Relaxed Logging for Replay of Multithreaded Applications

Aliaksandra Sankova

Dissertation for the Degree of Master of
Information Systems and Computer Engineering

Jury

President:	Prof. Daniel Jorge Viegas Gonçalves
Advisor:	Prof. Doutor Luís Eduardo Teixeira Rodrigues
Member:	Prof. Doutor Vasco Miguel Gomes Nunes Manquinho

July 2015

Acknowledgements

Firstly, I would like to thank my advisor, Prof. Luis Rodrigues, for guiding and supporting me over this year, and also for patience. I would like to express my gratitude to the members of thesis committee for their feedback and fairness. I would specially like to thank Nuno Machado and Manuel Bravo for the fruitful discussions and their helpful comments, they were constant source of guidance and support on every stage of the research work.

This project has been funded with support from the European Commission. This report reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Lisbon, July 2015

Aliaksandra Sankova

For my family, dear friends and colleagues who support me continuously.

Resumo

A banalização do uso de processadores com vários núcleos oferece novas oportunidades para explorar o paralelismo na programação. Infelizmente, a programação concorrente é uma tarefa inerentemente complexa, sendo frequente a ocorrência de erros no acesso a estruturas de dados partilhadas, erros estes que só se manisfestam quando ocorrem alguns encadeamentos (entre-muitos) das instruções das várias tarefas que se executam concurrentemente. Reproduzir o encadeamento exacto que manisfesta o erro pode ser uma tarefa muito complexa e morosa, sem a ajuda de ferramentas adequadas.

Neste contexto, uma das principais técnicas para atingir esse objetivo é a que foi designada por *gravação e reprodução*, a qual consiste em gravar a informação necessária durante a execução de um programa para capturar o encadeamento que manifesta o erro, de forma a que execução falhada possa ser reproduzida mais tarde. Esta técnica resolve o problema de reprodução do erro, mas, infelizmente, em muitos casos acarreta uma diminuição substancial no desempenho da execução da aplicação, devido ao custo da tarefa de gravação.

Esta dissertação apresenta um estudo das abordagens existentes para gravação e reprodução, reflete sobre as vantagens e limitações de cada sistema e propõe uma nova abordagem que tenta reduzir o custo de gravação, sem onerar em demasia o processo de reprodução. São apresentados resultados da avaliação de um protótipo desenvolvido para validar as ideias propostas.

Abstract

The advent of multi-core processors brought new opportunity to exploit parallelism in programs. However, developing concurrent programs is a sophisticated task, due the new type of bugs that may appear and that may be very difficult to find and to correct. In particular, when multiple threads access shared memory, a buggy parallel program can lead to subtle data races that generate errors in some particular thread interleavings. There is a growing interest on building tools that help to reproduce such interleavings, helping the programmer to correct the code.

One of the main techniques to achieve this goal is what has been called *record and replay*: it consists of logging relevant information during the execution of a program that allows the interleaving that causes the bug to be reproduced later. This technique solves the problem of bug reproduction but, unfortunately, in many cases it introduces a substantial slowdown in the execution of the application.

This dissertation presents a study of existing approaches to record and replay, reflects on trade-offs and decisions of each system, and proposes a new approach of relaxed logging that aims at reducing the cost of the record phase without introducing a substantial increase in the time required to execute replay phase. We have implemented a prototype to validate these ideas and have evaluated it using several benchmarks.

Palavras Chave
Keywords

Palavras Chave

gravação e reprodução

aplicações paralelas

erros de concurrência

depuração de código

ferramentas de software

Keywords

record and replay

parallel applications

concurrency bugs

debugging

software tools

Contents

List of Figures

List of Tables

Acronyms

OS Operation System

DMA Direct Memory Access

SMT Satisfiability modulo theories

SMP Symmetric Multiprocessing

SPE Shared Program Element

JSON JavaScript Object Notation

JVM Java Virtual Machine

SC Sequential Consistency

PSO Partial Store Ordering

TSO Total Store Ordering

1 Introduction

This thesis addresses the problem of reproducing bugs in concurrent programs that rely on the shared memory paradigm. We study mechanisms to ensure such bugs will appear during re-execution in a way that will empower the programmer to infer the nature of this bug and therefore allow fixing it.

1.1 Motivation

The possibility to reproduce bugs in concurrent programs is important and essential when we develop a software that exploits multithreading opportunities offered by multi-core processors. It can be used not only for debugging but also to treat security vulnerabilities and analyse performance. Our targeted object of study, concurrent programs that rely on the shared memory paradigm, can be characterized by different sequential threads that execute in parallel, quite often in different cores of a multi-processor, communicating with each other by reading and writing in shared variables. In order to coordinate and communicate among each other, such threads need to use explicit synchronization, such as *locks* or *semaphores*(Dijkstra 2002). In particular, when accessing shared data, the logic of the program must ensure that the threads use the required synchronization to avoid *data races*(Netzer and Miller 1989). A data race occurs when different threads access a shared data structure without synchronization, and at least one of those accesses is a write. Data races can be avoided by the correct use of synchronization primitives however these primitives are hard to master. An incorrectly placed or missing synchronization primitive may not only eliminate the race but even create other bugs, for example introduce *deadlocks*(Coffman, Elphick, and Shoshani 1971). Furthermore, the interleavings that cause the bug may happen only in some rare circumstances, and be very hard to reproduce. This makes the debugging of concurrent programs an extremely difficult and tedious task. In this context we studied techniques and tools that simplify the reproduction of concurrency bugs and came up with an idea of how the state of the art could be improved. Delving into the topic,

one of the main techniques to achieve the reproduction of concurrency bugs is what has been called *record and replay* (Choi and Srinivasan 1998; Georges, Christiaens, Ronsse, and De Bosschere 2004; Ronsse and De Bosschere 1999; LeBlanc and Mellor-Crummey 1987a; Park, Zhou, Xiong, Yin, Kaushik, Lee, and Lu 2009; Altekar and Stoica 2009; Huang, Liu, and Zhang 2010; Yang, Yang, Xu, Chen, and Zang 2011; Zhou, Xiao, and Zhang 2012; Jiang, Gu, Xu, Ma, and Lu 2014) (or *deterministic replay* (Huang, Zhang, and Dolby 2013)). Record and replay relies in instrumenting the application in order to record all sources of non-determinism at runtime, including inputs, interrupts, signals, and scheduling decisions. For multi-core environments, it is also necessary to record the order by which different threads have accessed shared variables. This way, deterministic replay can be achieved by re-executing the application while enforcing all points of non-determinism to comply with the information stored in the log. Previous work that studied the problem of reproducing bugs in concurrent programs discovered that faithfully logging a concurrent program's execution requires inserting additional synchronization (to ensure that the thread interleaving is correctly traced). This, in combination with the large amount of information that may be required to be captured, can induce an unacceptable slowdown in the application (Georges, Christiaens, Ronsse, and De Bosschere 2004; Ronsse and De Bosschere 1999; Altekar and Stoica 2009; Huang, Liu, and Zhang 2010). To address this issue, some approaches attempted to reduce the amount of synchronization used to register the log, the amount of information included in the log, or even both. For instance, one can trace only partial information during the production run and then, at replay time, use search techniques to infer thread interleavings that are compliant with the (partial) log information (Park, Zhou, Xiong, Yin, Kaushik, Lee, and Lu 2009; Jiang, Gu, Xu, Ma, and Lu 2014). However, since reducing the amount of information logged hinders the replay determinism, the challenge lies in identifying the best trade-off between recording cost and inference time.

This work focus on exploring a relaxation of the logging procedure, allowing to avoid synchronization when recording concurrent accesses to shared variables, in order to lower the recording cost. To allow this relaxation, we also devise a replay mechanism that can search the different interleavings that are compliant with the information stored in order to find de sequence that reproduces the bug.

1.2 Contributions

This work studies the main techniques to achieve the reproduction of concurrency bugs and explores a novel combination of techniques to achieve this goal. We decided to name our proposal Optimistic REcOrd and REplay (OREO). We relax the recording order of write operations on shared state. This makes the record operation more efficient at the cost of a more sophisticated replay phase. To address the challenges of finding the interleaving that causes the bug from the relaxed log we resort to SMT solvers, by encoding the information recorded as a set of constraints that any buggy interleaving must satisfy. More precisely, the thesis makes the following contributions:

- It introduces the new technique to record and replay concurrent programs.
- From an experimental evaluation of the proposed technique it provides insights on its advantages and limitations.

1.3 Results

The results of this work can be enumerated as follows:

1. a specification of the algorithms to trace, collect and analyze information in order to replay concurrent executions;
2. explanation of a constraint model used between the system and SMT solver in order to get a faithful replay;
3. an implementation of a prototype of this system;
4. experimental evaluation using real-world applications and third-party benchmarks with concurrency bugs.

1.4 Research History

The aim of our work was to explore the potential benefits that could be achieved from relaxing the logging phase, in particular, by avoiding locks when storing the order of write

operations. To address the replay based on relaxed information we first attempted to derive variants of the replay techniques described by Zhou, Xiao, and Zhang (2012). However, this approach proved to be unfeasible. Therefore, we opted to resort to the use of SMT solvers to find the interleaving that reproduces the bug, by encoding the information collected during the record phase as a set of constraints that the faulty schedule needs to satisfy. This work was performed in the fruitful collaboration with Nuno Machado and Manuel Bravo.

1.5 Structure of the Document

The rest of this document is organized as follows. For self-containment, Section 2 provides an introduction to record and replay and describes previous work related to the problems that we introduce. Chapter 3 presents the architecture and algorithms used in the proposed solution and Chapter 4 presents the results of the experimental evaluation study. Finally, Chapter 5 concludes this document by summarizing its main points and discusses future work.

2 Related Work

This chapter provides a description of the record and replay technique, a justification for its existence and alternatives to this approach, followed by a review of the related work that is relevant to the project. Section 2.1 introduces us to basic concepts of the technique, Section 2.2 presents some alternative approaches and Section 2.3 shows us main challenges; Section 2.4 describes the design choices and trade-offs that one has to take into account when implementing this kind of system; Section 2.5 presents an overview of the most relevant record and replay solutions and depicts the state of the art; Section 2.6 sums up the chapter.

2.1 Record and Replay

Record and replay of multithreaded applications (also known as deterministic replay) is a popular technique to reproduce non-trivial bugs, in particular bugs of non-deterministic nature. The goal of this technique is to allow a given execution of a program to be faithfully reproduced, which requires the ability to reproduce not only the inputs provided to the program, but also all non-deterministic events, such as interrupts and thread interleavings. In general, techniques that provide deterministic replay operate in two distinct phases: record and replay, which are better explained in Subsections 2.1.1 and 2.1.2 respectively.

2.1.1 Record Phase

Record phase can be further divided into two sub-parts, decision-making and data processing:

- define the relevant data: suggest inputs and shared variables that are important for future assurance of reproduction to follow the same steps as the original run.
- log suggested information: put into a trace file values of the inputs and shared variables, selected as relevant.

In this phase, important questions are granularity of log (discussed in Subsection 2.3.3) and performance overhead (discussed in Subsection 2.3.2).

2.1.2 Replay Phase

The *Replay phase* can also be divided into two sub-parts, consulting and analysing the trace and finally, re-execution:

- *read the log*: consult the trace, produced on the *Record phase*, we decide which steps we will take to repeat the original run. It may require transformation of the existing log to a new replay-specified log with usage of axillary data structures.
- *reproduce the execution*: re-execute the program in a controlled manner, ensuring that the run reconstructs the original execution. In other words, we force the replay of non-deterministic events that we recorded beforehand.

In this phase, important questions are performance overhead, referred together with recording overhead in Subsection 2.3.2, and the ability to reproduce the original run at all.

2.1.3 Why is it important?

Deterministic replay can be used for a wide range of applications. The most common among them are the following (Pokam, Pereira, Danne, Yang, King, and Torrellas 2009):

Debugging Deterministic replay is very useful in debugging and testing as it provides a possibility to hold cyclic re-executions and, therefore, travel in time within the program run. As a debug measure, cyclic re-executions may introduce an illusion of reverse execution (Boothe 2000; Feldman and Brown 1988; Zelkowitz 1973) and help to detect faults in various programs, sequential and with high probability in non-deterministic executions (Georges, Christiaens, Ronsse, and De Bosschere 2004; LeBlanc and Mellor-Crummey 1987a; LeBlanc and Mellor-Crummey 1987b).

Security The possibility to reproduce execution can be used to search exploits of newly discovered vulnerabilities (Joshi, King, Dunlap, and Chen 2005) analyse status of the system

under attack (Dunlap, King, Cinar, Basrai, and Chen 2002; King, Dunlap, and Chen 2003) or efficiently perform expensive security checks (Chow, Garfinkel, and Chen 2008).

Fault-tolerance Developers can use deterministic replay to recover the state of the system after a crash (Bressoud and Schneider 1996).

2.2 Alternatives

To better justify why we decided to focus on deterministic replay, we briefly present other techniques (even though we are not addressing them in this work):

Online Debugging This is a classical way to debug a program that is known from a majority of IDEs, where inline assertions and break points are used. Such an approach normally requires the developer to have an idea of of a bug's nature, which may not work when the bugs are introduced by data races.

Execution synthesis Automated version of online debugging, this technique relies on bug reports and static analysis (Zamfir and Candea 2010). It does not require runtime tracing of a program. Hence, it introduces no runtime overhead which can be seen as its main advantage. However, there are some bug reports that the system cannot process and, therefore, some bugs that the system cannot reproduce. Limitations of bug reports do not exist in the deterministic replay approach.

Fault Localization Naturally after failing reproductions, there is a technique that aims to discover the bugs and their nature before making an attempt to reproduce them. Many systems use various approaches: static or dynamic analysis (Lu, Park, Hu, Ma, Jiang, Li, Popa, and Zhou 2007), model-checking (Brat, Havelund, Park, and Visser 2000), etc. Such systems are prone to omission such as marking some bugs falsely positive or vice versa, treating normal code falsely negative.

Deterministic Execution Close to deterministic replay, deterministic execution provides the same ability to reproduce a program with main difference in targeted reduction of inherent non-determinism of the application. Kendo (Olszewski, Ansel, and Amarasinghe 2009a) tracks the progress of each thread using performance counters to construct a deterministic

logical time that is used to compute an interleaving of shared data accesses. CLAP (**?**) tracks thread local execution paths. The main drawback of such approach is its tendency to avoid such sources of non-determinism as user input.

Log-based Debugging Basing on logged information only, log-based debugging automatically infers possible cause of the bug. Real systems use machine learning techniques and data mining (Xu, Huang, Fox, Patterson, and Jordan 2009). The advantage that these systems provide in avoiding runtime overhead costs by operating offline is offset by the huge traces or by inability to reconstruct a detailed execution state due to lack of information.

2.3 Challenges

Although simple in theory, building a deterministic replay system poses several issues. Within the following subsections we discuss the most relevant challenges that need to be addressed in the context of record and replay.

2.3.1 Non Determinism

The main difficulty of concurrency bugs reproduction is their non-deterministic nature, which, from time to time, makes them disappear when specific interleaving does not occur. There are many sources of non-determinism in a system (Tanenbaum, Austin, and Chandavarkar 2013). Deterministic replay address this issue by capturing them, however, what to consider as non-deterministic input depends on which level we are operating on. As no system deals with all forms of non-determinism (Cornelis, Georges, Christiaens, Ronsse, Ghesquiere, and Bosschere 2003) and different systems target different levels of abstraction, let us consider the user and system level for better understanding, which kind of events we will capture after selecting particular granularity. We also differentiate input non-determinism and memory non-determinism.

2.3.1.1 Input Non-determinism

User Level At the user level, sources of non-determinism originate from the operating system (OS). Among them are certain system calls, signals, special architecture instructions (Pokam, Pereira, Danne, Yang, King, and Torrellas 2009).

- System Calls: various system calls become non-deterministic when they depend on external sources, such as information from a network card or a disk. As an example we can mention Unix system call $gettimeofday()$.
- Signals: possibility to receive asynchronous signals makes the program non-deterministic.
- Special Architectural Instructions: on some architectures various instructions are considered as non-deterministic. As an example we can mention the *rdtsc* x86 instruction, which reads the CPU's timestamp, or *rdpmc* x86, which returns value of the performance counter.

System Level: At the system level, major sources of non-determinism are signals from I/O devices, hardware interrupts, and writes performed by Direct Memory Access (Pokam, Pereira, Danne, Yang, King, and Torrellas 2009).

- I/O: majority of instructions consider memory mapped I/O, which does not guarantee that reads from these devices can be repeated. Therefore, they have to be recorded.
- Hardware Interrupts: used to notify the processor that some data can be consumed, hardware interrupts change the control flow of execution in an asynchronous way. A recorder needs to log the point at which the interrupt has arrived and the source of the interrupt (e.g. disk I/O, network I/O, timer interrupt, etc).
- Direct Memory Access (DMA): these are operations of writing directly to memory without process awareness. The recorder needs to log all the values written by DMA with respect to their timing.

2.3.1.2 Memory Non-determinism

Memory interleaving input non-determinism appears in concurrent execution. However, in multi-core machines, an additional source of non-determinism is present. This is the order in which all threads in the system access shared memory, known as memory races. Having memory races in execution means that during various runs of a program different threads may end up winning the race when trying to access a piece of shared memory. They occur between synchronization operations (synchronization races) or between data accesses (data races) (Pokam, Pereira, Danne, Yang, King, and Torrellas 2009).

- Synchronization Races: are used to determine the order of other operations in a program's execution: locks that provide mutual exclusion in a critical region, etc. However, the order in which synchronization operations themselves are executed is non-deterministic.

- Data Races: occur when there are two or more unsynchronized concurrent accesses to the same shared memory location and at least one of them is to perform a write operation.

To provide deterministic replay of any concurrent program on a single-processor system, it is enough to record thread order decisions made by the scheduler (Ronsse and De Bosschere 1999). On multiprocessor machine this will not be sufficient as threads actually execute simultaneously on different processors. Thus, the solution is to manage capture the outcome of each thread shared access, which is not trivial do to efficiently.

2.3.2 Runtime Overhead

The record phase may induce a non-negligible overhead during the execution of a program. The more information is included in the log, i.e., the finer the granularity of logging, the easier to ensure deterministic replay, but the larger it is the recording overhead. Approaches that opt to log less information, i.e., that perform logging at a coarser granularity, are less expensive, but make it harder to reproduce the original schedule in reasonable time. As such, there is an inherent trade-off between *logging accuracy* and *replay efficiency*. According to the taxonomy (Cornelis, Georges, Christiaens, Ronsse, Ghesquiere, and Bosschere 2003), good deterministic replay should be *time efficient* - the performance overhead should be minimized in order to maintain the use of the program acceptable.

2.3.3 Log Size

On the record phase we are saving information to some trace file or log. This log should have adequate size and the amount of space, dedicated to store that log, should be minimal. Hence, another relevant metric of quality of deterministic replay systems is the *space efficiency* during the record phase (Cornelis, Georges, Christiaens, Ronsse, Ghesquiere, and Bosschere 2003) - the size of the log file containing the non-deterministic events captured during the recording phase should be minimized as well.

2.3.4 Security Issues

When logging information during production runs, one must take into account that some collected data may be sensitive. This raises security and privacy issues if the log needs to be shared with others (for instance, if the log is sent to the developer). Therefore, the degree of user data disclosure during recording is also considered to be important when devising record and replay solutions. Despite that, in our work, we do not explicitly address this concern, because it is somewhat orthogonal to the techniques that we plan to experiment with.

2.4 Design Choices

A large diversity of techniques has been proposed to perform record and replay. This section concerns a brief taxonomy, where the main design choices regarding the development of a deterministic replay system have been identified, particularly, how the solution is implemented, approach taken to record and reproduce an execution of the program and level of determinism provided.

2.4.1 Implementation approach

Deterministic replay techniques are usually classified according to the amount of specialized support that is built in hardware for this particular purpose. In particular, we consider three different types of systems. For self-containment purposes, in Section 2.6, we will make an overview of systems using the three techniques above. However, our work we will be mainly focusing on software-only approaches as hardware modifications reduced by other two approaches nowadays are only available as simulations.

2.4.1.1 Hardware-Only Systems

These types of systems are able to provide deterministic replay with significantly lower recording overhead than in software solutions, supporting the replay of data-races on multiprocessors in particular. Examples of hardware-only implementations that are considered as the state of the art are given in (Montesinos, Ceze, and Torrellas 2008; Xu, Bodik, and Hill 2003).

Unfortunately, bringing these concepts to reality requires expensive non-commodity hardware. Implementing deterministic replay on hardware ties the approach to a particular architecture.

2.4.1.2 Hardware-Software Systems

This is hybrid approach that combines both hardware and software techniques, aiming at the reduction of necessary hardware modifications preserving gain in the recording time. The state of the art system (Montesinos, Hicks, King, and Torrellas 2009) represents this search for a sweet-spot between hardware cost and efficiency but, like pure hardware solutions, tie the approach to specific architectures.

2.4.1.3 Software-Only Systems

Despite of not being as efficient as the ones above, software-only implemented systems are the most common tendency in research of deterministic replay due to their advantage of being more general. Looking for support of deterministic replay on heterogeneous architecture, many solutions have been implemented so far (Georges, Christiaens, Ronsse, and De Bosschere 2004; Ronsse and De Bosschere 1999; Choi and Srinivasan 1998; LeBlanc and Mellor-Crummey 1987a; Olszewski, Ansel, and Amarasinghe 2009b; Bhansali, Chen, De Jong, Edwards, Murray, Drinić, Mihočka, and Chau 2006; Huang, Liu, and Zhang 2010; Yang, Yang, Xu, Chen, and Zang 2011; Jiang, Gu, Xu, Ma, and Lu 2014; Huang, Liu, and Zhang 2010; Altekar and Stoica 2009; Park, Zhou, Xiong, Yin, Kaushik, Lee, and Lu 2009; Zhou, Xiao, and Zhang 2012). They can be applied to many different off-the-shelf architectures. Some of them (Huang, Liu, and Zhang 2010; Zhou, Xiao, and Zhang 2012) are considered as the state of the art.

2.4.2 Record and Replay Approach

Another way of classifying record and replay systems consists of looking at the type of information that is recorded in the log. Here, it is possible to distinguish approaches as content-based, order-based, and search-based (Cornelis, Georges, Christiaens, Ronsse, Ghesquiere, and Bosschere 2003).

2.4.2.1 Content-Based Approach

Content-based systems are also called data-driven as they record and replay the data read by each instruction. Literally, they log the values of all the relevant inputs and shared variables read by each thread, such that the exact same values can be used during replay. The major drawback of this approach is that it generates very large logs and may induce severe slowdowns in the execution of the program, making approach inefficient (Bhansali, Chen, De Jong, Edwards, Murray, Drinić, Mihočka, and Chau 2006).

2.4.2.2 Order-Based Approach

Instead of tracking the values of variables, order-based systems track the control flow of the program (such as timing of interactions with program files or I/O channels) from a given initial state. these types of systems are also called control-driven. According to this approach it is not necessary to record every instruction to replay an execution, which allows to reduce the amount of traced data. However, read and write accesses to shared memory locations still need to be tracked in order to support the reproduction of the thread interleaving. Such tracking is called the *exact linkage* between reads and writes into shared memory. This technique imposes lower overhead on record phase and creates smaller amounts of logs comparing to content-based solutions. The main challenge with this approach is to ensure that the initial state of the program is exactly the same in both the original and the reproduction run. Unfortunately, the initial state may often depend on the availability of external resources such as cores in multicore processors, that could affect the internal state of the program being replayed. Furthermore, to ensure that the log faithfully captures the thread read-write linkage, it is generally necessary to introduce additional synchronization during the record phase. Thus, even if order-based approaches represent an improvement over pure content-based approaches, most systems implemented with this approach still induce a large overhead at runtime (Altekar and Stoica 2009; Huang, Liu, and Zhang 2010).

2.4.2.3 Search-Based

This approach was created with a goal to mitigate an extremely high cost for some applications incurred by assurance of bug reproducibility in every execution, which sometimes would cause up to 100x slowdown (Park, Zhou, Xiong, Yin, Kaushik, Lee, and Lu 2009). The main

idea is to not log the exact thread read-write linkage and instead, to rely on a post-recording phase to construct a feasible interleaving from a *partial log* (Altekar and Stoica 2009; Park, Zhou, Xiong, Yin, Kaushik, Lee, and Lu 2009; Zhou, Xiao, and Zhang 2012) or to infer the missing information(Huang, Zhang, and Dolby 2013). This way, it is possible to substantially reduce the recording overhead at the cost of smaller determinism guarantees and a potentially longer replay. However, since the search space increases exponentially with the amount of missing information regarding the ordering of thread shared accesses, search-based approaches need to carefully balance the inference time and the recording overhead. Since pure content- and order-based systems incur an overhead is too high to be practical, most recent solutions have adopted a search-based approach. Our solution follows this trend, as well.

2.4.3 Determinism Level

Very often producing the exact execution on replay is an excessively expensive task, thus systems calibrate the level of determinism they provide depending on the execution they try to replay and problems they try to tolerate. Some of these levels of determinism "granularity" are presented below, and for better understanding are accompanied with code examples in Figure 2.1.

Original run	Value determinism	Output determinism	Path determinism	Non-determinism
T1.1 x = 1 T2.1 y = 0 T1.2 y = 1 T2.2 if (y == 1) T2.3 x = -1 T1.3 if(x <0) T1.4 ERROR	T2.1 y = 0 T1.1 x = 1 T1.2 y = 1 T2.2 if (y == 1) T2.3,x = -1 T1.3 if(x <0) T1.4,ERROR	T2.1 y = 6 T1.1 y = 1 T2.2 if (y == 1) T2.3,x = -1 T1.3 if(x <0) T1.4,ERROR	T1 Wx T2 Wy T1 Wy T2 Ry T2 Wx T3 Rx T1 ERROR	T1.1 x = 1 T1.2 y = 1 T2.1 y = 0 T2.2 if (y == 1) ..
ERROR	ERROR	ERROR	ERROR	-

Figure 2.1: Level of determinism, provided in record and replay systems

2.4.3.1 Value Determinism

One of the strongest types of determinism that could be provided by record and replay systems is value determinism, which guarantees that all the read and write operations will happen with the same values, including so-called "blind" writes, which are not followed by corresponding read operations. This approach guarantees that output-visible and output-invisible events will

appear in the secondary execution (Altekar and Stoica 2009). Its main drawback is extensive trace files. With a purpose to mitigate log sizes can be provided in relaxed form where either less events are being logged (Park, Zhou, Xiong, Yin, Kaushik, Lee, and Lu 2009) or the execution order of only write operations is being traced (Zhou, Xiao, and Zhang 2012), however, with a price of difficulties in replay phase.

2.4.3.2 Output Determinism

Output determinism states that the replay run outputs the same values as the original run (Altekar and Stoica 2009). It provides us with assurance of replay of output-visible events and faults such as crashes, core dumps, file corruptions, assertion violations. However, this type of determinism is considered weaker as it provides no guarantees regarding non-output properties of the original execution. Also, it does not guarantee that read and write operations happened in the same order as in original run. Depending on purposes of developer, replay can be relaxed by lowering of granularity of output events, for example on debug we might concentrate on failure events ignoring the exactness of value output and so on.

2.4.3.3 Path Determinism

This approach does not record values of operations but instead an execution path of each thread, aiming to repeat it on replay. Losing guarantees to replay the bug, it is still used to obtain in order to obtain symbolic execution of the program (Huang, Zhang, and Dolby 2013).

2.4.4 Other choice criteria

Range in Time: indicates whether the system requires the replay start point to be predetermined (*static time range*) or if it may be changed (*dynamic time range*). Systems with static time range usually rely on checkpoints, whereas systems with dynamic time range allow to replay an execution backwards.

Multiprocessor and Data Race support: in a single-processor system, it suffices to record the synchronization order and the thread scheduling decisions to deterministically replay any concurrent program (Ronsse and De Bosschere 1999). In these systems, parallelism is actually an abstraction: only one thread physically executes and accesses memory at

a given point in time. However, in a multi-processor environment (SMP and multicores), providing deterministic replay becomes much more challenging. As threads do actually execute simultaneously on different processors, logging the thread scheduling in each processor is no longer enough to know the exact access ordering to shared memory locations. A way to address this issue is to capture the full thread schedule, which is not easy to do to efficiently. This is also the reason why replaying data races is a major challenge for multi-processor deterministic replay systems.

Immediate Replay: this criterion indicates whether a system is able to replay an execution at the first attempt or not. For instance, it is common for search-based systems to relax immediate replay guarantees in order to reduce the recording overhead.

2.5 Example Systems

In this section, we provide an overview of some of the most relevant record and replay systems representing the design choices previously described. To facilitate the comparison among the systems, we present an overall view on all the systems in the Table 2.2.

2.5.1 Hardware-Only Systems

Many of the early record and replay systems have been designed considering the use of hardware support to detect data races, often piggybacking on the cache coherence protocol. For instance, Bacon and Goldstein (Bacon and Goldstein 1991) used a snooping bus to record all cache coherence traffic. In general, there are two main approaches to capture the information in hardware-only systems:

- *Data-driven:* Memory is logically divided into blocks (that consist of one or multiple memory words) and a timestamp is associated with each block. Every time a processor accesses a block, the timestamp is recorded and updated. This approach is also called point-to-point and FDR(Xu, Bodik, and Hill 2003) is the state of the art system, operating in this way.

- *Path-driven:* The log identifies sequences of data accesses that have been executed without interference from other threads. Each entry in the log, also called a *chunk*, stores the

address of all words that have been accessed in the sequence. Because of chunk division, this approach is also called chunk-based. For conciseness, the set of identifiers that are part of the chunk is stored as a *Bloom Filter* (Pokam, Pereira, Danne, Yang, King, and Torrellas 2009). Data accesses are added to the current chunk until a conflict exists. At that point, the chunk is recorded and a new chunk is initiated.

The major advantage of hardware-only systems is that they allow to achieve deterministic replay with little runtime overhead. Unfortunately, they are impractical in general due to the significant cost and amount of hardware modifications they require. Also, in many cases, one is not interested in logging the entire program but only parts of it. However, supporting selective logging would require even more changes to the hardware.

We briefly summarize some of the most relevant hardware systems in the next paragraphs:

2.5.1.1 FlightDataRecorder

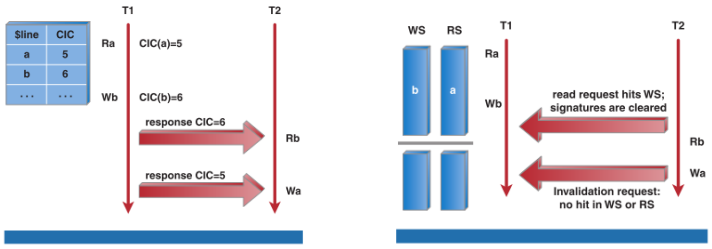
FlightDataRecorder (Xu, Bodik, and Hill 2003), whose name is inspired in the mechanisms used in avionics to trace flight data, is a system designed to continuously log data about a program's execution with the goal of supporting the replay of the last second of a (crashed) execution. Since the system traces a substantial amount of information, the state of the application is periodically check-pointed, such that the log only needs to preserve the data accesses after the last checkpoint. FDR leverages on previous work for implementing the check-pointing operation, namely on the SafetyNet mechanism (Sorin, Martin, Hill, and Wood 2002). After a checkpoint, FDR logs access the cache. For this purpose, the system maintains an *instruction counter* (IC) for each core and a *cache instruction count* (CIC) field for each cache line; the CIC stores the IC of last instruction that accessed the cache line (Pokam, Pereira, Danne, Yang, King, and Torrellas 2009). FDR implements a number of optimizations (Netzer 1993) to avoid storing information that can be inferred from previous entries. Thus, on the left side of Figure 2.2 only dependency from T1-Wb to T2-Rb is logged, as T1-Ra to T2-Wa is consequentially implied by T1-Wb to T2-Rb. With these optimizations, the authors claim that FDR can induce less than a 2% slowdown in the execution of a program. In terms of space overhead, FDR produces logs of 35 MB in an interval of 1,33s. For a longer period, such as 3 hours of replay, the size of the log would amount to 320 GB.

2.5.1.2 ReRun

ReRun (Hower and Hill 2008) is a path-driven approach, that identifies and logs sequences of data accesses that do not conflict with other threads. Such sequences, called *episodes*, are stored in a compressed format (using bloom filters as described before) along with a timestamp that identifies the order of the episode with regard to other episodes. Similar to point-to-point approach in FDR(Xu, Bodik, and Hill 2003), chunk-based approaches can also use transitive reduction to create smaller traces. As shown on the right side of Figure 2.2, the remote read T2-Rb conflicts with the write signature of T1 and causes T1 to end its chunk and to clear its signatures. Consequently, the request T2-Wa does not conflict and the dependency T1-Ra to T2-Wa is implied. In contrast to point-to-point approaches in which a timestamp is stored with each memory block, chunk-based approaches only need to store a timestamp per core to order chunks between threads. The bloom filter that encodes the data accesses during the episode, denoted *episode signature*, is used to detect conflicts among episodes. A conflict exists when one episode tries to access a data item that has been accessed by another episode. ReRun uses Lamport Clocks as timestamps for episodes, i.e., each core keeps a logical clock that is updated, according to Lamport's rules, to keep track of causal dependencies between reads and writes. During replay, episodes are executed according to the order of their timestamps. Although ReRun aimed at reducing the size of logs (when compared to FDR), it requires logical clocks to be piggybacked with every cache coherence operation, which is an additional source of overhead. In particular, the authors claim that ReRun's overhead is approximately 10%.

2.5.1.3 DeLorean

DeLorean (Montesinos, Ceze, and Torrellas 2008) is another path-driven hardware system for deterministic replay. As ReRun(Hower and Hill 2008), DeLorean also logs chunks using signatures, but in a different way: in DeLorean's multi-processor execution environment, cores are continuously executing chunks that are separated by register checkpoints. To detect a conflict, the system compares chunks' signatures. However, the updates of a chunk can only be seen after chunk commits. If a conflict is found, the chunk is squashed and re-executed. As with ReRun, replay determinism is achieved by replaying chunks according to their timestamp order.



(a) Point-to-point (b) Chunk-based

Figure 2.2: Hardware-only techniques (Source: Intel Corporation, 2009)

2.5.2 Hybrid Systems

Hybrid systems combine hardware and software support for record and replay. They aim at supporting a wider range of scenarios while reducing the costs associated with building dedicated hardware. In the following paragraph, we refer to one such system.

2.5.2.1 Capo

Capo (Montesinos, Hicks, King, and Torrellas 2009) uses hardware support to record the interleaving of threads and software support to trace other sources of non-determinism. It also provides support for logging only a subset of the entire program (for instance, the user code but not the operating system code). This is achieved by defining an abstraction named *Replay Sphere*, that encapsulates the set of threads whose operation need to be logged, and by defining the explicit and implicit transitions that allow the core to enter and leave the replay sphere. This system's concept can be better understood with Figure 2.3, which depict logical representation of system. Here, SW - software, HW - hardware, RSM - Replay Sphere Manager, RSCB - Replay Sphere Control Block, CPU - Central Processing Unit. Capo generates a combined log of 2.5 bits to 3.8 bits per kilo-instruction of the program, which results in a slowdown of the system execution on the order of 21% to 41%.

2.5.3 Software-Only Systems

There is a great variety of software-only techniques that provide record and replay. In the following, we briefly describe some of the most relevant software-based solutions proposed over

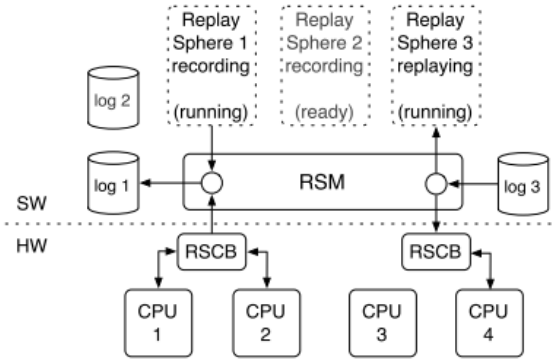


Figure 2.3: Logical representation of Capo

the years. For better understanding, we present an example on Figure 3.2 with the log structure of all the systems, listed in Table 2.1

2.5.3.1 Instant Replay

Instant Replay (LeBlanc and Mellor-Crummey 1987a) can be considered a pioneer software-only record and replay system. It follows an order-based approach: the system records the history of accesses to all the shared objects with respect to a particular thread. As this requires to uniquely identify shared objects dynamically, each object is extended with a custom version number that gets incremented after every write access during both record and replay. The computation model of Instant Replay is based on the CREW (Concurrent-Reader-Exclusive-Writer) protocol, which assigns two possible states to each shared object:

- *Concurrent Read*: implies a state where no processor is allowed to write, but all processors are allowed to read the value with no restrictions.

- *Exclusive Write*: implies a state where only one processor (called the *owner*) is allowed to read and write, and other processors are deprived of access.

This method makes the record phase quite costly and, due to the small granularity of shared memory accesses, tends to create enormous trace files. It has been reported that performance suffers an overhead up to 10x times of original execution time.

2.5.3.2 DeJavu

DeJavu (Choi and Srinivasan 1998) is another order-based system, designed at IBM ten years after Instant Replay, with the purpose of providing deterministic replay for Java programs. This system is based on capturing the total order of thread accesses, thus allowing to deterministically replay non-deterministic executions. To this end, DejaVu uses a global timestamping scheme, which exists in two categories:

- *Critical events*, which encompass synchronization points and shared memory accesses, relevant to the record and replay process.
- *Non-critical events*, which are those that only influence the thread where they get executed, so their scheduling is not utterly relevant.

This approach becomes less appealing when the number of threads and cores of a processor increases, as the overhead to capture the global order of all thread events becomes extremely high.

2.5.3.3 RecPlay

RecPlay (Ronsse and De Bosschere 1999) is a successor of DejaVu, but, unlike the latter, uses Lamport clocks instead of a global clock. RecPlay is based on the assumption that most programs do not have data races, and that the synchronization races are intentional and beneficial. As such, this solution traces thread accesses only to synchronization variables (such as monitoring of entries and exits). Since these Lamport timestamps are stored in the trace in a compressed from, the runtime slowdown is not very large. In the replay phase, the trace is consulted for every synchronization operation(Ronsse, Christiaens, and De Bosschere 2001). The drawback of this approach is the loss of determinism (as many shared memory accesses may not be synchronized) and the impossibility to replay problematic runs in the presence of data races(Ronsse, De Bosschere, Christiaens, de Kergommeaux, and Kranzlmüller 2003).

2.5.3.4 JaRec

JaRec (Georges, Christiaens, Ronsse, and De Bosschere 2004) is a portable record and replay system, designed especially for Java applications. This system operates at the bytecode level.

Its working principle is very similar to RecPlay (Ronsse and De Bosschere 1999), as it is also based on assumption that applications are data-race free. In particular, JaRec tracks only the lock acquisition, thus it is not able to reproduce buggy execution caused by data races as well. In other words, this system provides deterministic replay, but only until the first data race condition. This proviso makes JaRec and its predecessor RecPlay (Ronsse and De Bosschere 1999) unattractive in practice. In terms of performance degradation, JaRec's recording overhead lies between 10% to 125%, depending on the scale of the benchmark used.

2.5.3.5 iDNA

iDNA (Bhansali, Chen, De Jong, Edwards, Murray, Drinić, Mihočka, and Chau 2006) is an instruction level tracing framework, based on dynamic binary instrumentation. This system addresses non-determinism by tracking and restoring changes to registers and main memory. In order to do so, it maintains a copy of the user-level memory, which is implemented as a direct mapped cache. iDNA monitors the data values at every dynamic instance of instructions during the execution, and tracks the order of synchronization operations, which means that it does not support the reproduction of data races. An interesting feature of iDNA is the possibility to replay threads independently, as each of them maintains its own copy. But this can also be seen as a source of large log files. In fact, iDNA produces, on average, dozens of mega-bytes per second of trace sizes and incurs runtime overhead of 11x.

2.5.3.6 LEAP

LEAP (Huang, Liu, and Zhang 2010) is a deterministic replay solution for concurrent Java programs on multi-processors. LEAP is based on the observation that there is no need to track the global ordering of thread accesses to shared memory locations. Instead, it suffices for each shared variable to track only the thread interleaving that it sees (denoted *access vector*). Therefore, this solution produces a set of vectors containing the local access order with respect to the shared memory locations, rather than a single vector with the global order. As each shared variable has its own synchronization, this approach allows accesses to different variables to be recorded in parallel, thus imposing lower runtime overhead. Despite providing slightly weaker determinism guarantees, the authors prove that LEAP's technique does not affect the correctness of the replay. One of the limitations of LEAP is that it does not distinguish between

different instances of the same class, which creates false dependencies between different objects and a consequent increase in the recording overhead. Nevertheless, the experimental evaluation in (Huang, Liu, and Zhang 2010) showed that LEAP is up to 10x times faster than global order approaches (e.g. Instant Replay (LeBlanc and Mellor-Crummey 1987a) and DejaVu (Choi and Srinivasan 1998)) and 2x to 10x faster than JaRec (Georges, Christiaens, Ronsse, and De Bosschere 2004), albeit it still incurs huge overhead for applications with many shared accesses. As for space efficiency, trace sizes range from 51 to 37760 KB/sec.

2.5.3.7 ORDER

ORDER (Yang, Yang, Xu, Chen, and Zang 2011) was developed in order to record and reproduce non-deterministic events inside the Java virtual machine (JVM). This system follows an order-based approach and is based on two main observations: *i)* good locality at the object level with respect to thread accesses, and *ii)* frequent object movements due to garbage collection. ORDER literally records the order of threads accessing shared objects, eliminating unnecessary dependencies introduced by moving objects within memory during garbage collection. It also implements an offline log compression algorithm, used to filter out remaining unnecessary dependencies from thread-local and assigned-once objects, caused by imprecise static compiler analysis. To this end, ORDER extends the header of each object with following five meta-data fields:

- *Object Identifier* (OI) that works as an unique hash of the object.
- *Accessing thread identifier* (AT) and *access counter* (AC), which are used to maintain the current status of the object's access time-line. Every time-line recorded by ORDER can thus be interpreted as *"the object OI is accessed by thread AT for AC times"*.
- *Object-level lock* that is used to protect the whole object and synchronize the recording of the accesses to it.
- *Read-Write flag*, which records whether the current time-line record is read-only or read-write, for future log compression.

In both record and replay phases, ORDER relies on instrumentation actions added to the JVM. At *record* time, the system compares the AT in the object header with the identifier of the

current accessing thread (CTID). If the access belongs to the same thread, ORDER increments the corresponding AC. Otherwise, ORDER appends the tuple *(AT, AC)* to the log and proceeds with the execution. During *replay*, the process is similar: the system checks if AT == CTID and if the requesting thread is the expected one (according to the log), it decrements the AC allowing the thread to continue executing. Otherwise, the thread gets blocked until its turn. Performance evaluation results show that ORDER is 1.4x to 3.2x faster than LEAP.

2.5.3.8 CARE

CARE (Jiang, Gu, Xu, Ma, and Lu 2014) is a very recent application-level deterministic record and replay technique for Java concurrent programs. CARE employs an order-based approach that leverages thread locality of variable accesses in order to avoid recording all read-write linkages. Concretely, during the record phase, CARE assigns each thread with a software *cache*. This cache is updated every time the thread reads or writes on a shared variable, and is queried whenever the thread performs a read operation. Write operations are always synchronized and recorded into the trace file whenever a new thread writes on a given shared variable, whereas read operations are only logged in the presence of a *cache miss*. A cache miss occurs when the value read from the shared variable differs from the one previously buffered in the cache, meaning that another thread must have written on this variable before. At this point, CARE logs the exact read-write linkage by redoing the read action again with synchronization.

In the replay phase, CARE does not try to determine all non-recorded read-write linkages. Instead, it simulates the behavior of all caches and overrides read values from memory by values buffered in thread-local caches. This provides value-determinism guarantees at replay. Evaluation shows that CARE resolved all missed linkages for sequentially consistent replay, and exhibited 3.4x reduction on runtime overhead and 7x reduction on log size when compared to LEAP.

All the aforementioned systems employ an order-based approach. In the following, we describe some of most relevant search-based solutions.

2.5.3.9 PRES

PRES (Park, Zhou, Xiong, Yin, Kaushik, Lee, and Lu 2009) is a search-based record and replay system for C/C++ concurrent programs. Its underlying idea consists of minimizing the recording overhead during production runs, at the cost of an increase in the number of attempts to replay the bug during diagnosis. To this end, PRES records solely a partial trace of the original execution, denoted *sketch* (the authors have explored five different sketching techniques that represent a trade off between recording overhead and reproducibility). Later, in order to reconstruct the non-recorded information, PRES relies on an intelligent offline replayer to search the space of possible thread interleavings, choosing one that fits the sketch. As the search space includes all possible schedules, it grows exponentially with the number of data-race conditions. To address this issue, PRES leverages on feedback produced from each failed attempt to guide the subsequent one and on heuristics to explore the search space efficiently. Most of the time, this mechanism allows to successfully replay bugs in a few number of attempts (1-28 tries according to the experiments). In terms of performance slowdown, authors report an overhead from 28% (for network applications) to several hundred times (for CPU-bound applications).

2.5.3.10 ODR

ODR (Altekar and Stoica 2009), similarly to PRES, relaxes the need for generating a high-fidelity replay of the original execution by inferring offline an execution that provides the same outputs as the production run. In other words, ODR provides the so-called *output determinism*, which authors claim to be valuable for debugging due to: *i)* the reproduction of all output visible errors, such as core dumps and various crashes, *ii)* the assurance of memory-access values being consistent with the failure, and *iii)* not requiring values of data races to be identical to the original ones. However, this system provides no guarantees regarding the non-output properties of the execution, which makes replaying data races very challenging. To address this, ODR uses a technique, called *Deterministic Run Inference (DRI)*, to infer data-race outcomes, instead of recording them. Once inferred, the system substitutes these values in future replays, thus achieving output-determinism.

As an exhaustive search of the space data races is unfeasible for most programs, DRI employs two facilitating techniques: *i)* guiding the search, which allows to prune the search space by

leveraging the partial information recorded at runtime, and *ii)* relaxing the memory-consistency of all possible executions in the search space, which allows to find output-deterministic executions with less effort.

According to ODR's evaluation, while recording causes a modest slowdown of 1.6x, the inference time at the replay phase ranges from 300x to over 39000x the original application time (for ODR's low-recording overhead mode), with some searches not completing at all. Authors do not provide information about trace sizes.

2.5.3.11 STRIDE

STRIDE (Zhou, Xiao, and Zhang 2012) is a state-of-the-art search-based solution that targets value determinism by recording bounded shared memory access linkages instead of exact ones. Under the sequential consistency assumption, STRIDE infers a failure-inducing interleaving in polynomial time. Its recording scheme logs read operations without adding extra synchronization, which reduces the runtime overhead with respect to pure order-based approaches (Huang, Liu, and Zhang 2010). Write operations, on the other hand, are still captured in a synchronized fashion. To allow the reconstruction of the global thread schedule, STRIDE logs read operations with write fingerprints. More concretely, STRIDE associates a version number to all the recorded write operations and, for each read operation, STRIDE records a pair consisting of the value returned by the read operation and the latest version of write that read can possibly link to, i.e., the *bounded linkage*. This bounded linkage is later leveraged by STRIDE's search mechanism to quickly find the correct match between reads and writes. Since STRIDE is the base system of the work in this thesis, we describe its architecture in slightly more detail than the previous systems. The essential concepts of STRIDE are then the *execution log*, the *memory model* and the *legal schedule*. The *execution log* is divided into three parts:

- LW_x, which is a vector containing the local total order of the writes performed by different threads on the shared variable x.
- LA_i, which registers the order of lock/unlock operations on lock l. It allows to reproduce deadlocks.
- TR_i, which corresponds to the read log of thread i.

The *memory model*, in turn, defines the set of values committed by writes that are allowed to be returned by a read. STRIDE uses the most strict memory model for concurrent programs – *sequential consistency* (Lamport 1979). As defined by Lamport, under sequential consistency, the result of any concurrent execution is the same as if the operations on all the processors are executed in some sequential order and the operations of each individual thread appear in the program order. In this context, the *legal schedule* represents a total order of the read-write operations that conform with the memory behavior rules of STRIDE.

Using these concepts, STRIDE is able to, from an imprecise execution log, generate a feasible thread access ordering such that all read and write operations conform to the sequential consistency memory model. The main advantage of this approach is the low runtime overhead due to the avoidance of additional synchronization when logging reads on shared variables. Authors claim that STRIDE incurs on average of 2.5x smaller runtime slowdown and 3.88x smaller log than LEAP. On the other hand, this approach has the downside of losing some determinism guarantees.

2.5.3.12 CLAP

CLAP (Huang, Zhang, and Dolby 2013) is another state-of-the art search-based solution that targets path determinism, created for C/C++ multithreaded programs. Path determinism means this solution does not track values of operations but the path of each thread local execution or, in another words, control flow decisions. Such an approach allows to shrink the search space and gain significant time on the replay phase, as a possibility to take an incorrect execution path be reduced, as will the time of interference. The main idea is to create a constraint model that would represent possible interleavings, encode it as a set of formulae and solve it with help of SMT solver. To further reduce time the solution search time, CLAP generates various candidate schedules to the solver and checks if they satisfy the residuary part of constraints. The main drawback of the approach, used in CLAP is the amount of candidate schedules the system may generate before hitting feasibility. It also produces huge traces, difficult to process with increasing complexity of applications under instrumentation. Thus, depending on programs instrumented, runtime overhead would lay between 9.3% and 269%.

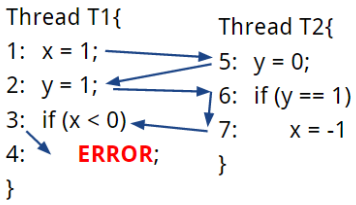


Figure 2.4: Example of thread interleavings

2.5.4 Example of Logs Produced by Different Systems

To help understand the differences among the logs produced by some of the systems above, we present, in Table 2.1, an illustration of their output for the example of Figure 3.2.

Instant Replay, DejaVu (global t), RecPlay (Lamport)	JaRec	iDNA	LEAP (access vectors)	ORDER (consider x and y as fields of one object)	CARE (cache per thread) [x-?,y-?]	PRES/ODR (min logs, more replay attempts)	STRIDE simplified (LW write log, TR read log)
1(T1) 5(T2) 2(T1) 6(T2) 7(T2) 3(T1) 4(T1)	T1 Lock/Unlock T2 Lock/Unlock T1 Lock/Unlock T2 Lock/Unlock T1 Lock	[x] set to 1 by T1 [y] set to 0 by T2 [y] set to 1 by T1 [x] set to -1 by T2	x:T1,T2,T1 y:T2,T1,T2	(1,T1,1) (1,T2,1) (1,T1,1) (1,T2,2) (1,T1,1)	T1:W[x-1,y-?] T2:W[x-?,y-0] T1:W[x-1,y-1] T2:cache miss! R[x-1,y-1]	SYNC: (T1,1,Lock) (T1,2,Unlock) (T2,3,Lock) (T2,4,Unlock) ... RW: (T1,1,Wx) (Tw,2,Wy) ...	LWx: T1(x1),T2(x2) LWy: T2(y1),T1(y2) TRi: T2(1,y2),T1(-1,x2) height

Table 2.1: Logs used in various systems

2.6 Summary

Table 2.2 summarizes the record and replay systems presented above. The systems are classified according to their implementation type and record and replay approach, along with some additional relevant criteria, namely the range in time, multi-processor support, ability to replay data-races, and ability to immediately replay without a state exploration stage. We also add a short reference to the main features of each system. It is important to highlight that doing a precise comparative analysis of these systems is almost impossible, since their evaluations have been performed on different benchmarks and using distinct criteria.

System	Implementation	R&R Approach	Main feature of System	Range in Time	Multi-processor support	Data Races support	Immediate Replay	Year
Flight Data Recorder	hardware	point-to -point	continuous logging	static (checkpoints)	✓	✓	✓	2003
ReRun	hardware	chunk	isolate episodes (non-conflicting access sequences)	static	✓	✓	✓	2008
DeLorean	hardware	chunk	logs chunks using signature	static (checkpoints)	✓	✓	✓	2008
Capo	hybrid	hybrid	Replay Sphere	static (checkpoints)	✓	✓	✓	2009
Instant Replay	software	order	full history of accesses	dynamic	✓		✓	1987
DeJavu	software	order	captures thread schedule	dynamic		✓	✓	1998
RecPlay	software	order	uses Lamport clock	dynamic	✓		✓	1999
JaRec	software	order	logs lock acquisitions	dynamic	✓		✓	2004
iDNA	software	order	tracks changes to registers	dynamic	✓		✓	2006
LEAP	software	order	uses access vectors for shared variables	dynamic	✓	✓	✓	2010
ORDER	software	order	mitigates influence of garbage collection	dynamic	✓	✓	✓	2011
CARE	software	order	uses cache scheme	dynamic	✓	✓	✓	2014
PRES	software	search	different recording schemes	dynamic	✓	✓		2009
ODR	software	search	Deterministic Run Inference	dynamic	✓	✓		2009
STRIDE	software	search	bounded linkage scheme	dynamic	✓	✓		2012
CLAP	software	search	SMT solver	dynamic	✓	✓		2013

Table 2.2: Summary of the presented systems.

Hardware solutions impose very small overheads (up to 2%), but require expensive hardware modifications. Hybrid solutions, in turn, rely on less hardware modifications and are still able to achieve a modest performance overhead (between 20% and 40%). However, they still suffer from compatibility issues, which makes this approach unattractive.

On the other hand, software-only solutions seem to be the most attractive due to the possibility of being easily deployed on commodity machines. Despite that, they require a clear trade-off between strong determinism guarantees and time and space overhead. In single-processor systems, it suffices to log the thread preemption points made by the scheduler to achieve deterministic replay, whereas, for multi-processor systems, racing memory accesses may affect the execution path, hence making deterministic replay much more challenging. As such, since DejaVu records solely scheduling invocations in the Java Virtual Machine, it only provides deterministic replay of multithreaded Java applications on uni-processors.

Among the solutions that strive to provide record and replay for multi-core systems, it is possible to see the variety of trade-offs when observing Table 2.2. For instance, Instant Replay aims to provide a global order of operations, adding extra synchronization to enforce replay. iDNA, in turn, records all the values read from or written to a memory cell as well as the thread synchronization order, while JaRec and RecPlay abolish the idea of global ordering and use Lamport clocks to maintain partial thread access orders. Unfortunately, they assume that programs are race-free and, therefore, are only able to ensure deterministic replay up until the first race. As for LEAP and Order, two state-of-the-art order-based techniques, they record the exact order of shared memory accesses, but incur a high performance slowdown. PRES and ODR achieve smaller recording overhead by means of searching heuristics that explore the non-recorded thread interleaving space, but at the cost of more replay attempts. Finally, STRIDE, a very recent search-based system, introduces a novel technique that relies on bounded linkages to relax the logging of read operations, albeit still requiring synchronization for tracking writes. Regarding time and space overhead, one can argue that, over the years, solutions have been improving these indicators. Nowadays, one considers the a recording technique as efficient if the overhead is generally less than 10% (Park, Zhou, Xiong, Yin, Kaushik, Lee, and Lu 2009).

After getting to know all of these systems, the following question arose: *Is it possible to further reduce the recording overhead by relaxing the logging of shared write operations in addition to read operations, while maintaining the ability to reconstruct the schedule in a reasonable*

amount of time? We believed so and the goal was set: experiment with a relaxation of the logging procedure, avoid synchronization when recording concurrent accesses to shared variables, compensate it with replay mechanisms that can look for different interleaving that are compliant with the information stored, in order to reproduce the bug in affordable time. The next section further explains this idea, by presenting the architecture envisioned for our system.

3 OREO

This chapter presents OREO, a novel technique for record of replay that aims at exploring a new trade-off between the overhead imposed during the record phase and the time it takes to reproduce the buggy execution during the replay phase.

3.1 Rationale

OREO is inspired by two complementary works that explore different points in the design space of record and replay systems, namely STRIDE (Zhou, Xiao, and Zhang 2012) and CLAP (Huang, Zhang, and Dolby 2013).

The main idea of STRIDE is to reduce (slightly) the recording overhead while still ensuring a fast replay. As many of the early record and replay systems STRIDE still aims at capturing in a shared log the interleavings that occur in a faulty run when accessing shared variables. However, they avoid the use of locks when recording the read accesses to shared variables. By using this optimization, STRIDE was able to, on average, reduce the runtime overhead by 2.5x and to produce 3.88x smaller logs, with respect to pure order-based solutions (e.g. LEAP (Huang, Liu, and Zhang 2010)). One problem of this optimization is that STRIDE no longer logs the exact shared read-write linkages. Still, STRIDE implements mechanisms to infer the correct linkages from the inexact information stored in the logs that can work on polynomial time.

On the other hand, CLAP produces logs that are local to each thread, avoiding any sources of additional contention among threads for recording purposes. The downside of this approach is that it becomes much harder to infer the thread interleaving that produces the bug. Therefore, during the replay phase CLAP has to resort to *Satisfiability Modulo Theories* (SMT) solvers to find the right thread interleaving, by encoding the information stored in the independent logs produced by each thread as constraints that the faulty execution needs to satisfy, however still imposing enormous traces and huge overhead in case of instrumentation of complicated

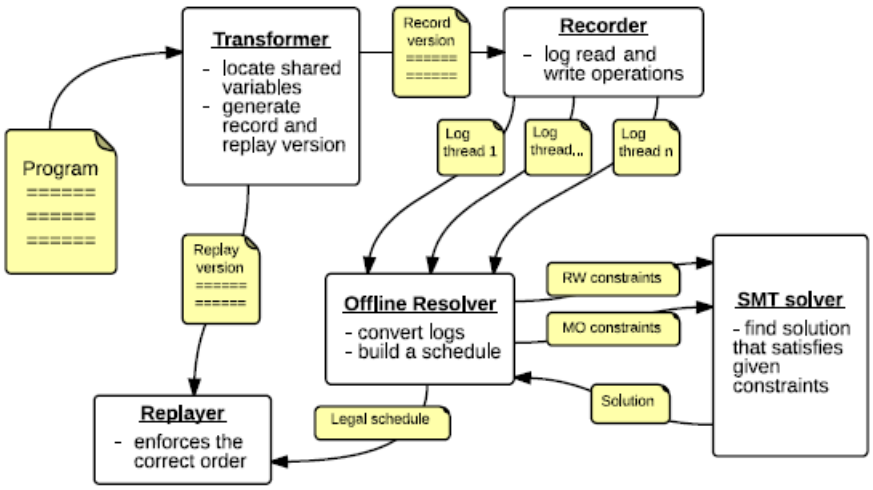


Figure 3.1: Components of proposed system

applications.

OREO aims at exploring a point in the design space that is between these two systems. In fact, OREO still uses shared information during the record phase, to capture a richer set of constraints. However, contrary to STRIDE, OREO avoids locks in both read and write operations, thus the logs produced by OREO are even less accurate than those captured by STRIDE. Thus, as in CLAP, we also resort to SMT solvers to find the faulty schedule. By being more relaxed that STRIDE we expect to be more efficient during the record phase. By capturing more information than CLAP, we expect to obtain a set of additional constraints that help in finding the faulty execution faster than those that can be obtained with CLAP logs.

3.2 OREO Architecture and Components

The OREO architecture follows the general model of software-only record and replay systems: program is preliminary instrumented, in order to allow to trace information at runtime. The resulting logs are then used to to replay the original run.

Figure 3.1 illustrates the architecture of our system, which in particular consists of four internal components and one external, namely the *transformer*, the *recorder*, the *offline resolver*, the *replayer* and external *SMT solver*. We describe each component as follows.

- *Transformer.* This component is responsible for the program instrumentation and analysis.

It takes the bytecode of an arbitrary Java program and produces two versions: the record version and the replay version. Our transformer role consist of various tasks: i) shared program elements (SPE), i.e. variables that can be concurrently accessed by different threads, ii) collect instructions of access to SPE, iii) collect information about threads. In addition to it, the transformer also instruments two versions of the code: a *record version* and a *replay version*, that will serve as input to the recorder and the replayer, respectively.

- *Recorder.* This component executes the record version of the program (previously instrumented by the *transformer*) and stores the relevant events into a trace file. Concretely, the *recorder* produces per thread logs, containing the write operations, the read operations, the lock acquisition order and information about thread creation. Unlike STRIDE, we do not introduce any kind of synchronization to log write operations, so writes with the same version are allowed.

- *Offline Resolver.* This component processes logs produced by the *recorder*, with the goal of producing a feasible execution ordering of events, i.e., the *legal schedule*. To this end, the offline resolver changed philosophy of its existence, to recall that the name remained unchanged. It first was created to build up a schedule by bounded linkage infer, time to time referring to conflicts resolution. After its role changed, in current system prototype the offline-resolver is used to convert logs from recorder to a set of SMT formulae, consisting of Read-Write constrains that represent local order or order per thread, and Memory Model constraints, which we build with the algorithm, similar to bounded linkage infer from STRIDE. All this we pass to the *SMT solver*

- *SMT Solver (external)* is a high-performance theorem prover that is accessed by a part of code from *offline-resolver* called SMT Connector. It returns a set of SMT formulae describing expressions that satisfy provided constraints, which is further converted to a *legal schedule*.

- *Replayer.* This component controls the scheduling of threads to enforce a deterministic replay using both the access vectors and the thread identity information. In such a way, if there is a bug in execution, it will get triggered in a deterministic way. Thus, developers working with multithreaded applications, gain possibility to repeat the events in their program (so-called cyclic debugging) and get the vision of what is happening among threads.

3.3 Trade-offs in OREO

3.3.1 Gains on the Record Phase

Conservative record and replay solutions introduce a significant amount of additional synchronization to ensure that accurate information regarding thread interleavings is stored in the log. Recent work proposed a number of techniques to reduce this amount of synchronization. Our work also explores this path: we introduce a relaxation of the logging procedure removing locking from write operations, used in STRIDE. Removing these locks allows us to gain an increase in execution speed. Although tracing the write order without synchronization may cause conflicts in the write versions mechanism (e.g., two writes may be recorded with the same version number), the experiments in (Zhou, Xiao, and Zhang 2012) showed that the read/write operation and the corresponding recording operation tend to occur contiguously for most programs. Our experiments in Section 4 further support this claim.

3.3.2 Implications for STRIDE's Linkage Inference Mechanism

Composing a feasible execution requires a *happens-before graph* (Musuvathi, Qadeer, Ball, Basler, Nainar, and Neamtiu 2008), a data structure that reveals the relative execution order of the threads in a concurrent execution. Nodes in this graph represent events within the thread and edges form a partial-order that determine the execution order. In such a way, the graph encodes the legal schedule constraints, which, in turn, needs the exact read-write linkages. To turn bounded linkages (Zhou, Xiao, and Zhang 2012) to exact ones, STRIDE uses a simple linear scan, presented below.

```
1. procedure LinkageInfer
2.   for all thread T_i, i in [1,K] do
3.     for all Rx_i(v) in T_i with bounded linkage bl do
4.       SearchForMatch(Rx_i(v), bl)
5.     end for
6.   end for
7. end procedure
8.
9. procedure SearchForMatch(Rx_i(v), bl)
10.   (k = bl; k > 0; k − −) do
```

```
11.     if WriteValueOf(LWx[k]) == v
12.       return LWx[k]▷ found exact linkage
13.     end if
14.    end for
15. end procedure
```

The core procedure is $SearchForMatch$: for each read operation (we suppose it reads variable x), STRIDE searches from the upper bound bl backward to index 1 in the the local write log (LWx) and stops at the first write that writes the value returned by this read. This algorithm is productive as Zhou et al. use additional synchronization in their system, so no writes with the same version occur. The time complexity of this algorithm is $O(Kn)$, where n is the total length of the execution log, and K is the number of threads. This is because, although the lower bound for the search in line 10 is 0, the jth read in thread Ti cannot match a write of an older version than the bounded linkage of the (j-1)th read. Therefore, the loop from the Line 3 to Line 5 in the worst case examines $O(n)$ operations. Since STRIDE only queries $O(n)$ times for the exact linkages, the average execution time of $SearchForMatch$ is $O(Kn/n) = O(k)$, which is extremely fast if only a small number of exact RW-linkages are to be recovered.

This algorithm seemed appealing to our solution as well, assuming that there would not be many conflicts. As such, we decided to extend this algorithm for OREO as presented below.

```
1. procedure LinkageInfer
2.   for all thread T_i, i in [1,K] do
3.     for all Rx_i(v) in T_i with bounded linkage bl do
4.       SearchForMatch(Rx_i(v), bl)
5.     end for
6.   end for
7. end procedure
8.
9. procedure SearchForMatch(Rx_i(v), bl)
10.  (k = bl; k > 0; k − −) do
11.    if SelectAllWrites(LWx[k] == v).size == 1
12.      return LWx[k]▷ found exact linkage
13.    end if
14.    else ResolveConflict(Rx_i(v), LWx[k])
15.    end else
```

```
16.   end for
17. end procedure
```

As the authors of STRIDE claimed this system is able to find the corresponding write for each read in time $O(1)$ for most cases, we expected that our algorithm to run with time complexity $O(c)$, in practice, where c is the number of write conflicts for a given version number.

The procedure $ResolveConflict(Rx_i(v), LWx[k])$ is called when in case of a write conflict. Here, we order writes in such a way that values Rx_i would conform with Wx_i, trying various combinations of these writes matching reads (starting from the first write operation from the bottom of log). However, it might be the case that reads from different threads are affected by the way one orders the conflicting writes. Therefore, in order to correctly resolve the conflicts, we have to consider simultaneously all reads that depend on the values of the conflicting writes.

In the worst case, we would have to test $c!$ write ordering variants for each version with conflicts to find one that is feasible. Since there might be several writes in such a condition in the whole log, we decided to cast the problem of finding the legal schedule as an SMT constraint solving problem, rather than doing a backwards scan of the logs, as in STRIDE. This way, we can easily use an SMT solver to obtain a legal schedule, by building a constraint formula that encodes the read and write operations, as well as their possible linkages (based on the values).

3.3.3 Using SMT for Replay

The idea to use SMT constraint solving to replay an execution is not new. For instance, Lee et al. have been experimenting with offline symbolic analysis for deterministic replay at the hardware level(Lee, Said, Narayanasamy, Yang, and Pereira 2009; Lee, Said, Narayanasamy, and Yang 2011) as the majority of modern processors operate with relaxed memory model in order to enable performance optimization. However, their solutions are not applicable in our case as they require expensive hardware modifications. Bringing their ideas to software-only systems would mean aiming at value determinism and therefore, feeding the SMT solver with non-trivial symbolic expressions, to find the legal schedule. To circumvent this drawback, CLAP (Huang, Zhang, and Dolby 2013) collects per-thread path profiles at runtime and uses them to guide a symbolic execution of the program, collecting symbolic information with respect to data-flow, control-flow and synchronization operations. This way, CLAP is able to build a constraint model

that requires the solver just to reason on orderings of operations, instead of having to find the actual values returned by the read operations that allow to satisfy the constraints.

However, since CLAP only records the execution path that each thread followed during the production, in the constraint model, it has to generate constraints that match each read on a shared variable to all the writes on that variable. As a result, CLAP exhibits poor scalability.

We believe that OREO moves a step forward towards the sweet spot between finding a legal schedule inference via constraint solving and recording overhead. Since we trace the exact value that a read operation saw at runtime, we are able to substantially reduce the amount of writes that it can be matched with. As a result, the constraint model built by OREO is simpler than that of CLAP, and the SMT solver is capable of finding a solution more easily.

3.4 Constraint Model

In this section we present the set of constraints that OREO passes to the SMT solver to obtain a legal schedule. Our idea is to use logged values to encode a constraint model representing the feasible thread interleavings that conform with the original execution. To facilitate constraint creation we use a dictionary data structure, where the key is a combination of a shared variable identifier with the value of a write operation. The value itself is a list of all the events that happened to the same field with the same value - a collection of writes with different versions. From above, we encode all the necessary constraints into a formula Φ constructed by a conjunction of four sub-formulae:

$$\Phi = \Phi mo \wedge \Phi rw \wedge \Phi wv \wedge \Phi so$$

where Φmo stands for the memory order constraints, Φrw denotes the read-write constraints over captured SPE accesses, Φwv are constraints which denote that among write operations the earliest has the lowest version number and Φso denotes synchronization order constraints that define the order of thread-related events. We further explain each constraint further.

3.4.1 Intra-thread Constraints

The following constraints are built using the information collected in a single per-thraed log, and aim to restrict the order of events within a thread.

3.4.1.1 Memory Order Constraints

Memory Order Constraints allow us to preserve partial order in which instructions are executed in each thread and represent data flow within the execution. This is important as it allows us to follow the original flow and therefore, reproduce the bug when needed. In order to produce these type of constraints we parse events from the log and sort them according to the value of eventId, per-thread counter of events. Thus, event with a smaller eventID value happened earlier in what corresponds to a sequential consistent memory model.

3.4.1.2 Write Versioning Constraints

A state where writes should be ordered according to the versions captured at runtime. We again sort events observed within thread execution and sort them by version value so the least version would denote the earliest event. This constraint reduces time of linkage inference when in the global execution different threads managed to perform write operations with the same version.

3.4.2 Inter-thread Constraints

These constraints aim to provide global order sequence of events from all the log instances hence adhere to a legal schedule of the original execution with the maximum alignment.

3.4.2.1 Read-Write Constraints

Read-write constraints represent a linkage between read and write operations across the execution, stating their execution order. Let us consider a read operation R_i on a shared variable where i is a particular version of captured access and W_i is one or more write operation on the same variable. OR_i will denote the order of read and OW_i the order of write operations respectively. To compose read-write constraints we scan the log from the end to the beginning, looking for a corresponding write access that would match in value and in version to each read operation. If a perfect match is found, we add a string denoting that the write operation happened before the read one:

$$OW_i < OR_i$$

If there are more write operations with the same version, we take a slightly different approach. Let us consider that W_i^{T1} and W_i^{T2} denote write operations on the same shared variable, which were occasionally captured with the same version (due to data races). There will be two mutually exclusive ways to sort them out:

1. R_i happened after W_i^{T1}. Then either W_i^{T2} had to happen before W_i^{T1} or this local ordering is not feasible.

2. R_i happened after W_i^{T2}. Then either W_i^{T1} had to happen before W_i^{T2} or this local ordering is not feasible.

Adding such branching to will enlarge the search space but also increase probability to actually repeat the original execution. Thus, for this particular situation we will add the following constraint:

$$((\mathrm{R}_i < W_i^{T1}) \wedge ((W_i^{T2} < W_i^{T1}) \vee (R_i < W_i^{T2}))) \wedge ((R_i < W_i^{T2}) \wedge ((W_i^{T1} < W_i^{T2}) \vee (R_i < W_i^{T1}))) \quad (3.1)$$

In such a way, n write operations with the same version will produce $n!$ mutually exclusive ways to match them to each read. Here SMT solver will help us to gain better increase in time of execution than custom algorithm with Java data structures.

3.4.2.2 Synchronization Order Constraints

Synchronization Order Constraints in OREO aim at enforcing the control flow of the program and therefore, ordering *partial order* events such as START, EXIT, JOIN, FORK while relaxing the model on *locking* and such events as SIGNAL, WAIT. This approach allows us to speed up building a model for Z3 while still being able to determine happens-before relation in general and avoid locking. In such a way, OREO imposes 4 types of partial order constraints:

1. $START < EXIT$ denotes the thread lifetime. The thread cannot finish before starting.

2. $FORK < START$ denotes how the child thread should be created.

3. $FORK < JOIN$ denotes the global rule of child thread creation.

4. $EXIT < JOIN$ denotes how the child thread should be finished.

Finally, the constraint formulae set is sent to a SMT solver.

3.4.3 Our model versus CLAP

Our approach is similar to one proposed in CLAP(Huang, Zhang, and Dolby 2013), however it is operating on a simpler memory model. The main differences are listed:

- Simplified read-write: OREO significantly reduces the size of the read-write constraints generated by CLAP, by tracing the value that each read saw at runtime. This way, OREO encodes only the matching of a read with the set of possible writes that wrote that same value.
- Eliminated locking constraints: OREO records the locking order at runtime, which allows to eliminate the locking constraints produced by CLAP altogether. Note that this set of constraints is cubic on the number of locking pairs, which results in a substantial reduction of the solving time. This fact was previously in shown in (Bravo, Machado, Romano, and Rodrigues 2013).

3.5 Lifecycle of the program on example

For better understanding of how OREO works, we present a simple flow basing on the example used in Chapter 2 to represent logs produced by each system.

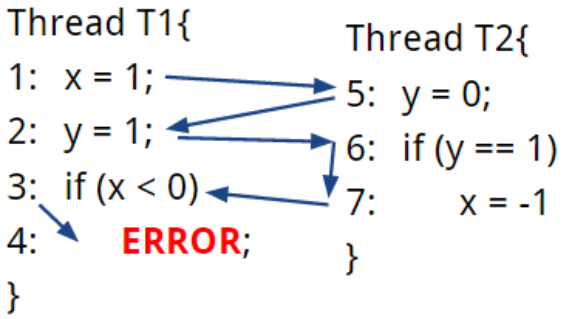


Figure 3.2: Example of thread interleavings

3.5.1 Transformer Output

The execution of the transformer module would produce following information.

```
--- Runtime version generated ---
--- Replay version generated ---
*** *** *** *** *** *** *** *** *** ***
*** SHARED VARIABLES [2]
   0 - "x.STATIC"
   1 - "y.STATIC"
*** SYNCHRONIZATION VARIABLES [1]
   0 - "java.util.concurrent.locks.ReentrantLock.OBJECT"
```

3.5.2 Recorder Output

Further, we would execute the recorder version parametrized by data about SPE, provided by transformer

```
[OREO-Recorder] T1 started running
[OREO-Recorder] T2 started running
[OK]
```

Therefore, recorder would produce logs for Thread 0 from which we fork Thread 1 and Thread 2 respectively. A sample set of event entries in logs for Thread 1 and Thread 2 can be seen on Figure 3.3 and 3.4 respectively. As you can see, event 2 in Thread 1 and event 3 in Thread 2 are conflicting as they are both write operations with the same version.

3.5.3 Offline Resolver Output

Further, basing on events logged by the recorder, we are building constraints for SMT. At first, we would build Memory Order Constraints. A partial log for Thread 1 and Thread 2 consists of 8 total operations, of which 2 are start, 4 are write and 2 are read. Let us hide the details of thread exit, forks and joins. Thus, each of these events will have a position from 0 to 7 in the final schedule. In the end, we create a partial order constraint for each thread MOthI, where I is the thread number.

```
(echo "MEMORY-ORDER CONSTRAINTS -----")
(declare-const O-START-T1_0 Int)
```

```
[
  {
    "eventId":"0",
    "threadId":"1",
    "eventType":"START"
  },
  {
    "eventId":"1",
    "threadId":"1",
    "value":"1",
    "eventType":"WRITE",
    "fieldId":"x",
    "version":"0"
  },
  {
    "eventId":"2",
    "threadId":"1",
    "value":"1",
    "eventType":"WRITE",
    "fieldId":"y",
    "version":"0"
  },
  {
    "eventId":"3",
    "threadId":"1",
    "value":"-1",
    "eventType":"READ",
    "fieldId":"x",
    "version":"1"
  }
  ...
]
```


Figure 3.3: Thread 1 recorder log


```
[
  {
    "eventId":"0",
    "threadId":"2",
    "eventType":"START"
  },
  {
    "eventId":"1",
    "threadId":"2",
    "value":"0",
    "eventType":"WRITE",
    "fieldId":"y",
    "version":"0"
  },
  {
    "eventId":"2",
    "threadId":"2",
    "value":"1",
    "eventType":"READ",
    "fieldId":"y",
    "version":"1"
  },
  {
    "eventId":"3",
    "threadId":"2",
    "value":"-1",
    "eventType":"WRITE",
    "fieldId":"x",
    "version":"0"
  }
  ...
]
```


Figure 3.4: Thread 2 recorder log

```
(assert (>= O-START-T1_0 0))
(assert (<= O-START-T1_0 7))
(declare-const OW-field_x-v0-T1_1@1 Int)
(assert (>= OW-field_x-v0-T1_1@1 0))
(assert (<= OW-field_x-v0-T1_1@1 7))
(declare-const OW-field_y-v0-T1_2@1 Int)
(assert (>= OW-field_y-v0-T1_2@1 0))
(assert (<= OW-field_y-v0-T1_2@1 7))
(declare-const OR-field_x-v1-T1_3@-1 Int)
(assert (>= OR-field_x-v1-T1_3@-1 0))
(assert (<= OR-field_x-v1-T1_3@-1 7))
```

```
(assert (! (< O-START-T1_0 OW-field_x-v0-T1_1@1 OW-field_y-v0-T1_2@1
    OR-field_x-v1-T1_3@-1 ):named MOth1 ))
(declare-const O-START-T2_0 Int)
(assert (>= O-START-T2_0 0))
(assert (<= O-START-T2_0 7))
(declare-const OW-field_y-v0-T2_1@0 Int)
(assert (>= OW-field_y-v0-T2_1@0 0))
(assert (<= OW-field_y-v0-T2_1@0 7))
(declare-const OR-field_y-v1-T2_2@1 Int)
(assert (>= OR-field_y-v1-T2_2@1 0))
(assert (<= OR-field_y-v1-T2_2@1 7))
(declare-const OW-field_x-v0-T2_3@-1 Int)
(assert (>= OW-field_x-v0-T2_3@-1 0))
(assert (<= OW-field_x-v0-T2_3@-1 7))
(assert (! (< O-START-T2_0 OW-field_y-v0-T2_1@0 OR-field_y-v1-T2_2@1
    OW-field_x-v0-T2_3@-1 ):named MOth2 ))
(assert (distinct O-START-T1_0 OW-field_x-v0-T1_1@1 OW-field_y-v0-T1_2@1
    OR-field_x-v1-T1_3@-1 O-START-T2_0 OW-field_y-v0-T2_1@0 OR-field_y-v1-T2_2@1
    OW-field_x-v0-T2_3@-1 ))
```

Read-Write constraints are responsible for conflict resolution and therefore in the following way (analog to STRIDE linkage infer represented in constraints):

```
(assert (< OW-field_x-v0-T2_3@-1 OR-field_x-v1-T1_3@-1))
(assert (< OW-field_y-v0-T1_2@1 OR-field_y-v1-T2_2@1))
```

Write versioning constraints are not created for this simple example but in general if we would have more write operations with a bigger version, we would create a per field constraint, sorting write operations in ascending order (smaller version happened first). Thread constraints are simplified for this example:

```
(assert (< O-START-T1_0 O-EXIT-T1_10))
(assert (< O-START-T2_0 O-EXIT-T2_10))
...
```

In the end we append commands that submit the model to z3 and further clean it for reuse, necessary for cycling debug :

```
(check-sat)
(get-model)
(reset)
```

We obtain the following solution, which will be further encoded for use in the replayer.

```
sat
(model
    (define-fun O-START-T1_0 () Int 0)
    (define-fun OW-field_x-v0-T1_1@1 () Int 1)
    (define-fun O-START-T2_0 () Int 2)
    (define-fun OW-field_y-v0-T2_1@0 () Int 3)
    (define-fun OW-field_y-v0-T1_2@1 () Int 4)
    (define-fun OR-field_y-v1-T2_2@1 () Int 5)
    (define-fun OW-field_x-v0-T2_3@-1 () Int 6)
    (define-fun OR-field_x-v1-T1_3@-1 () Int 7)
    ...
    (define-fun O-EXIT-T1_10 () Int ...)
```

We further convert it into log for the replayer.

3.5.4 Replayer

The Replayer consumes a log with the following format (field 5 was first accessed by Thread 1, then Thread 2):

```
Access vector for field x
[1, 2, 1]
Access vector for field y
[2, 1, 2]
```

3.6 Implementation Details

3.6.1 Transformer

The OREO transformer is developed on top of Soot(Vallée-Rai, Co, Gagnon, Hendren, Lam, and Sundaresan 1999), a static program analysis framework for optimizing Java bytecode, developed at McGill University in 1999. This framework supports three intermediate representations for representing Java bytecode: Baf, a streamlined representation of Java's stack-based bytecode; Jimple, a typed three-address intermediate representation suitable for optimization; and Grimp, an aggregated version of Jimple. Our transformer takes the bytecode of the Java program selected for analysis, and performs its instrumentation on Jimple, in particular:

- localizes shared program elements (SPEs), which we recognize in two types: shared variables (static and instance elements of class that can be accessed by different threads), and synchronization variables (locks and monitors).
- produces record and replay code versions by injection of Jimple probes into original code.
- visits each Jimple statement and performs tasks they describe.

Our transformer can instrument only Java programs. Depending on nature of SPE it inserts different probes:

- Operations on shared variables beforeLoad, afterLoad, beforeStore, afterStore;
- Operations on synchronized variables beforeMonitorEnterStatic, afterMonitorEnterStatic, recording calls to *signal()* and *await()* beforeConditionEnter, afterConditionEnter;
- Monitor acquisition operations for instance invocations beforeMonitorEnter, afterMonitorEnter, exitMonitor;
- Thread behavior and control flow events mainThreadStartRun, threadStartRun, threadExitRun, and so on.

Thus, the OREO transformer can be seen as a simplified version of the transformer, proposed by Huang et al.(Huang, Liu, and Zhang 2010). We inject less instructions, do not provide

annotation-based specifications for inserted end-points and therefore, do not support user specified end points on record. Transformer output provides us with the number of shared variables and the number of synchronization variables that are necessary to call record or replay driver.

3.6.2 Recorder

When the record version is running, we pass through all injected Jimple probes and collect information about accesses to SPE in internal data structures. To represent each operation, we implemented a java class Event that can exist in 8 following types: read, write, lock, unlock, join, fork, start, exit. For each thread we create a counter, that assigns a partial order position to operations within the thread. Unlike STRIDE, we do not introduce any kind of synchronization to log write operations, so writes with the same version number are allowed. A conflict can be detected when there is more than one write operation on the same shared variable with the same event version number - as we increase version counter on each access, we can see a bug introduced by data races. We also maintain statistics of conflict ratio per run. On every thread fork, the parent thread ID is added to the thread creation order list. In order to save logs, we insert a ShutDownHook into the JVM Runtime in the recorder as an end point. When the program execution is over, the ShutDownHook will be invoked to perform saving of captured traces from internal data structures to physical files. Logs are stored in the JSON format.

3.6.3 Offline-Resolver

The SMT Solver used in OREO is the Z3(De Moura and Bjørner 2008), a high-performance theorem prover developed at Microsoft Research that is being used in several projects since February 2007. It is targeted at solving problems that arise in software verification and software analysis. While supporting various theories, it has proven to be useful in such application areas as extended static checking, predicate abstraction, test case generation. In our system we interact with the Z3 through a component written in Java, the Z3Connector, which uses Z3's binary API inside, providing us with a simplified interface to submit formulas in a text format. The reason for using the Z3 in OREO is its ability to produce models as part of the output by assigning values to the constants in the input and generating partial function graphs for predicates and function symbols. This allows us literally return a legal schedule, which requires a small conversion to be executed on replay driver.

3.6.4 Replayer

The Replayer controls the scheduling of threads and operations performed by them. Replay is based on information from the access vectors, received from the model provided by Z3, and the thread creation order list. Before creating a new thread, if compares the ID of the parent thread to the ID at the head of the thread creation order list. If they match, the new thread is allowed to be created and the head of the list is removed. Before replaying any access, threads coordinate with each other by semaphores, counting amount of SPE already accessed. Process is quick as the threads accessing different SPEs can execute in parallel.

3.7 Summary

In this chapter we described our solution, named OREO, which stands for "Optimistic approach to REcord phase" of deterministic replay applied to multithreaded applications. We presented the system architecture and its components and a memory model, accompanied by a small example. We have also described some relevant implementation details.

4 Evaluation

In this chapter, we present an evaluation of our OREO prototype. In particular, we conducted experiments with the purpose of evaluating our system according to the following criteria:

- *Recording overhead*, in terms of performance slowdown imposed by the logging, with respect to the native execution.

- *Log size*, in terms of the amount of information stored in the trace files during the production run.

- *Amount of write conflicts*, in order to understand the impact of tracing write operations without synchronization and to assess the likelihood of finding write versioning conflicts that vary with the nature of the application (e.g. the percentage of write operations, the number of threads, and the shared accesses).

- *Inference Time*, to evaluate whether our system will be able find and produce a legal schedule that triggers the concurrency bug in a practical amount of time.

As test subjects, we first used a micro-benchmark, named *Bank*, that simulates (unsynchronized) transfers on bank accounts. As this micro-benchmark allows to easily tune the number of threads and the number of shared accesses, we were able to assess OREO's benefits and limitations with respect to STRIDE in the presence of different execution scenarios.

In addition, we also tested OREO against STRIDE with four other third-party benchmarks: two from the IBM ConTest suite (Farchi, Nir, and Ur 2003) and two from the *Java Grande*[1] suite.

[1] http://www.javagrande.org

4.1 Bank Micro-benchmark

In order to evaluate both the performance and the replay ability of OREO, we started by developing a micro-benchmark that allows to easily tune the number of threads and shared accesses used in the experiments. This micro-benchmark consists of an application that simulates transfers between bank accounts. Since the threads concurrently update the accounts with no synchronization, the final balance may not be correct.

An execution of the benchmark corresponds to performing a number num_ops of operations (i.e., transfers between accounts). Here, each thread executes num_ops/num_threads transfers, where num_threads indicates the number of threads executing. Once a thread finishes its transfers, it does a sanity check to verifies whether the sum of all individual accounts is equal to the expected total balance. In case of inconsistency, the thread raises an exception that prints a message indicating that a bug has occurred.

To assess how OREO compares against STRIDE in the presence of different complexity scenarios, we executed the bank micro-benchmark with 12 distinct configurations. These configurations were obtained by varying the number of operations of the program for the set of values $\{50, 100, 500, 1000\}$, as well as the number of threads for the set of values $\{2, 4, 8\}$. Finally, the number of accounts (i.e., of shared variables) was set to 10.

4.1.1 Recording Overhead

Figure 4.1 depicts the recording overhead, with respect to the native execution time, of both OREO and STRIDE for the bank micro-benchmark, when varying the number of threads and the number of operations. The plots show that OREO imposed less runtime slowdown than STRIDE for all cases. On average, STRIDE incurred 64% recording overhead, whereas OREO achieved an overhead of 45%, thus being 1.4x faster than STRIDE.

Another interesting observation that we can draw from Figure 4.1 is that the overhead does not increase linearly with the number of threads. This is because an increasing number of threads implies a smaller of operations executed by thread.

Finally, we expected the overhead reduction achieved by OREO to increase with the number of operations, as the negative effect of using locks would be more visible. We believe that this

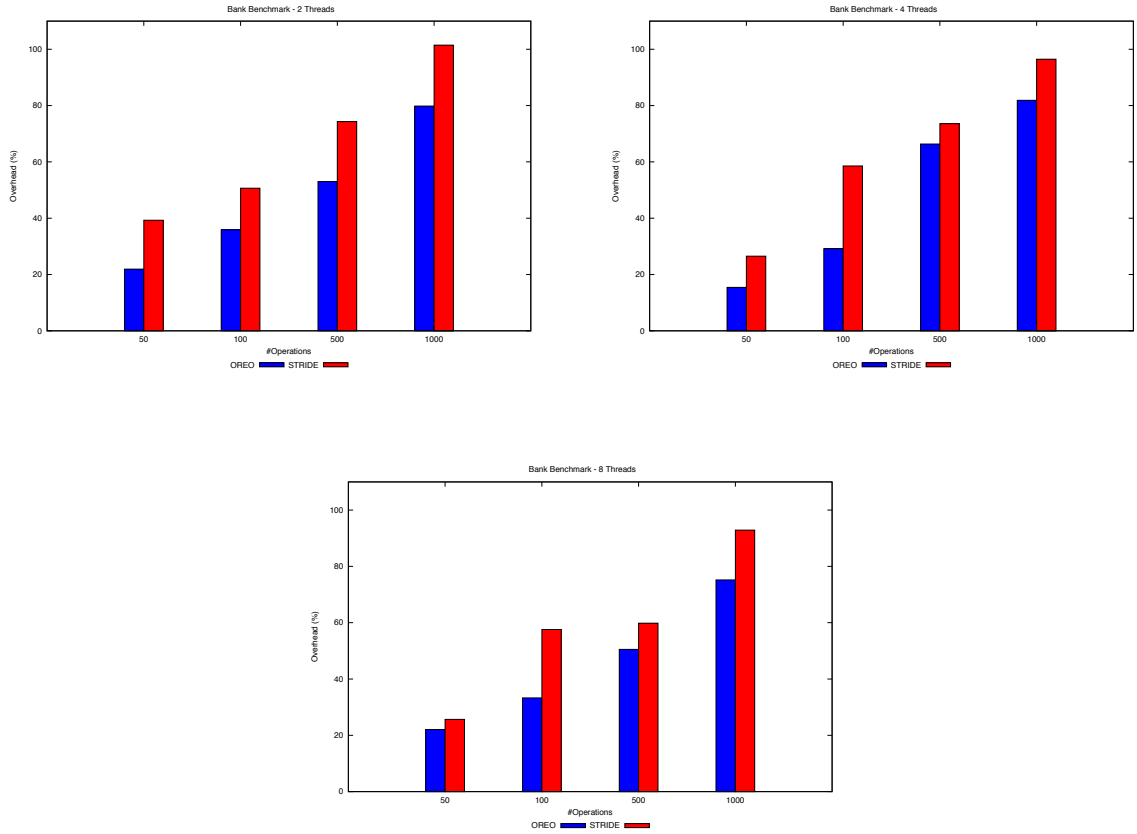


Figure 4.1: Recording overhead (%) for OREO and STRIDE for benchmark Bank, executed with 2, 4, and 8 threads. Results are averaged over 5 runs.

did not happen in this benchmark because the amount of writes did not comprise the majority of the operations of the program. As a result, the negative impact of STRIDE's additional synchronization ended to be diluted in the overall instrumentation cost, required to log the events.

4.1.2 Log Sizes

Since our implementations of OREO and STRIDE share the same data structures and event objects, the trace files generated by the two techniques are identical. As such, we solely report the log sizes for OREO. Table 4.1 reports these results.

As expected, Table 4.1 shows that the size of the logs increase with the number of operations performed during the execution. Also, despite being the same amount of operations, having more

	#Operations			
#Threads	**50**	**100**	**500**	**1000**
2	45KB	74KB	305KB	597KB
4	47KB	77KB	307KB	598KB
8	54KB	82KB	311KB	603KB

Table 4.1: Log sizes for OREO.

	#Operations			
#Threads	**50**	**100**	**500**	**1000**
2	5s	50s	3h56m	>8h
4	23s	4m22s	5h16m	>8h
8	59s	4m1s	7h1m	>8h

Table 4.2: Amount of time required to solve the constraint model with the read-write linkages and produce a legal schedule.

threads also results in slightly larger logs, because each thread has a dedicated array where it stores its events. Hence, there is a small fixed space cost resulting from these per-thread data structures.

4.1.3 Write Conflicts

Curiously, for all the experiments, we have never observed version conflicts when logging write operations without additional synchronization. This provides further support our initial claim that version conflicts are rare.

4.1.4 Inference Time

Table 4.2 reports the time OREO took to produce a legal schedule, corresponding to the solving time of the constraint model. It is possible to see that an increase in the number of operations has a significant impact on the solving, as the SMT solver required 10x more time to solve the model when doubling the amount of operations from 50 to 100. Moreover, for the cases where the benchmark was configured to perform 1000 operations, the solver took even more than 8h. This indicates that for long executions, it might be useful to perform a first stage when read-write linkages are resolved solely using write versions. Then, in case of version conflicts, one could resort to the SMT solver to produce the legal schedule.

As a final remark, we highlight the fact that OREO was able to reproduce the bugs in the

Program	#Threads	#Shared Accesses	Recording Overhead (%)		Log Size	#Write Conflicts	Solving Time
			OREO	STRIDE			
TwoStage	5	41	4.8	41.7	3.1KB	0	0.13s
Airline	11	94	13.7	43.5	7.5KB	0	1.93s
Crypt	3	57	1.3	10.6	4KB	0	0.23s
LUFact	2	4037	42.5	64.3	373KB	0	29m

Table 4.3: Results for third-party benchmarks.

different executions using the inferred schedules.

4.2 Third-Party Benchmarks

In order to further assess OREO's benefits and limitations, we also conducted experiments with four third-party benchmarks. We used the benchmark bugs *TwoStage* and *Airline* from the IBM ConTest benchmark suite (Farchi, Nir, and Ur 2003), and programs *Crypt* and *LUFact* from the Java Grande benchmark[2] (both with input size B).

Table 4.3 reports the results of our experiments. Columns 1-3 characterize the programs in terms of their number of threads and shared accesses. Columns 4 and 5 report the recording overhead for OREO and STRIDE, respectively. Column 6 shows the log size generated by OREO. Column 7 indicates the number of write conflicts observed during the execution and, finally, column 8 contains the time required to solve the constraint model.

4.2.1 Recording Overhead

Comparing the recording overhead of OREO to that of STRIDE for the third-party benchmarks, it is possible to see that, once again, OREO outperforms STRIDE for all test cases. The most prominent case is *TwoStage*, where OREO was 8.7x faster than STRIDE, reducing the recording overhead from 41.7% to 4.8%. This program has only 41 shared accesses and has some synchronization operations on the original code as well, which caused OREO's instrumentation to not be too harmful for the program's performance. On the other hand, since STRIDE injects extra locks to record write operations, it imposed a non-negligible slowdown at runtime.

[2]http://www.javagrande.org

4.2.2 Log Sizes

The log sizes produced by OREO (which are identical to those of STRIDE) ranged from 3.1KB for *TwoStage* to 373KB for *LUFact*. As expected, the logs were larger for programs with more shared accesses, such as *LUFact*.

We stored the traces as strings in JSON format, to ease user readability. Therefore, we believe that the space overhead could be reduced by using more efficient data structures, such as serializable objects, and compression techniques.

4.2.3 Write Conflicts

Similarly to the previous section, we did not observe any write conflicts on the third-party benchmarks. Curiously, this did not even occur in the *Airline* program, which had 11 threads running. We believe that this is due to the fact that, despite having many concurrent threads, *Airline* did not have many shared write operations to incur version conflicts.

4.2.4 Inference Time

Regarding inference time, we can see in Table 4.3 that OREO is able to find the read-write linkage in a very short amount of time for programs *TwoStage*, *Airline*, and *Crypt*. For *LUFact*, however, OREO took 29m to produce a legal schedule, due to the large number of shared accesses in this program.

To further compare OREO against prior work, we generated the CLAP's constraint model for the *LUFact* test case and tried to solve it[3]. The solver found a solution in 1h, which was twice the time it took for OREO.

4.3 Summary

The results from our experiments showed that OREO incurs, on average, 1.4x and 2.6x less recording overhead than STRIDE, respectively for the Bank micro-benchmark and the third-

[3]We experimented with *LUFact* alone, as this was the program for which the constraint model exhibit more complexity and the solver had required more time to find a solution.

party benchmarks. In terms of constraint solving, OREO was also 2x faster than CLAP to resolve the read-write linkages for the program *LUFact*.

After all, OREO was able to produce a legal schedule, capable of replaying the original execution, within 7 hours for most test cases, which we argue to be a practical amount of time to be used in debugging.

5 Conclusions

5.1 Conclusions

This dissertation has presented and evaluated OREO, a novel approach to record-and-replay. OREO aims at making the logging phase more efficient, using a relaxed logging procedure and offline algorithms to link operations and produce a replay schedule. OREO combines features of different state-of-the-art systems, such as STRIDE(Zhou, Xiao, and Zhang 2012) and CLAP(Huang, Zhang, and Dolby 2013) to get better results. We have implemented OREO and performed an experimental evaluation assess its advantages and limitations.

For future work we would like to explore adaptive mechanisms in the record phase, to switch between exact and relaxed write logging, thus allowing to automatically tune the recording scheme to an application's nature.

References

Altekar, G. and I. Stoica (2009). Odr: output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 193–206. ACM.

Bacon, D. F. and S. C. Goldstein (1991). *Hardware-assisted replay of multiprocessor programs*, Volume 26. ACM.

Bhansali, S., W.-K. Chen, S. De Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau (2006). Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd international conference on Virtual execution environments*, pp. 154–163. ACM.

Boothe, B. (2000). A fully capable bidirectional debugger. *ACM SIGSOFT Software Engineering Notes 25*(1), 36–37.

Brat, G., K. Havelund, S. Park, and W. Visser (2000). Java pathfinder-second generation of a java model checker. In *In Proceedings of the Workshop on Advances in Verification*. Citeseer.

Bravo, M., N. Machado, P. Romano, and L. Rodrigues (2013). Towards effective and efficient search-based deterministic replay. In *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems*, HotDep '13, New York, NY, USA, pp. 10:1–10:6. ACM.

Bressoud, T. C. and F. B. Schneider (1996). Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems (TOCS) 14*(1), 80–107.

Choi, J.-D. and H. Srinivasan (1998). Deterministic replay of java multithreaded applications. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pp. 48–59. ACM.

Chow, J., T. Garfinkel, and P. M. Chen (2008). Decoupling dynamic program analysis from execution in virtual environments. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pp. 1–14.

Coffman, E. G., M. Elphick, and A. Shoshani (1971). System deadlocks. *ACM Computing Surveys (CSUR) 3*(2), 67–78.

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

De Moura, L. and N. Bjørner (2008). Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340. Springer.

Dijkstra, E. W. (2002). *Cooperating sequential processes*. Springer.

Dunlap, G. W., S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen (2002). Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review 36*(SI), 211–224.

Farchi, E., Y. Nir, and S. Ur (2003). Concurrent bug patterns and how to test them. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS'03, pp. 286–293. IEEE Computer Society.

Feldman, S. I. and C. B. Brown (1988). Igor: A system for program debugging via reversible execution. In *ACM SIGPLAN Notices*, Volume 24, pp. 112–123. ACM.

Georges, A., M. Christiaens, M. Ronsse, and K. De Bosschere (2004). Jarec: a portable record/replay environment for multi-threaded java applications. *Software: practice and experience 34*(6), 523–547.

Hower, D. R. and M. D. Hill (2008). Rerun: Exploiting episodes for lightweight memory race recording. In *ACM SIGARCH Computer Architecture News*, Volume 36, pp. 265–276. IEEE Computer Society.

Huang, J., P. Liu, and C. Zhang (2010). Leap: lightweight deterministic multi-processor replay of concurrent java programs. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 207–216. ACM.

Huang, J., C. Zhang, and J. Dolby (2013). Clap: recording local executions to reproduce concurrency failures. In *ACM SIGPLAN Notices*, Volume 48, pp. 141–152. ACM.

Jiang, Y., T. Gu, C. Xu, X. Ma, and J. Lu (2014). Care: cache guided deterministic replay for concurrent java programs. In *ICSE*, pp. 457–467.

Joshi, A., S. T. King, G. W. Dunlap, and P. M. Chen (2005). Detecting past and present intrusions through vulnerability-specific predicates. In *ACM SIGOPS Operating Systems Review*, Volume 39, pp. 91–104. ACM.

King, S. T., G. W. Dunlap, and P. M. Chen (2003). Operating system support for virtual machines. In *USENIX Annual Technical Conference, General Track*, pp. 71–84.

Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on 100*(9), 690–691.

LeBlanc, T. J. and J. M. Mellor-Crummey (1987a). Debugging parallel programs with instant replay. *Computers, IEEE Transactions on 100*(4), 471–482.

LeBlanc, T. J. and J. M. Mellor-Crummey (1987b). Debugging parallel programs with instant replay. *Computers, IEEE Transactions on 100*(4), 471–482.

Lee, D., M. Said, S. Narayanasamy, and Z. Yang (2011). Offline symbolic analysis to infer total store order. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pp. 357–358. IEEE.

Lee, D., M. Said, S. Narayanasamy, Z. Yang, and C. Pereira (2009). Offline symbolic analysis for multi-processor execution replay. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 564–575. ACM.

Lu, S., S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou (2007). Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *ACM SIGOPS Operating Systems Review*, Volume 41, pp. 103–116. ACM.

Montesinos, P., L. Ceze, and J. Torrellas (2008). Delorean: Recording and deterministically replaying shared-memory multiprocessor execution ef? ciently. In *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*, pp. 289–300. IEEE.

Montesinos, P., M. Hicks, S. T. King, and J. Torrellas (2009). Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *ACM Sigplan Notices*, Volume 44, pp. 73–84. ACM.

Musuvathi, M., S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu (2008). Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, Volume 8, pp. 267–280.

Netzer, R. and B. P. Miller (1989). *Detecting data races in parallel program executions*. University of Wisconsin-Madison, Computer Sciences Department.

Netzer, R. H. (1993). *Optimal tracing and replay for debugging shared-memory parallel programs*, Volume 28. ACM.

Olszewski, M., J. Ansel, and S. Amarasinghe (2009a). Kendo: efficient deterministic multithreading in software. *ACM Sigplan Notices 44*(3), 97–108.

Olszewski, M., J. Ansel, and S. Amarasinghe (2009b). Kendo: efficient deterministic multithreading in software. *ACM Sigplan Notices 44*(3), 97–108.

Park, S., Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu (2009). Pres: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 177–192. ACM.

Pokam, G., C. Pereira, K. Danne, L. Yang, S. King, and J. Torrellas (2009). Hardware and software approaches for deterministic multi-processor replay of concurrent programs. *Intel Technology Journal 13*(4).

Ronsse, M., M. Christiaens, and K. De Bosschere (2001). Cyclic debugging using execution replay. In *Computational Science-ICCS 2001*, pp. 851–860. Springer.

Ronsse, M. and K. De Bosschere (1999). Recplay: a fully integrated practical record/replay system. *ACM Transactions on Computer Systems (TOCS) 17*(2), 133–152.

Ronsse, M., K. De Bosschere, M. Christiaens, J. C. de Kergommeaux, and D. Kranzlmüller (2003). Record/replay for nondeterministic program executions. *Communications of the ACM 46*(9), 62–67.

Sorin, D. J., M. M. Martin, M. D. Hill, and D. A. Wood (2002). Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pp. 123–134. IEEE.

Tanenbaum, A. S., T. Austin, and B. Chandavarkar (2013). *Structured computer organization*. Pearson.

Vallée-Rai, R., P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan (1999). Soot-a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, pp. 13. IBM Press.

Xu, M., R. Bodik, and M. D. Hill (2003). A" flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pp. 122–133. IEEE.

Xu, W., L. Huang, A. Fox, D. Patterson, and M. I. Jordan (2009). Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 117–132. ACM.

Yang, Z., M. Yang, L. Xu, H. Chen, and B. Zang (2011). Order: Object centric deterministic replay for java. In *USENIX Annual Technical Conference*.

Zamfir, C. and G. Candea (2010). Execution synthesis: a technique for automated software debugging. In *Proceedings of the 5th European conference on Computer systems*, pp. 321–334. ACM.

Zelkowitz, M. V. (1973). Reversible execution. *Communications of the ACM 16*(9), 566.

Zhou, J., X. Xiao, and C. Zhang (2012). Stride: Search-based deterministic replay in polynomial time via bounded linkage. In *Proceedings of the 2012 International Conference on Software Engineering*, pp. 892–902. IEEE Press.