Fault Reproduction for Multithreaded Applications

Angel Manuel Bravo Gestoso

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Examination Committee

Chairperson:	Prof. Doutor Pedro Sousa
Supervisor:	Prof. Doutor Luís Eduardo Teixeira Rodrigues
Members:	Prof. Doutor João Cachopo
	Prof. Doutor Johan Montelius

July 2013

Acknowledgements

I would like to thank to everyone I consider that has contributed to the development of this thesis. Although some of them have simply contributed by emotionally supporting me, I consider them as important as those that directly contributed.

First of all, I wish to thank to my advisor Luis Rodrigues. Without his guidance I could not have finished this thesis. He motivated me from the beginning, and he always had a wise comment to share.

Nuno Machado, colleague from the GSD group at INESC-ID. He not only guided and wisely advised me, he also cheered me up in many situations in which I doubted I was capable to finish the thesis. I hope to collaborate with him in the future and I would like to wish him the best for his career. Apart from Nuno, I would like to thank to João Matos and Nuno Diegues (also from the GSD group) for helping me out. To conclude with INESC-ID, I also want to thank to the SAT group at INESC-ID for being friendly and giving me some interesting hints that helped me to move forward.

My colleagues from EMDC. They made the last two years unforgettable and I would like to keep in touch with all of them. Among them, I would like to thank to Muhammet Orazov for the fruitful discussions we had. It helped me to clarify my ideas.

Special thank to Maria, who bore my continuous mood shifts. I hope you will keep bearing them. Thank you for following me around the world.

Finally, and most important, I would also like to thank to my family. They support me in any decision I take. Although they do not fully comprehend what is this thesis about, they suffered even more than me with every new obstacle I had to overcome.

Lisboa, July 2013

Angel Manuel Bravo Gestoso

European Master in Distributed Computing, EMDC

This thesis is part of the curricula of the European Master in Distributed Computing (EMDC), a joint program among Royal Institute of Technology, Sweden (KTH), Universitat Politecnica de Catalunya, Spain (UPC), and Instituto Superior Técnico, Portugal (IST) supported by the European Community via the Erasmus Mundus program.

My track in this program has been as follows:

First and second semester of studies: IST

Third semester of studies: KTH

Fourth semester of studies: IST

A mis abuelas.

Resumo

Escrever aplicações distribuídas e paralelas é uma tarefa bastante complicada. Devido a esta dificuldade, é comum erros de software aparecerem durante a fase de desenvolvimento das aplicações e, frequentemente, após a sua instalação. Além disso, as técnicas de depuração clássicas revelam-se insuficientes devido ao não-determinismo induzido por este tipo de aplicações.

As técnicas de gravação e reprodução foram criadas com o intuito de ajudar os programadores na tarefa de depuração. Estas técnicas são compostas por duas fases principais. A fase de gravação começa por capturar todos os eventos não-deterministas da execução. Posteriormente, durante a fase de reprodução, a execução original pode ser reproduzida, permitindo assim pesquisar a causa dos erros. Infelizmente, rastrear todos os eventos não-deterministas introduz um elevado custo adicional.

Esta dissertação apresenta o Symber, uma ferramenta de gravação e reprodução para aplicações Java concorrentes, que combina gravação parcial com um mecanismo de inferência baseado em execução simbólica. Deste modo, o Symber é capaz de reduzir o custo adicional introduzido ao rastrear apenas o caminho de execução local e a ordem na qual os trincos são adquiridos.

Os nossos resultados demonstram que o Symber induz uma sobrecarga competitiva em comparação com outras técnicas relacionadas, mantendo ainda a capacidade de reproduzir erros de concorrência de forma eficiente.

Abstract

Writing distributed and parallel applications is rather difficult. Because of this difficulty, many bugs appear during development and frequently, on deployed applications. Classical debugging techniques are not enough anymore because of the non-determinism induced by this kind of applications.

Record and replay techniques have been created in order to help developers. These techniques are composed by two main phases. The record phase captures all non-deterministic events of the execution. Then, during the replay phase, the original execution can be repeated in order to find the causes of the bugs. Unfortunately, tracing all non-deterministic events introduces a large overhead.

This thesis presents Symber, a record and replay tool for multithreaded Java applications that combines partial logging with an inference mechanism based on symbolic execution. Thus, Symber is able to reduce the overhead introduced by only logging the local execution path and the order in which the locks are acquired.

Our results demonstrate that Symber produces a competitive overhead in comparison to other techniques and still maintains the ability to efficiently replay concurrency bugs.

Palavras Chave
Keywords

Palavras Chave

Erros de Concorrência
Depuração
Reprodução Determinista
Mecanismos de Inferência
Execução Simbólica

Keywords

Concurrency Bugs
Debugging
Deterministic replay
Inference mechanisms
Symbolic execution

Índice

List of Figures

List of Tables

Acronyms

RDTSC ReaD Time Stamp Counter

RDPMC ReaD Performance-Monitoring Counter

I/0 Input/Output

DMA Direct Memory Access

MPI Message Passing Interface

PVM Parallel Virtual Machine

CREW Concurrent Read Exclusive Write

SPE Shared Program Elements

JVM Java Virtual Machine

RVM Research Virtual Machine

MPL Message Passing Libraries

CRC Cyclic Redundancy Check

TCP Transmission Control Protocol

UDP User Datagram Protocol

DRI Deterministic-Run Inference

SI-DRI Search Intense DRI

QI-DRI Query Intense DRI

HDFS HaDoop File System

GFS Google File System

DDRI Distributed Deterministic-Run Inference

RDV Replay Dependence Vector

PC Path Condition

JPF Java PathFinder

1 Introduction

This thesis addresses the problem of faithfully replaying buggy executions of multithreaded programs. The ability to replay a concurrency bug can be extremely relevant in the process of developing and debugging parallel applications. For this purpose, the thesis proposes a technique to reduce the amount of information that has to be logged in order to support the subsequent faithfully replay of the bug.

1.1 Motivation

Parallel and distributed applications are extremely difficult to design and implement. Furthermore, the code is very difficult to debug because some errors only appear when a specific thread interleavings occurs, and these interleavings may be hard to produce during test runs. In fact, there is evidence that a majority of this bugs are related to concurrency problems (Lu, Park, Seo, & Zhou 2008). As a result, it is not rare that concurrent applications are deployed with bugs, including some large and widely used applications such as MySQL, Apache, Mozilla and OpenOffice.

Classical debugging techniques, such as cyclic debugging, consist of repeating the faulty execution until the cause of the bug is found. Unfortunately, this mechanism cannot be easily applied for parallel and distributed systems, because of the non-deterministic nature of the executions may prevent the interleaving that causes the error to be reproduced in an useful number of re-executions. Non-determinism is introduced by several sources. For instance, in a parallel program, the order in which threads access a shared variable can differ across executions. The order in which nodes exchange messages in distributed systems also introduces non-determinism. Therefore, a bug may only appear in a specific execution. The bugs that do not appear deterministically in every execution, even if the same input is provided, are called “heisenbugs”. All these issues make the development and debugging of applications more complex. Furthermore, some bugs may only appear in some concrete deployments that may difficult to predict and/or difficult to reproduce in the testing environment; therefore, it is somehow unavoidable that some bugs get unnoticed before the software is released.

Record and replay techniques have been designed to mitigate these difficulties. The goal is to log enough information during the “normal” execution of the application such that, if an error occurs, the development team can later reproduce the buggy execution. In detail, logging runs with the application

and tries to capture as many non-deterministic events as possible. Since the amount of non-deterministic events can be extremely large, and the application may be required to execute for a long period of time before a bug is found, the main challenge of this phase is to be able to perform logging with small spatial and time overhead. On the other hand, the replay phase is aimed at re-executing the original execution by using the logs that have been created during the record phase.

The first record/replay techniques have been designed for reproducing the bug on the first attempt. Although this is a desirable goal, this resulting approach is quite expensive in terms of spatial and time overhead of the recoding phase, because all non-deterministic events have to be traced in runtime. In consequence, alternative approaches have been attempted recently. In particular, systems such as (Altekar & Stoica 2009) and (Huang, Zhang, & Dolby 2013), are based on the observation that the overhead imposed on the user-side execution can be more disruptive than a longer developer-side debugging execution. Thus, these techniques reduce the overhead of the recording phase by not tracing all non-deterministic events, at the cost of possibly longer replay phases. Since in order to replay the buggy execution one needs to deterministically reproduce all non-deterministic events, these new approaches introduce a new phase between the recording and the replaying phase. This new phase is in charge of inferring the information that has been omitted from the logs.

Our work extends these results, by introducing techniques that allow to further reduce the logging overhead while, at the same time, being able to provide sufficient information to the developers in order to help them to find the causes that produce the bug. We take CLAP (Huang, Zhang, & Dolby 2013) as our starting point, since we believe that among all approaches does the best balance between recording overhead, inference time and the information that is provided to the developers. Therefore, our system, named Symber, is an evolution of the CLAP system. Symber is aimed at substantially reducing and simplifying the inference phase by slightly increasing the recording overhead.

1.2 Contributions

This work addresses the problem of bug reproduction in multithreaded applications. More precisely, the thesis analyses, implements and evaluates techniques to efficiently record and replay buggy executions in the context of parallel applications. As a result, the thesis makes the following contributions:

- A novel record/replay technique for parallel programs that holds a lightweight recording phase and a substantially reduced inference phase in comparison to other approaches.
- A constraint model inspired by CLAP's constraint model that simplifies and speeds up the searching of the buggy execution during the inference phase.

1.3 Results

The results produced by this thesis can be enumerated as follows:

- A prototype that provides lightweight deterministic replay of multithreaded applications called Symber. The prototype achieves a light recording overhead by avoiding tracing all non-deterministic events and leverages on symbolic execution for inferring the unknown information.
- An experimental evaluation of the implemented prototype based on a developed micro- benchmark and third-party benchmarks.
- As a consequence of the experimental evaluation, the thesis has also produced a simplified version of CLAP for Java applications and a version of Ditto (Silva) due to the unavailability of the code.

1.4 Research History

The original goal of this work was to address the replay of *distributed* concurrent applications. However, after a study of the related work we did found that: 1) the original goal was too ambitious to be achieved in the short timeframe available to produce the thesis; 2) that improvements could be made to existing tools to replay (non-distributed) concurrent applications and that these improvements would be relevant when extending the work to address distributed applications. Therefore, we opted to concentrate on building and evaluating Symber. During my work, I benefited from the fruitful collaboration with several members of the GSD team and, in particular, from the collaboration with Nuno Machado that is addressing cooperative logging in his PhD work, a technique that can be used in combinations with the techniques proposed in this thesis.

1.5 Structure of the Document

The rest of this document is organized as follows. Chapter 2 provides an introduction to the different technical areas related to this work. Chapter 3 introduces Symber and Chapter 4 presents the results of the experimental evaluation study. Finally, Chapter 5 concludes this document by summarizing its main points and future work.

2 Related Work

2.1 Introduction

Considerable work has been done in the development of deterministic replay debugging techniques. The main problem that these techniques face when applied to parallel and distributed applications is that different executions can produce different outputs. As it has been mentioned before, non-determinism is one of the main challenges in the design, optimization, and debugging of parallel applications.

Non-deterministic behaviour can be classified into:

- *External non-determinism:* The application returns different results for different executions with the same input.
- *Internal non-determinism:* The application always returns the same result but the internal execution is different.

The related work section is organized as follows. Firstly, we state which are the main causes of non-determinism for any kind of application. Secondly, we introduce record and replay techniques and we identify the main challenges. Furthermore, we briefly summarize some interesting solutions for both parallel and distributed programs. Thirdly, we present some solutions that use inference mechanisms in order to reduce the recording overhead. Fourthly, we introduce the concept of incremental replay, that it is useful to tackle some of the challenges faced when recording long running applications. Finally, we summarize and compare the different approaches.

2.2 Causes of non-determinism

Non-determinism is not exclusive of parallel and distributed programs, it can also be found in classical sequential programs. According to (Ronsse, De Bosschere, & de Kergommeaux 2000), the causes of non-determinism in sequential, parallel and distributed programs are the following:

Causes of non-determinism in sequential programs

The input is the main cause of non-determinism for this kind of programs. According to (Pokam, Pereira, Danne, Yang, King, & Torrellas 2009), input sources can be classified as follows:

- *System calls:* For instance, the gettimeoftheday() function. Recording their effects must be enough since they are always produced at the same point of the execution.
- *Signals:* Asynchronous signals can be received at different time across different runs. Not only the effect but also the timing should be recorded in order to properly replay the execution.
- *Special Architectural instructions* such as RDTSC and RDPMC on x86 architecture.
- Other such as *location of program stack* and *location of dynamic libraries.*
- *I/O calls.*
- *Hardware interrupts.*
- *Direct Memory Access (DMA).*
- Some *processor-specific instructions.*

Some inputs are more difficult to replay than others since not only the content needs to be captured but also the exact moment in the execution has to be logged.

Causes of non-determinism in parallel programs

The main causes of non-determinism in parallel programs are the shared memory operations. The sequence of instructions that produces non-determinism are called race conditions. They are characterized by concurrent accesses to the same shared location, where at least one of those accesses modifies the value. There are two types of races:

- *Synchronization races:* those in which the programmer has used synchronization primitives, for instance, semaphores or monitors.
- *Data races:* those in which programmers did not use synchronization primitives. They are usually a consequence of a programmer errors.

Causes of non-determinism in distributed programs

For distributed programs, the order in which messages are exchanged between machines is the main source of non-determinism. For instance, in some message-passing libraries, such as MPI or PVM, promiscuous receive operations and test operations for non-blocking message operations make the execution non-deterministic. By promiscuous receive operations, we refer to those operations that can receive a message from multiple processes. On the other hand, test operations for non-blocking message operations are used to check whether an asynchronous message has been received or not. Depending on several factors, these operations can be executed more times in some executions than in others.

On the other hand, Java does not have either promiscuous receive operations or test operations for non-clocking message operations. Nevertheless, the behaviour of these operations can be replicated by using threads and timeouts.

It seems obvious that these classes are not exclusive. Therefore, the causes of non-determinism for sequential programs are also found in parallel and distributed programs.

2.3 Record and replay techniques

In order to be able to faithfully replay the execution, we need to record all non-deterministic events. Record/replay techniques are designed to take non-determinism into account. As the name implies, these techniques use two distinct phases: the record and the replay phase. The former tries to log every non-deterministic event that can lead to a non-deterministic execution. On the other hand, the replay phase tries to faithfully replay the original execution using the previously recorded log. The execution is now deterministic and can be run several times in order to discover the origin of the bug. Once the execution is recorded, the way to proceed is similar to that of classical debugging techniques such as cyclic debugging.

2.3.1 Challenges

The main challenges of a record-replay technique are the following:

- *Reducing the time overhead,* such that, tracing mode can be activated in production code. Leaving the tracing activated all the time can be the only way of assuring the correct replay of the faulty execution.
- *Minimizing the space overhead.* The more information we trace, the easier and more flexible the replay execution is. Therefore, if we have an application where the normal execution is rather short and we have enough resources for logging a great amount of data, it could be better logging as

much information as possible. On the other hand, if the application runs for a long period of time, we should try to be as much efficient as possible in terms of space overhead. Incremental replay techniques can be used in order to tackle this challenge. It is important to highlight that depending on the application, more or less information can be traced.

- *Ensuring faithful re-execution:* In relation with the space overhead challenge, we need to record enough information to faithfully replay the execution. The log, which is obtained during the record phase, should provide the sufficient information to eliminate every non-deterministic event.
- *Saving network bandwidth:* In distributed programs, saving network bandwidth is a very important factor. Since network bandwidth is usually the bottleneck of this kind of programs, the extra communication between nodes should not incur a great overhead in terms of bandwidth.
- *Ensuring privacy and security:* Some private content may be traced during the record phase; therefore, privacy concerns arise. The solution to this matter is based on making the bug report anonymous. Nevertheless, this solution is not trivial and it usually needs user support.

2.3.2 Taxonomies

Record-Replay systems can be classified in three broad categories, according to the degree of hardware support for logging: software based, hardware based, and an hybrid. The advantages of using a software based approach are that they are compatible with every already commercialized chip since it does not need any change on the hardware. Nevertheless, the introduced overhead can play a significant role in terms of performance. A software approach can be backed up with hardware support in order to diminish the time overhead.

On the other hand, record/replay systems can be also classified according to the kind of information they trace. From this perspective, they can be classified into two main groups: content-based and order-based. The former traces the content of every non-deterministic event. For instance, in the context of shared memory operations, content-based approach would trace the value that was read by every read operation. The main advantage of this approach is that every processor can be replayed in isolation since all the needed data has been already traced. For instance, when replaying the execution of a process involved in a distributed program, every network event can be emulated; therefore, the remaining processes do not need to be re-executed. Furthermore, the replay execution could be even faster than the original since there is not network latencies involved in the execution. Nowadays, this technique is only used for tracing I/O and some system calls such as gettimeofday() because it incurs a great space overhead.

Due to the overhead of content-based logging, most of the used approaches are based on tracing the order in which non-deterministic events occur. Therefore, when the content of the events is automatically

reproducible by the program itself, such as the messages of a distributed program, there is no need of tracing it. In consequence, logging the order of those messages should be enough for assuring a deterministic replay. However, order-based approaches increments the complexity of the replay phase.

Recently, another classification has appeared (Zamfir, Altekar, Candea, & Stoica 2011). It is based on the kind of determinism that the solutions provide. According to the authors, record and replay systems can also be classified as follows:

- *Perfect determinism:* This is the ideal determinism that a record and replay technique should provide. Providing perfect determinism means that the replay execution behaves in the same way as the original execution in every aspect of the program. Therefore, every non-deterministic event of the original execution is properly reproduced during the replay execution. In practice, tracing every non-deterministic event is infeasible; therefore, some relaxation in the determinism should be applied.

- *Value determinism:* This kind of determinism is the one provided by the majority of the solutions. It is based on assuring that the processes read and write the same values during both the original and the replay execution.

- *Output determinism:* It assures that the replay execution produces the same output than the original. When this approach is used, a different path that leads to the same output can be used during the replay execution. This kind of determinism allows to have a lighter record phase; therefore, both the spatial and time overhead of that phase can be substantially reduced.

- *Failure determinism:* This is the most relaxed form of replay determinism. Therefore it permits a very light trace phase but it does not provide a lot of information to the developer in order to find the cause of the bug. It simply assures that the replay produces the same failure that the original run.

- *Debug determinism:* It is based on the idea that record and replay techniques should not only exhibit a low overhead during record phase, but they also have to provide enough information to users in order to effectively debug the failure. The main idea is to heavily trace the most bug-prone parts of the application while relaxing determinism in the rest. The main challenge of this type is found in how to decide which those bug-prone parts are.

2.3.3 Record and replay techniques for parallel programs

In the following paragraphs, we overview some of the most popular record/replay solutions for parallel programs. Every solution is focused on capturing the non-determinism that race conditions introduce.

InstantReplay (LeBlanc & Mellor-Crummey 1987) is a software-only based record and replay technique for parallel and distributed programs. It is order-based; therefore, it is efficient in terms of time and spatial overhead. It is aimed at providing repeatable execution of tightly coupled systems. It also gives support for loosely coupled systems. It provides value determinism.

InstantReplay repeats the buggy execution by reproducing the relative order of the events. It does not depend on any particular interprocess communication method. It does not require synchronized clocks. It is decentralized; therefore, it does not introduce the bottleneck that having an overloaded central component might introduce.

In InstantReplay, every interaction among processes is treated as a shared object. Every write over a shared object changes a version number associated with the object. During the record phase, the version of the object is traced by the thread before writing. The number of reads between writes is also traced. Using this tracing mechanism is sufficient to assure a faithful replay. They claim that it has a minimum impact on program performance.

In order to efficiently implement the InstanReplay technique, several aspects have been taken into consideration:

- **Simulating the external environment:** As for cyclic debugging, it is assumed that the original execution and the subsequent replays happen in equivalent environment. In consequence, it is assumed that the programs to be replayed are not depending on the physical characteristics. Therefore, InstantReplay considers enough to provide the same amount of resources that the original execution for simulating the external environment.
- **Communication through shared objects:** InstantReplay identifies two kind of operations: write operations which modify the state of the shared object and read operations which do not modify the state. Share objects are augmented with two counters: version and readers. The former is incremented for each write operation over the object. The latter represents the number of read operations performed in the object for a specific version. This counters are the basic mechanism for replaying the execution.

  Thus, in the record phase, the process that is about to perform a write operation, it firstly logs the version of the object and the readers counter. Then, it performs the operation. Finally, the process increments the version counter and sets the readers counter to zero. On the other hand, a read operation increments the readers counter and logs the version counter.

  In order to serialize the modifications in the shared objects counters, we need to use an algorithm. InstantReplay does not force the users to use an specific algorithm. Concurrent-read-exclusive-write (CREW) is the algorithm that the authors used in their implementation.
- **Data structures for program monitoring:** Each process uses its own history tape in order to trace the version of the read and written objects. In the replay phase, each process compares

his history tape with the version of the shared object that it is about to write or read. If the versions match, the operation can be executed; otherwise, it has to wait until the version of the object is the matching one.

According to the authors, InstantReplay can also be used for message-passing programs. For instance, they propose to model the communication among processes as shared ports, mailboxs or a memory segments. Thus, each shared port would be a shared object in which InstantReplay can apply all the previously mechanisms in order to faithfully replay the buggy execution.

In their evaluation, the most optimize version of InstantReplay performs almost as fast as the unmonitored version of the application.

LEAP (Huang, Liu, & Zhang 2010) is motivated by the observation that conventional techniques for tracing parallel programs, such as Dejavu (Choi & Srinivasan 1998) and RecPlay (Ronsse & De Bosschere 1999), introduce a huge overhead (between 10x and 100x) during the execution. Furthermore, conventional techniques trace synchronized events; while the majority of heisenbugs appear in non-synchronized shared memory accesses. LEAP shows that it is possible to efficiently trace shared memory accesses; therefore, it is possible to faithfully replay the original execution without introducing a large runtime overhead. LEAP is a software-based approach that uses an order-based logging strategy and provides value determinism.

LEAP is based on the local-order of accesses to shared memory locations by concurrent threads. Instead of having every process tracing locally the accesses to the shared object (such as InstantReplay), each shared variable traces the order of thread accesses that it sees during the execution.

Multiple challenges appear:

- Static shared variable localization: It is statically done. If it is efficiently implemented, it saves runtime overhead.
- Consistent shared variable and thread identification across runs. How to match identities across runs.
- Non-unique global order.

In order to locate the shared variable accesses, LEAP uses a modified version of the ThreadLocalObjectAnalysis (Soot framework).

LEAP approach requires a mechanism to properly identify objects across runs (classical hash-based object identification cannot be used). For this purpose, LEAP uses a static field-based shared variable identification scheme. It divides the shared variable space into three categories: variables that serve as monitors, class variables, and thread escaped instance variables. All of them are called

shared program elements (SPE). LEAP authors claim that this identification is consistent across runs and does not introduce runtime overhead.

Unfortunately, this approach introduce two problems. On one hand, there is no distinction between several instances of the same type; therefore, all the instances share the same vector. The authors demonstrate that this feature does not introduce any inconsistency in the replay phase with regard to the determinism of the replay execution. However, the approach cannot uniquely identify scalar variables that are alias to shared array variables. In order to tackle this problem, an alias analysis is performed beforehand. Once all the aliases are identified, they share the same SPE. The resulting logging introduces constraints on the concurrency achieved by the instrumented version; therefore, a potential penalty in the performance.

LEAP also requires unique thread identification. Since the creation of threads across runs is non-deterministic, LEAP uses a more sophisticated approach than mapping id with names. Namely, LEAP introduces extra synchronization steps during the thread creation, in order to force the same order in the replay than in the original execution.

In order to assure a faithful replay, LEAP forces every thread to perform the same number of SPE accesses in the replay run than in the original run.

Implementation: LEAP is composed by three components:

- *Transformer*
- *Recorder*
- *Replayer*

The LEAP transformer takes the byte-code of a Java program and creates two new versions of the program: the record version and the replay version. In order to generate the record and replay versions, the LEAP transformer instruments three main aspects of the program: SPE accesses, thread initialization, and end points. In the record version, the purpose of the inserted instrumentation is to record the order in which those critical events occur. On the other hand, the instrumentation in the replay version is aimed at synchronizing the replay execution with the original one. I.e. the instrumentation reads the log and forces the execution to follow it.

The LEAP monitor is invoked on each critical event to record the id of the thread into the access vector of the SPE.

In order to save storage space, the representation of the access vectors is substituted by two new vectors: one which stores the thread id and a second which stores number of accesses of each thread. Therefore, storage space is saved when a SPE is accessed by the same thread several times consecutively. Furthermore, a list that represents the order of the thread creation is maintained.

Finally, the LEAP replayer is in charge of faithfully replaying the execution using the traces. Firstly, the driver loads the access vectors and the list of threads creation. Then, the execution can start. Every time a thread is created or an access to an SPE is performed, the log (list of threads and access vectors) are checked and followed.

The authors compare LEAP's implementation with other well-known implementations such as JaRec (Georges, Christiaens, Ronsse, & De Bosschere 2004) and Dejavu. Their results show that LEAP performs better in all the experiments they run. Furthermore, the size of the log is also smaller for LEAP than for the other approaches; however, for some applications, LEAP log size is still considerable large.

Ditto (Silva) is a software-based and order-based approach. It is built on top of an open source JVM called RVM. Therefore, Ditto can be applied to a Java program without changing the source code of the application. It gives support for parallel programs and assumes that the user is in charge of providing the same inputs during the replay execution. It provides value determinism.

The record phase is based on timestamps, which are associated with threads and shared variables. When a shared variable is modified, both the time-stamp of the thread and the time-stamp of shared variable are increased. Before increasing the timestamps, their values are traced. Moreover, the number of reads between writes is also traced when writing a shared variable. This mechanism gives support for recording data races (those which are intentionally done by the user and not synchronized). A similar approach is used for recording synchronization. On the other hand, the replay phase is based on forcing those critical events to occur when the timestamps match those registered in the trace.

The authors propose some optimization to the algorithm above, in order to decrease its spatial overhead. They argue that the order of some events is automatically implied by the program order and by previous constraints. For instance, if the sequential execution of the programs contains consecutive operations in the same thread, those in between operations do not need to be traced. Nevertheless, this approach incurs a greater time overhead since all this decision are made on-the-fly.

As with LEAP, the thread order creation also has to be traced. In consequence, a consistent thread identification between executions is needed.

Ditto has been compared with DejaVu, JaRec, and LEAP. They used some micro-benchmarks in order to test the approaches. The results show that Ditto performs better when considering performance indicators such as the number of threads, number of memory access, and number of shared objects. Furthermore, in general, the trace file is smaller. Nevertheless, for some applications, Ditto still presents a large overhead. For instance, for one of the benchmarks, the time overhead is 4729%.

2.3.4 Record and replay techniques for distributed programs

We now summarize the main ideas of several record and replay solutions for distributed programs. All these techniques try to eliminate the non-determinism introduced by the exchanged of messages among nodes. Some solutions try to minimize the number of logged messages; while other solutions try to get rid of promiscuous read operations by explicitly specifying the sender of the message.

In the following paragraphs, the solutions are summarized. Some of the approaches are not only solutions for distributed programs, but they are also capable to delete the non-determinism of parallel programs.

Netzer (Netzer & Miller 1992) is a record/replay technique for message-passing programs. It is an order-based technique that avoids tracing the order of every message. It traces a minimal but sufficient number of messages in order to assure a faithful replay execution. The authors claim that it is optimal in most of the cases and effective in the worst case. The decisions with regard to which messages must be traced (because they introduce non-determinism) is taken in runtime. Netzer's approach provides value determinism.

Only racing (concurrent) messages are logged. Two messages race if any of them can be received first. According to the authors, messages that are not racing do not introduce non-determinism; therefore, they do not need to be traced. In order to replay the execution for debugging, we need to enforce the recorded order of the racing messages.

Message tracing using on-the-fly race detection: The racing messages are detected in runtime. In order to identify racing messages, the authors claim (and prove it in their paper) that this can be achieved by keeping track of the "happen-before" relations. In this sense, if there is no "happen-before" relation between the previous received message and the sender of the new message, the first of those two messages has to be traced. Caused relations can be captured by using vector clocks, as they did in their implementation.

The tracing algorithm behaves optimal when each message is involved in only one race or in multiple races when races are transitive. Nevertheless, it is not optimal when non-transitive races appear. In this case, extra messages are traced.

Replaying the execution: Faithful replay is provided by re-executing every race as it was traced. They propose two approaches:

- Forcing the delivery of the messages but not the arrival of them. Thus, the messages can be sent without taking into consideration the trace. Then, the message will only be delivered by the receiver when the trace matches.
- Forcing both message arrival and message delivery. Thus, a message do not reach the receiver until it its turn.

The first approach can be easily implemented using a buffer. The nodes can send messages at any point, but it will not deliver them until the constraints imposed by the trace have been satisfied. Sent messages that cannot be delivered immediately are stored in the buffer. Unfortunately, the capacity of the buffer may, in some cases, be exhausted.

The second approach avoids the space limitations of the buffer, but it requires additional synchronization messages among processes. This incurs an extra overhead.

According to the authors, their tracing strategy is effective. It traces no more than 19% of the messages. However, it introduces a performance penalty in the range of 8% to 14%.

MPL* (Kergommeaux, Ronsse, & Bosschere 1999) is a software-based technique that combines an order-based approach and a content-based approach. The technique is based on the observation that, as it is mentioned at the beginning of this section, non-determinism in message passing libraries is introduced by promiscuous receive operations and nonblocking test-operations. A promiscuous receive operation is an operation that expects to receive a message from any other process (without specifying an specific process). On the other hand, nonblocking test-operations are those operations that check whether a message has been received. MPL reproduces these non-deterministic events using a record/replay technique. Their approach introduces small time overhead in both trace and replay phases and a small spatial overhead for traces. In order to get a reduction of the trace files, a novel technique based on the aggregation of events is used. It provides value determinism.

In order to reproduce the order in which promiscuous receive operations are performed in the original execution, it is sufficient tracing the actual sender of the message when it is received. Thus, it is possible to replace the promiscuous receive operation by the corresponding point-to-point receive operation on-the-fly when replaying.

On the other hand, it is also necessary to reproduce the nonblocking test-operations. Since these operations are identical for a specific message, it is sufficient counting how many of those are present in the execution and log the number. When replaying, the traced number is decremented every time MPL executes a test operation. Then, when the number is zero a wait operation is introduced; therefore, the replay execution does not execute more test operations for that message. In order to face unsuccessful series of test operations, a bigger number is logged. Therefore, zero is never reached and it would fail in the same way than the original execution. This aggregation of events dramatically decreases the size of the trace file. Furthermore, we do not need to use the library in some cases since we already know how many times it fails; therefore, an improvement in the performance is achieved.

Their evaluation shows that both the record and the replay phase incur an extremely low overhead. Unfortunately, this technique can be only applied to some specific message passing libraries.

MPreplay (Erik-Svensson, Kesler, Kumar, & Pokam 2009) is a hardware solution for supporting record and replay techniques in message-passing programs for message-passing architectures. It uses an order-based approach and provides value determinism.

The main reason for providing hardware support is the overhead incurred by software-only approaches. Using hardware support, the overhead is notably reduced.

MPreplay considers the use of three different hardware architectures:

- Hardware support for tracing the order of all racing messages. The authors based their algorithm on an approach introduced in (Netzer, Brennan, & Damodaran-Kamal 1996).
- Hardware support for a novel algorithm based on scalar timestamps.
- MPreplay-I which gives support to incremental replay techniques (see Section 2.5).

In order to support the tracing algorithm, each processor maintains a vector with an entry for every processor in the architecture. All messages exchanged in the system are tagged with the vector of timestamps and the sender id. When the destination process receives a message, it checks whether the message could have arrive earlier by comparing the piggybacked vector with his own. In order to provide this support, the hardware is augmented with a Race Log Buffer, which stores traced messages, a R-Logic, which analyses the racing messages on-the-fly, and a Receive Buffer that stores the appended data of the messages (vector and sender). Replay execution basically processes the messages following the log.

The second approach is based on scalar timestamps. In this case, instead of appending the whole vector of timestamps (with an entry for each process) on each message, only the sender and the receiver entries are appended. Therefore, this approach is more scalable since simply a constant number of integers is added to each message no matter the number of processes present in the system; while in the previous approach the size of the vector depends on the number of processes.

This strategy reduces the overhead. However, the number of logged messages is greater. The main reason of this is that the tracing algorithms iare based on tracing messages that do not hold the happen-before relation. Thus, due to reduction of information piggybacked in messages in the second algorithm, the vector of timestamps is less frequently updated. In consequence, less happen-before relations are found between processes and more messages are logged. The hardware is similar to the previous approach.

Finally, the authors have also designed a hardware solution for supporting incremental replay. It is based on checkpointing and message logging. The authors assume that the naive hardware gives support for checkpointing. However, they design a hardware solution for logging the needed messages based on (Zambonelli 1999).

The evaluation shows that vector timestamp logging technique produce a 10% of slowdown on average when compared to not logging version. Moreover, scalar timestamp logging technique performs even better. It only produces a 1% of slowdown on average. Nevertheless, as expected, less messages are logged when using the vector timestamp logging technique.

Non-intrusive (Gioachin, Zheng, & Kalé 2010) is as software-based record/replay technique for debugging parallel programs. The approach is rather different to the previous approaches. It provides value determinism.

The authors present the novel idea of using a 3-step procedure for recording and replaying the execution. This 3-step procedure allows them to only trace in detail the processes involved in the bug. Therefore, they can reduce the spatial overhead. Furthermore, this algorithm permits to replay the execution in a controlled environment.

Also, the authors claim that previous approaches introduce a large overhead that can hide some concurrency bugs. Using their approach, this does not happen since the first tracing phase is rather light.

3-step procedure: The procedure is based on two algorithms. The first algorithm is in charge of logging enough information to deterministically replay the execution. In this case, it simply logs the order in which messages are processed by each process. In some cases, tracing the order might not be enough. In order to tackle this problem, the authors propose an optimization based on content-based approaches. However, in order to reduce the huge overhead introduced by content-based approaches, they propose to trace a checksum of the content (Using exclusive-OR or "Cyclic Redundancy Check" (CRC32) operations).

The second algorithm traces detailed information about a set of processes. This set has to be chosen by the user after the execution of the first algorithm. In the second algorithm, the content of the messages is traced. Since only a subset of the processors, few gigabytes is the amount of data expected to be traced.

The procedure includes three steps. User executes the application on a large target machine. It uses the first algorithm to be able to deterministically replay the execution. This first step is executed until a bug is found.

Once a bug appears, we can execute the second step. Using the log of the first step, the application is re-executed for tracing in detail a sub-set of processors. The users have to choose the processors. Faulty processors are always good candidates.

After the detailed tracing, we can re-execute the processors, chosen during the second step, in a single-core machine. We can use conventional debugging techniques for this step since the content of all messages is traced. In case we cannot reproduce the bug in the last step, we can go back to the second step and pick different processors to be traced in detail.

The authors claim that step two could be executed using fewer physical resources by virtualizing the environment. In any case, it would introduce some problems such as the identification of processors (which is needed in order to comprehend the log of the first step) and incur some execution overhead.

Furthermore, the authors mention that in long executions, extra computation is needed. They propose two approaches: checkpointing and periodically flushing logs to disk.

Their experiments are based on measuring the overhead of the first step of the procedure. Results show that record phase incur a very low overhead when the size of the messages that processor exchange is small. However, when using CRC checksum techniques and increasing the size of the messages, it produces 2x slowdown. They claim that real applications tend to have a small size of messages; therefore, they argue that it is not a problem.

DJVM (Konuru, Srinivasan, & Choi 2000): Although Java does not have promiscuous receive operations, non-determinism still exists in Java programs due to the use of threads. For instance, a distributed Java program can maintain connections to several other nodes using sockets and a separate thread per blocking receive operation. Therefore, when a message is received, the thread in charge of that channel is activated. In this way, depending on network latencies, in a different execution of the program, the messages could be received in different order.

DJVM proposes a software-based deterministic replay technique for distributed Java applications. It extends DejaVu, which is a record/replay solution for parallel Java programs, by adding support for distributed programs. Dejavu provides value determinism.

DJVM is based on capturing the logical thread schedule information. The logical thread schedule is defined as a sequence of intervals of critical events. The authors refer to critical events to those whose execution order might alter the execution. For instance, shared variable accesses and synchronization events are classified as critical events. In consequence, an interval of critical events is a sequence of consecutively executed events (critical and non-critical) on a particular thread.

The algorithm to trace the logical thread schedule information uses two counters: a global and a local counter. The former is shared by all the threads of the same JVM; while the local one is accessible by an specific thread. When executing a critical event, the global counter is incremented; therefore, every critical event is uniquely identified. This identification permits DJVM to efficiently characterize the intervals by the first and the last critical event contained in the interval. That is the only information that is traced; in consequence, the log is efficiently created. On the other hand, the local counter also ticks with every critical event. DJVM uses the difference between the global and the local counter to distinguish the intervals in runtime (Choi & Srinivasan 1998).

Since the global counter is shared among threads, DJVM has to synchronize accesses. In order to efficiently do it, JVM has been augmented with one atomic operation that updates the global

counter and executes the event. Thus, the maintenance of the global counter is transparent from the point of view of the application. However, the atomic operation cannot be used for synchronization events since, according to the authors, it can produce deadlocks (Choi & Srinivasan 1998). Thus, DJVM is forced to execute synchronization events outside of the atomic operation.

During the replay phase, a thread waits until the global counter is equal to the first critical event of its next interval. Once the global counter matches, the thread consecutively executes all the events until the last critical event of the current interval is reached. In addition, each critical event increments the global counter.

DJVM can be applied in three different scenarios:

- *Closed world case:* Where all involved JVMs are modified.
- *Open world case:* Where only one of the involved JVMs is using DJVM.
- *Mixed world case:* Where some JVM are using DJVM and others not.

Furthermore, it is important to highlight that the authors identify and give support to three kind of sockets: point-to-point stream (TCP sockets), point-to-point datagram (UDP socket) and point-to-multiple-points datagram (which is treated as an special case of UDP socket).

Reported experimental results show that the record overhead is pretty low when the number of threads is not greater than 8 (between 0.7% and 9.3%). Nevertheless, as the number of threads is increased, the overhead also considerably increases (69.57% with 32 threads). On the other hand, the log size remains acceptable.

liblog (Geels, Altekar, Shenker, & Stoica 2006) is a software-based replay debugging solution for distributed C/C++ applications. The authors claim that liblog is the first replay tool able to support long-running applications and deterministic replay of an arbitrary subset of participant nodes. Furthermore, it works in a mixed environment where only some nodes log information. It provides value determinism.

The main ideas of their solution are: first, liblog is a library-based tool on top of libc (standard library of the C programming language). Second, it logs the contents of every incoming message so that the receiver of the message can replay the execution without the sender process. Furthermore, in order to assure a deterministic replay, the order of the messages is also traced. For this purpose, each message piggybacks an 8-byte Lamport clock. Third, in order to provide a central replay, a central console has to download logs and checkpoints of every process that it wants to replay. Having a central replay introduces obvious drawbacks, such as network bandwidth and migratable checkpoint system; nevertheless, they claim that it makes possible to operate on distributed state.

However, their solution has several limitations. First, the size of the logs can be huge due to the amount of information liblog traces. The authors state that liblog cannot be used for high-

throughput applications due to this limitation. Second, liblog simply addresses POSIX application and it is only supported by operating systems that support run-time library interposition. Third, some network overhead is introduced due to the information that is piggybacked in every message. Fourth, since the application can be deployed in a mixed environment, a perfect consistency cannot be assure because some message could not be traced. Finally, the authors state that a library-based tool is incomplete since cannot deal with every possible source of non-determinism.

Their experiments show that the CPU overhead imposed by liblog is sufficiently low and it does not heavily affect the network performance. Regarding spatial overhead, they admit that it could consume considerable disk space.

CURRF (Wang, Han, Zhou, & Fang 2009) is software-based approach that is based on tracing the content of the logged events. CURRF introduces the concept of code-based replay. The main idea behind this code-based replay is that some non-deterministic events do not need to be traced. The authors claim that it is impossible to automatically infer whether an event has to be traced or not. Therefore, CURRF delegates this responsibility to the user. In consequence, users have to decide which events should be traced by annotating the source code of the application.

CURRF provides value determinism. However, the accuracy and the completeness of the traced logs, and in consequence of the replay phase, depends on the annotations that the developer does. It is important to highlight that CURRF does not directly support mutithreaded programs.

CURRF approach has obvious limitations. For instance, the user that annotates the application has to be aware of how the logger works and has a deep knowledge of the application to be debugged. Even if the user satisfies these two requirements, this does not assure that the user picks all the critical events to be recorded. Thus, if the developer is not able to identify all operations that modify the execution behaviour, CURRF will no be able to re-execute the buggy execution.

The authors tested their system using two applications: a two-phase commit algorithm and wget. The experiments show that CURRF is easy to use, useful and incurs a low overhead.

Recon (Lee, Sumner, Zhang, & Eugster 2011) is a software-based approach. It leverages the record phase of Jockey (Saito 2005). The authors of Jockey claim that it commonly introduces an overhead close to zero. Nevertheless, they found some cases in which the overhead was close to 30%. According to the author of Recon, Jockey is a lightweight logger that accomplishes their requirements.

Once Jockey has logged the execution, Recon is able to replay it by following the logs. The main contribution of Recon is that, depending on the user needs, a more heavyweight instruction-level instrumentation can be done. Thus, for each re-execution of the buggy execution, the users can express using a SQL-like interface which extra information they would like to log. In order to

achieve this, Recon dynamically instruments the replay version of the target application based on the query that the users provide.

This approach permits the users to incrementally acquire more information about the execution; therefore, it makes easier to discover the source of the bug.

Their experiments show that the spatial overhead introduced by Jockey in non-long running applications (maximum execution they test took less than 10 minutes) is lower than 18MB. Moreover, the time overhead is never above 8% of the original execution. They also test their replay phase. The results show that the heaviest instrumentation could perform even 25 times slower than the replay version that Jockey's logs suggest.

2.4 Inference mechanisms

This section discusses how inference mechanisms can be used to improve record/replay techniques.

An inference mechanism, in the context of record/replay systems, refers to any technique that based on a constrained amount of information (w.r.t the original execution) is able to infer extra information that is needed by the replayer. Many of the inference mechanisms used by record/replay techniques are based on symbolic execution.

In consequence, the main goal of the inference phase is to reduce the overhead that the record phase introduces to the original application. The idea is to trace less information than needed for faithfully replaying the execution during the record phase. Therefore, it reduces both the runtime overhead and the size of the logs; however, it incurs two main drawbacks. Firstly, it makes the replay phase more complex; therefore, it introduces a significant overhead. For some applications such as large distributed systems, this overhead may be infeasible. Secondly, the execution path of the original and the replay execution have a high probability of being different. This entails that the developer will have less useful information for finding the cause of the bug. It might even happen that the replay execution shows a different bug that the original.

Despite the drawbacks, some kind of relaxation in the determinism provided by the solutions seems to be needed in order to tackle the challenges. The key design point is placed on deciding the amount of data that have to be traced, in contrast to the amount of inference computation we need.

In the following paragraphs, we summarize some of the record and replay solutions that use inference mechanisms.

ODR (Altekar & Stoica 2009) is a software-based approach. It uses an hybrid of content-based and order-based approach. In addition, it benefits from inference mechanism to reduce the time

and spatial overhead of the record phase. It is aimed at providing a faithful replay execution of a non-deterministic parallel program. It provides an output-determinism model.

The main challenge addressed by ODR is to reproduce the output of the original execution without recording the outcomes of every data-race by resorting to inference mechanisms. In consequence, the record and replay technique is augmented with a new phase: Deterministic-Run Inference (DRI) phase. This new phase is placed between the record phase and the replay phase.

The key point is to balance how much information the system need to record and how much information has to be inferred. ODR experiments with two approaches: search intense DRI (SI-DRI) and query intense DRI (QI-DRI). The former simply records the input-trace, lock-order and a path sample (order of input, output and lock instructions). The latter, in addition, records the branch-control of every thread.

ODR results show that using SI-DRI approach introduces less than 1.6x of slowdown on average. On the other hand, QI-DRI approach introduces an slowdown between 3.5x and 5.5x. Regarding the inference runtime, the slowdown is huge in both approaches. For instances, for some applications it does not even finished in 24 hours using SI-DRI approach.

CLAP (Huang, Zhang, & Dolby 2013) is a record/replay tool for C and C++ multithreaded programs. Regarding the kind of determinism that CLAP provides, it could be placed between value determinism and output determinism. CLAP tries to reduce the recording overhead by using inference mechanisms.

A CLAP execution can be divided in three main phases: the recording phase, the inference phase and the replaying phase. The recording phase tries to reduce the time overhead by not tracing either data races or synchronization operations. In contrast, CLAP traces the local execution path.

The inference phase uses symbolic execution. The path trace (the output of recording phase) is used to speed up this phase. Having the path trace makes the inference phase faster since only one execution path has to be explored. In other approaches, such as ODR, several execution paths have to be explored in order to find the proper execution path. In consequence, expensive operations, such as state backtracking, have to be performed. The path searching increments the complexity and the time overhead of the inference phase.

During the symbolic execution, CLAP gathers information about the execution such as the shared read/write operations, the order in which those shared read/write operations are performed by each thread, the synchronization operations and the path condition (a set of constraints that represents every symbolic branch decision). Thus, once the symbolic execution has finished, CLAP translates the gathered information into a global formula that whose solutions represent thread schedule candidates. Then, the constraint solving phase starts.

The authors propose a parallel constraint solving algorithm that permits CLAP to check different thread schedule candidates in parallel. In addition, CLAP also proposes an optimization in their constraint solving algorithm in order to reduce the number of context switches to the minimum. The authors argue that it is easier for developers to understand the bug when few context switches are present in the thread schedule.

Once a thread schedule that reproduces the buggy execution has been found, the replaying phase starts. It simply contains a thread scheduler component that follows the trace generated by the inference phase.

The evaluation of the system shows that CLAP is efficient in terms of time recording overhead. In terms of correctness, CLAP is capable of finding the buggy execution in the applications the authors tested.

DCR (Altekar & Stoica 2010) is a tool for debugging data-center applications such as HDFS/GFS (Ghemawat, Gobioff, & Leung 2003), HBase/Bigtable (Chang, Dean, Ghemawat, Hsieh, Wallach, Burrows, Chandra, Fikes, & Gruber 2008) and Hadoop/MapReduce (Dean & Ghemawat 2008). It is a software-based approach. It mixes inference mechanism and a content-based approach. The main idea of DCR is based on the observation that reproducing the original behaviour of the control-plane code should be enough for deterministically replay the execution. The control-plane code of a data-center is the code that controls data-flow such as discovering where a requested object is stored or maintaining the consistency of a replicated object. The authors claim that over 90% of the newly-written code in data-center applications is control-plane code; therefore, it is the main source of bugs. Furthermore, it is responsible for only 1% of the data traffic. The kind of determinism that this approach provide is called control-plane determinism. Nevertheless, it is simply a specific debug determinism.

In consequence, their approach is based on relaxing its determinism guarantees. DCR records control-plane inputs and outputs. Inputs refer to any communication between CPU; including both CPUs in the same node and CPUs in different nodes. Since data-plane inputs are not recorder, DCR cannot use traditional replay mechanism of re-executing the program with the same inputs. In contrast, an inference mechanism has to be used. They designed DDRI (Distributed Deterministic Run Inference) in order to infer the unknown inputs of the re-execution.

DDRI works in two phases: global formula generation and global formula solving. The former translates the distributed application into a formula where the data-plane inputs are the unknown variables. The latter tries to solve the formula; therefore, it provides the sufficient information to instantiate a control-plane deterministic run. Nevertheless, for this kind of applications, a huge formula is created in the first phase. Solving these formulas is impossible in many cases; therefore, they overcome this challenge by creating a new version of DDRI.

The new version, called JIT-DDRI, is an on-demand version of DDRI that works with an smaller amount of data. According to the authors, this decision arises from the observation that developers simply need to inspect a portion of the execution. Therefore, it is enough to solve those portions of the formula in which the developer is interested.

DCR satisfies the following requirements:

- *Always-on operation*
- *Whole-system replay:* The system is able to replay all involved nodes.
- *Out-of-the-box use:* It is able to replay arbitrary user-level application on modern commodity hardware.

In order to evaluate their system, the authors compare DCR with a version, called C&D, that traces both control and data plane non-determinism. As expected, DCR utilizes less bandwidth resources in record phase than C&D. On the other hand, they also compare the time overhead. DCR performs about 60% better than C&D. Furthermore, the execution is only 20% slower than the native execution. Real data-center applications have been used for the experiments.

2.5 Incremental replay

Some additional problems may appear when trying to debug parallel and distributed programs. For instance, in long-running executions, we cannot trace forever since the resources are limited. Furthermore, after a crash, the replay has to start from the beginning, and in long-running executions, this might be a problem. In order to solve this issues, incremental replay techniques has been introduced.

For supporting incremental replay, processes have to perform periodic checkpointing to save their state. In addition, some messages have to be traced in order to correctly reload the processes states. For instance, in some approaches, in-transit messages are traced; otherwise, that information would not be part of the checkpointed state of the system.

Considerable research has been done in how to perform the checkpointing and message logging. This research has been mainly done in the context of fault tolerance computing. Incremental replay is simply the application of those methods in the context of record/replay techniques. Thus, due to the characteristics of the record/replay techniques, some of these incremental replay techniques may differ from classical checkpointing techniques.

For instance, according to (Netzer & Xu 1993), the replay execution does not need to start from a consistent checkpoint; therefore, some parts of the coordination between processes might be simplified. On the other hand, checkpointing techniques applied for providing fault tolerance only require the ability

of replaying from the last checkpoint. Nevertheless, according to (Netzer, Subramanian, & Xu 1994), this is not necessary true for debugging purposes.

These techniques are not substitutive of the previous record/replay techniques. Tracing the order of the messages between participants is still needed. Incremental replay is a technique to improve and accelerate the replay execution.

In this subsection, we first summarize some of the classical checkpointing techniques. Then, we briefly describe some specific incremental replay techniques exclusively designed for record/replay techniques.

2.5.1 Checkpointing

According to (Elnozahy, Alvisi, Wang, & Johnson 2002), there are three main checkpointing techniques:

- Coordinated checkpointing
- Uncoordinated checkpointing
- Communication-induced checkpointing

Coordinated checkpointing technique forces every process to save their states in a coordinate way. The coordinator (one of the processes) starts the procedure instructing every other process to save the state. Once every process has taken its snapshot, the checkpointing procedure can finish. All in-transit messages are logged. Otherwise, some messages might get lost. For instance, in a scenario in which A sends a message to B, then A takes its snapshot and B takes its before the message arrive, the message would not be part of the state saved by the checkpointing technique.

The main disadvantage of this approach is that since all processes are forced to take the snapshot, it could be an inappropriate moment and lots of messages could be logged. Nevertheless, it makes the recovery much easier than other approaches.

On the other hand, uncoordinated checkpointing allows processes to locally take snapshots independently. Those checkpoints are not coordinated; therefore, every process can pick the most appropriate moment to take them. There are several disadvantages with this approach. The first disadvantage, and the most important, is that domino-effects can appear. A domino-effect can force to restart the execution far back in the execution line. Therefore, the effort of having snapshots of the system can be worthless since in the worst case, the system could be forced to restart the execution from the beginning. Secondly, a process could take useless checkpoints. These useless checkpoints will not compose a global consistent state. Thirdly, uncoordinated checkpointing forces to maintain an undefined number of checkpoints. Therefore, a garbage collector has to be periodically used in order to delete the useless stored checkpoints.

Finally, communication-induced checkpointing tries to avoid domino-effect without requiring taking all snapshots in a coordinate fashion. In this technique, processes take two kind of checkpoints: local and forced. The former is an uncoordinated checkpoint; therefore, there is no need for communication between processes. On the other hand, a forced checkpoint is similar to a coordinate checkpoint. This kind of checkpoint is taken when a progress of the recovery line is needed. Processes piggyback protocol-information in every message that they exchange. Based on that information, when a process realizes that the creation of a useless checkpoints is likely to happen, it forces a coordinated checkpoint to break this tendency.

2.5.2 Incremental replay approaches

Although some of the previously presented record/replay techniques already solved this problem. In the following paragraphs, we present some articles that exclusively focused on incremental replay solutions.

Adaptive (Netzer & Xu 1993): This article presents an adaptive message logging for incremental replay of message-passing programs. It tries to log a minimum amount of messages while eliminating every domino-effect. It is a software based approach.

Despite the logging algorithm is an approximation, they claim that it effectively performs. Their experiments show that only 1 - 10 % of the messages are typically logged.

According to the authors, a replay does not need to start from a consistent checkpoint. In other words, any set of snapshots is valid in order to replay the execution. Therefore, they chose an uncoordinated checkpointing technique in order to minimize the number of logged message.

In this sense, the main contribution of the work is on how to detect which messages should be traced in order to remove the domino-effects. The main goal is to replay any interval of the execution. In this case, the intervals are delimited by the local checkpoints.

They define domino-effect message as those messages that cause dependences with previous intervals of the same process. The intervals are bounded by the taken snapshots. Thus, if the system tries to replay the second interval of a process execution and a message m1 forces to also replay the first interval; then, m1 is a domino-effect message and it has to be logged. In consequence, an execution has a domino-free replay if every interval can be replayed without replaying previous intervals of the same process.

Figure 2.1 shows an example in which m1 is a domino-effect message. If m_1 is not traced (assuming m_2 neither), in order to replay $I_{1,2}$, the replayer is forced to also replay $I_{1,1}$.

In order to achieve this goal, they designed an algorithm that dynamically checks whether the message is creating a domino-effect dependence or not. A replay dependence vector (RDV) is used

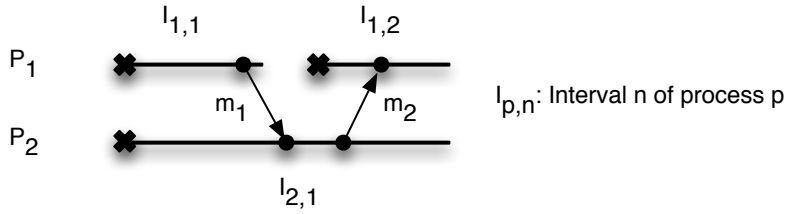

Figure 2.1: Example: Domino-effect messages

for checking these dependences. Each process maintains an RDV which contains the earliest interval in which the owner of the vector has a dependence with other processes. In his own position of the vector, it maintains his current interval. RDVs are piggybacked in every message; therefore, the receiver of the message can determine whether there is a dependence or not. In case there is a domino dependence, the message is traced. They claim that their algorithm guarantees a domino-free replay.

Regarding replay phase, the first step of their replay method is to compute the restart line. This line indicates which intervals of each process have to be replayed in order to re-generate the messages, that are needed by the interval that the user wants to replay. Once the restart line is generated, the replayer has to compute which fraction of the related intervals the system has to replay. Therefore, the replayer does not need to always re-execute the whole related intervals. This optimization improves the performance of the replay execution.

Critical-Path (Netzer, Subramanian, & Xu 1994) is a software-based incremental replay technique. Critical-Path tries to improve the previous solution by claiming that some domino dependences are short and quick to reproduce. Therefore, there is no need to log every message that produces a domino dependence. Furthermore, deleting all domino dependences does not guarantees a fast replay. Sometimes, the replay could be faster by logging some messages that are not domino-effect messages. Their experiments show that improvements in terms of spatial overhead can be achieved.

The main goal of this new approach is to permit the users to define an upper bound in terms of replay execution time. This bound allows the algorithm to log simply the needed messages in order to guarantee the upper bound. Therefore, for some particular executions and upper bounds, we need to log less messages than using other approaches such as domino-free approach.

In order to guarantee a bound, some assumptions have to be made:

- A re-executed run needs the same cpu time that in the original execution
- Every message needs the same time to be delivered in both executions.

Tracing algorithms: Critical-Path supports two algorithms. Both are based in the same idea.

Nevertheless, the second is optimized in terms of bandwidth; therefore, overhead.

The main idea of both algorithms consults in checking, after every checkpoint or synchronization operation whether we need to log a message taking into account an upper bound. This upper bound is imposed by the user and the system itself. This is done in runtime.

The algorithm simply requires one integer to be piggybacked in the messages. In consequence, the authors claim that the incurred overhead is rather low. On the other hand, finding the minimum set of messages to log is an NP-complete problem; therefore, their solutions are approximate algorithms.

Replaying the execution from an intermediate interval: The replay execution is done in several steps:

- Firstly, the log is scanned to find the intervals that have to be replayed in order to replay the execution from the requested interval.
- For each interval, the algorithm has to find the last event, on which the interval the algorithm tries to replay, depends. This event is also called requisite event.
- Once the previous computation is done, Critical-Path re-executes the intervals on the same process that was previously executed. Every interval is executed until the requisite event is executed.

An optimization can be done in order to improve the performance. If the execution of an interval gets blocked because of a blocking receive operation, a different interval (originally executed in the same processor) can start its execution. This mechanism avoids wasting time due to blocking operations.

The evaluation of the technique shows that Critical-Path is able to satisfy most of the user bounds by only logging between 0.1 - 5% of the messages.

Summary

This section presents a comparison between all the solutions we have previously presented. This summary does not pretend to be a performance comparison since each system has used different applications in order to test their systems. Furthermore, systems are envisioned to accomplish different requirements and to achieve different goals.

Table 2.1 summarizes the presented systems. It categorizes them according to different properties such as the technique they are using (approach field), whether is software-based approach, whether gives support for parallel and/or distributed programs, and which kind of determinism is provided.

System	Approach	Software only	Parallel Programs	Distributed Programs	Determinism
InstantReplay	Record&Replay	Yes	Yes	Yes	Value
LEAP	Record&Replay	Yes	Yes	No	Value
Ditto	Record&Replay	Yes	Yes	No	Value
Netzer	Record&Replay	Yes	No	Yes	Value
MPL*	Record&Replay	Yes	No	Yes	Value
MPreplay	Record&Replay	No	Yes	Yes	Value
Non-intrusive	Record&Replay	Yes	Yes	Yes	Value
DJVM	Record&Replay	Yes	Yes	Yes	Value
liblog	Record&Replay	Yes	Yes	Yes	Value
CURRF	Record&Replay	Yes	No	Yes	Value
Recon	Record&Replay	Yes	Yes	Yes	Value
CLAP	R&R + Inference	Yes	Yes	No	Value
ODR	R&R + Inference	Yes	Yes	No	Output
DCR	R&R + Inference	Yes	Yes	Yes	Control-plane (Debug)
Adaptive	Incremental replay	Yes	-	Yes	-
Critical-Path	Incremental replay	Yes	-	Yes	-

Table 2.1: Summary of the presented solutions

Although experimental results are not comparable, techniques (such as CLAP, ODR and DCR) that use inference mechanisms produce a lower runtime overhead. Nevertheless, those approaches apply a relaxation in the kind of determinism that they provide. Thus, CLAP, ODR, and DCR, although reproducing the bug, provide less useful information to the developers. In consequence, it might be more difficult for the developers to find the source of the bug.

3 Symber System

This chapter introduces Symber, a system that provides fault reproduction for multithreaded applications, based on inference mechanisms and record/replay techniques. The chapter starts with Section 3.1, introducing the main design goals of Symber. The chapter follows with a description of CLAP, a similar system for C/C++ programs. An overview and the architecture of Symber is found in Sections 3.3 and 3.4. Then, Section 3.6 describes the constraint model in detail. Finally, in Section 3.7, we talk about the implementation details and the remarkable challenges we faced.

3.1 Introduction

As mentioned in previous chapters, the main goal of Symber is to deterministically replay a buggy execution in multithreaded applications. Since directly applying record and replay techniques introduces an unbearable overhead, Symber combines the techniques used by tools such as (Silva) and (Huang, Liu, & Zhang 2010) with an inference phase, that helps to reduce both the spatial and time overhead incurred by the tool. It is worth to mention that, to the best of our knowledge, Symber is the first record/replay tool for Java that uses symbolic execution as inference mechanism. Since we had to develop all the support for inference from scratch, the development was particularly complex and we have faced several new challenges that are enumerated later at the end of the chapter.

The idea of using inference has been previously applied with success in systems like ODR (Altekar & Stoica 2009), DCR (Altekar & Stoica 2010), and CLAP (Huang, Zhang, & Dolby 2013). However, we consider that previous works have not thoroughly addressed all the factors that must be considered when inference is used. Namely, we are concerned in achieving the right tradeoff among the following factors:

- The time overhead of the record phase, that basically depends on the amount of information the tool traces.
- The time that the inference phase takes; the more information our tool traces during the record phase, the faster the inference phase is.
- The amount of debugging information that the tool provides to the developer. From our point of view, this is extremely important factor. We consider that a useful tool should not be only able

to replay the bug, but it should also provide some useful information to the developer. The main purpose is to help the developer to find the origin of the bug.

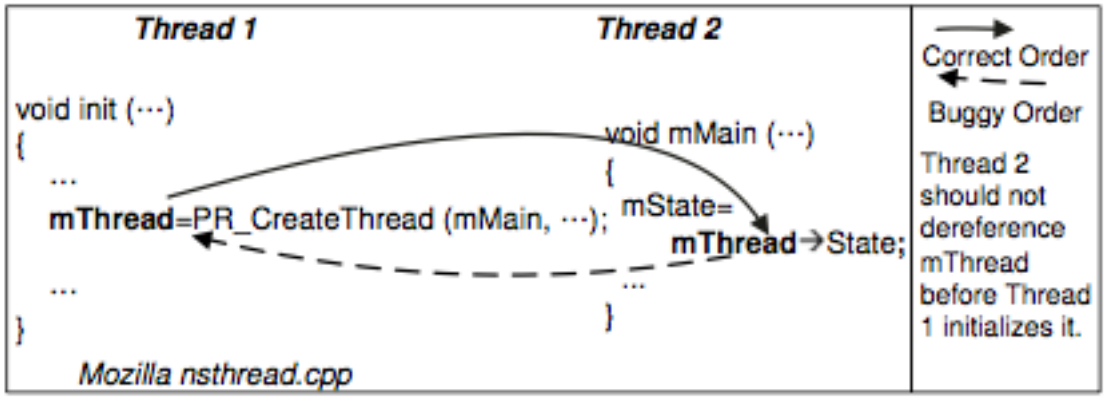

Figure 3.1: Order violation bug.

We now discuss how these factors are taken into consideration by previous work:

- ODR is characterized by an efficient record phase. However, the inference phase can take too much time in the worst scenario. Furthermore, it provides little clues about the source of the bug. Figure 3.1 shows a simple bug that produces this behaviour in ODR. In this example, the ODR replayer will be able to reproduce the bug; nevertheless, the produced thread schedule may not represent the buggy execution. This happens because the data race that produces the bug is not involved in any *if* condition.

- On the other hand, we consider that CLAP meets the requirement of providing to the developer enough information for finding the origin of the bug. However, the inference phase needs too much time to find the buggy execution in the worst scenario because the amount of possible executions is not sufficiently restricted.

- Finally, DCR is aimed at data-center applications. Therefore, its approach it is not comparable to ours. Nevertheless, the idea of considering a heavy-weight recording for control-plane code is also based on the observation that previous approaches do not give enough information to the developers.

Our hypothesis is that by adding a small overhead in the record phase (when compared to what CLAP and ODR do), it should be possible to dramatically reduce the inference phase. The design and development of Symber was performed with the goal of validating this hypothesis.

3.2 CLAP

Symber can be seen as an evolution of the CLAP system, that we have adapted in two different dimensions: first, CLAP works for C/C++ programs and we were interested in building a tool for Java programs; second, we have change the amount of information captured in the record phase, to speed up the inference phase. Given the importance of CLAP to our work, in this section we provide a detailed introduction to CLAP.

CLAP is a record/replay tool that includes a inference phase based on symbolic execution. This inference phase is placed between the record phase and the replay phase of the traditional record and replay techniques. The goal of the inference phase is to find the buggy execution based on the traces generated by the record phase. It works for multithreaded C/C++ applications that use PThreads.

Record phase As it has been discussed in the previous chapter, in order to faithfully replay a buggy execution, one needs to remove all non-determinism from the replay run. This involves capturing in the record phase all non-deterministic events, including information on how threads interleave. Unfortunately, we have also discussed that this is extremely expensive. To mitigate this problem, CLAP proposes to trace the execution path locally, i.e. without synchronization between threads, as an alternative. It provides a more efficient record phase but, unfortunately, it is not enough to completely eliminate the non-determinism from the replay. Therefore, some extra computation is needed in order to remove it.

The purpose of tracing the execution path is to guide the symbolic execution on which the inference phase is based. Thus, there is no need to explore all the possible executions since the path is defined. This approach accelerates the inference phase since there is no need to backtrack to previous states as traditional symbolic executors do.

Symbolic execution CLAP uses a concolic (concrete+symbolic) symbolic execution, in the sense that only operations over shared variables create fresh symbolic symbols, whereas operations on other variables are concrete as in a normal execution. During the execution, there is no need to care about the data races since no concrete values are assumed when reading from a shared variable. The goal of this phase is to gather enough information for generating a formula that represents the execution.

Formula generation Once the symbolic execution has finished, CLAP generates a formula composed by constraints that represent the execution. The goal of this formula is to sort read and write operations and, thus, eliminate data races. By eliminating data races, the non-determinism present in the application is also eliminated since, indirectly, it represents how threads interleave. CLAP generates the following formula:

$\phi = \phi_{path} + \phi_{bug} + \phi_{so} + \phi_{rw} + \phi_{mo}$

where:

- ϕ_{path} denotes the path constraints. These are gathered by the symbolic executor. Each constraint represents one symbolic branch decision. A symbolic branch decision refers to an *if* statement in which at least one operand is symbolic.
- ϕ_{bug} represents the bug.
- ϕ_{so} denotes the synchronization order constraints. They are composed of two types of constraints, namely *locking constraints* and *partial order constraints*. Locking constraints are only concerned with the *lock* and *unlock* operations. On the other hand, partial order constraints are related to *fork*, *start*, *join*, *exit*, *wait*, and *signal* operations.
- ϕ_{rw} represents the relationship between *read* and *write* operations, and restricts the order of those operations.
- ϕ_{mo} denotes the order in which *read*/*write* operations have been executed in a specific thread. CLAP not only supports the sequential consistent memory model, but it also supports both the partial store ordering consistency model and the total store ordering consistency model (in case they are supported by the hardware).

Constraint solving algorithm CLAP does not try to feed a constraint solver with the whole formula. Instead, it first divides consecutive shared access points into segments with different sizes, according to whether exists a context switch between them or not. Then, CLAP leverages a technique, denoted preemption-bounding, where it tries to solve the constraint formula by iteratively increasing the number of context switches (in case the solver fails to return a solution). This allows to find the bug-reproducing schedule with minimal thread context switches, which may be useful to understand more easily the cause of the error.

Furthermore, CLAP is able to speed up this process, as formulas with different number of context switches can be solved in parallel.

Replay phase Once the solver has found a solution, the produced thread interleaving is used to enforce the bug reproduction. In practice, CLAP runs a previously instrumented version of the program that, before each shared memory access, checks the schedule to see whether the running thread is the one that should execute the operation at that moment. If it is, the execution proceeds normally, whereas if it not, the thread is blocked and waits until its turn.

3.3 Overview of the Tool

A regular execution of Symber starts with a static analysis. The goal of this analysis is to prepare the target application for Symber. This preparation is basically an instrumentation of the application code. As an output of this phase, a record and a replay version of the application are generated. Subsequently, the record phase can start. Traces are filled during this phase with not only the execution path but also with the locking order. Then, using the traces, the inference phase starts. This is the most complex and slow phase because it has to find an execution that triggers the same bug than the original execution among all the possible executions. Once the buggy execution has been found, a trace of the accesses to shared variables is created and fed to the replay phase. Then, the buggy execution can be replayed as many times as needed using the trace generated during the inference phase. Figure 3.2 illustrates the different phases involved in a Symber execution.

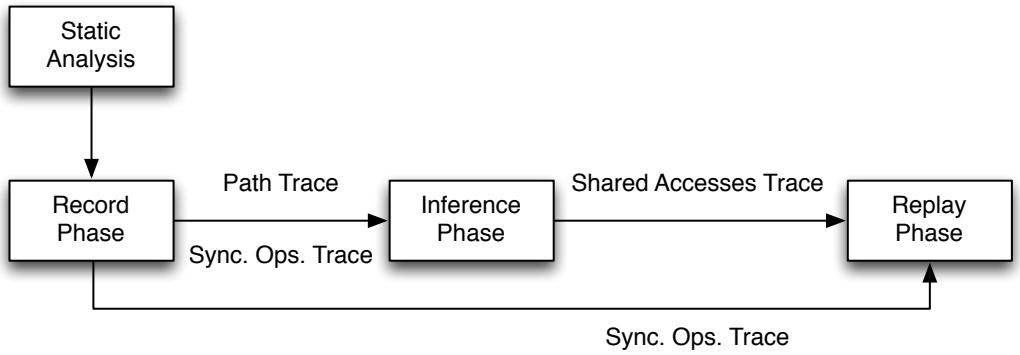

Figure 3.2: Symber phases.

The inference phase is based on symbolic execution (Floyd 1993) (King 1976) (Cadar & Sen 2013). It uses both the path and the locking order traces to guide the execution. It creates a new fresh symbolic symbol for each shared read operation. Therefore, we do not assume anything with regard to data races. During the symbolic execution, Symber creates constraints that represent the execution and feeds them to a constraint solver. The constraint solver sequentially outputs solutions for those constraints that represent thread schedule candidates. The candidates are sequentially checked until the bug is reproduced are sequentially checked until the bug is reproduced. Finally, once the buggy execution has been found, a trace with the order of shared accesses and the lock order is generated and sent to the replayer.

Symber uses some ideas that were introduced by CLAP; nevertheless, it differs in many aspects. Similarly to CLAP, it locally logs the execution path on runtime. This path profiling does not need any synchronization among threads; therefore, it is substantially cheaper than other techniques based on tracing the order or the value of shared memory accesses. However, we believe that logging the local execution path does not provide sufficient useful information. In consequence, the inference phase could need to generate and check an unbearable number of execution candidates. For this reason, Symber not

only logs the execution path but it also traces the locking order (by locking order we refer to a global order of synchronization operations such as *signal* and *wait* operations or *lock* and *unlock* operations).

There are several remarkable considerations behind this decision:

- Synchronization operations are already synchronized among the threads. Therefore, tracing the order does not incur a major overhead. It produces the opposite effect of tracing the order of shared memory accesses which might not be synchronized. Since one of the major challenges is to reduce the recording overhead of classic record and replay techniques, we opt for only tracing the locking order.
- The evaluation of ODR presented in (Altekar & Stoica 2009) supports our approach. On their paper, they show that the overhead incurred by tracing the locking order is rather low.
- Tracing the locking order reduces the amount of possible solutions dramatically because there is no need to explore all executions defined by different order of synchronization operations. In other words, it restricts the search space that the constraint solver has to explore. Thus, we believe that the inference phase of Symber should be faster and simpler than CLAP's inference phase.

Furthermore, in consequence to this decision, the inference phase of Symber differs in many parts to the inference phase of CLAP as we will see in the following sections.

3.3.1 Example

Figure 3.3 shows a simple example that helps to visualize the benefits of Symber in comparison to CLAP. It represents a multithreaded application that contains both synchronized and non-synchronized accesses to shared variables. R_xn (W_xn) denotes a read (write) operation over the shared variable x (n simply denotes the line in the source code).

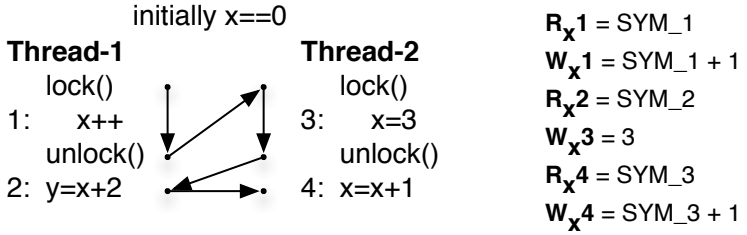

Figure 3.3: Simple multithreaded program.

During the symbolic execution phase of CLAP and Symber, shared read/write operations are identified and logged. A read operation creates a new fresh symbolic symbol (E.g. the operation R_x1 creates

the symbolic symbol SYM_1). For write operations, the tool simply stores the value that has been written.

Once the symbolic execution has finished, CLAP generates a set of constraints that help to infer the original execution. Figure 3.4 presents a simplified version of the constraints that CLAP would generate for the example. OR_xn denotes the order of the corresponding operation (R_xn) in the to-be-computed thread schedule. Memory-order constraints represent the order in which the operations have been locally executed by each thread. Thus, each row depicts the operations executed by one thread. Read-write constraints are aimed at ordering the shared read/write operations among threads.

In Figure 3.4, read-write constraints have already been simplified by the memory-order constraints. Thus, according to CLAP's constraint model, SYM_1 can only be equal to 0, 3 or $SYM\_3 + 1$, SYM_2 can exclusively be equal to $SYM\_1 + 1$, 3 or $SYM\_3 + 1$, and SYM_3 has to be equal to $SYM\_1 + 1$ or 3. Furthermore, when assigning a concrete value to a symbolic variable, order restrictions among the operations are also added. E.g. if we assign 0 as a value for SYM_1 (x's initial value), operation R_x1 has to be executed before any other write operation. Apart from the information represented in the Figure 3.4, CLAP cannot extract more information from the symbolic execution since no assumptions can be made regarding the order in which locks are acquired.

memory-order constraints

($OR_x1 < OW_x1 < OR_x2$) &&
($OW_x3 < OR_x4 < OW_x4$)

CLAP read-write constraints

((**SYM_1** = 0 & ($OR_x1<OW_x1$) & ($OR_x1<OW_x3$) & ($OR_x1<OW_x4$)) |
(**SYM_1** = 3 & ($OW_x3<OR_x1$) & ($OW_x1<OW_x3$ | $OW_x1<OR_x1$) & ($OW_x4<OW_x3$ | $OW_x4<OR_x1$)) |
(**SYM_1** = SYM_3+1 & ($OW_x4<OR_x1$) & ($OW_x1<OW_x4$ | $OW_x1<OR_x1$) & ($OW_x3<OW_x4$ | $OW_x3<OR_x1$)))
&&
((**SYM_2** = SYM_1+1 & ($OW_x1<OR_x2$) & ($OW_x3<OW_x1$ | $OW_x3<OR_x2$) & ($OW_x4<OW_x1$ | $OW_x4<OR_x2$)) |
(**SYM_2** = 3 & ($OW_x3<OR_x2$) & ($OW_x1<OW_x3$ | $OW_x1<OR_x2$) & ($OW_x4<OW_x3$ | $OW_x4<OR_x2$)) |
(**SYM_2** = SYM_3+1 & ($OW_x4<OR_x2$) & ($OW_x1<OW_x4$ | $OW_x1<OR_x2$) & ($OW_x3<OW_x4$ | $OW_x3<OR_x2$)))
&&
((**SYM_3** = SYM_1+1 & ($OW_x1<OR_x4$) & ($OW_x3<OW_x1$ | $OW_x3<OR_x4$) & ($OW_x4<OW_x1$ | $OW_x4<OR_x4$)) |
(SYM_3 = 3 & ($OW_x3<OR_x4$) & ($OW_x1<OW_x3$ | $OW_x1<OR_x4$) & ($OW_x4<OW_x3$ | $OW_x4<OR_x4$)))

Figure 3.4: Constraints generated by CLAP for the example shown in the Figure 3.3.

On the other hand, Symber not only traces the local execution path (as CLAP does), but it also logs the order in which locks are acquired. Thus, apart from having the symbolic execution guided by the path trace, it is also guided by the locking order. In consequence, information regarding the global execution path is generated by the symbolic executor (in case the application has synchronization operations).

Figure 3.5 shows the constraints generated by Symber. Assuming that the original execution followed

the execution depicted by the arrows in Figure 3.3, Symber has the ability to generate the synchronization constraints that represent the locking order. Thus, Symber can assume that the instruction $x++$ was executed before the instruction $x = 3$. Knowing this extra information, let Symber simplify the read-write constraints , and, in consequence, the number of possible solutions. Figure 3.4 shows the read-write constraints after the simplification done by both the synchronization constraints and the memory-order constraints.

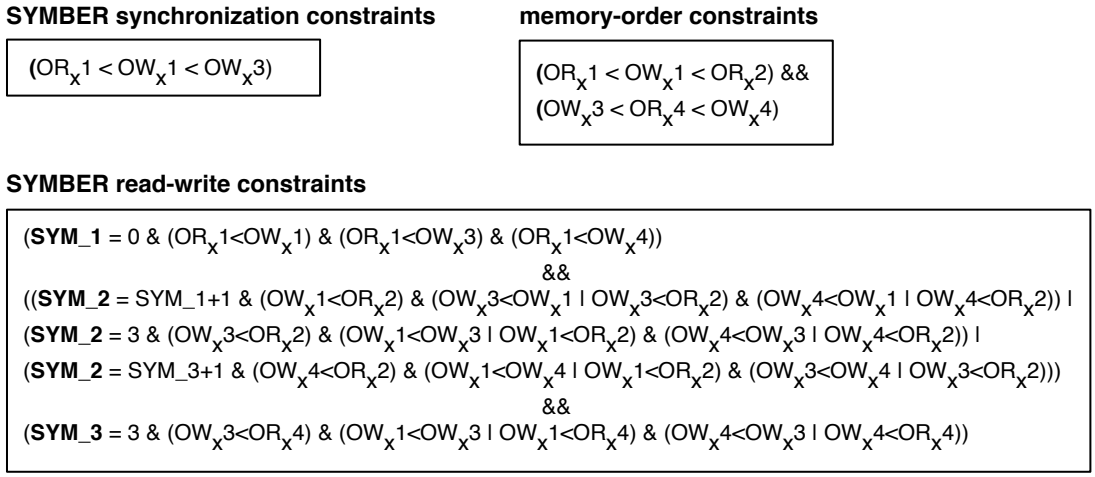

Figure 3.5: Constraints generated by SYMBER for the example shown in the Figure 3.3.

For the example depicted by the Figure 3.3, Symber read-write constraints only have three solutions, while, for CLAP, the number of solution is multiplied by four (12). Furthermore, CLAP also needs to infer the order in which the locks were acquired during the original execution. This fact notoriously complicates CLAP's constraint solver component.

In the worst case scenario, when there are no shared accesses in synchronized blocks, the number of solutions of Symber's constraints is equal to the number of solutions that CLAP generates. Furthermore, we can assume that not having synchronized shared accesses implies not having synchronized blocks at all. In consequence, record overhead would be the same for both CLAP and Symber.

3.4 Architecture

Symber is a complex tool that is composed of several components, as depicted in Figure 3.6. Those components can be aggregated in four groups as follows:

- **Transformer:** Its main goal is to generate both the record and the replay version of the target application.
- **Recorder:** It is in charge of filling the trace.

- **Inference Tool:** It contains all necessary components for finding the buggy execution.
- **Replayer:** It replays the buggy execution by following the traces.

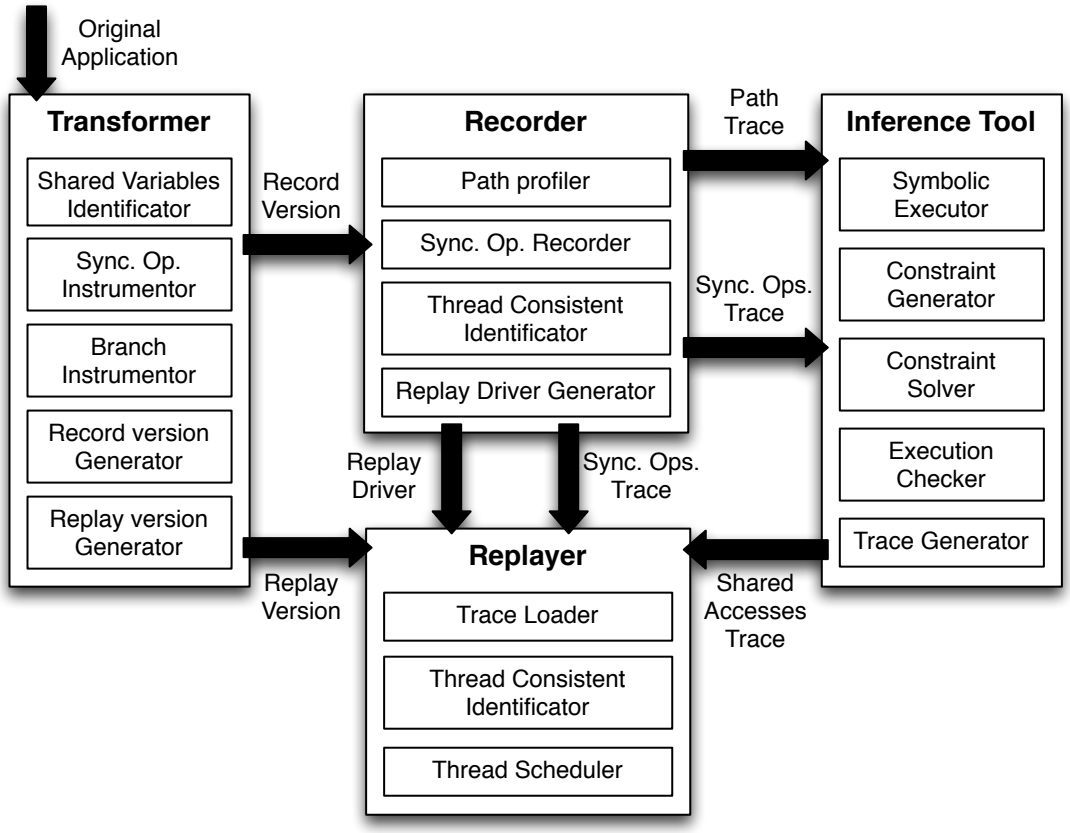

Figure 3.6: Symber architecture.

3.4.1 Transformer

The transformer generates both the record and replay version of the target application. It uses the target application as an input and instruments it with additional instructions, as follows. The record version instrumentation is aimed at tracing the execution path and the locking order. On the other hand, the replay version instrumentation is aimed at adding the needed mechanisms for following the traces. Furthermore, the transformer has the ability to identify shared variable accesses. This identification is used during the inference phase.

3.4.2 Recorder

This component is composed by the path profiler, the synchronized operations recorder, and the replay driver generator modules. The path profiler is in charge of creating the path trace on runtime.

The synchronized operations recorder is used for generating the locking order trace. Finally, the replay driver generator creates the replay driver. This driver is needed during the replay phase and it is in charge of executing the replayer and loading the generated traces.

3.4.3 Inference Tool

This tool comprises several other sub-components (see Figure 3.6). All those components work together in order to find the buggy execution. The inference tool receives the path trace and the locking order trace from the recorder. Then, it symbolically executes the application following the traces in order to gather information about the execution using the symbolic executor module. With the gathered information, the constraint generator is capable of generating the constraints. The constraint solver module is in charge of finding solutions to the set of constraints. Finally, the trace generator module has the ability to create the trace, needed by the replayer, from the solutions that the constraint solver provides.

3.4.4 Replayer

The replayer component is able to replay the buggy execution as many times as the developer needs. It loads the trace generated by the recorder and the inference tool. For the bug reproduction, the replayer does not require the path trace since it is already represented by the shared accesses trace generated by the inference tool. Once the trace is loaded, using the thread scheduler component, it is able to follow the trace and replay the buggy execution.

3.5 Phases in Detail

This section is aimed at explaining in detail the phases that are involved in a Symber execution. It also motivates the need for those phases and how they have been implemented. The multiple phases, depicted in Figure 3.2, can be classified as follows:

- Static analysis
- Record phase
- Inference phase
- Replay phase

Each phase is executed by one of the Symber components. Thus, the static analysis is performed by the transformer, the record phase is executed by the recorder, the inference tool is in charge of the inference phase, and the replayer performs the replay phase.

3.5.1 Static analysis

This is the first phase of Symber's execution. It statically analysis the target application. It identifies all shared variable accesses and logs them into a file. This file will be used during the inference phase. Furthermore, during this phase, the record and the replay version are created.

On one hand, for the record phase, the static analysis needs to prepare the application for tracing the local execution path and the locking order. In order to trace the local execution path, we opted for a simple approach. For each *if* statement, two method invocations are injected. One right before the statement (*beforeIfStmt* hereafter) and one right after (*afterIfStmt* hereafter). Thus, *beforeIfStmt* is called in case the if statement is about to be executed and *afterIfStmt* is executed only if the condition is satisfied.

Switch statements are instrumented in a different way. The tool injects one method invocation after any of the targets of the switch statement. Thus, we are able to trace which *case* clause has been executed on runtime. In addition, a method is inserted right before the *switch* as it is done with the *if* statements.

We are aware that some more efficient approaches could have been used for tracing the execution path (Ball & Larus 1996). Nevertheless, that is not the goal of Symber. Therefore, due to the temporal constraints we had to develop the prototype, we opted for this more straightforward approach. Furthermore, the path profiler is pluggable component; therefore, changing it by a more efficient one is rather easy and it does not require large modifications to the current Symber architecture.

On the other hand, in order to trace the locking order, we not only need to trace lock acquisitions but we also have to log any signal and wait operation. The instrumentation is made by inserting *beforeMonitorEnter* method before any signal operations and *afterMonitorEnter* method right after any wait operation or lock acquisition.

Regarding the replay version instrumentation, read/write operations are wrapped by *beforeRead*, *afterRead*, *beforeWrite* and *afterWrite* methods. On the other hand, since it also has to follow the locking order trace, synchronization operations are also wrapped by *beforeMonitorEnter* and *afterMonitorEnter* methods.

In the following sections we describe in detail the operation of these wrappers.

3.5.2 Record phase

This phase uses the version generated by the static analysis. It runs the application and generates both the path trace and the locking order trace.

Figure 3.7 shows an example of how the path profiling algorithm works. In the figure, methods in **bold** are the instrumented methods. The method *beforeIfStmt*, which goes right before an if statement, provisionally stores (represented by [] in the example) a "false" on the thread's path trace. If the next method for that thread, among those two methods, is an *afterIfStmt* method, that value is changed to true and stored in the permanent path execution log. In case next invocation is another *beforeIfStmt*, the provisional "false" is stored into the permanent path execution log. A similar idea is used for loops and switch statements.

	c1 == True c2 == False	c1 && c2 == True	c1 == False c2 == True	c1 && c2 == False
beforeIfStmt()	[F]	[F]	[F]	[F]
if (c1){				
afterIfStmt()	T	T		
stmt 1				
}else{				
stmt2				
}				
stmt3				
beforeIfStmt()	T , [F]	T , [F]	F , [F]	F , [F]
if (c2){				
afterIfStmt()		T , T	F , T	
stmt4				
}				
...				

Figure 3.7: Path profiler.

On the other hand, for tracing the locking order, we have been inspired by the algorithms implemented by Ditto(Silva). We consider as synchronization operations any monitor acquisition (synchronized method, synchronized block or lock method) and any signal/wait operation (*wait*, *notify*, *notifyAll*, *await*, *signal*, and *signalAll*). This algorithm is aimed at tracing the global order of synchronization operations per locking object. For this purpose, we maintain a clock for each locking object and for each thread. Both *beforeMonitorEnter* and *afterMonitorEnter* implement the algorithm presented in Figure 3.8.

Apart from filling the logs, this phase also generates the replay driver. It basically links the generated trace and the target application with the replayer. The goal of this driver is to automatize the process. Thus, once the inference phase finishes (and, in consequence, the trace that represents the buggy execution has been generated), the user simply needs to run the replay driver to faithfully replay the

```
Parameters: o is the synchronization object
                t is the current thread
  method afterMonitorEnter(o, t)
      TRACE(o.syncClock)
      newClock  = MAX(t.syncClock, o.syncClock) + 1
      o.syncClock = newClock
      t.syncClock = newClock
  end method
```

Figure 3.8: Recording locking order algorithm.

buggy execution.

3.5.3 Inference phase

This is the most complex phase. We can divide it into smaller phases in order to better explain in detail how it works. Figure 3.9 shows how the phase is divided and how those sub-phases interact.

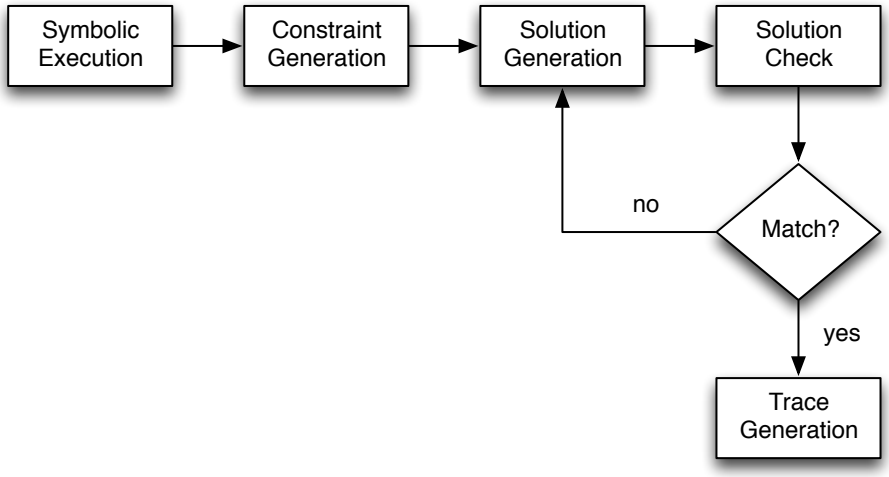

Figure 3.9: Inference phase in detail.

Symbolic execution First, the path, locking order, and shared variable operation traces are read. Then, the symbolic execution starts. It is a concolic symbolic execution; therefore, some variables are concrete and some are symbolic. In our tool, any shared read operation creates a new fresh symbolic variable. The identification of shared variable operations is made using the shared variable operation trace generated by the static analysis phase. This trace simply lists all shared memory operations.

Path and locking order traces are used for guiding the symbolic execution. Thus, for each branch in which at least one of the operands is symbolic, the decision is taken based on the path trace. On the other hand, before executing a synchronization operation, the locking order trace has to be checked. In the

case the thread is invoking the lock in the logged order, the thread continues the execution. Otherwise, the thread gets blocked and a context switch is performed. The thread keeps blocked until it is his turn according to the locking order trace.

Wait operations are an special case since the trace represent the order in which the threads are acquiring the lock after being waiting. Thus, threads are allowed to invoke a wait method at any time. Nevertheless, when a signal operation is called, before letting a waiting thread grab the execution, it is checked whether it matches the trace or not.

The goal of the symbolic execution is to gather as much information about the execution as possible. For this purpose, it records the following useful information:

- Path condition (PC): It gathers constraints related to the symbolic branch decisions. Thus, when a decision is taken, it is translated into a constraint and added to the PC. For instance, considering the following if statement (where $sym1$ is a symbolic variable): $if(sym1 > 5)$. If, according to the path trace, the condition was satisfied in the original execution, the constraint that is added to the path condition would be written as $sym1 > 5$, otherwise $sym1 \leq 5$.
- Shared memory accesses sorted by thread. Thus, it represents a local order of shared memory accesses.
- Write set and read set per shared variable.
- Initial values of the shared variables.
- Synchronized shared memory accesses sorted by locking object. Since the symbolic execution is guided by the locking order, we can be sure that this order is part of the buggy execution.

Once the symbolic execution has finished and all the information has been gathered, the symbolic execution sub-phase finishes and the constraint generation phases starts.

Constraint generation This sub-phase takes all the information produced by the symbolic execution and generates a global formula. This formula represents the group of possible executions to which the buggy execution belongs. Section 3.6 presents the constraint model in detail.

Solution generation Once all the constraints have been created, the formula is solved by a constraint solver. We are not interested on the values of the symbolic variables; we are just interested on the global order of read/write operations because it represents how the threads interleave during the execution.

Having the symbolic execution guided by the locking order let us to create the synchronization order constraints. This substantially reduces the search space and the number of infeasible solutions that the

solver might suggest. In contrast, since CLAP cannot guide the symbolic execution by the locking order (because it does not trace it), it cannot generate the synchronization order constraint and; therefore, a huge number of the solutions might represent an infeasible execution. E.g. CLAP might try an order of synchronization operations that produces deadlock, when the buggy execution does not.

Solution check and trace generation The checker takes the solution proposed by the solver and executes the target application in order to check whether it produces the bug or not. In case the bug is reproduced, the final trace is generated and the inference phase finished. Otherwise, it requests another solution to the constraint solver and the solution check phase starts again. Since the buggy execution belong to the group of the possible solutions, this phase will always finish.

The trace is generated according to the global order of read/write operations that the solution represents. Thus, each operation executes, in the order that results from the execution, the algorithm depicted in Figure 3.10. In order to generate the trace, each shared variable maintains a clock and a read counter. Furthermore, threads maintain an extra clock. It is important to notice that the clock maintained by the thread for read/write operations is not the same that the one used for synchronization operations. In consequence, threads have two independent clocks for tracing both read/write and synchronization operations.

```
Parameters: o is the shared object
            t is the current thread
  method readOperation(o, t)
      TRACE(o.storeClock)
      o.loadCount = o.loadCount + 1
      if o.storeClock > t.clock then
          t.clock = o.storeClock
      end if
  end method
  method writeOperation(o, t)
      TRACE(o.storeClock, o.loadCount)
      newClock  = MAX(t.clock, o.storeClock) + 1
      o.storeClock = newClock
      o.loadCount = 0
      t.clock = newClock
  end method
```

Figure 3.10: Tracing shared variable accesses algorithms.

An optimization has been applied in order to reduce the size of the generated traces. The idea is to store deltas values instead of absolute values. Thus, large numbers are never traced and a cheaper representations can be used (such as *short* instead of *int*).

```
Parameters: o is the shared object
            t is the current thread
  method beforeRead(o, t)
      monitorEnter(o)
      targetStoreClock = getNextLoadConstraint(t)
      while o.storeClock < targetStoreClock then
          wait(o)
      end while
      monitorExit(o)
  end method
  method afterRead(o, t)
      monitorEnter(o)
      if o.storeClock > t.clock then
          t.clock = o.storeClock
      end if
      o.loadCount = o.loadCount + 1
      o.notifyAll()
      monitorExit(o)
  end method
```

Figure 3.11: Read wrapper algorithm.

3.5.4 Replay Phase

The replay phase starts by reading the trace. This trace contains the order in which read/write operations and the synchronization operations have to be executed for reproducing the buggy execution. Then, the execution begins. Since it runs the instrumented version of the target application, each time a read/write operation or a synchronization operation is about to be executed, our injected methods (*beforeRead*, *beforeWrite*, or *beforeMonitorEnter*) are invoked. The purpose is to block the thread in case it does not have to execute the operation. Once, the thread has been allowed to execute the operation (because the trace matches the current execution), the trace advance and the clocks are updated by our second part of injected methods (*afterRead*, *afterWrite*, and *afterMonitorEnter*).

Figures 3.11, 3.12, and 3.13 show the algorithms that are implemented by the wrappers. This set of algorithms compose the "Thread Scheduler" component of the replayer.

Again, these algorithms are inspired by the algorithms used in Ditto(Silva). However, Symber is working above the JVM machine while Ditto is working at JVM's level; therefore, some adjustments have been done.

In contrast to other tools, such as LEAP (Huang, Liu, & Zhang 2010), Symber does not identifies shared variables by field name. We believe that using an instance identification permits to reduce the replay overhead and the identification is more accurate. Thus, Symber is capable of differentiate between instances of the same class.

```
Parameters: o is the shared object
            t is the current thread
  method beforeWrite(o, t)
      monitorEnter(o)
      targetStore = getNextStoreConstraint(t)
      while o.storeClock < targetStore.storeClock
          ||  o.loadCount < targetStore.loadCount then
          wait(o)
      end while
      monitorExit(o)
  end method
  method afterWrite(o, t)
      monitorEnter(o)
      newClock  = MAX(t.clock, o.storeClock) + 1
      o.storeClock = newClock
      o.loadCount = 0
      t.clock = newClock
      o.notifyAll()
      monitorExit(o)
  end method
```

Figure 3.12: Write wrapper algorithm.

```
Parameters: o is the synchronization object
            t is the current thread
  method beforeMonitorEnter(o, t)
      monitorEnter(o)
      targetSyncClock = getNextSyncConstraint(t)
      while o.syncClock < targetSyncClock then
          wait(o)
      end while
      monitorExit(o)
  end method
  method afterMonitorEnter(o, t)
      monitorEnter(o)
      newClock  = MAX(t.syncClock, o.syncClock) + 1
      o.syncClock = newClock
      t.syncClock = newClock
      o.notifyAll()
      monitorExit(o)
  end method
```

Figure 3.13: Monitor wrapper algorithms.

3.6 Constraint Model

Symber constraint model is inspired by CLAP. Nevertheless, it is much simple since we do not need to infer the locking order because it has been traced. Therefore, we can discard CLAP's "Synchronization Order Constraints". According to CLAP, the total size of the synchronization constraints of their model is:

$$N_{lo}(2|S|^3 + 2|S|) + N_{jo} + N_{fo} + N_{sv}(2|SG||WT| + |SG|)$$

where N_{lo} denotes the number of locking objects, S denotes the set of lock/unlock pairs of a specific locking object, N_{jo} denotes de number of join operations, N_{fo} represents the number of fork operations (start method in Java), N_{sv} denotes the number of signal variables, SG is the set of signal operation that operates in a specific signal variable and WT represents the set of wait operations that can be mapped by a specific signal operation.

We note that removing these constraints from our formula simplifies the constraint solving task, considering that we discard a cubic formula from our model.

Furthermore, CLAP cannot straightly solve the formula since the majority of the solutions would represent an infeasible execution and going trough all the solutions would require an unreasonable amount of time (depending on the target application). The lack of the locking order trace is the cause of this drawback. In contrast, Symber's constraint model notoriously restricts the number of infeasible executions. Therefore, the constraint solver and execution check phases get substantially simplified.

Symber's final formula is a composition of constraints. It can be written as:

$$\alpha = \alpha_{pc} \wedge \alpha_{rw} \wedge \alpha_{mo} \wedge \alpha_{so} \wedge \alpha_b$$

where α_{pc} denotes the path condition constraints, α_{rw} represents the read-write relationship constrains, α_{mo} represents the memory order constraints, α_{so} denotes the synchronization order constraints and α_b represents the bug.

The notation of the following constraints is simplified to make it readable, but more factors are taken into account for uniquely identify each read/write operation.

Path condition (α_{pc}) It represents the execution path. Thus, each constraint bounds the value of the symbolic variables involved in the if statement that generated the constraint. This group of constraints restricts the number of solutions for the read/write constraints.

Bug constraint (α_b) It is basically the translation of the bug into a constraint. It helps to bound the possible solutions of the formula to those in which the bug is triggered. Depending on the target

application and the bug, this constraint might be already included in the path condition. For instance, if the bug is triggered whether an if condition is satisfied or not.

Read-Write constraint (α_{rw}) It expresses the relationship between reads and writes operations for a specific shared variable. Thus, we try to discover for each read operation which write operation precedes it. The constraints for each read operation (r) are written as follows:

$$\left(V_r = init \bigwedge_{\forall w_j \in W} O_r < O_{w_j}\right) \bigvee_{\forall w_i \in W} \left(V_r = w_i \wedge O_{w_i} < O_r \bigwedge_{\forall w_j \neq w_i} O_{w_j} < O_{w_i} \vee O_{w_j} > O_r\right)$$

where V_r is the read value, *init* is the initial value of the shared variable, W is the write set for that shared variable, O_r denote the order of r and O_{w_i} the order of the write operation w_i.

If the value assigned to the read operation is *init*, O_r has to precede every $O_{w_i} \in W$. Otherwise, it has to be preceded by at least one write operation (O_{w_i}). Thus, V_r has to be assigned to its immediate predecessor. In consequence, some restrictions with regard to the operations order are induced. For instance, the read operation (O_r) has to happen after the write operation to which is assigned to (O_{w_i}) and no more write operations can be placed between them.

Memory order constraint (α_{mo}) It represent the local order for all read/write operations executed by a thread. Thus, if thread i executes O_{r_j} before O_{w_k} and O_{w_o} before O_{r_j}, a constraint is created as follows:

$$O_{w_o} < O_{r_j} < O_{w_k}$$

Synchronization order constraint (α_{so}) It represents the global order of shared variable accesses under a specific locking object. For those read/write operations that are performed in a synchronized block, we can assume a global order among all threads. Thus, if the locking order for a specific locking object o is t_j, t_i and t_k and each of those synchronized blocks performed a shared variable access operation (O_{r_j}, O_{r_i} and O_{w_k} respectively), the constraint would be generated as:

$$O_{r_j} < O_{r_i} < O_{w_k}$$

In addition, in the context of the synchronization order constraints, Symber is able to infer extra synchronization points between threads by taking into consideration *join* and *start* methods. Thus, Symber adds a constraint that represents the relationship between the joined/started thread and the parent thread. For instance, if thread i executes O_{r_x}, and then, creates a new thread j by calling the method *start*, a new constraint is generated as:

$$O_{r_x} < O_{w_y}$$

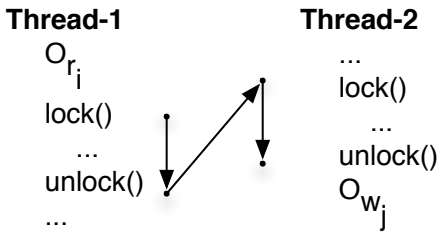

Figure 3.14: Synchronization constraints optimization.

where O_{w_y} represents the first shared read/write operation executed by the thread j. On the other hand, regarding the join method, if thread i calls method $join$ on thread j, a new constraint is generated as follows:

$O_{w_y} < O_{r_x}$

where O_{w_y} represents the last shared read/write operation executed by the thread j, and O_{r_x} represents the first shared read/write operation performed by the thread i after the thread j joined.

Furthermore, an optimization has been applied in order to restrict the search space even more. Since the symbolic execution is guided by the locking order of the buggy execution, Symber can still infer more synchronization points between threads. For two consecutive lock acquisitions performed by different threads on the same locking object, Symber adds a constraint to the formula specifying that the shared read/write operation executed immediately before the first acquisition has to be executed before the first shared read/write operation performed after the second release of the lock. Figure 3.14 shows a simple example that helps to understand the optimization. In the example, $Thread-1$ executes the shared read operation O_{r_i} before acquiring the lock, and $Thread-2$ performs a shared write operation O_{w_j} after realising the lock. The arrows depict the order in which the locks are acquired. Since $Thread-1$ acquires the lock before $Thread-2$, Symber can add the following constraint to the formula:

$O_{r_i} < O_{w_j}$

3.7 Implementation

The implementation of Symber has been a long and challenging part of the work required to prepare the thesis. As it is mentioned in the introduction of this chapter, Symber is the first record and replay tool for Java that includes an inference phase based on symbolic execution as part of the process.

Symber is completely implemented in Java. Soot (Vallée-Rai, Co, Gagnon, Hendren, Lam, & Sundaresan 1999) tool has been used for the static analysis and the instrumentation of the target application. On the other hand, we have used the open source project Java PathFinder (Visser, Havelund, Brat, &

Park 2000) for implementing Symber's symbolic executor. This executor not only uses jpf-core, which is the basic package, but it also uses several extensions such as jpf-symbc (Păsăreanu & Rungta 2010), which is envisioned for symbolic execution, and jpf-concurrent (Ujma & Shafiei 2012) that let Java PathFinder inspect and understand java.util.concurrent library. We decided to use jpf-concurrent since we consider that the use of java.util.concurrent is generalized among Java users.

Regarding constraint solvers, we have implemented Symber thinking of the constraint solver as a pluggable component. Therefore, it is quite straight forward to substitute it by a different one. We mainly experiment with Choco2 (Jussien, Rochart, Lorca, et al. 2008). Nevertheless, we are thinking of extend Symber to support MiniZinc (Nethercote, Stuckey, Becket, Brand, Duck, & Tack 2007) and Z3 (De Moura & Bjørner 2008) as constraint solvers.

3.7.1 Challenges

We had to overcome many challenges during the implementation phase. Before starting the implementation, given that it was impossible and non-productive to implement the entire system from scratch, a significant amount of time and effort was spent in searching and learning tools that could be used as potential building blocks. Working with open source projects is difficult and it requires a previous deep study of their implementation. During our implementation we also found some bugs in some of the open source project we were using (for instance in jpf-concurrent). We solved them and, thus, we have also been able to contribute to a community that had helped us many times. In any case, without those projects and the help of the open-source community, the implementation would have taken much more time. In the following paragraphs, we detail how we solved some of the most significant challenges we found during the implementation.

Thread consistent identification Different execution of the same multithreaded application can identify threads in a different way. Since Symber executes the same application several times along all the phases, we need to remove that non-determinism from our tool. Therefore, all our components implement the same mechanism for identifying the threads.

The main thread is assigned to the identifier "1". When a new thread is created, its identifier is created by the concatenation of his parent id, ":" and a children counter that gets incremented every time a new thread is created. The children counter is local for each thread.

Figure 3.15 shows an example of the application of this mechanism.

Thread termination This problem is related to the amount of information that is captured during the recording phase. Since we need to be as efficient as possible during the recording, we do not log any

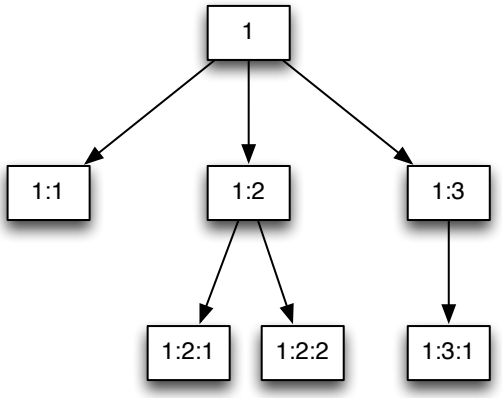

Figure 3.15: Example: Thread consistent identification.

extra information apart form the path trace and the locking order trace. Therefore, we do not exactly know which is the last instruction executed by each thread.

We solved this problem in different ways for the different phases. In order to circumvent this problem during the inference phase, if during the symbolic execution, the thread is about to execute an *if* statement or a synchronization operation and the traces for any of those statements have been already consumed, the thread is immediately killed.

On the other hand, during the replay phase, we apply a similar approach. In this case, when a thread is about to execute a read/write operation or a synchronization operation that has not been traced, the thread is killed.

We are aware that this is not the perfect solution since some extra instructions might be executed; nevertheless, we consider that those extra instruction do not modify the accuracy of our tool.

Shared variable identification Other record and replay tools, such as LEAP (Huang, Liu, & Zhang 2010), opt for a field identification of shared variables. Thus, different instances of the same class are treated as one. This increments the recording overhead (for those tools that also record read/write operations order) and it is not compatible with our inference tool since some of the constraints in the path condition might not be compatible with the read/write constraints. We actually experiment some cases in which the constraint solver was not able to find a solution because of this faulty identification.

In consequence, we decided to identify shared variables by the instance. This makes our constraint solver to generate less constraints and to correctly work.

Early termination In some cases, the symbolic execution can experiment an early termination. Since the order in which threads interleave is not traced for non-synchronized parts of the code, it might happen that an instruction that ends the execution is performed in the symbolic execution before than it was

CLAP	Symber
It only records the execution path	It records the execution path and the locking order
Its symbolic execution is only guided by the path trace	Its symbolic execution is guided by the path trace and the locking order
Its constraint model contains synchronization order constraints that increments the complexity of the constraints	It is capable of discard the synchronization order constraints; therefore, the final formula is easier to solve
Most of the solutions proposed by the constraint might represent an infeasible execution	Reduced number of infeasible thread schedule candidates. It simplifies the solution generation and the solution checking phases.

Table 3.1: Main differences between Symber and CLAP

executed during the buggy execution. Thus, part of the code of the remaining threads is never executed, and, in consequence, the trace generated by the symbolic execution would no be consistent with the replayer.

Examples of termination instructions are $System.exit()$ and uncatched exceptions. Thus, Symber has to avoid executing those kind of instructions until it is safe. For this purpose, when a thread is about to execute a termination instruction, it firstly checks whether there are remaining threads alive. If any thread is still alive, Symber set the current thread as "terminated" but it does not execute the termination instruction. Therefore, the remaining threads can continue the execution. In consequence, the last thread alive has to end the execution once both the path trace and the locking trace are completely consumed.

Summary

In this chapter we have presented Symber in detail. Symber is a novel tool for replaying concurrency bugs that combines logging and inference to achieve a good tradeoff among the record overhead and the time required to replay the execution. Symber is inspired by CLAP but incorporates several interesting changes. Table 3.1 captures the main differences between CLAP and Symber.

In the next chapter we provide an experimental evaluation of Symber and discuss its performance.

4 Evaluation

In this chapter we present an experimental evaluation of the Symber system. We start by presenting the evaluation methodology and then, we present the results for a number of experiments that evaluate different aspects of the system. The chapter is concluded with a discussion of the results.

4.1 Evaluation Methodology

Our evaluation addresses the following three complementary aspects of the tool:

- The overhead imposed by Symber, namely the recording overhead and the space overhead (trace files size). The first aspect captures the overhead introduced by the tracing algorithms. The second one measures the size of the generated traces for each of the approaches.
- The Symber capacity for reproducing bugs. This criterion mainly tells us whether Symber is able to reproduce the concurrency bug in the tested applications.
- The number of solutions that are checked by the inference tool until the buggy execution is found. Finally, this criterion captures how fast Symber finds the buggy execution. Thus, the fewer checked solutions, the better the tool performs in terms of inference efficiency.

In order to have a comparative assessment of how well Symber performs with regard to other tools, we have also implemented a non-optimized version of Ditto and a simplified version of CLAP:

- We were forced to implement a version of Ditto since the tool was not available. Our Ditto-like tool is implemented at the application level while the original implementation of Ditto is done at JVM level. Therefore, results reported by our Ditto-like tool are expected to be worse than the original tool. Nevertheless, the results still allows us to assess how Symber compares to one full-recording tool.
- On the other hand, in order to confirm that Symber reduces the complexity of the inference phase (and, in consequence, speeds it up), it is necessary to compare Symber with CLAP. However, CLAP is a tool aimed at C/C++ programs and Symber is aimed at Java applications. Thus, we created a simplified version of CLAP for Java applications.

We have evaluated Symber using a combination of micro-benchmarks and third-party benchmarks. All experiments were run in a machine with an Intel Core 2 Duo at 2.26 Ghz, with 4 GB of RAM, and running Mac OS X.

4.2 Time and Spatial Overhead

The experiments presented in the section have two main goals:

- Demonstrate that Symber's recording overhead is slightly larger than CLAP's recording overhead but still competitive in comparison to the baseline.
- Show that Symber still introduces substantially smaller overhead in comparison to full-recording tools.

4.2.1 Microbenchmarks

We have used a number of microbenchmarks for intensively analyse and compare Symber. We have mainly run six different experiments. The variables of our experiments are the following:

- Number of threads
- Number of shared read/write operations
- Number of synchronization operations
- Number of shared variables
- Number of branches
- Percentage of shared read/write operations that are synchronized

In our experiments, we have varied one parameter and kept constant the rest of them. Thus, we can appreciate how the isolated variation of one of the parameters affects both the time and the spatial overhead of the recording phase. Using this methodology, we have defined six different microbenchmarks, depicted in Table 4.1.

Using these benchmarks we have measured the performance of: i) the baseline time (without tracing any information); ii) a Ditto-like implementation that traces the order of synchronization operations and shared read/write operations; iii) CLAP that only traces the execution path, and; iv) Symber that traces the execution path and the order of synchronization operations (the locking order). Furthermore, as it

Benchmark	#Threads	#Branches	#Accesses	#Shared Variables	% Sync. Accesses
Microbenchmark 1	*variable*	10^7	10^7	4	50%
Microbenchmark 2	4	10^7	*variable*	4	50%
Microbenchmark 3	4	10^7	10^7	4	0%
Microbenchmark 4	4	10^7	10^7	*variable*	0%
Microbenchmark 5	4	*variable*	10^7	4	0%
Microbenchmark 6	4	10^7	10^7	4	*variable*

Table 4.1: Microbenchmarks

is mentioned in the previous chapter, we have also implemented in Symber an optimization for reducing the size of the traces. Thus, in the plots that show a comparison between traces size, we include both the optimized version of Symber and the non-optimized version. For all microbenchmarks, we present a figure that is composed by two plots: the plot on the left shows the execution time of the microbenchmark while the plot on the right presents the size of the generated traces for each of the tools.

We now describe the results of each microbenchmark in detail.

Number of threads In this experiment we assess how Symber scales as the number of threads increases. Figure 4.1 shows the time and spatial overhead obtained with Microbenchmark 1. As one can see, both CLAP and Symber execution times are close to the baseline. We can conclude that for both tools the number of threads does not affect the execution time. On the other hand, for the Ditto-like application, apart from incurring a considerable larger overhead in comparison to the other approaches, the more threads the application runs, the greater the time overhead is (from 34 seconds to 40 seconds). This is due to the synchronization of shared read/write operations produced by the recording algorithm.

In terms of spatial overhead, we can appreciate that the number of threads does not significantly vary the size of the traces. It is important to highlight the significant difference between the optimized version of Symber (431KB in the worst case) and the Ditto-like tool (more than 39 MB in the worst case).

Number of shared accesses This experiment, using Microbenchmark 2, checks how Symber behaves, in comparison to the other approaches, when the number of shared read/write operations increases. Since we keep the percentage of synchronized accesses as the number accesses is increased, the number of synchronization operations is also increased.

Figure 4.2 shows the time and spatial overhead for this experiment. Regarding recording overhead, as expected, the increment of accesses does no affect CLAP that maintains the same distance to the baseline along the experiment (7s). Still, since the number of synchronization operations also increases, Symber slightly augments the distant to both CLAP and the baseline. Finally, the Ditto-like tool is

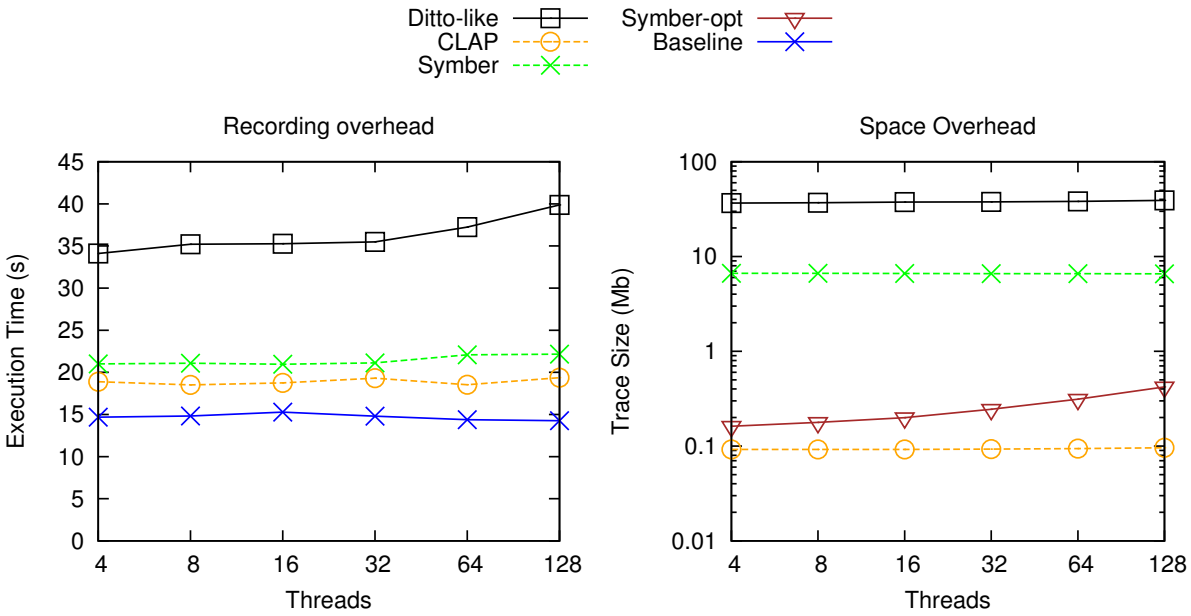

Figure 4.1: Results for Microbenchmark 1.

heavily affected by both the increment of accesses and the increase of synchronization operations. It starts introducing an overhead of 135% and it finishes introducing an overhead of 165%.

When considering the spatial overhead, we can observe that the Ditto-like tool, and the two versions of Symber, increase the size of the traces as the number of accesses, and in consequence the number of synchronization operations, increases. However, for the optimized version of Symber, the size of the trace is still rather small when 30 million of accesses are performed (259KB).

Number of synchronization operations This experiment assesses how Symber scales as the number of synchronization operations increases. Figure 4.3 shows the time and spatial overhead with Microbenchmark 3. Regarding the recording overhead, both CLAP and Symber start almost from the same point. This is due to the small amount of synchronization operations (only 1000). As the number of synchronization operations increases, the gap between Symber and CLAP executions increases. However, the distance between them holds quite small. This supports our hypothesis. On the other hand, the Ditto-like tool execution time grows exponentially as the number of synchronization operations is increased.

In terms of spatial overhead, the figure shows that both versions of Symber are affected by the increment of synchronization operations. Nevertheless, the optimized version of Symber maintains the size of the trace below 257KB.

This results confirm that even for large amounts of synchronization operations, Symber is still competitive in comparison to CLAP.

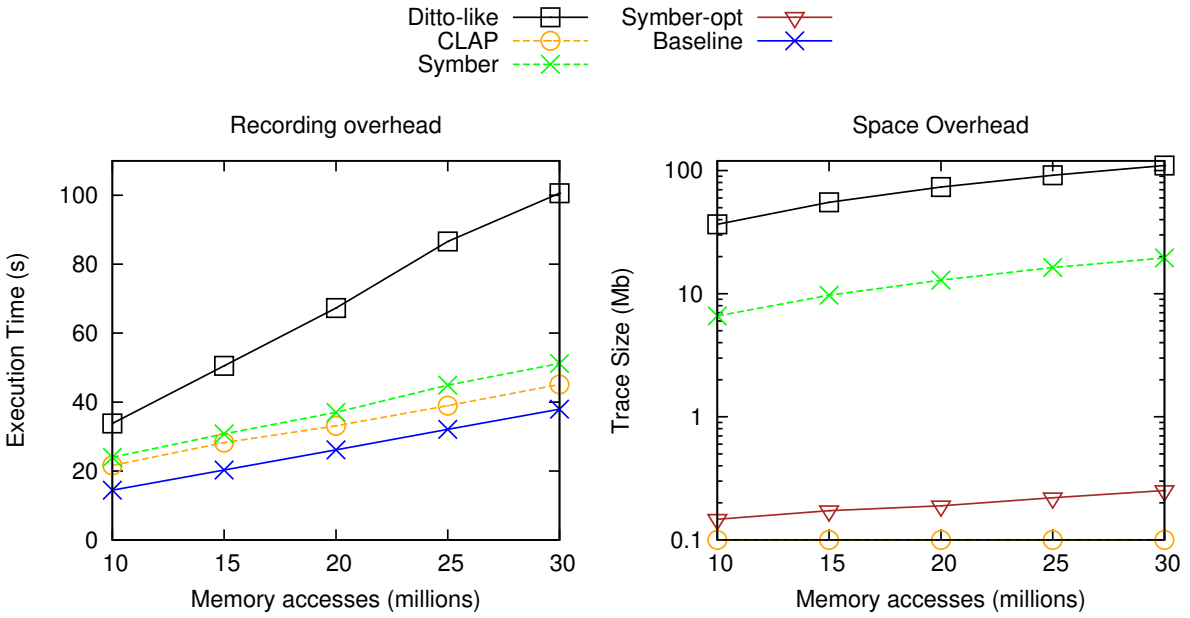

Figure 4.2: Results for Microbenchmark 2.

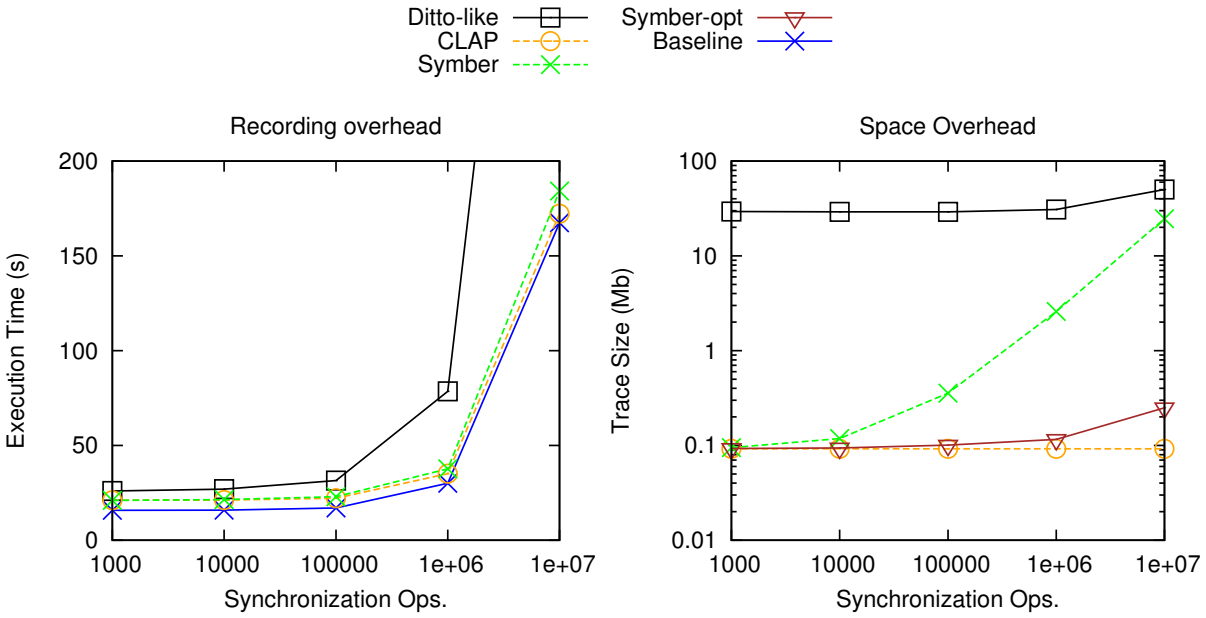

Figure 4.3: Results for Microbenchmark 3.

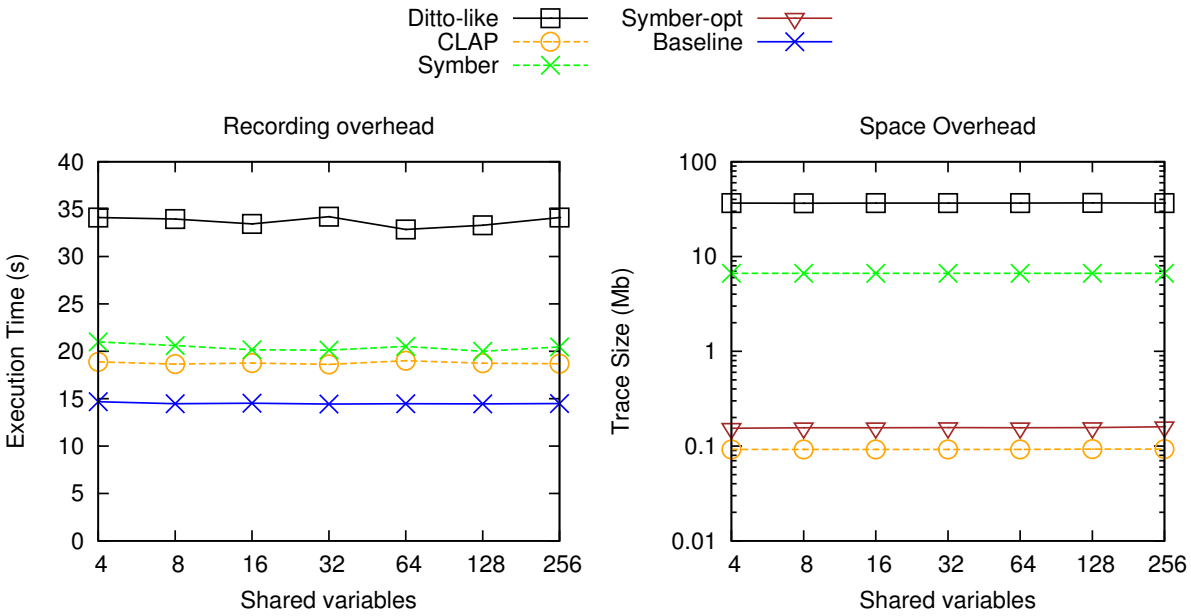

Figure 4.4: Results for Microbenchmark 4.

Number of shared variables This experiment checks how the number of shared variables affects the performance of Symber. Figure 4.4 shows the time and spatial overhead using Microbenchmark 4. As it is expected, neither Symber nor CLAP are affected by the increment of shared variables. On the other hand, we were expecting an improvement in the performance for Ditto-like application since read/write operations are made sequential by locking the shared variable. Thus, we were expecting that the more shared variables the lower the overhead introduced by Ditto-like application. Nevertheless, the results show that the number of shared variables is not a determinant factor. Note that the work of (Silva) confirms this result. Regarding the spatial overhead, the optimized version of Symber achieves a pretty small trace size (lower than 164KB). As the figure shows, the number of shared variables does not affect the size of any trace. This is also an expected result.

Number of branches This experiment measures how the number of branches affects Symber. We have decided to keep the number of accesses constant and increase the number of branches. This is also useful to visualize when a full-recording tool and a tool based on tracing the execution path (Symber and CLAP) might converge. Figure 4.5 shows the time and spatial overhead using Microbenchmark 5. As expected, when the number of branches is zero, Symber and CLAP are pretty close to the baseline (actually CLAP does not incur any overhead). As the number of branches increases, both CLAP and Symber become separated from the baseline. Since the number of synchronization operation is constant ($512 * 10^4$), the distance between CLAP and Symber is also steady. Furthermore, we can notice that when the number of branches is 25 millions, the overhead of tracing the branches almost compensate the overhead introduced by the Ditto-like tool. Regarding the spatial overhead, the optimized version of

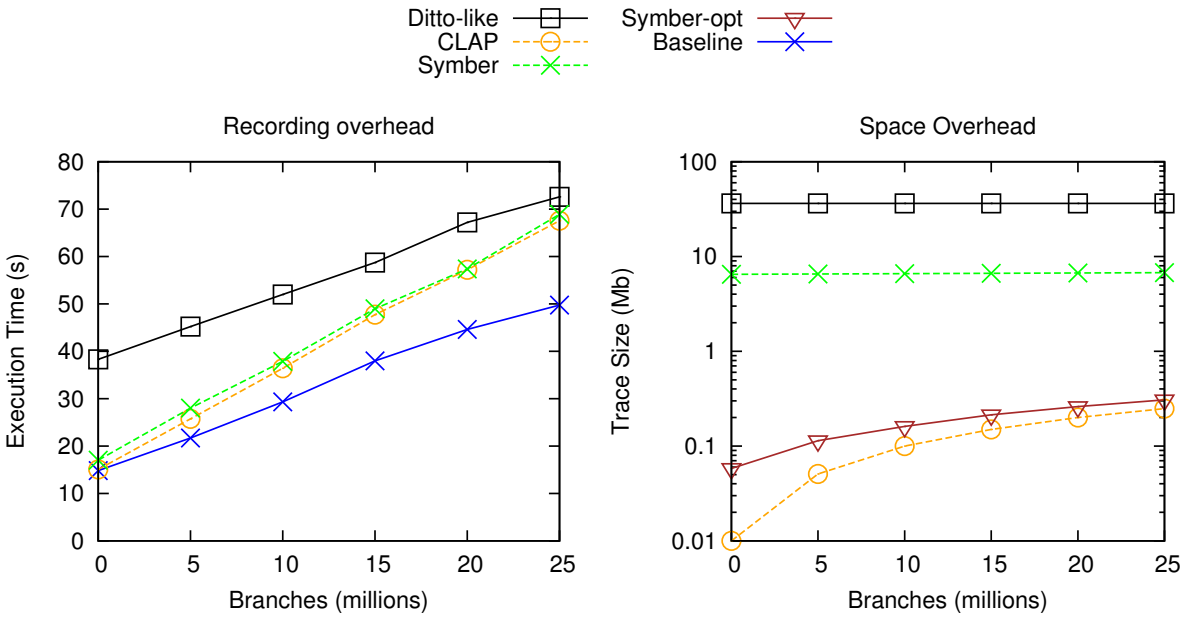

Figure 4.5: Results for Microbenchmark 5.

Symber still generates traces many times smaller than a full-recording tool.

Synchronized accesses vs non-synchronized accesses This experiment tests how the approaches behaves when the ratio between synchronized and non-synchronized shared read/write operations varies. Regarding the runtime overhead, we expect Symber to converge to CLAP when there are no synchronized accesses. Figure 4.6 shows the time and spatial overhead using Microbenchmark 6. As we were expecting, CLAP and Symber converge when there are no synchronized accesses. The maximum difference between them is produced when all the accesses are synchronized since Symber has more synchronization operations to trace. On the other hand, the full-recording tool seems to be heavily affected by the ratio.

When considering the spatial overhead, both versions of Symber converge with CLAP when there are no synchronized accesses, since there are no synchronization operations to trace. As in every previous experiment, traces for CLAP and the optimized version of Symber hold rather small.

4.2.2 Third-party Benchmarks

In order to test Symber with more realistic applications, third-party benchmarks have been used. The IBM ConTest benchmark suite (Farchi, Nir, & Ur 2003) is composed by multiple applications that contain concurrency bugs. Table 4.2 shows a brief description of the used applications in terms of lines of code (LOC), number of threads (#Threads), number of shared variables (#sv), number of branches (#Br) and the bug-pattern.

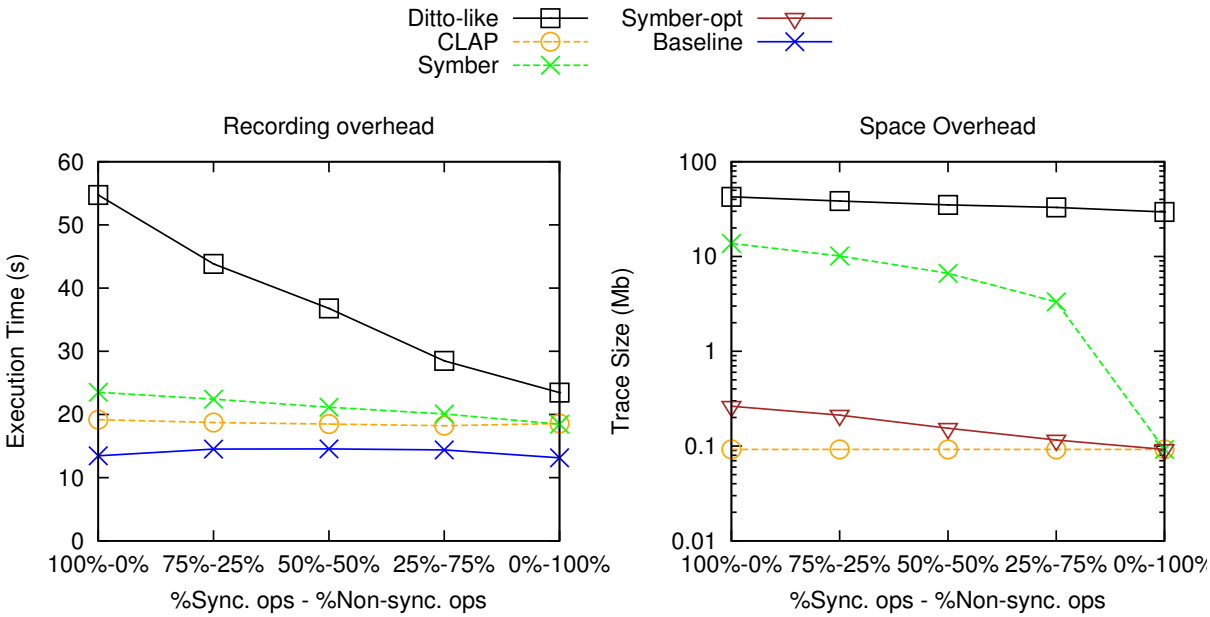

Figure 4.6: Results for Microbenchmark 5.

Program	**LOC**	**#Threads**	**#sv**	**#Br**	**Bug**
TwoStage	136	16	3	164	Two-stage
Piper	165	21	4	160	Missing condition for Wait

Table 4.2: Description of the IBM ConTest benchmark applications used in the experiments

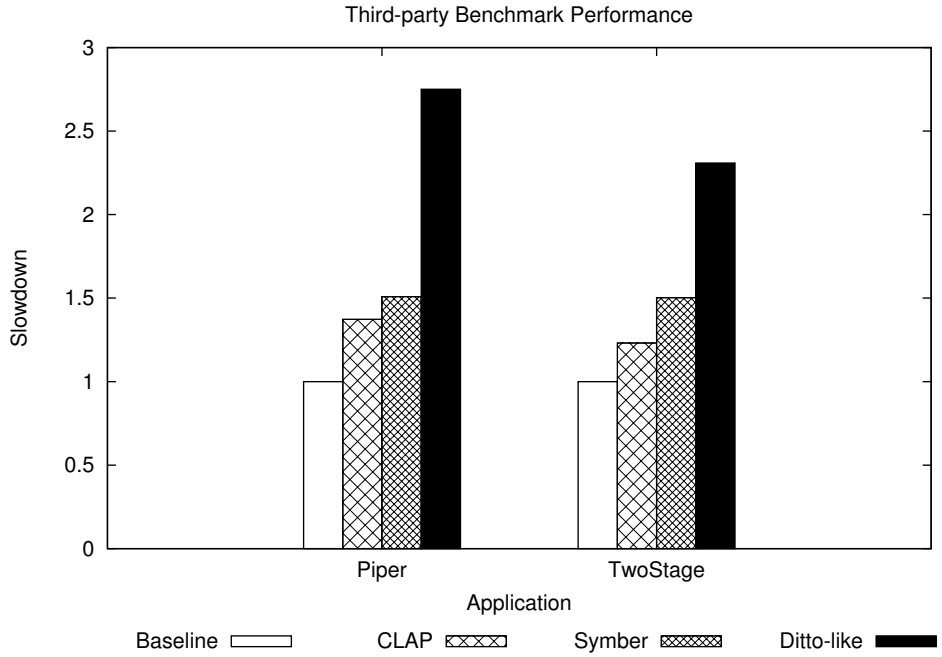

Figure 4.7: Third-party benchmarks. Performance slowdown.

Program	Baseline	Ditto-like	CLAP	Symber
TwoStage	16.5868ms	38.293ms (131%)	20.4114ms (23%)	24.9112ms (50%)
Piper	17.8668ms	49.1391ms (175%)	24.5357ms (37%)	26.9579ms (51%)

Table 4.3: Runtime overhead comparison between Ditto-like tool, CLAP and Symber

Since our microbenchmarks have already demonstrate that the size of the traces generated by the optimized version of Symber are small, we now focus on evaluating the runtime overhead. Figure 4.7 shows the slowdown (times slower) introduced by the approaches. In the figure, the baseline, CLAP, Symber, and a full-recording tool are compared. Thus, as expected, the full-recording tool introduces a large slowdown (almost 3x for Piper application). On the other hand, CLAP and Symber maintain an effective overhead where it never goes beyond the 51%.

Table 4.3 reports the results. All data were average over ten runs. The reduction of the overhead produced by Symber in comparison to the full-recording tool is significant. Furthermore, Symber still maintains a competitive overhead in comparison to CLAP.

Figure 4.8 presents the slowdown introduced by Symber in detail. Thus, for each benchmark, the slowdown has been divided into three boxes. The black box represents the naive application, the white box represents the slowdown introduced by tracing the locking order and the grey box represents the slowdown introduced by the path profiler. As the figure shows, Symber incurs a similar overhead for all applications. This totally depends on the application as the michrobenchmarks suggest. Nevertheless, as

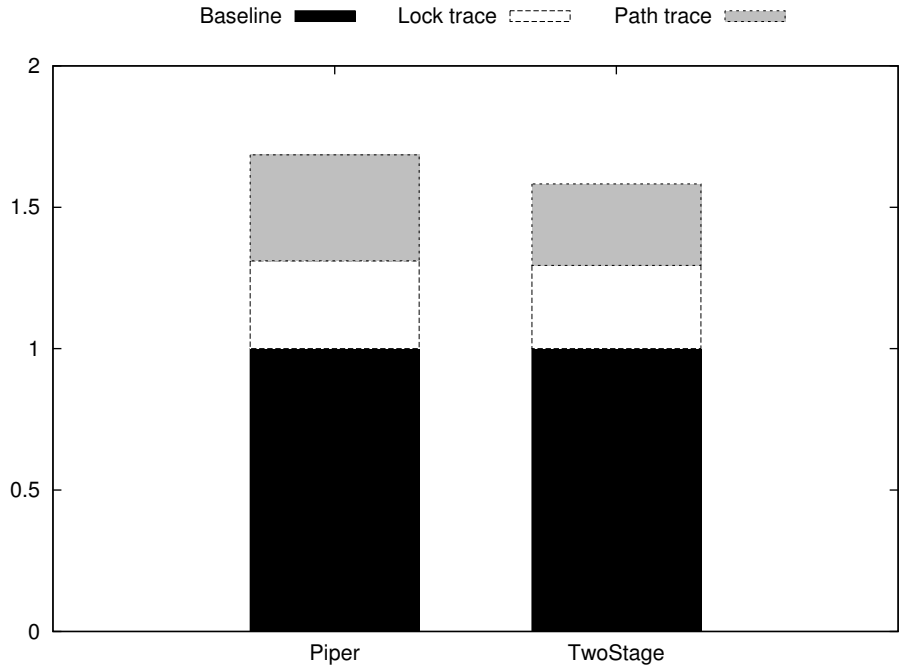

Figure 4.8: Third-party benchmarks. Symber detailed slowdown.

the Figure 4.8 shows, the slowdown introduced by both tracing mechanisms tends to be similar. None of the tracing mechanisms need extra synchronization; therefore, their use is always more efficient that tracing the order of shared read/write operations.

4.3 Capacity of Reproducing Bugs

Apart from measuring both the time and the spatial overhead that Symber introduces during the record phase, we also have to evaluate Symber's capacity for reproducing bugs. Thus, we have decided to use the third-party benchmarks used in the previous section to test Symber.

Table 4.4 presents the results obtained by the bug-reproducibility experiment over IBMContest applications. Columns 2-4 report the number of instructions (#Ins) executed during the symbolic execution, the number of shared accesses (#SAs) seen by the symbolic executor, and the number of constraints (#Cons.) generated by the constraint generator component. Columns 5-8 report the number of variables (#Vars) created by the symbolic execution, the total number of variables (#VSolver) seen by the constraint solver, the number of solutions generated (#cs) until the buggy execution is found, and whether Symber was able to reproduce the concurrency bug or not.

As one can see, even for medium-size applications, the number of constraints generated by the constraint generator might be pretty large. For instance, the TwoStage program, with only 136 LOC and 274 shared read/write operations, generates 1,113,117 constraints. On the other hand, Symber is

Program	#Ins	#SAs	#Cons.	#Vars	#VSolver	#cs	success?
TwoStage	13891	274	1113117	455	1113282	1	**yes**
Piper	15211	264	37617	459	37793	1	**yes**

Table 4.4: Overall results of Symber bug-reproducibility experiment

very efficient finding the buggy solution. E. g. Symber was capable to find the buggy execution for both TwoStage and Piper applications after only checking one solution. We believe that the optimization presented in the previous chapter, in which Symber is able to infer new synchronization points due to the order in which locks are acquired, permits Symber to be very efficient because the number of infeasible solutions is dramatically reduced.

4.4 Efficiency of the Inference Mechanism

Finally, we have also compared Symber with CLAP in terms of number of constraints and number of variables generated. Table 4.5 presents the results. As it is expected, Symber dramatically reduces the number of both constraints and variables seen by the constraint solver.

	#Variables			#Constraints		
Program	**CLAP**	**Symber**	**Reduction**	**CLAP**	**Symber**	**Reduction**
TwoStage	5407122	1113282	↓71.4%	5406748	1113117	↓71.4%
Piper	580609	37793	↓93.5%	579686	37617	↓93.5%

Table 4.5: #Constraints and #Variables comparison between CLAP and Symber

4.5 Discussion

Evaluating our system in comparison to CLAP, the baseline and a full-recording tool (Ditto-like) serves us to confirm that a full-recording tool introduces an unbearable runtime overhead in many cases. Furthermore, the size of the traces might be problematic in applications with larger number of shared accesses and synchronization operations. However, in our experiments, traces never reach unmanageable sizes. This data supports the use of an inference phase.

On the other hand, our results confirm that locally tracing the execution path introduces a bearable runtime overhead. Furthermore, the size of the traces generated by the path profiler are almost insignificant. For instance, in our branches test, when the number of branches is 25 millions, the path profiler still generates a negligible trace of 255KB.

More importantly, the results confirm that tracing the locking order is a cheap operation. Every experiments has denoted that the overhead introduced by tracing the synchronization operations is rather low. E.g. in the worst case scenario when the number of synchronization operations is 10^7, the overhead introduced by Symber in comparison to CLAP is lower than 7%. This confirmation is important since Symber is mainly based on this observation. Furthermore, real applications tend to have a greater number of branches than synchronization operations; therefore, we consider that tracing the path is a more expensive mechanism in terms of time overhead than tracing the locking order.

When using the third-party benchmarks, Symber has proved its ability to replay concurrency bugs very efficiently. Apart from the experiments summarized in Table 4.4, we have also tested Symber with our own applications. Nevertheless, we consider that additional evaluation effort needs to be done, in order to fully analyse Symber's capability for replaying concurrency bugs. Unfortunately, due to the tight calendar, we had no time to extend the evaluation to some real applications.

Regarding the comparison between Symber and CLAP, results are promising, since a very significant reduction in terms of number of constraints and number of variables is achieved. As it is mentioned in Section 3, Symber's constraint model is inspired by CLAP's constraint model. Thus, having fewer constraints and variables directly means that the solver search space is smaller. This observation is explained by the following facts:

- Symber constraints are a subset of CLAP constraints plus synchronization order constraints.
- Synchronization order constraints are aimed at pruning the read/write constraints. Thus, the search space is reduced.
- All variables in Symber formula are interrelated since all of them are directly related to the order in which the shared read/write operations occur. On the other hand, CLAP introduces sets of constraints that are not related to each other such as the locking constraints and the partial order constraints.
- Symber already knows the order in which locks are acquired (because of the locking order trace). In contrast, CLAP has to infer it. Thus, the number solutions that CLAP have to check might be dramatically increased.

Reducing the search space implies that the worst case scenario (when all possible solutions have to be checked in order to find the buggy execution) is also reduced. Still, we would like to experiment with Symber and CLAP in order to confirm these observations. Unfortunately, we did not have sufficient time for implementing the constraint solving algorithm proposed by CLAP (which is not trivial). Thus, we could not experimentally confirm this observation.

```
...
try {
  ...
  queue.removeFirstElement();
  ...
} catch (Exception e){
  [some code]
}
...
```

Figure 4.9: Example of exceptions.

Finally, it is important to emphasize that Symber is still a research prototype that, currently, has the following limitations:

- Exceptions affect the execution path. Thus, simply tracing the output of if and $switch$ statements is not enough. For instance, Figure 4.9 tries to illustrate this limitation. In the example, *queue* is a shared variable; therefore, a symbolic reference is created when it is read. Then, $queue.removeFirstElement()$ will never throw an exception since the method is never actually executed (because is a symbolic symbol). In consequence, the symbolic execution does not enter into the catch section and part of code, that might have been executed in the original execution, is not executed. This behaviour might produce inconsistency between the trace generated by the inference tool and Symber's replayer.

- The constraint solving phase, with the current system implementation, may take a very large amount of time for medium-large applications. All experiments have been performed using Choco2 (Jussien, Rochart, Lorca, et al. 2008) as constraint solver. Based on the exhibited performance, we believe that Symber should switch to MiniZinc (Nethercote, Stuckey, Becket, Brand, Duck, & Tack 2007) or Z3 (De Moura & Bjørner 2008).

- Although the experiments show good recording overhead when the number of branches is large, we still believe that tracing every $if/switch$ statement is considerable expensive. A more efficient tracing algorithm might be used such as the one proposed in (Ball & Larus 1996).

We expect to overcome most of these limitations in the near future.

Summary

This chapter has presented an experimental evaluation of Symber. The results indicate that Symber provides an interesting tradeoff between the runtime overhead, and the capacity and time to reproduce bugs. In the next chapter we identify some ideas that are interesting to exploit as future work.

5 Conclusions

5.1 Conclusions

Reproducing concurrency bugs is an expensive and difficult task due to the non-determinism introduced by the communication among threads. Record/replay techniques have been designed to circumvent these challenges. These techniques make the execution deterministic by tracing all the sources of non-determinism in an execution. Unfortunately, it has been demonstrated that classical record/replay techniques are expensive in terms of runtime overhead.

This thesis has presented Symber, a record/technique that proposes a cheaper record phase in order to reduce the runtime overhead. Symber reduces the recording overhead by not tracing all non-deterministic events. Thus, a new phase is introduced in order to infer the missing information. A Symber execution is composed by three phases: the record phase, the inference phase and the replay phase. The record phase traces the local execution path and the order in which locks are acquired. The inference phase infers the missing information needed for faithfully replaying the buggy execution. This phase is based on a symbolic execution guided by both the local execution path and the locking order. Once the symbolic execution has finished, a formula that represents the buggy program is generated and fed to a constraint solver. The solutions proposed by the solver are translated into a trace that represent how shared read/write operations interleave. When the buggy execution is found, the inference phase finishes and the trace that represents the buggy execution is used by the replayer to reproduce the concurrency bug.

Although there are other approaches that use symbolic execution in its inference phase ((Altekar & Stoica 2009), (Altekar & Stoica 2010), and (Huang, Zhang, & Dolby 2013)), we believe that Symber provides a new and interesting tradeoff among the following factors: recording overhead, efficiency of the inference phase, and the information that the tool gives to the developers.

This thesis has also presented an exhaustive evaluation of Symber. We have used a combination of microbenchmarks and the IBM Contest benchmark suite for our experiments. The results have shown that tracing the locking order is not an expensive mechanism. Furthermore, we have evaluated Symber's ability to reproduce concurrency bugs. The results have shown that Symber efficiently reproduce the bugs.

We have also compared Symber and CLAP in terms of number of constraints and number of variables

generated by the inference phase. Symber heavily reduces the number of both. We consider that both the number of constraints and the number of variables directly affect the efficiency of the inference phase.

5.2 Future Work

Although this thesis has presented an intensive evaluation of Symber, we plan to improve it further with real applications. Microbenchmarks and third-party benchmark suits are sufficient for testing the performance of the tool, but we consider than in order to fully analyse the correctness of Symber, more application should be tested.

Furthermore, as Section 4.5 suggests, we need to experimentally compare Symber to CLAP. For this purpose, a full CLAP tool for Java applications has to be implemented. Our simplified version misses the constraint solving algorithm and; therefore, we cannot directly compare the efficiency of both approaches.

Apart from completing the evaluation, in order to overcome the limitations of our approach, Symber also needs to extend its usability to Java applications in which exceptions can take part of the execution path. For this purpose, more information needs to be traced during the record phase. E. g. Symber could identify which invocations to methods (when the callee is a shared object) might throw an exception and count the number of times that the invocation is performed before the exception is thrown. Thus, the behaviour might be reproduced during the symbolic execution and the trace generated by the trace generator would be consistent with the replayer.

On the other hand, we have realized that Choco2 (the constraint solver used by Symber) performs poorly for the kind of constraints that Symber generates. Thus, it would be interesting to compare different constraint solvers. We are considering to add MiniZinc and Z3 constraint solvers to Symber. Furthermore, the inference phase might be speeded up. The current version of Symber sequentially checks the solutions generated by the constraint solver until the buggy execution is found. This process can be clearly parallelized by distributing the solutions to different machines (or processes) and individually checking whether the solutions reproduce the concurrency bug or not.

Finally, we still believe that tracing the local execution path is an expensive mechanism. We are convinced that probabilistic record/replay can be combined to Symber in order to reduce the recording overhead.

References

Altekar, G. & I. Stoica (2009). ODR: output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP'09, Big Sky, Montana, USA, pp. 193–206. ACM.

Altekar, G. & I. Stoica (2010, March). DCR: Replay-Debugging for the Datacenter. Technical Report UCB/EECS-2010-33, EECS Department, University of California, Berkeley.

Ball, T. & J. R. Larus (1996). Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 29, Paris, France, pp. 46–57. IEEE Computer Society.

Cadar, C. & K. Sen (2013, February). Symbolic execution for software testing: three decades later. *Commun. ACM 56*(2), 82–90.

Chang, F., J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, & R. E. Gruber (2008, June). Bigtable: a distributed storage system for structured data. *ACM Trans. Comput. Syst. 26*(2), 4:1–4:26.

Choi, J.-D. & H. Srinivasan (1998). Deterministic replay of Java multithreaded applications. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, SPDT '98, Welches, Oregon, USA, pp. 48–59. ACM.

De Moura, L. & N. Bjørner (2008). Z3: an efficient SMT solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pp. 337–340. Budapest, Hungary: Springer-Verlag.

Dean, J. & S. Ghemawat (2008, January). MapReduce: simplified data processing on large clusters. *Commun. ACM 51*(1), 107–113.

Elnozahy, E. N. M., L. Alvisi, Y.-M. Wang, & D. B. Johnson (2002, September). A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv. 34*(3), 375–408.

Erik-Svensson, C., D. Kesler, R. Kumar, & G. Pokam (2009). MPreplay: architecture support for deterministic replay of message passing programs on message passing many-core processors. Technical Report UILU-09-2209, University of Illinois.

Farchi, E., Y. Nir, & 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, Nice, France.

IEEE Computer Society.

Floyd, R. (1993). Assigning meanings to programs. In T. Colburn, J. Fetzer, & T. Rankin (Eds.), *Program Verification*, Volume 14 of *Studies in Cognitive Systems*, pp. 65–81. Springer Netherlands.

Geels, D., G. Altekar, S. Shenker, & I. Stoica (2006). Replay debugging for distributed applications. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, ATEC '06, Boston, Massachusetts, USA, pp. 27–27. USENIX Association.

Georges, A., M. Christiaens, M. Ronsse, & K. De Bosschere (2004, May). JaRec: a portable record/replay environment for multi-threaded java applications. *Softw. Pract. Exper. 34*(6), 523–547.

Ghemawat, S., H. Gobioff, & S.-T. Leung (2003). The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, Bolton Landing, New York, USA, pp. 29–43. ACM.

Gioachin, F., G. Zheng, & L. V. Kalé (2010, July). Robust Record-Replay with Processor Extraction. In *PADTAD '10: Proceedings of the 8th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, Trento, Italy, pp. 9–19. ACM.

Huang, J., P. Liu, & 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*, FSE '10, Santa Fe, New Mexico, USA, pp. 207–216. ACM.

Huang, J., C. Zhang, & J. Dolby (2013). CLAP: recording local executions to reproduce concurrency failures. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, PLDI '13, Seattle, Washington, USA, pp. 141–152. ACM.

Jussien, N., G. Rochart, X. Lorca, et al. (2008). Choco: an open source java constraint programming library. In *CPAIOR'08 Workshop on Open-Source Software for Integer and Constraint Programming (OSSICP'08)*, Paris, France, pp. 1–10.

Kergommeaux, J., M. Ronsse, & K. Bosschere (1999). MPL: Efficient Record/Replay of nondeterministic features of message passing libraries. In J. Dongarra, E. Luque, & T. Margalef (Eds.), *Recent advances in parallel virtual machine and message passing interface*, Volume 1697 of *Lecture Notes in Computer Science*, pp. 141–148. Springer Berlin Heidelberg.

King, J. C. (1976, July). Symbolic execution and program testing. *Commun. ACM 19*(7), 385–394.

Konuru, R., H. Srinivasan, & J.-D. Choi (2000). Deterministic replay of distributed Java applications. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, Cancun, Mexico, pp. 219–227.

LeBlanc, T. & J. Mellor-Crummey (1987). Debugging parallel programs with Instant Replay. *Computers, IEEE Transactions on C-36*(4), 471–482.

Lee, K. H., N. Sumner, X. Zhang, & P. Eugster (2011). Unified debugging of distributed systems with Recon. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, DSN '11, Hong Kong, pp. 85–96. IEEE Computer Society.

Lu, S., S. Park, E. Seo, & Y. Zhou (2008). Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, Seattle, Washington, USA, pp. 329–339. ACM.

Nethercote, N., P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, & G. Tack (2007). MiniZinc: towards a standard CP modelling language. In *Proceedings of the 13th international conference on Principles and practice of constraint programming*, CP'07, pp. 529–543. Providence, RI, USA: Springer-Verlag.

Netzer, R., S. Subramanian, & J. Xu (1994). Critical-path-based message logging for incremental replay of message-passing programs. In *Distributed Computing Systems, 1994., Proceedings of the 14th International Conference on*, Pozman, Poland, pp. 404–413.

Netzer, R. & J. Xu (1993). Adaptive message logging for incremental replay of message-passing programs. In *Supercomputing '93. Proceedings*, Portland, Oregon, USA, pp. 840–849.

Netzer, R. H. B., T. W. Brennan, & S. K. Damodaran-Kamal (1996). Debugging race conditions in message-passing programs. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, SPDT '96, Philadelphia, Pennsylvania, USA, pp. 31–40. ACM.

Netzer, R. H. B. & B. P. Miller (1992). Optimal tracing and replay for debugging message-passing parallel programs. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, Supercomputing '92, Minneapolis, Minnesota, USA, pp. 502–511. IEEE Computer Society Press.

Pokam, G., C. Pereira, K. Danne, L. Yang, S. King, & J. Torrellas (2009). Hardware and software approaches for deterministic multi-processor replay of concurrent programs. *Intel Technology Journal 13*(4), 20–41.

Păsăreanu, C. S. & N. Rungta (2010). Symbolic PathFinder: symbolic execution of Java bytecode. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, Antwerp, Belgium, pp. 179–180. ACM.

Ronsse, M. & K. De Bosschere (1999, May). RecPlay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst. 17*(2), 133–152.

Ronsse, M., K. De Bosschere, & J. C. de Kergommeaux (2000). Execution replay and debugging. *arXiv preprint cs/0011006*.

Saito, Y. (2005). Jockey: a user-space library for record-replay debugging. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, AADEBUG'05, Monterey, California, USA, pp. 69–76. ACM.

Silva, J. Ditto - deterministic execution replay for java virtual machine on multi-processor. Master's thesis, Instituto Superior Técnico.

Ujma, M. & N. Shafiei (2012, April). jpf-concurrent: an extension of Java PathFinder for java.util.concurrent. *ArXiv e-prints*.

Vallée-Rai, R., P. Co, E. Gagnon, L. Hendren, P. Lam, & V. Sundaresan (1999). Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '99, Mississauga, Ontario, Canada, pp. 13–. IBM Press.

Visser, W., K. Havelund, G. Brat, & S. Park (2000). Model checking programs. In *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on*, Grenoble, France, pp. 3–11.

Wang, N., J. Han, Y. Zhou, & J. Fang (2009). CURRF: a code-based framework for faithful replay distributed applications. In *Proceedings of the 2009 Fourth International Conference on Frontier of Computer Science and Technology*, FCST '09, Shanghai, China, pp. 308–316. IEEE Computer Society.

Zambonelli, F. (1999). An efficient logging algorithm for incremental replay of message. In *Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, IPPS '99/SPDP '99, San Juan, Puerto Rico, USA, pp. 392–398. IEEE Computer Society.

Zamfir, C., G. Altekar, G. Candea, & I. Stoica (2011). Debug determinism: the sweet spot for replay-based debugging. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, HotOS'13, Napa, California, USA, pp. 18–18. USENIX Association.