UNIVERSITY OF CALIFORNIA, SAN DIEGO

Deterministic Replay using Processor Support and Its Applications

A dissertation submitted in partial satisfaction of the requirements for the degree

Doctor of Philosophy

in

Computer Science

by

Satish Narayanasamy

Committee in charge:

Brad Calder, Chair
Fred Chong
Paul Chou
Dean Tullsen
Geoff Voelker

2007

The dissertation of Satish Narayanasamy is approved, and it is acceptable in quality and form for publication on microfilm:

_____

_____

_____

_____

_____

_____

Chair

University of California, San Diego

2007

DEDICATION

For My Parents

EPIGRAPH

Debugging is twice as hard as writing the program, so if you write the program as cleverly as you can, by definition, you won't be clever enough to debug it.

– *Kernighan's Law*

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I owe this thesis to my advisor Brad Calder. Right from picking me up at the airport when I first landed in this foreign country till my defense talk, he has guided me, inspired me, goaded me and has seen me grow as a researcher. Thank you!

I must thank Dean Tullsen, Geoff Voelker and George Varghese for their help and advise at various times during my graduate life. Thanks to Steve Swanson and Michael Taylor for their help during my interview process. Thanks to Paul Chau and Fred Chong for serving on my thesis committee and for providing useful feedback. I must also thank my undergraduate advisor Ranjani Parthasarathy for introducing me to computer architecture and encouraging me to take on open-ended projects in her courses. Without her influence I doubt if I would have pursued a PhD.

Cristiano and Gilles contributed a lot to this thesis. I would also like to thank all of my labmates for attending my practice talks, discussions, administering the clusters, dinners, movies, foosball, tennis, ping-pong, jokes, perpectives, yo-yo, and more.

Thanks to all my friends and family for all their support and tolerating my insane work schedule. I thank Ritu for her love, companionship and being there to share the ups and downs. Finally, I could not have reached this point in life without the continuing support and selfless love of my parents. I dedicate this thesis to them.

BugNet presented in Chapter III was the result of collaboration with Gilles Pokam at Intel and Brad Calder at the University of California San Diego. I thank my co-authors for allowing me to present the results of our collaboration in my dissertation.

Strata presented in Chapter IV and software implementation of BugNet

presented in Chapter III was the result of collaboration with Cristiano Pereira and Brad Calder at the University of California San Diego. I thank my co-authors for allowing me to present the results of our collaboration in my dissertation.

Replay-based automatic data race classification presented in Chapter V was the result of collaboration with Zhenghao Wang, Jordan Tigani, Andrew Edwards at Microsoft and Brad Calder at the University of California San Diego.

Chapter III contains material that appears in "BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging", in *32nd Annual International Symposium on Computer Architecture*, Satish Narayanasamy, Gilles Pokam, Brad Calder. The dissertation author was the primary investigator and author of this paper.

Chapter III contains material that appears in "Software Profiling for Deterministic Replay Debugging of User Code", in *5th International Conference on Software Methodologies, Tools and Techniques (SoMET)*, Satish Narayanasamy, Cristiano Pereira, Brad Calder. The dissertation author was the primary investigator and author of this paper.

Chapter IV contains material that appears in "Recording shared memory dependencies using strata", in *Proceedings of the 12th international confer-*

ence on Architectural support for programming languages and operating systems (ASPLOS), Satish Narayanasamy, Cristiano Pereira, Brad Calder. The dissertation author was the primary investigator and author of this paper.

Chapter V contains material that appears in "Automatically Classifying Benign and Harmful Data Races Using Replay Analysis", in *ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)*, Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, Brad Calder. The dissertation author was the primary investigator and author of this paper.

VITA

2001	Bachelor of Engineering in Computer Science and Engineering College of Engineering, Anna University, India.
2005	Master of Science in Computer Science University of California, San Diego
2007	Doctor of Philosophy in Computer Science University of California, San Diego

PUBLICATIONS

Weihaw Chuang, Satish Narayanasamy, and Brad Calder. "Accelerating Meta Data Checks for Software Correctness and Security". *The Journal of Instruction-Level Parallelism*, Volume 9, June 2007.

Smruthi Sarangi, Satish Narayanasamy, Bruce Carneal, Abhishek Tiwari, Brad Calder and Josep Torrellas. "Patching Processor Design Errors Using Programmable Hardware". *IEEE Micro Special Issue: Top Picks from Computer Architecture Conferences*, January 2007.

Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards and Brad Calder. "Automatically Classifying Benign and Harmful Data Races Using Replay Analysis". *International Conference on Programming Language Design and Implementation (PLDI)*, June 2007.

Satish Narayanasamy, Ayse Coskun and Brad Calder. "Predicting Faults Based on Anomalies in Speculative Execution". *Design, Automation, and Test in Europe (DATE)*, April 2007.

Weihaw Chuang, Satish Narayanasamy, Brad Calder and Ranjit Jhala. "Bounds Checking with Taint-Based Analysis". *International Conference on High Performance Embedded Architectures & Compilers (HiPEAC)*, January 2007.

Satish Narayanasamy, Bruce Carneal and Brad Calder. "Patching Processor Design Errors". *IEEE International Conference on Computer Design (ICCD)*, October 2006.

Satish Narayanasamy, Cristiano Pereira and Brad Calder. "Software Profiling for Deterministic Replay Debugging of User Code". *5th International Conference on Software Methodologies, Tools and Techniques (SoMET)*, October 2006.

Satish Narayanasamy, Cristiano Pereira and Brad Calder. "Recording Shared Memory Dependencies Using Strata". *12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2006.

Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Osvaldo Colavin and Brad Calder. "Unbounded Page-Based Transactional Memory". *12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2006.

Satish Narayanasamy, Cristiano Pereira, Harish Patil, Robert Cohn and Brad Calder. "Automatic Logging of Operating System Effects to Guide Application-Level Architecture Simulation". *International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*, June 2006.

Satish Narayanasamy, Gilles Pokam and Brad Calder. "BugNet: Recording Application Level Execution for Deterministic Replay Debugging". *IEEE Micro Special Issue: Top Picks from Computer Architecture Conferences*, December 2005.

Satish Narayanasamy, Gilles Pokam and Brad Calder. "BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging". *32nd International Symposium on Computer Architecture (ISCA)*, June 2005.

Satish Narayanasamy, Hong Wang, Perry Wang, John Shen and Brad Calder. "A Dependency Chain Clustered Microarchitecture". *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2005.

Satish Narayanasamy, Yuanfang Hu, Suleyman Sair and Brad Calder. "Creating Converged Trace Schedules Using String Matching". *10th International Symposium on High Performance Computer Architecture (HPCA)*, February 2004.

Satish Narayanasamy, Timothy Sherwood, Suleyman Sair, Brad Calder and George Varghese. "Catching Accurate Profiles in Hardware". *9th International Symposium on High Performance Computer Architecture (HPCA)*, February 2003.

ABSTRACT OF THE DISSERTATION

Deterministic Replay using Processor Support and Its Applications

by

Satish Narayanasamy
Doctor of Philosophy in Computer Science
University of California, San Diego, 2007
Professor Brad Calder, Chair

The processor industry is at an inflection point. In the past, performance was the driving force behind the processor industry. But in the coming many-core era, improving programmability and reliability of the system will be at least as important as improving raw performance. To meet this vision, this thesis presents a processor feature that assists programmers in understanding software failures.

Reproducing software failures is a significant challenge. The problem is severe especially for multi-threaded programs because the causes of failure can be non-deterministic in nature. The proposed processor feature continuously logs a program's execution while sacrificing very little performance (1%). If the program crashes, the developer can use the log to debug the failure by deterministically replaying every single instruction executed as part of the failed program's execution. Two key mechanisms enable this deterministic replay feature. One is BugNet, a checkpointing technique, which logs all of the non-deterministic input to a thread by logging the values of load instructions. The other is Strata, a logging primitive for recording shared-memory dependencies in a snoop-based or a directory-based shared-memory multi-processor. The former is sufficient for uni-processor systems and the later is required for multi-processor systems.

As a proof-of-concept, this thesis presents a software implementation of BugNet replayer built using the Pin instrumentation tool.

To understand the space requirements of the BugNet recorder for debugging, this thesis empirically quantifies how much of a program's execution need to be logged and replayed in order to understand the root cause of a majority of bugs. Finally, to demonstrate the utility of the deterministic replay feature, this thesis presents a software tool built using a deterministic replayer that finds data race bugs in shared-memory multi-threaded programs and automatically prioritizes them. The data race detection tool was built in collaboration with Microsoft. It has been used to find and fix data race bugs in production code, including Windows Vista and Internet Explorer.

I

Introduction

Debugging software is becoming increasingly challenging due to the increasing complexity in the software and hardware systems. Leveraging the exponential growth in processor's performance over the last few decades, sophisticated software systems have been built. If we consider the latest operating systems for instance, Windows Vista's code base contains about 40-50 million lines of source code and Debian 3.1 contains over 200 million lines [101]. These mammoth software systems purportedly contains thousands of bugs [52] and the number of man-years required to develop a software system will keep increasing as the size of these software systems continues to increase. The advent of many-cores (multiple cores on the same processor chip) will further exacerbate the problem associated with growing complexity of software systems [3], because future applications will have to be parallelized to take advantage of a many-core processor. Traditionally, developing and debugging a parallel system has been an onerous task, because of the difficulty in understanding the non-deterministic interactions between the parallel threads in a multi-threaded system. Ensuring correctness in the face of growing system complexity, at both the hardware and software levels, is critical to the evolution of computing systems.

Significant effort has already gone into developing tools and method-

ologies for improving software engineering practices. To date, however, processors have offered very little support for developing robust software. This thesis presents a processor feature that enables deterministic replay of a program's execution, which significantly enhances a programmer's ability to understand and debug software bugs.

I.A Deterministic Replay and its Uses

When a software system crashes, the programmer needs a mechanism to determine the causes of the failure. Unfortunately, current systems provide just the final state of the system (core dumps) [53, 63], and it is very challenging to understand the causes of the crash by looking only at the final system state.

A solution to this problem is to provide system support for continuously recording information during a program's execution, which can be used by the programmer to deterministically replay the last few seconds of the failed program's execution. A deterministic replayer is one that is capable of executing exactly the same sequence of instructions with exactly same input and output operands like in the original execution.

Providing system support for deterministic replay will improve the debugging process in several ways. First, if the bug is non-deterministic in nature, a deterministic replayer ensures that a programmer can reproduce the bug. This property is especially important for the multi-threaded programs, which are prone to non-deterministic bugs like data races. Second, a deterministic replayer can be combined with an interactive debugger like `gdb` or Microsoft's Visual Studio to build a time travel debugger [42]. A time travel debugger will significantly improve a programmer's productivity. Third, any dynamic analysis can be performed over the recorded program's execution, offline during deterministic replay. An offline dynamic analysis has an important advantage in that it is not limited

by its performance overhead as it does not affect the behavior of a program's execution. Examples for such dynamic analysis are intrusion detection [26], memory leak detection [88], finding uninitialized variables etc. Chapter V presents a dynamic analysis based on a deterministic replayer, which automatically finds data races in the multi-threaded programs. Fourth, if the recorder is efficient, software vendors can use them to capture a remote site failure and debug it by deterministically replaying the failed execution.

Apart from debugging, system support for deterministic replay can also be useful for providing fault tolerance [48] and developing architectural simulators [59].

I.B Need for Processor Support for Deterministic Replay

Proving the correctness of even a small scale software system has remained a holy grail for computer scientists. Even after extensive Quality Assurance process, complex software built today still contain significant number of bugs. Commercial pressure to reduce the time-to-market and the ability to distribute patches over the Internet has only aggravated today's software systems' reliability. Tracking down and fixing bugs in production software can be a nightmare, costing a significant amount of time and money. Bugs in production software account for nearly 40% of computer system failures [51], and according to NIST [97], they cost the U.S. economy an estimated $59.5 billion annually!

Reproducing and debugging a bug at a customer site is difficult due to diversity in the hardware devices and the operating systems used by the customers. Also, non-deterministic bugs such as the data race bugs are difficult to reproduce and debug. To capture a bug at a customer site, we need an execution recorder that incurs very little performance overhead but still supports deterministic replay. Even during the testing process, a recorder with a prohibitive

performance overhead will significantly affect the behavior of a program's execution, which will significantly limit its use.

Existing software based replayers either do not support deterministic replay of multi-processor systems [26, 42, 12, 81, 91] and/or incur a high performance overhead [6] (on the order 15 times slowdown when compared to the native execution). Processor support is absolutely essential to provide almost zero overhead deterministic replay debugging solution, so that even the production runs can be continuously logged. If the program crashes, the developer can use the log to debug the failure by deterministically replaying the last second of the program's execution that preceded the crash.

This thesis has three main parts to it. First is BugNet [60, 61], a checkpointing and logging solution for recording sufficient information to deterministically replay a program's execution in a uni-processor system. Second is Strata [57], a logging primitive for recording shared-memory dependencies. Shared-memory dependencies are necessary for deterministically replaying a multi-threaded program on a multi-processor system. The final part of this thesis demonstrates an application of deterministic replay in enabling offline analysis to find data races in multi-threaded programs [58, 62]. The rest of this section introduces each of these three parts.

I.C BugNet for Deterministic Replay of a Uni-Processor Systems

To enable deterministic replay of a program's execution in a uniprocessor system, we need an ability to record the initial execution state of the program and all the non-deterministic inputs read by the program. Non-deterministic input include input from the external system such as I/O, DMA, processor clock, etc.

Prior deterministic replay solutions [104, 26, 91] employ a copy-on-write

based checkpointing solution to capture the initial execution state (memory and register state) of the program. In addition, they require additional support to explicitly identify every source of non-deterministic input and to log the values read from those sources. As there can be so many different sources of non-deterministic input, it is tedious to implement a recorder and replayer using this approach. Also, the recorder and replayer will be dependent on a particular operating system or a system configuration, which makes it difficult to maintain and port them. System independence is a necessary property for a processor-based deterministic replay solution for it to be useful across diverse systems used at the customer sites.

The key contribution in BugNet is the system-independent checkpointing and logging mechanism that supports deterministic replay of a program's execution on a uniprocessor system. BugNet logs the architectural register state at the beginning of a checkpoint, and then logs a memory value when it is first accessed by a load instruction. The same memory value is logged again only if it has been modified by an external event. Thus, unlike traditional checkpointing mechanisms, BugNet avoids the complexity of capturing information about system calls, I/O, interrupts, DMA, etc. As a result, BugNet can support deterministic replay of only the application's execution and the libraries it uses, but not the full system. However, BugNet does support deterministic replay of user code across *all* non-deterministic system events, including context switches and interrupts. In addition, BugNet does not require a final core dump of the system state for replaying, which significantly reduces the amount of data that must be sent back to the developer.

I.D Strata for Deterministic Replay of a Multi-processor System

For debugging multi-threaded programs executing on a multi-processor system, in addition to recording non-deterministic input from the external system, we also need to record the non-deterministic interactions between the concurrent threads. That is, for a shared-memory multi-threaded program, we need to capture the dependencies between the memory operations executed across concurrent threads.

To accomplish this, the state-of-the-art deterministic replay solution for a multi-processor system called the Flight Data Recorder (FDR) [104] determines the shared memory dependencies by monitoring the coherence messages. To record a shared memory dependency, FDR records the memory counts of the dependent threads (point-to-point log). FDR implements the Netzer transitive optimization [64] using processor support to reduce the number of shared-memory dependencies that need to be recorded.

Instead of using a point-to-point log, this thesis proposes using a *Stratum* log [57] to record the shared memory dependencies. A stratum consists of the memory counts of all the threads at the time when it is logged. A stratum separates all the memory operations that were executed in all the threads before the time when it is recorded, from those that will be executed after it is recorded. Using this property of Strata, we can implement a transitive optimization that is 12 times more effective than a point-to-point logging solution in terms of log size. Also, unlike the earlier mechanisms, Strata can be used to capture multi-threaded dependencies in a snoop-based multi-processor system. Further, Strata can be recorded using 1/16th the hardware required for implementing a point-to-point logging solution.

I.E Deterministic Replay for Debugging

This thesis explores the use of deterministic replay for debugging. The process of debugging with the help of a deterministic replayer is referred to as Deterministic Replay Debugging (DRD). This thesis empirically quantifies certain variables relevant to the DRD process. This includes an empirical analysis [58], which quantifies how much of a program execution has to be logged and replayed, in order to understand the root cause of a majority of bugs in the production code. The result of this analysis shows that we can understand the root cause of a majority of bugs in the open source programs if we have the ability to replay about 10 million instructions that precedes the crash. Further, we examine the potential benefit of using dynamic slicing along with a deterministic replay debugger.

To illustrate the benefits of deterministic replay debugging, this thesis describes an offline dynamic analysis tool based on a replayer, which automatically finds harmful data races. This dynamic data race detection tool was built in collaboration with the Microsoft Corporation. It is based on the replayer that was independently developed at Microsoft [6].

Many concurrency bugs in multi-threaded programs are due to data races. There have been many efforts to develop static [8, 32, 56] and dynamic mechanisms [85, 25, 108, 66, 71, 1, 77] to automatically find the data races. However, most of the prior work has focused on finding the data races and eliminating the false positives. Even if we manage to eliminate all the false positives, not all of the remaining true data races are harmful. In fact, in the production code, we found that only 10% of the true data races are actually harmful. The remaining 90% were all *benign* data races. They were benign in the sense that the programmer was convinced that they do not affect the program's correctness and so the programmer intentionally chose to avoid the overhead of synchronization. Thus, reporting all the true data races places a huge burden on the developers as

they have to manually triage and eliminate a large number of benign data races. Triaging data races is a time consuming and tedious exercise.

The replay-based dynamic analysis presented in this thesis [62] automatically classifies the data races into two categories – the data races that are potentially benign and the data races that are potentially harmful. We discuss our experiences in using our dynamic race classification approach on an extensively stress-tested build of Microsoft's Windows Vista and Internet Explorer. The proposed technique was able to automatically filter out over half of the real benign data races, classifying them as potentially benign, which can be ignored by the developers. In addition, all of the harmful data races were correctly classified as potentially harmful. They were reported to the developers, and they all have been fixed in the production code.

I.F Contributions

This thesis makes the following contributions:

- Motivates the need for providing processor support for deterministic replay, presents a comprehensive solution for supporting deterministic replay with and without using processor support, and explores an application of deterministic replay for debugging.

- Presents the BugNet checkpointing and logging solution that captures all the non-deterministic input read by a thread of program's execution by logging the values of the load instructions. BugNet-based recorder and replayer are easy to implement and maintain. Evaluation of the processor-based BugNet recorder design is also presented.

- Presents Strata logging solution for recording the shared-memory multi-threaded dependencies, which are necessary for replaying the execution of a

multi-processor system. Strata logs are 12 times smaller, and requires 1/16th the hardware when compared to the prior solution. Unlike earlier schemes, Strata logs can be recorded in a snoop-based multi-processor system as well.

- Presents an analysis of the bugs in the open source programs to determine the resource requirements of a recorder. This analysis introduces a notion of *replay window length* for a bug, which is the number of instructions that need to be replayed to understand the root cause of the bug.

- Makes an observation that not all the data races are harmful bugs. Presents a unique replay-based dynamic analysis tool that automatically classifies the benign and harmful data races. This tool has been used to find and fix several bugs in Windows Vista and Internet Explorer, and continues to be widely used at Microsoft. These applications demonstrate the need for providing processor support for deterministic replay.

I.G Organization

Chapter II discusses in detail about deterministic replay, its applications and existing software and processor based solutions. Chapter III presents the BugNet checkpointing and logging technique. It also discusses the usage models for BugNet and characterizes the replay window length for various open source bugs. Chapter IV describes Strata and discusses how it can be recorded in both snoop-based and directory-based multi-processor systems. Chapter V presents a replay-based dynamic analysis tool that automatically classifies benign and harmful data races. Chapter VI concludes with a discussion on the opportunities for future work.

II

Background and Related Work

Computer system trends have increased the importance of providing efficient solutions to find and prevent software bugs. Lowering hardware costs have significantly reduced hardware's importance in terms of total computer cost [34, 69]. Lower hardware costs and increasing software complexity has increased the software's component in the total cost of ownership of a system. In addition, with the wide spread use of the Internet and how easy it is to release patches, software is released with more potential bugs than in the past. The need for multi-threaded programs for even desktop applications has never been as compelling as it is today, as we step into the multi-core era. Multi-threaded programs conventionally have been difficult to develop, debug, and maintain. Given these trends it is just as important to examine efficient hardware support for software correctness, security, and debugging as it is to increase the performance of the next generation of processors.

This thesis proposes a processor feature that provides support for deterministically replaying a program and demonstrates the utility of this feature for debugging. In this section, we first define what deterministic replay is, motivate its uses, and then discuss various challenges in implementing such a feature. This is followed by a detailed discussion on prior software-based and hardware-based

solutions for providing replay support. Finally, we discuss prior work on providing support for various other software-reliability oriented processor features, including the debugging features that got implemented in real processors.

II.A Deterministic Replay

When a computer system crashes, the programmer needs a mechanism to determine the causes of the failure. Unfortunately, current systems provide just the final state of the system (core dumps) [53, 63], and it is very challenging to understand the causes of the crash by looking only at the final system state. However, if we can deterministically replay exactly what happened, then it will help us understand the system crash.

Various layers of abstraction for a computer system executing a multi-threaded program is shown in Figure II.1. A full system deterministic replay should be able to replay all of the hardware and software components in a computer system.

Deterministically replaying a hardware component in a system (eg: a processor or a video card, which are shows as the outermost abstraction layer in Figure II.1) would involve recording the hardware component's internal state (by reading from the JTAG ports) and replaying the electrical signals in these devices. There have been works that focus on providing support for deterministically replaying a hardware component of a computer system [83]. This is useful for debugging bugs in hardware device.

However, to debug a software program, it is sufficient to deterministically replay just the software. We say that we can deterministically replay a software program, if we can replay exactly the same sequence of instructions with exactly the same input and output values like in the original execution. To support deterministic replay of a software program, however, we need to record and

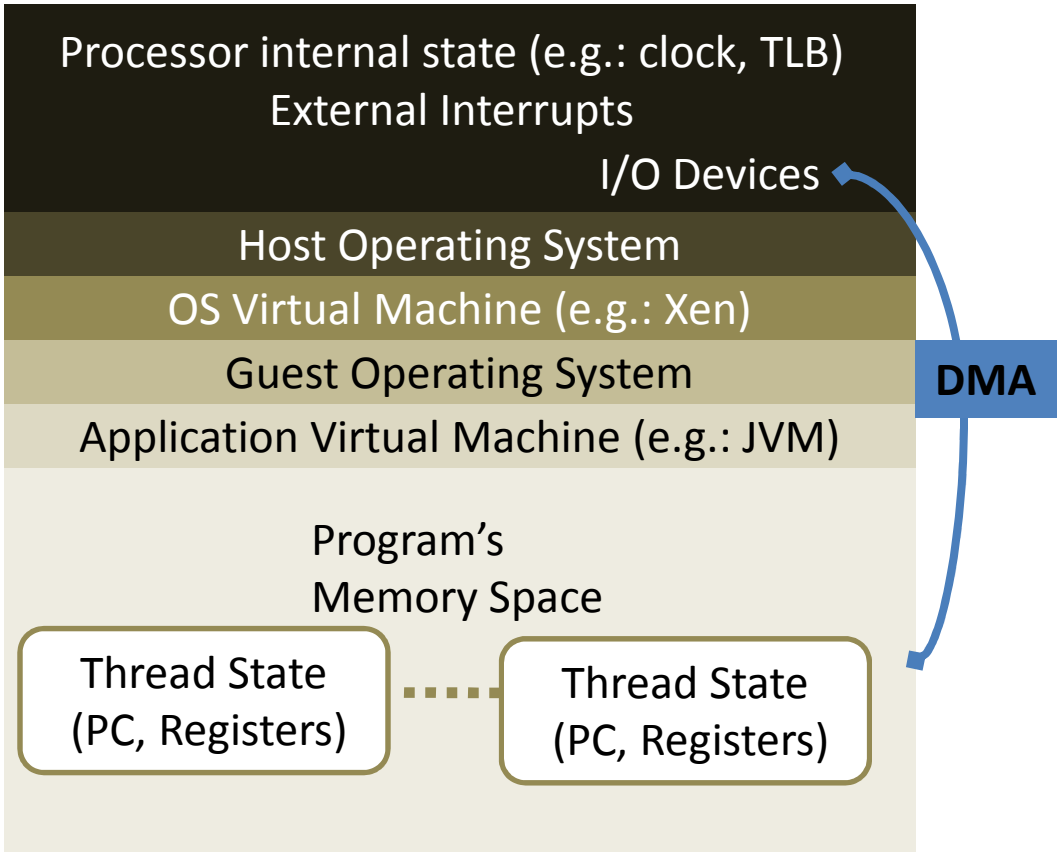

Figure II.1: Layers of abstraction in a typical computer system.

replay the values that are read from any external hardware device (that is, we need to emulate the hardware device), which is one of the challenges in supporting deterministic replay of a program's execution.

This thesis focuses on providing support for debugging software. Hence, the rest of the discussion in this section will focus on supporting deterministic replay of just the software systems.

II.B Deterministic Replay Uses

This section discusses several applications of providing support for deterministic replay.

II.B.1 Deterministic Replay for Debugging

Providing system support for deterministic replay will improve the debugging process in several ways.

Reproduce Non-Deterministic Bugs

Non-deterministic bugs or Heisenbugs are hard to reproduce. The reason is that they are dependent not just on the input to the program but also on the environment (operating system, run-time libraries, debugging environment, processor speed etc.) in which the program is running. For example, a memory allocator might randomly choose a location in the program's address space to allocate memory for a buffer. This can lead to non-deterministic buffer overflow bugs. In a C program, a bug due to an un-initialized variable can be non-deterministic. Another class of non-deterministic bugs are the data race bugs in multi-threaded programs, which are dependent on the order in which threads are executed.

A non-deterministic bug encountered in a program's execution is hard to debug, because when a programmer executes the program in a debugger with exactly the same input the bug might not appear again. With deterministic replay support, however, conventional debuggers like `gdb` or Microsoft's Visual Studio can reproduce a bug encountered during a program's execution any number of times.

Reproduce Remote Site Failures

If the recorder is efficient, software vendors can use them to capture a remote site failure and debug it by deterministically replaying the failed execution any number of times. Customers and beta-testers will find such a mechanism useful to report the software failures to the developers.

Time Travel Debugging

With deterministic replay support, an interactive debugger like `gdb` or Microsoft's Visual Studio can be enhanced to support a time travel debugger [42, 6, 78]. A time travel debugger provides functionalities such as step backward, reverse breakpoints and reverse watchpoints [7]. These features are complementary to step forward, breakpoints and watchpoints found in commonly used debuggers like `gdb`.

Using a breakpoint, a programmer can fast forward a program's execution to a particular line in the source code. Reverse breakpoint on the other hand allows a programmer to go back in program's execution time till the execution reaches a desired line in the source code. Watchpoints and reverse watchpoints are similar to breakpoints and reverse breakpoints, except that watchpoints are not set on a particular function or a source line, but on a variable.

The reverse debugging functionalities discussed above can be supported

using the following approach based on deterministic replay support [42, 6]. A recorder would create checkpoints at regular intervals. A program's execution can be replayed from the beginning from any of these intermediate checkpoints. Let us say that the programmer wants to step back by one instruction. To reach a desired point in the program's execution, the debugger would start the replay from the beginning of the checkpoint that contains the target instruction till the replay reaches the target instruction. If the checkpoint interval lengths are small enough, then stepping back would appear instantaneous to the programmer. The recorder can use relatively longer checkpoint interval lengths for efficiency, and more checkpoints can be created during replay to shorten the checkpoint interval length. Reverse breakpoints and reverse watchpoints can be supported similar to step backwards functionality. More details can be found here [7, 42, 6].

These reverse functionalities are very useful during cyclic debugging when a programmer wants to examine a portion of a program's execution over and over again. With a conventional debugger like `gdb`, however, a programmer has to restart the program's execution from the beginning, which is not a productive way of using a programmer's time. Thus, a time traveling debugger significantly improves a programmer's productivity. To build such a debugger we need deterministic replay support.

Offline Dynamic Analysis

Dynamic analysis techniques can automatically find bugs in a program's execution. Valgrind [88] and Purify [35] are examples of dynamic analysis tools. Such tools can automatically find memory access violations such as memory leaks, uninitialized variables, data races [85], detection intrusions [26] etc. in a program's execution. However, dynamic analysis tools like Valgrind slows down a program's execution by a factor of 20 to even 100 times. As a result, they cannot

analyze an execution behavior of a program on a real machine. If we can record a program's execution without intruding the program's behavior, then we can perform the time consuming dynamic analysis during replay. Chapter V presents a unique dynamic analysis based on a deterministic replayer, which automatically finds data races in the multi-threaded programs.

Experience with Using Deterministic Replay Debugging

Microsoft has developed a software record and replay tool called iDNA [6] in parallel with our BugNet work (BugNet is discussed later in Chapter III). It is based on a load-based checkpointing mechanism like that of BugNet to enable deterministic replay. Experiences of this dissertation author in using iDNA is discussed in this section.

iDNA has been used to trace thousands of executions of Microsoft's applications and these traces are stored in a centralized repository. At a developer's site, a developer can afford to allocate large disk spaces for replay logs, and so even full execution of a program is recorded. In addition to iDNA, Microsoft has also built dynamic tools that analyze these recorded program executions by replaying them using the traces. Thousands of bugs (including very many non-deterministic bugs like data races) in Microsoft's applications like Windows Vista and Internet Explorer have been automatically found using these tools. Also, the testers and the users within the organization find it easier to report a bug in a program's execution to the developer through the logs that can replay that execution, instead of having to write a detailed bug report. The author of this dissertation built a dynamic analysis tool based on iDNA that automatically finds data races bugs. The details of this tool are discussed in Chapter V.

However, iDNA is about 15x slower for computationally intensive programs and about 5x slower for interactive programs, when compared to their

native executions. This is because it is a purely software-based implementation. Because of the high performance overhead of iDNA, it cannot be used to capture a natural behavior of a program's execution (especially interactive applications like Internet Explorer) on a real system. The processor-based BugNet support presented in this thesis, however, incurs negligible performance overhead. Therefore, developers can use BugNet for recording programs without altering their behavior on a real system and debug them. Also, the customers can use BugNet to capture the bugs even in the production runs and report them to the developers, which is not feasible with iDNA.

II.B.2 Deterministic Architectural Simulators

Simulators used for studying a processor architecture [9, 29] have to correctly execute a program to analyze the architectural characteristics for that particular program's execution. Typically, architectural simulators use a functional system emulator to correctly execute the system calls and interrupts invoked by the program. These functional system emulators are complex to develop and maintain. To emulate a system call, an emulator would invoke an equivalent system call in the host system on which they are running. The values returned by an emulator for a system call are dependent on the host system's environment, which can lead to non-deterministic behavior in the program being simulated. Also, the simulation of a multi-threaded program on a multi-processor system can be non-deterministic because of the synchronization and data races in the multi-threaded program. Non-determinism in architectural simulations is an issue, because it is hard to compare two architecture designs when a program's execution is not the same across the two simulations.

To avoid the complexity of developing and maintaining a functional emulator, and to ensure deterministic simulation of a program's execution, one

can use a deterministic replayer instead of a functional emulator [59].

II.B.3 Fault Tolerance

System support for deterministic replay can also be useful for providing fault tolerance. ExtraVirt [48] keeps a replicated copy of a virtual machine and ensures deterministic replay in that replicated copy. The output from the original execution and the replicated copy is compared to ensure fault tolerance against transient faults.

II.C Overview of Prior Record and Replay Solutions

Figure II.1 shows the various layers in a typical computing stack. The recorder and the replayer can be implemented at any of these layers - in the virtual machine monitor [26], or the guest operating system [91], or the application's virtual machine such as the Java Virtual Machine [13], or by compiling or instrumenting the application itself.

Recording a software system's execution essentially involves two parts - checkpointing and logging all the non-deterministic input read by the system being recorded.

Checkpointing is necessary to retrieve the initial execution state of the system from where we would like to replay. A checkpoint mainly constitutes the memory state and the architecturally visible register states of the software system. Most of the record and replay systems [42] create a new checkpoint at regular intervals called checkpoint intervals, instead of just one checkpoint at the beginning of a program's execution. This is useful in two ways. One, if we run out of space to hold the logs, we can discard the logs from the oldest checkpoint and still be able to replay some parts of the program's execution. Two, during debugging, the programmer can fast forward to any intermediate instant in the

program's execution without having to replay from the beginning of the program's execution.

Checkpointing [28] is relatively an easy problem to solve. Some systems [26] just take a complete checkpoint of the software system's memory and register state at the beginning of the execution. However, this solution is inefficient and not useful for taking checkpoints at regular intervals. The most common solution is to employ some form of a copy-on-write checkpointing mechanism [42, 7, 31, 91] to reduce the amount of information recorded and also to reduce the performance overhead. It works as follows. Instead of logging the values in all the memory locations at the beginning of a checkpoint interval, a memory location's value is logged only when it gets modified for the first time within the checkpoint interval. The memory state of the process at the beginning of a checkpoint interval can be rebuilt, during replay, by starting with the process's complete memory state at the end of the program's execution and progressively restoring the values from the latest checkpoint log. This thesis presents a checkpointing solution that is different from the copy-on-write mechanism. It involves checkpointing just the register state. The memory state is captured by logging the values of the load instructions, which also captures all the non-deterministic input to the program. This means that the final complete memory state (core dump) is not required for replay, which is necessary in a copy-on-write based checkpointing mechanism to rebuild the memory state at the beginning of a checkpoint. This solution will be discussed in detail in Chapter III.

The second part of a recorder involves logging all the non-deterministic events that can influence the execution of the software system. Logging all the non-determinism is the most difficult part of recording a software system's execution. Any value read from the system external to the recorded system is considered to be non-deterministic. If the recorder is implemented in the appli-

cation's space, then any value read from the layers above the application state shown in Figure II.1 is considered to be non-deterministic. This includes all the system interactions such as the values read from the system calls, interrupts, instructions that read processor's state such as processor's clock (e.g.: `rdtsc`), all the I/O including the memory-mapped I/O and DMA. Apart from these non-deterministic system interactions, if the application is multi-threaded, then there could be non-deterministic interactions between the threads reading and writing from the shared memory locations.

If the recorder is implemented in the higher layer of abstraction such a virtual machine monitor, then the only non-deterministic input in a uniprocessor system can be from the external host operating system and the underlying hardware. Though the decisions of the guest operating system and hence its thread scheduling will be deterministic, to ensure deterministic execution of the virtual monitor and its processes, we still need to take care of the system calls to the host operating system, all the I/O including memory-mapped I/O, DMA and all the non-deterministic instructions.

Capturing all these sources of non-deterministic interactions is what makes it challenging in providing record and replay support. Providing this support in a way that the recorded execution can be replayed across different platforms and processor architectures is even more challenging. Apart from being accurate in capturing all the non-determinism, a recorder should also be efficient in terms of both performance and space overhead. A recorder with a high performance overhead can interfere with the program's execution, making it difficult to record and understand a program's execution on a real system.

II.D Prior Record and Replay Systems

In this section, we discuss various prior proposals for recording and replaying a program's execution. We start with a description of a simple recorder for a deterministic system. Then we discuss various solutions that provides support for recording non-deterministic system interactions such as system calls, I/O and DMA. Then we discuss solutions for taking care of non-determinism in multi-threaded programs executing on a uniprocessor system. This is followed by a description of the software solutions for recording a multi-threaded program running on a multi-processor system. Finally, we discuss how hardware support can solve some of the limitations of the software-based solutions in replaying multi-processor systems. We conclude with a discussion on the contribution of this thesis and compare them with the prior solutions.

II.D.1 Replaying a Program's Execution on a Deterministic System

IGOR: IGOR [31] is one of the earliest recorders that supported replay of a program's execution (IGOR does not support operating system replay). IGOR assumes that a program's execution is completely deterministic. That is, it does not handle non-determinism due to system interactions and races in the multi-threaded programs. Therefore, it provides support for just taking a checkpoint of the program's memory and register state. IGOR's checkpoint mechanism is a form of *copy-on-write* checkpointing. IGOR is based on operating system support. The OS supports a system call that would determine the program's virtual pages that were modified after the last checkpoint. To create a checkpoint at a particular instance in a program's execution, the data in all the modified pages are logged as part of the checkpoint. To replay from a particular dynamic instruction in a recorded program's execution, IGOR first determines the most recent checkpoint that precedes the starting point of replay. Then, the complete

memory state is reconstructed for that particular checkpoint. This is done by scanning the recorded log file backwards looking for the most recent log for each virtual memory page. The program is replayed from the reconstructed memory image using an emulator up-to the instruction specified by the user. The performance overhead of the recording phase is between 50% to 400%, while the replay overhead is on the order of 140 times when compared to the native execution.

II.D.2 Recording Non-Deterministic System Interactions

A program gets its input values from the external system through system calls (asynchronous interrupts), asynchronous interrupts and also memory-mapped I/O. Some of the I/O activity can also get delegated by the operating system to a DMA (Direct Memory Access) processor, which can modify the program's memory state concurrently with the program's execution. Also, a program can execute instructions that return non-deterministic values. For example, the instruction RDTSC (ReaD TimeStamp Counter) in x86 reads the processor's clock and writes it to an architectural register that is part of the program's register state. All these are sources of non-deterministic input to a program. To ensure deterministic replay of a program's execution, all of these non-deterministic input need to be recorded and replayed. In this sub-section, we discuss how prior proposals addressed this problem. None of the tools described in this section can record and replay a multi-threaded program's execution on a multi-processor system.

Boothe [7] developed a debugging tool to record and replay a program's execution. The tool supports reverse debugging features such as backward stepping and reverse breakpoint. It is based on compile-time instrumentation. The tool creates checkpoints at regular intervals and also captures the non-deterministic input through the system calls. To create a checkpoint, the

tool leverages the operating system's copy-on-write based forking mechanism. The tool instruments all the system calls in the program to record the return values from the system calls. However, the semantics of of each system call could vary in terms of how they return their values. That is, the specifications about the memory and register state side-effects of a system call varies across the system calls. Therefore, the tool has to explicitly take care of each of type of system call. Boothe in his paper [7] acknowledges the complexity of this solution. The tool supported only 35 out of 262 system calls in the UNIX system. It cannot handle asynchronous interrupts, memory-mapped I/O and DMA. The performance overhead is less than a factor of two when compared to the native execution. Chapter III presents a system-independent BugNet solution. It automatically captures the input values through all of the system calls, interrupts and DMA without having to explicitly take care of each of these sources of non-determinism.

jRapture: In jRapture [94], Java API classes that interact with the JVM and the underlying system through JNI calls are modified to record the non-deterministic input from the external system. But jRapture only records the values returned by the system call (both return by value and reference). It does not capture the side-effects to the application's memory state. Also, it does not record the interrupts, memory-mapped I/O and the values read through DMA.

Tornado: Tornado [19] requires modifications to the Linux operating system's kernels to trace the values written to the user space in the kernel mode. In Linux, the operating system can write to the user memory space only through a few write primitives defined in the code. Tornado instrumented these write primitives to keep track of the user memory locations that are modified by the system call. At the end of the system call, the tracer operating in the user space invokes `ptrace()` system call to obtain all the user memory locations modified by the system call and records the values for those locations. The performance

overhead of Tornado is less than a factor of two. Tornado, however, does not handle asynchronous interrupts, memory-mapped I/O and DMA.

Flashback: Flashback [91] provides operating system support for rolling back a program's execution and replaying it. It does not record a program's execution permanently. To rollback a program's execution, Flashback implements the following checkpoint mechanism. To create a checkpoint of a process, Flashback forks the process similar to Boothe's checkpoint implementation [7]. One of the two process is called the shadow process. The program's execution can be rolled back by restoring the state from the shadow process at any later point. To replay from a checkpoint, Flashback needs to record the non-deterministic input values from the system. Similar to Boothe's implementation, Flashback provides special attention to every type of system call to capture their return values. Interrupts are also recorded and replayed. However, it does not handle read and write to the memory-mapped locations that are shared between multiple processes. A simple copy-on-write checkpoint for the shared-memory locations wouldn't be sufficient, because it would require that we replay all the processes sharing the memory. Flashback, however, provides support for rolling back and replaying only the application program being debugged, and not the full system. While Flashback provides support for rolling back all the threads of a multi-threaded program, it does not support deterministic replay of all the threads after rollback even on a uniprocessor system. This is because it does not record and replay the thread scheduling order.

ReVirt: Unlike the other systems we have described so far, ReVirt [26, 42] takes care of all forms of non-determinism in a uniprocessor system. The recorder and the replayer are built inside the virtual machine monitor (UML-Linux). Unlike the previous systems we described, it supports complete deterministic replay of the entire virtual process and all the guest operating systems

and applications that are running inside the virtual process. To achieve this, ReVirt takes a checkpoint consisting of the entire virtual disk and the state of the virtual process (this checkpoint mechanism was later improved using a copy-on-write policy in TTVM [42]). After taking a checkpoint, ReVirt records all the non-deterministic events and input read from the host system on which the virtual process is running. (The reader can refer to Figure II.1, which shows the computing stack where the virtual machine monitor lies below the host processor, hardware devices and the host operating system). ReVirt assumes that there cannot be any inter-process communication between the virtual process and the other processes running in the host system. Therefore, ReVirt does not record all the non-deterministic events in the host system, but only those that affect the virtual process and the applications running inside the virtual process.

When an interrupt is delivered to the virtual process, ReVirt records it using a timestamp. ReVirt uses the number of branches executed after since the last interrupt as the timestamp, which is calculated from the performance counters of the host processor. In addition to recording the timestamp, architectural register contents of the virtual process are also logged.

In addition to recording the asynchronous interrupts, ReVirt records the input read from the host system. This is done by intercepting all the host system calls that could potentially read values from the host system. Apart from the host system calls, instructions such as `rdtsc` (read timestamp counter) and `rdpmc` (read performance counter) can return non-deterministic values and need to be logged. ReVirt configures the process control register (CR4) to generate a trap to the virtual process when any of these non-deterministic instructions are executed. On receiving a trap, the return values of the non-deterministic instruction is logged.

During replay, ReVirt ensures that the interrupts are delivered to the

virtual process exactly at the same instruction. Like in the recording phase, ReVirt uses the performance counters to keep track of the current timestamp value (branch count). Replay can be performed only on the hosts with the same processor type as the one used for recording because the implementation is dependent on the performance counters.

Earlier version of ReVirt [26] did not handle memory-mapped I/O, DMA and input read through privileged instructions such as `IN/OUT` instructions. Latest version of ReVirt called TTVM (Time Traveling Virtual Machine) [42], however, takes care of these non-determinism. The authors of TTVM modified the host OS's memory-mapping and DMA allocation routines to invoke equivalent system calls in the host operating system. TTVM sets page protection to all the virtual memory locations that are either memory-mapped or allocated for DMA transfer. On receiving a trap, TTVM records the value read from these locations. This mechanism might be slow as it generates a trap for every read to a memory-mapped location or a DMA read, but fortunately there are not too many memory-mapped and DMA accesses in the systems they evaluated. TTVM also supports a faster copy-on-write based checkpointing mechanism. The performance overhead is on the order of 10%-15% and the space overhead is about 85 KB/sec for SPECweb99.

ReVirt or TTVM handles multi-threaded programs running inside the virtual machine. This is possible, because the virtual process itself is replayed and therefore the thread schedule inside the virtual process is deterministic. However, it cannot support multi-processors as that would require support for recording the order between the memory operations executed in the concurrent threads.

II.D.3 Replaying Multi-threaded Program on a Uniprocessor System

The threads in a shared-memory multi-threaded program communicate by reading and writing to the shared memory locations in the program's address space. The order of the reads and writes across the threads is non-deterministic. Therefore, to replay a multi-threaded program, in addition to dealing with the non-deterministic system interactions, we should also be able to take care of the non-deterministic shared-memory reads and writes. This problem is easier to solve, if we assume that the multi-threaded program is running on a uni-processor machine. Because in that case, it is sufficient if we just record the scheduler decisions [13, 81]. In the previous section we discussed ReVirt [26], which also takes care of deterministic replay of a multi-threaded program on a uniprocessor system. ReVirt was able to provide that support without recording the thread schedules, because it deterministically replayed the entire virtual process responsible for scheduling the threads. In this section we describe tools that do not have the capability for full system replay, but still manage to replay a multi-threaded program by recording the thread scheduling order.

DejaVu: DejaVu [13] is a record and replay infrastructure based on run-time instrumentation. It is built on top of IBM's Java Virtual Machine for servers called Jalapeno.

DejaVu instruments some of the input functions such as `Date()` to record the non-deterministic values read by the application from the system outside of the JVM. However, it does not record all possible non-deterministic input from the external system. For example, it does not discuss support for file I/O, DMA etc.

Jalapeno cross-optimizes the application code, the instrumentation code and the run-time system (such as garbage collector). Hence, DejaVu has to make sure that the recorder and the replayer's instrumentation code invoke symmetric

behavior in the run-time system. Also, during replay, a programmer might want to examine the application's execution state. DejaVu also needs to make sure that the JVM is not perturbed when the replay is halted and the application's state is probed. To ensure this, during replay, the JVM running the application is executed under another remote JVM. The remote JVM halts both the application and its JVM before probing their state.

To handle multi-threaded programs, DejaVu records the order in which threads are scheduled. Though the application's thread are scheduled by the JVM, the behavior of JVM itself is non-deterministic as its decisions are based on the wall clock time read from the external system. Therefore, DejaVu records the order in which threads are scheduled. The order is recorded using the current count of the *yield points*, which are the pre-determined points in a program's execution where there can be a context switch. However, this is not sufficient for replaying races between the threads running on a concurrent system.

In summary, DejaVu can replay multi-threaded program's on a uniprocessor system. DejaVu does not ensure deterministic replay for uni-processor systems, however, as it does not handle file I/O, etc. Further, DejaVu does not support intermediate checkpoints, and so the replay has to start from the beginning of a program's execution. Therefore, time travel debugging is not possible.

RSA: Repeatable Scheduling Algorithm (RSA) [81] also focuses on recording the order in which threads are scheduled to record a shared-memory multi-threaded program executing on a processor machine. RSA is based on operating system support (Mach OS) for capturing the context switches. It also requires compile-time instrumentation to keep track of the instruction count (updated at every backwards branch), which is used to record the time of a context switch. This approach incurs an overhead of 10-15%, higher than DejaVu which uses yield counts instead of the instruction counts for recording the thread sched-

ules. RSA does not discuss any support for reproducing non-deterministic input read from the external system such as from the I/O devices, clocks, etc. Also, like DejaVu, it does not provide intermediate checkpointing support necessary for time travel debugging.

II.D.4 Recording Multi-threaded Programs on a Multi-Processor System

All the approaches that we have described so far cannot replay a multi-threaded program's execution on a multi-processor system. The basic requirement to solve this problem is that the races (both synchronization and data races) in multi-threaded programs are recorded and replayed. In this sub-section, we discuss techniques that address this problem.

Recap: Recap [68] instrumented every shared read at compile-time and recorded its value during the program's execution. This solution can replay a multi-threaded program. But still it does not provide the order between the memory operations executed across all the threads. Also, tracing the value of every shared-memory read is prohibitively expensive in terms of both log size and performance overhead [18].

InstantReplay: Instead of recording the values of shared-memory reads, InstantReplay [44] recorded the order in which the shared-memory objects were read and written by the threads. This was achieved by forcing every shared-memory access to a shared object through a procedure/stub. The stub acquires the lock for accessing the shared object, updates the version number if the access is a write, and then records the version number in the log file. InstantReplay is suitable for systems where the objects are shared at a coarser level (through monitors and message queues). However, performance degradation and space overhead when the program uses fine-grained shared memory accesses can be

severe. Also, it relies on the assumption that the shared accesses will acquire locks in the proper order. If there are data races, however, they will not be recorded and replayed correctly. Netzer [64] proposed an optimization to reduce the number of logs by using the transitivity property. That is, an order is not recorded if that order is transitively implied by the earlier logs. We will discuss this technique in more detail in Chapter IV.

RecPlay and JaRec : The size of the logs containing the order between the shared-memory operations was further reduced using Lamport clocks per objects in RecPlay [79]. RecPlay uses a JIT to instrument the synchronization operations. It records only the order between the synchronization operations. Therefore it cannot deterministically replay the data races. JaRec [33] implemented a similar to solution that efficiently logs the order between the shared-memory access within a JVM. Neither RecPlay nor JaRec can deterministically replay a multi-threaded program with data races.

TraceBack: TraceBack [4, 96] is a static instrumentation based tool that collects and reconstructs control flow information of multi-threaded applications. It cannot replay a program. However, a programmer can trace through the control flow of the program. TraceBack collects a trace containing the control flow information for each thread separately. The traces collected in all the threads are ordered using timestamps (processor's clock is used as a timestamp). A timestamp is recorded at every synchronization point in the program's execution. If any two execution regions in two threads are not ordered by any synchronization, then TraceBack will not be able to order the records collected for those two execution regions. This limitation might make it difficult to understand the data race bugs. TraceBack's performance overhead is about 5%-60%, when compared to all other software-based tools, it is more suited for tracing and debugging production runs on multi-processors. However, since TraceBack does

not support replay, it cannot support many functionalities of deterministic replay such as time travel debugging, offline dynamic analysis, etc., that we discussed in Section II.B.

II.D.5 Hardware Support for Replaying Multi-Processor Systems

One of the first hardware techniques for supporting deterministic replay for a program executing in a multi-processor system was proposed by Bacon and Goldstein [5]. Their design was for a bus based system. They observed that dependencies between the threads executing in a multi-processor system can be captured by monitoring the coherence messages on the bus. However, they recorded all the coherence traffic on the bus, which resulted in prohibitive space overhead. Also, their system cannot handle non-determinism due to the system interactions.

The amount of information that needs to be logged to record the memory access ordering can be reduced by applying the Netzer's transitive optimization [64]. FDR [104, 106] is a hardware design that implements the Netzer transitive optimization in a directory based system. FDR adopts the SafetyNet [89], a copy-on-write checkpoint mechanism, for retrieving a consistent full system state corresponding to a prior instance in time. Additionally, it records all the inputs coming into the system (I/O, interrupts, DMA transfers) to enable replaying. With this recorded information, starting from the retrieved full system state, the original program execution can be replayed.

ReEnact provides an approach for rolling back and replaying the execution using thread level speculation support [74]. Its main goal is to dynamically detect data races. CORD [73] extends ReEnact to efficiently capture some, but not all, of the RAW dependencies in a snoop based system in order to detect data races.

II.D.6 Discussion

There is a long history of research to support record and replay of a program's execution mainly to support debugging. However, very few systems supported complete deterministic replay in the presence of system calls, interrupts, DMAs, memory-mapped I/O and races in multi-threaded programs.

ReVirt [26, 42] based on virtual machine support achieves accurate deterministic replay for a uni-processor system. Also, it is efficient both in terms of space and run-time overhead for recording and replaying. However, it has to take care of all the corner cases that could lead to non-deterministic behavior in the virtual process and the applications running inside the virtual processor (eg: DMA, memory-mapped I/O, instrument every system call that could read input from the system, privileged instructions that read non-deterministic input from the external system etc.). As a result, much of the implementation is tied to the specific guest and the host operating system and even the processor type. In theory, it is possible to use the same solution and port the system. However, in practice, it is a tedious exercise, because each system configuration can give rise to unique sources of non-determinism and the system needs to take care of them. Also, ReVirt does not support multi-processor replay.

FDR [104, 106] is a hardware design that primarily focuses on capturing the order between the shared memory operations to support multi-processor replay. The non-deterministic input from the system is recorded using a solution that is pretty much same as in ReVirt, in that every possible source of non-determinism is identified and recorded. For a hardware-based solution, it is desirable to have a solution that is completely system-independent so that the processor feature can be used across diverse system environments. Also, a system independent solution will enable replay of a recorded execution across different system environments, which is important for replaying remote system failures.

This thesis proposes a logging and checkpointing solution that is system-independent. It is based on efficiently recording the values of the load instructions. Our load-based checkpoint mechanism is operating system independent. Therefore, it is relatively easy to develop and maintain. Also, it naturally handles DMA, memory-mapped I/O and other such system interactions. Further, it can deterministically replay multi-threaded programs running on a multi-processor system. We discuss the hardware implementation of this checkpointing and logging solution in Chapter III. The software implemenation of this solution can be found here [58]. In parallel with our development of BugNet, Bhansali et al. [6] at Microsoft developed a tool called iDNA to support deterministic replay using a load-based checkpointing scheme. However, they did not focus on precisely capturing the shared memory dependencies, which we can capture using our BugNet software tool. This thesis discusses a dynamic analysis tool that we built based on iDNA in Chapter V. The tool can automatically find data races and prioritize them by filtering out the potentially benign data races.

This thesis also proposes a solution to record the shared-memory dependencies efficiently using processor support in Chapter IV. Unlike FDR [104], the proposed strata-based solution is applicable for snoop-based systems and the log sizes are 12 times smaller.

II.E Processor Support for Debugging and Software Correctness

Historically, processors have in fact provided support for debugging and detecting programming errors. Myers talks about architectural features for improving software reliability in his book published in 1978 [55]. His book discusses architectural support for detecting uninitialized variables, type violations, access-right violations, checking if the value of a variable is within an expected range,

etc. Some of these features were actually implemented in the Burroughs B1700 processor [67].

Johnson [40] discussed the requirements for architectural support for debugging, primarily for designing interactive debuggers. He discussed a range of features for implementing an interactive debugger with breakpoints and watchpoints. In this section, we first discuss about the support that exist in modern processors for efficiently implementing breakpoints and watchpoints in an interactive debugger. Then we discuss the recent research proposals for providing processor support for dynamically checking the correctness of the program's execution.

II.E.1 Support in the Modern Processors for Breakpoints and Watchpoints

One common support that is found in many current day processors [39, 41, 90] is the support for breakpoints and watchpoints which are used in interactive debuggers like `gdb` [92]. Breakpoints and watchpoints enable the user to control the execution of the program and monitor the execution state. Using breakpoints, the debugger can stop the program execution when the execution reaches a particular point in the source code. Watchpoints on the other hand are useful to monitor accesses to arbitrary memory locations. Hardware can enable efficient implementations of these breakpoints and watchpoints by providing a set of dedicated registers that can hold the addresses of the instructions (to support breakpoints) or the memory locations (to support watchpoints). Whenever an instruction is executed, the address of the instruction or the address of the memory operand is compared against the contents of the special registers. If there is a match, then the control is transferred to the debugger and eventually to the user.

Unfortunately, current processors support only a limited number of registers for this purpose and hence the number of breakpoints or watchpoints that can be implemented using them are also limited. If the user needs more breakpoints or watchpoints, then the debugger has to single step through the program. Single stepping involves context switching to the debugger after executing every instruction of the program that is being debugged. The debugger is responsible for making the address comparisons once it gets back the control. Clearly, single stepping would incur a very high performance overhead and affects the inter-activeness of the debugger.

Another drawback of the existing hardware schemes is that they do not support conditional breakpoints and watchpoints. In the case of conditional breakpoints and watchpoints, it is just not enough to compare the addresses before stopping program execution, but in addition, a user specified condition needs to be satisfied. Existing hardware mechanisms cannot check these user conditions and hence, the control has to be transferred to the debugger whenever an address match is found.

Recently, Corliss et al. [17] proposed a hardware mechanism called DISE to implement a number of conditional breakpoints and watchpoints that can be used to implement fast interactive debuggers.

II.E.2 Processor Support for Software Correctness

Apart from providing support for *offline debugging* through breakpoints and watchpoints, it is also important to develop techniques to debug those bugs that occur at the customer site. The reason for this is that, even after extensive Quality Assurance process, a complex software system built today still contains significant number of bugs. These software bugs account for nearly 40% of computer system failures [51] and according to NIST [97] they cost the U.S. economy

an estimated $59.5 billion annually!

While we may have to live with the fact that bugs are unavoidable in complex software, we can try to ameliorate the problem by incorporating safety checks and dynamic bug detection mechanisms during production runs. Obviously, the performance overhead due to these mechanisms should be minimal if we want them to be used in production runs and here is where hardware can play an important role.

Software reliability can be improved by dynamically monitoring the program execution and verifying various properties. The goal is to dynamically detect a bug so that we can prevent the bug from corrupting the system state and potentially even recover to a consistent system state [69, 89]. *Memory access violations* are considered to be the most common form of bugs [95], especially in the case of unsafe languages like C/C++. Hence, there has been considerable interest in providing architecture support to detect memory related bugs [111, 110, 76]. iWatcher [111] provides sophisticated watchpoints to debug applications. It associates tags with memory locations, and when these locations are accessed, a specific function is executed to perform monitoring. SafeMem [76] and AccMon [110] are other recent proposals that provide architectural support to catch memory violations dynamically during program execution.

Apart from causing incorrect program behavior for certain inputs, memory related bugs also open up opportunities for attackers to subvert the security of the system [102]. For example, if the accesses to arrays or in general pointer accesses are not guarded properly then they can be exploited by an attacker to launch buffer overflow attacks which are the most common form of attacks [82]. Software based solutions such as PointGuard [20], StackGuard [21] try to prevent *buffer overflow attacks* with the goal of improving software security but they cannot protect against all buffer overflow attacks. Recently Tuck et al. [99] pro-

posed architecture support to improve PointGuard. There are several architecture mechanisms focused on just preventing buffer overflow attacks [45].

III

BugNet: Deterministic Replay of a Uni-Processor System

Chapter II described what deterministic replay of a software system is, and motivated the need for supporting deterministic replay. Section II.C discussed various challenges in supporting deterministic replay. First challenge is to provide an efficient mechanism for creating checkpoints at regular intervals during a program's execution. The second challenge is to capture all forms of non-deterministic input read from the system that is external to the recorded software system. The third main challenge is to record and replay the synchronization and data races in a multi-threaded program that is running on a multi-processor system. This chapter provides a processor-based checkpointing and logging solution called BugNet that solves the first two problems. Chapter IV will address the third challenge.

The BugNet checkpointing and logging solution presented in this chapter assumes processor support. BugNet primarily focuses on deterministically replaying the instructions executed in the user code and the shared libraries, and not the full system. BugNet, however, allows replaying an application across in-

terrupts and system calls. Deterministic replay of the user code and the libraries is sufficient for debugging a significant number of application-level bugs. The processor-based BugNet solution incurs less than 1% overhead when compared to the native execution, and therefore is useful for recording even the production runs.

This chapter is organized as follows. Section III.A discusses Flight Data Recorder's (FDR) [104] checkpointing and logging solution for capturing non-deterministic system input. We choose to compare our solutions with FDR, because to our knowledge, FDR is the only system that attempts to provide support for deterministic replay in the presence of all forms of non-determinism, including support for multi-processor replay. Also, FDR's performance overhead is low enough that it is suitable for capturing bugs even in the production runs, which is one of our goals. Section III.B presents the BugNet architecture. Section III.C quantifies how much a program's execution need to be recorded to debug a majority of the bugs. Also, it analyzes the space and performance overhead of BugNet's solution, and compares BugNet with FDR. Section III.D presents a few of extensions to the BugNet architecture for handling self-modifying code, handling interactive applications with frequent interrupts and supporting deterministic replay of operating system code. Section III.E discusses how and when BugNet logs would be collected. It also discusses an implementation of a replayer based on BugNet's logs and how it would be used for debugging. Section III.F concludes this chapter.

III.A Flight Data Recorder

The goal of FDR [104] is to provide architectural support for deterministically replaying the last one second of the full system execution that preceded a system crash. FDR supports full system replay. That is, it can replay the execu-

tion of the operating system code in addition to the user code and shared libraries. This includes deterministic replay of interrupts and system call routines.

FDR continuously records three kinds of information:

- Checkpoint Information - FDR creates checkpoints at regular intervals (let us call it checkpoint intervals) during a program's execution. To create a checkpoint, FDR uses the SafetyNet checkpoint mechanism [89]. The checkpoint provides a consistent main memory state from where one could start the replay. SafetyNet is a copy-on-write mechanism for checkpointing the state of the main memory. To create a checkpoint, instead of taking a snapshot of the entire main memory state, just the registers of all the processor context is logged. After that, whenever a memory location is modified for the first time, the value that is the over-written is logged. Using the final main memory state and the checkpoint logs, it is possible to rebuild the entire state of the main memory at the beginning of the checkpoint interval. Note that this would require a snapshot of the final state of the entire memory state.

- Interrupts and External Inputs - To deterministic replay the execution from the beginning of a checkpoint interval, FDR records all the interrupts (with a timestamp), input read from all the I/O devices, and DMA transfers. Separate hardware buffers are used to record values read from each input device. Like we pointed out in Section II.D, such an approach is complex to implement because of the several possible sources of non-determinism. Also, an implementation of such a solution would be dependent on the particular operating system and external hardware devices.

- Memory Races - To replay a multi-processor system, order of memory accesses across all the processors is recorded using an additional Memory Race

Buffer (MRB). This is based on Netzer's transitive optimization [64]. Chapter IV in this thesis discusses an alternative solution based on Strata.

On average, the combined size of all of the FDR's logs is about 34 MB [104] (after compression using LZ compressor [112]) to replay 1 sec of execution of a processor node. The amount of hardware state required to collect all the required information to enable full system replay is about 1.3 MB of on-chip hardware and 34 MB of main memory space. In addition, the final snapshot of the entire physical memory image is required to rebuild the initial state of a checkpoint interval. This could be on the order of 1 GB depending on the memory footprint of the application and the size of the main memory chip used in the system. BugNet, however, does not require the final snapshot of the entire memory state. Also, the on-chip hardware required to support BugNet's checkpoint and logging solution is lesser (on the order of 48 KB). More importantly, BugNet solution based on recording the values of load instructions is a system independent solution. Therefore, BugNet's implementation can be used to record execution on any operating system and system configuration, which is important for a processor-based feature. Also, a program recorded on a particular operating system can be replayed on any other operating system as the logging and replay solution is system-independent.

III.B BugNet Architecture

This section first gives an overview of the BugNet architecture, and then discusses each component of the BugNet architecture in detail.

III.B.1 BugNet Architecture Overview

BugNet is a checkpointing and logging solution for supporting deterministic replay. It focuses on recording the execution of the user code and shared

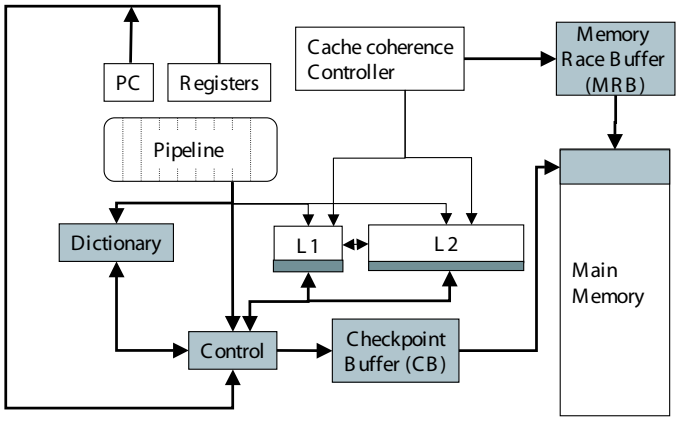

Figure III.1: BugNet Architecture.

libraries, which is sufficient for debugging a significant number of bugs in the application code. BugNet does support deterministic replay of the user code across interrupts and system calls. But, the developer will not be able to analyze what goes on during the execution of these interrupts and systems call routines because they are not executed during deterministic replay of the user code. Section III.D.3 discusses how BugNet can be extended to support deterministic replay of operating system code as well.

BugNet logs sufficient information to capture all forms of non-deterministic input read by the user code and the shared libraries from the external system (through system calls, interrupts, DMA, memory-mapped I/O, and non-deterministic instructions). If a thread in a shared-memory multi-threaded program reads a value written by another thread, that value is also logged. Only the order between the memory operations executed in concurrent threads is not logged by BugNet. Chapter IV discusses a solution to solve this problem.

In BugNet, a new checkpoint is created at the beginning of each checkpoint interval to allow execution to be replayed starting at the first instruction of the interval. Therefore, a *checkpoint interval* represents a window of committed

instructions that is captured by the checkpoint being logged. During tracing, a maximum size (number of committed instructions) is specified for the checkpoint interval. When this limit is reached, a new checkpoint interval is initiated. But a checkpoint interval can be prematurely terminated on encountering an interrupt or a context switch. During a checkpoint interval, enough information is recorded in a log to start the replay of the program's execution at the start of that checkpoint interval.

BugNet is built on the observation that a program's execution is essentially driven by the values read when executing load instructions. Hence, in order to replay a checkpoint interval, one needs to record just the initial register state and then record the values of load instructions executed in that interval. During a checkpoint interval, the memory values could have been modified by the interrupts, especially the I/O interrupts and the DMA transfers. In the case of shared-memory multi-threaded programs, the shared memory could have also been modified by other threads during a checkpoint interval. However, by logging the load values, we ensure that we have recorded information required for deterministic replay. The recorded information include any memory values that were updated by the interrupt handlers or by other threads in a shared memory processor.

Figure III.1 shows the major components in the BugNet architecture. Components shaded in gray are the new additions to the baseline architecture of a modern processor. BugNet operates by creating checkpoints at the beginning of checkpoint intervals. At the start of a checkpoint interval, a snapshot of the architectural state is recorded in the *Checkpoint Buffer* (CB). The recorded architectural state includes the program counter and register values. After initialization, whenever a load instruction is executed, a new log entry is created to record the load value. All this is stored in the CB.

Recording the result value of every load instruction would clearly be expensive in terms of both space and performance overhead. But we note that a load value needs to be recorded only if it is the first access to the memory location in the checkpoint interval. The result value of the other loads can be trivially generated during replay. To support this optimization, BugNet uses a bit with every word in the cache. The bit is set when the word is accessed for the first time and the value is logged. Later accesses to the word are not logged as the bit would be set. When applying this optimization, special care needs to be taken to handle interrupts and shared memory accesses by remote threads. If a memory address is modified by these external entities in the system, BugNet makes sure that the future load reference to that address is logged. In order to further optimize the trace size, BugNet uses a dictionary based compressor, which is shown in the Figure III.1. The compressor exploits the frequent value locality in load values [107].

The log that contains the load values and the initial architectural state information for a checkpoint interval is referred to as the *First-Load Log (FLL)*. The information recorded in the FLL for a checkpoint interval is sufficient to replay that interval. This allows us to re-execute the instructions with exactly the same input and output register and memory values as in the original execution. This is true even in the case of multi-threaded programs, because the FLL for a thread contains the necessary information to replay each thread independent of the other threads. In order to debug data races between these threads, we need to record additional information to reproduce the ordering of memory operations across the threads. We use the *Memory Race Buffer* (MRB) to record such information in a log called *Memory Race Logs (MRL)*. Chapter IV discusses how the memory ordering information is recorded in the MRB.

The CB and MRB are FIFO queues that are memory backed to al-

low the collection of larger logs than can be stored in these dedicated hardware buffers. The operating system provides support for managing the memory space that needs to be allocated for BugNet's use. The goal is to continuously log program execution. If the allocated memory space is exhausted, some recorded information need to be discarded so that part of the memory can be freed up and used for recording the latest execution of the program. However, while freeing up the memory we need to make sure that whatever log that is left is still usable for replaying a portion of program's execution. Thus, BugNet creates checkpoints at regular intervals. When the allocated memory space fills up, the logs corresponding to the oldest checkpoint for a thread are discarded.

In Section III.C we analyze the bugs in the open source programs and determine that an ability to replay the last 10 million instructions before a program's crash is sufficient for debugging a majority of the bugs. The allocated memory/disk space for recording a program's execution should be able to hold sufficient information for replaying at least 10 million instructions. This ensure that when the program crashes, we would have mostly likely captured adequate information for debugging the crash.

The space allocated for recording BugNet logs can be increased by spilling over the logs from the main memory to the disk. That is, instead of discarding the logs corresponding to the oldest checkpoint, when the allocated main memory space is full, those old logs can also be written to a disk in the system.

When the operating system detects that a program has encountered a fault, before terminating the application, the OS first records the instruction count of the current checkpoint interval and the program counter of the faulting instruction in the current FLL and then stores all the logs collected for that application to a persistent storage device. The logs would be then sent back to

the developer for debugging. BugNet is also useful for recording a program's execution at the developer's site during testing and development. Deterministic replay enables travel debugging capabilities and offline dynamic analysis discussed in Chapter II.B.

III.B.2 Checkpoint Scheme

For checkpointing, the program's execution is divided into multiple checkpoint intervals, where the interval length is specified in terms of the number of instructions executed. At the end of a program interval, the current checkpoint is terminated and a new one is created. In addition, interrupts and system calls can also terminate a checkpoint interval, which we will describe later. Finally, an exception in the execution would terminate the checkpoint interval and initiate the collection of logs. The logs would then be sent back to the developer for debugging.

A new checkpoint is recorded by creating a new FLL delimiter in the Checkpoint Buffer (CB) and initializing the checkpoint interval's FLL with the following header information:

- **Process ID and Thread ID** - are required to associate the FLL with the thread of execution for which it was created.
- **Program Counter and Register File contents** - are needed to represent the architectural state at the beginning of the checkpoint interval. This information will later be used by the replayer to initialize the architectural states before beginning to replay the program execution using the recorded load values.
- **Timestamp** - is the system clock timer when the checkpoint was created. This is useful for ordering the FLLs collected for a thread according to their time of creation.

The FLL after getting initialized with the above information will be appended with the output values of the load instructions executed during the checkpoint interval.

III.B.3 Tracking Load Values

Within a checkpoint interval, a load accessing a memory location needs to be logged only if it is the first access to that memory location. The values of other loads can be re-generated during replay. Logging just the "first-loads" to a memory location will significantly reduce the number of load values that need to be recorded. In order to do this optimization, we associate a first-load bit with every word in the L1 and L2 caches. At the start of a checkpoint interval all these bits are cleared. When a load accesses a word for which the bit is not set, then the load value will be logged in the Checkpoint Buffer, and the bit is turned on. If the bit is set for a word in the cache, then it implies that the value of that word has already been logged, and hence, future load accesses to it need not be logged.

This approach is adapted from FDR [104]. FDR's goal was to track the first store to a particular location during a checkpoint interval, and to log the value that is overwritten while executing the first store. FDR focused on stores, because it uses the store values to repair the final core image to retrieve the memory state at the start of a checkpoint interval. For BugNet, if the first access to a particular memory location is a store then we would set the bit and not log the value of the store. The future load accesses to this memory location would not be logged as well, as the bit would be set. The store values are not logged, and this mechanism works, because the stores and their values will be reproduced during replay.

When a cache block is replaced from the L2 cache, all the first-load bits

associated with the words in that cache block will be cleared. Therefore, logged values for addresses (blocks) evicted from the L2 will be re-logged when the block is brought back in and those same addresses are accessed again. The first-load bits in the L2 cache are used to initialize the first-load bits in the L1 cache when bringing in a block to the L1 from the L2. When an L1 block is evicted, its first-load bits are stored into the first-load bits of the L2 cache.

The above first-load optimization will be effective for long checkpoint intervals. This is because the greater the number of loads/stores executed, the higher the probability that a memory location has already been logged. As a result, the amount of information recorded to replay an instruction will decrease with longer checkpoint intervals.

During replay, we need to determine whether the value for a load instruction is recorded in the log or not. If it is recorded, then the load executed during replay needs to get its value from the FLL. If it was not recorded then it is certainly not the first access to the memory location that it is accessing. By simulating memory state during replay, the value can be obtained by reading from the simulated memory state. To determine when to consume a load value during replay, as part of each log entry we have a field to specify the number of load instructions that were skipped since the last load instruction was logged. The following is the format of each log entry in the checkpoint to record the information for the load instruction:

```
(LC-Type, Reduced/Full L-Count,
 LV-Type, Encoded/Full Load-Value )
```

The second field, *Reduced/Full L-Count*, represents the number of load instructions skipped (not logged) between the current load instruction being logged and the last load instruction logged. To record the full L-Count value, one would require `log(checkpoint interval length)` bits, since the L-Count

cannot be greater than the maximum checkpoint interval size used. We found that the majority of L-Count values can be represented using just 5 bits. Hence, we record L-Count using 5 bits whenever its value is less than 32. If the L-Count value exceeds 32, then we resort to recording the full L-Count value. The logs that contain the full L-Count values are distinguished from the logs that contain the 5-bit L-Count values by using one additional bit, the *LC-Type*.

The fourth field, *Encoded/Full Load-Value*, is used to record the load value. Again, we try to avoid recording the full 32-bit load value. To achieve this, we use a 64-entry dictionary that captures the most frequently occurring load values. If the load value is found in the dictionary, then we use 6 bits to represent the position of the value in the dictionary. If the load value is not found, then the full 32-bit value is recorded. *LV-Type* is the bit that is used to distinguish between the two cases.

To track load instructions we just record their output values in the log. Neither the effective address nor the address of the PC of the load instruction is logged, since they can be produced during replay of the thread's execution. In Section III.E we describe the replaying mechanism in detail.

Dictionary-based Compressor

In BugNet, the load values are compressed using a dictionary based compression scheme. It has been shown in [107] that the load values exhibit frequent value locality. That is, over 50% of all the load values can be captured using a small number of frequently occurring values. In addition, value predictors have been shown to provide impressive compression ratios [11].

In our approach, a 64-entry fully associative table, called the *Dictionary Table*, is used to capture these frequently occurring load values. The dictionary table is emptied at the beginning of the checkpoint interval and is updated with

the execution of each load instruction. Before logging a load value into the FLL, the value is looked up in the dictionary table. If there is a hit, instead of storing the full 32-bit value, we will store a 6-bit encoding. The 6-bit encoding corresponds to the rank of the value in the dictionary table. In our design, the rank corresponds to the index into the dictionary table used to find the matching value.

In a checkpoint interval, the dictionary table will be continuously updated as load instructions are executed. As a result, the position of a value in the dictionary table can keep changing during an interval. Therefore, the encoding that we use to compress a value can change over the course of a FLL. This is valid since we simulate the dictionary state during replay. During replay we know the initial dictionary state (which is the empty state) at the start of a checkpoint interval, and all the subsequent executed load instructions update the table. Therefore, at any instant of time, while executing a load instruction during the replay, the state of the dictionary table will be the same as its state during the original execution.

For *every* load that gets executed within the interval, the dictionary table will be updated as follows. Each entry in the table has a 3-bit saturating counter to keep track of the frequency of the value stored in the entry. When a load value is found in an entry in the dictionary table, the 3-bit saturating counter corresponding to that entry is incremented until it saturates. If the updated counter value is greater than or equal to the counter value of the previous entry in the table, then the two values will swap their positions (rankings) in the table. This ensures that very frequently occurring values will percolate to the top of the table. When a load value is not found in the dictionary table, then it is inserted into the entry with the smallest counter value. If there are multiple candidates, then the entry occupying the lowest position in the table is chosen

for replacement.

III.B.4 Handling Interrupts and Context Switches

Interrupts can be either asynchronous or synchronous. Asynchronous interrupts are caused by sources external to the executing application code, like I/O and timer interrupts. On the other hand, synchronous interrupts (also commonly referred to as traps) are triggered while executing program instructions. Reasons for traps include arithmetic overflow exceptions, invoking a system call or an event like a page fault.

Since our goal is to replay and debug only the application code, we do not record what goes on during any interrupts. So we do not record the output of load instructions executed as part of the interrupt handler and operating system routines servicing the interrupts. Nevertheless, we need to track how the interrupt affects the execution of the application. Interrupts are likely to modify the memory state (e.g. I/O interrupt) and they can even change the architectural state of the program's execution by modifying the program counter or registers.

A straight forward solution is to solve this problem by prematurely terminating the current checkpoint interval on encountering an interrupt and create a new one when the control returns to the application code. If we create a new checkpoint after servicing the interrupt, we are guaranteed to have the right program counter value, as it will be initialized in the header of the new FLL. Also, the bits used to track the first-loads would have been reset, thus ensuring that the necessary load values are logged to replay the instructions that were executed after the interrupt. The architecture that we model while discussing our results in Section III.C assumes this approach.

A more aggressive solution would be to allow the first-load bits to be tracked across the checkpoints and interrupts. This would help in reducing the

number of load instructions logged, when restarting a new checkpoint after the interrupt. The solution needs to make sure that the first-load bits are correctly invalidated when the memory state is updated during the interrupt or context switch. This approach is discussed in more detail in Section III.D.3 along with a solution for recording and replaying the execution of operating system code as well.

III.B.5 Handling External Input

The mechanism described in the previous section to handle interrupts is adequate to handle I/O interrupts as well. A memory mapped I/O mechanism works by mapping the address space of a device to the program's virtual address space. The values are read from the device through load instructions using the virtual address corresponding to the program's address space. These memory locations will not be cached in the processor. Therefore, the value of every load to a memory-mapped I/O device is logged. Similarly, the values of all non-cacheable loads are also logged.

The OS can initiate a DMA transfer to service an I/O system call. In such cases, the control will return to the application code but the DMA transfer can proceed in parallel. Like in FDR [104], we assume that a DMA write would use the underlying cache coherency protocol and invalidate the cache block in the processor executing the application. This would ensure that the bits used for the first-load optimization are reset. Thus, the values in the modified memory location would be recorded later when they get referenced by an application load.

Our scheme of recording only the first-load values avoids logging the data copied into the process's address space until it is referenced in the application. Even though a large amount of data can get copied into the process's address space, not all of it will necessarily be used by the program execution preceding

the crash. Our scheme ensures that we log just the necessary values that are in fact consumed by those instructions that need to be replayed later.

III.B.6 Support for Multi-threaded Applications

We assume a shared memory multiprocessor system to execute multi-threaded programs. In shared memory multi-threaded applications, remote threads executing on other processors can modify the shared data within a checkpoint interval. This problem is the same as the one that we discussed regarding DMA transfers. When a shared memory location is modified by a remote thread, the corresponding cache block will be invalidated before the update. This would reset all the bits used to track first-loads to that cache block. As a result, future load references to the same cache block would result in recording the value written by the remote thread in the FLL.

The FLL corresponding to a checkpoint interval is sufficient to replay the instructions executed in that interval. This is true even in the case of multi-threaded applications. Any thread can be replayed independent of the other threads as we would have recorded all the input values required for executing that thread. However, in order to assist debugging data races one would require an ability to infer the order of instructions executed across the threads. Chapter IV discusses the solutions to record the order between the memory operations executed in concurrent threads. This information is recorded as the Memory Race Log (MRL) in the Memory Race Buffer (MRB) shown in Figure III.1. In this section we describe only how BugNet's FLL checkpoint logs and MRL logs are collected and correlated during replay.

FDR's [104] checkpoint mechanism uses barrier synchronization to support shared memory multi-threaded applications. The mechanism ensures that the checkpoint intervals across all the threads start at the same instant of time.

This approach is not desirable in our BugNet architecture because of its overhead in terms of performance, especially when we want to create checkpoints with a smaller interval length. Moreover, we want to have the flexibility of terminating a checkpoint independent of other threads, like when interrupt events are encountered. Hence, we allow the threads to create and terminate checkpoints intervals independent of the other threads. As a result, the checkpoint intervals across different threads may not start at the same time. To support asynchronous checkpoints across threads, we record checkpoint identifiers as part of every memory race log entry, as described in the next section.

Memory Race Log

The purpose of the Memory Race Log (MRL) is to log the shared memory dependencies between the threads in a multi-threaded program. Chapter IV discusses how MRL is collected in directory-based and snoop-based multiprocessor systems.

III.B.7 Memory Backing

The First-Load Logs stored in the Checkpoint Buffer and the Memory Race Logs stored in the Memory Race Buffer (MRB) are memory backed at two different locations in memory. The amount of memory space devoted for this purpose is managed by the user and/or the operating system to ensure that the performance impact is within tolerable limits. The amount of memory space and disk space devoted for BugNet logs will determine the number of instructions that can be replayed.

The contents in the on-chip buffers are lazily written back to the main memory whenever the memory bus is idle. Since we compress the log entries as they are generated, the contents in the buffer are lazily written back to memory

at any point. This memory backed solution can potentially impact the memory bandwidth requirements due to extra traffic to main memory. When the processor is accessing main memory on encountering a cache miss, it will most probably be stalled waiting for data to be obtained from the main memory. Thus, the rate at which loads are logged in the FLL will be reduced. We found when simulating the SPEC benchmarks that there is sufficient bandwidth to write the logs back to memory when the memory bus is idle, and the on-chip buffers need to be only large enough to hold bursts in the logging.

III.B.8 On Detecting a Fault

The operating system will know when the program executes an instruction that causes the thread to be terminated. An arithmetic exception due to division by zero or a memory operation accessing an invalid address are some examples that can trigger the program to crash. Once the operating system detects that the program has executed a faulting instruction, it records the current instruction count and the program counter of the faulty instruction in the FLL. This is used to determine when to stop replaying and to correctly identify the faulty instruction. Then, the OS collects the FLLs and MRLs corresponding to the application from the main memory and hardware buffers. It scans through the headers of all the logs and uses the process identifier in the header to identify the logs that correspond to the application. These logs are more useful for debugging than the traditional core dumps that today's operating systems collect when a program crashes. The usage models for BugNet logs for debugging are discussed in Section III.E.

III.C Results

BugNet is based on the principle that the bugs can be reproduced, isolated and fixed by replaying a window of the program's execution that immediately preceded the crash. Though debugging by replaying the program is considered to be an effective technique in the software engineering community [7, 78], there has been no study on the length of the replay window of execution required to capture a majority of the bugs. In this section, we first quantify this length by studying popular desktop applications. This study reveals that replaying 10 million instructions is adequate to debug a significant number of bugs. Based on this result, this section analyzes the trace sizes and the amount of hardware resources that need to be allocated for BugNet. This section also compares BugNet with FDR [104].

III.C.1 Methodology

To evaluate BugNet, we use a handful of programs from the SPEC 2000 suite to evaluate the online compressor and to analyze the size of the log required for different interval sizes. These include `art`, `bzip`, `crafty`, `gzip`, `mcf`, `parser` and `vpr`. These programs were compiled on x86 platform using -O3 optimizations.

We also provide results for five programs used in the AccMon [110] study, and a handful of other programs that are in the top 100 programs downloaded from the *sourceforge.net* web site. The AccMon programs used are `bc`, `gzip`, `ncompress`, `polymorph`, `tar`. The single threaded sourceforge programs are `ghostscript`, `gnuplot`, `tidy` and `xv`. We also analyze a bug `napster`, which a multi-threaded program that we obtained from sourceforge.

We make use of Pin [49], an x86 binary rewriting tool to create the BugNet logs.

III.C.2 Replay Window Length

Using BugNet only a portion of a program's execution can be captured due to the limited memory and disk space available on a system. Our premise is that, in order to fix a bug, one needs to examine only a window of a program's execution that immediately precede a crash. However, it is not clear what the size of this window has to be in order to fix a majority of the bugs. We now quantify the replay window size required for fixing bugs, by matching the execution histories of the correct and the incorrect program executions corresponding to several open source bugs.

We define the *replay window* length for a bug to be the number of dynamic instructions executed between the source of a bug and the point where the buggy program crashes. We identify the source of the bug in a program's execution using a heuristic that identifies the bug source as the point in the buggy program's execution where its output starts to deviate from those of the correct program's execution. One use of this measurement is that it helps us to understand the log space requirements of the BugNet logger. Thus, it helps us to analyze the efficacy of the deterministic replay debugging technique.

Following is the methodology we used to quantify the replay window length for a bug. In order to determine the replay window length for a bug, we take the two binaries corresponding to two versions of the same program. One binary corresponds to the source code that contains the bug. Another binary corresponds to the same source code with the bug fixed. We execute these two binaries with the same input that exposes the bug in the buggy program's execution.

Figure III.2 shows the buggy behavior of a version of gzip program with respect to the correct program's execution. The x-axis represents the buggy program's execution time. The y-axis represents the magnitude of the difference

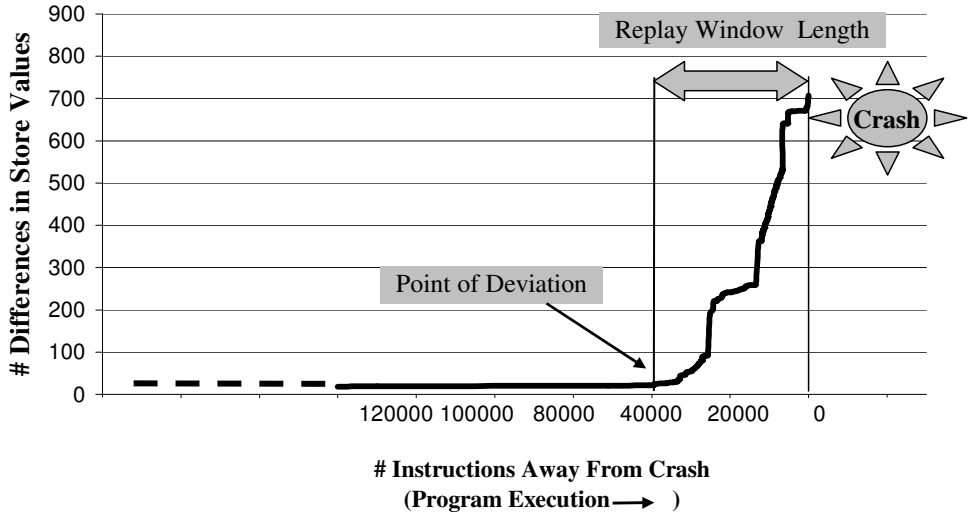

Figure III.2: Replay Window Length showing a buggy program's execution behavior relative to the correct program's execution for gzip.

in the store output values between the correct program's output and the buggy program's output, averaged over intervals of 10,000 instructions.

We found that the two executions follow similar execution path up until a point. After a point in the buggy program's execution, which is really the starting point of the source of the bug, the buggy program's execution starts to deviate from the correct program's execution. We define the source of the bug to be this deviation point in the buggy program's execution. In order to fix this bug, a developer needs to examine what happened at that deviation point and also the execution of the program after that point. Therefore, we define our replay window length to be the number of dynamic instructions executed in the buggy program between this deviation point and the end of the buggy program's execution (that is, the point of crash).

In order to find the deviation points for several bugs automatically, we used the following algorithm. For each bug, we collected the memory traces for the buggy program's execution as well as the correct program's execution. The

memory trace is a trace of store instructions' effective addresses and values. We then do a longest subsequence matching between these two traces to determine where they start to deviate focusing on the store values. Our algorithm is similar to the one used by the popular `vimdiff` utility that is used to compare the textual differences between any two files. Once we have matched between the two traces as much as possible, we then determine the deviation point of the buggy program's execution as follows. We compute the number of store output values that differ in the two executions for every interval of 10,000 instructions in the buggy program's execution. This plotted data looks like the graph shown in Figure III.2. We can find the point of deviation by finding the first point of inflection in the graph where the number of differences in store values exceed 30. The number of instructions executed after the deviation point till the end of buggy program's execution is the replay window length.

Note, our approach for calculating the replay window length is not useful for debugging the program. That is, it does not automatically identify the source of a bug in a program, as it assumes that we have the correct version of the program. It is only used to quantify a bound on the replay window length that lets us determine how much of a program's execution need to be logged.

Figure III.3 presents the replay window length required to analyze the bugs in the Siemen benchmark suite [38]. The y-axis shows the number of instructions from the point of deviation until either the program crashed or it terminated with a wrong result. Each point in the x-axis represents a bug examined for a benchmark suite. The number of bugs analyzed for each benchmarks are several 100s to 1000s. The result shows that the point of deviation (root cause) for a majority of bugs (inputs) occurred within the last 1 million instructions of execution.

We also studied the bugs found in some popular open source programs.

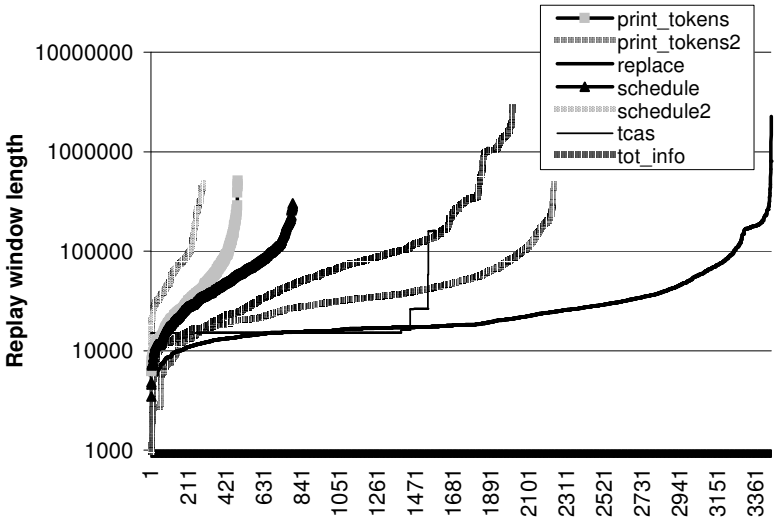

Figure III.3: Replay window length required to analyze the bugs in the Siemen benchmark suite.

Table III.1: Open source programs with known bugs. The first 5 programs are from the AccMon study [110], and the rest of the programs are from sourceforge.net

Application	Bug Location	Bug Description
bc 1.06	storage.c line 176	Misuse of bounds variable corrupts heap objects
gzip 1.2.4	gzip.c line 1009	1024 byte long input filename overflows global variable
ncompress- 4.2.4	compress42.c line 886	1024 byte long input filename corrupts stack return address
polymorph-0.4.0	polymorph.c lines 193, 200	2048 byte long input filename corrupts stack return address
tar 1.13.25	prepargs.c line 92	Incorrect loop bounds leads to heap object overflow
ghostscript-8.12	ttinterp.c line 5108, ttobjs.c line 279	A dangling pointer results in a memory corruption
gnuplot-3.7.1	pslatex.trm line 189	A buffer overflow corrupts the stack return address
tidy 34132	istack.c at line 31	Null pointer dereference
xv-3.10a	xvbrowse.c line 956, xvdir.c line 1200	A long file name results in a buffer overflow
napster-1.5.2	nap.c line 1391	Dangling pointer corrupts memory when resizing terminal

The bugs that we studied are listed in Table III.C.2. The second column in the table gives the details about the location in the source code of the applications that needed to be changed in order to fix the bug. The third column describes the nature of the bug. The set of bugs listed in the Table III.C.2 covers a large variety of bugs. It includes memory corruption bugs like dangling pointer accesses (`ghostscript`), buffer overflow (`gzip`) and null pointer dereferences (`gnuplot`).

Replay window lengths for these open source programs are again determined using the approach we described in Section III.C.2. Figure III.4 presents the replay window length required to analyze these real bugs. We can note that in the common case the length of the replay window is less than 10 million instructions. The worst case is `ghostscript` for which the replay window length is over 100 million instructions.

We now show the amount of log size (without compression) that needs to be collected in order to analyze the open source bugs in Figure III.5. For `ghostscript` we require about 10 MB of BugNet's log size to capture its replay window length of over 100 million instructions (The bug in `ghostscript` requires the longest replay window length).

Dynamic slicing [98, 109] is a powerful technique to ease the job of debugging. One can integrate dynamic slicing into a replayer, so that the programmer can choose to analyze the execution of only those instructions that produce the value for the instruction that resulted in a crash instead of examining the entire replay window. The second bar Figure III.4 shows the number of dynamic instructions that are on just the dynamic slice. Dynamic slicing results in about one-third reduction in the number of instructions that are required to be examined within the replay window to potentially fix the bug. We also study the number of memory locations touched by the programs within the replay window. Results for this study are shown in Figure III.6. This shows the number of unique

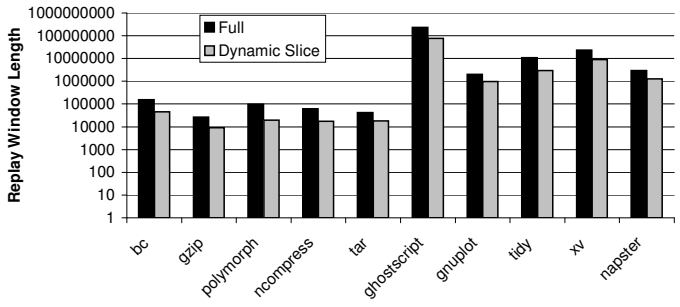

Figure III.4: Replay window length required to analyze the bugs in open source programs.

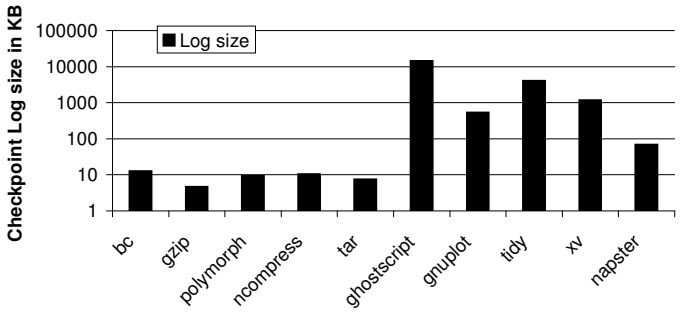

Figure III.5: Log size required to capture the replay window.

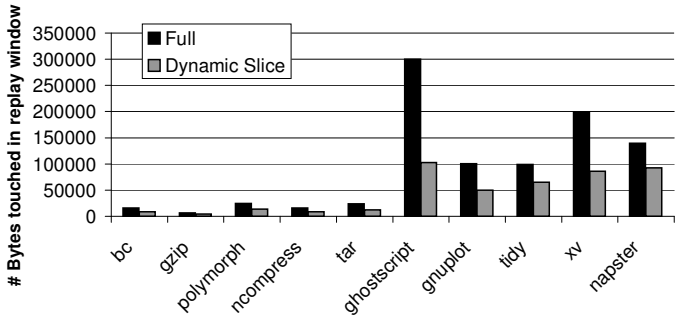

Figure III.6: Memory footprint touched within the replay window.

memory bytes touched in the replay windows for both the full replay window and just the dynamic slice. Using the dynamic slice can reduce the amount of memory locations that might need to be examined by about a half.

The replay window size is a knob that can be tuned by the operating system or the user. The replay window size is essentially dependent on the amount of main memory space that can be allocated for storing all of the FLLs and MRLs used to capture the desired number of instructions for replay. The user should be able to specify a desired replay window size and the maximum performance penalty that they are willing to pay for it. Based on this input, the operating system can dynamically tune the memory space allocation. If the OS can determine that the applications running at a particular instant of time are not memory intensive and that considerable amount of free space is available, then it can increase the memory space allocation to BugNet. On the other hand, if the performance degradation goes above the tolerable limits it can tune down the space allocated. In addition, the customer using the application can specify a minimal replay window size that can capture a majority of the bugs. In We showed that a replay window of size 10 million instructions is enough to capture the majority of bugs. If a bug occurred, but not enough state was kept to track down the bug, then the customer may be asked to increase the replay window size.

DRAM and disk sizes have seen exponential growth over the last few decades dramatically reducing the unit cost for storage. This is an encouraging trend for using BugNet feature to record executions. Introduction of Phase-Change RAM (PCRAM) [46] technology is likely to sustain this trend. Unlike, DRAM, PCRAM is a persistent-storage device. It is expected to be as fast as DRAM, but only 1/4th of DRAM's cost. Therefore, PCRAM could be used as a substitute for DRAM main memory or perhaps as another level in the

cache/memory/disk hierarchy. A good usage for the cheap space available in fast persistent-storage technology like PCRAM would be recording BugNet logs.

While using BugNet at the developer's site for debugging space is of a lesser concern as the developer can afford to allocate more disk space for recording a a test run.

III.C.3 Sensitivity Analysis

In this section we will discuss how the FLL sizes varies based upon the checkpoint interval lengths and the replay window lengths. Also, we will study the efficiency of the dictionary based compression algorithm that we discussed in Section III.B.3. For this study we use SPEC programs, since they have standard inputs, which are well analyzed.

Figure III.7 presents the FLL sizes collected for a replay window of 100 million instructions (the longest replay window length that we observed among all the open source bugs that we analyzed in Section III.C.2) using different checkpoint interval lengths ranging from 10K to 100 million instructions represented along the x-axis. Clearly, as the interval size increases, FLL sizes decrease. This is a result of applying our "first-load" optimization described in Section III.B.3. For longer checkpoint interval lengths, it is more probable that a particular memory location referenced by a load has already been recorded and hence the frequency of recording a load instruction decreases resulting in smaller FLL sizes.

Figure III.8 shows the sizes of FLLs that are needed to replay a window of 10 million to 1 billion instructions. For these results we assume a constant checkpoint interval length of 10 million instructions. On an average, FLLs of size 225 KB are required to replay 10 million instructions and about 18.86 MB for replaying 1 billion instructions.

The results presented so far assume a 64-entry dictionary table for com-

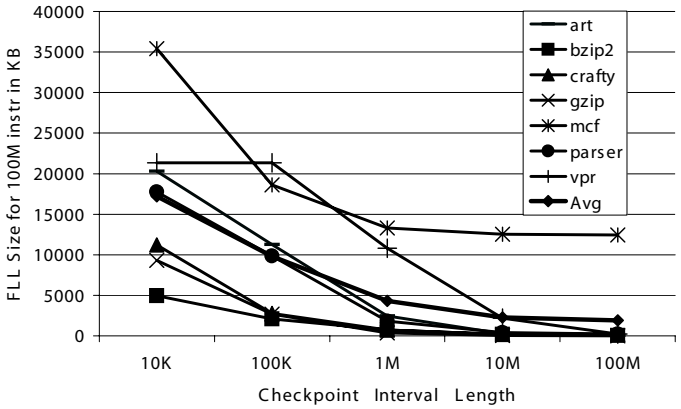

Figure III.7: Total size of FLLs required to replay 100 million instructions captured using different checkpoint interval lengths.

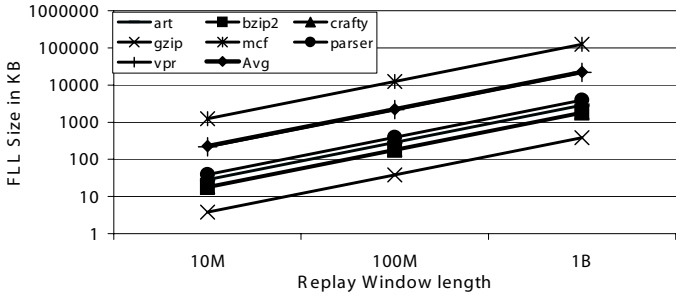

Figure III.8: Total size of FLLs required to replay a window of 10 million to 1 billion instructions. FLLs were collected using a 10 million checkpoint interval length.

pression. We will now discuss the efficiency of our compression technique described in the Section III.B.3. Figure III.9 shows the percentage of values that were compressible (found in the table) using our dictionary table approach varying the dictionary size. A dictionary of size 64 is capable of compressing 50% of the values on average, which is the size used for the rest of the results in this paper.

Figure III.10 shows the compression ratio of FLLs we achieve for various dictionary sizes. On average, we achieve about a 50% compression using a 64-entry dictionary. While a larger dictionary table results in higher compression ratio, it would increase the hardware costs, especially given that the dictionary table is fully associative.

Finally, we used SimpleScalar x86 [10] to examine the performance overhead of BugNet and found it to be less than 0.01% for these SPEC programs. We found for the SPEC programs that the overhead of BugNet is less than 0.01% due to (a) the fact that we use an incremental compression scheme that allows us to lazily write the compressed log entries to memory when the bus is free, and (b) the SPEC programs do not have a lot of interrupts or system calls.

III.C.4 Complexity of FDR Vs BugNet

FDR's proposal is to have the ability to replay the last 1 second of execution, which can be approximated to a replay window of length one billion instructions, which will vary depending on the processor speed and also the IPC of the program. For a fair comparison with FDR, we discuss using the BugNet architecture to also capture 1 billion instructions. But from the results shown in the Table III.C.2, a replay interval of 10 million instructions should be sufficient to fix many of the bugs in the applications we examined. We therefore also discuss using BugNet to generate logs to replay 10 million instructions.

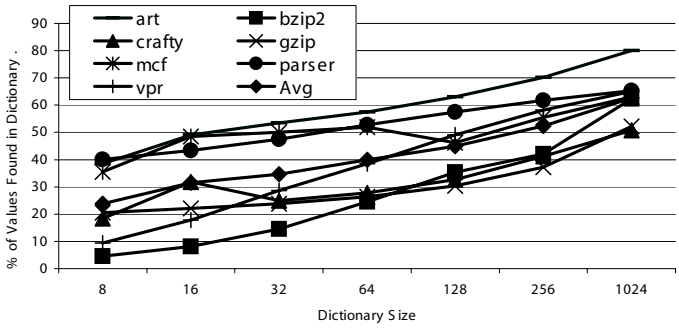

Figure III.9: Percentage of load values found in the dictionary table of various sizes.

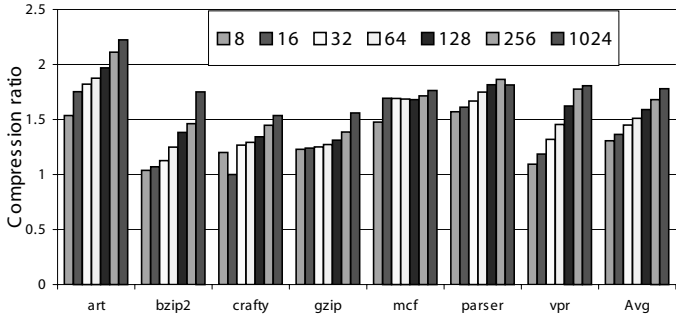

Figure III.10: Compression ratios achieved while compressing FLLs using different sizes for the dictionary table. The results are gathered using 10 million checkpoint interval length.

For the rest of this section all the results presented for BugNet assume a checkpoint interval of size 10 million instructions. Note, a checkpoint interval is different from a replay window. To replay a window of execution, we will use the logs from multiple checkpoints if the checkpoint interval length is less than the desired replay window length. Also on encountering an interrupt we terminate and create a new checkpoint as described in Section III.B.4.

Log Size Complexity

Table III.2 compares the sizes of BugNet and FDR logs. We compare the amount of memory storage required for replaying 10 million and 1 billion instructions in BugNet. The result for replaying 1 billion instructions in FDR corresponds to replaying one second of execution [104]. If an entry is NIL in the table, then it implies that the log is not used in the mechanism.

The FLL sizes required to replay 10 million and 1 billion instructions are about 225 KB and 18.86 MB on average for the SPEC applications. This assumes a checkpoint interval length of 10 million. In addition, to debug data races we will require memory race logs which will be discussed in Chapter IV.

FDR to support replaying 1 billion instructions would require 18 MB of cache and memory logs as described in [104], plus memory race logs of size 2 MB. The combined size of these is roughly the same as the sizes of using FLLs for capturing 1 billion instructions. However, FDR requires additional information to enable full system replay. FDR records Interrupt, I/O and DMA logs whose sizes may vary widely depending on the nature of the application. For I/O intensive applications, these logs might have prohibitive sizes. In addition, FDR requires a core dump image whose size can range up-to 1GB, based on the application's memory footprint and the main memory size.

Our results show that a log of size 225 KB can replay 10 million in-

Table III.2: Comparison of log sizes in FDR and BugNet. Interrupt, Program I/O and DMA log sizes will depend on the characteristic of the program. I/O intensive applications will require large sizes for these logs.

Log size BugNet Vs FDR			
	BugNet:10M	BugNet:1B	FDR:1B
FLL	225 KB	18.86 MB	NIL
Memory Race log	Strata IV	Strata IV	2 MB
Cache Chk-pnt Log	NIL	NIL	3 MB
Mem Chk-pnt log	NIL	NIL	15 MB
Core Dump	NIL	NIL	128MB-1 GB
Interrupt Log	NIL	NIL	Depends
Prg I/O Log	NIL	NIL	Depends
DMA Log	NIL	NIL	Depends

Table III.3: Comparison of hardware complexity in FDR and BugNet. For BugNet we consider support for replaying 10 million instructions as it is adequate to capture the replay window for most of the programs in Table III.C.2. Hardware complexity is also shown for BugNet to capture a 1 billion replay window, which is the window size assumed for FDR.

Hardware Complexity: BugNet Vs FDR			
	BugNet:10M	BugNet:1B	FDR:1B
CB	16 KB	16 KB	NIL
MRB	32 KB	32 KB	32 KB
Compression	64-entry CAM	64-entry CAM	LZ HW
Chk-pnt Interval	10M instr	10M instr	1/3 sec.
Cache Chk-pnt Buf	NIL	NIL	1024 KB
Mem Chk-pnt Buf	NIL	NIL	256 KB
Interrupt Buffer	NIL	NIL	64 KB
Input Buffer	NIL	NIL	8 KB
DMA Buffer	NIL	NIL	32 KB
Total HW Area	48 KB	48 KB	1416 KB

structions of an application's execution. This should be enough to reproduce and debug a majority of the bugs, at least for the program's we examined. In addition, BugNet's small traces (sometimes on the order of only hundreds of KB) should encourage users to communicate the logs back to the developer.

Hardware Complexity

Table III.3 compares the hardware complexity of BugNet and FDR [104]. Like in the previous section, here again we compare the configuration of BugNet to capture 10 million and 1 billion instructions.

The main hardware structures used in BugNet are the CB, MRB hardware buffers and a fully associative 64-entry dictionary table as shown in the Figure III.1. The size of the CB needs to be only large enough to tolerate bursts in our logging. In addition, we perform incremental compression of each log entry, which allows us to lazily write the logs into main memory and free up space in the CB. The sizes of the CB, MRB and dictionary table will be a constant irrespective of the length of replay window that we are trying to capture, since the logs are memory backed.

In comparison, FDR requires about 1416 KB of on-chip hardware to record enough information for full system replay. FDR assumes hardware implementation of LZ [112] compression. The LZ compressor is block-based, so the hardware buffer size needs to be large enough to collect a block of information before compressing and storing it back to main memory, and it also needs to be large enough to tolerate bursts. Cache and Memory checkpoint buffers are used to record information required by the SafetyNet checkpoint mechanism, whose sizes are 1 MB and 256 KB respectively. Since FDR aims to achieve full system replay it has to record all the external inputs for which it requires three additional buffers - 64 KB interrupt buffer to record interrupts, 8 KB input buffer to record

program I/O and a 32 KB DMA buffer to record DMA writes. In summary, the total on-chip hardware requirement for BugNet is about 48 KB, whereas FDR requires 1416 KB.

III.D BugNet Extensions for Handling Self-Modifying Code and Frequent Interrupts

In this section we present three extensions to the BugNet architecture described in Section III.B. First, we describe a few issues with the baseline BugNet approach in reproducing code regions, and discuss how by logging code regions we can handle self-modifying codes easily. Second, we examine the effectiveness of the baseline BugNet logging approach in the presence of interrupts and system calls, and present a solution to allow efficient logging in the presence of frequent interrupts and system calls. Finally, we discuss how BugNet can be extended to record and deterministically replay operating system code as well.

III.D.1 Extending BugNet to Record Code Regions

In this section we describe a few issues with the BugNet approach in dealing with reproducing the code regions, and discuss solutions to address those issues.

BugNet's Approach to Logging Code Locations

In addition to data, BugNet also needs to make sure that information about the code executed during logging is recorded. In Section III.B, we assumed the support of an operating system device driver that records such information into a *Code Log*. Code Log contains information about the loading of all code (static binaries and dynamic libraries) for a program being monitored. Each

entry in the code log contains (a) the name and path of the binary or library loaded, (b) a checksum to represent the version of the binary/library, and (c) the starting address where it was loaded. This information is required during replay. A replayer would load the same exact code into the same exact location as occurred during logging, which is needed to provide deterministic replaying.

When BugNet is enabled for a program's execution, the device driver logs the above information for the binary and shared libraries currently loaded. In addition, as new shared libraries are loaded, they will be logged. The code log is kept as long as the program's execution is monitored with BugNet.

Advantages of Logging Code

There are two disadvantages if we chose to assume that the developer has access to the binaries to perform deterministic replay debugging.

The first issue is that it does not support replaying of self modifying code. In the BugNet architecture, only up to the last second of execution before the crash would be available for replaying. Therefore, if code was dynamically generated before that, there would be no record of this new code in the log even if the dynamically generated code was used during the checkpoint interval. If you tried to use logs with the original binaries, incorrect execution would of course occur. To address this we examine logging the code as well as data.

The second issue in only having the original BugNet Code Log is that during debugging it requires having the exact same set of system and shared libraries available to replay execution. If the developer is only interested in tracing through the application level code, they would still have to find and set up the exact same shared library environment in which the bug occurred. Logging code provides an important advantage here, in that it allows deterministic replaying without having any access to any of the original binaries or the shared libraries.

Therefore, a developer needs to have access to the source code of only the binaries and the shared libraries that they want to actually examine during debugging.

Logging Code to Support Self Modifying Code

Code regions can be logged similar to the data regions. When an instruction is first accessed in a checkpoint, the instruction is logged and the first-load bit is set for that instruction in the instruction cache. If the instruction is accessed again and the bit is set, then it does not need to be logged. The first-load bits are reset in exactly the same way they are reset for data, and at the L2 cache level, the data and code blocks are of course treated the same.

When logging the instruction, we log (1) the number of instructions executed since the last instruction was logged, and (2) the instruction word. The PC is not logged, since it will be deterministically regenerated during replay. The instruction difference is used to indicate exactly which dynamic instruction needs to consume the logged instruction bits during replay. While creating a log, since we do not log every instruction, in addition to the instruction we also need to log the number of instructions that were skipped between the last log entry and the current log entry. Instead of using the instruction counts, we use branch counts which serves the same purpose as instruction counts, but is more efficient to keep track of branch counts during replay.

If a code region was modified in a self-modifying program before a checkpoint, the modified code would be logged as part of the checkpoint code log when it is accessed for the first time in the checkpoint interval. If there is any code generated during the checkpoint, then it will be correctly stored to memory during replay and consumed as code during replay.

Code Logging Results

The first bar in Figure III.12 shows the log size for the BugNet approach described earlier in Section III.B. The y-axis shows the number of megabytes of log required to record 1 million instructions. This result assumes a checkpoint interval of length 1 million instructions. The second bar shows the log size for logging both the code and the data for SPEC programs. Compared to the size of the data logs, the amount of extra logging required is negligible for the SPEC programs. The reason for this is that these programs have good code locality and spend most of their execution in a few loops. Thus, resulting in a high cache hit ratios, and therefore small log sizes required to capture the code. The same is not true for the interactive programs we examined, shown in Figure III.13. For these programs we see an average overhead of 39% for logging the code when compared to baseline BugNet. We next show how we significantly reduce the log size overhead for interactive applications by applying our interrupt optimization.

III.D.2 Efficient Logging Across Interrupts

This sub-section first motivates the need for providing efficient support for handling system calls and interrupts. Then it discusses an extension to BugNet architecture described in Section III.B to efficiently log a program's execution even in the presence of frequent interrupts and system calls.

Frequency of Interrupts

BugNet architecture described in Section III.B addresses the problem of logging the values in the application that are changed during an interrupt or a system call by prematurely terminating the checkpoint interval when an interrupt or system call is invoked. When a new checkpoint is created, all the first-load bits in the cache of the processor executing the application is reset. This ensures that

any memory address modified by the interrupt service routine will be re-logged when the application reads it later in the execution.

This works fine for computationally intensive programs, which do not have a lot of interrupts/system calls. For the SPEC programs we examined, 8 of the programs had *zero* system calls during the 5 billion instructions of execution we examined, and 3 programs (`equake`, `gcc`, and `vortex`) had the most number of system calls with 1 system call being invoked every 1 million instructions.

Figure III.11 shows the number of instructions between system calls and interrupts on the y-axis that were executed by user interactive programs. The figures shows that an interrupt occurred on average every 20,000 instructions. Restarting a checkpoint on every interrupt creates a significant amount of logging overhead for BugNet, because it will not be able to benefit as much from the first-load optimization.

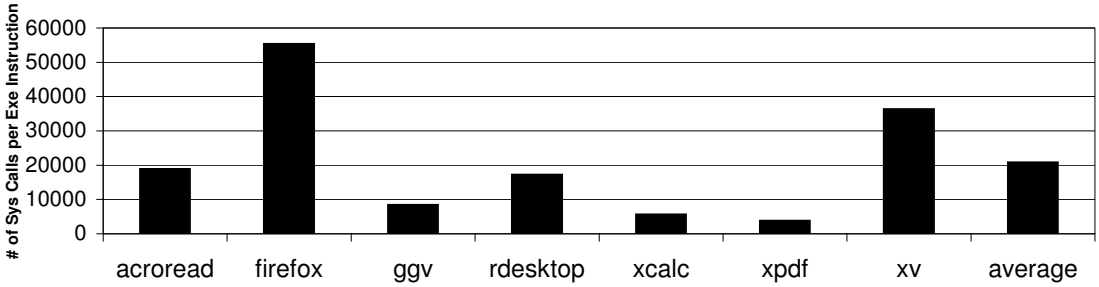

Figure III.11: Number of instructions executed between two system calls/interrupts for the interactive programs.

BugNet Architecture Extension for Handling Frequent Interrupts

Current architectures allow data to stay cached across system calls and interrupts to benefit from the data locality across quick system events. Therefore, a lot of the first-load information should still be in the cache between interrupts, which we should be able to exploit to reduce the BugNet log sizes. This is espe-

cially necessary for the interactive applications with frequent interrupts shown in Figure III.11.

To address the problem of frequent interrupts, we propose to have the architecture assign the first-load bits for a cache to a specific thread ID. Therefore, a set of first-load log bits for a cache would be owned by a specific thread ID. Whenever that thread accesses a cache block it uses the first-load bit as we described in Section III.B to filter the amount of logging for load accesses.

When BugNet is invoked to monitor a process, the OS knows the set of threads for which it needs to record the logs. When we assign one of these threads to a processor node, if its thread ID is different than the one that currently owns the first-load bits, then the first load-bits are cleared. Then this new thread becomes the owner of those bits. If any other thread that is not being monitored by BugNet is scheduled on that processor node, then the architecture will reset the first-load bit only when a block is evicted by this other thread or when this other thread writes to an existing block.

In addition, operating system support is used to note when a thread is running in user mode, versus in the operating system. When running in the system mode, any writes that hit in the cache will also clear that block's first-load bit. This allows first-load bit information to be used across the interrupts as long as the data stays in the cache unmodified for the thread ID that the bits are currently assigned to.

Supporting Multiple Threads with First-Load for a Cache : The architecture described so far provides support for assigning one thread ID to a given set of load bits maintained in a cache. This can easily be extended to support multiple threads that are being monitored, where the hardware would support up to N sets of load bits. Each set of load bits would be assigned to a thread being monitored. Any write hit by a thread would clear all of the other

load bits for that cache block. When a new thread to be monitored is assigned a set of first-load bits, all of those bits are cleared. This could be useful when multiple threads are sharing the same cache (for example, a shared L2 cache in a multi-core processor). But, for our results we model having only one set of load bits per cache.

Interrupt Logging Results

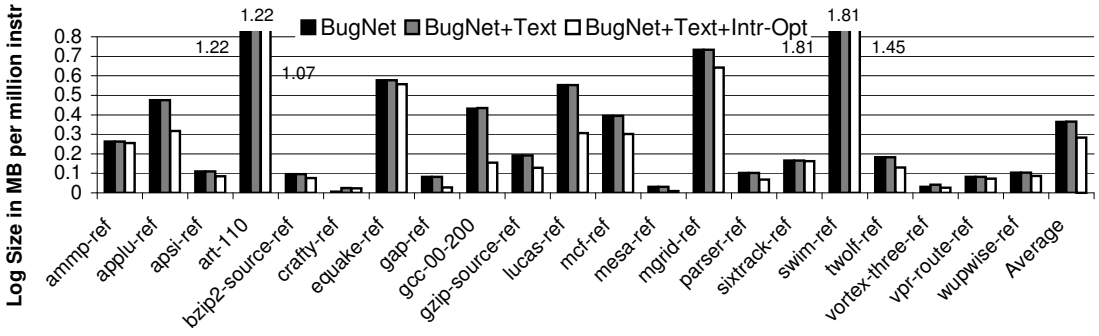

Figure III.12: Log size comparison for SPEC programs.

The third bar of figures III.12 and III.13 show the results of applying our interrupt optimizations on the log sizes for SPEC and interactive programs respectively. For the SPEC programs, the log sizes are reduced by 23% when compared to logging the code and text and resetting the FLL bits after each interrupt and system call. The impact is even greater for the interactive programs due to the large number of interrupts present in their execution. An average of 44% in log size reduction is obtained when applying the optimization. The reduction is proportional to the number of system calls present in each program. Note that after applying our two optimizations, the log size of both code and data combined is smaller than the log size of the baseline BugNet architecture that only logs the data.

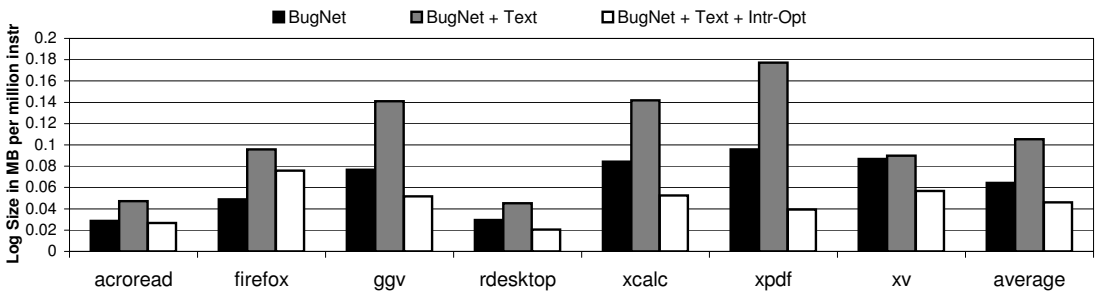

Figure III.13: Log size comparison for interrupt intensive programs.

III.D.3 Extending BugNet for Enabling Deterministic Replay of Operating System Code

BugNet's checkpointing and logging solution described in Section III.B can deterministically replay only the user code and the shared libraries. However, it can be be extended in future to record and replay operating system and driver code as well to overcome this limitation. Section III.D.1 discussed the benefits of logging the code regions. If we capture the code regions, then the system code can also be easily recorded using the following approach.

When there is an interrupt or a system call invoked, in addition to terminating the checkpoint of the application, a new checkpoint would be created to capture the execution of the service routine. That checkpoint will have the FLL and the code log necessary for replaying the service routine. Similarly, whenever an operating system routine gets context switched in, a new checkpoint will be created for that routine. The header of the FLL has the timestamps that orders all the checkpoint logs. Thus, the application code and the operating system routines can be replayed in the same order as the original execution.

Operating system routines might execute privileged instructions that read the machine state. For example, `rdtsc` instruction reads the processor clock and writes it to the register. Care must be taken to log the register values written by all such non-deterministic instructions in addition to the first-load logs and

the code log.

III.E Using BugNet for Deterministic Replay Debugging

This section first discusses how and when a program's execution would be recorded using BugNet. It is followed by a discussion on the BugNet replayer that we implemented using the Pin instrumentation tool [49]. Finally, this section discusses how the BugNet deterministic replayer can be used for debugging a program.

III.E.1 Collecting BugNet Logs

To report a failure that a customer encounters in their environment to the developer, today, the customer has to file a bug report. A bug report typically contains a verbal description of the actions that were taken by the user before the failure along with the final core dump. Microsoft's Dr. Watson tool [53] and Mozilla's Talkback [63] are examples of the current solutions to automatically generate these bug reports.

However, it is very difficult for a developer to reproduce the bug specified in a bug report. The developer has to set up a system environment (operating system, shared libraries etc.) that replicates that of the customer's environment. In particular, non-deterministic bugs (like the data races) are very difficult to reproduce. Using BugNet, however, a customer can record the buggy execution and send the BugNet logs to the developer. Customers can afford to record an execution in their environment, because the BugNet recorder is efficient both in terms of space and performance overhead. In Section III.C we showed that with about 225KB of FLL we can record about 10 million instructions on average. Also, the performance overhead of the processor-based BugNet solution is negligible (less than 1%). Thus, BugNet is useful for recording even the production

runs, and so we can capture the bugs that manifest at the customer site. These days software vendors do a limited release of their software to a selected group of users for beta-testing. Beta-testers could turn on the BugNet feature to record their test runs. On encountering a software failure, the operating system would send the recorded BugNet logs to the developer. The developer would use the BugNet logs to deterministically replay the buggy execution and debug it.

In addition to the customers, developers too would find BugNet useful for recording their test runs during the development of a program. They could record an execution of their test run using BugNet and use the BugNet log for debugging. Recording a program's execution using a software-based solution like iDNA [6] (iDNA implements BugNet-like checkpointing and logging solution using an instrumentation tool, which can record multi-threaded program's even on multi-processor systems) would be 5 to 15 times slower than the native execution. Because of the high performance overhead, a software-based recorder like iDNA cannot capture a natural behavior of a program's execution (especially interactive applications like Internet Explorer) on a real system. The processor-based BugNet support, however, incurs negligible performance overhead. Therefore, BugNet would be useful to developers for recording a program's execution without altering their behavior on a real system.

In the following sections, we describe an implementation of the BugNet replayer that can replay a program's execution using the BugNet logs collected by customers and developers. We also discuss how the developers can use the BugNet replayer for debugging.

III.E.2 BugNet Replayer

We first describe an implementation of the BugNet replayer. In the next sub-section we describe how it can be used for debugging. We implemented a

prototype of a replayer as a proof of concept for BugNet using the Pin dynamic instrumentation tool [49]. For our study, we collected the BugNet logs for an interval of a program's execution using the Pin Dynamic Instrumentation tool [49]. Now we describe the implementation of our BugNet replayer to deterministically replay a checkpoint interval.

To replay a checkpoint interval using BugNet's FLL and code log, the replayer (which is a dynamic instrumentation tool built using Pin) first reserves the region of memory that was read/written by the application's data and code regions during logging. The replayer keeps track of two counts - an instruction count (number of instructions replayed) and a load count (number of load instructions replayed). The instruction count is initialized with the counter value read from the first entry in the code log, and the load count is initialized with the counter value read from the first entry in the FLL.

The replayer then uses the header information in the FLL to initialize the register values and the program counter. This is done by using the ExecuteAt() API of Pin. The ExecuteAT() API of Pin starts the program's execution from the instruction specified by the program counter. Once replay starts, all the instructions replayed updates the register and the memory state of the application's execution state just like in a normal execution. Thus, the replayer deterministically replays a sequence of instructions.

The replay breaks before the execution of every load instruction and every branch instruction to keep track of the load count and the instruction count respectively. When the instruction count matches the instruction stride specified in the next entry of the code log, the instruction code from the log entry is copied into the application's memory space. The address for the application's memory space is specified by the current program counter value. The instruction count maintained by the replayer is then reset and the counter starts again from

zero. Similarly, while replaying a load instruction, if the replayer's load count matches the load count of the next entry in the FLL, the value from FLL is copied to the application's memory space. The address for the application's memory space would be the effective address of the load instruction. Effective address computation is part of any load instruction's executions. Therefore, the replayer would be able to compute the effective address for the load instruction, because the replayer reproduces the input values for the effective address computation just like it reproduces the input values for executing any other instruction.

To retrieve the load value from the FLL, the replayer has to decompress the log entry in the FLL. The format of an FLL log entry was described in Section III.B.3. In Section III.B.3, we described a dictionary based compression for compressing the load values recorded in FLL. Using the bit $LC - type$, the replayer determines if the recorded value should be retrieved from the FLL itself or from the dictionary.

If the load value is to come from the log entry (that is, the $LV - Type$ bit is set), we use the next full 32-bit value in the log entry. If not, the next 6-bits in the log entry specifies an index to the dictionary. To generate the correct values in the dictionary table during replay, the replayer updates the dictionary with the value of every replayed load instruction exactly in the same way as it would have been updated during logging (explained in Section III.B.3). Using the index read from the FLL entry, the corresponding dictionary entry is then read and that value read is returned for the load.

During replay, all the interrupts (including system calls) are turned into NOPs, since we need not simulate what goes on during an interrupt. To replay past the interrupt, we just continue replaying the next checkpoint interval recorded for the thread's execution. If the replay reaches the end of a checkpoint interval, then replayer starts replaying from the checkpoint log for the following

interval.

The replayer can also replay a thread of a multi-threaded program's execution using the same algorithm described above. The FLL and the code log collected for an interval of a thread's execution has sufficient information to replay that interval of the thread's execution. However, to debug multi-threaded programs we also need to be able to retrieve a valid sequential order of memory operations across all the threads. This is possible using the information logged in the MRLs. Chapter IV presents a solution to record the memory order in MRL and it also discusses how to replay the order between memory operations using the log.

III.E.3 Using BugNet Replayer for Debugging

The BugNet replayer can deterministically replay a program's execution using the BugNet logs recorded at a customer's site or at a developer's site. In this section, we discuss how the BugNet replayer can be used for debugging.

Time Travel Debugging Using BugNet Replayer

The BugNet deterministic replayer is useful for building a time travel debugger [6, 42, 7] that we described in Section II.B. Using a time travel debugger, a programmer can go to any point in a program's execution time within the replay window by stepping forward/backwards, using breakpoints, reverse breakpoints, watchpoints and reverse watchpoints.

After reaching a particular point in program's execution, the programmer can examine the execution state (memory and register states) of the program at that point. Note that BugNet logs do not contain a core dump containing the final state of the entire system/main memory. As a result, the replayer cannot construct the complete state of the memory state. Therefore, the programmer

would be able to examine only those memory state that are either read or written within the recorded replay window length of the program's execution. This might cause slight inconvenience for the programmer in inferring the cause of a bug. However, it would not prevent the user from isolating the bug, because we expect that the memory addresses untouched by the program's execution preceding the crash are most likely not to be responsible for the buggy behavior. That said, optionally, the BugNet system can be made to collect the core dumps as well. The core dump the values of all the memory locations, including those untouched within the recorded replay window. Using this information, the BugNet replayer can construct the entire state of the memory at any point within the replay window length.

Limitations of BugNet based Time Travel Debugging

There are few limitations of BugNet, however. First, the programmer can examine the program's state only at an instant within the replay window length, but not before that. As we showed in Section III.C, replay window length of 10 million instructions is sufficient for debugging a majority of the bugs. While using BugNet during testing and development, however, programmers can afford to capture potentially the entire execution of a program, which would overcome this limitation.

The second limitation of the BugNet checkpoint and logging approach described in Section III.B is that it enables replay of only the user code and the shared libraries, but not the full system. This is sufficient for debugging the user code that does not have complex interactions with the operating system routines like the drivers and the interrupt handlers. But, BugNet would not be useful to debug the drivers or the operating system, or the complex interactions between these and the user code.

Even though BugNet cannot replay the system code, it still provides deterministic replay of a program's execution before and after servicing interrupts and context switches. Hence, the user can examine the values of the parameters passed to an interrupt, and the modified values loaded and consumed after servicing the interrupt. This can allow the user to debug some bugs that are dependent on operating system interactions. Also, we replay all of the operating system shared library code. The user code along with the OS library code comprise a significant portion of a program's execution, and therefore BugNet logs should be sufficient to track down a majority of the application-level bugs. However, it is possible to support deterministic replay of the operating system code as well using the BugNet extension we described in Section III.D.3.

The third limitation that BugNet cannot detect a bug that produces incorrect results. It can detect a bug only when the operating system or the application itself can identify that the program has encountered a fault or exception (eg: floating point exception, assertion failure, invalid memory access, etc.). As a result, some of the bugs producing incorrect results might not be captured at the customer site. Detecting a bug at the customer site is necessary, because only the log for last 1 sec or so of a program's execution is recorded at the customer site. Since BugNet cannot detect a bug producing incorrect results, it does not know the right time to dump the logs to a persistent storage and capture the replay window necessary to capture such a bug. Future work could address this problem to automatically detect bugs that cause the program to produce incorrect results.

Automatic Debugging Using BugNet Replayer

Section II.B described how a deterministic replayer is useful for analyzing a program's execution offline. A BugNet's replayer can be used to replay a program's execution. During replay, the program's execution can be analyzed

to automatically find bugs such as memory leaks, uninitialized variables, buffer overflows etc. Chapter V presents a novel dynamic analysis tool built on top on a deterministic replayer to automatically find data race bugs in multi-threaded programs.

III.F Summary

BugNet is a checkpointing and logging solution that captures the non-deterministic system input read during an application's execution. BugNet focuses on replaying only the user code and the shared libraries to find application level bugs. To achieve this, BugNet's logs for a checkpoint interval contain the register state at the start of the interval and a log of memory values (code and data) when they are first accessed. This is enough information to achieve deterministic replay of a program's execution, without having to replay what goes on during interrupts and system calls. This results in small log sizes - FLL size of around 225KB is enough to capture a replay window size of 10 million instructions. We also showed that 10 million instructions are sufficient for debugging a majority of the bugs in the open source programs. We found that BugNet incurs performance overhead, and the area overhead is around 48 KB for the hardware buffers required to record the log. More importantly, BugNet is a system-independent solution. If supported in a processor, the BugNet feature can be used to record an application's execution on any operating system. Also, the logs collected for a program's execution on a particular system can be used to replay that program's execution on any other system.

Acknowledgments

The text in this chapter is in part a reprint of the material from the paper, Satish Narayanasamy, Gilles Pokam and Brad Calder, "BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging", in the *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, June 2005. The dissertation author was the primary researcher and author and the co-authors involved in the publication [60] directed, supervised, and assisted in the research which forms the basis for that material.

The text in this chapter in in part a reprint of the material from the paper, Satish Narayanasamy, Cristiano Pereira and Brad Calder, "Software Profiling for Deterministic Replay Debugging of User Code", in the *5th International Conference on Software Methodologies, Tools and Techniques (SoMET)*. The dissertation author was the primary investigator and author of this paper.

The author also would like to thank Professor Yuanyuan Zhou and her students for providing a set of programs with known bugs and inputs that expose them.

IV

Strata: Deterministic Replay of a Multi-Processor System

In Chapter III we presented the BugNet checkpoint and logging mechanism for supporting deterministic replay of a program's execution in the presence of all forms of non-determinism, including system interactions (system calls, interrupts, memory mapped I/O, DMA, reading processor clocks, and context switches). BugNet is useful for recording a shared-memory multi-threaded program as well. BugNet records each thread separately, and if a thread reads a value written by another thread it records them (refer Section III.B.6). Thus, using BugNet we can deterministically replay each thread in a shared-memory multi-threaded program.

However, to debug a multi-threaded program, a programmer would like to replay the order between the memory operations executed by all the threads. This is necessary for understanding non-deterministic bugs like data races in multi-threaded programs. Thus, to make a deterministic replayer complete, in addition to BugNet we need an ability to replay the memory order, which is the focus of this chapter.

One of the first hardware support for deterministic replay of a shared-memory multi-processor system was proposed by Bacon and Goldstein [5]. Their design was for a bus based system. They observed that memory order can be captured by recording the coherence messages on the bus. However, they recorded all the coherence traffic on the bus, which can result in a large log size. Also, their system cannot handle non-determinism due to the system interactions.

The amount of information that needs to be logged to record the memory access ordering can be reduced by applying the Netzer's transitive optimization [64]. Flight Data Recorder (FDR) [104] is a recent hardware proposal that implements the Netzer transitive optimization in a directory based multi-processor system with a sequentially consistent memory model.

FDR [104] logs the shared memory dependencies in a *Memory Race Log* (MRL), which is maintained for each processor node. FDR uses Bacon and Goldstein [5]'s observation to detect the shared memory dependencies for a thread by monitoring its coherence messages. In order to reduce the size of the memory race logs, FDR proposed a hardware design for supporting the Netzer optimization [64]. We refer to the logging method used by FDR as the *point-to-point* logging approach, because to capture a dependency, it logs the instruction counts of both the dependent operations.

In this chapter, we propose capturing the shared memory dependencies using *Strata* [57]. A stratum is logged when a shared memory dependency needs to be captured. It consists of the memory counts of all the threads at the time when it is logged. A stratum separates all the memory operations that were executed in all the threads before the time when it is recorded, from those that will be executed after it is recorded. Since the stratum is recorded just before the execution of the dependent memory operation, the stratum separates that memory operation from the earlier memory operation in which it is dependent

on.

The benefits of using strata are (1) it enables us to design a hardware solution for logging shared memory dependencies in both snoop-based and directory based systems, whereas the previous point-to-point logging solution only supported directory based systems, (2) the strata logging approach does not require us to log the shared memory write-after-read (WAR) dependencies, which can be determined during replay, (3) a single stratum can capture many different dependencies and as a result the strata logging approach reduces the number of memory dependencies logged even more than the prior Netzer optimization for point-to-point logging solution, and (4) the hardware required to create the strata log is smaller than what is required for implementing the point-to-point logging solution [104].

This chapter is organized as follows. Section IV.A provides a brief description of the FDR's point-to-point logging approach [104]. This is our baseline for comparison. Section IV.B introduces Strata and discusses the algorithmic aspects of using strata for capturing the shared-memory dependencies. This includes a transitive optimization to reduce the size of the strata logs. Section IV.C presents a hardware design for recording strata in a snoop-based multi-processor system, while Section IV.D provides a solution for a directory-based system. Section IV.E evaluates the proposed strata-based hardware design and Section IV.F concludes this chapter.

IV.A Baseline: The Point to Point Approach to Log Shared Memory Dependences

To replay and debug multi-threaded applications we need to record the memory dependencies that exist across all the threads. To accomplish this, the prior techniques used the point-to-point logging approach proposed by Flight

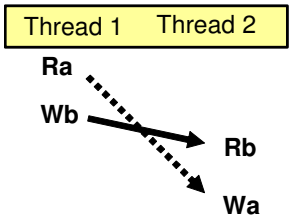

Figure IV.1: Netzer Transitive Optimization [64].

Data Recorder(FDR) [104]. In this section, we discuss FDR's algorithm and its hardware design to capture the shared-memory dependencies.

IV.A.1 Point-to-Point Logging and Netzer Optimization

FDR captures all forms of shared-memory dependencies: read-after-write (RAW), write-after-write (WAW), and write-after-read (WAR) dependencies. These dependencies are logged in the *Memory Race Logs*. Logging every dependency seen during execution is impractical, because it would lead to unmanageable memory race log sizes.

In order to reduce the log sizes, FDR implemented the Netzer optimization [64] in hardware. FDR's hardware design assumed a directory-based system with a sequentially consistent memory model. We briefly explain the Netzer algorithm [64] used by FDR here with an example.

The Netzer algorithm works by exploiting the transitive property in a system that assumes sequential consistency. A simple example for two threads, T1 and T2, is shown in Figure IV.1, where each thread executes a write and a read. The subscripts represent the address locations.

FDR records the dependency $Wb \rightarrow Rb$ in a memory race log. The dependency between the two threads is recorded using the two instruction counts of the dependent threads, which is a form of time-stamp used in FDR. This is

sufficient because all that we need to know while replaying is that T1 should have been executed *at least* until the memory operation *Wb*, before T2 can execute its memory operation *Rb*. Later, when we observe the second dependency $Ra \rightarrow Wa$ between T1 and T2, it does not have to be recorded, because it is transitively implied by the previously recorded dependency.

We call the logging approach used in FDR as the *point-to-point* logging approach, because each dependency is logged by explicitly logging the instruction counts of the two dependent memory operations executed in two different threads.

IV.A.2 Hardware support for Point-to-Point Logging

In order to capture all the shared-memory dependencies between the threads, we should first be able to detect them when the program is executing. FDR [104] records shared-memory dependencies between the processor nodes (and not the threads). We believe that this is sufficient information, because we can map the recorded dependencies between processor nodes back to the threads during replay. This requires that we know which thread was executing on a processor node at a given time. This is required in BugNet, as it can replay only user level code, and hence cannot reproduce the thread scheduling orchestrated by the operating system.

Intra-node dependencies (dependencies within the same processor node) need not be logged, because they are trivially revealed by the program order. To detect dependencies between the processor nodes, FDR uses an observation [5] that the shared-memory dependencies are revealed by the coherence messages in a multi-processor system. There can be a cross-node shared-memory dependency (dependency between two different processor nodes), only when a processor node encounters a read/write cache miss to a cache block. If there are processor nodes in the system that have a read/write permission to the cache block, then the

appropriate dependency (RAW or WAW or WAR) with those processor nodes is detected. If none of the processor nodes in the system have read/write permission for the memory block (in other words, the block is not cached anywhere), then the directory entry would have the information about the last writer to the memory block. If the last writer is different from the processor node that generated the read/write miss, then a cross-node dependency is detected. In the MESI directory protocol, when a clean block is evicted, the directory is not informed (silent eviction). The directory entry would continue to hold information about the readers in the system (sharers) for the cache block, until a processor executes a write to that cache block. At that instant, the cross-node WAR dependencies are detected between the current writer and the past readers (including those that silently evicted the cache block sometime in the past).

However, for a system based on a snoopy protocol, there is no directory to hold the last writer information and the list of readers for the blocks that have been evicted from the cache. Hence, additional support is required in snoop-based systems, which this is not addressed in the prior proposals [104]. In Section IV.C, we present a hardware design to record strata for capturing the shared-memory dependencies.

For a directory based system, there is still a corner case, which was not addressed in the prior FDR and BugNet proposals [104]; one that is related to paging. If a physical page is swapped out, then future accesses to that page would not find correct shared-memory dependencies, because the directory does not hold information for the swapped out page. As a result, information about the processor nodes that last accessed the swapped out memory blocks is lost. This is not a problem in our strata logging approach, which is explained in detail in Section IV.B.3.

Hardware support for Implementing Netzer Algorithm

Going back to our example, assume that FDR [104], by observing the cache coherence messages, detects the dependency $Wb \rightarrow Rb$ between the two processors P1 and P2 executing the threads T1 and T2 respectively (the threads are shown in Figure IV.1). This dependency is recorded in the *Memory Race Log* in the processor node P2 in which T2 is running. The dependency is recorded using the instruction counts corresponding to the memory operations Rb and Wb.

When the second dependency between the processors P1 and P2 is detected due to the dependency $Ra \rightarrow Wa$, FDR needs to determine, if the dependency can be transitively implied by the previous log entry or if has to be logged again in the P2's memory race log. In order to do so, P2 needs to know that the instruction count of the previous write access to the location "a" in the processor P1 is less than the instruction count that was last recorded in P2 for the processor P1 .

Thus, to implement the Netzer optimization for point-to-point logging approach, the time-stamp information (instruction count) is kept track of along with each cache block. The instruction count of a cache block tells the logging mechanism, *when* the block was last accessed by the processor node. To keep track of this information, about 6.25% [104] of L1 and L2 cache area is required, which translates to about 128KB area overhead for a 2MB L2 cache. Further, the memory race log is buffered locally in a 32KB Memory Race Log Buffer in each processor node as described in FDR [104]. The hardware implementation of the strata logging approach is less complex, and also the strata log size is 5.8x smaller without compression and 12x with compression than that of the memory race log.

IV.B Using Strata to Determine Shared Memory Dependencies

In this section, we discuss an algorithm to capture the shared memory dependencies across the threads of an application. The hardware implementation of the algorithm described here is presented later in Sections IV.C and IV.D.

IV.B.1 Capturing Shared Memory Dependencies using Strata

We assume a sequentially consistent memory model. In a sequentially consistent memory model there exists a total order between the memory operations executed across all the threads. All the threads' memory operations should be consistent with that total order, which means a thread's read must get the value of the most recent write in the total order. The total order must also respect a thread's program order.

Our goal is to record sufficient information during a program's execution, which would allow us to replay the total order observed during a program's execution.

To capture a dependency between two shared memory operations, in the point-to-point logging approach that we discussed in Section IV.A, the memory counts of the two dependent memory operations are logged. Instead, we propose using a logging primitive called a *stratum*. A stratum consists of the execution states in terms of the memory counts of all the running threads at the time when it is recorded. A memory count for a processor node is the number of memory operations executed since the beginning of the checkpoint interval. To capture a shared memory dependency, we record a stratum just before executing the succeeding memory operation. If two memory operations are dependent on each other, we refer to the memory operation that occurred earlier in time as the preceding memory operation or simply the *predecessor*. The one that occurred

later in time is referred to as the *successor*. The recorded stratum separates all the memory operations across all the threads, executed before the time when the stratum was recorded, from those that were executed after it was recorded. Since the stratum is recorded just before the succeeding memory operation is executed, it separates the predecessor and the successor in time (that is, it captures the dependency between the two dependent operations).

Figure IV.2 shows the memory operations executed in three threads. The subscripts of the read and write memory operations identify them. The fields inside the braces, show the address and the output value for a memory operation. The strata are represented as horizontal lines. For instance, strata $S1$ separates the successor $W2$ from the predecessor $W1$.

One advantage of using the strata to capture the shared memory dependencies is that, we can apply an effective dynamic transitive optimization to reduce the size of the strata log. Also, the hardware required to implement the transitive optimization for strata is significantly less than what is required for implementing a similar optimization for the point-to-point logging approach [104].

We further reduce the strata log size by not logging information for the WAR dependencies. This is based on our following observation: To reproduce the total order during replay, it is sufficient to capture just the cross-thread RAW and WAW dependencies using strata. We show how the cross-thread WAR dependencies can be determined through an offline analysis during replay. We also do not have to record strata to capture the intra-thread RAW and WAW dependencies, because those are trivially revealed by a thread's program order. Therefore, the discussion in this section focuses on capturing only the cross-thread (inter-thread) RAW and WAW dependencies using strata.

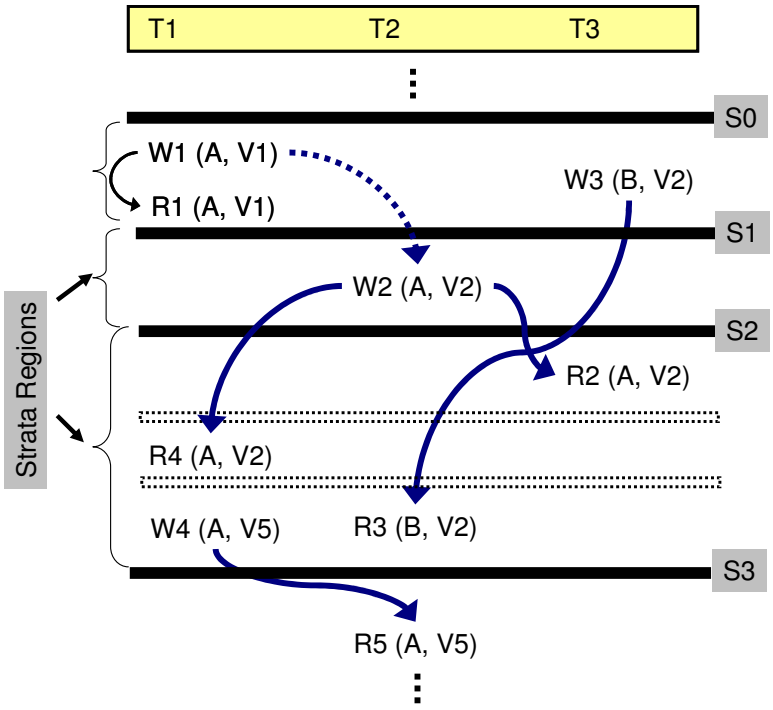

Figure IV.2: Recording Strata Log. The strata that are logged are shown as solid horizontal lines. The strata that are not logged by applying the transitive optimization are shown with dotted rectangular boxes. The RAW dependencies are shown as solid arrows and the WAW dependency is shown as a dotted arrow.

IV.B.2 Optimizing Strata Log Size

We call the log containing the strata as the *Strata Log (SL)*. We use the example shown in the Figure IV.2 to explain how a strata log is created. We do not have to record anything for the intra-thread dependencies. In the example, *W1*→*R1* is an intra-thread RAW dependency, which is revealed during replay by thread $T1$'s program order.

In a naive implementation, the SL has a stratum recorded for every cross-thread RAW and WAW dependency. However, the offline analysis algorithm only requires that, for each observed cross-thread RAW or WAW dependency, there is *at-least* one stratum in the strata log that separates the predecessor and the successor. This means that one stratum can be used to separate more than one cross-thread RAW or WAW dependency.

In the example shown in Figure IV.2, the stratum $S1$ is logged when the WAW dependency *W1*→*W2* is observed during a program's execution. The recorded stratum allows us to determine that the write $W1$ has to be executed before the write $W2$ during replay. Similarly, the stratum $S2$ is logged to capture the *W2*→*R2* dependency.

However, when the RAW dependency *W2*→*R4* is observed, we do not have to log an additional stratum. The reason is that, the stratum $S2$ is sufficient for us to determine that $W2$ preceded $R4$ in time. For the same reason, we do not have to log a stratum for the RAW dependency *W3*→*R3*. $S1$ or $S2$ is sufficient to capture the RAW dependency *W3*→*R3*. Also note that the memory operations, $W3$ and $R3$, involved in the RAW dependency are accessing a memory location different from the one that triggered the creation of the strata $S1$ and $S2$.

For the RAW *W4*→*R5* dependency, we have to log an additional stratum $S3$, because none of the earlier strata separate $W4$ and $R5$ in time. Thus, a stratum for a cross-thread WAW or RAW dependency is logged only if the preceding

operation in the dependency gets executed after the last recorded stratum.

IV.B.3 Advantages of Strata

We now describe the advantages of using strata to record cross-thread RAW and WAW dependencies.

Efficient Transitive Optimization

A stratum consists of the current memory counts of all the threads. In the point-to-point logging approach used in FDR [104], to record a dependency, the memory count for the time when the preceding memory operation was executed is logged along with the memory count of the succeeding memory operation. Logging the memory count for the predecessor is less "strict" than logging the current memory counts of all the threads. Therefore, a single stratum can potentially capture many RAW and WAW dependencies. Thus, the transitive optimization used to reduce the SL size is more efficient than the Netzer transitive optimization used for point-to-point logging [104].

No WAR Logging

Capturing shared memory dependencies using strata allows us to ignore WAR dependencies while logging and determine them offline. Not capturing WAR, reduces the strata log size and the hardware support required for recording them using strata.

Efficient Hardware Implementation

The amount of hardware required to implement the transitive optimization to reduce the strata log size is significantly less than what is required to implement the Netzer optimization in FDR [104].

Supports Snoop Based Systems

In a directory system, when a dirty block is evicted, the directory maintains the information about the processor that last accessed that block. However, in a snoop based system that information is lost, which is a problem for the point-to-point logging approach used in the prior works [104]. This problem is solved if we use the strata for logging RAW and WAW dependencies, because we do not require precise information about the predecessors during replay. We just need to know whether there is a RAW or WAW dependency due to the evicted cache blocks or not. We detect dependencies with evicted dirty blocks by keeping track of the set of evicted blocks in a bloom filter (a bit vector indexed by the hash of an address). The hardware implementation for a snoop-based system will be described in detail in Section IV.C.

Handles Paging

In Section IV.A, we pointed out a limitation of the point-to-point solution in handling paging in the middle of recording a program's execution. When a page is swapped out, all the information about who accessed what block and when it was accessed is lost, as it can be found neither in the private caches of the processor nodes nor in the directory (the directory does not maintain any information for the memory blocks in the swapped-out pages).

To address this problem, we log one additional stratum when a physical page is re-mapped. The recorded stratum separates the memory operations executed before and after the paging activity, which is sufficient for the offline analysis to determine the dependencies. Since the paging activity is less frequent, when compared to the frequency at which we create strata during normal execution, additional strata logged due to paging constitute a small portion of the strata log.

IV.B.4 Off-line Analysis to Determine a Total Order

In this section we discuss a replayer can determine a total order between the memory operations. Assume for now that the replayer knows all the memory operations executed in each thread along with their addresses. Assume that the replayer also knows their program order. We postpone the discussion on how to get this information during replay for the BugNet approach (described in the previous chapter III and the FDR [104] approach to Sections IV.B.4 and IV.B.4 respectively.

Assuming that a replayer has the above information, using the strata log, the replayer can infer the total order between the memory operations observed during logging. We use the example shown in Figure IV.2 to explain our algorithm. Figure IV.2 shows the memory operations along with their addresses and output values. But for our offline analysis we do not require information about the output values.

We define a *strata region* to consist of all the memory operations executed across all of the threads between two strata. Figure IV.2 has three strata regions: $S0 - S1$, $S1 - S2$ and $S2 - S3$. There is a total ordering between the strata regions because the strata are ordered by time. Therefore, if we order the memory operations executed in each strata region in isolation, and then order all the memory operations across strata, we will get a total order for all the memory operations. We first discuss how to order the memory operations executed within a strata region.

The memory operations executed in a thread are ordered by the program order. For the example shown in Figure IV.2, we know that $W4$ was executed after $R4$. Hence, we have to determine only the cross-thread dependencies. However, within a strata region, there cannot be any cross-thread RAW or WAW dependency. This is true because the strata log is created in such a way that

there is at-least one stratum that separates the memory operations involved in a cross-thread RAW or WAW.

The above property simplifies our job to finding the cross-thread WAR dependencies within a strata region. Since we are assured that there is no cross-thread RAW dependency within a strata region, if we find a read and a write in two different threads, such that the addresses for both the operations are the same, then we are guaranteed that the write has to be executed after the read during replay (WAR dependency). For example, in Figure IV.2, both $R2$ and $W4$ access the same memory location and they are within the same strata region $S2 - S3$. If $R2$ was executed after $W4$ during logging, a stratum would have been logged between the two operations. Since there is no stratum separating the two, we know that the dependency between those two operations is a WAR dependency, $R2 \rightarrow W4$.

Once we have identified all the cross-thread WAR dependencies for a strata region, we can determine a valid total order for the memory operations of a strata region. While determining the total order, we make sure that the program order is preserved in addition to the inferred WAR dependencies. For example, a valid total order for the memory operations of the strata region $S2 - S3$ is the following: $R2 \rightarrow R3 \rightarrow R4 \rightarrow W4$. For the strata regions $S0 - S1$ and $S1 - S2$ there are no cross-thread WAR dependencies. Hence, for those regions we just need to make sure that the program order is preserved. A valid total order for the strata region $S0 - S1$ is $W1 \rightarrow R1 \rightarrow W3$.

Now that we have determined a total order for the memory operations of each strata region, we can order all the memory operations using the recorded total order for the strata regions. For example, we know that $S0 - S1$ happened before $S1 - S2$, $S1 - S2$ happened before $S2 - S3$ and so on. Therefore, the memory operations of the strata region $S0 - S1$ should precede the memory oper-

ations of the strata region $S1 - S2$ in the total order. We can therefore determine a total order for all the memory operations within these three strata regions. In our example, a valid total order is *W1*→*R1*→*W3*→*W2* →*R2*→*R3*→*R4*→*W4*. However, we obtained this total order based on the assumption that we have knowledge of all the memory operations and the addresses that they accessed. The next two sections explain how to obtain this information during replay using the checkpoint logs of BugNet (described in Chapter III) and FDR [104].

Replaying Total Order in BugNet Replayer

The BugNet's First Load Log (FLL) (refer Chapter III) is created for each thread. It captures the values of the load instructions executed in a thread, which is sufficient to deterministically replay that thread. It is sufficient even in the presence of shared-memory updates, because a load accessing an address written by another thread will notice that the address' value has changed, and will log that value in the its FLL.

By deterministically replaying each thread individually, we create a *replay* trace for each thread. In the *replay trace* of a thread, we have information about all the memory operations executed by that thread in the program order along with the addresses that they accessed. Using this information, and the offline analysis described earlier, we can derive a total order between all the memory operations. The Pin [49] based BugNet replayer that we discussed in Section III.E replays the threads concurrently, but at the same time preserves the total order deduced from the strata logs. To enforce the order, for each thread, the replayer keeps track of the memory count starting from the beginning of the checkpoint interval for the thread. When a thread reaches the memory count specified in the next entry in the strata log, it is stalled until all the other threads also reach their respective memory counts specified in the strata log entry. Thus, a programmer

would be able to see the dependencies between the memory operations during replay.

Replaying Total Order for a Copy-On-Write based Replayer

FDR [104] uses a copy-on-write checkpoint scheme along with a *redo* log to deterministically replay the full system. In a copy-on-write checkpoint scheme, whenever a memory location is updated, the old value residing in the memory location is logged. In addition, the final state of all the memory locations is logged at the end of a checkpoint. Using the final state, and the log of memory updates, one can determine the memory values at the beginning of a checkpoint. With this information, one can start replaying. However, during replay, we also have to reproduce all the system interactions and the shared memory dependencies. To reproduce the system interactions (like interrupts, system calls and DMA transfers) FDR explicitly logs such information in what we call a *redo* log.

In order to deterministically replay using the FDR checkpoint logs, we need information about the shared memory dependencies. The strata log that we described earlier can be used for this purpose. We can deterministically replay the full system using the FDR's copy-on-write logs and the strata log as follows.

We start the replay from the first strata region in the checkpoint and then proceed to replay the following strata regions in order. However, it is not straightforward to replay from the start to the end of a strata region without the knowledge of potential WAR dependencies that may exist within the strata region. We solve this problem by performing a search through the possible memory orderings for a strata region.

We first begin the replay for a strata region without assuming any WAR dependency and the only order we preserve is the program order. During the search, we may observe a read and a write executed in two threads with the

same address. We know for sure that this is a WAR dependency and not a RAW dependency, because during logging, we create strata in such a way that there are no RAW or WAW dependencies within a strata region. However, in our replay experiment, while searching for a correct memory ordering for the strata region, we might have executed the write before the read. If so, we take note of the WAR dependency and start replaying again from the start. In the subsequent replay experiments to find a correct memory ordering, we enforce the WAR dependences that were found in the earlier replay experiments.

During replay, we are guaranteed to not wander down a control path that is different from the recorded program execution. For that to happen, some load would have to have *read* an incorrect value written by another thread during the replay. However, that would be a cross-thread RAW dependency, which is not valid, since there cannot be any RAW dependencies within a strata region. During replay, if we find a RAW dependency, then this means that this is really a WAR dependency, and we take note of this newly found dependency and restart replay from the beginning of the strata region. In that replay and the subsequent replays, we will not allow the write to execute till the dependent predecessor read in the other thread has executed. For example, consider the strata region $S2 - S3$ in Figure IV.2. During our replay search for a correct memory order, it is possible that the write $W4$ is executed before the read $R2$. After noting this WAR dependency, in our subsequent replays, if we reach $W4$ before executing $R2$, we will stall the thread T1 till $R2$ in thread T3 has executed.

We continue the above process till we are able to replay up to the next stratum without encountering a RAW dependency. This gives us a final memory ordering for the strata region, and that is used for deterministic replay debugging. This ordering lists instructions in the same order as observed during program execution.

IV.B.5 Correlating Strata Logs to BugNet/FDR Checkpoint Logs for Replay Debugging

In Section III.B we mentioned about Memory Race Logs (MRL) that record the shared-memory dependencies. Strata log serves the purpose of MRL. The strata log needs to be correlated and used in conjunction with the checkpoint logs of BugNet. If FDR's copy-on-write checkpointing mechanism is used for capturing system interactions, then strata logs need to be correlated with FDR's checkpoint logs. The following approach takes care of this problem.

A strata log is created at the same instant in all the processors. Section IV.C and Section IV.D describes how this is done for snoop-based and directory-based systems respectively. When a strata log is created in a processor, the first entry is initialized with the memory counts of all the processors. The memory count of a processor is the total number of memory operations that the processor has executed since the logging process began. Each processor keeps track of its private memory count. We assume a 32-bit counter for the results in this thesis.

When a new checkpoint is created in FDR [104] or BugNet, the current memory count value is stored in the new checkpoint header. In the case of BugNet, like we described in Section III.B, a checkpoint with then FLL for a thread running in a processor node is created independent of the other threads in the system. We described the contents of FLL in Section III.B. In addition, to those contents, the current running memory count of the processor is recorded in the FLL. Unlike BugNet, in the case of FDR, a global checkpoint is created. The checkpoint header of the global checkpoint contains the memory count of all the processor nodes. Thus, the header of the checkpoint log (FLL in the checkpoint log case of BugNet) is initialized with the memory count.

As we replay the memory operations in a checkpoint interval, the mem-

ory counts of those memory operations can be derived from the initial memory count recorded in the checkpoint header. Also, each entry in the strata log contains the memory counts of all the processor nodes. Thus, an entry in a strata log containing the memory counts can be mapped to the corresponding memory operations during replay.

To reduce the strata log size we examine a compression technique that logs the difference between memory counts in 16-bit values. Instead of logging the 32-bit memory count value, we just log the difference between the memory count for the processor in the immediately preceding stratum and the current memory count for that processor. Even when we log just the stride values, it is still possible to correlate the strata logs with the checkpoint logs of FDR and BugNet, because at the beginning of the strata log we log the full memory count values of all the processor nodes. If we start a new strata log, the header of the new strata log would contain the current memory count of all the processors. This allows us to correctly map the strata log entries with the FDR and BugNet checkpoint log headers.

With the above information we can replay the program's execution for deterministic replay debugging using the BugNet/FDR checkpoint logs in combination with the strata logs, as long as we have a little more information about system events. For BugNet, the only other piece of information needed for deterministic replay is the order of thread context switching. To address this, BugNet has a context switch log to record the time of the context switch using the memory count of the processor, as well which thread is context switched in and which thread is context switched out for the processor. For FDR [104], it does not need a context switch log, since it can deterministically replay the operating system thread scheduling, but it does need the redo log, which provides the ability to replay all of the inputs to the system.

Since these systems provide deterministic replay, and we now have a total order for the memory operations, they can be used to single step through multi-threaded execution for debugging. This allows the developer to observe the interaction between the threads through the shared memory reads and writes, which is useful to track down bugs due to data races.

IV.B.6 Processor Effects on the Logging

We now discuss how to handle logging at the block level, prefetching, and how out-of-order execution affects the strata logs.

Capturing Dependencies at the Cache Block Level

We detect the shared memory dependencies at the granularity of cache blocks. This is because, we detect dependencies by observing the cache coherence messages, which operate at the granularity of the cache blocks. As a result, we might detect a false shared memory dependency (due to two processors accessing different words in the same cache block), and log a stratum for it.

However, the above is not an issue for our offline analysis. When a false dependency is detected, in the worst case, one additional stratum is logged. This is not issue, because the additional stratum just specifies a much stricter (but still a valid) ordering between memory operations.

Prefetching and Out-of-Order Execution

A hardware prefetcher or a software prefetch instruction can bring a memory block into the cache which might not be eventually used (read or written) by the processor. This might result in unnecessary strata being logged. However, additional strata do not compromise correctness.

Non-blocking caches and out-of-order execution in modern processors

can send or receive a coherence request/reply for a cache block out-of-order (out of program order). In systems implementing aggressive speculation, a cache block may be accessed even before its coherence is done. However, even in these systems, if the processor supports sequential consistency (which is what we assume and model), then it makes sure that the cache access and the coherence activity appears to be in the commit order of the instructions (program order). Therefore, our strata logs and coherence messages associated with the strata logs are consistent with the program commit order.

IV.C Hardware Implementation for Snoop-based Systems

This section discusses support for creating Strata Logs (SLs) in snoop-based multi-processor systems. As we discussed in Section IV.A, the previous Point-to-Point approach [104] does not provide a solution for snoop-based systems.

IV.C.1 Detecting Cross-Node RAW and WAW for Cached Blocks

To explain our approach, let us assume for now that the caches are of infinite size. This means, once a processor node accesses a memory block, it stays in its private cache till another processor writes to it. If another processor writes to the block, then it gets invalidated. Therefore, a memory block once fetched into some processor's cache resides in at least one of the processors' caches throughout the lifetime.

Our goal is to detect cross-node RAW and WAW dependencies. We achieve this by monitoring the coherence messages. Whenever a processor node encounters a read or a write miss for a memory block, it places a request on the bus. If any other processor node has a dirty copy of the memory block, which means the processor wrote to the block, then there exists a RAW or a WAW

dependency. Therefore, when the owner of the block replies on the bus, the reply is piggybacked with a *log stratum* bit, whose value is set. The log stratum bit instructs other processor nodes in the system to log a stratum. Each processor node, logs their current memory count in their strata log. Our design ensures that the memory count logged for the processor that generated the read or write miss, corresponds to the memory operation executed prior to the read or write. This ensures that the stratum separates that read or write from all the prior memory operations.

Note that the memory count for each processor representing the stratum is logged into each processor's own strata log. Therefore, we need to be able to construct a global strata log from the individual per processor strata logs. However, we create the strata logs in all the processor nodes at the same time like we described in Section IV.B.5. They are initialized with the full 32-bit memory count values of the respective processor nodes at the time of creation. Thus, the strata logs across all the processor nodes always stay synchronized. That is, the first entry in a processor's strata log corresponds to the first entry in every other processor's strata log, and it is the same case for the rest of the entries in the strata logs as well.

IV.C.2 Detecting Cross-Thread RAW and WAW for Evicted Blocks

The previous section assumed infinite caches. Let us now waive this assumption. With finite size caches, there exists an issue for the snoop-based protocol when a dirty block is evicted from the cache. When a memory block is evicted out of the cache, information about the last writer to that block is lost.

We solve the above problem for a snoop-based system using a separate bloom filter [87] in each processor node. Our bloom filter is a bit vector indexed by a hash of the memory address. Since we are interested in detecting only the

cross-node RAW and WAW dependencies, we must keep track of the fact that there was a writer to this memory block. Hence, whenever a dirty block is written back (evicted) to main memory over the bus, all the other processor nodes snoop the bus, and set the bit in their private bloom filters by indexing them using the hash of the physical address of the memory block that is being written back.

If a processor node encounters a read or write miss for a memory block, it checks its private bloom filter to see if, in the past, some other processor node had written to that memory block. If the bit for the block is set in the bloom filter of the processor that encountered the read miss, then we know that there may be a potential cross-node RAW or WAW dependency. Hence, a stratum has to be logged. To log the stratum, we piggyback the coherence request message (generated by the processor that encountered the read/write miss) with the log stratum bit set to true. All the processors snooping the bus will see a set stratum bit. This forces each processor node to log its current memory count in its strata log. Whenever a new stratum is logged, all the processors clear all of the bits in their bloom filters. We can clear all of the bloom filters, because the recorded stratum separates all the reads and writes that follow the stratum from those writes that were executed in the past. Essentially, by clearing the bloom filters we are implementing the transitive optimization for the writes to the uncached blocks.

The bloom filter essentially predicts whether an uncached block was written after the last recorded stratum. The bloom filter guarantees that there are no false negatives. That is, we will not miss any RAW or WAW dependency due to aliasing in the hash indexed bit vector. However, there could be false positives, which means that we may end up logging a few more strata than we need to. For our results, we use a bloom filter of size 128 bytes (1024 entries, one bit per entry) per processor-node, which resulted in less than 1% of additional

strata for most programs we examined.

IV.C.3 Implementing Transitive Optimization for Cached Blocks

When we detect a cross-node RAW or WAW dependency, we do not have to log a stratum if the write operation involved in the dependency was executed before the last recorded stratum. Earlier, we explained how this optimization is implemented for uncached blocks by just clearing the bloom filter bits when a stratum is logged. If the dirty block is cached in one of the processor nodes, then we need a way to know if the write to that block occurred before or after the last recorded stratum.

Unlike the Netzer transitive optimization used in the previous Point-to-Point proposal [104], which required storing the instruction count with each cache block, in our approach all we do is associate a single bit with each cache block. We call this bit the *dependence bit*. The dependence bit for a cache block indicates whether the cache block was written before or after the last recorded stratum. The dependence bit for a cache block is set whenever there is a write to the cache block and is reset whenever a stratum is logged.

When we have a read or write miss, and a dirty block is found in another processor's cache, then this means that there is a RAW or WAR dependency, but we *log a stratum only if the dependence bit is set for that cache block*. Also, while evicting a dirty cache block, *the bloom filters of the processor nodes are updated only if the dependence bit for the evicted cache block is set*. If the dependence bit was not set for the dirty block, we do not need to keep track of it in the bloom filter nor log a stratum. This is because a stratum has already been logged since the last time the block was modified.

Whenever we log a stratum, in addition to clearing the bloom filters for all the processor nodes, we also clear all the dependence bits in all the caches.

This is valid, because all the writes that were executed before logging the stratum are separated from the memory operations that are going to be executed after recording the stratum.

IV.C.4 Complexity Advantage of not Logging WAR

We need to capture just the RAW and WAW dependencies like we described in Section IV.B.4. However, in our experimental evaluation to be discussed in Section IV.E, we studied the size of the strata log required to capture *all* the shared memory dependencies, including the WAR dependencies. We briefly discuss here, the additional hardware required to capture the WAR dependencies.

To capture the WAR dependencies, first, we need to be able to detect the WAR dependencies. This is straight-forward as long as the blocks are cached. However, if a block read by a processor is evicted, we need to keep track of it similar to how we tracked the dirty block evictions. This requires a broadcast on the bus even when a clean cache block is evicted, which is not normally required in a MESI protocol. Each processor node snoops the broadcast message to set the corresponding bit in the private *read* bloom filter. Note that this bloom filter is an addition to the *write* bloom filter already used for tracking the evicted dirty blocks. Also, we need an additional dependence bit that tells us whether the block was read by the processor before or after the last recorded stratum.

Because of this additional hardware complexity, our primary solution focuses on recording strata for only RAW and WAW, and determining RAW using offline analysis as described in Section IV.B.4.

IV.C.5 Hardware Comparison to Point-To-Point Logging

The additional logic added to our architecture are the bloom filter per processor, and one dependence bit per cache block in the private caches of each

processor. In our scheme, we do not have to tag each memory block with the instruction count like in the prior work [104] to record the shared memory dependencies. We therefore avoid the additional 6.25% area overhead in the L1 and L2 caches used in the prior techniques.

IV.D Hardware Implementation for Directory Based Systems

In this section, we discuss how we can capture the shared memory dependencies using the Strata Log for directory based systems. Figure IV.3 shows the changes required in a directory based system to record the strata log. It shows one processor node in the multi-processor system. It also shows the directory controller where we record the strata log. For our simulations, we use an 8 KB hardware buffer to buffer the writing of the stratum to the strata log in the main memory. We also keep track of a memory count vector in the directory, dependence bits in the directory and also in the private cache of each processor node. The functions of these architectural extensions to support recording strata logs in a directory-based system in discussed in this section.

IV.D.1 Capturing RAW/WAW using the Strata Log

To explain our approach, let us assume a system with a centralized directory, and that the directory knows for each block if it has been written since the last stratum was logged. We will waive these assumptions later in this section. Since the processor nodes are not connected by a common bus like in the snoop-based systems, it is not efficient to create the strata logs in the processor nodes. We instead chose to create the strata log in the directory controller.

When a cross-node RAW or WAW dependency is observed at the directory, a stratum has to be logged by the directory controller. To log a stratum, the

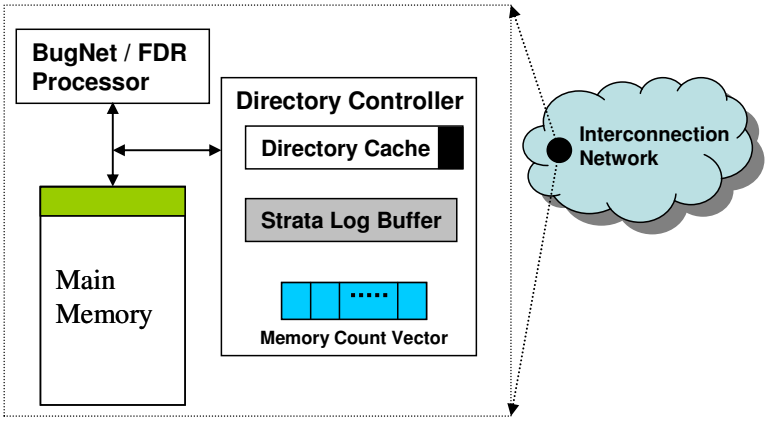

Figure IV.3: Strata logging support in a directory based system.

directory controller needs to know the current execution states (memory counts) of all the processor nodes. One way to achieve this is by polling all the processor nodes for their current memory counts. However, this incurs a heavy communication cost. To avoid this communication cost, we instead propose to have each directory controller keep track of a vector of memory counts (MVector), one for each processor node in the system. Each count represents the last time the processor accessed the directory. A stratum is logged using these memory counts. The vector of memory counts is updated as follows. Every time a processor node performs a read or a write request to the directory due to a cache miss, it piggybacks its current memory count along with the coherence request. Also, when a dirty block is written back to memory, the current memory count is also piggybacked in the write update coherence message. We update the MVector for the processor each time the memory count is piggybacked on a coherence message.

In this scheme, some of the memory counts in the vector can be stale (not up-to-date) when a stratum is logged, relative to the current memory counts on all of the processors. This is fine, since the memory counts that the directory sees can be used to log a stratum across all of the processors, which is valid

in terms of capturing the shared memory dependencies. Consider the example shown in Figure IV.4. The example shows four processor nodes. The read and write operations are shown along with their addresses. The RAW and WAW dependencies are also shown using arrows. In the figure, assume that the stratum $S0$ has been logged due to some previous dependency. The processor nodes $P3$ and $P4$ update their memory counts in the directory controller when they encounter read misses (due to $R1$ and $R2$) for the address C. Later, when $P1$ sends a write miss request for $W2$, a WAW dependency is detected with $P2$, which currently has a dirty copy of the memory block (written by $W1$). A stratum is logged to capture this *W1*→*W2* WAW dependency. The directory controller logs the stratum $S1$ using the memory counts in its MVector. Note that the memory counts for the processor nodes involved in the dependency are always up-to-date when the stratum for the dependency is logged. In our example, the memory counts of $P1$ and $P2$ are up-to-date when the stratum $S1$ is logged. Since the stratum will separate the dependent operations in time, the memory count logged for the processor node $P1$ is one less than the memory count corresponding to the write operation $W2$.

After logging the stratum $S1$, the processor node $P2$ encounters a write miss for $W5$ and updates the memory count in the MVector. Later, the RAW dependency *W3*→*R4* is observed between $P4$ and $P3$, when $P3$ encounters a read miss for $R4$. The directory controller logs the stratum $S2$ to capture this RAW dependency. Note that the memory counts are up-to-date for $P3$ and $P4$ while logging the stratum $S2$. However, it is not the same case for $P1$ and $P2$. In fact, for $P1$, the memory count logged in $S2$ is same as the memory count that was updated when $P1$ sent a coherence request for the write $W2$.

In spite of using stale memory counts while logging the strata, the following two properties that are essential for our offline analysis are still preserved:

(1) There exists at least one stratum between the memory operations involved in a cross-node RAW or WAW dependency. Thus, a strata region cannot have any cross-node RAW or WAW dependencies. (2) The strata regions are non-overlapping, because the value of a memory count in the MVector either increases or stays the same. These properties allows us to consider one strata region at a time during offline analysis, and determine a total order for the memory operations executed with a strata region.

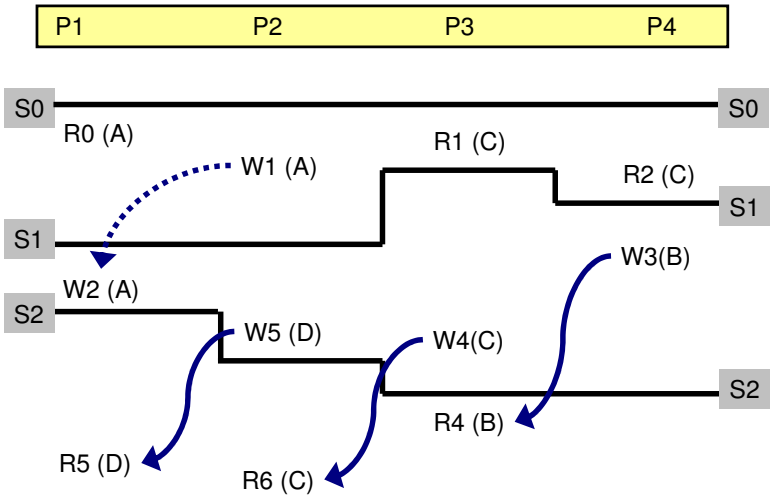

Figure IV.4: Example of a Strata log in a directory based system.

IV.D.2 Determining RAW and WAW Dependencies in the Directory

Similar to our snoop-based implementation, we use a dependence bit per cache block to determine if a dependency needs to be logged. We also have a dependence bit for each directory entry in the directory cache. The dependence bit in the processor's cache is set whenever the processor writes to the cache block. In the directory systems, the dependence bit in the directory cache is set whenever a dirty block is written back to the memory and that dirty cache block has its dependence bit set.

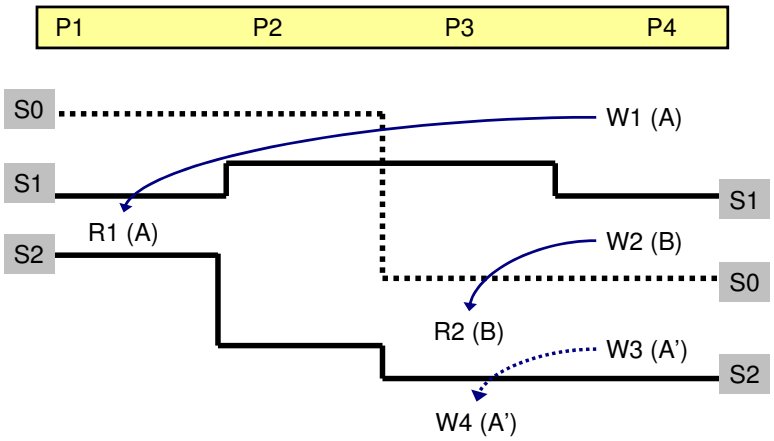

Figure IV.5: Strata log collected in a system with two directories.

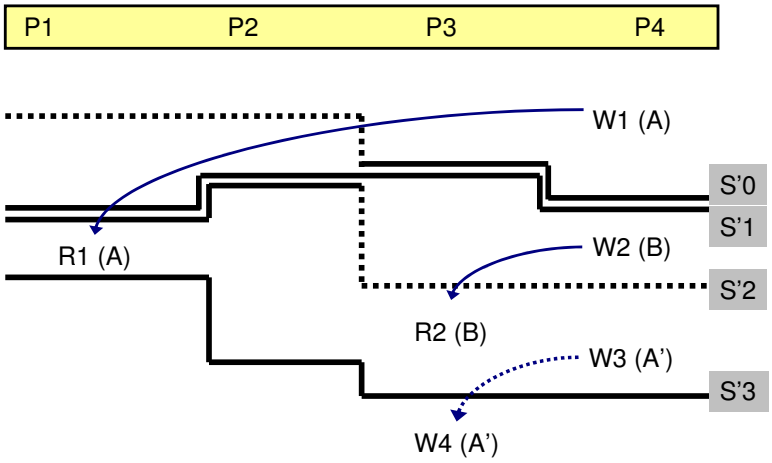

Figure IV.6: Combined strata log created from the strata logs of two different directories during replay.

While servicing a processor's read or write miss request for a cache block, a potential RAW or WAW can exist if some processor node in the system has exclusive access to the cache block. In that case, the directory controller sends a data fetch request to the processor node that has exclusive access. The processor node with exclusive access to the cache block, piggybacks the value of that block's dependence bit on the coherence reply message to the directory. We detect a RAW or WAW dependency only if the dependence bit information received through the coherence reply is set to true (that is, the owner had written to the cache block).

In case when the block in not cached in any of the processor node, while servicing a processor's read or write miss request for a cache block, we detect a RAW or WAW dependency if the dependence bit is set for the directory entry of the cache block. If the bit is set, we log a stratum to separate the dependency. If the directory cache cannot keep track of every block in memory, and we get a miss in the directory cache for the memory block, then we conservatively log a stratum on a directory cache miss. Thus, the dependence bits in the private caches of the processor nodes and in the directory cache allows us to detect the RAW and WAW dependencies and log a stratum appropriately to capture them.

When a directory logs a stratum, the dependence bits for all the entries in that directory can be cleared. This is because, any write that had set a dependence bit in the directory entry earlier is ordered by the stratum that the directory is currently logging. To reduce the amount of logging as much as possible, we would also like to clear the dependence bits in the processor caches, whenever a stratum is logged. We would only be able to do that for all of the processors, if the stratum contained the current memory counts for all the processor nodes. However, this would require additional coherence messages between the directory and every other processor node in the system. Therefore,

we chose to reset the dependence bits only for the two processor nodes that are involved in the RAW/WAW dependency that triggered the creation of the stratum. This is valid because the new stratum contains the up-to-date memory counts for those two dependent processor nodes. Hence, all the writes executed in those two processors before logging the stratum are ordered by the recorded stratum.

Consider again the example shown in Figure IV.4. Assume that after the write $W5$, the dirty block (with address D) was immediately written back to the memory. This sets the dependence bit in the directory entry corresponding to the memory block with address D. When the stratum $S2$ is logged, the dependence bits in the directory are reset. This includes the dependence bit for the block D. Later, when $R5$ accesses the same block, a new stratum is not logged, even though there is a cross-node RAW dependency ($W5 \rightarrow R5$), because the stratum $S2$ already separates $W5$ and $R5$.

Now let us explain another example to show how the dependence bits in the caches are used. The processor node $P3$ executes the write $W4$ to the address C and sets the dependence bit in its cache block. The cache block remains in the dirty state until the time when processor node $P2$ executes the read $R6$. Clearly, there is a cross-node RAW dependency $W4 \rightarrow R6$ between $P3$ and $P2$. However, when the stratum $S2$ was created for the RAW dependency $W3 \rightarrow R4$, the dependence bits in the private caches of $P3$ and $P4$ would have been reset. As a result, the dependency information piggybacked along with the coherency reply from $P3$, to service the read miss for $R6$ $W4 \rightarrow R6$, tells the directory that a new stratum to capture the RAW dependency $W4 \rightarrow R6$ is not required. Thus, a stratum is not logged for the RAW dependency $W4 \rightarrow R6$.

IV.D.3 Strata for a Distributed Directory

In this section, we relax the assumption of a centralized directory and show that our approach is applicable for distributed directories as well. In the case of distributed directories, each directory controller captures the dependencies for the addresses that it services using a strata log. So we have multiple strata logs, which are combined into one unified strata log during offline analysis. Figure IV.5 shows an example of two strata logs, collected in a distributed directory system with two directories. The solid strata are collected in one directory and the dotted ones are the strata collected for the other directory.

A strata log collected in a directory serves the purpose of determining the dependencies between the memory operations accessing the addresses mapped to that directory. Therefore, we are still guaranteed that each cross-node RAW and WAW dependency is separated by at least one stratum in one of the strata logs. In Figure IV.5, addresses A and A' are assumed to be mapped to one directory (the strata log for this directory is shown using solid lines). The address B is mapped to the other directory. It can be seen that the solid strata $S1$ and $S2$ separate *W1*→*R1* and *W3*→*W4* dependencies respectively, while the dotted stratum $S0$ captures the *W2*→*R2* dependency.

However, unlike in a centralized directory, the strata regions in the strata logs collected in different directories can be overlapping. The reason for this is that the MVectors used to log the strata across the directories are not updated in the same way. An entry for a processor node in the MVector is updated only when that processor node communicates with that directory to resolve a miss or when it is writing back a dirty cache block.

Overlapping strata regions are an issue, because in the offline analysis that we described in Section IV.B.4, one strata region is analyzed at a time. To solve this problem, in our offline analysis, we first combine the multiple strata

logs for the different directories such that there are no overlapping regions in the combined strata log. When combining the strata logs, we look to see if there are any two strata that are intersecting. A stratum in each log contains the memory counts for each processor when the stratum was recorded in the directory, and from these counts we can easily determine if two strata are intersecting. If there are two intersecting strata, we use their memory counts to make non-overlapping equivalent strata, which are put into the combined strata log. For example, in Figure IV.5 the strata $S0$ and $S1$ are intersecting. For these two strata, we create three strata $S'0$, $S'1$ and $S'2$ in the combined log in such a way that none of the new strata intersect, and these new strata still separate the regions of memory operations that the two strata were originally created to separate. Figure IV.6 shows the combined strata log. It is stricter because the number of strata, between any two dependent memory operations, is either the same as before removing the intersections or greater. For example, in Figure IV.6, the reads and writes involved in a RAW or a WAW dependency are still separated by at least one stratum. There are two strata $S'0$ and $S'1$ that separate the cross-node RAW dependency, *W1*→*R1*, whereas in the strata log shown in Figure IV.5 there is only one stratum $S1$ to separate those dependent operations. The RAW dependency *W2*→*R2* (observed in the second directory) is captured by the stratum $S'2$.

Once we have the strata log with non-overlapping strata regions, we can use the offline analysis that we described in Section IV.B.4 to determine a total order for all the memory operations.

IV.D.4 Complexity Advantage of Not Logging WAR

If we wanted to also capture WAR shared memory dependencies with our strata approach, we additionally need to have read dependence bits in the processors' caches and in the directory cache. These additional dependence bits

are required to determine whether there was a read to a memory block after the last recorded stratum. When there is a write miss, we can detect if there is a WAR dependency, and using the read dependence bit information we can decide whether to log a stratum or not. We use this approach for analyzing the log size overhead of capturing all the shared-memory dependencies in Section IV.E. However, like we described in Section IV.B.4, we do not have to record WAR dependencies because they can be determined using an offline algorithm.

IV.D.5 Hardware Requirements

The additional hardware states required for creating the strata log is just one dependence bit per cache block in each processor node and one dependence bit per directory entry in the directory.

IV.E Results

In this study, we used Simics [50] to record the logs as well as to model the architecture support required for logging. In Simics, we modeled a four node CMP processor with 64KB L1 caches and a 2MB L2 cache, and modeled both directory and snoop protocols.

To evaluate our shared memory dependency logging improvements, we used data parallel programs from the Splash benchmark suite [103]. We could only get five of the main benchmarks to compile, and we provide results for all of these, which are `barnes`, `ocean`, `radiosity`, `raytrace` and `water`. We focus on these for tracking shared memory dependencies, since they represent a workload which will stress the shared memory dependency logging. We ran each program configured with 5 threads on a 4 processor node system. Each program was run until the total number of memory operations across all the threads reached 400 million memory operations per program. We found this to roughly execute about

100 million memory operations per thread.

IV.E.1 Logging Performance Overhead

In terms of performance overhead, we found that the logging incurs only a 1% slowdown due to the extra memory traffic from logging, which is consistent to the low (few percent) overhead reported in FDR [104]. This is because the logs are written to 8KB memory race buffer and so they do not pollute the caches at all. Also, logs are written back to the main memory with a low priority.

IV.E.2 Strata Logging Results

In Figure IV.7, we present results for the number of log entries to capture shared memory dependencies. The x-axis represents the programs and the y-axis represents the number of log entries generated by the point-to-point (P2P) scheme with Netzer optimization, our new Strata directory and Strata snoop architectures. Results are shown in terms of the average number of log entries for 1 million memory operations. The figure shows five bars. We first concentrate on the first, second and fourth bars, representing P2P, Strata for Directory and Snoop cache coherence protocols. A log entry for P2P is created for every RAW, WAW and WAR shared memory dependency. The log entry for our Strata results represent the number of logged strata, when logging strata for only RAW and WAW. The results show that Strata has significantly less log entries when compared to P2P. For our directory approach we require 10.5x less number of logs than P2P, and for snoop based system the number of log entries is reduced by a factor of 9.9x.

The reasons for these savings are two-fold. First, Strata only records RAW and WAW dependencies, therefore saving all log entries related to WAR dependencies. More importantly, our scheme implements a transitive optimization,

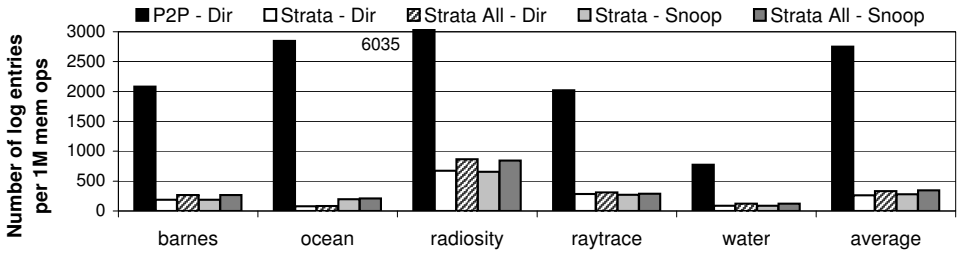

Figure IV.7: Number of P2P and Strata Log Entries

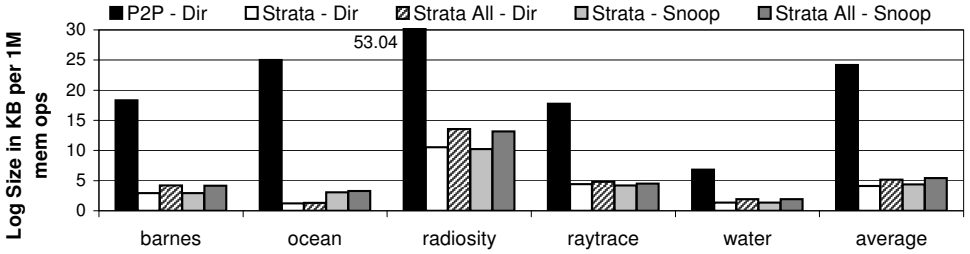

Figure IV.8: Log size for Recording Memory Dependencies

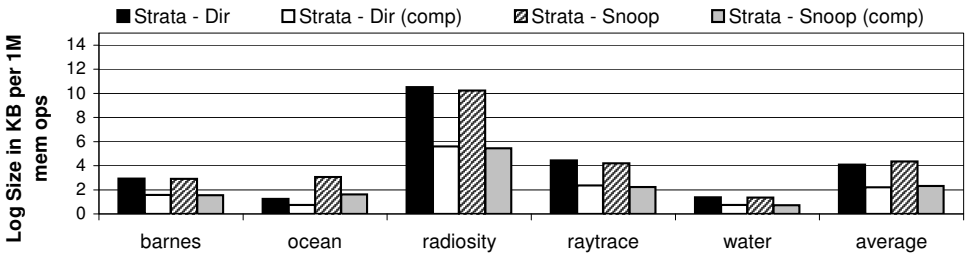

Figure IV.9: Compressed Log sizes for Recording Memory Dependencies

which yields significant savings in reducing the number of strata to be logged. Strata Snoop does slightly worse than Strata Directory because of aliasing in the bloom filters.

In Figure IV.7, the third and fifth bars (All) show the number of strata log entries if we create a stratum for logging all shared dependencies (including WAR). The mechanism to record WAR using strata was described in Section IV.C.4 and Section IV.D.4. Strata logs for recording WAR dependencies incur an overhead of 25% additional log entries when compared to logging only RAW and WAW dependencies. In addition, it requires the extra hardware explained in Sections IV.C.4 and IV.D.4, in order to monitor whether reads occurred after the last logged stratum. From the results, we see that most of the savings, in terms of the number of log entries come from our stricter (and more efficient) strata transitive optimization. The transitive optimization results in significantly fewer log entries when compared to the point-to-point (P2P) Netzer optimization.

Figure IV.8 shows the log sizes in terms of bytes for every 1 million memory operations (y-axis) for each approach. The first bar shows the P2P approach. It requires 9 bytes to record a shared memory dependency [104]. The second and the fourth bars show the log sizes for recording only RAW and WAW dependencies using our Strata Directory and Strata Snoop designs. The third and the fifth bars show the log sizes for logging strata for recording RAW, WAW, and WAR. For every stratum four words are logged, one word is required for the memory count of each processor. These are our uncompressed results. When compared to P2P, our shared memory dependency logs are 6x and 5.6x smaller on average for the directory and snoop-based systems respectively. If all the dependencies are logged using strata (which is not needed unless one wants to avoid the overhead of offline algorithm during replay), the ratios go down to 4.5x and 4.4x for the directory and snoop-based systems respectively. Figure IV.8 shows

a 25% reduction in log size for not having to log WAR dependencies with Strata. It has be noted that the main advantage of recording only WAW and RAW and figuring out the WAR off-line is that, we avoid the hardware complexity required to capture the WAR dependencies (refer Section IV.C.4 and Section IV.D.4).

Figure IV.9 shows the results for Strata log sizes without and with compression. We show results for logging only the RAW and WAW dependencies, since the same trend holds for logging all dependencies as well. Without compression, the Strata approach logs 4 words for each stratum logged, as described before. For the compressed results, instead of logging a full 32-bit memory operation count for each processor, we log only 16-bits if the memory count stride (difference between the previous stratum memory count and the current stratum memory count for a processor) can be expressed using 16-bits. If 16-bites are not sufficient to log the stride, then we log the full 32-bits. To distinguish between these two formats, we need one bit for each memory count entry per processor. Therefore, we approximately have two words per log entry after compression. The first two bars show the results for the Strata Directory approach without and with compression, and the last two for the Strata Snoop approach without and with compression. The results show that with the simple form of compression that we described, our log sizes are 47% smaller than not using the compression. When using compression, the Strata log sizes are on average 12x times smaller than P2P.

Overall, the results show that the storage overhead of logging the shared memory dependencies for 1 million operations is roughly 24KB in the case of P2P approach [104]. Whereas, for strata-based approach it is 4.1KB in a directory-based system and 4.4KB for a snoop-based system without using compression. With compression, the sizes are 2.2KB and 2.3KB for the directory and snoop-based systems. To put these log sizes in perspective with the rest of the logging

done for BugNet, for these programs, the First Load Log (FLL) size required to capture the execution for 1 million memory operations is 26.6 KB on average without compression. Therefore, the Strata logs account for about 15% of the total log storage needed to provide deterministic replay with BugNet for these programs.

Another interesting observation is that the Snoop approach results in more log entries than the Directory approach. This is especially perceivable for the program `ocean` in Figure IV.7. This is due to aliasing in the bloom filters, which result in false positives when detecting dependencies with uncached blocks. We set an entry in the bloom filter when a dirty copy of a block is written back to the main memory. However, if another entry aliases to the same bloom filter entry, then on a miss, we would detect a false dependency between two operations, and log a stratum. Note that this is not a correctness issue. It just results in redundant strata log entries, and hence larger log sizes. To measure the effects of aliasing in our schemes, we measured how many strata were logged due to aliasing. For all benchmarks except `ocean` less than 1% of the strata were logged due to aliasing. For `ocean` however, 64% of the strata logged are due to false positives. This is because of the large number of dirty block evictions encountered during execution, resulting in heavy use of the bloom filters for this particular benchmark.

IV.E.3 Bandwidth Overhead

We also collected results for the overheads of our approach in terms of communication bandwidth. For calculating this overhead, we accounted for the extra bytes transmitted on the bus for recording the shared memory dependencies. The baseline bandwidth is computed as the number of bytes transmitted on the bus to serve the read and write misses, plus the size of coherence messages that

handled those misses. For the P2P approach, the bandwidth overhead is due to the instruction counts piggybacked on the write update reply messages and invalidation reply messages. The overhead is about 10% extra bandwidth. For our approach for directory-based systems, the overhead is due to the memory operation counts piggybacked on the messages sent to the directory as a result of the write misses, read misses and write evicts. Also, the coherence replies from the exclusive owners in response to data fetch request need to be piggybacked with the dependence bit. The bandwidth overhead our strata based approach in a directory-based system is about 12%, which is 2% higher than what P2P requires.

In a snoop-based system, the coherence reply and the request messages are piggybacked with one additional bit that instructs the processor nodes whether to log a stratum or not. This is the source of additional bandwidth overhead. In addition, before paging, an additional message is broadcast on the bus instructing all the processor nodes to log a stratum. For the programs we examined, we found that these do not incur any appreciable communication overhead - both in terms of number of bytes communicated and in terms of the number of messages communicated.

IV.E.4 Scalability

We finally point out some aspects regarding the scalability of our strata approach. Note that the number of entries in a strata log is proportional to the number of processor nodes, and not threads. Therefore, logging overhead would scale linearly with the number of processor nodes in the system. However, for our directory approach, since we clear the dependence bits of only the dependent processor nodes, our transitive optimization to reduce the number of strata logs may not be as efficient when the number of processors increase. In P2P, as the

number of processor nodes increase, the number of point-to-point dependencies also potentially increase. In our case, the size of each stratum increases with the number of processors, but so does the benefit of our transitive optimization. Because, one stratum capture dependencies across all the processors, and not just the dependent processors.

IV.F Conclusion

This chapter proposed a new logging mechanism called *Strata*, which complements the BugNet architecture discussed in the previous Chapter III. Using strata, we can capture the shared-memory dependencies in both snoop-based and directory-based system. Using this information we can replay the memory order in a multi-threaded program, which helps in debugging concurrency bugs such as data races.

A stratum is logged across all of the processors every time a shared memory dependency needs to be captured. We log a stratum only to capture the RAW and WAW dependencies. The WAR dependencies are determined through offline analysis. A stratum provides a strict time ordering between memory operations that occurred before and after the stratum executed across all the processors. We found that the strata log is 5.8x smaller without compression and 12x smaller with compression when compared to the log size required in the previous point-to-point logging approach [104]. Another advantage is that our strata approach requires less hardware than the point-to-point approach. In addition, based on the notion of strata, we were able to design a shared memory dependency logging solution for snoop-based architectures, which the previous proposals did not address.

The techniques presented in this chapter assumed a sequentially consistent memory model. Recently, Xu et al. [106] proposed an extension to their

FDR design to support systems with a weaker Total Store Ordering (TSO) model. Future work could extend the strata-based approach presented in this thesis to support weaker memory models.

Acknowledgments

The text in this chapter is in part a reprint of the material from the paper, Satish Narayanasamy, Cristiano Pereira, and Brad Calder, "Recording shared memory dependencies using strata", in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (ASPLOS)*. The dissertation author was the primary investigator and author of this paper, the co-authors involved in the publication [57] directed, supervised, and assisted in the research which forms the basis for that material.

V

Replay-based Automatic Data Race Detection

In Chapters III and IV we presented processor-based solutions for recording and replaying a program's execution. We provided a comprehensive solution to deal with all forms of non-determinism including non-deterministic system interactions and also non-deterministic races in multi-threaded programs.

Deterministic replay support has several uses, some of which we briefly discussed in Section II.B. One of the uses of deterministic replay is that we can perform any dynamic analysis over the recorded program's execution during replay to automatically find bugs. Unlike traditional dynamic analysis, offline dynamic analysis does not intrude with a program's execution as the underlying deterministic replayer ensures that the program's execution is exactly same as the one observed during recording. As an example, this chapter presents an offline dynamic data race detection tool based on a deterministic replayer to automatically find concurrency bugs in the multi-threaded programs. The novel aspect of this tool is that it automatically classifies the data races detected into potentially benign and potentially harmful categories.

The data race detection tool presented here is based on a software implementation of BugNet-like checkpointing and logging mechanism. We have developed a software-based recorder and replayer based on the BugNet checkpointing and logging mechanism [58]. Microsoft has also independently developed a software-based recorder and replayer called iDNA [6], which also uses a BugNet-like checkpointing and logging mechanism. The tool presented in this chapter was developed by the dissertation author in collaboration with Microsoft using iDNA.

The rest of the chapter is organized as follows. Section V.A motivates the need for a data race detection tool and introduces the key functionalities of the tool. This tool is based on a recorder and replayer tool called iDNA [6], which implements a BugNet-like checkpointing and logging mechanism discussed in Chapter III. A brief overview about iDNA is presented in Section V.B. Section V.B also discusses a happens-before relation based mechanism to find data races during replay. Section V.C describes the details of our offline analysis tool that automatically classifies if a data is potentially benign or harmful. Section V.D discusses some of our experiences in using the tool to find bugs in Microsoft's applications and Section V.F concludes this chapter.

V.A Introduction

We would like to demonstrate the utility of deterministic replay support for performing offline dynamic analysis. As an example, we present a novel dynamic analysis tool that automatically finds the most harmful data races.

Automatically detecting data races in shared-memory multi-threaded programs is a very hard problem. Data race detection tools, even the dynamic analysis tools, tend to report a large number of data races. However, only a handful of them are harmful. A *harmful* data race is one that is a source of a

concurrency bug, which can affect the correctness of a program's execution. A developer will consider fixing only the harmful data races. Ideally, an automatic race detection tool should report only the harmful data races to a developer. However, many existing tools report data races that can never occur at all. Such data races are *false positives*. Even if we manage to eliminate all the false positives, not all of the remaining true data races are harmful. In fact, in production code, we found that only 10% of the true data races are actually harmful. The remaining 90% were all *benign* data races. They were benign in the sense that the programmer was convinced that they do not affect the program's correctness and so the programmer intentionally chose to avoid the overhead of synchronization. Thus, reporting all the true data races places a huge burden on the developers as they have to manually triage and eliminate a large number of benign data races. Triaging data races is a time consuming and tedious exercise.

The offline dynamic analysis tool presented in this chapter serves two of our goals. Our first goal is to improve programmer productivity by finding and reporting only the harmful data races. To achieve this, we propose an offline dynamic analysis tool that automatically classifies the *true data races* into *potentially benign* and *potentially harmful* data races. This allows developers to prioritize the data races that need be triaged.

We find that reporting accurate information about the potentially harmful data races is very important because triaging a data race bug is a tedious exercise. Triaging a data race bug is difficult for the following reasons:

- **Requires Domain Expertise:** The effects of a data race are hard to discern, as understanding them involves analyzing multiple program states across multiple threads. Domain expertise is usually required to understand if a data race is benign or harmful in our production code.
- **Time Consuming:** The number of true data races is too large for the

developers to go through and triage them all. It can be very time consuming for a developer to triage a data race.

- **Hard to Figure Out:** Even if someone with domain expertise examines the data race, they tend to incorrectly believe it is benign when it is actually harmful, or vice versa.

Besides finding the harmful data races accurately, another problem is generating information that will help convince the developer about the existence of a data race bug. Therefore, our second goal is to generate a concrete, reproducible scenario for a potentially harmful data race. The reproducible scenario helps the programmer in debugging as it enables one to understand the harmful effects of the data race reported.

The offline data race detection tool based on a deterministic replayer presented in this chapter meets the above two goals. It can automatically classify the data races into potentially benign and potentially harmful categories. An integral part of our solution is the ability to record a program's execution in a replay log and replay the program's execution using the log. The proposed dynamic data race detection analysis is performed offline, during replay. In addition to finding the potentially harmful data races, the analysis also produces useful information for each data race. Using that information, a developer can replay a program execution in two different ways – the original execution order and an alternative order. In the alternative order, which has the order of the two memory operations involved in the data race reversed. The two replays help the developer understand how a potentially harmful data race can produce different results based on the different interleavings between the two racing memory operations.

Data Race Detection: We focus on building a race detection tool with no false positives. Our definition of a data race is dependent on the *happens-*

before relationship [43]. We determine that there is a data race between two memory operations executed in two different threads if (a) at least one of them is a write, and (b) there is no synchronization operation executed between the two memory operations (that is, there is no happens-before relation to provide an order for the two operations). Going by this strict definition of what a data race is, a happens-before based algorithm does not report any false positives. However, a happens-before algorithm still reports a large number of data races, out of which many are benign. In order for our tool to be used in practice, we need to prioritize the data races so that a developer can focus on fixing and understanding the potentially harmful data race bugs.

Data Race Classification: Our approach automatically classifies data races by leveraging the ability to replay the program's execution. The key concept behind our analysis is as follows. For a data race, the checker analysis tool replays the execution twice for the two different orders between the memory operations involved in the data race. If the two replays produce the same result, then the checker determines that the data race is potentially benign. Otherwise, it classifies the data race as potentially harmful. The data races that our tool marks as potentially benign are not examined by the developers, but only those marked as potentially harmful are examined. We keep track of the results of this analysis for each pair of memory operations involved in a data race. There can be many instances of the same data race during a program's execution and across several different executions. By analyzing these instances we can observe several effects of the data race. Thereby, we get a much clearer picture about how to classify the data race.

The tool might incorrectly classify some data races. It might classify a data race as potentially benign when it could be harmful or vice versa. If we classify a benign data race as potentially harmful, then we end up using precious

developer's time. But once those races are manually identified as benign, they are marked as benign to prevent them from being classified as potentially harmful in the future analysis. If we classify a harmful data race as benign, they will not be examined by the developer. However, later on, when analyzing a different test case, the analysis may find an instance of the data race that exposes it as potentially harmful. The data race's classification will then be corrected as potentially harmful, and reported to the developer. Thus, the more the number of test cases analyzed, the more likely a harmful data race will be discovered (which is true for any dynamic analysis tool). This is a trade-off between coverage and accuracy that we make during development. We strive to reduce the number of data races reported to the developer, because there are just too many true data races found, and most of them are benign.

Data Race Report: At the end of our analysis, we provide the developer with precise information about the effects of each potentially harmful data race. The information includes the replay log and the two memory orders that were analyzed by the checker for the data race. One of the replays will produce the correct result and the other will produce a different result. Thus, a developer has the precise information about the memory operations involved in the data race, *and* also has the ability to replay the program in two different ways (two ways are the original execution order and the alternative order that is possible due to the data race) and understand the effects of the data race. If the same data race had occurred multiple times within the same or different execution scenario, we provide information for all of those instances to help the developer understand the various possible effects of a particular data race.

iDNA and Usage Model: We perform our dynamic data race analysis using the replay tool called iDNA developed by Bhansali et al. [6]. iDNA implements BugNet-like checkpointing and logging mechanism that we described

in Chapter III. iDNA provides the ability to record a program's execution in a replay log. Using the replay log, iDNA can replay a multi-threaded program's execution, even in the presence of all forms of non-determinism, including system interactions (system calls, interrupts, DMAs etc.) and multi-processor interactions. We extend iDNA to provide the ability to replay with two different thread interleavings for a data race, and provide the ability to examine the results of both orderings to see if they result in the same execution. In using our approach in a development environment, iDNA is first used to gather the replay logs for the product's test scenarios, with an overhead of about 10x on average [6]. We then run our off-line replay analysis to find all of the data races, and the off-line analysis classifies the data races into potentially benign or potentially harmful. For the potentially harmful ones, we provide at least two replay scenarios that will show how the data race can result in two different outcomes. This information, coupled with the ability to do reverse execution (also called time travel debugging) using iDNA [6] for the replays, provides a powerful platform for the developers to examine the potentially harmful data races.

We discuss our experiences in using our dynamic race classification approach on an extensively stress-tested build of Microsoft's Windows Vista and Internet Explorer. Our proposed technique was able to automatically filter out over half of the real benign data races, classifying them as potentially benign, which can be ignored by the developers. In addition, all of the harmful data races were correctly classified as potentially harmful. They were then reported to the developers, and they all have been fixed in the production code. Experiences with this offline data race detection tool has demonstrated the utility of providing support for deterministic replay debugging support.

V.B Finding Happens-before Replay Data Races

In this section, we provide a brief overview of the iDNA's [6] record and replay mechanism. It implements a checkpointing and logging mechanism similar to the BugNet mechanism (described in Chapter III) using a dynamic instrumentation tool. We then discuss a happens-before based data race detection algorithm that we implemented by extending the iDNA replayer. The happens-before based data race detector presented here does not report false positives at all. That is, if our detector finds a data race in a program's execution, then it is guaranteed that there is at least one instance of the data race in the execution, where two memory operations (read-write or write-write) not ordered by any synchronization operation.

V.B.1 iDNA Recorder

iDNA [6] provides the ability to record a multi-threaded program's execution in a replay log, which can be used to replay the execution. Here we will briefly discuss how iDNA works. More details can be found in [6].

iDNA uses a load-based checkpointing scheme (described in Chapter III) to record a program's execution. Let us first consider a single threaded application. At the beginning of a checkpoint interval, iDNA records the architectural state consisting of the values in the registers and the program counter. And then during the program execution, iDNA dynamically instruments the load instructions and records their values. The log size generated is reduced using a compression mechanism. The compression mechanism is a software implementation of the first-load optimization that we presented in Chapter III. More details on the software implementation can be found here [6].

Just like in BugNet, recording the values of load instructions executed by a program automatically takes care of all forms of non-determinism, including

system interactions (system calls, interrupts, DMAs) and multi-threaded interactions (even when multiple threads are executing on multiple processors). For example, if a system call or an interrupt modifies a memory location, the program needs to load the value from the memory location before it can use the value. Therefore, recording the values of the load instructions is sufficient to capture the system interactions. Even DMAs that concurrently modify the program's memory state can be taken care of by logging the load values. Also, in the case of multi-threaded programs, multiple threads may concurrently modify a shared memory location, but as long as we record the load values for a particular thread, we can replay that thread.

Note, iDNA does not log every load value. Just like in BugNet, it records only the load that accesses a memory location for the first time. In addition, if a memory value is modified by the external system (DMA, system call, another thread, etc.) outside of the thread, then the value of the subsequent load to that location is logged. iDNA also correctly deals with the dynamically loaded libraries and self-modifying code. A detailed explanation on how all this is done efficiently can be found in [6]. A more direct software-based implementation of BugNet's checkpointing and logging mechanism can be found here [58].

V.B.2 Sequencers for Multi-Threaded Programs

In the case of multi-threaded programs, a replay log is recorded for each thread individually. As we described above and in Section III.B.6, the replay log for a thread contains the initial architectural state of the thread and all the load values that are necessary to replay that thread correctly. Even if other threads are concurrently modifying the shared memory locations, it does not affect how a thread is replayed. Because, iDNA logs the values of all the required load instructions in the replay log. Thus, using the replay log of a thread, we can

replay the thread exactly how it was executed during the original execution.

However, to aid interactive debugging during replay and to enable multi-threaded program analysis, we want the ability to replay the thread interactions observed during the original execution. Instead of using Strata that we described in Chapter IV, iDNA provides this functionality by recording what are called *Sequencers*. A sequencer log consists of a global time-stamp value that is maintained by iDNA (one global time-stamp counter maintained across all the threads). The global time-stamp is incremented whenever a sequencer is logged in the replay log of any thread.

A sequencer is recorded when a synchronization instruction or a system call is executed. iDNA dynamically instruments the instructions with the lock prefix to recognize the synchronization operations. Whenever a synchronization operation is executed by a thread, a sequencer is logged. Since each sequencer consists of a time-stamp that is incremented monotonically, there exists a total order between all the sequencers recorded across all the threads. Figure V.1 shows an example for how sequencers are recorded in the replay log of each thread. The sequencers are labeled as S1, S2, etc. For the example, assume that $Si > Sj$ if $i > j$. With this log, we can determine that all the memory operations that were executed before the sequencer S1 in the thread T2, should have been executed before all the memory operations that were executed after the sequencer S3 in the thread T1 (because time-stamp for S3 is greater than S1).

V.B.3 iDNA Replayer

To replay a thread using a replay log, iDNA uses a dynamic instrumentation based replayer like the one we described in Section III.E. First the architectural state of a thread comprising of the registers and the program counter are initialized with the information read from the log. iDNA records both the

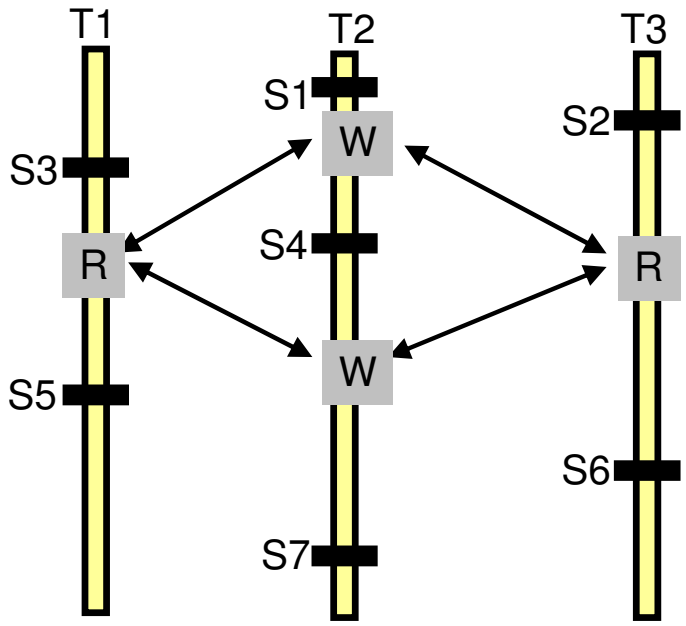

Figure V.1: Happens-before based race detection during replay using sequencers in the replay log.

code and the data that is read by a thread. So during replay, the execution starts from the instruction pointed to by the program counter. The load instructions are then dynamically instrumented so that the iDNA replayer can make sure that replayed loads return the correct values as recorded in the replay log.

In the case of multi-threaded programs, one sequencing region is replayed at a time. A sequencing region consists of the sequence of instructions executed between two consecutive sequencers logged in an iDNA log for a thread. For the example in Figure V.1, the instructions executed between $S3 - S5$ constitute a sequencing region. One sequencing region is replayed at a time and it is chosen from one of the thread as follows. The sequencing region that has the smallest starting sequencers among all the sequencing regions that are yet to be replayed is chosen for replay. For example, in the Figure V.1, the sequencing region $S1 - S4$ is replayed before $S2 - S6$. After replaying $S2 - S6$, the region

$S3 - S5$ is replayed and so on.

V.B.4 Finding Happens-Before Data Races

Using iDNA, we replay a multi-threaded program's execution using the above sequencers. We modified iDNA to analyze the program's execution to find data races between sequence regions during replay.

To find the data races, we use the sequencers recorded in the iDNA traces. Using the sequencers, we can determine the overlapping sequencing regions across different threads in a multi-threaded program execution. For example, in the Figure V.1, the instructions executed between $S3 - S5$ constitute a sequencing region. It overlaps with the sequencing regions $S1 - S4$ and $S4 - S7$ in the thread T2, and also with the sequencing region $S2 - S6$ in the thread T3. In other words, there is no happens-before relationship between the memory operations executed in the overlapping sequencing regions.

We then detect a data race using the following happens-before algorithm. If we find two memory operations in two overlapping sequencing regions, and at-least one of them is a write, then we consider that the two memory operations to be involved in a data race. There is a data race between those two memory operations, because there is no sequencer separating the two in time to specify an order between them. If there is no sequencer between two memory operations, then it implies that there was no synchronization operation that was executed during the program's execution to guard the shared memory accesses. Therefore, there is a data race between the two memory operations.

V.C Classifying Data Races by Replaying Both Orderings

In the previous section, we described how we find a set of data races in a given program's execution by looking for memory operations that are not ordered

by a happens-before relation. In this section, we present a replay-based dynamic analysis algorithm that automatically classifies data races as either potentially benign and potentially harmful.

V.C.1 Overview

Consider a data race between two memory operations executed in a particular execution of a multi-threaded program. The two memory operations involved in the conflict would have been executed in a particular order during the original execution recorded by iDNA.

During replay, there are two possible orders between the two memory operations involved in the data race. (If we consider more than two data races at a time, then there could be more than two orders between the memory operations, but in this study, we consider only one data race at a time). In our analysis, we replay the program's execution twice for both of those two orders and compare the memory and register live-outs of the two replays. If the live-outs are the same, then we classify the analyzed dynamic instance of the data race as potentially benign. Otherwise, it is classified as potentially harmful.

Figure V.2 shows a piece of code to illustrate how the proposed analysis works. This is a sanitized example of one of the harmful data races found during our analysis on production code. The example code essentially decrements a reference counter value. Then, it reads the reference counter value. If the value is zero it frees the memory pointed to by the variable "foo". Assume that there are two threads executing the same piece of code in parallel and that the programmer, by mistake, did not use any synchronization operations to guarantee the correct parallel execution.

The figure also shows two possible orderings for the memory operations when this piece of code is concurrently executed by two threads. The values of

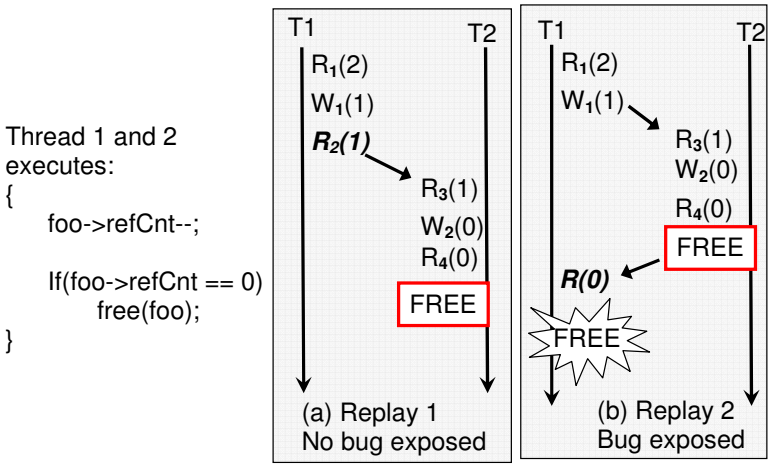

Figure V.2: Race Detection Example

the memory operations are shown inside the parenthesis. Figure V.2(a) shows the order observed during recording. Fortunately, for this ordering, the atomicity of the operations was not violated and hence the program executes correctly. However, during our dynamic analysis, we will detect data races between the read and the write operations executed in the two threads. For example, there is a data race between the read R2 in thread T1, and the write W2 in thread T2. During dynamic analysis, we can replay for the two possible orders between these two memory operations. One replay will be the same as the one observed during the original execution that was recorded as shown in Figure V.2(a). Another possible order is shown in Figure V.2(b). In the latter order, the R2 is replayed after W2. During that replay, we will catch a null pointer violation, when the replay tries to free the location "foo" in the thread T1. Thus, we determine that the data race is potentially harmful.

In contrast, when we examine the two orderings and both evaluate to the same result, then we classify that instance of the data race as potentially benign. However, if even one of the instance of the data race was found to be

potentially harmful, then we classify the data race to be potentially harmful. We next describe how we perform our replay analysis to determine if the two orderings arrive at the same result or not.

V.C.2 Mechanism for Alternative Replay

The above discussion assumes that it is possible to replay the two possible memory orders. We had to add the following support to provide this functionality on top of the iDNA replayer.

Our algorithm analyzes each data race in isolation. For a given data race, the goal is to replay the two possible memory orders between the two memory operations involved in the data race. The two memory operations involved in the data race are part of two sequencing regions in two different threads (a sequencing region constitutes the instructions executed between two sequencers in a thread, which we described in Section V.B).

By replaying the program's execution, we can determine all the instructions executed in each of the two sequencing regions that contain the data race. Using this information, we know which two dynamic instructions are the data race being considered, and we can examine both orders of those two instructions during replay. We replay both threads for the region up until we get to the data race instruction in each thread. We then replay the two orders to examine the differences exhibited by the data race. The first order is called the *original order*, since it matches the values seen during the original logged execution. The second order is called the *alternative order*. In our current implementation, we replay only till the end of the sequencing regions in the two threads.

In order to execute the instructions in the two sequencing regions for the two orders, we added to the iDNA replayer an ability to create a virtual processor. The virtual processor allows us to start with a set of sequences across the threads

and execute multiple different realities starting at that set of sequences. A virtual processor is created to execute the original and alternative replay orders. The virtual processor is initialized with the live-in memory values and the register states of the two threads. We orchestrate the execution of the two threads in the virtual processor to obey the ordering for the instructions involved in the data race. Whenever a memory location is read for the first time in the virtual processor, the virtual processor copies the value from the live-in memory. Then from that point on, the reads and writes to that memory location will be to the local copy in the virtual processor.

Alternative Replay Failure

While executing the alternative ordering, the replayer may come across a memory reference to an address not seen when the original log was taken, or it may come across a control flow change. The address may not have been logged or it may have been changed during replay, so we do not know what the value is. For the control flow change, it may jump to a piece of code that was not recorded as part of the logging or to an illegal address. In our current implementation, we classify all of these as *replay failures*. They are an indication that execution has changed enough from the alternative order that the data race is potentially harmful.

V.C.3 Classifying Data Races

After we have replayed the original and alternative orderings in the virtual processors, we compare the register and memory live-outs at the end of the sequencing regions to classify the data race as potentially benign or potentially harmful.

A data race between two memory operations may occur many times

during our analysis, and we examine each of those as a separate data race instance. Our current approach flags a data race instance as potentially benign only if the two replays result in exactly the same application state (both memory live-outs and register state) at the end of the replay. Otherwise, the data race instance is considered to be potentially harmful. The potentially harmful consist of the data races where the alternative replay resulted in different state, and also those that had a replay failure as described above.

After all of the instances for a data race have been examined, we classify the data race as potentially benign only if all of its instances are classified as potentially benign. Otherwise, the data race is classified as potentially harmful.

The data races classified as potentially benign are guaranteed to be benign for the test scenarios we examined, but they are not guaranteed to be benign for all possible scenarios. Another instance of the data race not captured in our replay logs between the same two memory operations may prove to be harmful. To add more confidence to our classification, several instances of the same data race should try to be found in the same execution or across the different test scenarios. If the replay analysis determines the data race to be potentially benign in all those instances, then we will have greater confidence that the data race is probably benign. The greater the number of instances studied, the greater is the confidence that a data race is benign.

For those data races that are classified as potentially harmful the two replays will enable the developer to have a better understanding of the effects of the data race that is reported. The two replays will show the differences in the outcomes of the program for the two different memory orders between the memory operations involved in the data race.

V.C.4 Advantages

Following are some of the key advantages of our offline replay-based data race detection and classification tool:

- Our analysis is at the instruction level and is not dependent on the specific synchronization methods. As a result, it is applicable to programs written in any language as it is agnostic to the synchronization methods used in the language. Our instruction based happens-before analysis does incur a heavy performance overhead. However, this is not a serious concern for our approach because we perform our analysis off-line during replay. An offline analysis does not interfere with the program's execution as the underlying replayer ensures that the execution is same as the one observed during recording. Therefore, our dynamic analysis can take more time to do the dynamic analysis.
- Unlike traditional approaches where it is hard to determine the possible effects of a data race, we will have two possible executions for the data race and produce the output for those executions. The ability to replay and see the differences in output between the two executions is of great value for the developer to understand the potential data race.

V.C.5 Future Work

Tools that produce a significant number of false positives tend not to get used by developers. So our goal in this study has been to reduce false positives as much as possible and for a data race that we find to be potentially harmful provide an example that clearly shows the effects of the data race. But coverage can also be improved in several different ways.

We used a data race detection mechanism based on the happens-before

relation, which does not have any false positives but has less coverage (that is more false negatives). However, our replay-based analysis can also be used to classify false positives in a data race algorithm that uses locksets [85] as potentially benign.

Another way to increase the coverage is to analyze the effects of many data races at the same time instead of analyzing one data race at a time. When we consider more data races together, we will have more than two memory orders to replay. If the live-outs of *any* two replays mismatch, we would classify the data race to be potentially harmful.

Our current implementation replays an alternative order that is sequentially consistent. If we consider a weaker memory consistency model, however, we will get more orders between the memory operations. Analyzing more orders between the memory operations will improve coverage.

The solution presented here can also be extended to be more precise in eliminating the potentially benign data races. Instead of comparing the live-outs of the two replays to classify a data race, we can classify based on whether we detect a perceivable bug in one of the two possible replays for the data race. Perceivable bugs are those that can be automatically detected when there is a memory access violation or assertion failure or any other exception. Thus, the harmful data races found will be guaranteed to be 100% correct.

V.D Results

In this section, we discuss our experiences in using our offline data race detection and classification tool. We found several harmful data race bugs in Microsoft's applications. They all were reported to the developers and have been fixed in the production code. Thus, the tool has clear value in assisting programmers in the debugging process.

Table V.1: Data Race Classification

	Potentially Benign		Potentially Harmful		
	Real Benign	Real Harmful	Real Benign	Real Harmful	Total
No State Change	32	0	-	-	32
State Change	-	-	15	2	17
Replay Failure	-	-	14	5	19
Total	32	0	29	7	68

However, the performance overhead of the dynamic analysis tool is high (280 times slower when compared to the native execution), because it performs many computations to find and classify the data races. Nevertheless, the analysis is performed offline and so it does not affect the execution behavior of the program. Therefore, we are able to tolerate the high performance overhead of our analysis tool. These results clearly demonstrate the utility of a deterministic replayer in building sophisticated dynamic analysis tools.

V.D.1 Methodology

We collected replay logs for 18 different executions of various services in Windows Vista and the Internet Explorer using iDNA recorder that we described in Section V.B. Among the 18 executions that we studied, the happens-before based algorithm that we described in Section V.B returned 16,642 instances of data race conflicts. Out of these 16,642 instances there were only 68 unique data races. The reason is that a data race (between the same two memory instructions in different threads) occurred more than once in the same execution or in different scenarios. For this study, we went through the painstaking effort to manually examine every single data race to determine if it was actually benign or harmful. All of the data races that were identified as truly harmful have been fixed in the production code.

The average log size for the replay logs collected using iDNA was about 0.8 bit per instruction. The total storage space required for the logs was 3.1 GB, which captured 33 billion instructions executed across all the different executions that we studied (about 96 MB to record a billion instructions). By compressing the log sizes using the Windows `zip` utility, we reduced the log sizes to about 0.3 bit per instruction.

To get an estimate for the time overhead for recording, replaying and analyzing programs we studied an execution of Internet Explorer, where we accessed a web site and browsed through a few pages. This study was carried out on a Pentium 4 Xeon 2.2GHz processor with 1GB RAM. The runtime performance overhead to collect the replay logs using iDNA [6] was about 6x when compared to the native execution. The iDNA replayer can replay the recorded execution with a performance overhead of 10x on average (relative to the native execution). However, using processor-based BugNet implementation we can reduce this performance overhead to less than 1% like we described in Chapter III. The execution had spawned 27 threads. When we ran our happens-before based race detection analysis, we found 2,196 instances of various data races. The overhead of executing the off-line happens-before race detection analysis was about 45x. The overhead of executing the replay analysis that we described in Section V.C to classify the data races was about 280x when compared to the native execution. We are able to tolerate the high performance overhead of our dynamic analysis, because it is incurred offline during replay where the program's execution is not affected.

V.D.2 Data Race Classification Results

We now described the accuracy of our tool in classifying the detection data races into potentially harmful and potentially benign categories.

Outcomes of Replay Analysis

We performed the replay-based data race classification analysis that we described in Section V.C over all the instances of the data races that were found using the happens-before algorithm. There are three possible outcomes when we perform the replay based analysis for an instance of a data race. The two replays for an instance may produce the same live-out. We call this outcome *No-State-Change*, because the memory order does not affect the state of the program's execution. Another possible outcome is that the two replays might produce different live-outs. We call this outcome *State-Change*. Finally, for some instances of data races we may encounter a replay failure while replaying for the alternative order for the reasons that we described in Section V.C. We call this outcome *Replay-Failure*. Note, that a replay failure is a good indicator that the data race is likely to cause a change in the program's state (in other words, the outcome is similar to State-Change).

There can be many instances for a given unique (static) data race. The final classification for a data race classifies the data race as No-State-Change only if all of its instances are No-State-Change. If, for any instance of the data race, the outcome was a State-Change, then we place the data race in the State-Change group. All of the remaining unique data races are classified as Replay-Failure. These are the data races for which none of the instances were classified as State-Change and at least one of the instances was classified as Replay-Failure.

Data Race Classification

Table V.1 presents the classification for all the unique static data races (a data race between the same two static instructions) that we studied. The rows in the table correspond to one of the three outcomes of the automatic replay analysis that we just described.

Based on the outcomes of the replay analysis for all the instances of a data race, our replay checker classifies the data race as either *Potentially-Benign* or *Potentially-Harmful*. These two classifications are shown in the table as the two aggregate columns. All data races classified as No-State-Change are potentially benign, and all data races classified as State-Change or Replay-Failure are classified as potentially harmful.

Table V.1 splits the potentially benign and harmful columns further into two groups: *Real-Benign* and *Real-Harmful*. The sub-columns correspond to the manual classification. In addition to the automatic classification, we also manually triaged each data race to determine if they were really benign or harmful. This was done to determine the accuracy of the automatic classification.

Potentially Benign Data Races

Table V.1 shows that out of the total 68 data races that we studied, 32 data races fell into the No-State-Change group. Since none of the instances of these data races can cause a state change or a replay failure, our automatic analysis classified these 32 data races as potentially benign. We manually verified each of these data races and found that they were all indeed benign.

Potentially Harmful Data Races

The data races accounted for in the second and the third rows in the Table V.1 were classified as potentially harmful. The reason behind this classification is that, in at-least one instance of a data race, if the outcome of the replay analysis was either a state change or a replay failure then it has the potential to be harmful. Based on this classification the automatic replay analysis classified 36 data races (17+19) to be potentially harmful.

Seven among the 36 data races were found to be harmful through

manual inspection, as listed in the sub-column named Real-Harmful under the Potentially-Harmful column. The automatic analysis correctly classified all the real harmful data races that it analyzed as potentially harmful. Two of these harmful data races are similar to the reference counting example that we discussed in Section V.C.

However, as we can see from the table not all of the potentially harmful races were found to be harmful in our manual classification. The sub-column named Real-Benign under the Potentially-Harmful column shows that 29 data races that were classified as potentially harmful are actually benign. The following are the two main reasons for the misclassification.

Misclassification Due to Approximate Computation: By manually inspecting these 29 data races, we found that 23 of them actually affect the execution of the program. As a result, our replay analysis will find a state change or a replay failure for most of the instances of these data races. Therefore, they were classified as potentially harmful. We took these potentially harmful data races to the developers. They described that these data races were left in the production code, because they chose to tolerate the effects of the data race rather than synchronize the code and lose performance. A good example where this kind of optimization is possible is the code region that was used to update a data structure maintaining statistics. In that case, the programmer consciously chose to gather approximate statistics and avoid the performance overhead required to accurately gather them. Another example is where the variable's value is used to make decisions that can affect only the performance and not correctness (e.g., time-stamp value used for making decisions on what to replace from a software cache). To optimize the synchronization overhead, programmers may choose to not synchronize operations on values such as time-stamps and statistics wherever appropriate. Since these data races were intended by the programmers, the are

classified as Real-Benign, even though they can change the program's execution.

Misclassification Due to Replayer Limitation: We now focus on the remaining 6 Real-Benign data races of the 29 data races that were incorrectly classified as Potentially-Harmful. When we manually triaged the six data races, we found them to be benign. Unlike the other 23 data races that we discussed earlier, these six data races did not affect the output or the state of the program. The reason why these six data races still got classified as Potentially-Harmful is that for at-least one of their instances, the outcome of the replay analysis was Replay-Failure. The replay failure was to due to the reasons that we described in Section V.C.2. When we manually analyzed these 6 replay failures, we actually found that the execution of the program wouldn't have been affected had the replays proceeded without failing. By adding additional support in iDNA to execute down unseen control paths, we should be able to correctly classify these six data races as no-state change and thereby classify them as potentially benign.

In conclusion, our approach classified 47% of the data races as potentially benign and they were all benign (none of them were harmful). Out of the other 53% of the data races that were classified as potentially harmful, only 20% of the 53% were found to be harmful.

V.D.3 Results for Each Dynamic Data Race Instance

Let us now discuss the results for the each of the instances that we analyzed for every static data race. We will also discuss the type of outcome that we obtained from the replay analysis for each instance.

Figure V.3 shows the number of instances that we analyzed for each of the 32 data races that were of the type No-State-Change, which we classified as Potentially-Benign. The number of instances for each unique data race varied from about 50 instances to just one instance. The greater the number of instances

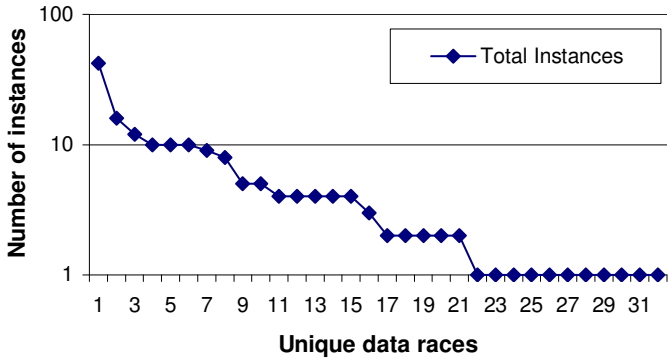

Figure V.3: Statistics for the unique data races classified as Potentially-Benign. Every instance of these data races resulted in No-State-Change and were actually Real-Benign. Total number of instances for each such data race are shown.

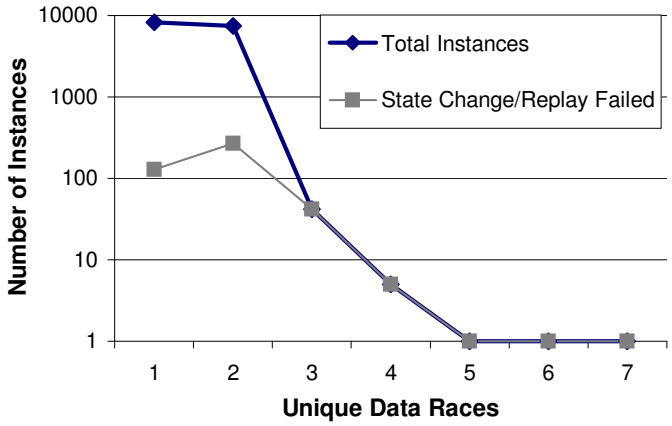

Figure V.4: Statistics for the unique data races that were classified as Potentially-Harmful and they were found to be Real-Harmful. Results are shown for total number of instances, and also for the number of instances that resulted in a State-Change or Replay-Failure.

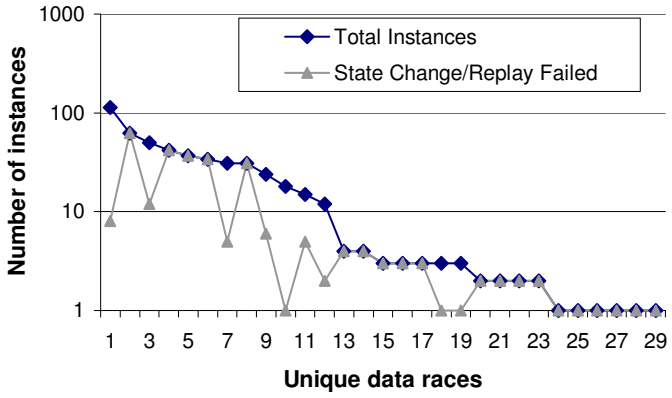

Figure V.5: Statistics for the unique data races that were classified as Potentially-Harmful, but they were actually Real-Benign. Results are shown for total number of instances, and also for the number of instances that resulted in a State-Change or Replay-Failure.

that we analyze and classify as No-State-Change, the greater the confidence we have in classifying them as Potentially-Benign.

Figure V.4 shows the number of instances that we analyzed for each harmful data race. As we can see, for some of the harmful data races we analyzed several thousand instances. However, only one in ten of those instances caused a replay failure or a state change. This shows that it is important to see those data races multiple times in order to catch them as Potentially-Harmful.

Figure V.5 shows the number of instances that we studied for the data races that we considered to be Potentially-Harmful, but when we analyzed them manually we found them to be Real-Benign. The main cause for this misclassification are the data races due to approximate computation, which we described in Section V.D.2.

V.D.4 Reasons for Benign Data Races

In this section, we describe the categories of benign data races that we were able to automatically classify as potentially benign.

1. **User Constructed Synchronization:** Programmers may construct their own synchronization primitives without using fences or the atomic operations provided in the instruction set architecture. For example, a garbage collector can maintain the reference counts for concurrent objects without using locks [23]. It is difficult to automatically infer the user constructed synchronization operations and so iDNA does not log a sequencer for a user constructed synchronization operation during the logging run. Because of this reason, the happens-before algorithm, will incorrectly classify a race between two user constructed synchronization operations, which is essentially correct synchronization, as a data race.

2. **Double Checks:** Double checks are used to optimize the synchronization overhead. A typical example for a double check is:

```
if(a) {    lock (..) { if(a) ... } }
```

   The read in the first check is not synchronized and so there can be a data race involving the read, but the data race is benign.

3. **Both Values are Valid:** Let us consider a data race between a read and a write operation. We found many instances where it is correct for the read operation to return either the old or the updated value (old value is the value in memory before the write and updated value is the value in memory after the write).

For example, when a buffer is shared between the producer and the consumer it can be correctly synchronized without using synchronization primitives. The producer writes to a buffer and increments the number of writes. The consumer reads the number of writes, and if it is greater than the number of entries it has consumed so far (referred to by a local variable), then it consumes an entry from the buffer. After consuming a value it updates its local variable representing the number of entries consumed. Without explicit synchronization, it is possible that the consumer might read a stale value for the buffer size. But that is fine, since it will just force the consumer to wait longer.

In another example, a shared variable was checked to decide which of the two versions of a function need to be used for doing a particular computation. Both the functions do exactly the same computation, but with different performance characteristics. The shared variable is written by another thread to specify the version of the function to be used. However, the read and the write need not be synchronized, because both versions will produce correct results, though one version might perform slower than the other.

Similar to this, we found the case where it just mattered if the memory value was zero or non-zero. The code was valid for multiple writers setting the memory value to non-zero without any synchronization, and it did not matter if the non-zero value written was the same.

4. **Redundant Writes:** If a write operation writes the same old value that already resides in the memory location then the data race between the write and a read operation in another thread will be benign (thus it can be considered as a special case of the previous category in the sense that both the old and the updated values are correct values to return for a read operation). In one of the programs we studied, we found that a thread was writing its

Table V.2: Benign Data Races.

	# Races
User Constructed Synchronization	8
Double Checks	3
Both Values Valid	5
Redundant Writes	13
Disjoint bit manipulation	9
Approximate Computation	23

process identifier returned by a system call to a shared variable read by another thread. The writes were redundant and did not affect the correctness of the program execution.

5. **Disjoint Bit Manipulation:** There can be data races between two memory operations where the programmer knows for sure that the two operations use or modify different bits in a shared variable. Programmers tend to use multiple bits in the same variable in order to optimize for performance.

Table V.2 shows the number of data races that we studied for each of the above categories of benign data races. It also shows that there were 23 data races that were due to approximate computation, which were mis-classified by the replay analysis. As we mentioned in Section V.D.2, there were six other benign data races that were misclassified as Potentially-Benign. These six were due to those benign data races that can affect the control flow of the program's execution. The rest of the benign data races were correctly classified as Potentially-Benign and the reasons for why they were benign are shown in Table V.2.

V.E Prior Work

The focus of this chapter is to provide an example for a dynamic analysis tool that can benefit from a deterministic replayer. The presented tool automatically finds data races and classifies them into potentially benign and potentially harmful categories.

The reader should refer Chapter II for a discussion on prior works that attempted to provide support for replaying a program's execution. In this section, we focus our discussions on prior work that dealt with race detection techniques, and compare the utility of our proposed replay-based data race detection tool with those techniques.

We classify prior work on data race detection into solutions based on static analysis and solutions based on dynamic analysis. We discuss both static and dynamic analysis tools in the following sections. Apart from data races, atomicity violations are another form of concurrency bugs. This section also discusses techniques that find atomicity violations.

V.E.1 Static Analysis

Data races can be found using type-based static analysis techniques [8, 32]. A type-based technique requires the programmer to specify the type of the synchronization operations [8, 32]. Automatically inferring information about the synchronization operations is difficult and there are some techniques that address this problem [84]. Static analysis can also be done using model checking techniques like BLAST [36] and KISS [75]. Model checking techniques can handle various synchronization idioms and can also produce counterexamples. The limitation of the model checking techniques is that the analysis algorithm does not scale well for large programs, which can limit their use.

There are techniques that statically implement a lockset [85] based al-

gorithm [93, 30, 72]. Naik et al. [56] recently proposed an analysis method that consists of a set of techniques that are applied in series like reachability and alias analysis to reduce the number of false data races.

The primary limitation of the static analysis techniques is their accuracy in terms of the number of false positives reported. Also, an even bigger problem (which is true even for existing dynamic analysis techniques) is that, among the true data races reported, a large proportion of them are benign data races. Benign data races are very hard to distinguish from the harmful data races during static analysis. For example, in one of the very recent proposals [56], for one program `jdbm`, the analysis returned 91 true (not false positive) data races. However, only 2 of them were found to be harmful. Static analysis techniques address this problem with manual annotations, but they require the programmer to get the annotation right, and there is a significant amount of existing code existing without annotations.

Dynamic analysis technique presented in this chapter can significantly reduce the number of candidate data races that need to be examined. The trade-off of course is that, the coverage will be lower than the static techniques. Also, our replay-based analysis can generate different replay scenarios for a data race found during a dynamic analysis. The user can use this information to understand the possible effects of the data race on a program's execution.

V.E.2 Dynamic Analysis

Dynamic analysis can be done either on-line or off-line. On-line analysis is one where the analysis is performed during the execution of a program. Whereas, off-line analysis is one that is performed offline during replay. Of course, the latter approach requires replay support for multi-threaded programs.

We first examine the trade-offs between the two approaches. We then

describe the dynamic race detection techniques in more detail and place our analysis tool in context.

When the Analysis is Performed: On-line Versus Off-line Analysis

A program's execution can be analyzed *on-line* when the program is executing to detect the data races. This approach incurs runtime overhead, and hence the dynamic analysis needs to be efficient in terms of performance. A majority of prior dynamic race detection techniques have focused on detecting data races on-line, either with the instrumentation support [85] or with the hardware support [1, 77]. There has also been attempts to ameliorate the performance cost of dynamic analysis using static optimizations [25, 108, 66, 71].

Alternatively, if we can efficiently record sufficient information about a program's execution to allow us to deterministically replay the execution, then we can do *off-line* analysis. The advantage of off-line analysis over on-line analysis is that the analysis itself does not have to be as performance efficient as it is has to be for on-line analysis. Only the recording part needs to be efficient. We can perform (many) sophisticated time consuming dynamic analysis over a recorded program's execution off-line. Also, the result of the analysis can enable the developer to examine the source of the data race by replaying the program.

There have been a few techniques that looked at doing off-line analysis [16, 79] to detect data races. Like we described in Chapter II, both RaceFrontier [16] and RecPlay [79] do not attempt to record the non-deterministic interactions between the threads. As a result, they are limited in their analysis in that they are able to detect only the first data race in the recorded program execution. In contrast, we use the iDNA [6] infrastructure that implements BugNet-like checkpointing mechanism, which enables us to replay multi-threaded programs across all forms of non-determinism, including non-deterministic shared

memory multiprocessor interactions. This allows us to examine all the data races in the recorded program execution.

How the Analysis is Performed: Happens-Before Versus Lockset

Dynamic race detection algorithms can be broadly classified into happens-before based algorithms [43, 64, 1, 15, 24, 22, 86, 70, 80, 54], lockset based algorithms [85, 100, 65, 2] and hybrid algorithms that combine the two [25, 108, 66, 71].

One class of data race detectors use the lockset algorithm. The lockset algorithm checks whether each shared variable in a program is consistently guarded by at least one lock. Eraser [85] implements the lockset algorithm using instrumentation to dynamically find the data races during a program's execution. This algorithm has been extended to object-oriented languages [71] and improved for precision and performance [2, 65, 100, 14]. The lockset algorithm is essentially a heuristics based algorithm and hence reports data races that can never occur at all (that is, it can report false positives). A recent work [47] reports that a lockset algorithm resulted in thousands of false positives for scientific applications.

There are race detectors that use the happens-before algorithm. The happens-before algorithm checks whether conflicting accesses to shared variables in a program are ordered by an explicit synchronization operation or not. Many dynamic race detectors implement the happens-before algorithm in software [80] . Hardware [54, 74] and Distributed-Shared-Memory [70, 77] implementations were also proposed to reduce the runtime overhead of these detectors. A recent hardware based proposal called ReEnact [74] detects data races using happens-before relation on-the-fly. Upon detection of a data race, it can rollback to a previous checkpoint and replay the execution. During replay, it tries to avoid the data race detected in the previous execution. The advantage of using a happens-

before algorithm is that it can detect the data races with no false positives because the analysis is based on whether there are two unordered conflicting memory operations or not. However, the resulting coverage can be less than the lockset algorithm.

It is also possible to combine these two algorithms [25, 108, 66, 71] to get coverage close to a lockset algorithm, and at the same time reduce false positives using happens-before relations.

These prior dynamic data race detectors did not focus on classifying real data races as potentially benign versus harmful data races. For example, RaceTrack [108] found 48 warnings in CLR regression test suite out of which there were 8 false positives. But more importantly, 32 were benign data races and only 8 were found to be harmful during manual inspection. Distinguishing between the benign and the harmful data races is a hard problem. To our knowledge no prior work has attempted to automatically identify the potentially benign data races, which is possible using our analysis tool. If can do that, then we can direct the developer's effort towards triaging the potentially harmful data races.

The focus in building our tool was to provide as much accurate information as possible. That is why we chose to use a happens-before based data race detection algorithm, since it does not report any false positive. Nevertheless, our analysis can also be used for analyzing the data races reported by a lockset based algorithm and its variations. The analysis should be able to filter out the benign data races and also the false positives produced by those algorithms.

V.E.3 Atomicity Violation Detection

There have also been work on finding concurrency bugs by checking for atomicity violations [2, 27, 105, 47]. If we can know which regions of code need to be executed atomically, then we can verify the atomicity properties either stati-

cally [2] or dynamically [27, 105, 47]. Also, there has been work on inferring the set of locks that need to be acquired to enforce the atomicity specified by the programmer [37]. Any violation of atomicity is a source of a bug, but every data race is not necessarily harmful. So checking for atomicity violations is more effective than finding data races. However, determining the atomic regions in itself is a significant challenge. Many techniques require the programmer to explicitly specify the atomic regions through annotations [2, 27, 37]. In SVD [105] and AVIO [47], the authors used heuristics to infer the atomic regions automatically. These methods are heuristic based, and, as a result, they report a high number of false positives when a code region is incorrectly determined to be atomic.

V.F Conclusion

Traditionally, dynamic analysis tools cannot afford to spend a lot of time on analyzing a program's execution because they are done in real time and more analysis would mean that the execution behavior of the program would be affected. However, using a deterministic replayer, we can perform any sophisticated dynamic analysis offline, during replay. The overhead of the dynamic analysis tool would not affect the execution of the program as the underlying replayer would ensure that the replay is exactly same as the one observed in the original execution when it was recorded. If the recorder is efficient (which is indeed the case for BugNet), we can record a program's execution without interferring with its behavior. Then during replay we can perform any number of sophisticated dynamic analysis.

As a proof-of-concept, this chapter presented an offline dynamic analysis tool that automatically detected data races and prioritized the most harmful data races. This tool was built using the iDNA recorder and replayer. iDNA is based on BugNet-like checkpointing and logging mechanism that we described in

Chapter III.

Our replay-based analysis tool finds data races in a recorded program's execution using happens-before relation. We showed that 90% of real data races found in the production code are benign. To reduce the triage effort, the tool automatically identified and filtered the data races that are potentially benign. To automatically find out if a data race is potentially benign or not, the tool replays the execution twice, once for each possible order between the conflicting memory operations. If the two replays for the two orders produce the same result, then the tool classifies the data race as potentially benign.

In addition to reporting harmful data races, the analysis also produces very useful information to assist a programmer in debugging the data race. For every data race, the tool dumps out the replay log along with the memory orders corresponding to the data race. Using that information, a programmer can replay the program in two different ways and understand the effects of different memory orders that are possible due to the data race. This information can be a significant aid to the developer.

We discussed our experiences in using our dynamic race classification approach on an extensively stress-tested build of Microsoft's Windows Vista and Internet Explorer. Our proposed technique was able to automatically filter out over half of the real benign data races, by classifying them as potentially benign, allowing the developers to ignore them. In addition, all of the real harmful data races were correctly classified as potentially harmful. The harmful data races that we found were reported to the developers, and all of them have been fixed in the production code. These results clearly demonstrate the utility of a deterministic replayer in building sophisticated dynamic analysis tools.

Acknowledgments

The text in this chapter is in part a reprint of the material from the paper, Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder, "Automatically Classifying Benign and Harmful Data Races Using Replay Analysis", in *ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)*, June 2007. The dissertation author was the primary researcher and author and the co-authors involved in the publication [62] directed, supervised, and assisted in the research which forms the basis for that material.

VI

Conclusion and Future Work

As we enter the many-core era, where we may have a processor with 100s of cores, harnessing parallelism through multi-threaded programming will be very important. However, until now, mostly only the expert programmers have been writing multi-threaded programs. This trend is set to change, because in the future, every programmer has to learn multi-threaded programming to take advantage of the many-cores. Therefore, it is important that we develop adequate programming methodologies and tools to support program development.

Until now, processor industry's focus has primarily centered around improving the performance of their processors. To address the future challenges in developing software for many-core processors, however, we need to provide processor support for program development.

This thesis identifies the need for deterministic replay support. Providing an ability to reproduce non-deterministic bugs (such as data race bugs in the multi-threaded programs), and the bugs that occur at the customer sites is an important problem. Deterministic replay support can solve this problem, but it would require an efficient implementation of a recorder. This thesis presents a processor-based solution for recording all forms of non-determinism that could influence a program's execution, and thereby we support deterministic replay. We

also demonstrate the utility of the deterministic replay support for automated debugging through a novel replay-based data race detection tool that finds concurrency bugs in the multi-threaded programs. This tool is currently being used at Microsoft for finding data race bugs in their applications. Many of the bugs found using the tool have already been fixed in their software systems such as Windows Vista and Internet Explorer.

In this section, we summarize our deterministic replay solutions, which mainly consists of two parts - BugNet for replaying non-deterministic system interactions, and Strata for replaying a multi-threaded program on a multi-processor system. We also summarize the key techniques used in the replay-based data race detection tool. Finally, we discuss how the deterministic replay solution presented in this work can be improved, describe some open problems, and also provide suggestions for developing new applications for deterministic replay support.

VI.A Summary

This section provides a summary of the solutions presented in this thesis for providing deterministic replay support.

VI.A.1 BugNet for Replaying Non-deterministic System Interactions

We focus on replaying just the user code and the shared libraries. Therefore, we have to record any non-deterministic input read from the operating system and the hardware layers below the operating system. Identifying and recording each source of non-determinism in the system is a complex solution to implement. We overcome this problem using a simple observation that we need to record just the input read by the application through the load values. Thus, BugNet's logs for a checkpoint interval contains the register state at the start of

the interval and a log of memory values (code and data) when they are first accessed. This is enough information to achieve deterministic replay of a program's execution, without having to replay what goes on during interrupts and system calls. More importantly, BugNet is a system-independent solution. If supported in a processor, the BugNet feature is useful for recording an application's execution on any operating system. Also, the logs collected for a program's execution on a particular system can be used to replay that program's execution on any other system.

Log size overhead is about 225KB for recording 10 million instructions. The performance overhead of our processor-based solution is less than 1%. We also showed that 10 million instructions are sufficient for debugging a majority of the bugs in open source programs. Thus, BugNet should be useful for recording a program's execution even during the production runs at a customer site.

VI.A.2 Strata for Replaying Non-deterministic Shared-Memory Multi-threaded Interactions

The threads in a shared-memory multi-threaded program running on a multi-processor communicate through the shared memory. These communications need to be synchronized. However, using the conventional synchronization primitives that are available today, the order of the memory operations across the threads is undefined. That is, the order of the memory operations executed across the threads is non-deterministic, and hence need to be recorded in order to replay a multi-threaded program's execution.

This thesis proposed a logging mechanism called *Strata*, which complements the BugNet architecture. Using strata, we can capture the shared-memory dependencies in both snoop-based and directory-based systems. A stratum provides a strict time ordering between the memory operations that occurred before

and after the stratum executed across all the processors. Using this information we can replay the memory order in a multi-threaded program, which helps in debugging concurrency bugs such as data races.

We found that for a directory-based system, the strata log is 5.8x smaller without compression and 12x smaller with compression when compared to the log size required in the previous point-to-point logging approach [104]. Another advantage is that our strata approach requires less hardware than the point-to-point approach. Also, unlike previous proposals [104, 106], strata is useful for recording a snoop-based multi-processor's execution.

VI.A.3 Applications of Deterministic Replayer for Automated Debugging

This thesis also demonstrated the utility of deterministic replay support for automated debugging. Dynamic bug detection tools such as Valgrind [88] cannot afford to spend a lot of time on analyzing a program's execution. This is because, Valgrind performs the analysis during a program's execution. As a result, the computation performed in the analysis code changes the behavior of the program that is analyzed by Valgrind.

One use of deterministic replay support is that we can record a program's execution and postpone all the dynamic analysis to the replay phase. If the recorder is efficient (which is indeed the case for BugNet), then we can capture the realistic behavior of a program running on a real system.

As a proof-of-concept, this thesis presented an offline dynamic analysis tool that automatically detected data races and prioritized the most harmful data races. One important result that we derived from using the tool to analyze real world applications is that, we found 90% of the true data races to be benign. Using a replay-based dynamic analysis, the tool automatically identified and fil-

tered the data races that are potentially benign. This analysis involved replaying an execution twice for each data race, once for each possible order between the racing memory operations. This was a useful analysis that reduced the number of data races that programmers have to analyze by 50%, and also found several harmful data race bugs in Internet Explorer and Windows Visa. All of those bugs have been fixed in the production code. Thus this tool has been shown to be very useful, but, as the analysis requires multiple replays, it incurs significant performance overhead, on average about 280 times when compared to the native execution. Such a high performance overhead in a dynamic analysis tool cannot be tolerated during a real program's execution without significantly affecting the program's behavior. However, deterministic replay support for offline dynamic analysis made it a viable solution.

VI.B Future Work

VI.B.1 Compressing BugNet's Logs

One of the limitation of BugNet and Strata is that the replay window length is limited by the amount of memory space available and the amount of logs generated. A straight-forward solution to ameliorate this problem is to compress the logs, and thereby increase the replay window length. In this thesis, we used a simple compression method based on frequent value locality in the programs (described in Chapter III). It reduced the BugNet logs by a factor of two.

However, if we assume that there will be idle cores in a many-core processor, then those cores can be used to continuously compress the logs collected in the active cores. We could execute a compression thread on an idle core that would compress BugNet's FLL (First Load Log) for a checkpoint interval. We have found that, if we employ a perfect first-load optimization [59] instead of

approximating it by using first-load bits in a finite sized cache, we can significantly reduce the log sizes (on average, we need only about 22 MB to capture a full execution of a SPEC benchmark that will have over 100 billion instructions, which is at least 10 times smaller than the log sizes we observed). Therefore, we can use a compression thread that would replay a checkpoint interval on an idle core, and during replay, it would keep track of a shadow memory for the entire memory state touched by the application. Thereby, it can perform a perfect first-load-optimization enabling us to capture much longer replay window lengths or significantly reduce the memory space allocated for BugNet.

VI.B.2 Reducing the Complexity of Strata and Supporting Relaxed Memory Models

Strata described in Chapter IV required non-trivial modifications to the coherency protocol and the memory sub-system to capture the shared-memory dependencies. However, we believe that this solution can be simplified using the following approach. In Section IV.B.4 we explained how using BugNet's First Load Log (described in Chapter III) for a thread we can deterministically replay that thread. Thus, we can determine the memory operations, their program order, their addresses, and their values using just the BugNet's logs. Using this information, we should be able to determine the order between the memory operations using a replay-based offline analysis. Perhaps, to bound the complexity of the offline algorithm we can log stratum-based hints (a hint is a stratum that contains the timestamps of all the threads) at pre-determined intervals. With the stratum-based hints, we just have to order the memory operations between two strata instead of ordering all the memory operations executed in the replay window.

We believe that the above offline analysis can also be extended to deter-

mine the memory order for a program executed on a multi-processor system with a weaker memory model. Using the values of the memory operations determined from the BugNet logs, we could determine the producer write operation for every read operation, and thereby determine the partial order between the memory operations.

VI.B.3 Using Virtual Machine Support with Strata

ReVirt [26] is a record and replay system based on Virtual Machine Monitor support. It is efficient (incurs about 10% performance overhead) and also it does not require processor support. However, a limitation of ReVirt is that it cannot support multi-processor replay. We believe there exists a hybrid solution, where we can use Strata-like support for recording multi-processor non-determinism and use ReVirt for recording non-deterministic system interactions. This approach could further reduce hardware requirements for deterministic replay, which would make it an even more attractive option for processor manufacturers to support deterministic replay.

VI.B.4 Open Problems in Supporting Deterministic Replay

One important problem that need to be addressed is non-technical in nature - a business incentive for providing support for debugging features. We need to think about a business incentive that would make it feasible for processor manufacturers to supports features like deterministic replay support. From the results in this thesis, it is quite clear that, to provide efficient replay support for multi-processor we need processor support. Also, it is clear that deterministic replay support is very useful for debugging non-deterministic concurrency bugs. Without deterministic replay support, multi-threaded programming would be pretty hard. Though such a feature would advance the computing field, it is still

not clear how a processor manufacturer might be able to increase their profit by supporting this feature. That said, in the past, processor manufacturers have indeed provided support for breakpoints and watchpoints. To bring many-cores and multi-threaded programming to the masses, features like deterministic replay are not a luxury but a necessity.

Privacy

Gathering information about a program's execution at the customer site and sending that to the developer to debug bugs in production code can be an effective strategy, except for one main issue for the customer - privacy. The recorded execution might contain sensitive information pertaining to the customer. One possible approach could be use an offline replay-based program analysis that would obfuscate the sensitive data but still makes sure that the execution is reproducible. This approach might not guarantee complete privacy, because any information that can be inferred about a program's execution can be considered a privacy leak. However, it could provide a reasonable solution, where it can make it hard for a developer to infer any sensitive information from the BugNet and Strata logs.

Detecting Incorrect Results

While using the deterministic replay feature at the customer site, we need an ability to detect the occurrence of a bug. The solutions we presented in this thesis can capture only those bugs that result in a crash. If the bug leads to an erroneous output, the proposed deterministic replay mechanism will not be able to detect the problem and collect timely information. To ensure software correctness and capture bugs at the customer site, architecture research is needed into providing low overhead hardware support to find bugs that lead to wrong

answers. At a developer's site, however, one can trace the full execution of a program, and so this might not be an issue.

One potential approach is to provide hardware support to allow developers to leave software checks in their code, which can trigger asserts or violations if an error occurs. Examples of these types of software checks include programmer defined asserts, bounds checking, dangling pointer checks, etc. Currently software vendors do not leave these software checks in their code because of the prohibitive execution overhead. Special instructions and hardware support should be able to provide a low overhead solution for this.

Another potential approach is to use hardware support to check for bug signatures that can occur during a program's execution. Current network systems detect worms and viruses based on anomalous patterns in the packets. Similarly, to track down software bugs during a program's execution, we may be able to use hardware support to classify a program behavior to be erroneous based on signature based anomaly detection. For example, a processor fault or a software bug might induce a program's execution to go down an infeasible path or trigger a spurious memory access pattern that is significantly different from a normal execution behavior. Such program level anomalies can be captured by detecting the anomalous patterns in processor events like branch mispredictions and cache misses. A key challenge here is to be able to monitor processor events and combine them to detect incorrect executions. 3D die stacking technology holds the promise for providing the required bandwidth to monitor the processor events. Hypervisor support can be used for analyzing the events and supporting recovery.

VI.B.5 More Applications of Deterministic Replay

In Section V.C.5 we discussed several possible extensions to our replay-based data race detection tools. Apart from data race detection, deterministic replay support can be used to build more dynamic analysis tools. For example, we can build tools that detect if a software system violates a privacy agreement, detect intrusions, perform performance optimizations, etc.

Dual Modular Redundancy for Multi-core Processors

To provide fault tolerance against transient errors, current systems such as IBM's zSeries and HP's Non-stop employ lock-stepping, where each logical processor has two microprocessors operating in lock-step. A fault is detected when their outputs mismatch. However, the cost of this dedicated hardware solution is very high. Moreover, lock-stepping two cores in future many-core processors will be impractical due to non-deterministic architectural optimizations (e.g. voltage scaling) that can alter the execution behavior of two cores. Without support for lock-stepping, one would require an ability to reproduce the input to a program's execution, so that the program can be re-executed and its output compared with the original execution. However, in a multi-core system executing a multi-threaded program, a thread's execution is dependent on the non-deterministic interactions with the other threads in the system. Therefore, we would require support like Strata to reproduce the memory order and validly compare the two executions in a dual modular redundant system.

Bibliography

[1] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting data races on weak memory systems. In *ISCA '91: Proceedings of the 18th Annual International Symposium on Computer architecture*, 1991.

[2] R. Agarwal, A. Sasturkar, L. Wang, and S. D. Stoller. Optimized run-time race detection and atomicity checking using partial discovered types. In *In Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 233–242, 2005.

[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, U.C. Berkeley, December 2006.

[4] A. Ayers, R. Schooler, C. Metcalf, A. Agarwal, J. Rhee, and E. Witchel. Traceback: first fault diagnosis by reconstruction of distributed control flow. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005.

[5] D. F. Bacon and S. C. Goldstein. Hardware assisted replay of multiprocessor programs. In *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 194–206. ACM Press, 1991.

[6] S. Bhansali, W. Chen, S. de Jong, A. Edwards, and M. Drinic. Framework for instruction-level tracing and analysis of programs. In *Second International Conference on Virtual Execution Environments*, June 2006.

[7] B. Boothe. Efficient algorithms for bidirectional debugging. In *Proceedings of the ACM SIGPLAN conference on programming language design and implementation*, pages 299–310, 2000.

[8] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming Systems, Languages, and Applications*, 2002.

[9] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[10] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[11] M. Burtscher and N. B. Sam. Automatic generation of high-performance trace compressors. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 229–240, 2005.

[12] J. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools, pp. 48-59, Welches, Oregon*, 1998.

[13] J. D. Choi, B. Alpern, T. Ngo, and M. Sridharan. A perturbation free replay platform for cross-optimized multithreaded applications. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, April 2001.

[14] J. D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 258–269, New York, NY, USA, 2002. ACM Press.

[15] J. D. Choi, B. P. Miller, and R. H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, 13(4):491–530, 1991.

[16] J. D. Choi and S. L. Min. Race frontier: reproducing data races in parallel-program debugging. In *PPOPP '91: Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 145–154, 1991.

[17] M. L. Corliss, E. C. Lewis, and A. Roth. Low-overhead interactive debugging via dynamic instrumentation with dise. In *Proceedings of 11th International Symposium on High-Performance Computer Architecture*, Feb 2005.

[18] F. Cornelis, M. Christiaens, M. Ronsse, T. Ghesquiere, and K. DeBosschere. A taxonomy of execution replay systems. In *Proceedings of the International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet*, 2003.

[19] F. Cornelis, M. Ronsse, and K. D. Bosschere. Tornado: A novel input replay tool. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 2003.

[20] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *12th USENIX Security Symposium, Washington DC*, August 2003.

[21] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, Jan. 1998.

[22] J. M. Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 24–33, 1991.

[23] D. L. Detlefs, P. A. Martin, M. Moir, and J. G. L. Steele. Lock-free reference counting. In *PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 190–199, 2001.

[24] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 1–10, 1990.

[25] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *PADD '91: Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, pages 85–96, 1991.

[26] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *5th Symposium on Operating System Design and Implementation*, 2002.

[27] T. Elmas, S. Tasiran, and S. Qadeer. Vyrd: verifying concurrent programs by runtime refinement-violation detection. In *PLDI*, 2005.

[28] E. N. Elnozahy, L. Alvisi, Y. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Survey*, 34(3):375–408, 2002.

[29] J. Emer, P. Ahuja, E. Borch, A. Klauser, C. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: A performance model framework. *Computer*, 35(2):68–76, 2002.

[30] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 237–252, 2003.

[31] S. I. Feldman and C. B. Brown. Igor: a system for program debugging via reversible execution. In *PADD '88: Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, pages 112–123, 1988.

[32] C. Flanagan and S. N. Freund. Type-based race detection for java. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 219–232, 2000.

[33] A. Georges, M. Christiaens, M. Ronsse, and K. D. Bosschere. Jarec: A portable record/replay environment for multi-threaded java applications. In *Software: Practice and Experience*, 2004.

[34] J. Gray. Distributed computing economics. Technical Report MSR-TR-2003-24, Microsoft Research, March 2003.

[35] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, 1992.

[36] T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 1–13, 2004.

[37] M. Hicks, J. S. Foster, and P. Pratikakis. Inferring locking for atomic sections. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, June 2006.

[38] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[39] Intel. The ia-32 intel architecture software developer's manual. *Instruction Set reference*, 2, 2001.

[40] M. S. Johnson. Some requirements for architectural support of software debugging. In *ASPLOS-I: Proceedings of the First International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 140–148, New York, NY, USA, 1982. ACM Press.

[41] G. Kane and J. Heinrich. Mips risc architecture. *Prentice-Hall*, 1992.

[42] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX 2005 Annual Technical Conference*, 2005.

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

[44] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transaction on Computers*, 36(4):471–482, 1987.

[45] R. B. Lee, D. K. Karig, J. P. McGregor, and Z. Shi. Enlisting hardware architecture to thwart malicious code injection. In *Proceedings of the International Conference on Security in Pervasive Computing (SPC-2003)*, March 2003.

[46] S. Lemon. Intel set for first public demo of pram. *http://www.techworld.com/storage/news/index.cfm?newsid=8552*.

[47] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 37–48, 2006.

[48] D. Lucchetti, S. K. Reinhardt, and P. M. Chen. Extravirt: Detecting and recovering from transient processor faults. In *Symposium on Operating System Principles work-in-progress session*, October 2005.

[49] 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 *Programming Language Design and Implementation*, Chicago, IL, June 2005.

[50] S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002.

[51] E. Marcus and H. Stern. *Blueprints for high availability*. John Willey and Sons, 2000.

[52] R. McLaws. Windows vista bug reports: An analysis. *http://www.windows-now.com/blogs/robert/archive/2006/07/05/Windows-Vista-Bug-Analysis.aspx*, 2007.

[53] Microsoft. Dr. watson overview. *http://oca.microsoft.com/en/dcp20.asp*.

[54] S. L. Min and J.-D. Choi. An efficient cache-based access anomaly detection scheme. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, pages 235–244, 1991.

[55] G. J. Myers. *Advances in Computer Architecture*. John Wiley & Sons, New York, 1978.

[56] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 308–319, 2006.

[57] S. Narayanasamy, C. Pereira, and B. Calder. Recording shared memory dependencies using strata. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 229–240, 2006.

[58] S. Narayanasamy, C. Pereira, and B. Calder. Software profiling for deterministic replay debugging of user code. In *5th International Conference on Software Methodologies, Tools and Techniques (SoMET)*, Oct 2006.

[59] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder. Automatic logging of operating system effects to guide application-level architecture simulation. In *International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS)*, June 2006.

[60] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *32nd Annual International Symposium on Computer Architecture*, June 2005.

[61] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Recording application level execution for deterministic replay debugging. In *IEEE Micro Special Issue: Top Picks from Computer Architecture Conferences*, January 2006.

[62] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)*, June 2007.

[63] Netscape. Netscape quality feedback system. *http://wp.netscape.com/communicator/navigator/v4.5/qfs1.html*.

[64] R. H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 1–11, 1993.

[65] H. Nishiyama. Detecting data races using dynamic escape analysis based on read barrier. *Third Virtual Machine Research & Technology Symposium*, pages 127–138, May 2004.

[66] R. O'Callahan and J. D. Choi. Hybrid dynamic data race detection. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 167–178, 2003.

[67] E. I. Organick. Computer system organization: The b5700/b6700 series. page 132, 1973.

[68] D. Z. Pan and M. A. Linton. Supporting reverse execution for parallel programs. In *Proceedings of the 1988 ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 1988.

[69] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, and N. Treuhaft. Recovery oriented computing (ROC): Motivation, definition, techniques, and case studies. In *UC Berkeley Computer Science Technical Report UCB/CSD-02-1175*, Berkeley, CA, March 2002. U.C. Berkeley.

[70] D. Perkovic and P. J. Keleher. Online data-race detection via coherency guarantees. In *OSDI*, pages 47–57, 1996.

[71] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 179–190, 2003.

[72] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 320–331, 2006.

[73] M. Prvulovic. Cord: Cost-effective (and nearly overhead-free) order-recording and data race detection. In *International Symposium on High-Performance Computer Architecture*, Feb 2005.

[74] M. Prvulovic and J. Torrelas. Reenact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *30th Annual International Symposium on Computer Architecture*, San Diego, CA, June 2003.

[75] S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 14–24, 2004.

[76] F. Qin, S. Lu, and Y. Zhou. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *Eighth International Symposium on High Performance Computer Architecture*, Feb. 2005.

[77] B. Richards and J. R. Larus. Protocol based data race detection. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 40–47. ACM Press, 1998.

[78] M. Ronsse and K. D. Bosschere. Debugging backwards in time. *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG)*, Sep 2003.

[79] M. Ronsse and K. de Bosschere. Recplay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2):133–152, 5 1999.

[80] M. Ronsse and K. de Bosschere. Non-intrusive on-the-fly data race detection using execution replay. In *Proceedings of Automated and Algorithmic Debugging*, Nov 2000.

[81] M. Russinovich and B. Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 258–266, New York, NY, USA, 1996. ACM Press.

[82] O. Ruwase and M. Lam. A practical dyanmic buffer overflow detector. In *11th Annual Network and Distributed Security Symposium (NDSS 2004)*, pages 159–169, San Diego, California, February 2004.

[83] S. R. Sarangi, B. Greskamp, and J. Torrellas. Cadre: Cycle-accurate deterministic replay for hardware debugging. In *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks (DSN'06)*, pages 301–312, Washington, DC, USA, 2006. IEEE Computer Society.

[84] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated type-based analysis of data races and atomicity. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 83–94, 2005.

[85] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[86] E. Schonberg. On-the-fly detection of access anomalies. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI)*, 1989.

[87] S. Sethumadhavan, R.Desikan, D.Burger, C.R.Moore, and S.W.Kecler. Scalable hardware memory disambiguation for high ilp processors. In *International Symposium on Microarchitecture*, 2004.

[88] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX'05 Annual Technical Conference*, April 2005.

[89] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. Safetynet: Improving the availability of shared-memory multiprocessors with global checkpoint/recovery. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 123–134, 2002.

[90] SPARC. The sparc architecture manual: Version 8. *Prentice-Hall*, 2, 1992.

[91] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference*, pages 29–44, 2004.

[92] R. Stallman, R. Pesch, S. Shebs, and R. M. Stallman. *Debugging with GDB:The GNU Source-Level Debugger*. Free Software Foundation, 2002.

[93] N. Sterling. Warlock - a static data race analysis tool. In *Proceedings of the USENIX Winter Technical Conference*, pages 97–106, 1993.

[94] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jrapture: A capture replay tool for observation-based testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2000.

[95] M. Sullivan and R. Chillarege. Software defects and their impact on system availability. In *21st International Symposium on Fault Tolerant Computing*, Montreal, 1991.

[96] G. Systems. Geodesic traceback - application fault management monitor., 2003.

[97] G. Tassey. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. NIST, Research Triangle Park, NC, May 2002.

[98] F. Tip. Generic techniques for source-level debugging and dynamic program slicing. In *TAPSOFT '95: Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 516–530, London, UK, 1995. Springer-Verlag.

[99] N. Tuck, B. Calder, and G. Varghese. Hardware and binary modification support for code pointer protection from buffer overflow. In *37st International Symposium on Microarchitecture*, pages 209–220, 2004.

[100] C. von Praun and T. R. Gross. Object race detection. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 70–82, 2001.

[101] Wikipedia. Source lines of code . *http://en.wikipedia.org/wiki/Source_lines_of_code*.

[102] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium*, pages 149–162, February 2003.

[103] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *22nd Annual International Symposium on Computer Architecture*, pages 24–36, 1995.

[104] M. Xu, R. Bodik, and M. Hill. A flight data recorder for enabling full-system multiprocessor deterministic replay. In *30th Annual International Symposium on Computer Architecture*, San Diego, CA, 2003.

[105] M. Xu, R. Bodik, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[106] M. Xu, M. D. Hill, and R. Bodik. A regulated transitive reduction (rtr) for longer memory race recording. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 49–60, 2006.

[107] J. Yang and R. Gupta. Energy efficient frequent value data cache design. In *IEEE/ACM 35th International Symposium on Microarchitecture*, pages 197–207, 2002.

[108] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 221–234, 2005.

[109] X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental evaluation of using dynamic slices for fault location. In *Sixth International Symposium on Automated and Analysis-Driven Debugging*, 2005.

[110] P. Zhou, W. Liu, F. Long, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. Accmon: Automatically detecting memory-related bugs via program counter-based invariants. In *37th International Symposium on Microarchitecture (MICRO)*, Nov 2004.

[111] P. Zhou, F. Qing, W. Liu, Y. Zhou, and J. Torrellas. iwatcher: Efficient architecture support for software debugging. In *31st Annual International Symposium on Computer Architecture*, June 2004.

[112] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.