the scheduler suspends one thread to proceed the other thread on the same variable access. However, reproducing both orders can be a difficult task. If one of the orders is in a rare thread interleaving, it will be hard to reproduce the execution and the validation may make a wrong decision.

RaceFuzzer [80] uses a random scheduling technique. For a given race set found by an imprecise race detector, it first randomly choose one thread to proceed it to the one of the accesses and the scheduler let the other thread to reach the other access. The approach is effective as a rare threads interleaving can be exposed easily due to the random scheduling.

Static race detection is efficient and provides good detection coverages. Nevertheless, the imprecise detection even with the filtering methods makes us difficult to use static detection approaches in practice.

5.2.3 Using Model Checking Techniques

Using Model Checker

Model checking technique is for verifying properties of a system composed of concurrent finite-state machines. For a given model of a system, a model checker systematically explores all possible thread interleavings to verify if the model satisfies a given specification. Model checking techniques can be used in various ways for data race detection. In [87], a system of thread executions is explicitly modeled to verify the property of ordered accesses of a shared resource. Various synchronizations and IPCs for VxWorks real-time operating system are modeled as timed automaton in

the UPPAAL modeling tool [40]. The property of data race free is verified by the model checking in UPPAAL.

Exploring Hidden Execution Paths/interleavings

The capability of exploring all execution paths in model checking techniques can be useful for data race detection/verification. CHESS [55] is a tool for exploring thread interleaving to find and reproduce bugs. The CHESS scheduler controls execution of threads to expose rare thread interleavings. The size of state space is reduced as the CHESS scheduler is non-preemptive and uses fewer preemption points. CHESS also has a capability of record and replay by remembering/replaying scheduling decision. DBug [82] keeps a global view of the system for state exploration. On every non-deterministic choice, a scheduling request is sent to the arbiter. The arbiter selects an event to execute next based on the global view and a permission is sent to the corresponding thread.

Predictive Analysis Technique

Predictive analysis techniques can be used to increase the coverage of concurrency bug detection (including race detection) by exploring rare thread interleavings [99, 100]. The approaches use symbolic analysis techniques based on event execution traces. Hence, the predictive analysis techniques have advantages over purely static approaches with less false alarms. In addition, events in a given execution trace are enumerated to infer more feasible interleavings uncovering more concurrency bugs than dynamic approaches.

Figure 5.6 shows an example for the predictive analysis technique. First, an execution trace ρ is collected. Trace ρ is a serialization of events in a total ordering of events such that, $\rho = e_1e_2e_3e_4e_5e_6$. Other possible interleavings can be explored

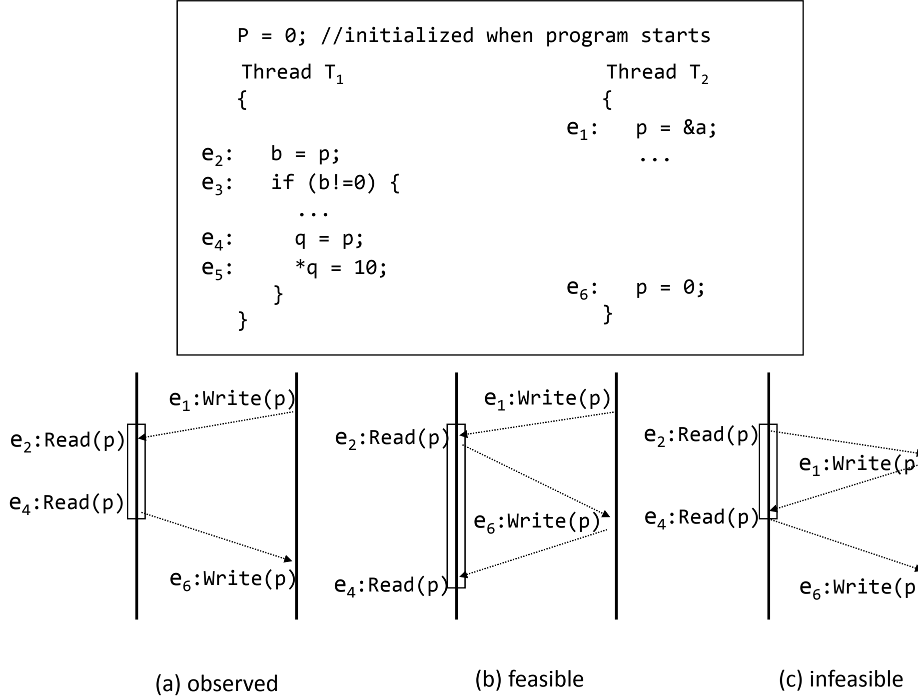


Figure 5.6: An Example for Predictive Analysis Technique [99]. (top): observed execution trace. The event number is listed on the left in the order of observation. (a): the execution order of events without atomicity violation. (b): a feasible execution ordering of events inferred by the predictive analysis technique. The inferred ordering contains an atomicity violation. (c): an infeasible ordering of events. This ordering is not explored.

by enumerating the events in trace ρ. Initially, any permutation of events is allowed and any infeasible ordering is pruned. For instance, if two events e_a and e_b are in one thread within a lock-unlock pair and an event e_c in another thread is protected by the same lock, then $e_a e_c e_b$ is not a valid enumeration of events. As shown in Figure 5.6, the enumeration $e_1 e_2 e_6 e_3 e_4$ in (b) represents a valid interleaving with an atomicity violation. On the other hand, the interleaving in (c), $e_2 e_1 e_3 e_4 e_6$, is not feasible since $e_2 \rightarrow e_1$ makes the branch statement in e_3 false and e_4 does not exist in the execution.

In the approaches, the problem of finding concurrency bugs is reduced to a satisfiability problem. A quantifier-free first-order formula Φ is built such that Φ is satisfiable if and only if there is an interleaving containing any bugs. In a simple

form, the formula Φ is expressed as,

$$\Phi := \Phi_{TM} \wedge \Phi_{SC} \wedge \Phi_{PRP} \tag{5.1}$$

where Φ_{TM} expresses a valid event ordering in each thread without considering interaction with other threads. Φ_{PRP} is the property constraint encoding failures by concurrency bugs. Φ_{SC} encodes all the valid thread interaction, i.e., valid thread interleavings. Φ_{SC} specifies valid mappings of shared memory reads and writes with considerations of happens-before relations in the trace. That is, for each variable x each shared read R_x should match a proceeding write W_x. Then, any other write W'_x should be before W_x or after R_x. As an example, consider the interleaving (c) in Figure 5.6. Read p in e_4 must be from the write in the initialization. Hence, write p in e_1 should be before the initialization (which is not possible) or after e_4. Since event e_1 is between event e_4 and the initialization, the interleaving is invalid and consequently Φ is not satisfiable. Synchronization operations can be similarly encoded. The formula Φ is decided by an SMT solver. For the complete description of the encoding and implementation, refer to [99, 100].

The uses of model checking techniques could be expedite debugging process (including detection of data races) by enabling exposures of hidden program paths or interleavings. However even optimization techniques in the approaches, the state explosion problem still remains. Cui et al. proposed one approach to reduce the size of state space [11]. The idea is to use a deterministic multithreading technique which make an execution of threads as deterministic as possible. Under the deterministic multithreading, the number of thread interleavings can be greatly reduced.

5.3 Vector Clock Based Race Detectors

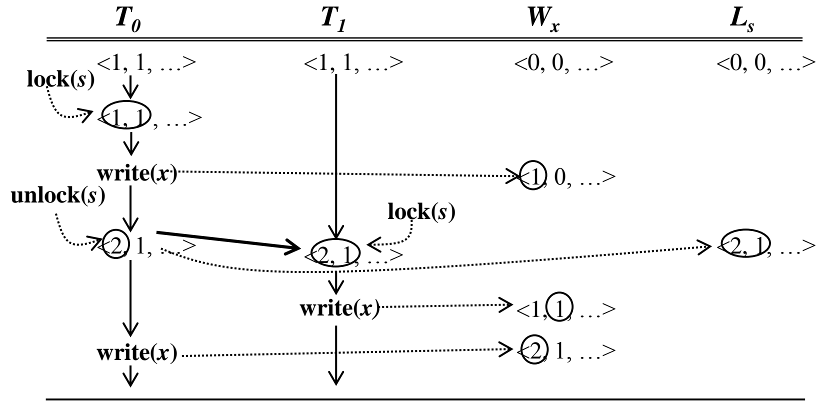
In vector clock based race detection approaches [19, 70], a data race is reported when two accesses on a memory location are not defined by the happens-before relation [39]. The approaches do not report false alarms as the detection is based on actual happens-before relations occurred during the execution of a program. However, the overheads of maintaining vector clocks for every memory location are considerably high. For backgrounds on the happens-before relation and vector clock, refer to Chapter 2.

5.3.1 DJIT+

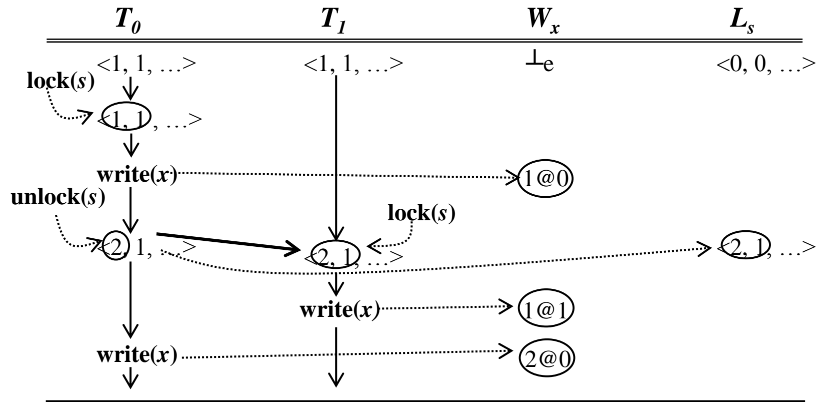
In the DJIT+ algorithm [70], an epoch [1] is defined as a code block between two release operations. DJIT+ detects only the first race for each memory location. For consecutive reads of a memory location in the same epoch, it is sufficient to check only the first read for the detection of the first race. This property is also true for consecutive write operations. With this property, the amount of race analysis can be greatly reduced.

The happens-before relation in a program execution is realized by the uses of vector clock [14]. During the execution, each thread has a vector clock. Let T_i be a vector clock for thread i. Each synchronization object s maintain a vector clock L_s to convey synchronization information from the releasing thread to the subsequent acquiring thread. Similarly a memory location x has a write vector clock W_x and a read vector clock R_x to record access history of the location. Upon initialization of a synchronization object or a memory location, all elements of the vector clock are set to zeros. When a thread is created, the vector clock elements are initialized to ones.

[1] It is defined as a timeframe in the paper. But we refer to it as an epoch for the consistency of discussion.



(a) Execution of DJIT+.



(b) Execution of FastTrack.

Figure 5.7: Example Executions of DJIT+ and FastTrack. T_0 and T_1 are vector clocks of thread 0 and thread 1, respectively. W_x and L_s are vector clocks for write x and lock s, respectively. Solid arrows show happens-before relations and dotted arrows indicate vector clock updates by the operations.

On a release operation of synchronization object s in thread i,

1. The vector clock entry for the thread itself is incremented, such that $T_i[i]{++}$.

2. The vector clock for the object s is updated to the element-wise maximum of vector clocks of thread i and object s.

Upon the subsequent acquire operation of the object s by thread j,

1. The vector clock for thread j is updated as the element-wise maximum of vector clocks of thread j and object s.

Notice that as threads are synchronized, the synchronization information is recorded in the threads' vector clocks which are conveyed by synchronization objects. If there has not been any synchronization from thread i to thread j either directly or transitively, $T_j[i]$ will keep the initialization value.

When thread i accesses a memory location x, a data race is detected by comparing vector clocks of the thread and the memory location. If the previous access represented by the vector clocks W_x or R_x, does not happens-before the current access represent by the thread vector clock T_i, then the access is a conflict. Upon the first write of x in an epoch by thread i,

1. A write-write data race is reported if there is another thread j whose write to x is not known to thread i. The check is done by element-wise comparison of vector clock elements of W_x and T_i. That is, it is a write-write race if $\exists j, i \neq j, W_x[j] \geq T_i[j]$. If there was a synchronization from thread j to thread i between the previous and current writes, then the accesses should have been ordered and $W_x[j] < T_i[j]$.

2. Thread i updates W_x such that, $W_x[i] = T_i[i]$.

3. Similarly with the read vector clock R_x, a read-write data race is checked and R_x is updated accordingly.

A similar protocol can be applied to read operations. Figure 5.7a shows an example of how DJIT+ detects a data race. In the example, consider the first write x in thread 0 and the write in thread 1. When thread 1 writes to x, it checks a write-write data race by comparing T_1 with W_x. The accesses are ordered by the happens-before relation and not a data race since $\forall i, i \neq 1, W_x[i] < T_1[i]$. As another example, consider the second write x in thread 0 and the write in thread 1, and assume that the

second write x in thread 0 is physically observed later than the write in thread 1. When thread 0 writes to x, it detects a data race since $W_x[1] \geq T_0[1]$.

By the property of checking only the first read and write in an epoch, run-time performance can be significantly improved. However, there still exists significant overheads in time and space for maintaining the vector clocks of shared memory locations.

5.3.2 FastTrack

FastTrack [19] is based on DJIT+ and provides a significant performance enhancement over DJIT+ with the same detection precision as DJIT+. FastTrack exploits the insight that, in most cases the last access of a memory location can provide enough information for detecting data races instead of using the full vector clock representation. An epoch representation denotes the last access of a memory location. If the last access was made by thread t at logical clock c, then the epoch is denoted as $c@t$ using only two scalars. For all writes to a memory location, the epoch representation can be used instead of the full vector clock because all writes to the location are totally ordered by the happens-before relation before the first race on the location. This leads to a reduction in time and space overheads from $O(n)$ (where n is the number of threads in the execution) to $O(1)$. For read operations, the epoch representation cannot be used all the time since read operations can be performed without locking (i.e., read shared). Thus, read vector clock, R_x, is replaced with an adaptive representation which uses a full vector clock only when the read is shared with other threads without protection. Based on the adaptive representation, the overhead for reads can be reduced from $O(n)$ to nearly $O(1)$.

Figure 5.7b shows an example of the FastTrack detection for the same execution shown in Figure 5.7a. In the example, except for the write vector W_x all the other

vector clock updates are identical as in the DJIT+ execution. For the write vector clock, an epoch represent with two scalars (i.e., $O(1)$ space overhead) is used instead of the full vector clock representation, (i.e., $O(n)$ space overhead). When thread 1 writes to x, the epoch representation 1@0 for the write vector is compared with the thread vector clock with $O(1)$ time overhead.

5.4 Dynamic Granularity Algorithm

FastTrack is a fast and space-efficient race-detection tool but it still needs to keep vector clocks for each memory location. This is not problematic for object-oriented programming languages since detection unit can be either a field or an object. For C/C++ programs, it is not easy to detect data structure boundaries (e.g., dynamically allocated *struct* or array) and moreover data are often protected in fine grained (e.g., a byte or a word). A simple way to rectify the problem is to use a fixed granularity. However using a fixed granularity (e.g., word) would produce a large number of false alarms and does not help reducing the overheads for many cases as shown in our evaluation results.

In this section, we present a dynamic granularity algorithm which enables vector clock based race detectors to use detection granularity as large as possible with minimal false alarms. The algorithm is described on the assumption of using DJIT+ or FastTrack detectors (That is, a thread's execution is defined as a sequence of epochs, and for consecutive accesses of a memory location in an epoch, only the first read and write are checked.) In the description, *two vector clocks are the same* when they are the same size and their contents are of equal value, and both a vector clock and an epoch representation are referred to as *a vector clock*.

The dynamic granularity is realized by sharing vector clocks with neighboring memory locations. The basic idea is illustrated in Figure 5.8. The sharing heuristic

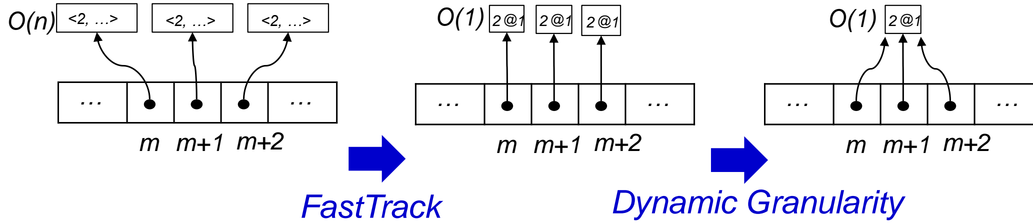


Figure 5.8: Idea of Dynamic Granularity. A large detection granularity is achieved by sharing vector clocks with neighboring memory locations.

is based on the following observations:

1. Neighboring memory locations (e.g., array, *struct*) tend to be accessed together whether the locations have data races or not. Hence, they can have the same vector clock.

2. At initialization, a data structure is often accessed in its entirety, e.g., zero-out an array, even if its elements are protected separately afterward.

3. There are groups of memory locations that are accessed only in one epoch for the entire lifetime of the location, e.g., dynamically allocated memory locations that are used temporarily.

With these observations, the dynamic granularity algorithm is realized with a vector clock state machine that is described in the following subsection.

5.4.1 *Vector Clock State Machine*

For a memory location, we maintain a read location and a write location separately. Hence, only the same access type (read or write) of vector clocks can be shared. Let L be a location which can be either a read or write location. When L is accessed for the first time, a vector clock is created for it. The sharing state of each location is maintained by a state machine attached to its vector clock as shown in Figure 5.9. The state machine basically has four states. In the first epoch access,

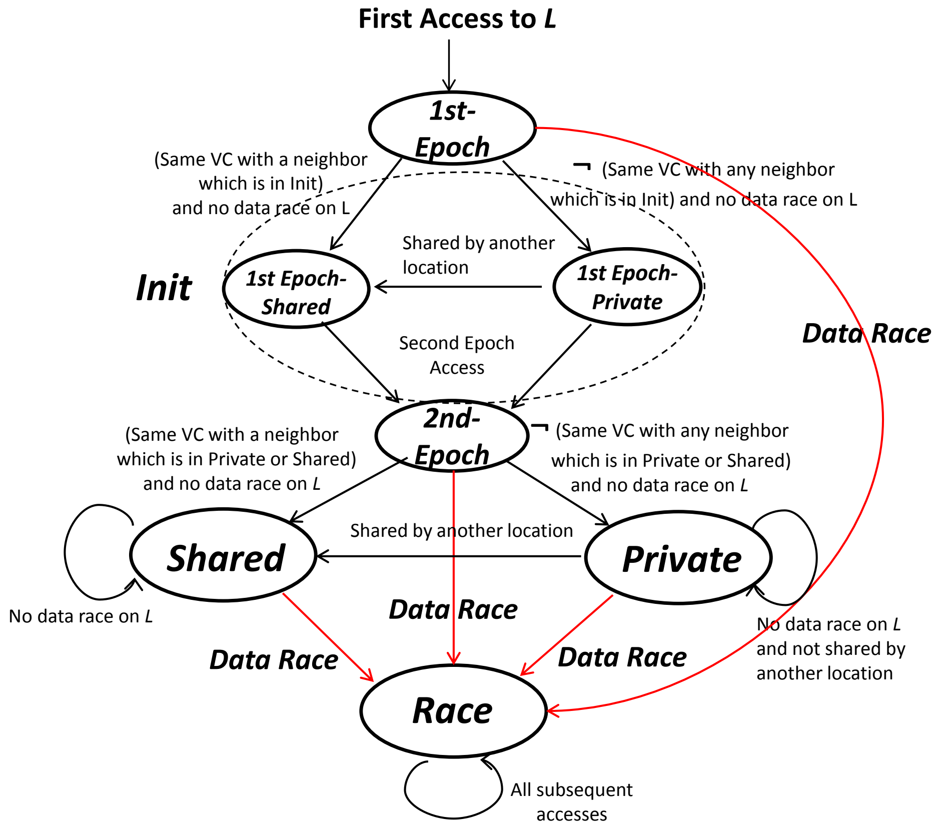


Figure 5.9: Vector Clock State Machine for Each Read or Write Location.

the vector clock is temporarily shared with its neighbor if the neighbor has the same vector clock. When the second epoch access begins at a location, the shared vector clock at the location is split and new sharing decision is made.

A neighbor of L is a memory location adjacent to L that is considered to share potentially a vector clock with L. A location L can have two neighbors that one is located left (a predecessor of L) and the other is located right (a successor of L). During the first epoch, the neighbors are the nearest predecessor and successor that have valid vector clocks. A new access to a location L initiates a vector clock in *Init* state. This vector clock can be shared with L's neighbors if they have the same vector clock and are in *Init* state as well. For an access to a location L in the 2nd epoch, the neighbors are at locations $L - size$ and $L + size$ where $size$ is the data size of the access. As long as the neighbors are not in *Init* or *Race* state, we compare the vector

clock of L with those of its neighbors. If the vector clocks are equal, the state is set to *Shared*. Otherwise, it is *Private*. The four states of a vector clock are explained in detail as follows,

Init: When L is accessed first time, its vector clock is initiated and is set to this state until the next epoch access. This *Init* state is intended to approximate the initialization process. Note that, even if two different memory locations in a data structure are protected separately (with two different locks), the locations of the data structure may be initialized together and have the same vector clock during the first epoch. Hence, if we make a firm sharing decision at the initialization, the vector clock can be inaccurately updated leading to a false alarm. However, starting from the 2$^{\text{nd}}$ epoch, a new sharing decision is made and the locations have their own private vector clocks.

While in *Init* state, L can be in *1st-Epoch-Shared* sub-state if one of the neighbors has the same vector clock and is in *Init* state. Thus, L shares temporarily its vector clock with its neighbors during the first epoch. When there is no neighbor that has the same vector clock as the location L, the state of L becomes *1st-Epoch-Private*. The state of L can transition to *1st-Epoch-Shared* when a new neighbor location L' is initiated and L' has the same vector clock as L. The rationale behind the temporary sharing is that there could be many memory locations that are accessed only in one epoch. Examples include dynamically allocated memory or groups of memory locations in a big data structure that are used only in one epoch. As our experimental results show, having this *Init* state saves a considerable amount of memory for some applications. Upon the next epoch access, L has its own vector clock and state, and the new sharing decision is made for the rest lifetime of the location L.

Shared: On the second epoch access of L, if there is no data race on L (and no read-read conflict for a read location) and there exists a neighbor that has the same

vector clock as L and is in either *Shared* or *Private*, the location L shares its vector clock with the neighbor. Also, the state of L can transition from *Private* to *Shared* when L becomes a neighbor of another location L' that has the same vector clock as L.

Private: When there is no neighbor that has the same vector clock as L, the state of L becomes *Private* on the second epoch access.

Race: On a data race, the state of L becomes *Race*. If there are memory locations sharing the same vector clock with L, the sharing is terminated and each of these locations become *Race* and is assigned with a private vector clock.

5.4.2 Dynamic Granularity

The dynamic granularity is achieved by sharing vector clocks with neighboring address locations. The detection starts with byte granularity for every location and the granularity is increased as more neighboring locations share the same vector clock. The vector comparison to determine vector clock sharing can be an expensive operation. However, following the vector clock state machine, there can be at most two sharing decisions for the lifetime of a memory location and it requires only $O(1)$ time overhead in most cases when the FastTrack algorithm is used. In fact, we can have a significant performance enhancement by the use of dynamic granularity since, as we change to a large granularity, multiple accesses may be treated as the same epoch accesses.

5.5 Implementation

We have implemented the FastTrack algorithm for data race detection of C/C++ programs with fixed (byte and word) and dynamic granularities. Intel PIN 2.11 [47] is used for dynamic instrumentation of the programs.

```
void memoryRead(uint addr, uint size, uint tid)
{
  if (nonSharedRead(addr) || sameEpoch(tid, addr))
    return;

  Location L = findReadAccess(addr);
  if (!L) {                              // The first access of addr
    L = insertRead(addr, size);
    shareFirstEpoch(L, addr, size);      // Temporary sharing
    L→ state = Init;
  } else if (L→state==Init) {            // Second epoch access
    split(L, addr, size);                // Split for new sharing
    shareSecondEpoch(L, addr, size);     // New sharing decision
    if (L→count>1)
      L→state = Shared;
    else
      L→state = Private;
  }

  //if race found on addr, split all vectors sharing with L
  //  and set states of locations to Race
  if (raceFound(addr))
    splitAndSetRace(L, addr);

  //remember access L into bitmap for this thread
  //the bitmap is reset at the next epoch of this thread
  insertEpochAccess(tid, addr);
}
```

Figure 5.10: Instrumentation Code for Memory Read.

5.5.1 Instrumentation

To trace all shared memory accesses, every data access operation is instrumented. If an instruction accesses non-shared memory (e.g., stack), the instrumentation routine returns immediately. Figure 5.10 shows pseudocode for memory read instructions. Memory write can be similarly described and we omit the FastTrack algorithm for clarity.

When an access is not the first read or write in an epoch, vector clock updates and data race checking on that access can be skipped according to the FastTrack algorithm. Checking the same epoch access can be costly since looking up a vector clock from a global data structure requires synchronization of threads. To reduce

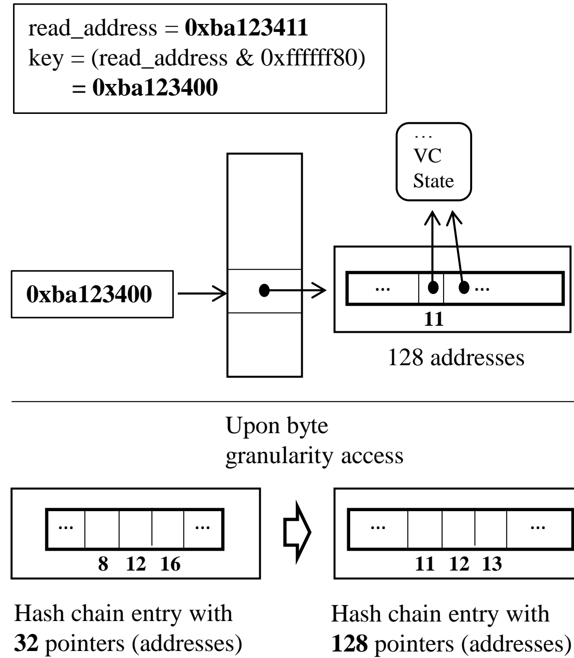


Figure 5.11: (Top): A separate chaining hash table implementation. Each entry can contain m addresses (shown a case $m = 128$). (Bottom): The size of indexing array in a hash entry is changed from $m/4$ to m when byte granularity access begins in the entry.

overhead, a per-thread bitmap is implemented. When the first access is made in an epoch, the access is set in the bitmap and the bitmap is reset for every lock release operation. Because the bitmap is a thread local data structure, checking the same epoch is more efficient than looking up a global data structure.

The mechanism for dynamic granularity is invoked at the first two epochs for each read or write location. Thus, the overhead can be negligible. Also it will be straightforward to apply dynamic granularity into existing data race detection tools.

5.5.2 Data Structures

To find the vector clock of each read or write location, a chained hash table is implemented as shown in Figure 5.11. For efficient sequential processing such as deleting vector clock entries from free() and vector clock sharing process, each

chain entry contains vector clocks for multiple memory locations rather than just one vector clock. Each hash chain entry has *an indexing array* which can contain up to m pointers for vector clock entries. For a 32-bit address, the upper address (upper $32 - \log_2 m$ bits of the address) is hashed into the table to locate the corresponding hash entry. Then, the vector clock entry for the address is indexed using the lower address (lower $\log_2 m$ bits of the address).

The use of indexing array makes sequential processing efficient, but a considerable amount of memory can be wasted. For instance, assume that an indexing array contains 128 byte addresses (i.e., 128 pointers) and the program only makes word-aligned accesses. Then, 3/4 of the indexing array (96 words) will not be used. To save memory on the indexing array, the size of the indexing array in the hash entry changes according to memory access patterns. When a new hash entry is created, it starts with an array of $m/4$ pointers since the most common access pattern is word access. When a byte access (i.e., the address is not word or half-word aligned) is detected, the array is expended to have m pointers.

5.6 Evaluation

In this section, we present the efficiency and effectiveness of our dynamic granularity algorithm. First, we show our experimental results of the FastTrack detector with fixed (byte and word) and dynamic granularities. Second, analysis results on the state machine are presented. Lastly, performance measures of two popular data race detection tools, Valgrind DRD [61, 94] and Intel Inspector XE [27], are compared with the FastTrack detector using dynamic granularity. All experiments were performed on Ubuntu 12.04 with kernel version 3.2.0 and Intel Core Duo CPU with 4 GB of RAM.

All experiments were run with 11 benchmarks: 8 from the PARSEC-2.1 bench-

mark suite [8] that are implemented with the POSIX thread library and 3 from popular multithreaded applications: *FFmpeg* [89], a multimedia encoder/decoder; *pbzip2* [30], a parallel version of bzip2; and *hmmsearch* [15], a sequence search tool in bioinformatics. For input sets of the PARSEC benchmark programs, the *simsmall* input set is used for *raytrace* while the *simlarge* is used for the rest 7 programs. Inputs for the other three programs are chosen to have similar run times as the PARSEC benchmark programs.

5.6.1 Performance and Detection Precision

Table 5.3 shows overall experimental results of the FastTrack detector with three different granularities. "Total shared accesses" column shows the total number of shared reads and writes during each benchmark program run. "Max. # of vectors in byte granularity" column indicates the maximum number of vector clocks present for the execution of each program in byte granularity run. These two columns combined with the number of threads, give us a general idea of the instrumentation overhead in the detection. "Slowdown" and "Memory overhead" columns report runtime and memory overhead of each detection mechanism as the ratios to the run time and maximum memory used in the un-instrumented program execution. "# of Detected Data Races" columns show the number of data races detected by each granularity detector.

Overall Results. The results show that the dynamic granularity detector is on average 1.43x and 1.25x faster than the byte-granularity and the word-granularity detectors, respectively. For memory overhead, on average the dynamic granularity detector consumes 60% less memory than the byte granularity detector and 23% less memory than the word granularity detector.

For benchmarks *facesim, fluidanimate, raytrace, canneal, streamcluster,* and *hmm-*

	Benchmark Program	Total shared accesses (million)	Max. # of vectors in byte granularity	# of threads	Base time (sec)	Base memory(MB)	Slowdown			Memory Overhead			# of Detected Data Races		
							Byte granularity	Word granularity	Dynamic granularity	Byte granularity	Word granularity	Dynamic granularity	Byte granularity	Word granularity	Dynamic granularity
PARSEC	facesim	8033	93,930,447	2	6.1	288	138	138	102	8.8	8.8	4.6	8909	8909	8909
PARSEC	ferret	3856	83,678,104	11	6.7	146	65	57	52	16.6	11.7	8.9	2	2	2
PARSEC	fluidanimate	2443	11,220,394	3	2.0	248	87	87	81	2.6	2.6	2.2	1	1	1
PARSEC	raytrace	18	3,291,927	3	9.5	170	27	27	27	2.1	2.1	2.0	13	13	13
PARSEC	x264	3392	12,550,683	256	2.2	49	75	55	64	20.8	9.6	9.0	1300	993	1313
PARSEC	canneal	359	7,141,372	3	6.5	104	13	13	13	5.3	5.3	5.1	0	0	0
PARSEC	Dedup	10003	9,208,539	7	7.7	2682	152	76	85	1.0	1.0	1.0	0	0	0
PARSEC	streamcluster	8030	2,277,958	5	3.8	30	245	245	137	4.8	4.8	3.7	1053	1053	1079
	ffmpeg	5790	7,195,586	9	3.0	95	121	106	109	4.0	3.0	3.1	1	9	1
	pbzip2	7239	8,842,583	6	5.7	67	64	49	39	5.4	4.2	3.4	0	0	0
	hmmsearch	38050	961,831	3	26.6	23	84	83	45	4.9	4.9	4.3	1	1	1
Average							97	85	68	6.9	5.3	4.3			

Table 5.3: Overall Experimental Results.

search, memory consumption is neither reduced nor does detection become faster when we switch from byte granularity to word granularity. Since the sizes of most accesses in those benchmarks are equal to or greater than a word, no vector clock is created for non-word-aligned locations. Thus, using word granularity does not help to reduce the overhead of vector clock operations. However, except for *raytrace, canneal*, the use of dynamic granularity enhances the detection both in time and memory space. This suggests the advantage of using a large granularity crossing word boundaries. The results from *ferret* and *pbzip2* show improvements both in word granularity and dynamic granularity, but the use of dynamic granularity has more benefits than the use of word granularity. It may be strange to see that the factor of memory overhead for *dedup* is 1.0 for all detectors. Note that the maximum overhead does not always occur when the maximum memory is used in the benchmark. In fact, *dedup* uses a large amount of memory (about 2.7 GB) at the beginning of the execution when the detection overhead is close to zero. Then, the memory usage is gradually decreased while the peak memory overhead from the detectors occurs afterward.

For detection precision, there are few discrepancies among the detectors as shown

Benchmark program	Base Memory (MB)	Byte Granularity				Word Granularity				Dynamic Granularity			
		Hash(MB)	Vector Clock (MB)	Bitmap (MB)	Overhead total (MB)	Hash(MB)	Vector Clock (MB)	Bitmap (MB)	Overhead total (MB)	Hash(MB)	Vector Clock (MB)	Bitmap (MB)	Overhead total (MB)
facesim	288	513	1505	132	2149	514	1503	132	2148	517	273	131	921
ferret	146	458	1573	66	2097	259	1072	57	1388	454	469	64	988
fluidanimate	248	132	180	27	339	132	180	27	338	132	74	27	233
raytrace	170	35	53	15	103	30	53	15	97	35	22	15	72
x264	49	77	233	7	317	33	89	7	129	77	44	7	128
canneal	104	87	176	52	315	87	176	52	315	87	155	52	294
dedup	2682	212	148	57	417	147	100	58	305	214	88	56	358
streamcluster	30	11	37	6	54	11	36	6	54	11	3	6	21
ffmpeg	95	37	118	7	162	18	43	7	68	37	28	7	72
pbzip2	67	49	141	17	207	37	89	17	143	50	4	16	70
hmmsearch	23	12	15	3	30	12	15	3	30	13	1	3	17
Average		148	380	35	563	116	305	35	456	148	105	35	288

Table 5.4: Memory Overhead of FastTrack Detection with Different Granularities.

in Table 5.3. With word granularity, 993 data races are reported for *x264* while the dynamic granularity detector reported more data races. When word granularity is used, non-word-aligned addresses are masked to word boundary and data races for those locations are detected as one race. That is how the word granularity detector reported less number of data races for *x264*. We carefully inspected data races from *x264* found by the dynamic granularity detector and noticed that there were 4 write locations which were sharing a vector clock with one location having a data race. More data races from *ffmpeg* by the word granularity detector and from *streamcluster* by the dynamic granularity detector are found to be false alarms due to inaccurate updates of vector clocks when large detection granularities are used.

Memory Overhead. Table 5.4 shows the details of memory overhead. For each granularity, three major overhead factors are shown. "Hash" column indicates the maximum memory used for the hash tables and the hash entries to index vector clocks. "Vector clock" column gives the maximum memory used to store vector clocks. The third column, "Bitmap", is the maximum memory used for the bitmap data structures for checking same epoch accesses. The overhead is measured based

Benchmark program	Max. # of vector clocks Byte Granularity (thousand)	Word Granularity (thousand)	Dynamic Granularity (thousand)	Avg. sharing count
facesim	93,930	93,808	16,991	5.5
ferret	83,678	52,375	24,689	3.4
fluidanimate	11,220	11,174	4,590	2.4
raytrace	3,291	3,285	1,319	2.5
x264	12,550	4,803	2,019	6.2
canneal	7,141	7,141	5,812	1.2
dedup	9,208	6,226	5,474	1.7
streamcluster	2,277	2,245	193	11.8
ffmpeg	7,195	2,620	1,696	4.2
pbzip2	8,842	5,570	265	33.3
hmmsearch	961	950	53	17.9

Table 5.5: Maximum Number of Vector Clocks Present.

on object size and is slightly underestimated since the size of memory allocated for a data object is usually little more than the actual size of the type.

The dynamic granularity algorithm saves a substantial amount of memory used for vector clock allocations (as shown in "Vector clock" columns). Another view of memory savings on vector clocks is shown in Table 5.5 which lists the maximum numbers of vector clocks during program executions. On average, the dynamic granularity detector uses roughly 4x and 3x less memory for vector clocks than the byte granularity and the word granularity detectors, respectively. Indexing costs of the byte granularity and the dynamic granularity detectors are almost same since the use of dynamic granularity does not save memory on indexing vector clocks (as shown in "Hash" columns of Table 5.4). The use of word granularity saves memory on indexing for some benchmark programs because the addresses are mostly word-aligned, thus using smaller indexing arrays in hash entries.

Slowdown. Speedups can be achieved in two ways by the use of a large granularity. Firstly, in DJIT+ based race detectors including FastTrack, the vector clock operations are performed only for the first read and write of a shared memory loca-

Benchmark Program	Slowdown			Same epoch		
	Byte Granularity	Word Granularity	Dynamic Granularity	Byte Granularity	Word Granularity	Dynamic Granularity
facesim	138	138	102	74%	74%	94%
ferret	65	57	52	78%	83%	87%
fluidanimate	87	87	81	89%	89%	94%
raytrace	27	27	27	65%	65%	68%
x264	75	55	64	67%	90%	78%
canneal	13	13	13	97%	97%	97%
dedup	152	76	85	85%	93%	85%
streamcluster	245	245	137	50%	51%	97%
ffmpeg	121	106	109	68%	90%	84%
pbzip2	64	49	39	95%	97%	95%
hmmsearch	84	83	45	83%	83%	98%
Average	97	85	68	77%	83%	89%

Table 5.6: Measures of Same Epoch Accesses.

tion during each epoch. The use of a large granularity makes multiple accesses as one access. Thus, there are more same epoch accesses enhancing the detection performance. In Table 5.6, we show the percentage of same epoch accesses along with the slowdown for each benchmark program. The results suggest that in most cases the performance gains from a large granularity are consistent with the percentage of same epoch accesses. For the cases of *canneal* and *raytrace*, as the percentages of same epoch accesses do not vary noticeably among different granularities, there is no performance enhancement by the use of a large granularity.

Second, speedup comes from the reduction of vector clock allocation and deallocation operations. For the case of *pbzip2*, the dynamic granularity detector is 1.6x faster than the byte granularity detector while the percentages of same epoch accesses are same. On the other hand, the average number of locations that share a vector clocks is 33.3 under dynamic granularity as shown in Table 5.5. This implies that there will be about 33 times less vector clock creation and deletion operations. The other interesting case is *dedup*. The program has the same percentage of same epoch accesses for both byte and dynamic granularities and the average number of

Benchmark program	Max. Memory(MB)		# of Detected Data Races	
	No sharing at Init	Sharing at Init	No Init state	With Init state
facesim	2180	1317	9210	8909
ferret	1808	1302	2	2
fluidanimate	604	551	18529	1
raytrace	348	334	13	13
x264	470	442	1315	1313
canneal	550	530	0	0
dedup	2729	2730	0	0
streamcluster	142	111	1079	1079
ffmpeg	301	294	1	1
pbzip2	359	225	2	0
hmmsearch	107	99	1	1
Average	873	721		

Table 5.7: Comparisons of State Machines with Different Configurations.

sharing vector clocks is only 1.7. However, the dynamic granularity detector is 1.78x faster than the byte granularity detector. The reason is that there are an excessive number of dynamic memory locations in *dedup*. On average, there is about 1.7 GB of memory allocated and de-allocated in the 11 benchmark programs whereas it is 14GB in *dedup*.

5.6.2 Analysis of State Machine

The sharing decision for realizing dynamic granularity is made twice for the lifetime of a location L (read or write). In the first epoch, L tries to share a vector clock with its neighbors temporarily. In the second epoch, a new sharing decision is made for the rest lifetime of L. We make a firm sharing decision at the second epoch (after initialization of L) since some groups of data structures can be initialized at the same segment of code even if they are accessed separately afterward. This design makes the sharing decision accurate. The temporary sharing at the first epoch may save a considerable amount of memory because there could be groups of locations that are accessed together only once in the same epoch and if that is the case, we do

Benchmark program	Base time (sec)	Base Memory (MB)	Slowdown			Memory Overhead			# of Detected Data Races		
			Valgrind DRD	Intel Inspector XE	Dynamic granularity	Valgrind DRD	Intel Inspector XE	Dynamic granularity	Valgrind DRD	Intel Inspector XE	Dynamic granularity
facesim	6.1	288	59	128	102	2.2	6.0	4.6	8909	31	8909
ferret	6.7	146	748	87	52	2.6	5.0	8.9	108	4	2
fluidanimate	2.0	248	--	89	81	--	12.4	2.2	--	7	1
raytrace	9.5	170	42	17	27	1.9	4.1	2.0	16	0	13
x264	2.2	49	143	246	64	3.2	22.1	9.0	988	218	1313
canneal	6.5	104	31	41	13	8.2	11.9	5.1	0	0	0
dedup	7.7	2682	--	--	85	--	--	1.0	--	--	0
streamcluster	3.8	30	66	108	137	4.2	17.5	3.7	1067	61	1079
ffmpeg	3.0	95	120	--	109	2.6	--	3.1	0	--	1
pbzip2	5.7	67	64	99	39	2.9	8.6	3.4	0	0	0
hmmsearch	26.6	23	74	64	45	4.4	21.9	4.3	1	2	1
Average			150	98	68	3.6	12.2	4.3			

Table 5.8: Performance Comparisons of Valgrind DRD, Intel Inspector XE and the FastTrack with Dynamic Granularity.

not have to keep a separate vector clock for each of them. Notice that there is no possibility of false alarms by the temporary sharing at *Init* state. Table 5.7 shows the effectiveness of this design. Column 2 and 3 show the maximum memory used without and with temporarily sharing at the first epoch. Column 4 shows the number of detected data races without *Init* state, i.e., no temporary sharing and the sharing decision is made only once in the first epoch. Comparing with column 5 in which *Init* state is added, there could be many false alarms as the consequence of improper sharing decisions made only in the first epoch. In addition, the results suggest that there are considerable numbers of memory locations that are used only in one epoch.

5.6.3 Case Studies

In this section, we present experimental results on two popular data race detection tools, DRD in Valgrind-3.8.1 [94] and Intel Inspector XE 2013 [27] update-5. Also comparison results with our dynamic granularity on FastTrack are given. DRD, a tool for programs written with the POSIX library, detects various errors including data

races, lock contention delays, and misuses of the POSIX library. The race detection algorithm in DRD is based on RecPlay [77]. Since DRD does not keep vector clocks for each memory location in its segment comparison approach, we expect that DRD uses less memory but is slower than the FastTrack detector. Intel Inspector XE is a memory and thread error checker that is capable of detecting various errors including data races, deadlocks, and cross-thread stack access.

Inspector XE provides a GUI with comprehensive analysis reports, including the source code location of an error, calling stack analyses, and suggestion for fixing any detected errors. Likewise, DRD provides execution context for each error as well as the location that the error occurs. Race report from our implementation of FastTrack is not as comprehensive as the two tools, but we provide the location of a race along with the previous access location, thread ids, and the race memory address. The information should be sufficient for developers to fix the problems easily.

For Inspector XE, the command-line version was used and only data race detection is enabled. The two detectors used byte granularity and all detectors, including our dynamic granularity version of FastTrack, traced all modules including shared libraries. For the dynamic granularity detector, we applied the similar suppression rules as in DRD, e.g., suppressed data races detected from *libc* and *ld*. The dynamic granularity detector and DRD report the first race for each memory location while Inspector XE uses a combination of instruction pointers and timeline when a race occurs to distinguish races. Thus, Inspector XE may report the same accesses on a specific memory location as multiple races or multiple accesses issued at the same instruction points as one race. DRD and Inspector XE classify the detected data races with execution context, but in the experimental results we list the raw number of data races before the classifications.

The comparison results are shown in Table 5.8. Both Inspector XE and DRD ex-

ited on *dedup* runs with out of memory warnings. DRD on *fluidanimate* and Inspector XE on *ffmpeg* ran for more than 24 hours before we stopped the analyses.

As we expected before the experiment, DRD is slower than the FastTrack detector with dynamic granularity and even slower than the FastTrack detector with byte granularity. However, DRD consumes less memory than the FastTrack detector with dynamic granularity. The comparison results also suggest that the dynamic granularity detector is as accurate as the other two detectors. All three detectors detected the same race for *hmmsearch* (Inspector XE reported the same race one more time in a different timeline). The dynamic granularity detector and Inspector XE reported the same races for three benchmarks, *ferret, fluidanimate,* and *streamcluster*. For *raytrace*, the dynamic granularity detector and DRD reported the same races, but DRD reported more races from pthread library which was suppressed by the dynamic granularity detector. DRD detected no race for *ffmpeg* while the dynamic granularity detector reported one race. We manually inspected the source code and found that it was a data race by the two worker threads accessing a shared variable without protection.

5.7 Related Work

5.7.1 Hybrid Race Detectors

Aside from the 3 basic approaches for race detections, [19, 70, 76, 77, 79, 94, 105], a variety of hybrid race detectors have been proposed in which Eraser's Lockset algorithm is combined with the happens-before algorithm to have better detection coverage and to avoid false alarms. O'Callahan and Choi [64] have proposed a race detection algorithm in which a subset of happens-before relations is added to a Lockset based detector. The detector is optimized by detecting redundant event accesses

and by the use of a "two phase" approach of detailed and simple mode detections. MultiRace [70] combines DJIT+ and Lockset algorithm and only check the first access in each time frame. In MultiRace, the number of detection operations is reduced based on the information produced from LockSet and false alarms from LockSet are filtered out by happens-before relations. ThreadSanitizer [81] is a hybrid race detector for C++ programs that offers tunable options to users. Its dynamic annotations allow the detector to be aware of user defined synchronizations. Thus, the tool hides certain false alarms and benign races. RaceTrack [105] incorporates the happens-before relation into the LockSet algorithm and only report races caused by concurrent accesses. One interesting idea in RaceTrack is the use of adaptive granularity. The detection granularity starts from object level and becomes field level when a potential race is detected. Unfortunately, the idea, based on object references, is not applicable to C/C++ programs.

5.7.2 Sampling/HW-assisted Approaches

LiteRace [50] is a sampling based race detector grounded in the cold-region hypothesis that infrequently accessed areas are more likely to have data races than frequently accessed areas. Accesses to code regions of different function units are sampled while all synchronization operations are collected. The sampler starts at a 100% sampling rate and the sampling rate is adaptively decreased until it reaches a lower bound. PACER [9] is another sampling based race detector that periodically samples all threads and offers a detection rate proportional to the sampling rate. These approaches offer reasonable detection rate with minimal overhead, but may miss critical data races.

As an alternative to software only race detectors, several hardware assisted race detectors have been proposed. In SigRace [57], data addresses are automatically

encoded in hardware address signatures and in a hardware module. The signatures are intersected with those of other processors to detect data races. Greatehouse et al. [23] proposed a demand-driven race detector that utilizes cache performance counters to detect data sharing between threads. When the data sharing is detected, a software race detector is enabled and run until there is no more data sharing. These approaches are efficient but require specific hardware making them impractical.

5.7.3 Data Race Detection for C/C++ Programs

While many researchers have focused on data race detection algorithms for Java programs, only a few of which have presented evaluation results for existing data race detection tools on C/C++ programs. Aikido [66] is a framework for shared data analysis in which sharing data is detected using per-thread page protection technique. The Aikido sharing detector is complementary to dynamic granularity and is effective to remove the instrumentation overhead of no-shared memory accesses. IFRit [13] is a dynamic race detection algorithm for C/C++ programs based on interference-free region which can limit the range of code instrumentation. IFRit has been compared with FastTrack and ThreadSanitizer [81]. Both researches applied the PARSEC benchmark suite in their performance evaluations, but only the *sim-small* input set was used and no memory overhead was reported. Moreover, none of them made an attempt to compare their tools with commercial grade data race detection tools.

5.7.4 Classifying/Surviving Data Races

Aside from detecting data races, a number of researches on classifying data races have been proposed. Among detected data races, many portions of them are harmless; shared data are intentionally not protected for performance reason; or the data races

are benign in a sense that they do not affect the correctness of the program. The motivation is for developers to more focus on harmful data races not distracted by a number of harmless data races. Narayanasamy et al. [60] proposed an idea of classifying data races using replay. The basic idea is to replay twice with two different orderings for a given data race. If the two replays produce the same memory and register values, the race is classified as a benign race. Portend [35] extends the idea considering different paths and different scheduling. Even if the two replays produce the same states, Portend does not conclude but explores paths before and after the race occurs and paths from different thread interleaving.

Veeraraghavan et al. proposed Frost [96] that protects programs from data races and introduced an outcome-based race detector, i.e., data races are reported when outputs of replicas diverge. Frost uses complementary schedules to force replicas to diverge on races. Depending on combinations of diverged outcomes, Frost reports data races and takes actions to survive data races. ToleRace [73] uses a concept of using thread local copy of a shared data and propagating the appropriate copy after detecting conflicting changes. ToleRace is focus on asymmetric races (i.e., one thread protects a shared variable properly while the other improperly accesses the variable) since, according to the claim in the paper, asymmetric races are very common in practice and most of them are benign.

5.7.5 Automatically Fixing Data Races

Once we have detected data races with one of the methods we have discussed, the races can be fixed by placing proper synchronization operations. However, placing correct synchronizations can be time-consuming and error-prune process resulting in new concurrency errors. Jin et al. have proposed a system that automatically fixes concurrency bugs including data races [31, 32]. In the approach, synchronization

operations are automatically placed based on a bug report. The bug report should contain enough details to place synchronizations and is required to be sound, i.e., no false positives. The challenges in the approach is to guarantee that no new bugs (e.g., deadlocks) are introduced due to the additions of synchronization. For instance, a synchronization operation (e.g., lock, unlock) can be added inside a branch or loop statement, and it can cause a deadlock when the branch/loop is not taken. Also, putting a synchronization in a recursive function call can cause a deadlock. To avoid those case, the system statically analyze the program and place synchronizations for both branch/loop taken and not-taken paths, and locks are replaced with reentrant locks.

5.8 Chapter Conclusions

In this chapter, we have presented a dynamic granularity algorithm for C/C++ programs that enables vector clock based race detectors to adjust detection granularity. A vector clock state machine is employed to determine when vector clocks can be shared. The state machine also considers the initialization patterns of data structures. Thus, possible false alarms due to vector clock sharing can be reduced.

Our experimental results show that the dynamic granularity detector outperforms the FastTrack detector with byte or word granularities and also outperforms two existing data race detection tools, Valgrind DRD and Intel Inspector XE.

Although our approach for efficient data race detection is promising, the runtime overhead would be still high for the routine uses. As multi-core machines are readily available today, the overhead can be alleviated by exploiting multiple/extra cores in a system. In Chapter 6, we study the overhead and scalability of data race detection on multi-core machines and propose a paralleling data race detection algorithm.

Chapter 6

DATA RACE DETECTION ON MULTI-CORE SYSTEMS

In Chapter 5, we have proposed an efficient data race detection for C/C++ programs. Although our approach is viable outperforming commercial grade data race detectors, the runtime overhead would be still high for the routine uses. In this chapter, we present a novel approach to parallelize data race detection in multi-core SMP (Symmetric multiprocessing) machines. In our approach, data access information needed for dynamic detection is collected at application threads and passed to worker threads. The access information is distributed in a way that the detection operation performed by each worker thread is independent of those of other worker threads. As a consequence, the overhead caused by locking operations in data race detector can be alleviated and multiple cores can be fully utilized to speed up and scale up the detection. Furthermore, since each worker thread deals with its own assigned access range rather than the whole address space, the executions of worker threads can exploit the spatial locality of accesses leading to an improved cache performance. We have applied our parallel approach on the FastTrack algorithm and demonstrated the validity of our approach on an Intel Xeon machine. Our experimental results show that the parallel FastTrack detector, on average, runs 2.2 times faster than the original FastTrack detector on the 8 core machine.

6.1 Introduction

As discussed in Chapter 5, static data race detectors [17, 34, 58, 71, 98] may produce excessive number of false alarms which hinders developers' focus on real data races. Hence, in practice dynamic detection approaches are often preferred

to static detectors due to the soundness of the detection. Nevertheless, the high runtime overhead impedes routine uses of the detection. There have been broadly two approaches to reduce the runtime overhead. The first approach is to reduce the amount of work that is fed into a detection algorithm. Sampling approaches [9, 50] can be efficient but we may miss critical data races in a program. DJIT+ [70] has greatly reduced the number of checks for data race analysis with the concept of timeframes. In [78], memory accesses that do not need to be checked can be removed from the detection by various filters. The use of large detection granularity can also reduce the amount of work for data race analysis. RaceTrack [105] uses adaptive granularity in which the detection granularity is changed from array/object to byte/field when a potential data race is detected. In dynamic granularity [84], starting with byte granularity, detection granularity is adapted by sharing vector clocks with neighboring memory locations. Another approach is to simplify the detection operations. For instance, by the adaptive representation of vector clock, FastTrack [19] reduces the analysis and space overheads from $O(n)$ (where n is the number of threads in the execution) to nearly $O(1)$.

Despite the recent efforts to reduce the overhead of dynamic race detectors, they still cause a significant slowdown. According to [84], the FastTrack detector imposes a slowdown of 97 times on average for a set of C/C++ benchmark programs. For the same benchmark programs, Intel Inspector XE [27] and Valgrind DRD [94] slow down the executions, on average, by a factor of 98 times and 150 times, respectively.

With multi-core architectures, one promising approach is to increase parallel executions of data race detector. Wester, et.al, has presented a strategy to parallelize data race detection [101]. In the approach, thread execution is time-sliced and executed in a pipe-lined manner. That is, each thread execution is defined as a series of timeframes and the code blocks in the same time frame for all threads are executed

in a designated core. Their parallel detector speeds up the detection and scales well with multiple cores by eliminating lock cost in the detection and by increasing parallel executions. However, the approach relies on a new multithreading paradigm, uniparallelism [97] which is different from the task parallel paradigm supported by typical thread libraries. In addition, it requires modifications on OS and shared libraries, and rewriting the detection algorithm.

In this chapter, we present a novel approach to parallelize data race detection in multi-core machines. Our approach does not require any change in the underlining system and we use the same race detection algorithm, such as FastTrack. Also, our parallelization does not alter the access information and order of memory accesses which are used for the sequential version of the original detection algorithm. Hence, the parallelized data race detection provides the same detection precision with increased performance and scalability. As in the FastTrack algorithm, we assume that the detection runs on SMP (Symmetric multiprocessing) based machines.

The idea is to separate race detection from application threads and to perform data race analysis in worker threads without inter-thread dependencies. Data access information for race analysis is distributed from application threads to worker threads based on memory address. In other words, each worker thread performs data race analysis only for the memory accesses in its own address range. Note that in a conventional race detector, each application thread performs data race analysis for any memory accesses occurred in the thread. Our parallelization strategy increases scalability as we can use any number of worker threads regardless of application threads. Speedups are attained as the lock operations in the detector program are eliminated, and the executions of worker threads can exploit the spatial locality of accesses.

We have applied our approach on the FastTrack algorithm [19] and demonstrated

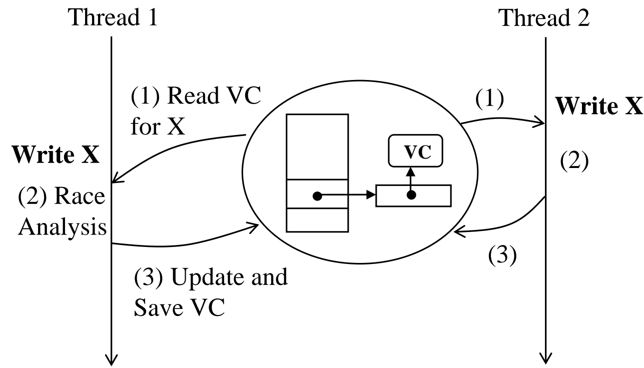


Figure 6.1: A High Level View of the FastTrack Detection When Two Threads Write to the Variable X.

the validity of our approach on an 8-core Intel Xeon machine. Our experimental results show that when 4 times more cores are used for the detection, the parallel version of FastTrack, on average, can speed up the detection by a factor of 3.3 over the original FastTrack detector. Even without additional cores, the parallel FastTrack detector runs 2.2 times faster on average than the original FastTrack detector.

The rest of this chapter is organized as follows. In Section 6.2, we present an analysis of overhead and scalability of the FastTrack algorithm on multi-core machines. Section 6.3 describes the parallel FastTrack algorithm, and in Section 6.4 we present the implementation of the parallel version of FastTrack. In Section 6.5, our experimental results are presented to show the validity of our approach. A concise survey of related work is presented in Section 6.6 and we conclude the chapter in Section 6.7. For the background on the FastTrack algorithm, refer to Section 5.3 of Chapter 5.

6.2 Overhead and Scalability of FastTrack

When a thread accesses a memory location, the FastTrack race detector must perform the following operations to analyze any data race. First, the vector clocks (for read and write) for the memory location are read from the global data structures. Second, the detection algorithm is applied by comparing the thread's vector clock

with the vector clocks for the memory location. Lastly, the vector clocks for the memory location is updated and saved into the global data structures. As illustrated in the following, these operations can lead to excessive overhead. In addition, as the detection is performed when every application thread make references to shared memory, the FastTrack detector incurs substantial runtime overhead and does not scale well on multi-core machines.

***Lock overhead*:** A dynamic race detector is a piece of code that is invoked when the application program issues data references to shared memory. Thus, if the application runs with multiple threads, so does the race detector. In the FastTrack algorithm, vector clocks should be read from and updated in global data structures as shown in Figure 6.1. When multiple threads access the global data structures, the accesses should be synchronized with lock operations at an appropriate granularity. Otherwise, the detector program itself will suffer from concurrency bugs including data races. As lock operations should be applied for every shared memory access, the overhead of race detection can be substantial. As shown in Table 6.2 of the next section, the locking overhead constitutes on average 17% and can be up to 44% of the execution time of the FastTrack detector.

***Inter-thread dependency*:** During the executions of application threads, it is often the case that a thread may block or condition-wait for the resource to be freed by another thread. Hence, CPU cores may not be effectively utilized even with sufficient number of application threads. Since the data race analysis is performed as a part of the execution of application threads, it can suffer from the same inter-thread dependencies as the application threads. Thus, when an application thread is inactive, no data race detection can be done for its memory accesses.

***Utilizing extra cores*:** The prevalence of multi-core technologies makes us believe that extra cores will be available for execution of an application. However, if

there were more CPU cores than the number of application threads, the race detection may not utilize these extra cores. We may increase the number of application threads to scale up the detection. This can lead to three potential problems. First, increasing the number of application threads may not be beneficial especially if the application is not computation-intensive. Second, changing the number of application threads may imply a different execution behavior including possible data races. Lastly, as shown in our experimental results, the detection embedded in application threads may not scale well when the number of cores increases.

***Inefficient execution of instructions*:** In an execution of the FastTrack detector, global data structures for vector clocks are shared by multiple threads, and each application thread is responsible for data race analyses of the memory accesses occurred in the thread. As a consequence, each application thread may access the global data structures whenever it reads or writes shared variables. Thus, the amount of data shared between threads is multiplied which can result in an increase of the number of cache invalidations. Also, as the working set of each thread is enlarged, the thread execution may experience a low degree of spatial locality and an increase of cache miss ratio. In addition, using lock operations on every shared memory access will incur frequent pipeline stalls leading to higher CPI. As shown in Figure 6.3, this performance penalty will become noticeable as we increase the number of application threads.

6.3 Parallel FastTrack Detector

To cope with the aforementioned problems of race detection on multi-core systems, we propose a parallel data race detection with which race analyses are decoupled from application threads. The role of an application thread is to record the shared-memory access information needed by race analysis. Additional worker threads are employed

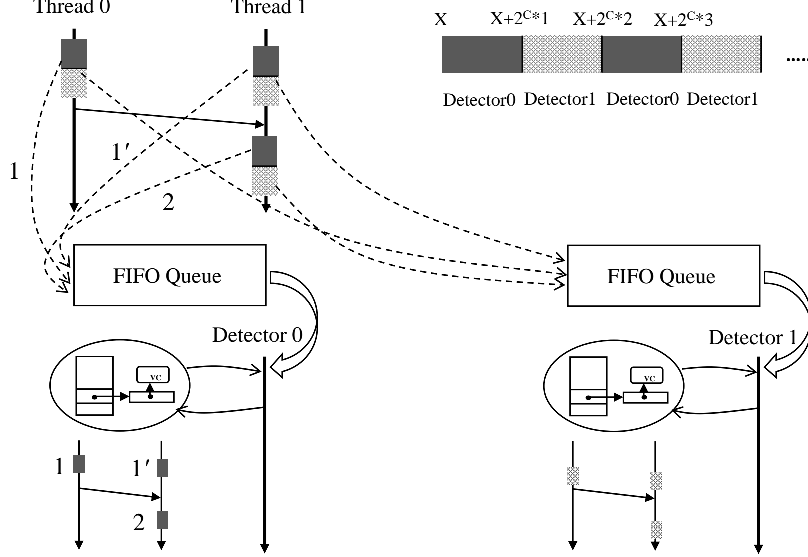


Figure 6.2: An Overview of Our Approach for a Case When Two Detection Threads Are Used. The address space is divided into two regions, and each detector is responsible only for its own address region.

to perform data race detection. We refer to the worker threads as detector/detection threads. The key point is to distribute the race analysis workload to detection threads such that (1) a detector's analysis is independent of other detection threads, and (2) the execution of application threads has a minimal impact to the race analyses performed in the detectors.

In the FastTrack detector, the same vector clock is shared by multiple threads as the detection for the memory location is performed by the multiple threads. Conversely, in our approach accesses to one memory location by multiple threads are processed by one detection thread. Assume that the shared memory space is divided into blocks of 2^C contiguous bytes and there are n detection threads. Then, accesses to the memory location of address *addr* by multiple threads are processed by a detection thread T_{id}. The detection thread is decided based on *addr* as follows,

$$T_{id} = (addr >> C) \bmod n \tag{6.1}$$

For each detection thread, a FIFO queue is maintained. Upon a shared memory

access of address *addr*, access information needed by the FastTrack race detection should be sent to the FIFO queue of detector T_{id}. Since the queue is shared by application threads and the detector, accesses to the queue should be synchronized. To minimize the synchronization, each application thread saves temporarily a chunk of access information in a local buffer for each detection thread. When the buffer is full or a synchronization operation occurs in the thread, then the pointer of the buffer is inserted to the queue and new buffer is created to save subsequent access information. Other than memory access information, execution information of a thread such as synchronization and thread creation/join is also sent to the queue. At the detector side, the pointers of the buffers are retrieved from the queue and the thread execution information is read from the buffer to perform data race analysis using the same FastTrack detection approach. An overview of the approach is shown in Figure 6.2.

Note that the parallelization does not change the precision of the original FastTrack detection since the same FastTrack algorithm is employed by the detection threads. The only concern is that the distribution of access information might change the processing order of memory accesses for the race analysis. However, the distribution of access information does not break the order of race analyses if the accesses already follow the happens-before relation. The order is naturally preserved by the use of the FIFO queues and synchronizations in the application threads. On the other hand, if the accesses are concurrent, they can be analyzed in any order for a detection of race. As an example, consider the access chunks sent to detector thread 0 in Figure 6.2. The access chunk 1 is inserted into the queue before the release operation in application thread 0 and the access chunk 2 can appear in the queue only after the synchronization acquire in application thread 1. Therefore, the order of analyses in detector thread 0 will be preserved as if the analyses are done in the application

threads.

The parallel FastTrack detector has an improved performance and scalability over the original version of FastTrack in a number of ways. First, as accesses to a memory location by multiple threads are handled by one detector, lock operations in the detection can be eliminated. Second, the race detection becomes less dependent on the application threads' execution than in the original FastTrack detector. Even when multiple application threads are inactive (e.g., condition waiting), the detector threads can proceed with the race analysis and utilize any available cores. Third, the detection operation can scale well even for the applications consisting of less number of threads than the number of available cores. Lastly, cache performance will be improved and there will be less data sharing. If there are n detection threads, each detector will be responsible for $1/n$ of the shared address space, and each detector does not share the data structures of vector clock with other detectors.

6.4 Implementation

6.4.1 Instrumentation and Optimization

We have implemented the parallel FastTrack detector based on our previous implementation of FastTrack described in Chapter 5. The original FastTrack detector is implemented for data race detection of C/C++ programs and Intel PIN 2.11 [47] is used for dynamic binary instrumentation of programs. To trace all shared memory accesses, every data access operation is instrumented. A subset of function calls is also instrumented to trace thread creation-join, synchronization, and memory allocation/de-allocation.

In the FastTrack algorithm, to check same epoch accesses, vector clocks should be read from global data structures with a lock operation. In our original FastTrack

implementation, we adopt a per-thread bitmap at each application thread to localize the same epoch checking and to remove the need of lock operations. Thus, only the first access in an epoch needs to be analyzed for a possible race. Even with this enhancement, the lock cost in the FastTrack detector is still considerably high as our experimental results show. Before any access information is fed into the FastTrack detector, we have applied two additional filters to remove unnecessary analyses. First, we filter out stack accesses assuming that there is no stack sharing. Second, a hash filter, which is similar to *"Duplicate filter"* in [78], is applied to remove consecutive assesses to an identical location. The second filter is a small hash-table like array that is indexed with lower bits of memory address and remembers only the last access for each array element. In PIN, a function can be in-lined into instrumented code as long as it is a simple basic block. To enhance the performance of instrumentation, an analysis function, written in a basic block, is used to apply the two filters, and put the access information into a per-thread buffer. When the buffer is full a non-inline function is invoked for data race analyses for the accesses in the buffer.

6.4.2 Parallel FastTrack

The instrumentation routine for every memory access for the parallel FastTrack is identical to the original FastTrack except the buffering of accesses. Instead of the per-thread buffer at each application thread, there is a buffer for each detection thread. That is, for every memory access, the detector thread is chosen based on the address of the access, as shown in Equation 6.1, and the access information is routed to the corresponding buffer. When the buffer is full or there is a synchronization operation, the buffer is inserted into the FIFO queue of the detector thread. For the FastTrack race detection, a tuple of {*thread id, VC (Vector Clock), address, size, IP (Instruction Pointer), access type*} is needed for each memory access. Since {*thread id, VC*} can

be shared by multiple accesses in the same epoch, only the tuple of {*address, size, IP, access type*} is recoded into the buffer.

Upon the detection starts, the number of detector threads is set to a predefined value and each detector is created as a PIN internal thread. The FIFO queue for each detector is a queue of pointers, each pointing to a buffer of access information. Allocating memory for the buffers of all memory accesses will incur substantial overheads both in time and space. To reduce the overheads from memory allocation, the FIFO queue maintains additional queue of used buffers. When a detector finishes data race analyses for accesses in a buffer, the buffer is not freed but sent to the FIFO queue for reuse. When an application thread puts a buffer into the FIFO queue, a new buffer is obtained from the queue if any used buffer is available in the queue.

6.5 Evaluation

In this section, we present the experimental results on the performance and scalability of our parallel FastTrack detection. First, we show the overhead analysis of the FastTrack detection to clarify why the FastTrack detection has high runtime overhead and does not scale well on multi-core machines, and how the parallel version of FastTrack alleviates the overhead. Second, the performance and scalability of the FastTrack and parallel FastTrack detections are compared. All experiments were performed on an 8-core workstation with 2 quad-core 2.27 GHz Intel Xeon running Red Hat Enterprise 6.6 with 12 GB of RAM.

The experiments were performed with 11 benchmark programs, 8 from the PARSEC-2.1 benchmark suite [8] and 3 from popular multi-threaded applications: *FFmpeg* [89] which is a multimedia encoder/decoder, *pbzip2* [30] as a parallel version of *bzip2*, and *hmmsearch* [15] which performs sequence search in bioinformatics. In the following subsections, the number of application threads that carry out the computation is con-

Benchmark Program	Number of accesses (million)			
	All	After stack filter	After hash filter	After the same epoch check
facesim	8,671	7,586	5,096	2,397
ferret	6,797	4,110	2,174	896
fluidanimate	10,184	9,870	4,674	2,171
raytrace	9,208	2,276	865	104
x264	4,776	4,028	2,369	257
canneal	2,714	2,668	903	16
dedup	10,793	10,687	3,938	1,797
streamcluster	19,540	17,720	7,888	4,026
ffmpeg	10,279	9,960	6,408	990
pbzip2	7,567	7,253	4,154	344
hmmsearch	21,912	6,579	3,241	1,308

Table 6.1: Number of Accesses Filtered and Checked in the FastTrack Detection (8 cores with 8 threads).

trollable through a command-line parameter. For the parallel FastTrack detection, the number of detection threads is set to the number of cores for all cases.

6.5.1 Analysis of Race Detection Execution

Table 6.1 shows the number of accesses that are filtered by the two filters and checked by the FastTrack algorithm. "All" column shows the number of instrumentation function calls invoked by memory accesses. "After stack filter" and "After hash filter" columns show the number of accesses after the stack and hash filters, respectively. The last column shows the number of accesses after removing the same epoch accesses with the per-thread bitmap. The last column represents accesses that are fed into the race analysis of the FastTrack algorithm, and we can expect that the lock cost will be proportional to the number in this column for each benchmark application.

Table 6.2 presents the overhead analysis of the FastTrack detection for running on 8 cores with 8 application threads. "PIN" column shows the time spent in PIN

Benchmark Program	Overhead (sec)						
	PIN	Filtering	Same epoch check	Lock	FastTrack	Total	% of lock overhead
facesim	22.4	32.1	67.4	89.6	245.5	457.0	19.6%
ferret	14.4	11.8	18.3	39.1	140.5	224.0	17.4%
fluidanimate	9.2	18.4	43.2	68.8	92.3	232.0	29.7%
raytrace	15.1	19.0	3.3	1.7	3.0	42.0	4.0%
x264	10.3	12.1	13.5	18.8	67.2	122.0	15.4%
canneal	9.4	8.1	8.9	0.2	2.4	29.0	0.6%
dedup	15.3	17.0	39.1	62.2	454.4	588.0	10.6%
streamcluster	9.2	11.8	47.6	125.9	94.5	289.0	43.6%
ffmpeg	25.8	0.0	139.7	64.3	170.2	400.0	16.1%
pbzip2	7.5	12.4	13.6	6.8	77.7	118.0	5.8%
hmmsearch	14.2	30.3	31.5	66.8	131.2	274.0	24.4%
Average							17.0%

Table 6.2: Overheads of the FastTrack Detector (8 cores with 8 threads).

instrumentation function without any analysis code. The execution time of filtering access and saving access information into the per-thread buffer is presented in "Filtering" column. The two columns signify the amount of time that cannot be parallelized by our approach as they should be done in application threads, and the scalability of our parallel detector will be limited by the sum of the two columns. The lock cost, shown in the "Lock" column, is extracted from the runs with locking and unlocking operations, but with no processing on vector clocks. The measure may not be very accurate due to the possible lock contention. However, it will still show a basic idea of how significant the lock overhead is. The overhead of locking is 17%, on average and it is up to 44% of the total execution time for *steamcluster* benchmark program. With the number of application threads equals to the number of cores, the average lock overheads on the systems of 2, 4, and 6 cores are 14.1%, 14.7%, and 15.2%, respectively. These overheads follow the similar pattern as the overheads shown in the table for an 8 cores system, and the results are omitted for the simplicity of the discussion.

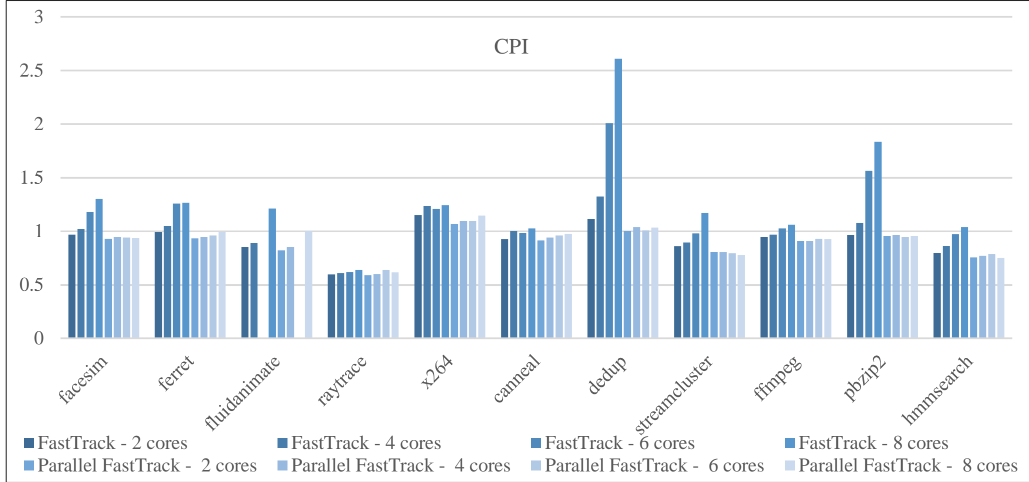


Figure 6.3: CPI Measures of the Race Detection Programs. For each benchmark, the first four columns are the CPIs of the FastTrack runs on 2, 4, 6, and 8 cores, and the second four columns are the CPIs of the parallel FastTrack runs on the similar machine configurations.

In Figure 6.3, we present the CPI (Cycles per Instruction) measures from the FastTrack and our parallel FastTrack detector runs. The CPI measures indirectly show (1) the performance degradation due to pipeline stalls by frequent lock operations, and (2) the cache performance as cache misses and invalidations can lead to memory stalls. The CPIs are measured with Intel Amplifier-XE [28]. For each benchmark program in Figure 6.3, the first four columns represent the CPIs of the FastTrack detector running on machines of 2, 4, 6, and 8 cores. The second four columns indicate the CPIs of the parallel FastTrack detector on the same machine configurations. For all cases, the number of application threads is equal to the number of cores. Since the benchmark program *fluidanimate* can only be configured with 2^n threads, the performance measures of *fluidanimate* with 6 application threads are not reported throughout this chapter.

The results in Figure 6.3 suggest that the CPIs of the FastTrack detection are higher than those of the parallel FastTrack detection. It is notable that, in the FastTrack detection, the CPI increases as we increase the number of application threads

Benchmark Program	2 cores 2 application threads			4 cores 4 application threads			6 cores 6 application threads			8 cores 8 application threads		
	Application	FastTrack	Parallel	Application	FastTrack	Parallel	Application	FastTrack	Parallel	Application	FastTrack	Parallel
facesim	77%	76%	92%	54%	55%	87%	39%	46%	78%	33%	41%	72%
ferret	88%	85%	88%	85%	79%	85%	81%	53%	81%	77%	40%	75%
fluidanimate	92%	89%	87%	86%	81%	87%	N/A	N/A	N/A	69%	73%	77%
raytrace	96%	89%	84%	89%	77%	73%	84%	67%	63%	83%	60%	56%
x264	87%	94%	87%	86%	90%	81%	81%	82%	71%	73%	66%	60%
canneal	89%	84%	79%	78%	70%	64%	66%	56%	51%	62%	51%	44%
dedup	77%	91%	92%	59%	83%	91%	37%	62%	87%	37%	72%	85%
streamcluster	96%	95%	92%	95%	87%	91%	91%	68%	90%	76%	77%	86%
ffmpeg	62%	72%	89%	46%	48%	88%	38%	36%	79%	28%	29%	72%
pbzip2	97%	96%	87%	96%	94%	90%	96%	93%	88%	94%	91%	85%
hmmsearch	99%	87%	84%	98%	67%	91%	99%	55%	91%	98%	46%	89%
Average	87%	87%	87%	79%	75%	85%	71%	62%	78%	66%	59%	73%

Table 6.3: Comparison of CPU Core Utilization.

and the number of cores. Note that, in the FastTrack detection, the vector clocks are organized in a global data structure and shared among all running threads. Locking operations, which need to flush the CPU pipeline, can also lead to a negative impact on the CPI. The increased CPI not only hurts the performance of race detection, but makes the detection not scalable. For the two programs, *dedup* and *pbzip2*, we can expect that the performance of the FastTrack detection would not be improved even with additional cores. On the contrary, the CPIs of the parallel FastTrack detector are stable as we change the number of cores. The detection thread performs data race analyses for an independent range of the address space and they don't share vector clocks with other detectors without lock operations.

In Table 6.3, the CPU core utilizations, measured with Intel Amplifier-XE [28], are reported. For each machine configuration, the experiments include running benchmark applications alone, benchmark applications with the FastTrack detection and with the parallel FastTrack detection. In general, we can observe that, when the applications cannot fully utilize the cores, adding the processing of the FastTrack de-

tection would not improve CPU utilization. On the other hand, the core utilization is improved under the parallel detection regardless of the executions of application threads. For instance, for *facesim*, *ferret*, and *ffmpeg* on an 8 core machine, the parallel detection nearly doubles the CPU core utilization of the FastTrack detection.

Ideally, the execution of the parallel FastTrack detector should utilize 100% of cores. There are largely two reasons why the parallel detection does not fully utilize the cores. First, application threads may not be fast enough in generating access information into the queues to make the detection threads busy. In other words, the queues become empty and the detection threads become idle. In the cases of *raytrace* and *canneal*, the applications use a single thread to process input data during the initialization of the programs. In our implementation of race detection, we disable race detection when only one thread is active. Hence, during the initialization process, all detection threads are idle. Also, a large amount of stack accesses can cause the detection threads idle since all the stack accesses are filtered out by the instrumentation code of the application threads.

The other reason is due to the serialization between application threads and the detection threads. To reduce the overhead, access information from an application thread is saved in a buffer (the size of 100k access entries in the current implementation) and is transferred to a detector when the buffer is full. However, when a synchronization event occurs during application execution, the buffer is moved into the queue immediately. Thus, frequent synchronization events in application threads can serialize the FIFO queue operations with detection threads.

6.5.2 Performance and Scalability

The performance results for the executions of the parallel and FastTrack detectors are compared and shown in Table 6.4. The experiments were performed on the ma-

Benchmark Program	2 cores (sec)				4 cores (sec)				6 cores (sec)				8 cores (sec)			
	Application	FastTrack	Parallel	Speedup	Application	FastTrack	Parallel	Speedup	Application	FastTrack	Parallel	Speedup	Application	FastTrack	Parallel	Speedup
facesim	5.5	718	461	1.6	3.9	519	251	2.1	3.4	484	194	2.5	3.2	457	154	3.0
ferret	5.4	304	247	1.2	2.9	192	133	1.4	2.1	228	102	2.2	1.6	224	83	2.7
fluidanimate	6.5	313	254	1.2	3.5	220	161	1.4	--	--	--	--	2.2	232	155	1.5
raytrace	9.4	105	104	1.0	5.2	63	62	1.0	3.6	49	54	0.9	2.9	42	42	1.0
x264	3.4	239	224	1.1	1.9	145	133	1.1	1.3	125	117	1.1	1.1	122	98	1.2
canneal	8.1	60	61	1.0	4.8	39	40	1.0	3.8	33	36	0.9	3.2	29	31	0.9
dedup	8.7	719	562	1.3	5.8	482	298	1.6	6.4	671	208	3.2	4.8	588	159	3.7
streamcluster	4.3	632	431	1.5	2.3	372	238	1.6	1.3	392	174	2.3	1.0	289	143	2.0
ffmpeg	6.2	563	379	1.5	4.4	434	198	2.2	3.9	407	159	2.6	3.7	400	127	3.1
pbzip2	5.7	219	208	1.1	3.1	128	109	1.2	2.0	128	77	1.7	1.6	118	59	2.0
hmmsearch	5.8	443	348	1.3	2.9	309	178	1.7	2.0	285	132	2.2	1.5	274	92	3.0
Average				1.2				1.5				1.9				2.2

Table 6.4: Performance Comparisons of the FastTrack and the Parallel FastTrack Detections. The number of applications threads and detection threads are set to the number of cores.

chines of 2 to 8 cores and the number of application threads is equal to the number of cores. In addition to the execution times, the speedup factor of the parallel detection over the FastTrack detection is included in the table.

Overall, the parallel detector performs much better than the FastTrack detector. This performance improvement is attributed to three factors: (1) the overhead of lock operations in race analyses, as shown in Table 6.2, is eliminated, (2) the parallel detection better utilizes multiple cores as presented in Table 6.3, and (3) the localized data structure in detection threads reduces global data sharing and improves CPI with less pipeline stalls, as shown in Figure 6.3. In addition, the speed-up factors of Table 6.4 (i.e., the ratio of execution time of the FastTrack detector to that of the parallel detector) increase with the number of cores. This is caused by the enhancements in core utilizations and CPIs when the parallel detection is executed on multi-core machines.

While the parallel detector achieves a speed-up factor of 2.2 on average over the FastTrack detection on an 8 core machine, some programs, such as *raytrace, canneal*

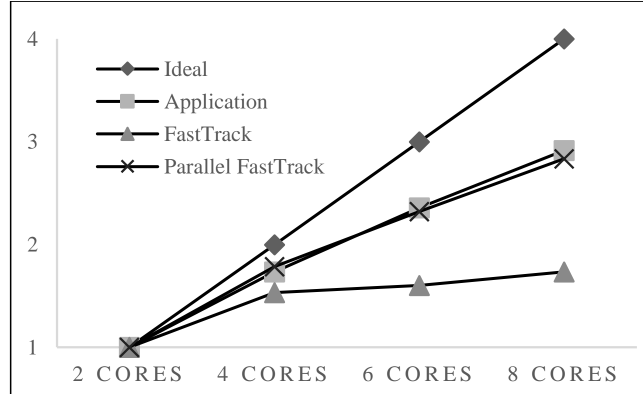


Figure 6.4: Scaling Factors of Race Detectors Where the Number of Threads Equal to the Number of Cores.

in our experiments, don't gain any speed-up with the parallel detection. As described in the previous subsection, the two programs run with a single application thread for a long period of time, and there are relatively small amount of accesses that must be checked by the FastTrack algorithm (as shown in the last column of Table 6.1).

Another view for the performance results of Table 6.4 is depicted in Figure 6.4 where the speed-up factors are drawn from 2 cores to 8 cores for application alone, the FastTrack detection, and the parallel detection. For comparison, the ideal speedup is added in the figure (e.g., 4x speed-up when 4x more application threads run). The figure suggests that the parallel FastTrack detector can scale up when we increase the number of cores in the systems. On the other hand, the FastTrack detector does not scale well due to the reasons explained previously.

In Table 6.5, we present the performance of parallel race detector when additional cores are available. Only two application threads are used for all the experiments in Table 6.5. As we increase the number of cores from 2 to 8, 6 additional cores can be used to run the detection threads in the parallel race detector. Note that the executions of application itself and the FastTrack detection obviously do not change since the number of application threads is fixed. On the other hand, the parallel FastTrack detector, that utilizes all additional 6 cores, produces an average speed-up

Benchmark Program	Application 2 cores (sec)	FastTrack 2 cores (sec)	Parallel FastTrack (sec) # of detectors = # of cores			
			2 cores	4 cores	6 cores	8 cores
facesim	5.5	718	461	249	194	156
ferret	5.4	304	247	129	97	79
fluidanimate	6.5	313	254	139	125	112
raytrace	9.4	105	104	83	97	83
x264	3.4	239	224	127	100	81
canneal	8.1	60	61	44	48	43
dedup	8.7	719	562	291	197	150
streamcluster	4.3	632	431	227	159	118
ffmpeg	6.2	563	379	197	176	142
pbzip2	5.7	219	208	108	88	75
hmmsearch	5.8	443	348	184	204	165

Table 6.5: Speedups with Additional Cores. For all cases, two application threads are used.

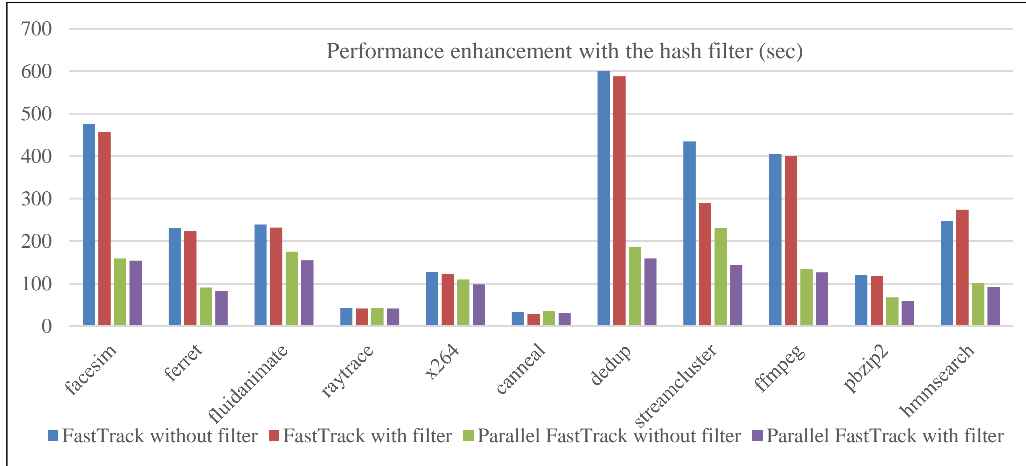


Figure 6.5: Performance Comparison with/without the Hash Filter. On average, the use of the hash filter improves the detection performance by 5% and 10% for the FastTrack and parallel detector, respectively.

of 3.3 when the performance of the parallel detection and the FastTrack detection is compared. This speedup is due to the effective execution of parallel detection threads that is separated from the application execution.

Figure 6.5 shows the performance enhancement with the hash filter. On average, the hash filter brings about 5% and 10% performance improvements for the FastTrack and parallel FastTrack detectors. In our current implementation, each thread

Benchmark Program	2 core 2 application threads (MB)			4 core 4 application threads (MB)			6 core 6 application threads (MB)			8 core 8 application threads (MB)		
	Application	FastTrack	Parallel	Application	FastTrack	Parallel	Application	FastTrack	Parallel	Application	FastTrack	Parallel
facesim	417	5137	5950	576	5450	6682	730	5600	7613	888	5810	7756
ferret	759	7011	5638	1365	8091	6224	1971	8900	7768	2577	10032	9506
fluidanimate	267	1362	2253	290	1440	2408	--	--	--	338	1605	3088
raytrace	80	741	1142	101	811	1746	121	878	1870	142	949	2651
x264	135	4282	4531	165	4092	7757	195	6530	11247	225	8292	13736
canneal	207	861	1380	359	1085	1785	510	1319	2219	662	1572	2616
dedup	2717	8265	7018	2709	8823	8069	3026	9175	8512	3371	9829	9409
streamcluster	110	668	1182	131	692	1424	151	761	1696	172	821	2037
ffmpeg	147	1519	2317	229	1746	2697	312	1968	3330	395	2239	3778
pbzip2	217	3914	4318	380	4497	4781	557	5078	5146	726	3912	6114
hmmsearch	161	599	1047	312	806	1406	464	1006	1676	615	1206	2529
Average	474	3124	3343	601	3412	4089	804	4122	5108	919	4206	5747

Table 6.6: Maximal Memory Usage of FastTrack and Parallel FastTrack Race Detections.

maintains hash filters of 512 and 256 entries for read and write accesses, respectively. We found out that, while an increase of the hash filter, more accesses can be removed from the checking of the same epoch access. However, there were performance penalties in cache accesses as the arrays of the hash filter are randomly accessed. There are significant performance enhancements for certain benchmark programs. For instance, in *streamcluster*, the performance gain due to the hash filter is 33% for the FastTrack detector and 38% for the parallel detector. This application frequently spins on flag variables and generates a substantial amount of accesses to few memory locations during short intervals. Thus, the hash filter can effective remove the duplicated accesses and improve the performance greatly. The use of the hash filter in the parallel detection can not only save redundant race analysis but also avoid the transfer of access information through the FIFO queue.

In Table 6.6, we present the maximum memory used during the executions of the application, the FastTrack detector, and the parallel detector. For the executions on an 8 cores machine (there are 8 detection threads), the parallel detector uses on

average 1.37 times more memory than the FastTrack detector. As we increase the number of detection threads, it is expected that additional memory is consumed by the buffers and queues to distribute access information from application threads to detection threads.

6.6 Related Work

Most related works on static and dynamic data race detections and their optimizations have been mentioned throughout this chapter and Chapter 5. It is worthy noting that a different approach to parallelize race detection is reported in [101] in which thread execution is time-sliced and pipelined. To incorporate this approach, application threads must be organized following the *uniparallelism* multithreading paradigm [97].

A variety of hybrid race detectors [64, 70, 81, 105] have been proposed in which Eraser's Lockset algorithm is combined with vector clock based algorithms. The approaches take advantage of vector clock based algorithm to reduce false alarms, and preserve the performance and/or the detection coverage of lockset based algorithm. RaceMob [36] adapts a crowdsourcing approach that can be considered as a hybrid race detection of static and dynamic approaches. In RaceMob, a hive is maintained to manage a set of candidate races produced by a static race detector. A candidate race is distributed to multiple users through an on-line community. The users validate the race with schedule steering to explore more execution ordering and each user is assigned with a different timeout value. RaceMob increases a statistical confidence on the validation results as more users participate.

For programs in specific domains, race detection algorithms should be revised to accommodate the specific execution environment. In a GPU program execution model, a large number of threads are scheduled with the same instructions on different

data sets, and the threads are synchronized with barriers. GRace [107] exploits the GPU execution model and uses static analysis to prune out memory accesses that cannot be in data races. It then detects data races for the unresolved memory accesses with dynamic analysis. Another specific domain of race detection is for Android applications which are event-driven programs comprising asynchronous tasks. In [25, 48], the happens-before relation is defined and inferred for the events to detect races in Android applications. A web application is another type of event-driven program in which each event is processed in a single threaded event handler. Raychev, et.al, has proposed an algorithm to effectively discover atomic-executed event handlers to avoid false alarms and to reduce the overhead of vector clock based race detector [74]. In [69], a happens-before relation is defined for JavaScript and HTML to detect races for web applications.

6.7 Chapter Conclusions

In this chapter, we present an efficient parallel dynamic data race detection on multi-core systems. We have analyzed the overhead of the FastTrack detector on the execution of multi-core machines. Based on the overhead analysis, we have devised a parallelization method to increase the performance and scalability of dynamic data race detection. In our parallel race detector, data race analyses are decoupled from execution of application threads, and each detection thread performs race analyses independently of other detection threads. Our experimental results show that the approach is viable for the race detection on multi-core environment.

Chapter 7

CONCLUSIONS

7.1 Summary

This thesis has presented a dynamic analysis framework for embedded software. As discussed in Chapter 3, the non-deterministic nature of multi-threaded program execution makes dynamic program analysis a challenging task. In addition, observing program execution may not even feasible due to probe effect caused by instrumentation overhead. Hence, we propose a deterministic replay that enables efficient and effective dynamic analysis of embedded software. We also present probe effect analysis and data race detection tools which are essential for building the deterministic replay framework.

In Chapter 3, we present the deterministic replay. We record an execution of program that exhibits a failure or performance anomaly. The recorded execution can be reproduced during replay as if there is no instrumentation overhead or the program is totally deterministic. Hence, any dynamic analysis (e.g., debugging, profiling) can be safely applied during replay. We keep the recording overhead minimal to ensure that the recording operation itself does not incur probe effect.

In the research works of dynamic program analysis, one of the most important metrics is instrumentation overhead since researchers are aware of potential probe effect caused by the instrumentation overhead. As for our deterministic replay, although a program execution is reproducible during replay, the recording operation (even if the overhead is minimal) might have changed the program execution. In Chapter 4, we propose a simulation-based analysis that decides whether a given in-

strumentation has changed the program execution or not. Since the analysis considers all factors that can affect a program execution including O/S and I/O events, any probe effect can be precisely detected.

In our thread execution model from Chapters 3 and 4, a program execution is represented as a partial order of synchronization and I/O events. While the model makes the program analyses efficient, one drawback is that the execution model may not work correctly in the presence of data races in the program execution. Hence, in Chapter 5 we present in-depth discussion of data race detection and propose an efficient data race detection algorithm for C/C++ program. The run-time and memory overheads have substantially been reduced by adapting large detection granularity considering memory access patterns.

As multi-core machines are readily available nowadays, our race detector can be more efficient and scalable by exploiting parallel executions. In Chapter 6, we start with the overhead and scalability analyses of race detection on multi-core machines. Based on the analyses, we propose a parallelized data race detection. Our idea is to decouple race detection analysis from application execution and localize the analyses for additional detection threads. As our experimental results show, our parallel FastTrack detector achieves a speedup of 2.2 times, on average, over the original FastTrack detector.

7.2 Future Work

The research works can be extended in a number of ways. For the deterministic replay, some implementation issues are remained. Currently, we provide wrapper functions only for the *pthread* library. Whenever there is a need to support other thread primitives, corresponding wrapper functions should be implemented. This rework can be avoided if the record/replay is implemented in lower level synchronization

primitives (e.g., in *glibc*).

In the simulation-based analysis, we only consider single core machines with priority-based preemptive scheduling. As a further step, our analysis can be extended for multi-core systems in which thread migration and multiprocessor scheduling should be considered. Also it would be interesting to attempt a hardware-assisted online detection mechanism for potential probe effect. Then, remedy actions, such as synchronization operations, can be inserted to ensure deterministic event ordering.

The sharing algorithm for dynamic granularity can be refined to remove possible false alarms. In the current implementation, read and write vector clocks are maintained separately. The decision of sharing read vector clocks can be guided by the status of write vector clocks. We also plan to enhance the vector clock state machine to accommodate access behavior after the second epoch so that the detection granularity can be changed more dynamically.

Our work for data race detection can be extended for other dynamic analyses such as atomicity and order violation detections. Since those analyses are also based on tracing memory accesses, our dynamic granularity and parallelizing race detection will be easily adapted to detect other concurrency bugs.

REFERENCES

[1] S. Adve. Data races are evil with no exceptions: Technical perspective. *Commun. ACM*, 53(11):84–84, Nov. 2010.

[2] B. Andersson, S. Blair-Chappell, and R. Mueller-Albrecht. Intel® debugger for linux*. 2012.

[3] P. Arafa, H. Kashif, and S. Fischmeister. Dime: Time-aware dynamic binary instrumentation using rate-based resource allocation. In *Proceedings of the Eleventh ACM International Conference on Embedded Software*, EMSOFT '13, pages 25:1–25:10, Piscataway, NJ, USA, 2013. IEEE Press.

[4] Arizona State University. CSE438-Embedded Systems Programming. http://rts.lab.asu.edu/web_438/CSE438_Main_page.htm.

[5] ARM Ltd. RealView® Profiler. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0412a/html/preface.html`.

[6] T. Bergan, J. Devietti, N. Hunt, and L. Ceze. The Deterministic Execution Hammer: How Well Does it Actually Pound Nails? In *Workshop on Determinism and Correctness in Parallel Programming w/ International Conference on Architectural Support for Programming Languages and Operating Systems (WoDet w/ ASPLOS)*, 3 2011.

[7] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments*, VEE '06, pages 154–163, New York, NY, USA, 2006. ACM.

[8] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton, NJ, USA, 2011. AAI3445564.

[9] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: Proportional detection of data races. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 255–268, New York, NY, USA, 2010. ACM.

[10] J.-D. Choi and S. L. Min. Race frontier: Reproducing data races in parallel-program debugging. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '91, pages 145–154, New York, NY, USA, 1991. ACM.

[11] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant. Parrot: A practical runtime for deterministic, stable, and reliable threads. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 388–405, New York, NY, USA, 2013. ACM.

[12] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 337–351, New York, NY, USA, 2011. ACM.

[13] L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H.-J. Boehm. Ifrit: Interference-free regions for dynamic data-race detection. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 467–484, New York, NY, USA, 2012. ACM.

[14] C. Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, Aug. 1991.

[15] R. D. Finn, J. Clements, and S. R. Eddy. HMMER web server: interactive sequence similarity searching. *Nucleic Acids Research*, 39(Web-Server-Issue):29–37, 2011.

[16] S. Fischmeister and P. Lam. Time-aware instrumentation of real-time programs. *IEEE Trans. Industrial Informatics*, 6(4):652–663, 2010.

[17] C. Flanagan and S. N. Freund. Type-based race detection for java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 219–232, New York, NY, USA, 2000. ACM.

[18] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 256–267, New York, NY, USA, 2004. ACM.

[19] C. Flanagan and S. N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 121–133, New York, NY, USA, 2009. ACM.

[20] Free Software Foundation, Inc. Debugging with gdb. `https://sourceware.org/gdb/onlinedocs/gdb/`.

[21] N. Froyd, J. Mellor-Crummey, and R. Fowler. Low-overhead call path profiling of unmodified, optimized code. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 81–90, New York, NY, USA, 2005. ACM.

[22] J. Gait. A probe effect in concurrent programs. *Softw. Pract. Exper.*, 16(3):225–233, Mar. 1986.

[23] J. L. Greathouse, Z. Ma, M. I. Frank, R. Peri, and T. Austin. Demand-driven software race detection using hardware performance counters. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 165–176, New York, NY, USA, 2011. ACM.

[24] D. R. Hower, P. Montesinos, L. Ceze, M. D. Hill, and J. Torrellas. Two hardware-based approaches for deterministic multiprocessor replay. *Commun. ACM*, 52(6):93–100, June 2009.

[25] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn. Race detection for event-driven mobile applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 326–336, New York, NY, USA, 2014. ACM.

[26] R. Huang, E. Halberg, and G. E. Suh. Non-race concurrency bug detection through order-sensitive critical sections. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 655–666, New York, NY, USA, 2013. ACM.

[27] Intel Corporation. Intel® Inspector XE. `https://software.intel.com/en-us/intel-inspector-xe`.

[28] Intel Corporation. Intel® Vtune™ Amplifier. `http://software.intel.com/en-us/intel-vtune-amplifier-xe`.

[29] Intel Corporation. Pin - A Dynamic Binary Instrumentation Tool. `http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`.

[30] Jeff Gilchrist. pbzip2. `http://compression.ca/pbzip2/`.

[31] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 389–400, New York, NY, USA, 2011. ACM.

[32] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu. Automated concurrency-bug fixing. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 221–236, Berkeley, CA, USA, 2012. USENIX Association.

[33] Joab Jackson. Facebook IPO Glitch. `http://www.cio.com.au/article/425234/nasdaq_facebook_glitch_came_from_race_conditions/`.

[34] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta. Fast and accurate static data-race detection for concurrent programs. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 226–239, Berlin, Heidelberg, 2007. Springer-Verlag.

[35] B. Kasikci, C. Zamfir, and G. Candea. Data races vs. data race bugs: Telling the difference with portend. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 185–198, New York, NY, USA, 2012. ACM.

[36] B. Kasikci, C. Zamfir, and G. Candea. Racemob: Crowdsourced data race detection. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 406–422, New York, NY, USA, 2013. ACM.

[37] M. Kim, H. Kim, and C.-K. Luk. Sd3: A scalable approach to dynamic data-dependence profiling. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 535–546, Washington, DC, USA, 2010. IEEE Computer Society.

[38] D. Kranzlmüller, R. Reussner, and C. Schaubschläger. Monitor overhead measurement of mpi applications with skampi. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 43–50, London, UK, UK, 1999. Springer-Verlag.

[39] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[40] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct. 1997.

[41] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36(4):471–482, Apr. 1987.

[42] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: Efficient online multiprocessor replayvia speculation and external determinism. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 77–90, New York, NY, USA, 2010. ACM.

[43] Y.-H. Lee, Y. W. Song, R. Girme, S. Zaveri, and Y. Chen. Replay debugging for multi-threaded embedded software. In *Proceedings of the 2010 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, EUC '10, pages 15–22, Washington, DC, USA, 2010. IEEE Computer Society.

[44] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, July 1993.

[45] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: Efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 327–336, New York, NY, USA, 2011. ACM.

[46] K. Lu, X. Zhou, T. Bergan, and X. Wang. Efficient deterministic multithreading without global barriers. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 287–300, New York, NY, USA, 2014. ACM.

[47] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[48] P. Maiya, A. Kanade, and R. Majumdar. Race detection for android applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 316–325, New York, NY, USA, 2014. ACM.

[49] A. D. Malony, D. A. Reed, and H. A. G. Wijshoff. Performance measurement intrusion and perturbation analysis. *IEEE Trans. Parallel Distrib. Syst.*, 3(4):433–450, July 1992.

[50] D. Marino, M. Musuvathi, and S. Narayanasamy. Literace: Effective sampling for lightweight data-race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 134–143, New York, NY, USA, 2009. ACM.

[51] T. Merrifield, J. Devietti, and J. Eriksson. High-performance determinism with total store order consistency. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 31:1–31:13, New York, NY, USA, 2015. ACM.

[52] P. Montesinos, M. Hicks, S. T. King, and J. 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*, ASPLOS XIV, pages 73–84, New York, NY, USA, 2009. ACM.

[53] C. Moreno, S. Fischmeister, and M. A. Hasan. Non-intrusive program tracing and debugging of deployed embedded systems through side-channel analysis. In *Proceedings of the 14th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '13, pages 77–88, New York, NY, USA, 2013. ACM.

[54] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, and R. Peri. Shadow profiling: Hiding instrumentation costs with parallelism. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 198–208, Washington, DC, USA, 2007. IEEE Computer Society.

[55] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association.

[56] A. Muzahid, N. Otsuki, and J. Torrellas. Atomtracker: A comprehensive approach to atomic region inference and violation detection. In *Proceedings of*

the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '43, pages 287–297, Washington, DC, USA, 2010. IEEE Computer Society.

[57] A. Muzahid, D. Suárez, S. Qi, and J. Torrellas. Sigrace: Signature-based data race detection. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 337–348, New York, NY, USA, 2009. ACM.

[58] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 308–319, New York, NY, USA, 2006. ACM.

[59] A. Nair, K. Shankar, and R. Lysecky. Efficient hardware-based nonintrusive dynamic application profiling. *ACM Trans. Embed. Comput. Syst.*, 10(3):32:1–32:22, May 2011.

[60] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 22–31, New York, NY, USA, 2007. ACM.

[61] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[62] R. H. Netzer and B. P. Miller. On the complexity of event ordering for shared-memory parallel program executions. In *In Proceedings of the 1990 International Conference on Parallel Processing*, pages 93–97, 1990.

[63] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, Mar. 1992.

[64] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 167–178, New York, NY, USA, 2003. ACM.

[65] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 97–108, New York, NY, USA, 2009. ACM.

[66] M. Olszewski, Q. Zhao, D. Koh, J. Ansel, and S. Amarasinghe. Aikido: Accelerating shared data dynamic analyses. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 173–184, New York, NY, USA, 2012. ACM.

[67] S. Park, S. Lu, and Y. Zhou. Ctrigger: Exposing atomicity violation bugs from their hiding places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 25–36, New York, NY, USA, 2009. ACM.

[68] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. Pres: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 177–192, New York, NY, USA, 2009. ACM.

[69] B. Petrov, M. Vechev, M. Sridharan, and J. Dolby. Race detection for web applications. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 251–262, New York, NY, USA, 2012. ACM.

[70] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 179–190, New York, NY, USA, 2003. ACM.

[71] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: Practical static race detection for c. *ACM Trans. Program. Lang. Syst.*, 33(1):3:1–3:55, Jan. 2011.

[72] M. Prvulovic. Cord: cost-effective (and nearly overhead-free) order-recording and data race detection. In *Proceedings of the 12th IEEE Symp. on High-performance Computer, Architecture*, HPCA, pages 232–243. IEEE Computer Society, 2006.

[73] P. Ratanaworabhan, M. Burtscher, D. Kirovski, B. Zorn, R. Nagpal, and K. Pattabiraman. Detecting and tolerating asymmetric races. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 173–184, New York, NY, USA, 2009. ACM.

[74] V. Raychev, M. Vechev, and M. Sridharan. Effective race detection for event-driven programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 151–166, New York, NY, USA, 2013. ACM.

[75] Rogue Wave Software, Inc. TotalView Debugger. `http://www.roguewave.com/products-services/totalview`.

[76] M. Ronsse, M. Christiaens, and K. De Bosschere. Debugging shared memory parallel programs using record/replay. *Future Gener. Comput. Syst.*, 19(5):679–687, July 2003.

[77] M. Ronsse and K. De Bosschere. Recplay: A fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, May 1999.

[78] P. Sack, B. E. Bliss, Z. Ma, P. Petersen, and J. Torrellas. Accurate and efficient filtering for the intel thread checker race detector. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, pages 34–41, New York, NY, USA, 2006. ACM.

[79] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, Nov. 1997.

[80] K. Sen. Race directed random testing of concurrent programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 11–21, New York, NY, USA, 2008. ACM.

[81] K. Serebryany and T. Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM.

[82] J. Simsa, R. Bryant, and G. Gibson. dbug: Systematic evaluation of distributed systems. In *Proceedings of the 5th International Conference on Systems Software Verification*, SSV'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

[83] Y. W. Song and Y.-H. Lee. Dynamic analysis of embedded software using execution replay. In *Proceedings of the IEEE 17th International Symposium on Object/Component-Oriented Real-Time Distributed Computing*, ISORC '14, pages 166–173, Washington, DC, USA, 2014. IEEE Computer Society.

[84] Y. W. Song and Y.-H. Lee. Efficient data race detection for c/c++ programs using dynamic granularity. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 679–688, Washington, DC, USA, 2014. IEEE Computer Society.

[85] Y. W. Song and Y.-H. Lee. On the existence of probe effect in multi-threaded embedded programs. In *Proceedings of the 14th International Conference on Embedded Software*, EMSOFT '14, pages 18:1–18:9, New York, NY, USA, 2014. ACM.

[86] Y. W. Song and Y.-H. Lee. Parallel data race detection on multi-core systems. 2015, Manuscript submitted for publication.

[87] N. Telkar, K. S. Chatha, Y.-H. Lee, G. Gannod, and E. Wong. A Technique for Verification of Race Conditions in Real Time Systems. In *the 2nd International Workshop on Software Verification and Validation*, 2004.

[88] The Eclipse Foundation. Eclipse. `https://eclipse.org/`.

[89] The FFmpeg developers. FFmpeg. `http://www.ffmpeg.org/`.

[90] The LTP project developers. Linux Test Project. `https://github.com/linux-test-project/ltp`.

[91] The Qt Company. Qt Project. `http://qt-project.org/`.

[92] U.S.-Canada Power System Outage Task Force. Final report on the august 14th blackout in the united states and canada: Causes and recommendations. *Technical report, Department of Energy*, 2004.

[93] Valgrind™ Developers. Callgrind: a call-graph generating cache and branch prediction profiler. `http://valgrind.org/docs/manual/cl-manual.html`.

[94] Valgrind™ Developers. DRD: a thread error detector. `http://valgrind.org/docs/manual/drd-manual.html`.

[95] Valgrind™ Developers. Helgrind: a thread error detector. `http://valgrind.org/docs/manual/hg-manual.html`.

[96] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Detecting and surviving data races using complementary schedules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 369–384, New York, NY, USA, 2011. ACM.

[97] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. Doubleplay: Parallelizing sequential logging and replay. *ACM Trans. Comput. Syst.*, 30(1):3:1–3:24, Feb. 2012.

[98] J. W. Voung, R. Jhala, and S. Lerner. Relay: Static race detection on millions of lines of code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 205–214, New York, NY, USA, 2007. ACM.

[99] C. Wang and M. Ganai. Predicting concurrency failures in the generalized execution traces of x86 executables. In *Proceedings of the Second International Conference on Runtime Verification*, RV'11, pages 4–18, Berlin, Heidelberg, 2012. Springer-Verlag.

[100] C. Wang, S. Kundu, R. Limaye, M. Ganai, and A. Gupta. Symbolic predictive analysis for concurrent programs. *Form. Asp. Comput.*, 23(6):781–805, Nov. 2011.

[101] B. Wester, D. Devecsery, P. M. Chen, J. Flinn, and S. Narayanasamy. Parallelizing data race detection. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 27–38, New York, NY, USA, 2013. ACM.

[102] Wind River Systems, Inc. VxWorks. `http://www.windriver.com/products/vxworks/`.

[103] F. Wolf, A. D. Malony, S. Shende, and A. Morris. Trace-based parallel performance overhead compensation. In *Proceedings of the First International Conference on High Performance Computing and Communications*, HPCC'05, pages 617–628, Berlin, Heidelberg, 2005. Springer-Verlag.

[104] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, pages 122–135, New York, NY, USA, 2003. ACM.

[105] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 221–234, New York, NY, USA, 2005. ACM.

[106] Q. Zhao, I. Cutcutache, and W.-F. Wong. Pipa: Pipelined profiling and analysis on multicore systems. *ACM Trans. Archit. Code Optim.*, 7(3):13:1–13:29, Dec. 2010.

[107] M. Zheng, V. T. Ravi, F. Qin, and G. Agrawal. Grace: A low-overhead mechanism for detecting data races in gpu programs. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 135–146, New York, NY, USA, 2011. ACM.

[108] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 263–271, New York, NY, USA, 2006. ACM.