UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Cooperative Concurrency Debugging

Nuno de Ferraz Almeida e Peixoto Machado

Supervisor: Doctor Luís Eduardo Teixeira Rodrigues

Thesis approved in public session to obtain the PhD Degree in Information Systems and Computer Engineering
Jury final classification: Pass with Distinction and Honour

2016

UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Cooperative Concurrency Debugging

Nuno de Ferraz Almeida e Peixoto Machado

Supervisor: Doctor Luís Eduardo Teixeira Rodrigues

Thesis approved in public session to obtain the PhD Degree in Information Systems and Computer Engineering
Jury final classification: Pass with Distinction and Honour

Jury

Funding Institutions

Fundação para a Ciência e Tecnologia

2016

Acknowledgements

This thesis is the culmination of a sinuous (yet enriching) journey of hard work and persistence, which I was only able to finish due to the guidance of incredible mentors and the support of amazing friends. I would like to deeply thank those that shared this journey with me and helped me along the way.

First of all, I would like to thank my advisor Luís Rodrigues. Your wise comments, alongside your enthusiastic attitude and creativity, were essential for me to bring this work to fruition. You have shaped me as a researcher and I am forever grateful to you for having taught me how to do meaningful research work.

To Brandon Lucia, a heartfelt thanks for believing in and choosing an unknown guy like me, among so many much brighter PhD students around the world, as your intern at Microsoft Research. It was a true privilege to collaborate on research with you. Your fresh ideas and insightful suggestions were fundamental for the results in this thesis. Also, spending the Summer of 2014 at Microsoft in Redmond, and then be a visiting research scholar at CMU one year later, were two of the best experiences of my life, and no words will ever be enough to properly thank you for these opportunities.

To Paolo Romano, a deep thank you for the invaluable help in the early stages of this thesis. Your passionate work ethic inspired me to go that extra mile to produce high quality work. In particular, thanks for joining me in those late night journeys to finish the papers on time.

To my fellow PhD students and researchers of the Distributed Systems Group at INESC-ID, thank you all for the helpful feedback and encouraging words. It was a pleasure to be part of such a talented and smart group. A special word for Nuno Diegues, Xavier Vilaça, Daniel Quinta, Beatriz Ferreira, Hugo Rodrigues, Pedro Mota, and Pedro Ruivo, for the companionship. You have been the best colleagues and friends I could have asked for. I will treasure our lunch discussions, board game nights, football matches, and trips forever.

To the members of the Focolare Movement, thanks for being my second family and for helping me believe that, despite all the sad news we watch everyday, the world is moving towards unity and that universal brotherhood is an ideal worth living for. Thank you also for the fantastic adventures and profound experiences we have shared. You make me a better person and I am truly grateful for having you all in my life.

To my housemates throughout these years in Lisbon: Luís Maia, Tiago Nogueira, José Nogueira, and Violeta Nogueira. Thanks for the amazing time we had together, the enriching conversations, the funny moments, and, of course, the cooking tips. In particular, thank you for always providing me with the perfect environment to rest my mind at the end of every exhausting day of work.

Finally and most importantly, I would like to thank my parents Margarida and José, and my sister Maria. I am grateful beyond anything I could write for your unwavering love, patience, and encouragement. Above it all, thank you for having educated me under the premise that even "if I have the gift of prophecy and can fathom all mysteries and all knowledge, and if I have a faith that can move mountains, but do not have love, I am nothing". You are my alpha and omega, and make me feel the most fortunate person on earth.

This work was partially supported by Fundação para a Ciência e a Tecnologia (FCT), via the individual doctoral grant SFRH/BD/81051/2011, and via the INESC-ID multi-annual funding with reference UID/CEC/50021/2013.

Lisboa, 2016

Nuno de Ferraz Almeida e Peixoto Machado

To my whole family.

Resumo

A complexidade inerente aos programas concorrentes, resultante das variações na ordem de execução dos eventos do programa, abre a porta a vários tipos de erros de concorrência. Os erros de concorrência são particularmente difíceis de depurar e corrigir. Em primeiro lugar, a natureza não-determinista destes erros torna a sua reprodução uma tarefa árdua. Em segundo, é complicado isolar a causa raiz de um erro de concorrência devido ao elevado número de operações e interações entre os fios de execução nas execuções com falha. Por último, as execuções onde o erro se manifesta são difíceis de capturar, visto que geralmente resultam de intercalamentos de operações muitos específicos que se manifestam raramente. Em consequência, os programas concorrentes são muitas vezes instalados com erros de concorrência latentes que podem originar falhas e degradar a fiabilidade dos sistemas em produção.

A presente tese aborda os três desafios acima mencionados, relacionados com a depuração dos erros de concorrência, focando-se em programas concorrentes com memória partilhada. Em particular, esta tese propõe:

- Uma técnica para reproduzir erros de concorrência através da combinação de mecanismos de gravação parcial cooperativa com análise estatística.
- Uma técnica, baseada numa análise diferencial entre intercalamentos de fios de execução erróneos e corretos, para diagnosticar automaticamente a causa raiz de falhas em programas concorrentes.
- Uma técnica para expor erros de concorrência latentes em sistemas em produção através da exploração de variações no entrelaçamento de operações e no fluxo de controlo de execuções corretas.

Foram desenvolvidos protótipos que concretizam as técnicas supracitadas. Com recurso uma extensa avaliação experimental, esta tese mostra que as técnicas desenvolvidas são eficazes, eficientes e que se comparam favoravelmente com outras soluções do estado da arte.

Abstract

The inherent complexity of concurrent programs, due to the variation of the ordering of program events across executions, opens the door for various types of concurrency bugs. Concurrency bugs are notoriously hard to debug and fix. First, the non-deterministic nature of concurrency bugs makes their reproduction challenging. Second, it is hard to isolate the root cause of a concurrency error due to the large number of thread operations and interactions among them in failing schedules. Finally, failing schedules are difficult to expose because they usually result from very specific thread interleavings, which manifest rarely. As a consequence, concurrent programs are often shipped with latent concurrency bugs that can originate failures in production and degrade the systems' reliability.

This thesis addresses these three challenges related to concurrency bugs, focusing on shared-memory multithreaded programs. In particular, we develop:

- A technique to replay concurrency bugs by combining cooperative partial logging and in-house statistical analysis.
- A technique based on differential analysis of failing and non-failing schedules to automatically diagnose the root cause of failures in concurrent programs.
- A technique to expose latent concurrency bugs in deployed programs by exploring variations in the schedule and control-flow behavior of non-failing executions.

We have built prototypes that implement all the aforementioned techniques. Through an extensive evaluation, we show that our tools are effective, efficient, and compare favorably to previous state-of-the-art solutions.

Palavras Chave
Keywords

Palavras Chave

Erros de Concorrência
Gravação e Reprodução
Depuração
Execução Simbólica
Resolução de Restrições

Keywords

Concurrency Bugs
Record and Replay
Debugging
Symbolic Execution
Constraint Solving

Acronyms

CREW	Concurrent-Reader-Exclusive-Writer
DSP	Differential Schedule Projection
DPSP	Differential Path-Schedule Projection
DPOR	Dynamic Partial Order Reduction
POR	Partial Order Reduction
R&R	Record and Replay
SCT	Systematic Concurrency Testing
SMT	Satisfiability Modulo Theories
SPE	Shared Program Element
TLO	Thread-Local Objects Analysis

Table of Contents

List of Figures

List of Tables

1 Introduction

The advent of the multicore era brought *concurrency* to the forefront of software development. By having threads executing simultaneously on different cores, concurrent programs[1] can increase on performance. As such, concurrent software has become very appealing for a wide range of domains including scientific computing, network servers, mobile devices, and desktop applications.

Unfortunately, writing correct (*i.e.*, without programming errors, also called *bugs*) and reliable concurrent software is much more difficult than writing its sequential counterpart [Kramer 2007]. Contrary to sequential software, whose space of possible states depends essentially on the program's input and the execution environment, the state space in concurrent software is substantially larger. In addition to inputs and variations in the execution environment, concurrent programs contain threads that interact by reading from and writing to shared memory locations. The number of possible interleavings between these thread operations is often extremely large and increases significantly the size of the state space of the program. The increased complexity of multithreaded programs makes them more prone to *concurrency bugs* (*i.e.*, errors in code that permit multithreaded schedules that result in undesirable behavior), which are very challenging to understand and debug.

To address the issues of debugging concurrent programs, this thesis proposes techniques that ease the developers' task of diagnosing and fixing concurrency bugs.

1.1 Problem Statement

Concurrency bugs pose a number of challenges that make their debugging notoriously hard. We enumerate the main challenges of concurrency bugs below.

[1] In this thesis, we assume that the term *concurrent program* refers to a *shared-memory multithreaded program*.

1. *Concurrency bugs are difficult to replay.* Without proper synchronization, operations in different threads may non-deterministically adhere to different execution *schedules* and, consequently, produce different results. Although most schedules are correct, some failing schedules can lead to undesirable behavior, like a crash or data corruption. When developers observe such a failure, the common procedure is to employ *cyclic debugging*, *i.e.*, they iteratively re-execute the program in an effort to reproduce the error and narrow down its root cause. However, failures due to concurrency bugs typically result from complex thread schedules with low probability of occurrence, which hinders the task of reenacting a failing execution.

2. *Concurrency bugs are hard to diagnose.* The key to debugging is understanding a failure's *root cause*, *i.e.*, the set of event orderings that are necessary for the failure to occur. The number of events that comprise a root cause is typically small [Lu, Park, Seo, & Zhou 2008; Burckhardt, Kothari, Musuvathi, & Nagarakatte 2010], but it is often unclear which events in a full schedule are truly relevant. Any operation in any thread may have led to the failure and blindly analyzing a full schedule is a metaphorical search for a *needle in a haystack*. Even if the programmer finds the root cause, they still must understand how to change the code in such a way the problematic events do not execute in the failure-inducing order, which is also difficult.

3. *Concurrency bugs are challenging to expose.* As referred before, the subset of thread orderings that lead to a failure often corresponds to a tiny portion of the space of possible execution schedules, and those few failing schedules may manifest rarely. Furthermore, the variation in the schedule in a failing execution may cause a variation in the execution's data-flow, and, subsequently, in its control-flow. Such *schedule-dependent branches* further complicate the task of uncovering failing schedules, because one has to explore not only the space of possible thread schedules for a given execution path, but also the space of different execution paths. Since this huge search space makes complete exhaustive testing infeasible, some failures will likely manifest in production. Failures in deployed systems are not harmless though: they have resulted in severe material costs [Associated Press 2004; NIST 2002], multiple security vulnerabilities [CVEdetails 2016; Sakurity 2016; Yang, Cui, Stolfo, & Sethumadhavan 2012], and, worse, in the loss of human lives [Leveson & Turner 1993].

Uncovering and debugging concurrency bugs in production systems exacerbates the aforementioned challenges. For example, it has been shown that the time required by a programmer to understand and fix an error in occurring "in the field" is directly related to the ability of reproducing it in-house [Kasikci, Schubert, Pereira, Pokam, & Candea 2015; Lu, Park, Seo, & Zhou 2008]. Bug reproducibility can be addressed by *record and replay* techniques [Huang, Liu, & Zhang 2010; Zhou, Xiao, & Zhang 2012; Yang, Yang, Xu, Chen, & Zang 2011], which trace information during production runs to later allow reproducing the failure deterministically. Unfortunately, fully recording a multithreaded execution in impractical, because it incurs high runtime overhead [Huang, Liu, & Zhang 2010; Yang, Yang, Xu, Chen, & Zang 2011]. This means that any concurrency debugging technique designed for production systems must judiciously track runtime information.

The work in this thesis aims at addressing the three aforementioned problems of concurrency bugs in the context of deployed programs. More concretely, this thesis proposes novel techniques to, respectively, replay, diagnose, and expose concurrency bugs in production environments. To cope with the runtime performance overhead challenge, we leverage the observation that software in production is usually executed by a large number of users. Hence, the mechanisms proposed in this thesis rely on *cooperative* tracing schemes that exploit the coexistence of multiple instances of the same concurrent program. The underlying intuition is simple: to share the burden of logging among those instances, by having each instance tracking only a subset of the necessary data (*e.g.*, the thread access ordering to shared memory locations).

In sum, the main goal of this thesis is to design, implement, and evaluate *novel cooperative techniques that allow replaying, diagnosing, and exposing concurrency bugs in production systems, with tolerable recording overhead.*

1.2 Summary of Contributions

The contributions of this thesis tackle the three challenges of concurrency debugging outlined in the previous section with techniques that use information captured from multiple production runs. For each challenge, we enumerate the contributions of this thesis below.

1. To address the challenge of deterministically replaying in-production concurrency bugs, this thesis proposes a novel *cooperative record and replay* approach, which relies on sta-

tistical techniques to *i)* detect similarities among partial logs (independently captured by different user instances of the same program), as well as *ii)* find combinations of partial logs that produce a full log capable of reproducing the failure.

2. To address the problem of isolating the root cause of in-production failures, this thesis proposes a novel *differential schedule projection (DSP)* technique that precisely pinpoints the events that were responsible for a failure by isolating the important control and data-flow changes between a failing and a non-failing schedule.

3. To tackle the challenge of exposing latent concurrency bugs in deployed programs, this thesis proposes a novel *production-guided search* that uncovers failing schedules by exploring variations in the schedule and control-flow behavior of multiple non-failing production runs.

1.3 Summary of Results

We have built prototypes of all the techniques mentioned in the previous section, and evaluated them using both benchmark and real-world concurrency bugs. This section overviews the our main results, whereas Chapters 3, 4, and 5 provide a detailled description of the techniques we have developped and a complete report of their evaluation.

1. We have implemented and evaluated **CoopREP**, a cooperative record and replay framework, on both standard benchmarks for multithreaded applications and real-world applications. The results highlight that CoopREP can successfully replay concurrency bugs involving tens of thousands of memory accesses, while reducing recording overhead with respect to state-of-the-art non-cooperative logging schemes by up to 13x (and by 3x on average).

2. We have implemented and evaluated **Symbiosis**, a root cause diagnosis system based on novel *differential schedule projections* (DSPs), using buggy real-world software and benchmarks. The findings of the experiments show that, in practical time, Symbiosis generates DSPs that both isolate the small fraction of event orders and data-flows responsible for the failure, and show which event reorderings prevent failing. In terms of root cause diagnosis accuracy, DSPs contain 90% fewer events and 96% fewer data-flows than the full

failure-inducing schedules. Moreover, we conducted a user study that provides evidence that, by allowing developers to focus on only a few events, DSPs reduce the amount of time required to pinpoint the failure's root cause and find a valid fix.

3. We have implemented and evaluated **Cortex**, a system that exposes concurrency bugs via production-guided path and schedule exploration, on popular benchmarks and real-world applications. The results demonstrate that Cortex is able to expose failing schedules with only a few (more precisely, 1 to 15) perturbations to non-failing executions, and takes a practical amount of time (less than 22% overhead).

1.4 Publications

Some of the results presented in this thesis have been published as follows:

- Nuno Machado, Paolo Romano, and Luís Rodrigues. "Lightweight Cooperative Logging for Fault Replication in Concurrent Programs". In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '12)*, 2012, ACM.
- Nuno Machado, Brandon Lucia, and Luís Rodrigues. "Concurrency Debugging with Differential Schedule Projections". In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI '15)*, 2015, ACM.
- Nuno Machado, Brandon Lucia, and Luís Rodrigues. "Concurrency Debugging with Differential Schedules Projections". In *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol. 25, No. 2, Article 14, 2016, ACM.
- Nuno Machado, Brandon Lucia, and Luís Rodrigues. "Production-guided Concurrency Debugging". In *Proceedings of the International Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*, 2016, ACM.

1.5 Thesis Roadmap

The rest of this thesis is structured as follows. Chapter 2 presents the background concepts on concurrency debugging and surveys prior work on this topic. Chapters 3, 4, and 5 describe and evaluate CoopREP, Symbiosis, and Cortex, respectively. Chapter 6 concludes this thesis by summarizing its main findings and presenting directions for future work.

2 Background and Related Work

This chapter presents the main concepts, challenges, and systems that served as inspiration for the work presented in this thesis. We start by describing the most common types of concurrency bugs, which are also the ones that are addressed by the techniques proposed in this dissertation. Then, we discuss the related work on concurrency debugging, by dividing it into three main categories according to their purpose: *record and replay systems*, *root cause diagnosis systems*, and *bug exposing systems*. We note that a comprehensive listing of the literature on each category is outside the scope of this thesis. Instead, we focus on describing some of the most prominent solutions proposed over the years to address the challenges of debugging concurrent programs. Finally, throughout this chapter, we also identify the limitations of prior work that motivated our contributions.

2.1 Types of Concurrency Bugs

Multithreaded executions are characterized by having multiple threads running concurrently on different cores. Although threads execute their own instructions individually, they can interact with each other by reading values from and writing values to shared memory locations. Therefore, the value returned by a given read operation can either be the result of a previous write in the same thread or a write in another thread. Since the order in which the different threads perform their operations is not defined *a priori*, the global thread schedule of a multithreaded execution may non-deterministically vary from run to run, and even yield different results for the same input. Figure 2.1 illustrates this fact by depicting a multithreaded program with two threads (*T1* and *T2*) that compete to execute a region of code guarded by a shared variable *free* (initially set to *true*).

Supposedly, only one thread should be able to enter the critical section during a program's run. However, depending on the thread interleaving, the program can reach an incorrect state

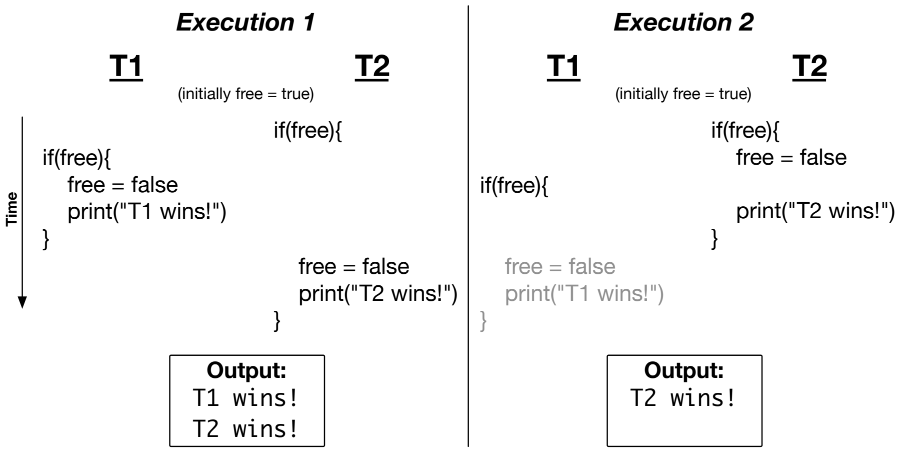


Figure 2.1: Two executions of the same concurrent program with different schedules and outputs. Instructions in light gray do not execute.

where both threads enter the critical section, as depicted in Execution 1 of Figure 2.1. Here, the two threads execute the branch condition with *free* equals to *true*, which allows them to enter the critical section and print the output message. On the other hand, in the second execution (Execution 2), *T2* enters the critical section and sets *free* to *false* prior to *T1* evaluating the conditional clause, thus preventing *T1* to enter the guarded region. As such, the output of the program for execution 2 contains only the message printed by *T2*.

To enforce an order between blocks of operations in different threads, developers resort to *synchronization* [Lampson & Redell 1980; Hoare 1974]. More formally, one says that synchronization operations allow establishing a *happens-before* relationship between events in different threads [Lamport 1978]. Consider two operations *A* and *B*, respectively in two threads *T1* and *T2*. If the memory effects of *T1* performing *A* become visible to *T2* before *T2* executes *B*, then *A happens-before B*.

The happens-before order of an execution schedule, either due to operations executing in the same thread or due to synchronization, determines the read-write linkage for shared variables, *i.e.*, to which write operation corresponds the value read by each read operation. Figure 2.2 represents a variation of the program in Figure 2.1 that relies on synchronization to constraint the orderings of thread operations and read-write linkages permitted in the program's execution.

The executions depicted in Figure 2.2 show that the synchronization operations (*i.e.*, `lock()` and `unlock()`) guarantee that the possible thread schedules always correspond to the correct

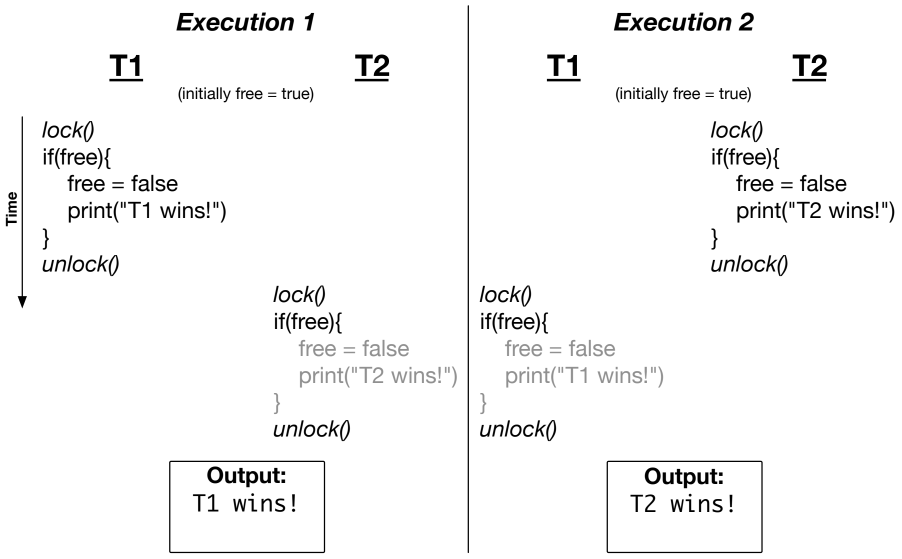


Figure 2.2: Example of multithreaded program that uses synchronization operations to enforce order between threads. Instructions in light gray do not execute.

scenario, where only one thread is indeed allowed to enter the critical section during the execution. Hence, the outcome illustrated in execution 1 of Figure 2.1 can no longer occur in the program of Figure 2.2. Despite that, executions 1 and 2 in Figure 2.2 still demonstrate that both threads have the opportunity to "win the race" and execute the protected region of code. The reason is because, albeit determining the order of other operations in a program's run, synchronization operations themselves may be accessed in a non-deterministic way.

Although the variation in the timing in which threads arrive at synchronization points (or access other shared memory locations) is often intended and beneficial (because it allows improving the performance of the program by executing parallel tasks on different cores), it might sometimes lead to undesirable outcomes. In fact, one can say that the programmer's difficulty to reason about the inherent non-determinism of concurrent programs is the main source of *concurrency bugs*. Concurrency bugs are errors in code that permit multithreaded schedules that result in misbehavior (like a crash or data corruption). Concurrency bugs result from code with misplaced synchronization points and incorrect patterns of accesses to shared variables by different threads. In the following sections, we discuss the most common types of concurrency bugs, namely data races, atomicity violations, ordering violations, and deadlocks.

2.1.1 Data Races

A data race occurs whenever *i)* there are two concurrent, unsynchronized (*i.e.*, not ordered by a happens-before relationship) accesses to the same shared memory location, and *ii)* at least one of the accesses is for writing. For example, Figure 2.1 contains a data race on *free*, as both threads can read from and write to this shared variable without being protected by any synchronization operation.

People in the research community have somewhat divergent opinions on how to classify data races. While some usually distinguish between "harmful" and "benign" data races [Kasikci, Zamfir, & Candea 2012; Narayanasamy, Wang, Tigani, Edwards, & Calder 2007; Naik, Aiken, & Whaley 2006], others take a stronger position and argue that any code that permits a data race is incorrect and, thus, all data races should be deemed concurrency bugs [Boehm 2011].

The distinction between "harmful" and "benign" data races stems from the disambiguation between the terms *data race* and *race condition*. Data races are characterized by the precise definition provided above, thus, they can be accurately detected via automated mechanisms [Kasikci, Zamfir, & Candea 2012]. In contrast, a race condition can be defined as a violation of the program's correctness due to an erroneous timing or ordering of the events in the execution schedule [Regehr 2011; Kasikci 2013]. Since this definition depends on semantic program-level invariants, one may not always be able to accurately detect race conditions.

Data races and race conditions are often linked: many race conditions are due to data races, and many data races lead to race conditions. In fact, all harmful data races can be considered race conditions. However, it should be noted that the concepts of data races and race conditions do not always overlap, nor one is the subset of the other. As an example, consider the two executions presented in Figure 2.3. Note that the ordering of the thread operations of each execution causes the respective assertion to fail. The reason is because $x = 0$ when *T1* reaches the assertion, which violates the condition $x > 0$. The two execution schedules are race conditions because the failure is due to a particular thread interleaving that violates a program invariant. However, conversely to the execution in Figure 2.3.a, the execution depicted in Figure 2.3.b does not include a data race, because the writes by *T1* and *T2* to x are protected by locks, as well as the read of x within the assertion. A more thorough distinction between data races and race conditions can be found in Regehr [2011].

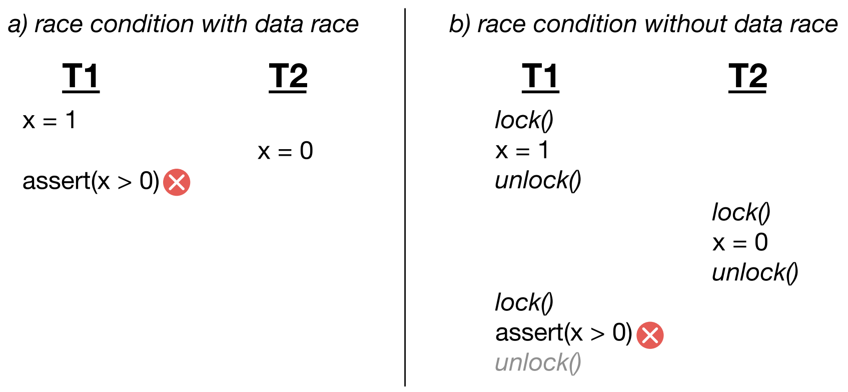


Figure 2.3: Two executions with race conditions from different programs. Execution *a)* includes a data race, whereas execution *b)* does not. Instructions in light gray do not execute, as they happen after the failure.

The claim that all data races are harmful [Boehm 2011] builds upon the observation that most popular programming language specifications (*e.g.*, C [Committee 2010] and C++ [Committee & Beker 2011]) do not provide clearly defined semantics for programs with data races at the source code level. The reason for such undefined semantics is to allow compilers to perform optimizations that reorder instructions, without being overly constrained by the language. Unfortunately, since those optimizations may break the code of programs with data races in arbitrary ways [Boehm 2011], according to the specifications of these languages, not only any code that permits a data race is considered buggy, but also no data race is deemed as benign.

Yet another reason for the aforementioned strong position about data races is related to the *architecture memory model* upon which the program executes. An architecture's memory model [Goodman 1989; Sewell, Sarkar, Owens, Nardelli, & Myreen 2010; Price 1995] determines: *i)* how threads interact through shared memory, *ii)* the value retrieved by a shared memory access, *iii)* the timing at which a variable update becomes visible to other threads, and *iv)* what assumptions hold when one is writing a program or applying some program optimization. When writing a program, developers typically assume a *sequentially consistent (SC)* memory model [Lamport 1979]. Under sequential consistency, the operations of each thread execute according to the program order, and the same order of operations is observed by all threads. However, in order to achieve better performance, most architecture memory models provide consistency guarantees that are more relaxed than SC. Relaxed consistency allows some hardware/compiler aggressive optimizations to modify the order of thread operations, which may cause different threads to observe different execution schedules. As such, since reasoning about data races running on weak memory models is extremely hard for humans, some researchers consider wiser to classify all data races as harmful.

For the purpose of this thesis, we consider that only data races that lead to failures under a sequentially consistent memory model (*i.e.*, data races that are race conditions) are concurrency bugs.

2.1.2 Atomicity Violations

Atomicity violations are another common class of concurrency bugs. Atomicity violations are intimately related to the concepts of *atomicity* and *serializability*, which we introduce below.

- *Atomicity* is a property of a multithreaded program segment that allows a sequence of operations from a thread to be perceived by other threads as if it has executed instantaneously. This means that other threads either observe all the results of the atomic sequence of operations or none of them. In other words, threads can never see the result of some instructions of the atomic sequence and not of others.
- *Serializability* is a property about a multithreaded execution, which guarantees that the result of the execution of a set of atomic sequences of operations is equivalent to that of some sequential execution of those atomic segments.

To enforce atomicity in a region of code, developers must enclose that block of instructions in a *critical section*. Critical sections can be implemented by means of synchronization (*e.g.*, via locks). However, if the developer omits or misplaces the synchronization points, the operations intended to execute atomically can be incorrectly interleaved by other thread operations on the same shared variables, thus causing an atomicity violation [Lucia, Devietti, Strauss, & Ceze 2008]. In other words, an atomicity violation occurs when regions of code that are supposed to execute atomically do not, due to an buggy implementation of the critical sections. The race conditions in Figure 2.3 are atomicity violations: the program fails because the write to x and the assertion check in *T1* should have executed atomically, but were erroneously interleaved by the write in *T2*. Note that an atomicity violation may or may not involve a data race, as demonstrated by Figure 2.3. Either way, a possible fix to this bug consists in encompassing *T1*'s operations in a single critical section.

If the thread schedule in a concurrent execution alters the outcome of a region of code that was intended to be atomic, then such execution is said to be *unserializable*. Violations of atomicity and violations of serializability are thus related in the sense that schedules that lead to atomicity violations are unserializable. In fact, prior work has demonstrated that if an execution is proved to be serializable (with respect to some specified atomic regions), then there are no atomicity violations [Xu, Bodík, & Hill 2005; Flanagan & Freund 2004; Flanagan, Freund, & Yi 2008; Lu, Tucek, Qin, & Zhou 2006].

In this thesis, we are only concerned with atomicity violations that are race conditions, *i.e.*, that lead to failures, such as a crash or hang. Therefore, unlike some previous work [Flanagan, Freund, & Yi 2008], we do not treat any violation of serializability as an atomicity violation. The reason is because programs may have violations of serializability that do not result in erroneous behavior.

2.1.3 Ordering Violations

Ordering violations are concurrency bugs that occur when the order in which two groups of operations (from different threads) should execute is reversed [Lu, Park, Seo, & Zhou 2008]. Similarly to atomicity violations, we also consider ordering violations as race conditions.

Figure 2.4 shows an ordering violation. The initialization of object *o* in *T1* should always happen before the method call in *T2*, as illustrated by Figure 2.4a. However, due to the absence of synchronization, the program permits the failing schedule depicted in Figure 2.4b, in which *T2* attempts to perform the method invocation before *o* is initialized.

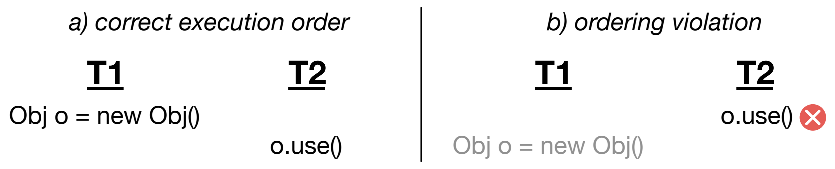
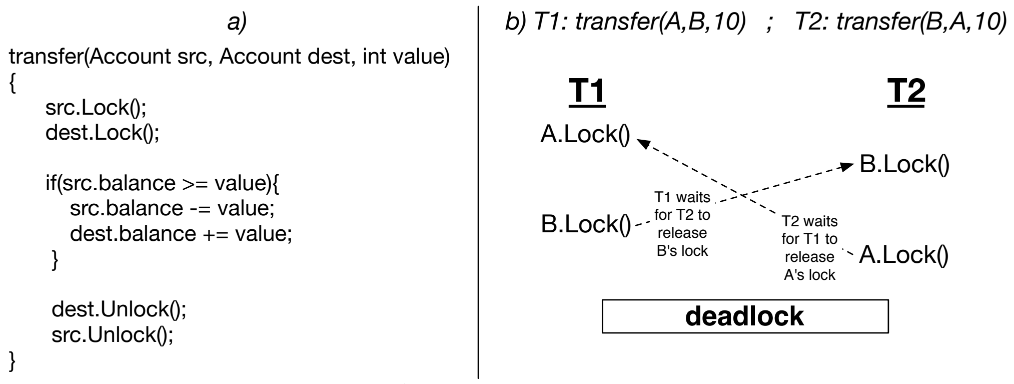


Figure 2.4: Example of an ordering violation. Execution *a)* depicts the correct schedule, whereas execution *b)* fails because the operations execute in the wrong order. Instructions in light gray do not execute, as they happen after the failure.

Ordering violations can be fixed using synchronization in such a way that the correct order is enforced in all executions. However, since this kind of concurrency bug is not related to atomicity problems, it typically does not suffice to add or change locks to fix an ordering violation. In fact, even if one removes the data race in Figure 2.4b by wrapping the operations of both threads within a lock, the ordering violation may still occur. In turn, a possible way to fix the bug in Figure 2.4 would be adding a memory barrier or, alternatively, a guard condition to *T2* that checks the state of o, in order to guarantee that o.use() is executed only after the object initialization.

2.1.4 Deadlocks

A deadlock occurs when two or more operations indefinitely wait for each other to release a previously acquired resource (*e.g.*, a lock or monitor). Figure 2.5 contains an example of a deadlock bug. The code in Figure 2.5a simulates a money transfer between two bank accounts. The method implements a critical section (by acquiring the locks for the source and destination accounts) in order to make sure that only a single thread is allowed to change the balances of the two accounts at a time.


a)

```
transfer(Account src, Account dest, int value)
{
    src.Lock();
    dest.Lock();

    if(src.balance >= value){
       src.balance -= value;
       dest.balance += value;
     }

     dest.Unlock();
     src.Unlock();
}
```

b) T1: transfer(A,B,10) ; T2: transfer(B,A,10)
T1
T2
A.Lock()
B.Lock()
T1 waits for T2 to release B's lock
B.Lock()
T2 waits for T1 to release A's lock
A.Lock()
deadlock

Figure 2.5: Example of a deadlock. *a)* source code of a method that transfers money between two accounts; *b) T1* and *T2* are in a deadlock because each one of the threads needs to acquire a resource that belongs to the other thread.

However, if two threads attempt to simultaneously run method transfer on two accounts A and B in opposite order, the execution of the program can hang due to a deadlock. As showed in Figure 2.5b, if *T1* executes transfer(A,B,10) and *T2* runs transfer(B,A,10), both threads will try to acquire the locks in reverse order. Hence, it can happen that *T2* acquires the lock for account

B right after *T1* has acquired the lock for account A, thereby yielding a deadlock scenario where both threads wait indefinitely for the other thread to release their lock.

Although the strategies employed to fix deadlock bugs are not usually straightforward [Lu, Park, Seo, & Zhou 2008], the concurrency bug in Figure 2.5 can be easily corrected by replacing the two account locks by a single global lock (albeit at the cost of performance degradation).

2.1.5 Other Concurrency Bugs

There are other kinds of concurrency bugs in addition to the ones discussed in the previous sections, namely livelocks [Musuvathi, Qadeer, Ball, Basler, Nainar, & Neamtiu 2008] and errors involving thread interactions via shared resources other than shared memory [Laadan, Viennot, Tsai, Blinn, Yang, & Nieh 2011]. Nevertheless, we opted for focusing on data races, atomicity and ordering violations, and deadlocks, because they are arguably the most pervasive types of concurrency bugs [Lu, Park, Seo, & Zhou 2008] and the target of the debugging techniques presented in this thesis.

2.2 Record and Replay Systems

Record and replay (R&R) tools help developers to overcome the problems associated with the reproduction of concurrent executions, namely those raised by non-determinism. Concretely, the R&R approach allows re-executing a multithreaded program, obtaining the exact same behavior as the original execution. R&R can be useful for a wide range of applications, namely fault tolerance [Bressoud & Schneider 1995], security [Joshi, King, Dunlap, & Chen 2005; Chow, Garfinkel, & Chen 2008], and testing [Musuvathi, Qadeer, Ball, Basler, Nainar, & Neamtiu 2008]. However, in the context of this thesis, we are interested in R&R techniques for debugging purposes, in particular, to reproduce failures caused by concurrency bugs.

Reenacting a previous run is possible as long as all non-deterministic factors that have an impact on the program's execution are replayed in the same way [Park, Zhou, Xiong, Yin, Kaushik, Lee, & Lu 2009]. Record and replay operates in the following two phases:

1. **Record phase.** Consists of capturing data regarding non-deterministic events (*e.g.*, thread schedule, network and disk inputs, the order of asynchronous events, etc.)

into a trace file.

2. **Replay phase.** The application is re-executed consulting the trace file to guarantee the replay of the non-deterministic events according to the original execution.

Unfortunately, obtaining a high fidelity replay of a multithreaded execution (on a multicore environment) incurs an extremely high recording overhead (10x-100x) [Park, Zhou, Xiong, Yin, Kaushik, Lee, & Lu 2009]. On the other hand, the less information is collected, the harder it will be to deterministically reproduce the failure. Thereby, the main challenge of R&R systems lies in finding the sweet spot between recording overhead and bug reproducibility guarantees. Over the past decades, a large body of research has been devoted to explore the tradeoffs between these two factors in the context of R&R for multiprocessors. This section overviews some of these prior efforts by dividing them into two broad categories: *hardware-assisted recording* and *software-only recording*. We defer to the literature for a more comprehensive analysis of the related work in R&R (*e.g.*, [Pokam, Pereira, Danne, Yang, & Torrellas 2009; Chen, Zhang, Guo, Li, Wu, & Chen 2015]).

2.2.1 Hardware-Assisted Recording

Bacon & Goldstein [1991] proposed one of the first hardware-based mechanisms for multiprocessor replay by attaching a hardware instruction counter to cache-coherence messages to identify memory sharing. Although fast, this mechanism produces a large log.

Later, new hardware extensions were proposed to achieve more efficient record and replay. Flight Data Recorder (FDR) [Xu, Bodik, & Hill 2003] focuses on recording enough information to re-construct the last second of the whole-system execution prior the failure. Like Bacon & Goldstein's scheme, FDR snoops the cache-coherence protocol, but it improves the former scheme by using a modified version of the Netzer's transitive reduction algorithm [Netzer 1993] to reduce the number of memory races traced.

BugNet [Narayanasamy, Pokam, & Calder 2005] uses FDR's mechanism to record inter-thread dependencies, but, unlike FDR, does not aim at providing whole-system replay. Instead, BugNet logs only the first read from shared memory in each checkpoint interval, or when a data race is detected, into a hardware-based dictionary. This is enough to guarantee the deterministic re-execution of application-level failures, as well as allow each thread to be replayed independently.

DeLorean [Montesinos, Ceze, & Torrellas 2008] are hardware-assisted R&R approaches, where instructions are atomically executed by processors as blocks (or chunks), similarly to transactional memory or thread-level speculation. This way, DeLorean only needs to log the block commit order, rather than the global thread interleaving.

Hardware-assisted approaches, although appealing due to their low recording overhead, have the drawback of relying on expensive and intrusive hardware modifications. Despite some efforts to reduce hardware complexity [Montesinos, Hicks, King, & Torrellas 2009; Honarmand, Dautenhahn, Torrellas, King, Pokam, & Pereira 2013], all the aforementioned techniques are still solely available on simulations. For that reason, most recent research has been focused on software-only approaches.

2.2.2 Software-Only Recording

Software-only R&R systems can be broadly classified as *order-based* or *search-based*. In the following, we discuss some of the most prominent solutions proposed over the years for each one of the two categories.

Order-based. Order-based approaches log and reproduce the ordering of critical events, such as shared-memory accesses or synchronization operations. InstantReplay [LeBlanc & Mellor-Crummey 1987] was one of the earliest attempts to achieve software-only order-based deterministic replay on multiprocessors. InstantReplay assigns a version number to shared memory objects and uses the CREW (concurrent-reader-exclusive-writer) protocol to implement version updates. At runtime, whenever a thread reads from/writes to a shared object, it stores the corresponding version into its own trace file.

SMP-Revirt [Dunlap, Lucchetti, Fetterman, & Chen 2008] operates on multiprocessor virtual machines and leverages commodity hardware page protection to detect races between multiple CPUs, instead of instrumenting every shared memory access. By tracking dependencies at page-level granularity, this approach works well in applications with coarse-grained data sharing, resulting in less than 10% performance degradation for up to 2 cores. However, for 4 cores and applications with fine-grained data sharing or false sharing, the performance of SMP-ReVirt drops significantly (>400% runtime overhead). Scribe [Laadan, Viennot, & Nieh 2010] proposes a number of optimizations in order to mitigate thrashing caused by frequent transfers of page ownership and improve the performance of multi-processor tracking. Concretely, Scribe defines a minimal ownership retention interval and forbids ownership transitions during the interval time. This technique allows relieving the contention among threads, but makes it harder to capture atomic violation bugs during recording.

DejaVu [Choi & Srinivasan 1998] uses a global clock to log the total order of threads accesses to shared memory in multithreaded Java applications. Although efficient in an uniprocessor environment, this approach is not practical for multicores platforms, as the contention in the single global lock imposes heavy performance slowdown.

JaRec [Georges, Christiaens, Ronsse, & De Bosschere 2004] drops the need for global ordering and uses a Lamport's clock [Lamport 1978] to record the synchronization order. This technique allows reducing both the log size and the runtime overhead, but has the downside of requiring the program to be free of data races in order to guarantee a correct replay. This constraint makes JaRec's approach unattractive for debugging real-world concurrent applications, because data races are among the most common types of concurrency bugs [Lu, Park, Seo, & Zhou 2008].

Chimera [Lee, Chen, Flinn, & Narayanasamy 2012] also builds upon the insight that it suffices to record the synchronization order to provide deterministic multiprocessor replay for data-race-free programs. Chimera addresses JaRec's limitations by performing a whole-program static data race detection to identify potential data races. To cope with false positives, Chimera instruments pairs of potentially racing instructions with a coarse-grained weak-lock mechanism, which provides sufficient guarantees to allow deterministic replay while relaxing the synchronization overhead imposed by traditional locks.

LEAP [Huang, Liu, & Zhang 2010] is a R&R Java system that introduced the idea of tracing the thread interleaving with respect to each individual shared memory location. In particular, LEAP associates an access vector to each shared field or synchronization variable to stores the identifiers of the threads that access that variable at runtime. This way, one gets local-order vectors of thread accesses performed on individual shared variables, instead of a global-order vector. LEAP relies on a conservative static analysis to identify shared variables, thus it is not capable of distinguishing accesses to the same field of different object instances. As a result, LEAP incurs large unnecessary contention overhead for programs that instantiate the same class multiple times. ORDER [Yang, Yang, Xu, Chen, & Zang 2011] mitigates LEAP's contention issues by recording and replaying shared accesses at object instance granularity, instead of class granularity.

CARE [Jiang, Gu, Xu, Ma, & Lu 2014] strives to reduce the amount of information recorded by exploiting thread access locality. CARE maintains a per-thread cache that buffers the most recent values read from shared variables, and only tracks the exact read-write linkages for variables being written with different values. Similarly to CARE, Light [Liu, Zhang, Tripp, & Zheng 2015] recently proposed capturing solely the relevant flow dependencies between reads and writes. To achieve this, Light maintains the last write access to a shared memory location such that when the location is read, the flow dependence is recorded into a thread-local buffer. The flow dependencies, along with the thread local orders (which are not logged because they can be easily inferred), are encoded as scheduling variables in a constraint system. Light then uses a Satisfiability Modulo Theories (SMT) solver to solve the constraints and derive a feasible replay schedule.

Search-based. Search-based approaches tradeoff high-fidelity bug reproducibility (offered by order-based solutions) for reduced record cost. These approaches typically capture incomplete information at runtime and, then, apply post-recording inference mechanisms to construct a valid failing schedule. Since an exhaustive search of the thread interleaving space is NP-complete [Gibbons & Korach 1997], search-based techniques essentially vary in the kind of information tracked during recording and in the strategy employed to explore the state space.

ESD [Zamfir & Candea 2010] proposed *execution synthesis*, a technique that does not require any tracing during the original execution and, therefore, incurs no runtime overhead. ESD leverages the core dump and the bug report from the failed run to synthesize program failures via symbolic execution (we further discuss symbolic execution in Section 2.4.1). To address the state space explosion problem, ESD relies on static analysis and heuristics to synthesize races and deadlocks. Despite being attractive for performance-sensitive applications, ESD can incur high inference times, as heuristics can suffer of lack of precision. Moreover, ESD requires the core dump of the application, which may not always be available.

ODR [Altekar & Stoica 2009] provides output determinism, meaning that it aims at inferring an execution with the same output as the production run, but not necessarily with the same schedule. To this end, ODR traces both the program inputs and outputs, the synchronization order, and a sample of the thread execution path. As ODR does not record the outcomes of data races, it has to infer them in order to successfully reproduce the failure. This is achieved by employing heuristics that leverage the information recorded to explore the space of possible schedules and find one whose output is identical to that of the original execution. Despite that, ODR's inference time can still be too long to be practical (more than 24 hours in some cases).

PRES [Park, Zhou, Xiong, Yin, Kaushik, Lee, & Lu 2009] focus on guaranteeing best-effort probabilistic replay. The underlying idea consists of minimizing the recording overhead during production runs, at the cost of an increase in the number of attempts to replay the bug during diagnosis. Similarly to ODR, PRES logs a partial trace of the original execution (dubbed sketch), which does not include the order of racy instructions. PRES reconstructs the complete failing schedule by relying on an intelligent offline replayer to search the space of possible data race outcomes that also conform with the sketch. To efficiently explore the state space, PRES leverages feedback produced from each failed attempt to guide the subsequent one. This mechanism often allows to successfully replay bugs in a few number of attempts.

Stride [Zhou, Xiao, & Zhang 2012] achieves search-based deterministic replay by recording bounded read-write linkages and then inferring an equivalent execution schedule in polynomial time. Inspired by the CREW semantics, Stride uses extra synchronization only to track the exact order of shared write operations. For read operations, Stride stores the latest version of the write that the read can possibly link to (*i.e.*, the bounded linkage). This bounded linkage allows the post-recording search to focus only on the writes with older versions for reconstructing a complete schedule equivalent to the original one.

Recently, CLAP [Huang, Zhang, & Dolby 2013] proposed an R&R approach based on guided symbolic execution and SMT constraint solving. Instead of capturing runtime information regarding the thread interleaving, CLAP traces the path that each thread executes locally. The purpose of tracing the control-flow choices is to guide the symbolic execution directly through the failure-inducing path, thus avoiding the need to explore multiple possible executions paths, as done in ESD [Zamfir & Candea 2010]. During symbolic execution, CLAP gathers symbolic information about accesses to shared memory and path conditions, with the purpose of encoding a constraint model that represents the possible thread schedules that lead to the failure. CLAP then feeds the constraint model into an SMT solver, which outputs the order of thread operations that cause the program to fail.

2.2.3 Limitations of Previous Work and Opportunities

Prior work on record and replay have explored several different tradeoffs between recording overhead and bug reproducibility guarantees. Despite that, all the previous solutions still strive to capture all the necessary information for bug replay from a single execution. For order-based techniques this typically means incurring higher overhead to guarantee that all relevant details for reproducing the error at the first attempt are tracked. For search-based techniques, taking into account solely one execution typically results in longer inference phases.

However, in practice, software is executed several times by multiple users [Candea 2011]. Thus, one could leverage the large number of production runs to achieve more cost-effective R&R. In this thesis, we built upon this observation and take a step forward towards finding the sweet spot between recording overhead and bug reproducibility guarantees. We introduce *cooperative record and replay*, a search-based technique that further reduces the logging overhead by distributing the information to be traced across multiple instances of the program. Chapter 3 describes cooperative record and replay in detail, as well as CoopREP, which is the framework we built to implement our technique.

2.3 Root Cause Diagnosis Systems

We define a *root cause* of a failure as the sequence of events that lead the program to fail, such that when the event ordering is changed or prevented from happening, the failure does not recur [Wilson, Dell, & Anderson 1993]. In this section, we review a variety of techniques that have been proposed over the years to aid developers understand the root causes of failures and debug their underlying bugs. Since this has been an extensively studied topic, our description is mostly focused on solutions that diagnose concurrency bugs. We classify these techniques into three types: *pattern analysis*, *statistical analysis*, and *other approaches.* The section ends by setting the contributions provided by this thesis in the state of the art of root cause diagnosis.

2.3.1 Pattern Analysis

Pattern analysis approaches [Park, Vuduc, & Harrold 2010; Lucia, Ceze, & Strauss 2010; Lu, Tucek, Qin, & Zhou 2006] search an execution, dynamically or by reviewing a log, for problematic patterns of memory accesses.

AVIO [Lu, Tucek, Qin, & Zhou 2006] relies on an offline training phase to learn benign data access patterns, in order to later report erroneous access interleavings in monitored runs. Falcon [Park, Vuduc, & Harrold 2010] and Unicorn [Park, Vuduc, & Harrold 2012] improve AVIO's accuracy by computing statistical suspiciousness scores.

ColorSafe [Lucia, Ceze, & Strauss 2010] assigns variables into "color sets" and then performs serializability checks on colors rather than on individual variables to detect and isolate single- and multi-variable atomicity violations. Although often effective, pattern analysis solutions have the drawback of missing bugs that do not fit the known patterns.

2.3.2 Statistical Analysis

Cooperative approaches diagnose concurrency failures via statistical analysis over information gathered from a large number of user instances of the program. Cooperative bug isolation (CBI) [Liblit, Aiken, Zheng, & Jordan 2003] pioneered this approach for sequential bugs. CBI uses lightweight instrumentation to sample both failing and successful production runs, capturing execution predicates (*e.g.*, variable values and control-flow properties). CBI then applies statistical inference to find which predicates are most likely to be related to a failure. Cooperative crug isolation (CCI) [Jin, Thakur, Liblit, & Lu 2010] extends CBI to support concurrency errors, namely by coordinating sampling across different threads in order to track interleaving-related predicates.

PBI [Arulraj, Chang, Jin, & Lu 2013] and LBRA/LCRA [Arulraj, Jin, & Lu 2014] also monitor production runs and employ statistical techniques to isolate the root cause of sequential and concurrency bugs. These systems leverage hardware features, such as performance counters and short-term memory, to diagnose the failure with low overhead. Unlike PBI, LBRA/LCRA does not perform sampling, but requires custom hardware extensions. Furthermore, due to the hardware limited capacity to store event traces, LBRA/LCRA only works well for bugs where the root cause is close to the failure point.

Bugaboo [Lucia & Ceze 2009] collects *context-aware communication graphs* that represent inter-thread communications via shared memory and uses statistical analysis to report anomalous communication events. Recon [Lucia, Wood, & Ceze 2011] improves the accuracy of Bugaboo's communication graphs with information about the actual interleavings that comprise the root cause of the bug.

More recently, Gist [Kasikci, Schubert, Pereira, Pokam, & Candea 2015] introduced *failure sketching*, which is an automated debugging technique that provides developers with a high-level explanation (denoted failure sketch) of a failure occurred in production. A failure sketch reports the instructions that result in misbehavior and summarizes the differences between failing and successful runs. Gist builds failure sketches via in-house static program analysis and in-production cooperative statistical analysis. To achieve lightweight runtime tracing, Gist relies on Intel Processor Trace [Intel Corporation 2013] and hardware watchpoints.

2.3.3 Other Approaches

Delta debugging [Zeller & Hildebrandt 2002] is a technique that isolates failure-inducing inputs in sequential programs by systematically narrowing the differences between passing and failing executions. The work of Choi & Zeller [2002] extends this technique to pinpoint the buggy portion of a multithreaded schedule by combining delta debugging with R&R and random schedule jitter. Triage [Tucek, Lu, Huang, Xanthos, & Zhou 2007] uses checkpointing to repeatedly replay the moment of failure and applies delta debugging along with dynamic slicing to perform failure diagnosis at the user site. DrDebug [Wang, Patil, Pereira, Lueck, Gupta, & Neamtiu 2014] is similar to Triage in that it also uses dynamic slicing for root cause diagnosis. However, DrDebug relies on R&R instead of checkpointing to reproduce the error, and is not limited to debugging concurrency bugs on single processors, like Triage.

Sherlog [Yuan, Mai, Xiong, Tan, Zhou, & Pasupathy 2010] combines program analysis with logs from failed production runs to automatically generate useful control- and data-flow information to help developers diagnose the root causes of errors, without requiring neither the reproduction of the failure nor any knowledge about the log's semantics.

Yet another approach to root cause diagnosis is computing unsatisfiable cores with *Satisfiability Modulo Theories* (SMT) solvers. The unsatisfiable core is the subset of constraints in constraint system that make such formulation impossible to solve (*i.e.*, there is no feasible solution that allows satisfying all the constraints in the model). Unsatisfiable cores can be used to isolate bugs by feeding an SMT solver with a constraint model (representing a correct execution of the program) and a failure-inducing constraint (*e.g.*, an assertion violation). Since the two sets of constraints will conflict, as one requires the program not to fail and the other requires the failure to occur, the SMT solver will output *unsatisfiable.* In addition, the solver also indicates the conflicting constraints (*i.e.*, the unsatisfiable core), which represent the root cause of the failure.

BugAssist [Jose & Majumdar 2011] pioneered the use of UNSAT cores to isolate errors in software programs, although it only supports sequential bugs. ConcBugAssist [Khoshnood, Kusano, & Wang 2015] extended BugAssist to handle bugs in multithreaded programs, as well as generate automated repairs by casting the binate covering problem as a constraint formulation. However, ConcBugAssist has the drawback of requiring model checking the entire program and compute all possible schedules that prevent the failure, which is hard to do in practice.

2.3.4 Limitations of Previous Work and Opportunities

As mentioned before, root cause diagnosis techniques based on pattern analysis cannot isolate bugs that do not fit the erroneous patterns. On the other hand, solutions that rely on statistical analysis are general, but have the downside of requiring many failing and non-failing execution traces to be able to accurately pinpoint the events that lead to the failure.

In this thesis, we also leverage the insight that a failing schedule typically deviates from a non-failing schedule in only a few critical ways to propose a novel technique to precisely pinpoint the root cause of concurrency bugs, denoted *differential schedule projection* (or DSP, for short). A key advantage of our approach is that it requires only a *single* failing execution, does not rely on statistical reasoning nor is limited to bug patterns, and produces precise results. Chapter 4 describes differential schedule projections in detail, as well as Symbiosis, which is the system that employs this technique to zero in on the root cause of concurrency bugs.

2.4 Bug Exposing Systems

Bug exposing systems perform explorative testing with the goal of uncovering failures in software. Testing is typically performed along two dimensions: *space of possible inputs* and *space of possible schedules*. For sequential bugs, it suffices to explore the former search space for an input that causes the program to fail. However, for concurrency bugs, one usually has to explore the two dimensions, in order to find both an input that leads the execution to the faulty portion of code and a thread schedule that triggers the failure.

Much work has been conducted in the past to generate comprehensive sets of inputs to provide code and specification coverage [Beizer 1990]. Most of this work is also applicable to concurrent programs, albeit with some extensions to increase code coverage [Sen & Agha 2006a].

On the other hand, finding a failing schedule still remains a challenging problem: the subset of thread orderings that lead to the failure often corresponds to a tiny portion of the space of possible execution schedules. Moreover, the existence of schedule-sensitive branches (*i.e.*, branches whose decision may vary depending on the actual interleaving of concurrent threads) further complicates the task of exposing failing schedules, because one has to explore not only the space of possible thread schedules for a given execution path, but also the space of different control-flow paths.

Since the huge search space of multithreaded programs makes complete exhaustive testing infeasible, bug exposing systems employ techniques to prune the search space and guide the exploration towards paths and schedules that are most likely to expose bugs. In the following, we discuss some prior work on concurrency testing, dividing the contributions into three broad categories: *symbolic execution*, *systematic testing*, and *other approaches*. We end the section by identifying the main limitations of these approaches, which this thesis intends to address.

2.4.1 Symbolic Execution

Symbolic execution [King 1976] is a testing technique that consists in executing a program replacing concrete input values by symbolic variables. A symbolic variable represents a set of possible concrete values. Assignments to and from symbolic variables and operations involving symbolic variables produce results that are also symbolic. When an execution reaches a branch dependent on a symbolic variable, it spawns two identical copies of the execution state – one in which the branch is taken, and another in which the branch is not taken. Spawned copies continue independently along these different paths and the process repeats for every new symbolic branch. Each path has a path constraint, encoding all branch outcomes on that path. Thus, the path constraint determines the possible set of concrete values for symbolic variables that lead execution down a particular path. When a path terminates or hits a bug, a test case can be generated by solving the current path constraint for concrete values. Excluding non-deterministic factors, such test case can be later used to re-execute the program, making it follow the same path and hit the same bug.

The ultimate goal of symbolic execution is to explore all feasible control-flow paths of a program [Myers & Sandler 2004]. The classic approach to symbolic execution consists in performing depth-first exploration of the paths using backtracking [Visser, Păsăreanu, & Khurshid 2004]. For large or complex programs, however, it becomes computationally intractable to maintain a precise state and solve the constraints required for an exhaustive exploration. To address the *path explosion* problem, researchers have proposed a number of search heuristics (such as random path selection [Cadar, Dunbar, & Engler 2008], state memoization [Godefroid 2007], and concrete test monitoring [Tillmann & De Halleux 2008]), as well as techniques to parallelize the state exploration [Bucur, Ureche, Zamfir, & Candea 2011].

Another approach to address the limitations of symbolic execution based testing is *concolic* (*conc*rete + symb*olic*) execution. Concolic execution [Godefroid, Klarlund, & Sen 2005; Burnim & Sen 2008] uses concrete and symbolic values simultaneously to explore distinct behaviors that may result from executing a program with different data inputs. The key idea is to run the program on some concrete input values and perform dynamic symbolic execution with the goal of gathering symbolic constraints at branch statements that may lead to alternate behaviors. The concrete execution is also used to simplify symbolic expressions that cannot be handled by the constraint solver.

Symbolic and concolic execution approaches have been mostly applied to find bugs in sequential programs [Cadar, Dunbar, & Engler 2008; Godefroid, Klarlund, & Sen 2005; Tillmann & De Halleux 2008; Burnim & Sen 2008]. Despite that, some recent efforts have made progress in extending classic symbolic/concolic execution tools to support multithreaded programs and concurrency testing. For instance, CUTE [Sen, Marinov, & Agha 2005] and jCUTE [Sen & Agha 2006b] (CUTE for Java) combine concolic execution with dynamic partial order reduction to systematically generate both test inputs and thread schedules. In the presence of data races, these systems generate new tests by either fixing the schedule and re-executing the program for a new input, or by keeping the same input and reordering the events involved in a data race to exercise a new schedule.

Con2colic testing [Farzan, Holzer, Razavi, & Veith 2013] leverages concolic execution to test concurrent programs with code coverage guarantees. Starting from a given initial input and thread interleaving, this system explores the state space by systematically forcing reads by a thread to return the value written by writes on another threads in order to produce new execution interleavings. The feasibility of a newly generated schedule is checked by means of a constraint solver. Con2colic testing is sound and complete (within the limitations of concolic execution), being only bounded by the time and space available to perform the computation. For this reason, con2colic testing can also be classified as a systematic testing system. We present systematic testing in the following section.

2.4.2 Systematic Testing

Systematic concurrency testing (SCT), also known as *stateless model checking*, consists in repeatedly executing a multithreaded program, controlling the ordering of thread operations so that a different interleaving is explored on each execution. This procedure goes on until all schedules have been explored, or until a time or schedule threshold is reached. Systematic testing has the advantage of being highly automatic, incurring no false positives, and allowing to replay a bug by enforcing the schedule that triggered it.

Since exploring all possible executions schedules of a program is intractable, due to the exponential nature of the search space, SCT techniques essentially differ on the strategy used to cope with schedule explosion. The two most widely adopted approaches to SCT are *partial order reduction* (POR) and *schedule bounding.* We now discuss each approach in more detail.

Partial Order Reduction. Systems that apply POR aim at reducing the number of schedules that need to be explored without false negatives, by exploring only one of each group of partial-order-equivalent set of executions [Godefroid 1996]. *Persistent/stubborn set POR* [Godefroid 1996; Valmari 1991] uses static analysis to compute, for each visited state, a provably-sufficient subset of the enabled transitions to be explored next, guaranteeing that the unselected transitions do not interfere with the execution of those being selected. Unlike persistent set POR, *sleep set POR* [Godefroid 1991] exploits information about the past of the search instead of the static structure of the program.

To address the inherent imprecision of static analysis and further alleviate state-space explosion incurred by the previous techniques, Flanagan *et al* [2005] proposed *dynamic partial order reduction* (DPOR). DPOR tracks dependencies between threads at runtime, rather than relying on a static conservative estimate of dependencies, to identify backtracking points where alternative paths in the state space need to be explored.

Although effective, the reduction achieved by POR and DPOR is limited by the happens-before relation, meaning that these techniques cannot reduce redundant interleavings that have different happens-before relation.

Schedule bounding. Schedule bounding techniques limit the set of schedules examined during testing, while preserving schedules that are likely to induce bugs. *Depth bounding* [Godefroid 1997] defines a maximum threshold on the number of steps (*i.e.*, synchronization or shared memory accesses) allowed in an execution. *Context bounding* [Qadeer & Rehof 2005; Musuvathi & Qadeer 2007a; Musuvathi, Qadeer, Ball, Basler, Nainar, & Neamtiu 2008; Yu, Narayanasamy, Pereira, & Pokam 2012] bounds the number of context switches in an execution, thus allowing threads to execute an arbitrary amount of steps between context switches. *Delay bounding* [Emmi, Qadeer, & Rakamarić 2011] bounds the amount of times a schedule can deviate from the interleaving defined by a given deterministic scheduler.

Note that, during concurrency testing, the bound on preemptions or delays can be increased iteratively, which permits exploring all schedules in the limit. However, the purpose of schedule bounding techniques is to explore bug-inducing schedules within a reasonable resource budget.

As final remark, it should be mentioned that some recent efforts have also attempted to combine POR with schedule bounding, achieving interesting results [Musuvathi & Qadeer 2007b; Coons, Musuvathi, & McKinley 2013].

2.4.3 Other Approaches

Among the vast body of work on non-systematic bug exposing techniques, we now present some prominent examples as follows.

RaceFuzzer [Sen 2008] attempts to expose harmful data races by randomly deciding the outcome of racing accesses (reported by a static data race detector) during testing runs.

AtomFuzzer [Park & Sen 2008] relies on annotations provided by developers to dynamically check for atomicity violations in multithreaded programs. AtomFuzzer uses a random scheduler to choose an arbitrary thread to run at every program state, favoring interleavings that correspond to atomicity violation execution patterns. DeadlockFuzzer [Joshi, Park, Sen, & Naik 2009] follows the same approach, but is directed towards exposing deadlock bugs.

CTrigger [Park, Lu, & Zhou 2009] handles state space explosion when exploring execution schedules by focusing the search on unserializable interleavings, which have low probability of occurring outside a controlled environment and typically result in atomicity violations.

PCT [Burckhardt, Kothari, Musuvathi, & Nagarakatte 2010] uses a priority-based randomized scheduler that finds concurrency bugs of depth d with a probabilistic guarantee after every run of the program. Here, the depth of a bug is defined as the minimum number of scheduling constraints that are sufficient to trigger the bug. This means that bugs with higher depth will occur in fewer schedules and, therefore, will be harder to find. SKI [Fonseca, Rodrigues, & Brandenburg 2014] extended the PCT algorithm to support interruptions with the goal of testing operating system kernels.

ConSeq [Zhang, Lim, Olichandran, Scherpelz, Jin, Lu, & Reps 2011] computes static slices to identify shared memory reads that are likely to affect potential failing statements (*e.g.*, assertions). ConSeq then records and replays correct runs and injects delays during re-executions in order to exercise suspicious interleavings that may uncover concurrency bugs.

MUVI [Lu, Park, Hu, Ma, Jiang, Li, Popa, & Zhou 2007] focuses on extracting multi-variable access correlations through the analysis of variable access patterns and the examination of what variables are usually read or written together. Doing this, MUVI is then able to find semantic and multi-variable concurrency errors.

MCR [Huang 2015] uses an approach based on constraint solving to expose concurrency bugs. Starting from an initial thread interleaving (dubbed seed interleaving), this system is able to simultaneously check properties and search for failures in all schedules that are equivalent (*i.e.*, with the same causal data-flow) to the seed interleaving. This is achieved by encoding all the possible orderings of thread operations in the seed interleaving as an SMT constraint system, and use a solver to solve the constraints. To further explore the state space, MCR iteratively generates new non-redundant schedules by enforcing read operations to return different values. MCR then uses the newly generated schedules as seed interleavings for subsequent iterations.

Yet another approach to testing multithreaded programs is *test synthesis*. Test synthesis receives a suite of sequential tests as input, and analyzes the traces from these sequential executions with the objective of generating bug-inducing multithreaded tests. Omen [Samak & Ramanathan 2014], Narada [Samak & Ramanathan 2015], and Intruder [Samak, Ramanathan, & Jagannathan 2015] use this approach to automatically synthesize tests aimed at exposing deadlocks, races, and atomicity violations, respectively. Test synthesis techniques, however, still require multithreaded tests to be executed and analyzed with dynamic detectors (*e.g.*, [Flanagan, Freund, & Yi 2008; Park, Vuduc, & Harrold 2010]) to find the concurrency bugs.

2.4.4 Limitations of Previous Work and Opportunities

The bug exposing techniques discussed in this section strive to fully cover the execution state space. However, due to the enormous number of possible schedules, it is often impractical to uncover all concurrency bugs during in-house testing. As a consequence, some failures may surface in production due to latent failing schedules in the shipped software.

To prevent the (potentially) catastrophic effects of in-production failures [Leveson & Turner 1993] and increase the reliability of deployed systems, this thesis proposes *production-guided schedule search*. Production-guided schedule search is a cooperative technique to expose path and schedule dependent failures by exploring variations in schedule and control-flow behavior in non-failing executions observed in deployment. By leveraging information from production runs, as well as symbolic execution and SMT constraint solving, production-guided schedule search is able to synthesize executions to guide the search for concurrency bugs. Moreover, by targeting failing schedules that are similar to non-failing production runs, production-guided search helps cope with the large execution search space, thus being a useful complementary technique to classic in-house testing approaches.

Chapter 5 provides a comprehensive description of production-guided schedule search, as well as Cortex, the system that implements this technique.

Summary

This chapter discussed the background concepts and some of the previous work most related to concurrency debugging. More concretely, we overviewed the state-of-the-art systems in the areas of record and replay, root cause diagnosis, and bug exposing. In the next chapters, we introduce and detail our contributions to each one of these three areas of concurrency debugging.

3 CoopREP: Cooperative Record and Replay of Concurrency Bugs

As discussed in Chapter 2, record and replay (R&R) systems allow re-executing a multi-threaded program deterministically, thus being useful to debug hard-to-reproduce concurrency errors. The main challenge faced by R&R solutions is to achieve low-overhead recording, while providing bug reproducibility guarantees.

We argue that one can further reduce the runtime slowdown incurred by either *order-based* or *search-based* R&R techniques (see Section 2.2.2 in Chapter 2), by devising cooperative logging schemes that exploit the coexistence of multiple instances of the same program. The underlying intuition is simple: to share the burden of logging among multiple instances of the same (buggy) deployed program, by having each instance tracking accesses to a random subset of shared variables. The partial logs collected from production runs can then be statistically analyzed *offline* in order to identify those partial logs whose combination maximizes the chances to successfully replay the bug.

In this chapter, we introduce CoopREP (standing for Cooperative Record and rEPlay), a search-based record and replay framework that leverages cooperative logging performed by multiple user instances.[1] Collaborative partial logging allows to substantially reduce the overhead imposed by the instrumentation of the code, but raises the problem of finding the combination of logs capable of replaying the failure. We tackle this issue by proposing several innovative statistical metrics, as well as two heuristics aimed at guiding the search of partial logs to be combined and used during the replay phase.

Cooperative record and replay is orthogonal to the previous R&R approaches in that one can apply our technique regardless of the kind of execution details captured at runtime. Our experiments show that CoopREP benefits classic order-based solutions by reducing the overhead of capturing the complete execution schedule in a single production run, although potentially

[1]For simplicity of description we use the terms *user instance* of a program and *instance* interchangeably in this chapter.

sacrificing the guarantee of immediate bug replay. For search-based solutions, which already trace partial information at runtime, CoopREP offers the possibility of further reducing the record cost, while still achieving similar bug replay ability.

This chapter is structured as follows. Section 3.1 introduces CoopREP, describing in detail its architecture. Section 3.2 presents the statistical metrics used to measure similarity between partial logs. Section 3.3 describes the two heuristics, which leverage the similarity metrics to merge partial logs and replay concurrency bugs. Section 3.4 shows how the heuristics fit in CoopREP's execution flow. Section 3.5 presents the results of the experimental evaluation. Finally, the chapter concludes with a summary of its main points.

3.1 CoopREP Overview

This section describes CoopREP, a framework that provides failure replication for concurrent programs, based on cooperative recording and partial log statistical combination.

3.1.1 System Architecture

Figure 3.1 depicts the overall architecture of CoopREP, which entails five components: *i)* the logging profile generator, *ii)* the transformer, *iii)* the recorder, *iv)* the statistical analyzer, and *v)* the replayer. We now describe each one of these components in detail.

Logging Profile Generator. The logging profile generator receives the target program as input and identifies the points in the code that should be monitored at runtime. The resulting logging profile is then sent to the transformer. Note that this profile is, in fact, a *partial logging profile*, because it contains only a subset of all the relevant events required to replay the concurrent execution.

Logging profiles can comprise different kinds of information (*e.g.*, function return values, basic blocks, shared accesses). However, we are mainly interested in capturing thread access orderings to synchronization variables (*e.g.*, locks and Java monitors), as well as to class and instance variables that can be manipulated by multiple threads. To locate these *shared program elements* (SPEs), CoopREP uses a static escape analysis denoted *thread-local objects analy-*

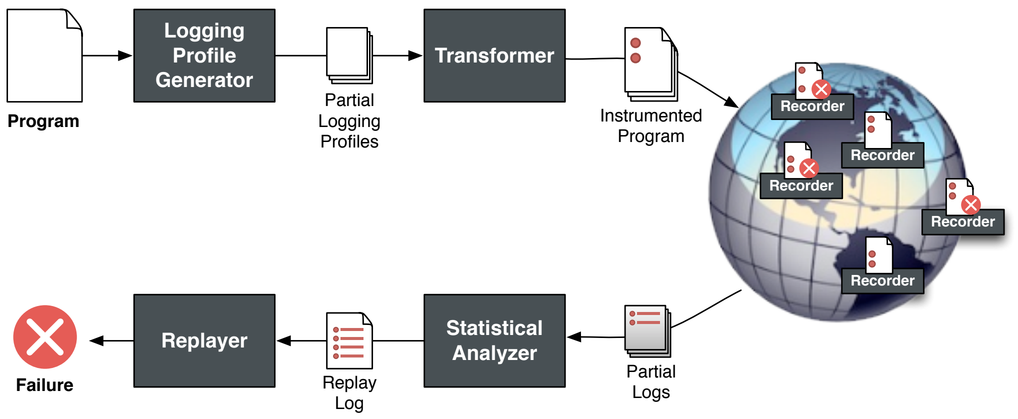


Figure 3.1: Overview of CoopREP's architecture and execution flow.

sis [Halpert, Pickett, & Verbrugge 2007] from the Soot[2] framework. Since accurately identifying shared variables is generally an undecidable problem, this technique computes a sound over-approximation, *i.e.*, every shared access to a field is indeed identified, but some non-shared fields may also be classified as shared [Huang, Liu, & Zhang 2010].

Transformer. The transformer is responsible for instrumenting the program. It consults the logging profile to inject event-handling runtime calls to the recorder component. The transformer also instruments the instructions regarding thread creation and end points of the program. User instances will then execute a given instrumented version of the program, instead of the original one.

In order to ensure consistent thread identification across all executions, we follow an approach similar to that of jRapture [Steven, Chandra, Fleck, & Podgurski 2000]. The underlying idea of this approach is that each thread should create its children threads in the same order, even though there may not be a consistent global order among all threads.

Concretely, we instrument the thread creation points and modify the new Java thread identifiers by new identifiers based on the parent-children order relationship. To this end, each thread maintains a local counter to store the number of children it has forked so far. Whenever a new child thread is forked, its identifier is defined by a string containing the parent thread's ID concatenated with the value of the local counter at that moment. For instance, if a thread t_i forks its j-th child thread, which in turn forks its k-th thread, then the latter thread's identifier will be $t_{i:j:k}$.

[2]http://www.sable.mcgill.ca/soot

Recorder. There is a recorder per user instance. The recorder monitors the execution of the instance and captures the relevant information into a log file. Then, when the execution ends or fails, the user sends its *partial log* to the statistical analyzer.

CoopREP can be configured to trace different kinds of information, but in this dissertation we employ the technique used by LEAP [Huang, Liu, & Zhang 2010] to track the thread interleaving.[3] In particular, we associate each shared program element (SPE) with an *access vector* that stores, during the production run, the identifiers of the threads that read from/write to that SPE. For instance, if a shared variable x is accessed twice by a thread *T1* and then once by a thread *T2* during an execution, then the access vector for x is $[1, 1, 2]$.

Using this technique, one gets several vectors with the order of the thread accesses performed on individual shared variables, instead of a single global-order vector. This provides lightweight recording, but relaxes faithfulness in the replay, as it allows the re-ordering of non-causally related memory accesses during the re-execution of the program. Still, it has been shown that this approach does not affect the correctness of the replay (a formal proof of the soundness of this statement can be found in [Huang, Liu, & Zhang 2010]).

Conversely to LEAP, CoopREP's recorder does not log access vectors for all the SPEs of the program. Instead, each user traces accesses to only a subset of the total amount of SPEs, as defined in the partial logging profile. In addition to access vectors, each CoopREP's partial log also contains a flag indicating the success or failure of the execution (successful executions can be useful for the statistical analysis, as we will see in Section 3.4).

Assuming that the program is executed by a large population of users, this mechanism allows not only gathering access vectors for the whole set of SPEs with high probability, but also reduce the performance overhead incurred by each user instance with respect to full logging.

Statistical Analyzer. The statistical analyzer is responsible for collecting and analyzing the partial logs recorded during production runs (from both successful and failing executions) in order to produce a complete replay trace, capable of yielding an execution that triggers the bug observed at runtime. The statistical analyzer comprises two sub-components: the *candidate generator* and the *replay oracle*, as depicted in Figure 3.2.

[3]In Section 3.5, we also evaluate CoopREP using a search-based record and replay approach, in addition to the order-based approach used by LEAP.

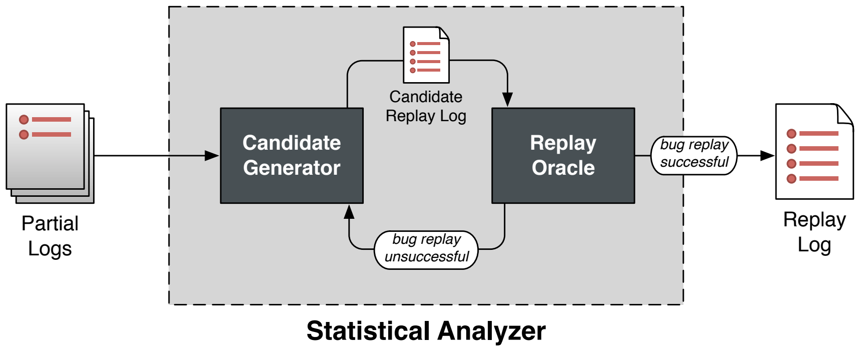


Figure 3.2: Detailed view of the statistical analyzer component.

The candidate generator employs one of our two novel heuristics (see Section 3.3) to identify similarities in partial logs and pinpoint those that are more likely to be successfully combined in order to reproduce the failure. In other words, the candidate generator tries to aggregate partial logs into smaller *clusters*, such that each cluster contains logs resulting from *similar* production runs. The similarity among production runs is evaluated by means of several statistical metrics (see Section 3.2). The rationale is that partial logs resulting from similar executions are likely to provide compatible information with respect to the thread interleaving that leads to the bug.[4] To generate a complete replay log, the statistical analyzer combines the access vectors from the partial logs within the same cluster.

The resulting candidate replay log (containing access vectors for all the SPEs of the program) is then sent to the *replay oracle*. The replay oracle is responsible for checking the effectiveness of the replay log in terms of bug reproducibility. Concretely, the oracle enforces an execution that respects the schedule defined in the replay candidate trace and checks whether the error is triggered or not. This is achieved by controlling the execution interleaving with the help of semaphores for suspending and resuming threads on demand.

In case the bug is not reproduced, the replay oracle asks the candidate generator for another candidate trace, and the procedure is repeated until the bug is successfully replayed or until the maximum number of attempts to do so is reached. This search procedure is detailed in Section 3.4.

Considering that the candidate replay log is composed of access vectors collected from

[4]In this thesis, we assume that all partial logs from failing executions refer to the same bug. In practice, one may cope with applications that suffer of multiple bugs by distinguishing them using additional meta data, such as the line of code in which the bug manifested or the type of exception that the bug generated.

independent executions, the resulting execution schedule might be infeasible, meaning that the replay oracle will fail to fully enforce the thread execution order specified in the replay log. In this case, the re-execution of program will hang, as none of the threads will be allowed to perform its subsequent access on an SPE. CoopREP copes with this issue by using a simple, yet effective, timeout mechanism, which terminates immediately the execution replay as soon as it detects that all threads are prevented from making progress.

CoopREP also needs to deal with the case in which the the bug is not triggered, although the replay oracle is able to fully enforce the thread interleaving indicated in the replay log. For crash failures, CoopREP simply detects the absence of the exception. On the other hand, in order to detect generic incorrect results, CoopREP requires programmers to provide information regarding the failure symptoms (*e.g.*, by specifying predicates on the application's state that can be dynamically checked via assertions).

Replayer. Once the statistical analyzer produces a feasible replay log, the developers can then use the replayer to re-execute the application and observe the bug being deterministically triggered in every run. This allows developers to re-enable *cyclic debugging*, as they can run the program repeatedly in an effort to incrementally refine the clues regarding the error's root cause.

Observing the architecture depicted in Figure 3.1, one can identify two main challenges in the cooperative record and replay approach: *i) how to devise the partial recording profiles*, and *ii) how to combine the partial logs recorded during production runs to generate a feasible failure-inducing replay trace.* The next sections show how we address these two issues, giving special emphasis to the techniques devised to merge partial logs.

3.1.2 Partial Log Recording

CoopREP is based on having each user instance recording accesses to only a fraction of the entire set of SPEs in the program. The subset of SPEs to be traced is defined by the logging profile generator prior to the instrumentation of the target program instances. The logging profile generator relies on a probabilistic scheme that selects a subset of the total SPEs using a uniform random distribution. The cardinality of the set of SPEs logged by CoopREP is a parameter configurable by the developer. This approach has the benefit of being extremely lightweight and ensuring statistical fairness, *i.e.*, on average, all SPEs have the same probability of being traced by a user instance. Thus, each instance has the same probability of logging SPEs that are very frequently/rarely accessed. As we will show in Section 3.5.5, this factor has a significant impact on the actual recording overhead of a given instance.

We believe that it would be relatively straightforward to increase the flexibility of the information recorded, either by re-instrumenting the program at the user's site or by extending the current recording framework to support SPE adaptive tracing (based, for example, on hints provided by the developer or on code analysis techniques [Machado, Romano, & Rodrigues 2013]).

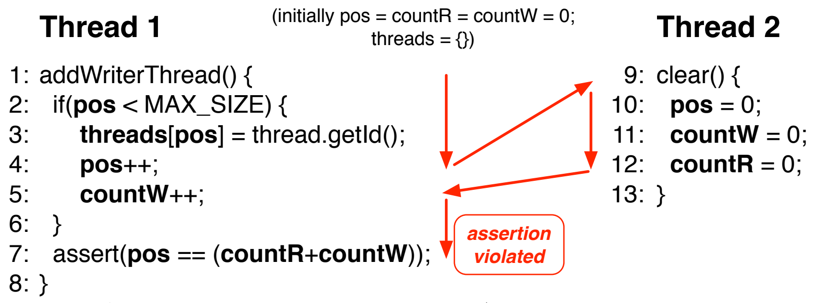
3.1.3 Merge of Partial Logs

Another major challenge of using partial recording (and the main focus of the work in this chapter) is how to combine the collected partial logs in such a way that the resulting thread interleaving leads to a feasible execution, capable of reproducing the bug during the replay phase.

In general, the following factors can make partial log merging difficult to achieve: *i)* the bug can result from a complex interleaving of multiple threads; *ii)* the combination of access vectors from different *failing* executions may enforce a thread schedule that leads to a *non-failing* replay execution; *iii)* the combination of access vectors from different failing executions may enforce a thread order that leads to an *infeasible* replay execution; *iv)* the number of possible combinations of access vectors grows very quickly (in the worst case factorially) with the number of partial logs available, which renders blind brute-force search approaches impractical in such a vast space.

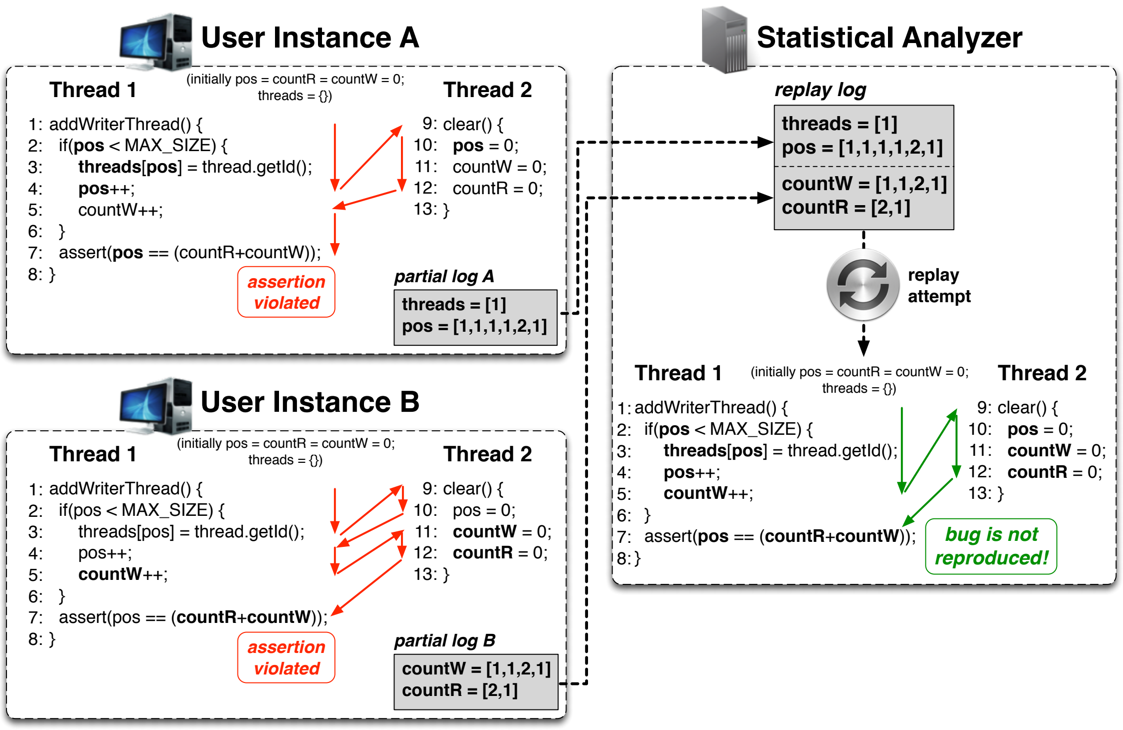
To exemplify factors *i)* and *ii)*, consider the multithreaded program with a concurrency bug depicted in Figure 3.3. This program has two threads and four shared variables (*pos*, *countR*, *countW*, and *threads*). If the program follows the execution indicated by the arrows, it will violate the assertion at line 7. This error is an atomicity violation: thread 1 increments the position counter after inserting a value in the array (`pos++` at line 4) but, before incrementing the counter *countW* (line 5), it is interleaved by thread 2, which sets both variables to 0 (lines 10 and 11). As such, when thread 1 reaches the assertion, the condition will fail because $pos = 0$ and $countW = 1$.

Let us now consider the scenario depicted in Figure 3.4. We can see that both user instances



```
Thread 1                                  Thread 2
1: addWriterThread() {                    9:  clear() {
2:   if(pos < MAX_SIZE) {                 10:   pos = 0;
3:     threads[pos] = thread.getId();     11:   countW = 0;
4:     pos++;                             12:   countR = 0;
5:     countW++;                          13: }
6:   }
7:   assert(pos == (countR+countW));
8: }
```

Figure 3.3: Example of an atomicity violation bug. Accesses to shared variables are depicted in bold.



```
Thread 1                                  Thread 2
1: addWriterThread() {                    9:  clear() {
2:   if(pos < MAX_SIZE) {                 10:   pos = 0;
3:     threads[pos] = thread.getId();     11:   countW = 0;
4:     pos++;                             12:   countR = 0;
5:     countW++;                          13: }
6:   }
7:   assert(pos == (countR+countW));
8: }
```

```
partial log A
threads = [1]
pos = [1,1,1,1,2,1]

partial log B
countW = [1,1,2,1]
countR = [2,1]

replay log
threads = [1]
pos = [1,1,1,1,2,1]
countW = [1,1,2,1]
countR = [2,1]
```

Figure 3.4: Example of a scenario where two failing partial logs originate a replay log that does not trigger the bug.

record partial logs corresponding to failing executions (user A traces accesses to SPEs *threads* and *pos*, whereas user B records $countW$ and $countR$). However, the complete replay log that results from merging the two partial logs enforces a successful execution, which does not reproduce the bug observed in the production run.

In the following sections, we show how CoopREP addresses the challenges related to both partial log incompatibility and vastness of the search space. First, we propose three metrics to compute similarity between partial logs. Then, we describe two novel heuristics that exploit these metrics to generate a failure-inducing replay log in a small number of attempts.

3.2 Similarity Metrics

CoopREP uses *similarity metrics* to quantify the amount of information that different partial logs have in common. Similarity can hold values in the [0,1] range and the rationale for the classification of the similarity between two partial logs is related to their number of SPEs with similar access vectors (*i.e.*, SPEs for which the logs have observed exactly the same or a very alike thread interleaving at runtime). Hence, the more SPEs with equal access vectors the partial logs have, the closer to 1 their similarity is.

We propose three metrics to measure partial log similarity: *Plain Similarity* (*PSIM*), *Dispersion Similarity* (*DSIM*), and *Dispersion Hamming Similarity* (*DHS*). Briefly, *PSIM* and *DSIM* differ essentially on the weight given to the SPEs of the program. On the other hand, *DSIM* and *DHS* weigh SPEs in the same way, but *DHS* uses a more fine-grained method than *DSIM* to compare access vectors.

We now describe each similarity metric in detail. Table 3.1 summarizes the formal notation that will be used to define the metrics.

Table 3.1: Notation used to define the similarity metrics.

Notation	**Description**
$\mathcal{S}$	Set of all SPEs of the program.
$\mathcal{S}_l$	Subset of SPEs traced by partial log l.
$\mathcal{AV}$	Set of all access vectors recorded for all SPEs by all partial logs. ($\mathcal{AV}^*$ indicates the set of *different* access vectors recorded for all SPEs by all partial logs.)
$\mathcal{AV}_s$	Set of all access vectors recorded for SPE s by all partial logs. ($\mathcal{AV}_s^*$ indicates the set of *different* access vectors recorded for SPE s by all partial logs.)
$l.s$	Access vector recorded for SPE s by partial log l.
$Common_{l_0,l_1} = \{s \mid s \in \mathcal{S}_{l_0} \cap \mathcal{S}_{l_1}\}$	Intersection of SPEs recorded by both partial logs l_0 and l_1.
$Equal_{l_0,l_1} = \{s \mid s \in Common_{l_0,l_1} \wedge l_0.s = l_1.s\}$	Intersection of SPEs with identical access vectors recorded by both partial logs l_0 and l_1.
$Diff_{l_0,l_1} = \{s \mid s \in Common_{l_0,l_1} \wedge l_0.s \neq l_1.s\}$	Intersection of SPEs with different access vectors recorded by both partial logs l_0 and l_1.

3.2.1 Plain Similarity

Let l_0 and l_1 be two partial logs, their *Plain Similarity* (*PSIM*) is given by the following equation:

$$PSIM(l_0, l_1) = \frac{\#Equal_{l_0,l_1}}{\#\mathcal{S}} \times \left(1 - \frac{\#Diff_{l_0,l_1}}{\#\mathcal{S}}\right) \tag{3.1}$$

where $\#Equal_{l_0,l_1}$, $\#\mathcal{S}$, and $\#Diff_{l_0,l_1}$ denote the cardinality of the sets $Equal_{l_0,l_1}$, $\mathcal{S}$, and $Diff_{l_0,l_1}$, respectively. The first factor of the product computes the fraction of SPEs that have equal access vectors. The second factor *penalizes* the similarity value in case the partial logs have common SPEs with different access vectors (otherwise, $\#Diff_{l_0,l_1}$ will be 0 and the value of the first factor will not be affected).

Note that *PSIM* is equal to 1 only when both logs are complete and identical, *i.e.*, when they have recorded access vectors for all the SPEs of the program ($\mathcal{S}_{l_0} = \mathcal{S}_{l_1} = \mathcal{S}$) and those access vectors are equal for both logs ($l_0.s = l_1.s, \forall s \in \mathcal{S}$). This implies that, for any two partial logs, their Plain Similarity will always be smaller than 1. However, the greater this value is, the more similar the two partial logs are.

Finally, it should be noted that functions $Equal_{l_0,l_1}$ and $Diff_{l_0,l_1}$ can be implemented very efficiently by comparing pairs of partial logs using the hashes of their access vectors.

3.2.2 Dispersion Similarity

In opposition to *PSIM*, *Dispersion Similarity* (*DSIM*) does not assign the same weight to all SPEs. Instead, it takes into account whether a given SPE exhibits many different access vectors across the collected partial logs or not. We refer to this property as *dispersion*.

The dispersion of an SPE depends, essentially, on the number of *concurrent and unsynchronized accesses to shared memory* existing in the target program, as well as on the number of threads. The reason is twofold. On the one hand, the thread interleaving of a given concurrent execution is heavily affected by memory races (see Section 2.1). On the other hand, the more threads the program has, the greater the number of possible outcomes for each memory race is. For instance, in Figure 3.3, we can see that shared variables *pos*, *countW*, and *countR* are subject to data races, as their accesses in both threads are not synchronized. As a consequence, the access vectors for these variables may easily contain different execution orders.

To capture the dispersion of a particular SPE, comparatively to the other SPEs of the program, we define an additional metric denoted *overall-dispersion*. This metric corresponds to the ratio between the number of *different* access vectors logged for a given SPE and the whole universe of *different* access vectors collected for all SPEs (across all instances). More concretely, we compute the overall-dispersion of an SPE s as follows:

$$\mathit{overallDisp}(s) = \frac{\#\mathcal{AV}^*_s}{\#\mathcal{AV}^*} \tag{3.2}$$

Note that the result of this metric varies within the range [0,1]. Also, note that *overallDisp* assigns larger dispersion to SPEs that tend to exhibit different access vectors across the collected partial logs. As an example, assume that we have two SPEs – x and y – and two partial logs containing identical access vectors for SPE x (say x_1) and different access vectors for SPE y (say y_1 and y_2). In this case we would have $\mathit{overallDisp}(x) = \frac{1}{3}$, and $\mathit{overallDisp}(y) = \frac{2}{3}$.

With overall-dispersion, we now define Dispersion Similarity as follows. Let l_0 and l_1 be two partial logs. The Dispersion Similarity (*DSIM*) between l_0 and l_1 is given by the equation:

$$\mathit{DSIM}(l_0, l_1) = \sum_{x \in \mathit{Equal}_{l_0,l_1}} \mathit{overallDisp}(x) \times \left(1 - \sum_{y \in \mathit{Diff}_{l_0,l_1}} \mathit{overallDisp}(y)\right) \tag{3.3}$$

Using overall-dispersion as weight in *DSIM*, we can bias the selection towards pairs of logs that have similar access vectors for SPEs that are often subject to different access interleavings. The rationale is that two partial logs having in common the same “rare” access vector, for a given SPE, are more likely to have been captured from compatible production runs.

3.2.3 Dispersion Hamming Similarity

Both *PSIM* and *DSIM* metrics rely on a binary comparison of access vectors, *i.e.*, the access vectors are either fully identical or not (see Equations 3.1 and 3.3). However, it can happen that two access vectors have the majority of their values identical, differing only on a few positions. For instance, let us consider three access vectors: $A = [1, 1, 1, 1]$, $B = [1, 1, 1, 0]$, and $C = [0, 0, 0, 0]$. It is obvious that, despite being all different, A is more similar to B than to C.

To capture the extent to which two partial logs differ, we defined another similarity metric, called *Dispersion Hamming Similarity (DHS).* This metric is defined as follows.

Let l_0 and l_1 be two partial logs, their Dispersion Hamming Similarity is given by the following equation:

$$DHS(l_0, l_1) = \sum_{s \in Common_{l_0,l_1}} hammingSimilarity(l_0.s, l_1.s) \times overallDisp(s) \tag{3.4}$$

where $Common_{l_0,l_1}$ is the set of SPEs recorded by both partial logs l_0 and l_1 (see Table 3.1), $hammingSimilarity(l_0.s, l_1.s)$ gives the value (normalized to range [0,1]) of the *Hamming similarity* of the access vectors logged for SPE s in partial logs l_0 and l_1, and $overallDisp(s)$ measures the relative weight of SPE s in terms of its overall-dispersion (see Equation 3.2).

Hamming similarity is a variation of the *Hamming distance* [Hamming 1950] used in several disciplines, such as telecommunication, cryptography, and information theory. Hamming distance is employed to compute distance between strings and can be defined as follows: given two strings s_1 and s_2, their Hamming distance is the number of positions which differ in the corresponding symbols. In other words, it measures the minimum number of substitutions needed to transform s_1 into s_2. On the other hand, Hamming similarity corresponds to the number of *equal* positions between s_1 and s_2.

As previously referred, in the context of CoopREP, we apply Hamming similarity to access vectors rather than strings.[5] As such, it might happen that two access vectors being compared have different sizes. We address this issue by augmenting the shorter vector with a suffix of N synthesized thread IDs (different from the ones in the larger vector), where N is equal to the size difference of the two access vectors.

[5]Additional metrics based on *edit distance* would be potentially usable, but, for the work in this chapter, we opted for metrics with more efficient computation.

3.2.4 Trade-offs among the Metrics

In this section, we compare the three partial log similarity metrics in terms of their time complexity and accuracy.[6] The need for greater accuracy can arise when the overall-dispersion weight values are not well balanced across the SPEs. This happens when some SPEs have identical access vectors in almost every partial log, while other SPEs exhibit access vectors that are equal (or very alike) only in a small subset of partial logs. As such, the ability to accurately identify those small subsets of similar partial logs becomes crucial to generate a feasible replay log.

Let S be the number of SPEs in a log and L the maximum length of an access vector recorded. Beginning with Plain Similarity, this metric compares the hashes of all overlapping SPEs between the two partial logs. Since the metric has to find the set of overlapping SPEs between the two logs in order to perform the hash comparison, it has a time complexity of $O(S^2)$. In terms of accuracy, one may argue that *PSIM* provides poor accuracy, since it assumes that every SPE has the same importance. Consequently, *PSIM* can be considered to be a simple and baseline metric to capture partial log similarity.

Dispersion Similarity also executes in $O(S^2)$ time, because, like *PSIM*, it solely compares hashes[7]. However, *DSIM* provides more accuracy than *PSIM*, as it assigns different weights to SPEs.

Finally, Dispersion Hamming Similarity is expected to further improve the accuracy provided by *DSIM*, by allowing to compare access vectors at a finer granularity, and not just by means of a binary classification (*i.e.*, equal or different). Unfortunately, this increase in accuracy comes at the cost of a larger execution time, because *DHS* requires comparing each position of the access vectors instead of just comparing their hashes, as done in *PSIM* and *DSIM*. As such, *DHS* executes with $O(S^2 \cdot L)$ complexity.

In sum, each one of the three metrics represent a particular trade-off between computation time and accuracy. Section 3.5.2 provides an empirical comparison of the similarity metrics to

[6]We consider accuracy to be related to the granularity degree used to measure similarity between access vectors. For example, measuring similarity at the level of the access vector as a whole (*i.e.*, by comparing their hashes) is more coarse-grained than measuring similarity at the level of each position of the access vector. Therefore, the former is less accurate than the latter.

[7]We are disregarding the time to calculate SPEs' overall-dispersion, as this computation is performed only once, at the beginning of the statistical analysis.

further assess the benefits and limitations of each one.

3.3 Heuristics to Merge Partial Logs

As mentioned in Section 3.1.3, testing all possible combinations of partial logs to find one capable of successfully reproducing the non-deterministic bug is impractical. To cope with this issue, we have developed two novel heuristics, denoted *Similarity-Guided Merge (SGM)* and *Dispersion-Guided Merge (DGM)*. The goal of the heuristics is to optimize the search among all possible combinations of partial logs (from failing executions) and guide it towards combinations that result in an effective (*i.e.*, bug-inducing) replay log.

Both heuristics operate by systematically picking a partial log to serve as basis to reconstruct a complete trace of the failing execution. To find this *base partial log*, the heuristics rank the partial logs in terms of *relevance*. Thereby, the heuristics differ essentially in the strategy employed to compute and sort the partial logs according to their relevance.

SGM considers the most relevant partial logs to be the ones that maximize the chances of being completed with information from other similar partial logs. The rationale is that if there is a group of partial logs with high similarity among themselves, then it should probably mean that they were captured from compatible production runs.

On the other hand, *DGM* focuses on finding partial logs whose SPEs account for higher overall-dispersion. The rationale is that, by ensuring the compatibility of the *most disperse* SPEs (because they were traced from the same production run), *DGM* maximizes the probability of completing the missing SPEs with access vectors that tend to be common across the partial logs. The downside of *DGM* is that its most relevant partial logs are less likely to have other potential similar partial logs to be combined with. Exactly because the dispersion of the SPEs in the relevant partial log is high, the access vectors for these SPEs are rarely occurring in other logs and, therefore, are not good matching points. This fact can hinder the effectiveness of the *DGM* heuristic when the capacity of the partial logs is not sufficient to encompass all the SPEs with high overall-dispersion.

In the following sections, we further describe *SGM* and *DGM* by detailing their two operation phases: *computation of relevance* and *generation of the next candidate replay log*. Next, we discuss the advantages and drawbacks of each heuristic. Finally, we present a general algorithm

Algorithm 1: sgm.computeRelevance

```
Input: $PLogs$: set with partial logs collected during production runs.
       $simMetric$: metric used to measure similarity between partial logs.
Output: $MostRelevant$: sorted set containing complete logs in descending order of
        relevance.
1  $MostRelevant = \emptyset$
2  for $baseLog \in PLogs$ do
3      $simSum = 0$
4      $simLogs = 0$
5      $KNN \leftarrow simMetric$.computeKNN($baseLog$)
6      while $KNN$.hasNext() do
7          $nLog \leftarrow KNN$.next()
8          $simSum$ += $simMetric$.measureSimilarity($baseLog$, $nLog$)
9          $baseLog$.fillMissingSPEs($nLog$)
10     end
11     $relevance = simSum / KNN$.size()
12     $MostRelevant$.addBasedOnRelevance($baseLog$, $relevance$)
13 end
14 return $MostRelevant$
```

Algorithm 2: sgm.genNextReplayLog

```
Input: $MostRelevant$: sorted set containing complete logs in descending order of
       relevance.
Output: $replayLog$: a complete execution trace ready to be replayed.
1  $replayLog \leftarrow MostRelevant$.next()
2  return $replayLog$
```

(employed by the statistical analyzer) that can leverage any of the two heuristics to successfully replay concurrency bugs.

3.3.1 Similarity-Guided Merge

Computation of Relevance. *SGM* classifies a partial log as being relevant if it is considered similar to many other partial logs. More formally, this corresponds to computing, for each partial log l, the following value of *Relevance*:

$$Relevance(l) = \frac{\sum_{l' \in kNN_l} Similarity(l, l')}{\#kNN_l} \tag{3.5}$$

where kNN_l is a set containing l's *k-nearest neighbors* (in terms of similarity) that suffice to fill l's missing SPEs, and $\mathit{Similarity}(l, l')$ is one of the three possible similarity metrics (*PSIM*, *DSIM*, or *DHS*). Note that kNN_l contains, at most, one unique partial log (the most similar) per missing SPE in l, which means that $\#\mathit{kNN}_l$ will always be smaller or equal than the number of missing SPEs in l. Also, if several partial logs have the same similarity with respect to l and fill the same missing SPE, kNN_l will pick one of them at random.

Unfortunately, in order to be able to compute $\mathit{Relevance}(l)$, *SGM* needs to measure the similarity between all possible pairs of partial logs. This happens because, for any partial log l, the composition of set kNN_l is only accurately known after measuring the similarity between l and all the remaining partial logs. The whole process to compute partial logs' relevance in *SGM* is presented in Algorithm 1.

For a given base partial log *baseLog*, *SGM* starts by computing the set of k-nearest neighbors *KNN* (line 5). Then, *SGM* fills the missing SPEs in *baseLog* with the information recorded by the partial logs in *KNN*, while calculating the corresponding relevance of the base partial log (lines 6-11). Note that the relevance of *baseLog* corresponds to the average similarity between *baseLog* and the partial logs contained in *KNN* (line 11). Finally, *SGM* stores the newly complete *baseLog* in *MostRelevant*, which is a set containing the complete logs sorted in descending order of their relevance (line 12).

Generation of the Next Candidate Replay Log. The generation of the next candidate replay log in *SGM* is straightforward. Since, during the computation of the relevance values, *SGM* already combines partial logs and generates complete execution traces (line 9 in Algorithm 1), it just needs to iteratively pick replay logs from the set of most relevant logs. Algorithm 2 illustrates this procedure.

3.3.2 Dispersion-Guided Merge

Computation of Relevance. *DGM* classifies a partial log as being relevant if it contains SPEs with high overall-dispersion. The *Relevance* of a partial log l in *DGM* is computed as follows:

$$\mathit{Relevance}(l) = \sum_{s \in \mathcal{S}_l} \mathit{overallDisp}(s) \tag{3.6}$$

By identifying partial logs that have traced the *most disperse* SPEs, *DGM* attempts to

Algorithm 3: dgm.computeRelevance

```
Input: PLogs: set with partial logs collected during production runs.
Output: MostRelevant: set containing partial logs sorted in descending order of their
        relevance.

1 MostRelevant = ∅
2 for baseLog ∈ PLogs do
3     dispSum = 0
4     for spe ∈ baseLog.getSPEs() do
5         dispSum += spe.getOverallDisp()
6     end
7     MostRelevant.addBasedOnRelevance(baseLog,dispSum)
8 end
9 return MostRelevant
```

Algorithm 4: dgm.genNextReplayLog

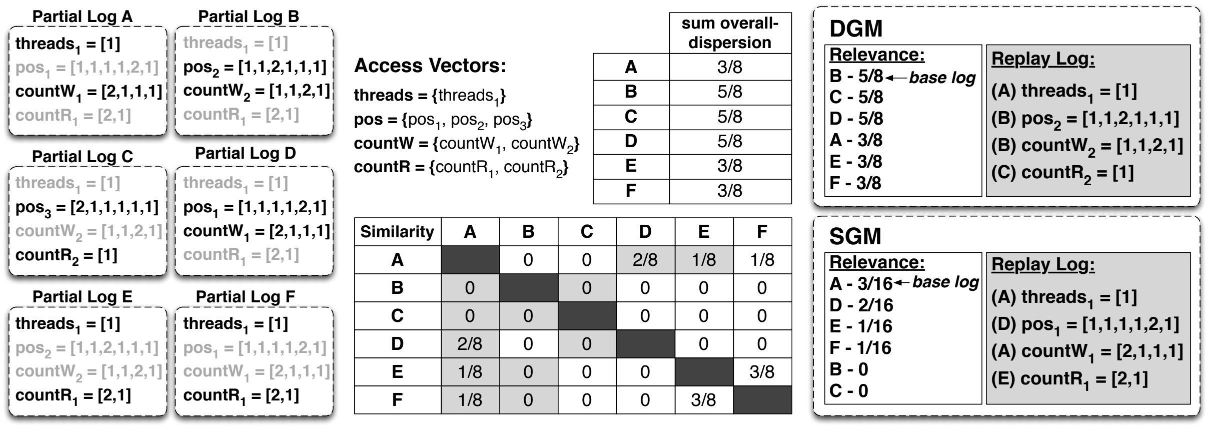
```
Input: MostRelevant: set containing partial logs sorted in descending order of their
       relevance.
       simMetric: metric used to measure similarity between partial logs.
Output: replayLog: a complete execution trace ready to be replayed.

1 replayLog ← MostRelevant.next()
2 KNN ← simMetric.computeKNN(replayLog)
3 while !replayLog.isComplete() ∧ KNN.hasNext() do
4     nLog ← KNN.next()
5     replayLog.fillMissingSPEs(nLog)
6 end
7 return replayLog
```

decrease the chances of completing the missing SPEs with non-compatible access vectors. Algorithm 3 shows how *DGM* computes relevance for a set of partial logs.

DGM computes the relevance of a given base partial log *baseLog* by summing the overall-dispersion values of the SPEs recorded by *baseLog* (lines 2-6). *DGM* then stores the base partial log in the sorted set *MostRelevant* according to the computed relevance (line 7).

Generation of the Next Candidate Replay Log. Algorithm 4 describes how *DGM* generates new replay logs. In opposition to *SGM*, *DGM* does not combine partial logs when computing their relevance. As a consequence, after picking the base partial log to build the next candidate replay trace, *DGM* has still to identify its k-nearest neighbors (line 2). Once the set of k-nearest neighbors *KNN* is computed, *DGM* then fills the missing SPEs in the base partial log with the information recorded by the logs in *KNN* (lines 3-6).



	sum overall-dispersion
A	3/8
B	5/8
C	5/8
D	5/8
E	3/8
F	3/8

Similarity	A	B	C	D	E	F
A	■	0	0	2/8	1/8	1/8
B	0	■	0	0	0	0
C	0	0	■	0	0	0
D	2/8	0	0	■	0	0
E	1/8	0	0	0	■	3/8
F	1/8	0	0	0	3/8	■

Figure 3.5: Example of the *modus operandi* of the two heuristics (*SGM* and *DGM*), using *DSIM* as similarity metric. The gray lines in the partial logs indicate the access vectors that were observed in the corresponding production run but not traced.

As final remark, note that the similarity metrics are orthogonal to the heuristics. For instance, *DGM*, although using overall-dispersion to calculate the relevance values, can employ any similarity metric (*i.e.*, *PSIM*, *DSIM*, or *DHS*) to compute the set of nearest neighbors for the base partial log.

3.3.3 Example

To better illustrate how the two heuristics operate, let us see an example. On the left side of Figure 3.5, we have six failing partial logs collected from user instances running the program in Figure 3.3. Notice that each partial log recorded only two of the four SPEs of the program (traced SPEs are depicted in black and the observed, but not recorded, SPEs are represented in gray). The subscript number in each SPE identifier indicates the hash of its respective access vector (*e.g.*, $pos_1 \neq pos_2$).

As shown at the center of Figure 3.5, there are eight different access vectors in total: $threads_1$, pos_1, pos_2, pos_3, $countW_1$, $countW_2$, $countR_1$, and $countR_2$. Therefore, we obtain the following values of dispersion for the four SPEs: $overallDisp(threads) = 1/8$, $overallDisp(pos) = 3/8$, $overallDisp(countW) = 2/8$, and $overallDisp(countR) = 2/8$.

Let us consider *DSIM* as similarity metric in this example (see Equation 3.3). We now describe how *SGM* and *DGM* operate to produce a complete replay log using the partial logs depicted in Figure 3.5.

SGM. This heuristic first calculates the similarity among the partial logs (see the similarity matrix at the center-bottom of Figure 3.5). For example, both logs A and D traced the same access vector $countW_1$ for their single overlapping SPE *countW*, hence *DSIM(A,D)* = *DSIM(D,A)* = 2/8.

Next, *SGM* computes the k-nearest neighbors for all partial logs and ranks them according to their value of relevance. The k-nearest neighbors for each log are marked with shaded cells in the similarity matrix at the center-bottom of Figure 3.5. For instance, the nearest neighbors for partial log A are partial logs D and E, because these are the logs exhibiting higher similarity to A that suffice to fill A's missing SPEs.

Recall from Section 3.3.1 that the most relevant log in *SGM* is considered to be the one with highest average similarity to the partial logs in the nearest neighbors set. The box at the bottom-right of Figure 3.5 shows the logs sorted in descending order of their relevance for *SGM*. As we can see, log A is the most relevant one (note that *Relevance(A)* = $(2/8 + 1/8)/2 = 3/16$) and, therefore, is chosen as the base partial log. The first candidate replay log in *SGM* will thus be composed by partial log A augmented with access vector pos_1 from D and $countR_1$ from E.

DGM. This heuristic considers the relevance of partial logs to be the sum of their SPEs' overall-dispersion. The table at the center-top of Figure 3.5 reports the relevance value for the partial logs in *DGM*. Since there are several partial logs with the highest value of relevance, namely B, C, and D, *DGM* simply picks one at random. Let us follow the alphabetical order and pick B as the base log. Given that B does not have any particular similar partial log (the similarity between B and any other partial log is 0, as reported in the similarity matrix in Figure 3.5), *DGM* fills B's missing SPEs with access vectors from other partial logs randomly picked (say A and C). Thereby, the candidate replay log for *DGM* will be composed by partial log B augmented with access vector $threads_1$ from A and $countR_2$ from C.

Finally, note that both replay logs produced by the heuristics allow for triggering the bug.

3.3.4 Trade-offs among the Heuristics

We now analyze the complexity of the heuristics in terms of the time they take to compute the relevance of partial logs and generate a replay log.

Recall from Section 3.2.4 that measuring the similarity between two partial logs requires either S^2 steps or $(S^2 \cdot L)$ steps depending on the similarity metric used (where S is equal to the number of SPEs in a logs and L indicates the maximum number of entries of the access vectors being compared). For the sake of simplicity, let us denote the complexity of computing the similarity between two partial logs by *simMetric(m)*, where $m \in \{$*PSIM, DSIM, DHS*$\}$ is one of the three similarity metrics presented in Section 3.2. In addition, let N be the number of partial logs collected across all user instances.

To compute the relevance of a given partial log, the *SGM* heuristic needs to calculate the similarity between that log and the $N-1$ remaining logs, which results in complexity $O(N \cdot \textit{simMetric(m)})$. Consequently, computing the relevance for all partial logs will have time complexity $O(N^2 \cdot \textit{simMetric(m)})$. Furthermore, inserting a log in the sorted set of the most relevant logs requires $log(N)$ steps. Therefore, the overall complexity of *SGM*, using similarity metric m for sorting partial logs according their relevance, is $O(N^2 \cdot \textit{simMetric(m)} \cdot log(N))$.

Regarding the generation of a new replay log, *SGM* incurs $O(1)$ complexity as it simply needs to return the next most relevant log from the sorted set (see Algorithm 2).

On the other hand, the *DGM* heuristic computes each partial log's relevance by simply summing the overall-dispersion of its SPEs, which requires S steps. Considering all partial logs, this procedure is executed N times. Therefore, the complexity of *DGM* to compute the set of most relevant partial logs is $O(N \cdot S \cdot log(n))$. In turn, when generating a replay log, *DGM* has also to measure the similarity between the base partial log and all the other logs. This requires computing the similarity between $N-1$ partial logs, which results in an overall complexity of $O(N^2 \cdot \textit{simMetric(m)})$ in the worst case scenario.

We now compare the two heuristics in terms of their benefits and limitations in replaying concurrency bugs via partial log combination.

SGM has the advantage of quickly identifying *clusters* of similar partial logs. This is useful when the majority of partial logs exhibits SPEs with different access vectors. In this scenario, finding a couple of partial logs with identical (or very alike) access vectors means that there is a high probability that their combination yields a feasible and effective replay log. The drawback of this heuristic is its additional complexity to rank the partial logs in terms of relevance.

DGM, on the other hand, has the key benefit of quickly computing the relevance of partial logs. Moreover, it is also effective when the number of SPEs traced by each partial log is sufficient to encompass all SPEs with high overall-dispersion. This means that, for a given base partial log, the remaining non-recorded SPEs should exhibit common access vectors. Hence, the replay log can be trivially produced by simply merging the base log with any other partial log (the example for *DGM* in Figure 3.5 illustrates this scenario). However, when the number of high-disperse SPEs is greater than the size of the partial log, *DGM* is not as effective as *SGM* because the partial logs with higher SPE overall-dispersion tend to have few similar partial logs (*e.g.*, partial log *B* in Figure 3.5 does not have similar partial logs at all).

3.4 Statistical Analyzer Execution Mode

After discussing the similarity metrics and the heuristics, we now describe how all these components fit together in CoopREP's execution flow. Algorithm 5 depicts the pseudo-code of the general algorithm employed by the *statistical analyzer* (see Section 3.1.1) to systematically combine partial logs in order to find a replay log capable of reproducing the bug. The algorithm receives a similarity metric and a heuristic as input, and starts with the candidate generator ranking the partial logs according to their relevance value (line 2). Next, the candidate generator builds a candidate replay log (line 6), following the strategies described in Sections 3.3.1 and 3.3.2, respectively for Similarity-Guided Merge and Dispersion-Guided Merge heuristics.

At the end of this process, the replay oracle enforces the schedule defined by the candidate replay log and verifies whether the bug is reproduced or not (line 7). If it is, the goal has been achieved and the algorithm ends. If it is not, the candidate generator produces a new replay log and sends it to the replay oracle for a new attempt. This procedure is repeated while the bug has not been replayed (*i.e.*, $hasBug = false$) and the candidate generator has more replay logs to be attempted. It should be referred that, in the worst case scenario, where all the replay logs generated by the heuristic fail to trigger the bug, the algorithm switches to a *brute force* mode. Here, all possible access vectors (collected by the partial logs) are tested for each missing SPE of the base partial log (lines 10-12), until the error is triggered or the maximum number of attempts to reproduce the bug is reached (*i.e.*, $replayOracle.\mathsf{reachedMaxAttempts}() = True$).

Corner Case: Partial Logs with 1 SPE. A special case in the cooperative record and replay approach occurs when the developer defines recording profiles that result in partial logs tracing information for a single SPE. This particular scenario, although minimizing the runtime overhead, hampers the goal of combining access vectors from potentially compatible partial

Algorithm 5: Statistical Analyzer's algorithm to produce a complete replay log via partial log combination.

```
Input: PLogs: set with partial logs collected from production runs.
       heuristic: heuristic to merge partial logs (it can be either sgm or dgm)
       simMetric: metric used to measure similarity between partial logs.
Output: replayLog: a complete execution trace ready to be replayed.

1  // Compute relevance
2  $MostRelevant \leftarrow candidateGen$.computeRelevance($PLogs$, $heuristic$, $simMetric$)
3  // Generate the next candidate replay log and attempt to replay the bug
4  $hasBug = false$
5  while ! $hasBug \wedge candidateGen$.hasNextReplayLog() do
6      $replayLog \leftarrow candidateGen$.genNextReplayLog($MostRelevant$, $heuristic$, $simMetric$)
7      $hasBug \leftarrow replayOracle$.replay($replayLog$)
8  end
9  // Brute Force attempts
10 while ! $hasBug \wedge$ ! $replayOracle$.reachedMaxAttempts() do
11     for all partial logs, test all possible combinations of access vectors to fill the missing
       SPEs and check whether the bug is reproduced
12 end
13 return $replayLog$
```

logs. Since partial logs with 1 SPE either do not overlap at all or overlap in their single SPE, it becomes pointless to apply the similarity metrics and the heuristics.

To address this issue, we compute the correlation between the bug and each access vector individually using a statistical indicator named *BugCorrelation*. *BugCorrelation* is adapted from the scoring method proposed by Liblit et al [2003] that, in addition to failing executions, leverages information from successful executions. Concretely, we classify access vectors based on their *Sensitivity* and *Specificity*, *i.e.*, based on whether they account for many failed runs and few successful runs. *BugCorrelation* is then computed as the harmonic mean between *Sensitivity* and *Specificity*, thus allowing the identification of access vectors that are simultaneously highly sensitive and specific.

Let F_{total} be the total number of partial logs, resulting from failing executions, that have traced a given SPE s. For an access vector v, let $F(v)$ be the number of failing partial logs that have recorded v for s, and let $S(v)$ be the number of successful partial logs that have recorded v for s. The three aforementioned statistical indicators are then calculated as follows.

$$Sensitivity(v) = \frac{F(v)}{F_{total}} \tag{3.7}$$

$$Specificity(v) = \frac{F(v)}{S(v) + F(v)} \tag{3.8}$$

$$BugCorrelation(v) = \frac{2}{\frac{1}{Sensitivity(v)} + \frac{1}{Specificity(v)}} \tag{3.9}$$

In summary, the higher the *BugCorrelation* value, the more correlated with the bug the access vector is. Therefore, when partial logs contain only one SPE, the candidate generator computes the statistical indicators for each SPE and builds a complete log by merging the access vectors that exhibit higher *BugCorrelation*. If there is more than one access vector with the highest value of *BugCorrelation* for a given SPE, different combinations are tested. However, if the bug does not manifest after these attempts, the candidate generator switches to a *brute force* mode, where it tests all possible combinations of access vectors as in Algorithm 5.

3.5 Evaluation

The main goal of CoopREP is to reproduce concurrency bugs with lower overhead than previous approaches, using cooperative logging and partial log combination. In this context, we are interested in evaluating CoopREP by answering the following questions:

- Which similarity metric provides the best trade-off between accuracy, execution time, and scalability? (Section 3.5.2)
- Which CoopREP heuristic replays the bug in fewer attempts? How do the heuristics compare to previous state-of-the-art deterministic replay systems? (Section 3.5.3)
- How does CoopREP's effectiveness vary with the number of logs collected? (Section 3.5.4)
- How much runtime overhead reduction can CoopREP achieve with respect to other classic, non-cooperative logging schemes? (Section 3.5.5)
- How much can CoopREP decrease the size of the logs generated? (Section 3.5.6)

3.5.1 Experimental Setting

We implemented a prototype of CoopREP in Java, using Soot[8] to instrument the target applications. We have also implemented two other state-of-the-art record and replay systems on top of CoopREP for comparison with their respective original, non-cooperative versions: *i)* LEAP [Huang, Liu, & Zhang 2010], which uses a classic order-based strategy to record information, and *ii)* PRES [Park, Zhou, Xiong, Yin, Kaushik, Lee, & Lu 2009], a search-based system. Similarly to CoopREP's approach, PRES only traces partial logs at runtime and, then, uses a post-recording exploration technique to produce a failure-inducing execution trace. Unlike CoopREP though, PRES does not combine logs from multiple user instances. As such, the version of CoopREP combined with PRES further reduces the amount of information that the original PRES version traces, by distributing it across multiple user instances of the program. Since PRES's code is not publicly available, we tried to reimplement it as faithfully as possible, according to the details given in [Park, Zhou, Xiong, Yin, Kaushik, Lee, & Lu 2009].

Regarding CoopREP's partial logging profiles, we varied the percentage of the total SPEs logged in each run according to three different coverage configurations – 25%, 50%, and 75%. In addition, we also considered the extreme case where partial logs only have one SPE. For each configuration, we used 500 partial logs from different failing executions, plus 100 additional partial logs from successful runs (used to compute the statistical indicators for the 1SPE case). To fairly compare the recording configurations, partial logs were generated from complete logs by randomly picking the amount of SPEs corresponding to the coverage configuration's percentage.

The maximum number of attempts of the heuristics to reproduce the bug was set to 500. Note that, following the 500 tries, one may resort to the brute force approach. Since we have verified, in our experiments, that brute force partial log combination brought no added value to the heuristics, we opted for reporting the results of the experimental evaluation solely up to the threshold of 500 replay attempts (see Section 3.5.3).

As a final remark, in the following sections, we assume the order-based version of CoopREP (*i.e.*, using LEAP on top of CoopREP) for the experiments, unless mentioned otherwise.

All the experiments were conducted in a machine Intel Core 2 Duo at 3.06 GHz, with 4 GB of RAM and running Mac OS X. However, the logs were evenly recorded from four different

[8]http://www.sable.mcgill.ca/soot/

machines: three of them with 4 GB of RAM, running Mac OS X, but with processors Intel Core 2 Duo at 2.26 GHz, 2.66 GHz, and 3.06 GHz, respectively; and the last one equipped with an 8 core processor AMD FX 8350 at 4GHz, 16 GB of RAM, running Ubuntu 13.04.

3.5.2 Similarity Metrics Comparison

The goal of the first question is to evaluate how effective CoopREP is in replaying concurrency bugs, and assess which partial log similarity metric – *Plain Similarity (PSIM)*, *Dispersion Similarity (DSIM)*, or *Dispersion Hamming Distance (DHS)* – exhibits the best trade-off between accuracy and execution time. To this end, we used several programs from the IBM ConTest benchmark suite [Farchi, Nir, & Ur 2003a], which contain various types of concurrency bugs.[9] Table 3.2 describes these programs in terms of their number of SPEs, the total number of shared accesses, the failure rate, and the bug pattern according to Farchi et al. [Farchi, Nir, & Ur 2003a].

Table 3.2: ConTest benchmark programs.

Program	**#SPE**	**#Total Accesses**	**Failure Rate**	**Bug Description**
BoundedBuffer	15	525	1%	deadlock (notify instead of notifyAll)
BubbleSort	11	52495	2%	atomicity violation (wrong/no lock)
BufferWriter	6	130597	35%	atomicity violation (wrong/no lock)
Deadlock	4	12	12%	deadlock
Manager	5	2368	28%	atomicity violation (wrong/no lock)
Piper	8	580	6%	ordering violation (missing condition for wait)
ProducerConsumer	8	576	15%	data race
TicketOrder	9	6662	2%	atomicity violation (wrong/no lock)
TwoStage	5	27103	1%	atomicity violation (two-stage)

Accuracy. In these experiments, we are interested in evaluating the impact of the metrics' accuracy on the effectiveness of CoopREP. Table 3.3 reports the number of attempts required by the *SGM* heuristic, using each the three metrics, to replay the bug for the configurations previously mentioned (25%, 50%, and 75%). We omit the results for *DGM* heuristic because they show similar trends.

By analyzing Table 3.3, we can see that, apart from program *Manager*, all three metrics allow the heuristic to replay the bugs for at least one recording scheme. Despite that, *PSIM* is

[9]We restrict our analysis to ConTest programs that have at least 4 SPEs.

Table 3.3: Number of attempts required by the *SGM* heuristic, using the three similarity metrics, to replay the ConTest benchmark bugs. The *X* indicates that the heuristic failed to replay the bug within the maximum number of attempts. The "-" indicates that the corresponding configuration resulted in an 1SPE case and, therefore, it is not applicable for metric comparison.

Program	**25%**			**50%**			**75%**		
	PSIM	***DSIM***	***DHS***	***PSIM***	***DSIM***	***DHS***	***PSIM***	***DSIM***	***DHS***
BoundedBuffer	233	1	1	1	1	1	1	1	1
BubbleSort	482	482	482	372	372	372	256	256	256
BufferWriter	*X*	*X*	*X*	*X*	*X*	*X*	17	13	13
Deadlock	-	-	-	1	1	1	1	1	1
Manager	-	-	-	*X*	*X*	*X*	*X*	*X*	*X*
Piper	*X*	*X*	*X*	133	1	1	1	1	1
ProducerConsumer	*X*	*X*	*X*	287	19	19	6	6	6
TicketOrder	*X*	*X*	*X*	93	43	43	47	25	25
TwoStage	-	-	-	466	15	15	62	10	10

generally less efective than *DSIM* and *DHS*, as shown by the results for *Piper*, *ProducerConsumer*, *TicketOrder*, and *TwoStage* (especially for a coverage of 50%). These results confirm our insight about the positive effect of taking into account SPE dispersion when measuring partial log similarity.

Another interesting observation is concerned with the variation of the bug replay ability of the *SGM* heuristic across the benchmarks, regardless of the similarity metric. For instance, even when partial logs contain 75% of the total number of SPEs, the bug in program *BubbleSort* was only reproduced at the 256th attempt. On the other hand, the error in *BoundedBuffer* was replayed almost always at the 1st attempt. This is due to the dispersion of the SPEs that, when very high, hampers heavily the combination of compatible partial logs. We will go deeper into how SPE dispersion affects the bug replay ability in Section 3.5.3.

Finally, note that the number of attempts to replay the bug increases with the decrease of the number of SPEs per partial log, as expected.

Execution Time. In order to understand which metric provides the best trade-off between accuracy and execution time, we have also measured the time required by the heuristic to compute the relevance of all the 500 partial logs collected. The outcomes of these experiments are presented in Table 3.4. Recall that, for *SGM*, calculating the relevance requires computing the similarity between every pair of partial logs. Hence, the results in Table 3.4 also account for these computations.

By analyzing the table, we can see that *DHS* exhibits, as expected, the worst performance,

Table 3.4: Execution time (in seconds) to compute the relevance of 500 logs in *SGM*, for each similarity metric.

Program	**25%**			**50%**			**75%**		
	PSIM	***DSIM***	***DHS***	***PSIM***	***DSIM***	***DHS***	***PSIM***	***DSIM***	***DHS***
BoundedBuffer	2.24	2.27	3.45	2.54	2.6	3.79	2.62	2.77	4.04
BubbleSort	31.78	32.70	53.59	64.9	69.16	155.99	88.84	92.43	252.19
BufferWriter	584.68	604.41	702.56	1371.95	1605.48	1937.79	1516.41	2082.03	3532.38
Deadlock	-	-	-	1.85	1.89	2.02	2.40	2.47	2.52
Manager	-	-	-	4.87	4.75	20.44	8.85	9.31	30.17
Piper	2.04	2.53	3.26	2.74	3.11	5.98	2.95	3.52	9.79
ProducerConsumer	2.1	2.24	3.79	2.18	2.53	4.84	2.45	2.73	8.33
TicketOrder	5.99	6.64	12.57	10.16	10.68	28.6	11.77	11.86	47.60
TwoStage	-	-	-	7.50	7.45	31.08	8.64	9.11	44.93

by being up to 5.2x slower than *PSIM* for *TwoStage* (when logging 75% of the SPEs). In turn, *DSIM* typically achieves almost the same execution time as *PSIM*, which can be explained by the fact that the computation of overall-dispersion for each SPE is performed only once, at the beginning of the analysis. For this reason, we believe *DSIM to be the most cost-effective metric to measure similarity between partial logs.*

Table 3.4 also shows that computation time is greatly affected by the number of shared accesses in the program (which are indicated in Table 3.2). For instance, *BufferWriter* is simultaneously the program with the highest number of accesses (>130K) and the highest computation time (almost one hour for the 75% scheme using *DHS*). However, it should be noted that the values reported in Table 3.4 encompass the time required to load the logs into memory, create the objects and data structures used by CoopREP, as well as compute the similarity between the partial logs. Given that the task of computing similarity can be easily performed in parallel or even in the cloud (which does not happen in our current prototype), we conducted some additional experiments to further assess the computational cost and the scalability of the three metrics. We present these experiments in the following section.

Scalability. To assess the scalability of the metrics, we measured the amount of time required to compare two partial logs using each one of the three similarity metrics. Table 3.5 reports these results for logs of sizes ranging from 1KB to 1GB, as well as the bootstrapping time for each case (*i.e.*, the time required to load the logs into memory and create the data structures used by CoopREP).

From Table 3.5, it is easily observed that the great majority of the execution time in

CoopREP's statistical analysis is devoted to bootstrapping. Moreover, it is possible to see that this value increases linearly with the size of the logs being analyzed.

On the other hand, the cost to measure similarity using either *PSIM* or *DSIM* metrics is practically unaffected by the size of the logs being compared, because these metrics use hashes of the access vectors to compute the similarity value. As such, one can conclude that these metrics are very scalable and efficient. The same does not apply to *DHS* because it needs to compare every single position of the access vector and, thus, its computation time depends heavily on the size of the log. For instance, *DHS* took around 40s to compare a single pair of logs of 1GB.

In summary, the results in Table 3.5 provide further evidence that *DSIM* is the similarity metric that exhibits better trade-off between effectiveness and efficiency.

Table 3.5: Time (in seconds) required by the three metrics to measure similarity between two partial logs.

Log Size	**Execution Time (s)**			
	Bootstrapping	***PSIM***	***DSIM***	***DHS***
1 KB	0.031	0.001	0.001	0.002
10 KB	0.139	0.001	0.001	0.010
100 KB	0.477	0.003	0.004	0.014
1 MB	1.712	0.004	0.007	0.032
10 MB	17.037	0.005	0.008	0.489
100 MB	119.413	0.016	0.022	4.121
1 GB	1202.054	0.024	0.032	40.078

3.5.3 Heuristic Comparison

We are now interested in evaluating the ability to replay bugs of our merging heuristics with respect to state-of-the-art order-based and search-based techniques. We denote by CoopREP$_L$ the version of CoopREP that applies cooperative record and replay to LEAP [Huang, Liu, & Zhang 2010], and by CoopREP$_P$ the version of CoopREP that implements PRES's approach.

To a smaller extent, we are also interested in comparing the Similarity-Guided Merge heuristic against the Dispersion-Guided Merge heuristic to assess which one replays the bug in fewer attempts. Finally, we also aim at investigating bug reproducibility for the *1SPE* case, in which CoopREP uses statistical indicators to merge partial logs that have traced only one SPE.

Regarding the test subjects, besides the programs of the ConTest benchmark suite (see Table 3.2), we used four concurrency bugs from real-world applications, also used in previous

Table 3.6: Real-world concurrency bugs used in the experiments.

Program	**#SPE**	**#Total Accesses**	**Failure Rate**	**Bug Description**
Cache4j	4	129	15%	atomicity violation
Derby#2861	14	2509	5%	atomicity violation
Tomcat#37458	22	89	16%	atomicity violation
Weblech	5	54	1%	atomicity violation

work [Huang, Liu, & Zhang 2010; Huang & Zhang 2012; Sen 2008; Lucia, Wood, & Ceze 2011]. These applications are described in Table 3.6 in terms of number of SPEs, number of shared accesses, failure rate, and bug-pattern. *Cache4j* is a fast thread-safe implementation of a cache for Java objects; *Derby* is a widely used open-source Java RDBMS from Apache (in the experiments we used bug #2861 from Derby's bug repository); *Tomcat* is a widely-used JSP and servlet container (in the experiments we used bug #37458 from Tomcat's bug repository); and *Weblech* is a multithreaded web site download and mirror tool. Notice that, albeit these applications have thousands of lines of codes and many shared variables, Table 3.6 reports solely the SPEs experienced by each application's bug test driver. Similarly to previous works [Huang, Liu, & Zhang 2010; Huang & Zhang 2012; Sen 2008; Lucia, Wood, & Ceze 2011], we use test drivers to facilitate the manifestation of the failure. Despite that, some bugs are still very hard to experience (*e.g.*, *Weblech*).

3.5.3.1 CoopREP$_L$ vs LEAP (order-based)

Table 3.7 reports the number of attempts, required by each heuristic, to replay the benchmark bugs. Note that LEAP is a pure order-based system and, thus, is able to replay all bugs at the first attempt.

An overall analysis of Table 3.7 shows that CoopREP$_L$ is able to replay all bugs except *Manager*. Moreover, in 11 of the 12 cases, the bug was reproduced at the first attempt (for at least one recording scheme), which proves the efficacy of the heuristics in combining compatible partial logs.

Comparing now *SGM* and *DGM* against each other, Table 3.7 shows that *DGM* tends to be more effective than *SGM*, especially when partial logs record a higher percentage of SPEs. To better understand why this happens, let us observe the benchmarks' SPE *individual-dispersion*

Table 3.7: Number of replay attempts required by CoopREP$_L$ with 1SPE, *SGM*, and *DGM* (both heuristics using *DSIM* metric). The *X* indicates that the heuristic failed to replay the bug within the maximum number of attempts. The "-" indicates that the corresponding configuration resulted in an 1SPE case and, therefore, it is not applicable to the heuristics. Shaded cells highlight the CoopREP$_L$ scheme with best trade-off between effectiveness and amount of information recorded.

Program	**1SPE**	***SGM + DSIM***			***DGM + DSIM***		
		25%	**50%**	**75%**	**25%**	**50%**	**75%**
BoundedBuffer	*X*	1	1	1	3	1	1
BubbleSort	*X*	482	372	256	1	1	1
BufferWriter	*X*	*X*	*X*	13	*X*	*X*	97
Deadlock	1	-	1	1	-	1	1
Manager	*X*	-	*X*	*X*	-	*X*	*X*
Piper	*X*	*X*	1	1	*X*	32	1
ProducerConsumer	*X*	*X*	19	7	*X*	1	1
TicketOrder	*X*	*X*	43	25	*X*	9	1
TwoStage	*X*	-	15	10	-	1	1
Tomcat#37458	1	1	1	1	1	1	1
Derby#2861	*X*	*X*	*X*	59	*X*	*X*	1
Weblech	1	-	1	1	-	1	1
Cache4j	1	-	1	1	-	1	1

depicted in Figure 3.6.[10] Here, individual-dispersion corresponds to the ratio between the number of *different* access vectors logged for a given SPE and the total number of access vectors collected for that SPE (across all instances).[11] It is computed as follows:

$$individualDisp(s) = \frac{\#\mathcal{AV}_s^*}{\#\mathcal{AV}_s} \tag{3.10}$$

The first conclusion that can be drawn from the figure is that programs with high average SPE individual-dispersion – *BufferWriter* and *Manager* – are also the ones for which replaying the bug was harder. In particular, the logs collected for *Manager* contained different access vectors for all SPEs of the program, which clearly represents unfavorable conditions for a partial log combination approach (in fact, none of the heuristics was able to reproduce the bug, as indicated in Table 3.7). On the other hand, programs with low SPE dispersion (such as *Deadlock* and *Tomcat*) were easily replayed by both heuristics, as well as when partial logs contained only a single SPE (see the "1SPE" column in Table 3.7). This confirms our insight that CoopREP's

[10]For the sake of readability and to ease the comparison, Figure 3.6 presents only the individual-dispersion values for the full logging configuration.

[11]Note that individual-dispersion differs from overall-dispersion (see Equation 3.2) because it accounts for different access vectors recorded for a particular SPE, rather than for the whole set of SPEs.

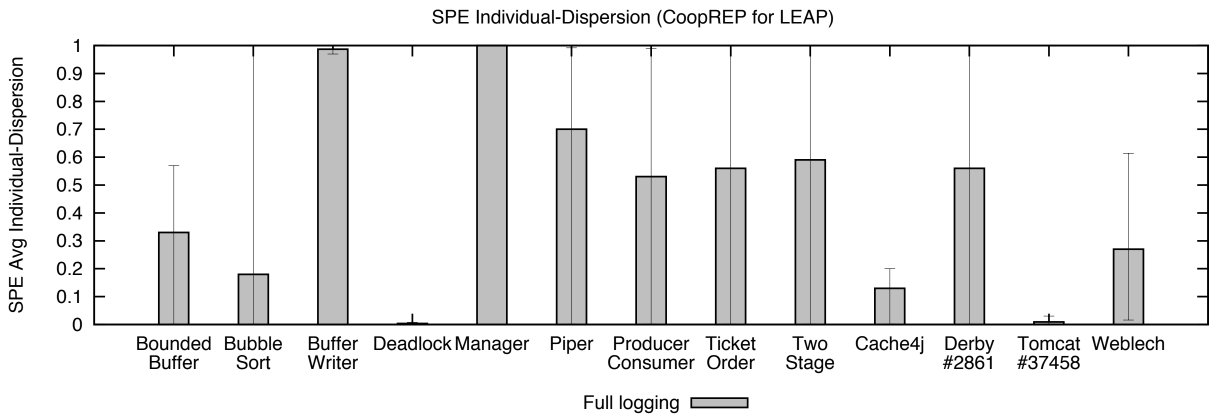


Figure 3.6: Average SPE individual-dispersion value for the benchmarks, with a full LEAP logging scheme. The error bars indicate the minimum and the maximum values observed.

bug replay ability depends on the program's SPE dispersion.

Figure 3.6 also shows that the SPE individual-dispersion is highly variant in most test cases (*Deadlock*, *Cache4j*, and *Tomcat* are the exceptions). This means that these programs have, simultaneously, SPEs for which the partial logs have recorded always the same access vector, and SPEs for which the partial logs have traced (almost) always different access vectors. Under these circumstances, *DGM* was clearly more effective than *SGM*, because it was able to pick, as the most relevant partial log, the one that had traced all the most disperse SPEs. This fact is particularly evident for programs *BubbleSort*, *TicketOrder*, and *Derby*, for the 75% recording scheme. These results support our claim that tracing all SPEs with very high dispersion in the same partial log increases the likelihood of filling the remaining non-recorded SPEs with very common (and, thus, compatible) access vectors.

However, *DGM* did not always outperform *SGM*: for the *BufferWriter* and *Piper* programs, *SGM* exhibited better effectiveness. This can be explained by the fact that, for these cases, the capacity of the partial logs was not sufficient to encompass all SPEs with high dispersion. As a consequence, *DGM* ended up having several partial logs with the same relevance value and simply picked one at random for each attempt (moreover, the majority of these most relevant logs did not have any similarity with the remaining partial logs). In opposition, *SGM* was able to quickly identify subsets of similar partial logs and, thus, generate a feasible replay log in fewer attempts.

3.5.3.2 CoopREP$_P$ vs PRES (search-based)

To further assess the benefits and limitations of cooperative logging, we also implemented a version of CoopREP, denoted CoopREP$_P$, using the approach proposed by PRES [Park, Zhou, Xiong, Yin, Kaushik, Lee, & Lu 2009], a state-of-the-art search-based record and replay system for C/C++ programs.[12]

PRES shares CoopREP's idea of minimizing the recording overhead during production runs at the cost of an increase in the number of attempts to replay the bug during diagnosis. As referred in Section 2.2.2, PRES first traces a *sketch* of the original execution and, then, performs an offline guided search to explore different combinations of (non-recorded) thread interleavings. To cope with the very large dimension of the space of possible execution schedules, PRES leverages feedback produced from each failed attempt to increase the chances of finding the bug-inducing ordering in the subsequent one.

The authors of PRES have explored five different sketching techniques that imply different trade-offs between recording overhead and reproducibility. Starting from a baseline logging profile that traced only input, signals and thread scheduling, the authors performed experiments with different sketches that incrementally record more information, namely the global order of synchronization operations, system calls, functions, basic blocks, and shared-memory operations, respectively. In this work, we opted for implementing a version of CoopREP that employs the PRES-BB recording scheme (which logs the global order of basic blocks containing shared accesses), as it provides a good trade-off between effectiveness and overhead according to the results reported in [Park, Zhou, Xiong, Yin, Kaushik, Lee, & Lu 2009]. We also thought of testing with a slightly more relaxed scheme, namely PRES-SYNC (which only traces the global order of synchronization operations) but, since our test cases have very few synchronization variables (0 to 3), we considered that applying a cooperative logging technique to this case would not be very interesting.

Concretely, PRES-BB logs the order in which threads access basic blocks containing operations on shared variables into a single vector. To allow for cooperative logging, we changed this recording scheme to treat each basic block (accessing shared variables) as an SPE, which means that CoopREP$_P$ traces a distinct access vector for each *shared basic block*. During replay phase,

[12]We implemented a version of PRES in Java for the experiments.

Table 3.8: Number of replay attempts required by PRES-BB, CoopREP$_P$ with *1SPE*, *SGM*, and *DGM* (both heuristics using *DSIM* metric). Column *#SPEs* indicates the number of basic blocks that contain accesses to shared variables. The *X* indicates that the heuristic failed to replay the bug within the maximum number of attempts. The "-" indicates that the corresponding configuration resulted in an 1SPE case and, therefore, it is not applicable to the heuristics. Shaded cells highlight the CoopREP$_P$ scheme with best trade-off between effectiveness and amount of information recorded.

Program	**#SPEs**	**PRES-BB**	**1SPE**	***SGM + DSIM***			***DGM + DSIM***		
				25%	**50%**	**75%**	**25%**	**50%**	**75%**
BoundedBuffer	22	1	1	1	1	1	1	1	1
BubbleSort	14	1	1	1	1	1	1	1	1
BufferWriter	9	1	*X*	*X*	4	4	10	2	1
Deadlock	3	1	1	-	1	1	-	1	1
Manager	13	1	*X*	*X*	*X*	*X*	*X*	*X*	*X*
Piper	12	1	*X*	4	1	1	6	3	1
ProducerConsumer	20	1	*X*	1	1	1	6	2	1
TicketOrder	14	2	*X*	*X*	316	218	*X*	6	2
TwoStage	9	1	*X*	*X*	*X*	44	*X*	*X*	1
Tomcat#37458	32	3	*X*	*X*	*X*	6	*X*	*X*	3
Derby#2861	15	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*
Weblech	7	1	*X*	1	1	1	1	1	1
Cache4j	6	1	*X*	1	1	1	1	1	1

we first generate a complete log using CoopREP$_P$'s heuristics and then apply PRES's replay scheme.

Table 3.8 compares both CoopREP$_P$'s and PRES's post-recording exploration techniques against each other, by focusing the evaluation on the number of replay attempts performed by each technique.

Similarly to the previous section, CoopREP$_P$ exhibits very good effectiveness. In particular, CoopREP$_P$ was able to replay all bugs with the same number of attempts as PRES-BB, apart from program *Manager* (though achieving an average reduction of log size of 3x, as we shall discuss in Section 3.5.6). Besides *Manager*, CoopREP$_P$ was also not able to reproduce the bug in *Derby*, but this error was not replayed by PRES-BB either.

Comparing now the two heuristics against each other, Table 3.8 shows once again that *DGM* tends to achieve better results than *SGM* (the former performed equally to or better than the latter in 11 of the 13 cases).

Another interesting observation is that CoopREP$_P$ required fewer replay attempts than CoopREP$_L$ for some test cases, namely *BufferWriter*, *BubbleSort*, and *Piper*. If we observe

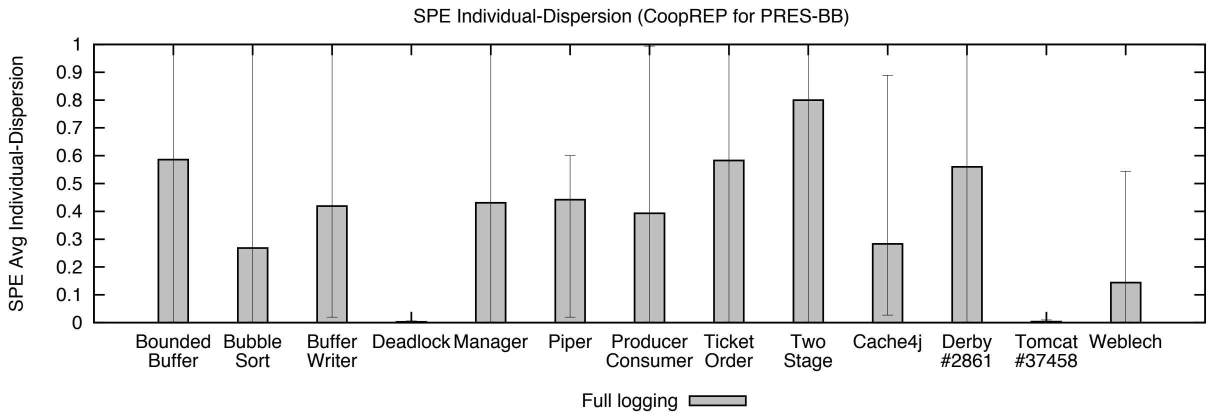


Figure 3.7: Average SPE individual-dispersion value for the benchmarks, with a full PRES-BB logging scheme. The error bars indicate the minimum and the maximum values observed.

the SPE individual-dispersion for PRES-BB for these programs (depicted in Figure 3.7), it is possible to see that the corresponding values are smaller than those of Figure 3.6.

We believe that this is the effect of considering SPEs as basic blocks with accesses to shared variables instead of shared variables themselves. Here, access vectors that previously encompassed all accesses to a given shared variable, now become scattered across different basic blocks. As a consequence, if a given basic block contains only accesses to a single shared variable, it will have fewer accesses in PRES-BB than its corresponding shared variable in LEAP, which results in a lower individual dispersion.

However, the opposite scenario might occur as well. If a basic block encompasses accesses to multiple shared variables, its resulting SPE individual dispersion is expected to increase. This is visible, for instance, in programs *TwoStage* and *TicketOrder*, which exhibited higher individual dispersion in Figure 3.7 and required more attempts to replay the bug in Table 3.8 than in Table 3.7.

3.5.3.3 Partial Log Combination vs Bug Correlation

We now compare the heuristics for partial log combination against the approach of tracing only one SPE per partial log and using the statistical indicators to measure correlation between the error and the access vectors (see Section 3.4). From the results shown in Tables 3.7 and 3.8, one can conclude that the former approach is clearly more effective for most cases. In fact, the *1SPE* approach was rarely able to replay the bug in our experiments, apart from the programs

Table 3.9: Number of the attempts required by *SGM* and *DGM* to replay the bugs in programs *Deadlock*, *BoundedBuffer*, and *Piper* when varying the number of partial logs collected. The *X* indicates that the heuristic failed to replay the bug within the maximum number of attempts. The "-" indicates that the corresponding configuration resulted in an 1SPE case and, therefore, it is not applicable to the heuristics.

	#Logs	**Deadlock**			**BoundedBuffer**			**Piper**		
		25%	**50%**	**75%**	**25%**	**50%**	**75%**	**25%**	**50%**	**75%**
SGM	**16**	-	1	1	*X*	*X*	1	*X*	*X*	*X*
	32	-	1	1	*X*	1	1	*X*	*X*	32
	64	-	1	1	*X*	1	1	*X*	*X*	6
	128	-	1	1	3	1	1	*X*	*X*	15
	256	-	1	1	1	1	1	*X*	*X*	1
	512	-	1	1	1	1	1	*X*	1	1
DGM	**16**	-	1	1	*X*	*X*	1	*X*	*X*	*X*
	32	-	1	1	*X*	1	1	*X*	*X*	1
	64	-	1	1	*X*	1	1	*X*	*X*	1
	128	-	1	1	34	1	1	*X*	*X*	1
	256	-	1	1	6	1	1	*X*	*X*	1
	512	-	1	1	3	1	1	*X*	32	1

with very low average SPE individual-dispersion. Nevertheless, for this kind of programs, *1SPE* becomes an appealing approach to adopt, as it provides the lowest runtime overhead, as we will see in Section 3.5.5.

3.5.4 Variation of Effectiveness with the Number of Logs Collected

Our third research question regards the impact of varying the number of logs collected on the number of attempts required to replay the bug. To this end, we chose three of the benchmarks bugs, successfully reproduced by both heuristics (for CoopREP$_L$), that provide a representative set of different average individual-dispersion values: *Deadlock*, *BoundedBuffer*, and *Piper*. For each one of these programs, we ran both *SGM* and *DGM* with a number N of (randomly chosen) partial logs, where $N \in \{16, 32, 64, 128, 256, 512\}$. Once again, we varied the percentage of SPEs logged as in the previous section.

Table 3.9 reports the outcomes of the experiments. As expected, the results show that bugs of programs with low average individual-dispersion (such as *Deadlock*) can be easily reproduced even when gathering a small number of logs. On the other hand, as the average individual-dispersion increases, the reproducibility of the program through partial logging schemes becomes much more challenging, thus requiring a greater number of partial logs to deterministically trigger the bug. Program *Piper* reflects this scenario well, as it was only possible to replay the bug with less than 75% of the total SPEs when logging 512 partial logs.

The results for *Piper* highlight another interesting observation: an increase in the number of partial logs collected does not always imply a decrease in the number of attempts needed to reproduce the error (see column 75% of *Piper* for *SGM* in Table 3.9). This because, for programs with higher average individual-dispersion, having more partial logs also might increase the chances of making "wrong choices". For instance, *SGM* took more attempts to find a bug-inducing replay log with 128 partial logs than with 64, basically because there were more partial logs considered relevant that ended up not being compatible with the partial logs in the k-nearest neighbor set.

Finally, note that these results provide additional evidence to support the claim that *DGM* is more effective than *SGM* when the partial logs trace a larger number of SPEs (*e.g.*, *Piper* with 75% recording coverage). Once again, this is because *DGM* is able to quickly identify the partial logs that have traced all the most disperse SPEs.

3.5.5 Runtime Overhead

In this section we analyze the performance benefits achievable via CoopREP's approach with respect to non-cooperative logging schemes, such as LEAP and PRES-BB. To this end, we compared CoopREP$_L$'s and CoopREP$_P$'s recording overhead respectively against that of LEAP's and PRES-BB's for the benchmark programs.

For each test case, we measured the execution time by computing the arithmetic mean of five runs. For CoopREP schemes, we computed the average value for the three different configurations of logging profiles (each profile configuration with five runs, as well).

3.5.5.1 CoopREP$_L$ vs LEAP

Figure 3.8 reports the runtime overhead on the programs of Table 3.7. Native execution times range from 0.01s (for *BubbleSort*) to 1.2s (for *Weblech*).

From the figure, it is possible to see that CoopREP$_L$ always achieves lower runtime degradation than LEAP. However, the results also highlight that overhead reductions are not necessarily linear. This is due to the fact that some SPEs are accessed much more frequently than others.

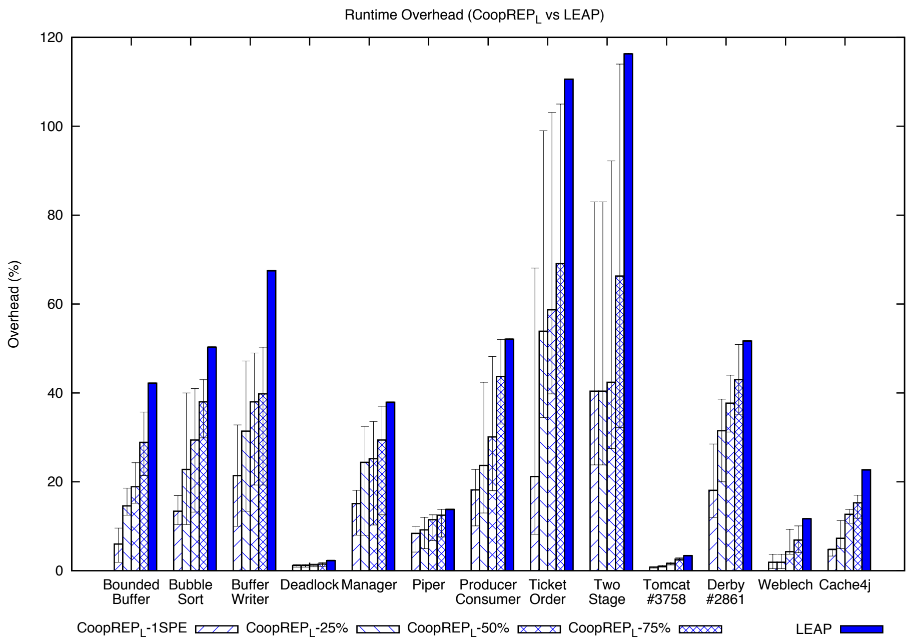


Figure 3.8: Runtime overhead (in %) of CoopREP$_L$ and LEAP, for the benchmark programs. The error bars indicate the minimum and the maximum values observed.

Since the instrumentation of the code is performed statically, the load in terms of thread accesses may not be equally distributed among the users, as already referred in Section 3.1.2.

Figure 3.8 also shows that it is typically more beneficial to use partial logging in scenarios where full logging has a higher negative impact on performance. The most notorious cases are *BufferWriter*, *TicketOrder*, and *TwoStage*, which are also among the applications with more SPE accesses (see Table 3.2). For example, in *TwoStage*, LEAP imposes a performance overhead of 116% on average, while CoopREP$_L$ incurs only 42% overhead on average when recording half the SPEs (which is sufficient to successfully replay the bug).

3.5.5.2 CoopREP$_P$ vs PRES-BB

We now compare the recording overhead between PRES-BB and CoopREP$_P$. Figure 3.9 plots our results.

Once again, CoopREP$_P$ exhibits always lower runtime slowdown than PRES-BB, although the reductions are not linear with the decrease in the partial logging percentage. Similarly to LEAP, the benchmarks for which the benefits of CoopREP$_P$ are more prominent are those that incur higher recording overhead, namely *BubbleSort*, *BufferWriter*, *TicketOrder*, and *TwoStage*. In particular, for *BubbleSort*, PRES-BB imposes an overhead of 276%, whereas CoopREP$_P$-

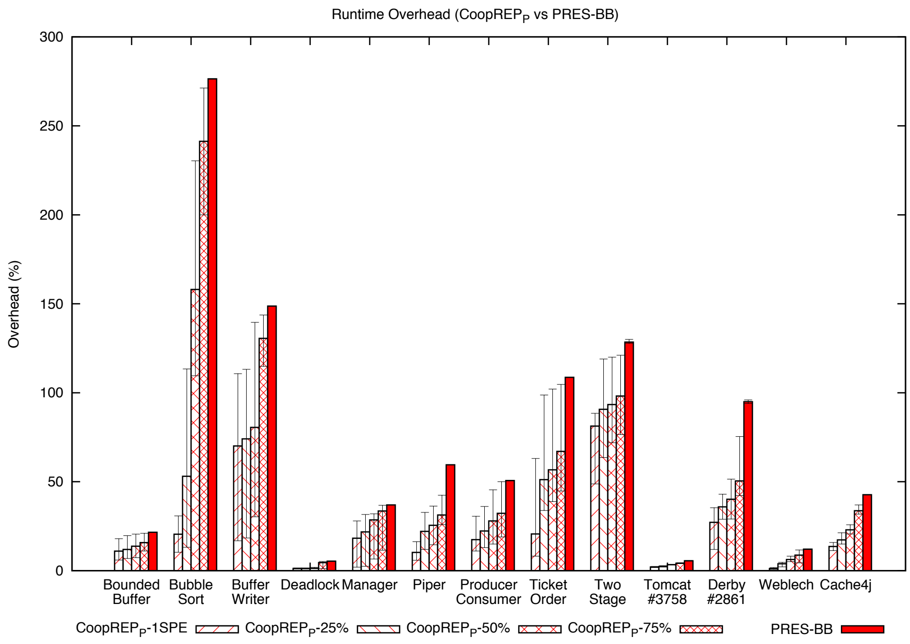


Figure 3.9: Runtime overhead (in %) of CoopREP$_P$ and PRES-BB, for the benchmark programs of Table 3.8. The error bars indicate the minimum and the maximum values observed.

1SPE incurs only 21% overhead on average and still reproduces the error.

3.5.5.3 Summary

To easily analyze the trade-off between effectiveness and recording overhead for the different approaches, we now match the most effective scheme for each system (as reported in Tables 3.7 and 3.8) to the corresponding average runtime penalty. The outcomes are reported in Table 3.10. Recall that LEAP is able to replay the bug at the first attempt for all test cases. For the cases where either PRES-BB or CoopREP were not able to reproduce the bug, we simply report the highest average recording overhead observed.

As shown by Table 3.10, CoopREP reduces the average recording overhead of PRES-BB and LEAP by 39% and 21% on average, respectively, while still being effective in replaying the failure for all benchmarks except for *Manager* and *Derby* (solely when using PRES-BB approach). In light of these results, we advocate the benefits of the cooperative record and replay approach with respect to other state-of-the-art approaches.

Table 3.10: Average runtime overhead incurred by the most effective CoopREP schemes for both PRES-BB and LEAP (according to Tables 3.7 and 3.8).

Program	**PRES-BB**	**Best of CoopREP$_P$**	**LEAP**	**Best of CoopREP$_L$**
BoundedBuffer	22%	*(1SPE)* 11%	42%	*(SGM-25%)* 15%
BubbleSort	276%	*(1SPE)* 21%	50%	*(DGM-25%)* 23%
BufferWriter	149%	*(DGM-75%)* 131%	68%	*(SGM-75%)* 40%
Deadlock	5%	*(1SPE)* 1%	2%	*(1SPE)* 1%
Manager	37%	*(X)* 34%	38%	*(X)* 29%
Piper	60%	*(SGM-50%)* 26%	14%	*(SGM-50%)* 12%
ProducerConsumer	51%	*(SGM-25%)* 22%	52%	*(DGM-50%)* 30%
TicketOrder	109%	*(DGM-75%)* 67%	111%	*(DGM-75%)* 69%
TwoStage	128%	*(DGM-75%)* 98%	116%	*(DGM-50%)* 42%
Tomcat#37458	6%	*(DGM-75%)* 4%	12%	*(1SPE)* 2%
Derby#2861	95%	*(X)* 51%	52%	*(DGM-75%)* 43%
Weblech	12%	*(DGM-25%)* 4%	12%	*(1SPE)* 2%
Cache4j	43%	*(DGM-25%)* 17%	23%	*(1SPE)* 5%
AVERAGE	*76%*	*37%*	*45%*	*24%*

3.5.6 Log Size

We now quantify the benefits achievable by CoopREP with respect to LEAP and PRES in terms of space overhead. To measure CoopREP's space overhead, we computed the average size of 10 partial logs picked at random, for each of the partial logging schemes.

3.5.6.1 CoopREP$_L$ vs LEAP

Figure 3.10 reports the sizes of the logs generated by LEAP and CoopREP$_L$ for the different benchmarks.

Unsurprisingly, the space overhead follows a trend that is analogous to that observed for the performance overhead: logs generated by CoopREP$_L$ are always smaller than those produced by LEAP, although it is possible to observe a significant variance in the log size for some programs. For instance, programs such as *BubbleSort* and *BufferWriter* have simultaneously partial logs with much smaller and very similar size comparing to those of LEAP. As in the performance overhead assessment, this variance is due to the heterogeneity in the frequency of accesses to SPEs. Figure 3.11 depicts this phenomenon, by plotting the average and variance of the number of SPE accesses for each benchmark.

From the analysis of Figure 3.11 it is possible to verify that programs with high variance in the log size correspond to those that also have high variance in the number of accesses per SPE.

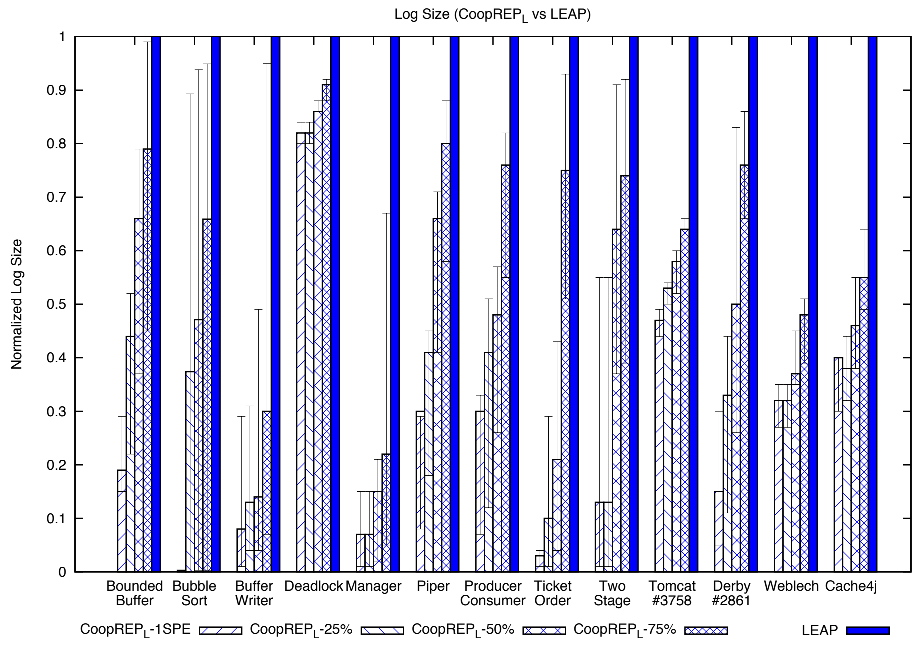


Figure 3.10: Log size reduction achieved by the various CoopREP$_L$ partial logging schemes with respect to LEAP. The error bars indicate the minimum and the maximum values observed.

For instance, *BubbleSort* has an SPE with only 2 accesses and another with more than 49700 accesses.

3.5.6.2 CoopREP$_P$ vs PRES-BB

Figure 3.12 depicts the log sizes for PRES-BB and CoopREP$_P$. The results are similar to those observed in Figure 3.10, *i.e.*, programs with greater disparity in SPE access frequency exhibit higher variance in partial log sizes. Also, for the programs containing more shared accesses, the benefits of partial logging are more clear. For example, in *BubbleSort*, a complete

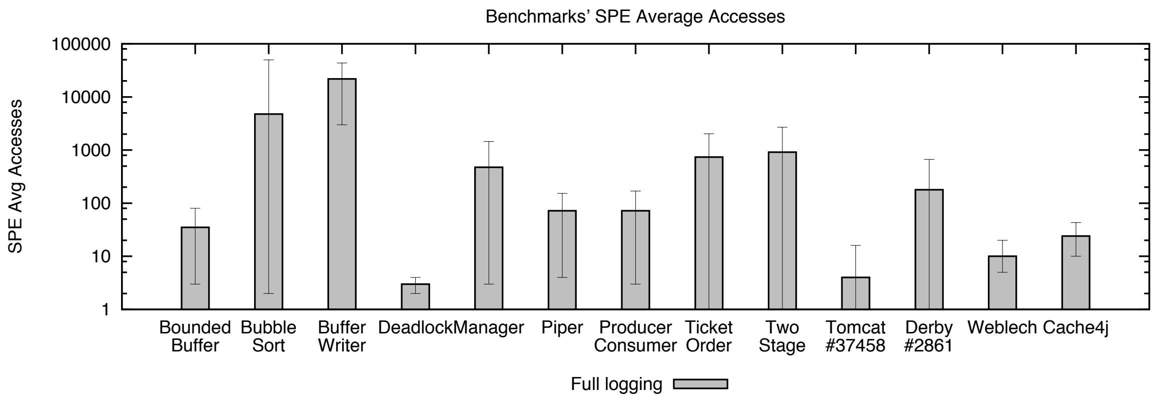


Figure 3.11: Variance in SPE accesses. The error bars indicate the minimum and the maximum values observed.

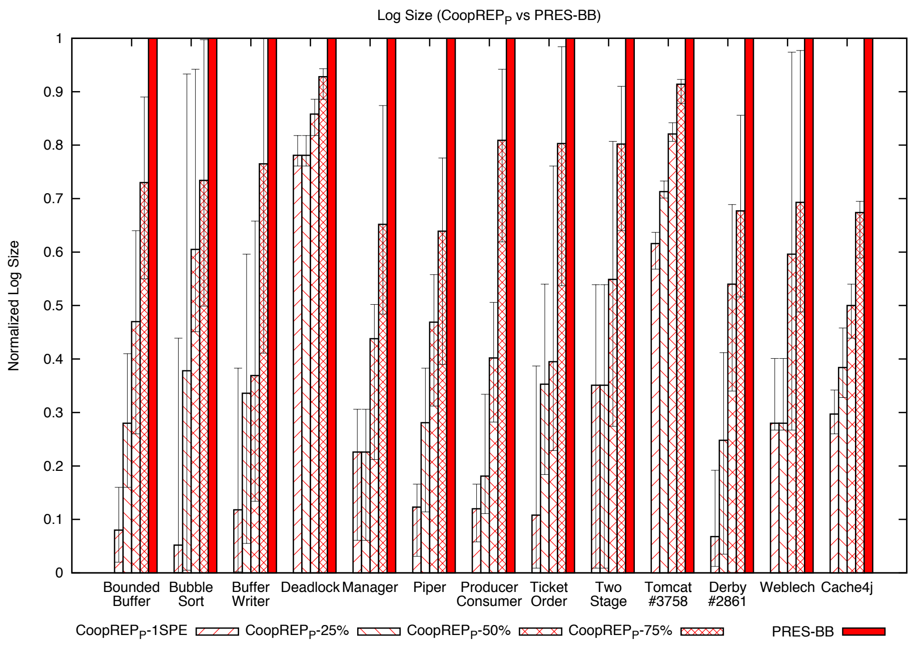


Figure 3.12: Log size reduction achieved by the various CoopREP$_P$ partial logging schemes with respect to PRES-BB. The error bars indicate the minimum and the maximum values observed.

PRES-BB log has 1.2MB, whereas the average log size for CoopREP$_P$-1SPE has 63KB on average (and 535KB at most), which suffices to replay the bug.

3.5.6.3 Summary

Table 3.11 presents, for each benchmark, the log size reduction achievable by CoopREP's most effective scheme with respect to PRES-BB and LEAP. The results in the table also support our claim about CoopREP being more cost-effective than full logging solutions. In particular, CoopREP produced logs that are, on average, 3x smaller than those generated by both PRES-BB and LEAP.

3.5.7 Discussion

In this section, we summarize the findings of our experiments by answering the research questions that motivated this evaluation study.

– *Which similarity metric –* PSIM, DSIM, *or* DHS *– provides the best trade-off in terms of accuracy, execution time, and scalability?*

The experiments presented in Section 3.5.2 reveal that *DSIM (Dispersion Similarity)* is, at

Table 3.11: Average size of the partial logs generated by the most effective CoopREP schemes for both PRES-BB and LEAP (according to Tables 3.7 and 3.8).

Program	**PRES-BB**	**Best of CoopREP$_P$**	**LEAP**	**Best of CoopREP$_L$**
BoundedBuffer	45KB	*(1SPE)* 4KB	6KB	*(SGM-25%)* 3KB
BubbleSort	1.2MB	*(1SPE)* 63KB	600KB	*(DGM-25%)* 209KB
BufferWriter	214KB	*(DGM-75%)* 164KB	3.3MB	*(SGM-75%)* 988KB
Deadlock	500B	*(1SPE)* 400B	500B	*(1SPE)* 400B
Manager	29KB	*(X)* 19KB	107KB	*(X)* 23KB
Piper	17KB	*(SGM-50%)* 8KB	7KB	*(SGM-50%)* 5KB
ProducerConsumer	15KB	*(SGM-25%)* 27KB	8KB	*(DGM-50%)* 4KB
TicketOrder	72KB	*(DGM-75%)* 57KB	71KB	*(DGM-75%)* 53KB
TwoStage	138KB	*(DGM-75%)* 111KB	52KB	*(DGM-50%)* 33KB
Tomcat#37458	3.4KB	*(DGM-75%)* 3.1KB	3KB	*(1SPE)* 2KB
Derby#2861	171KB	*(X)* 116KB	27KB	*(DGM-75%)* 20KB
Weblech	2KB	*(DGM-25%)* 400B	2KB	*(1SPE)* 600B
Cache4j	3KB	*(DGM-25%)* 1KB	3KB	*(1SPE)* 900B
AVERAGE	*148KB*	*44KB*	*318KB*	*103KB*

least for the considered set of benchmarks, the most cost-effective metric to measure similarity between partial logs. Comparing to *PSIM*, *DSIM* allowed CoopREP to reproduce the bug in the same or in a smaller number of attempts for all 9 test cases, thus providing more accuracy for a similar execution time.

Comparing to *DHS*, albeit *DSIM* has shown the same bug replay ability, it required much lower execution time to compute the similarity of a pair of partial logs. In fact, *DSIM* was up to 1252x faster than *DHS* to measure similarity for 1GB logs.

– *Which CoopREP heuristic replays the bug in fewer attempts? How do the heuristics compare to previous state-of-the-art deterministic replay systems?*

Section 3.5.3 evaluates CoopREP's effectiveness by providing a three-fold comparison. First, we compared the effectiveness of the two CoopREP heuristics: *SGM* and *DGM*. Next, we compared CoopREP against LEAP and PRES-BB in terms of bug replay ability. Finally, we compared the partial log combination approach against that of tracing only one SPE per partial log and producing a complete replay log by computing the access vectors most correlated to the error.

The results of the first comparison show that *DGM* tends to be more effective than *SGM* (the former replayed the bug in fewer attempts than the latter in 10 out of 39 test cases, whereas the opposite scenario was only verified in 3 of the 39 cases). In particular, *DGM* was able to reproduce the failure in very few attempts (more precisely, just 1 attempt for most cases) when the information included in the partial partial logs encompassed all the most disperse SPEs. When this did not happen, *SGM* was a better choice to find a feasible replay log, because it was quicker in pinpointing groups of similar partial logs.

The results of the second comparison show that CoopREP achieves similar replay ability to LEAP and PRES-BB. In fact, CoopREP$_L$ (*i.e.*, CoopREP combined with LEAP) was able to reproduce the error at the first attempt for 11 out of the 13 benchmarks. In turn, CoopREP$_P$ (*i.e.*, CoopREP combined with PRES-BB) was always able to match the same number of replay attempts as the corresponding non-cooperative PRES-BB solution.

The results of the third comparison show that using heuristics to combine partial logs is clearly better than using the *1SPE* approach for the large majority of the cases. However, if the program exhibits identical access vectors for almost all SPEs (*i.e.*, has a very low SPE dispersion), then tracing only one SPE per partial log and using statistical indicators is sufficient to typically reproduce the failure at the first attempt.

– How does CoopREP's effectiveness vary with the number of logs collected?

Section 3.5.4 evaluates the impact of varying the number of logs collected on the number of attempts required by CoopREP heuristics to replay the bug. The results show that, for programs with low SPE dispersion, the number of logs does not affect the effectiveness of the heuristics. However, for programs with high SPE dispersion, the smaller the number of collected logs, the more difficult it becomes to replay the bug. Concretely, for the program with highest SPE dispersion in our experiments, the bug was only replayed with less than 500 partial logs when each partial log contained 75% of the total SPEs of the program.

– How much runtime overhead reduction can CoopREP achieve with respect to other classic, non-cooperative logging schemes?

The results described in Section 3.5.5 show that CoopREP is, indeed, able to achieve lower recording overhead than previous solutions. For the most effective recording configuration in each test case, CoopREP incurred lower runtime penalty than both PRES-BB and LEAP. Overall, CoopREP imposed, on average, 3x less overhead than PRES-BB (achieving a maximal reduction of 13x) and LEAP (achieving maximal reduction of 6x).

– How much can CoopREP decrease the size of the logs generated?

The results for the space overhead presented in Section 3.5.6 follow an analogous trend to those of the runtime overhead: CoopREP generated smaller logs when compared to PRES-BB and LEAP. Concretely, for a similar bug replay ability, CoopREP's partial logs were, on average, 3x smaller than the logs produced by PRES-BB and LEAP.

Summary

This chapter addressed the challenge of deterministically replaying concurrency bugs for debugging purposes. We introduced CoopREP, a system that provides failure replication of concurrent programs via cooperative logging and partial log combination. CoopREP achieves significant reductions of the overhead incurred by other state-of-the art systems by letting each instance of a program trace only a subset of its shared programming elements (*e.g.*, variables or synchronization primitives). CoopREP relies on several innovative statistical analysis techniques aimed at guiding the search of partial logs to combine and use during the replay phase.

The evaluation study, performed with third-party benchmarks and real-world applications, highlighted the effectiveness of the proposed technique, in terms of its ability to successfully replay non-trivial concurrency bugs, as well as its performance advantages with respect to non-cooperative logging schemes.

Although CoopREP is useful to debug concurrency bugs by overcoming their non-deterministic nature, it still does not provide any hints with respect to the actual events that caused the failure. In the next chapter, we address the issue of isolating the root cause of concurrency bugs by proposing *differential schedule projections*.

4 Symbiosis: Root Cause Isolation with Differential Schedule Projections

The ability to deterministically reproduce a failure is undoubtedly useful for debugging concurrency bugs. However, re-enacting a failing schedule *per se* does not provide any clues on the exact events that caused the program to fail. Since any operation in any thread may have led to the failure, blindly analyzing a full schedule to isolate the root cause of the bug often remains a daunting task.

In this chapter, we present Symbiosis, a system that helps finding and understanding a failure's root cause, as well as fixing the underlying bug. Symbiosis starts from a failing schedule and uses symbolic execution and SMT constraint solving to generate a very similar non-failing schedule. Then, Symbiosis applies a differential analysis to report *only the important ordering and data-flow differences* between failing and non-failing schedules. These differences are data-flows between operations in the failing execution that do not occur in the correct execution and vice versa. We call the output of our novel debugging approach a *differential schedule projection* (DSP).

DSPs simplify debugging for two main reasons. First, by showing only what differs between a failing and non-failing schedule, the programmer is exposed to a very small number of relevant operations, rather than a full schedule. Second, DSPs illustrate not only the failing schedule, but also the way the execution *should* behave, if not to fail. Seeing the different event orders side-by-side helps understand the failure and, often, how to fix the bug.

We believe DSPs to be a significant improvement with respect to more traditional debugging approaches, such as *cyclic debugging* (*i.e.*, iteratively re-execute a program's failing execution in an attempt to understand the bug and narrow its root cause). Furthermore, Symbiosis produces a DSP from a *single* failing schedule, enabling its use for in-production failures that manifest rarely. This contrasts to prior work [Lucia & Ceze 2013; Kasikci, Schubert, Pereira, Pokam, & Candea 2015] that relies on statistical inference and, therefore, needs to capture information from a significant amount of failing executions in order to isolate the bug's root cause effectively.

Our evaluation in Section 4.3 shows that DSPs have, on average, 90% fewer events than full schedules and shows qualitatively, with case studies, that DSPs help understand failures and fix bugs. Furthermore, we conducted a user study with 48 participants to further support the claim that DSPs allow for faster bug diagnosis.

The rest of the chapter is organized as follows. Section 4.1 describes the Symbiosis system in detail, namely its components and how it operates to produce DSPs. Section 4.2 reports the implementation details. Section 4.3 presents the results of both the experimental evaluation and the user study, and discusses the main findings. Finally, Section 4.3.3.4 summarizes this chapter's main points.

4.1 Symbiosis

In this section, we start by presenting an overview of Symbiosis and how it operates. Then, we describe in detail each component of our system.

4.1.1 Overview

Symbiosis is a technique for concisely reporting the root cause of a failing multithreaded execution, alongside a non-failing, alternate execution of the events that make up the root cause. Symbiosis produces *differential schedule projections*, which reveal bugs' root causes and aid in debugging. Symbiosis has six phases (see Figure 4.1):

1. Static Analysis. Symbiosis starts by performing a static program analysis with two goals. The first goal is to instrument the beginning of each basic block in the program to trace the control-flow path followed in a concrete execution. The second goal is to identify shared variables. Non-private (*i.e.*, shared) variables and local variables derived from those variables are marked as symbolic. Marking shared variables as symbolic is a pre-requisite to Symbiosis's symbolic trace collection and generation mechanism (described next).

2. Symbolic trace collection. In a concrete failing program run, Symbiosis traces the sequence of basic blocks executed by each thread independently. The per-thread path profiles are used to guide symbolic execution, producing a set of per-thread traces with symbolic information (*e.g.*, path conditions and read-write accesses to shared variables).

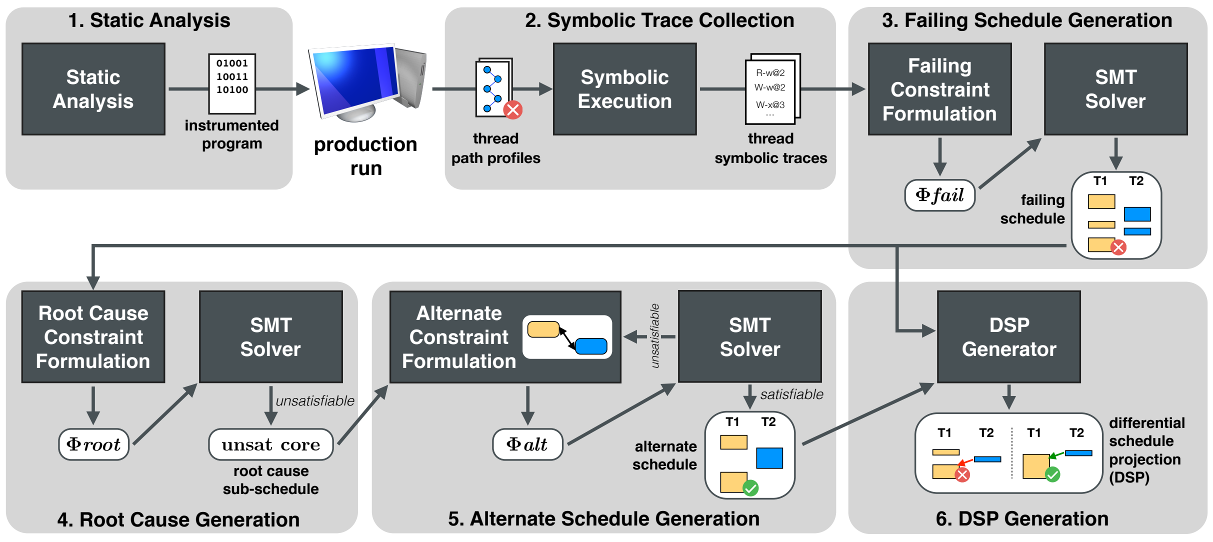


Figure 4.1: Overview of Symbiosis.

3. Failing schedule generation. Symbiosis produces a Satisfiability Modulo Theories (SMT) constraint formula over the information in the symbolic execution traces. The formula includes constraints that represent each thread's path, as well as the failure's manifestation, memory access orderings, and synchronization orderings. The solution to the SMT formula corresponds to a complete, failing, multithreaded execution. In other words, this solution specifies the ordering of events that triggers the error.

4. Root cause generation. Symbiosis produces an SMT formula corresponding to the symbolic traces, but specifies that the execution should not fail, by negating the failure condition. Combined with the constraints representing the order of events in the full, failing schedule, the SMT instance is unsatisfiable. The SMT solver produces an UNSAT core that contains the constraints representing the execution event orders that conflict with the absence of the failure. Since those event orders are necessary for the failure to occur, they comprise the failure's root cause sub-schedule.

5. Alternate schedule generation. Symbiosis examines each pair of events from different threads in the root cause sub-schedule. For each pair, Symbiosis produces a new SMT formula, identical to the one used to find the root cause, but with constraints implying the ordering of the events in the pair reversed. When Symbiosis finds an instance that is satisfiable, the corresponding schedule is very similar to the failing schedule[1], but does not fail. Symbiosis

[1] By similar we mean that the alternate schedule comprises the same events as the failing schedule and adheres to the same execution path.

reports the alternate, non-failing schedule that is identical to the failing schedule, but with the pair of events reordered.

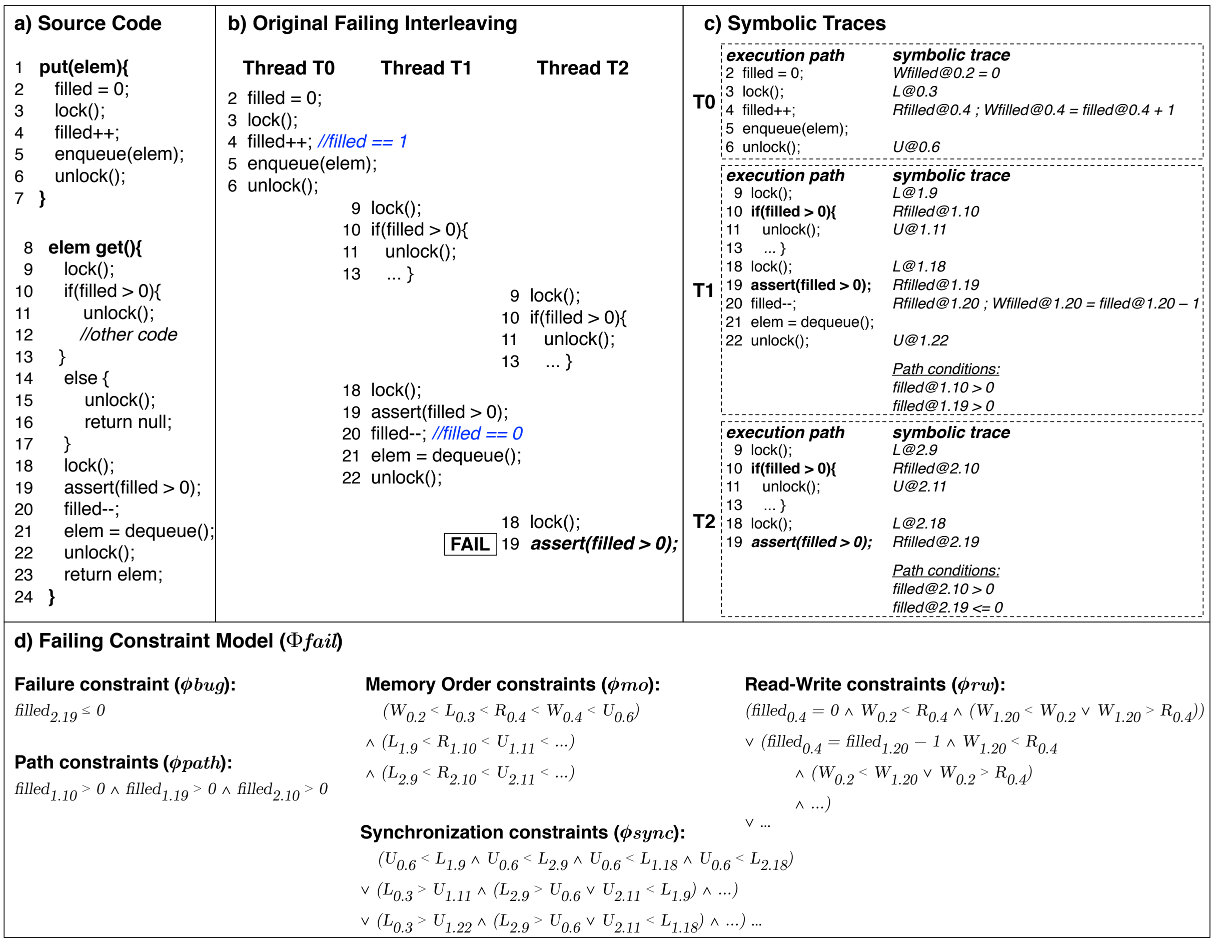
The experimental results in Section 4.3 indicate that this technique is effective, as Symbiosis was able to find a non-failing schedule by reordering less than 10 pairs of events for 10 out of 13 test cases.

6. Differential schedule projection generation. Symbiosis produces a differential schedule projection by comparing the failing schedule and the alternate, non-failing schedule. The DSP shows how the two schedules differ in the order of their events and in their data-flow behavior. Additionally, as the reordered pair from the alternate non-failing schedule eliminates an event order necessary for the failure to occur, it can be leveraged by a dynamic failure avoidance system [Lucia & Ceze 2013] to prevent future failures.

To better illustrate the main concepts of Symbiosis, we use a running example that consists of the modified version of *Pfscan* file scanner studied in prior work [Elmas, Burnim, Necula, & Sen 2013]. A slightly simplified snippet of the program's code is depicted in Figure 4.2a. The program uses three threads. The first thread enqueues elements into a shared queue. The two other threads attempt to dequeue elements, if they exist. A shared variable, named *filled*, records the number of elements in the queue. The code in the *get* function checks that the queue is non-empty (reading *filled* at line 10), decreases the count of elements in the queue (updating *filled* at line 20), then dequeues the element.

The code has a concurrency bug because it does not ensure that the check and update of *filled* execute atomically. The lack of atomicity permits some unfavorable execution schedules in which the two consumer threads both attempt to dequeue the queue's last element. In that problematic case, both consumers read that the value of *filled* is 1, passing the test at line 10. One of the threads proceeds to decrement *filled* and dequeue the element. The other reaches the assertion at line 19, reads the value 0 for filled, fails, terminating the execution. Figure 4.2b shows the interleaving of operations that leads to the failure in a concrete execution.

The next sections show how Symbiosis starts from a concrete failing execution (like the one in Figure 4.2b), computes a focused root cause, and produces a DSP to aid in debugging.



a) Source Code

```
1  put(elem){
2     filled = 0;
3     lock();
4     filled++;
5     enqueue(elem);
6     unlock();
7  }

8  elem get(){
9     lock();
10    if(filled > 0){
11       unlock();
12       //other code
13    }
14    else {
15       unlock();
16       return null;
17    }
18    lock();
19    assert(filled > 0);
20    filled--;
21    elem = dequeue();
22    unlock();
23    return elem;
24 }
```

b) Original Failing Interleaving

```
Thread T0              Thread T1              Thread T2
2 filled = 0;
3 lock();
4 filled++; //filled == 1
5 enqueue(elem);
6 unlock();
                       9  lock();
                       10 if(filled > 0){
                       11    unlock();
                       13    ... }
                                              9  lock();
                                              10 if(filled > 0){
                                              11    unlock();
                                              13    ... }
                       18 lock();
                       19 assert(filled > 0);
                       20 filled--; //filled == 0
                       21 elem = dequeue();
                       22 unlock();
                                              18 lock();
                                       FAIL   19 assert(filled > 0);
```

c) Symbolic Traces

```
T0  execution path        symbolic trace
    2 filled = 0;         Wfilled@0.2 = 0
    3 lock();             L@0.3
    4 filled++;           Rfilled@0.4 ; Wfilled@0.4 = filled@0.4 + 1
    5 enqueue(elem);
    6 unlock();           U@0.6

T1  execution path        symbolic trace
     9 lock();            L@1.9
    10 if(filled > 0){    Rfilled@1.10
    11    unlock();       U@1.11
    13    ... }
    18 lock();            L@1.18
    19 assert(filled > 0);  Rfilled@1.19
    20 filled--;          Rfilled@1.20 ; Wfilled@1.20 = filled@1.20 – 1
    21 elem = dequeue();
    22 unlock();          U@1.22

                          Path conditions:
                          filled@1.10 > 0
                          filled@1.19 > 0

T2  execution path        symbolic trace
     9 lock();            L@2.9
    10 if(filled > 0){    Rfilled@2.10
    11    unlock();       U@2.11
    13    ... }
    18 lock();            L@2.18
    19 assert(filled > 0);  Rfilled@2.19

                          Path conditions:
                          filled@2.10 > 0
                          filled@2.19 <= 0
```

d) Failing Constraint Model ($\Phi fail$)

Failure constraint (ϕbug):

$filled_{2.19} \leq 0$

Path constraints ($\phi path$):

$filled_{1.10} > 0 \wedge filled_{1.19} > 0 \wedge filled_{2.10} > 0$

Memory Order constraints (ϕmo):

$(W_{0.2} < L_{0.3} < R_{0.4} < W_{0.4} < U_{0.6})$
$\wedge (L_{1.9} < R_{1.10} < U_{1.11} < ...)$
$\wedge (L_{2.9} < R_{2.10} < U_{2.11} < ...)$

Synchronization constraints ($\phi sync$):

$(U_{0.6} < L_{1.9} \wedge U_{0.6} < L_{2.9} \wedge U_{0.6} < L_{1.18} \wedge U_{0.6} < L_{2.18})$
$\vee (L_{0.3} > U_{1.11} \wedge (L_{2.9} > U_{0.6} \vee U_{2.11} < L_{1.9}) \wedge ...)$
$\vee (L_{0.3} > U_{1.22} \wedge (L_{2.9} > U_{0.6} \vee U_{2.11} < L_{1.18}) \wedge ...) ...$

Read-Write constraints (ϕrw):

$(filled_{0.4} = 0 \wedge W_{0.2} < R_{0.4} \wedge (W_{1.20} < W_{0.2} \vee W_{1.20} > R_{0.4}))$
$\vee (filled_{0.4} = filled_{1.20} - 1 \wedge W_{1.20} < R_{0.4}$
$\wedge (W_{0.2} < W_{1.20} \vee W_{0.2} > R_{0.4})$
$\wedge ...)$
$\vee ...$

Figure 4.2: Running example. a) Source code. b) Concrete failing execution. c) Per-thread symbolic execution traces. d) Failing Constraint Model.

4.1.2 Symbolic Trace Collection

Like CLAP [Huang, Zhang, & Dolby 2013], Symbiosis avoids the overhead of directly recording the exact read-write linkages between shared variables that lead to a failure. Instead, Symbiosis collects only per-thread path profiles from a failing, concrete execution. As in prior work [Huang, Zhang, & Dolby 2013], Symbiosis's path profile for a thread consists of the sequence of executed basic blocks for that thread in the failing execution.

Symbiosis uses the per-thread path profiles to guide a symbolic execution of each thread and to produce each thread's separate symbolic execution trace. Symbolic execution normally explores all paths, following the path along both branch outcomes. Symbiosis, in contrast, guides the symbolic execution to correspond to the per-thread path profiles by considering only paths that are compatible with the basic block sequence in the profile. As symbolic execution proceeds, Symbiosis records information about control-flow, failure manifestation, synchronization, and shared memory accesses in each per-thread symbolic execution trace. Together, the traces are compatible with the original, failing, multithreaded execution.

Each per-thread symbolic execution trace contains four kinds of information. First, each trace includes a path condition that permits the failure to occur. A trace's path condition is the sequence of control-flow decisions made during the thread's respective execution. Second, the trace for the thread that experienced the failure must include the event that failed (*e.g.*, the failing assertion). Third, the trace must record synchronization operations, noting their type (*e.g.*, lock, unlock, wait, notify, fork, join, *etc.*), and the synchronization variable involved (*e.g.*, the lock address), if applicable. Fourth, the trace must record loads from and stores to shared memory locations. A key aspect of the shared memory access traces is that these are *symbolic*: loads always read fresh symbolic values and stores may write either symbolic or concrete values. Recall from Section 2.4.1 that a symbolic value holds the last operation that manipulated a value. Also, a symbolic value may, itself, be an expression that refers to other symbolic or concrete values.

Figure 4.2c illustrates a symbolic trace collection for our running example: it shows the execution path followed by each thread for the failing schedule in Figure 4.2b and the corresponding symbolic trace produced by Symbiosis. Each path condition in the trace represents a control-flow outcome in the original execution (*e.g.*, *filled@2.10* $>$ *0* denotes that thread *T2* should read a value greater than zero from *filled* at line 10). Thread *T2*'s trace includes the assertion that leads to the failure. Each trace includes both symbolic and concrete values in their memory access traces, as well as synchronization operations from the execution. Note that we slightly simplified the threads' traces to keep the figure uncluttered. *enqueue* and *dequeue* also access shared data but we only show operations that manipulate *filled* and perform synchronization because they are sufficient to illustrate the failure.

Symbiosis can leverage any technique for collecting concrete path profiles and generating symbolic traces. In our implementation of Symbiosis that targets C/C++, we use a technique very similar to the front-end of CLAP [Huang, Zhang, & Dolby 2013]: Symbiosis records a basic block trace and uses KLEE to generate per-thread symbolic traces conformant with the block sequence. Symbiosis for Java uses Soot [Vallée-Rai, Co, Gagnon, Hendren, Lam, & Sundaresan 1999] to instrument the program collect path profiles and JPF [Visser, Păsăreanu, & Khurshid

2004] for symbolic execution. The implementation details are described in Section 4.2.

4.1.3 Failing Schedule Generation

The symbolic, per-thread traces do not explicitly encode the multithreaded schedule that led to the failure. Symbiosis uses the information in the symbolic traces to construct a system of SMT constraints that encode information about the execution. The solution to those SMT constraints corresponds to a multithreaded schedule that ends in failure and is compatible with each per-thread symbolic trace.[2] This section describes how the constraints are computed.

The SMT constraints refer to two kinds of unknown variables, namely the *value variables* for the fresh symbolic symbols returned by read operations and the *order variables* that represent the position of each operation from each trace in the final, multithreaded schedule. We notate value variables as $var_{t.l}$, meaning the value read from variable var by thread t at line l. We notate order variables as $Op_{t.l}$, meaning the order of instruction Op executed by thread t at line l, where Op can be a read (R), write (W), lock (L), unlock (U), or other synchronization operations such as wait/signal (our notation differs slightly from [Huang, Zhang, & Dolby 2013] for clarity).

Figure 4.2d shows part of the system of SMT constraints generated by Symbiosis for our running example from the symbolic traces presented in Figure 4.2c. The system, denoted Φ_{fail}, can be decomposed into five sets of constraints:

$$\Phi_{fail} = \phi_{bug} \wedge \phi_{path} \wedge \phi_{sync} \wedge \phi_{rw} \wedge \phi_{mo} \tag{4.1}$$

where ϕ_{bug} encodes the occurrence of the failure, ϕ_{path} encodes the control-flow path executed by each thread, ϕ_{sync} encodes possible inter-thread interactions via synchronization, ϕ_{rw} encodes possible inter-thread interactions via shared memory, and ϕ_{mo} encodes possible operation reorderings permitted by the memory consistency model. The following paragraphs explain how Symbiosis derives each set of constraints from the symbolic execution traces.

[2]Note that technique used by Symbiosis to obtain a failing schedule is orthogonal to its ability to perform root cause isolation. Thus, we could have also used CoopREP along with a general R&R technique to generate the failing schedule and then apply Symbiosis (namely steps 4, 5, and 6 of Figure 4.1). However, since Symbiosis relies on symbolic execution and SMT constraint solving to produce DSPs, we opted for using CLAP's approach to generate the failing schedule to begin with.

Failure Constraint (ϕ_{bug}) The failure SMT constraint expresses the failure's necessary conditions. The constraint is an expression over value variables for symbolic values returned by some subset of read operations (*e.g.*, those in the body of an `assert` statement). Figure 4.2d shows ϕ_{bug} for the running example, representing the sufficient condition for the assertion in thread *T2* to fail.[3]

Path Constraint (ϕ_{path}) The path SMT constraint encodes branch outcomes during symbolic execution. Symbiosis gathers path conditions by recording the branch outcomes along the basic block trace from the concrete path profile. A thread's path constraint is the conjunction of the path conditions for the execution of the thread in the symbolic trace. The ϕ_{path} constraint is the conjunction of all threads' path constraints. Each conjunct represents a single control-flow decision by constraining the value variables for one or more symbolic operands. In our running example, the shared variable *filled* is symbolic, resulting in a ϕ_{path} with three conjuncts. The three conjuncts express that the value of *filled* should be greater than 0 when thread *T1* executes lines 10 and 19, as well as when thread *T2* executes line 10. Figure 4.2d shows ϕ_{path} for our example.

Synchronization Constraints (ϕ_{sync}) There are two types of synchronization constraints: *partial order constraints* and *locking constraints.*

Partial order constraints represent the partial order of different threads' events resulting from *start/exit/fork/join/wait/signal* operations. Concretely, *start*, *join*, and *wait* operations are ordered with respect to *fork*, *exit*, and *signal* operations, respectively. The constraints for *start/fork* and *exit/join* are easy to model, as they exhibit a single mapping: the *start* event of a child thread must always occur after the corresponding *fork* operation in the parent thread, whereas the *exit* event of a child thread must always occur before the corresponding *join* operation in the parent thread. Let S_t represent the *start* event of a thread t, E_t the *exit* event of thread t, $F_{tp,tc}$ the *fork* operation of child thread tc by parent thread tp, and $J_{tp,tc}$ the *join* operation of thread tc by thread tp. Their partial order constraints for these operations are then written as follows:

[3] For calls to external libraries, we also mark the result of the calls as symbolic (Section 4.2).

$$F_{tp,tc} < S_{tc}$$
$$E_{tc} < J_{tp,tc}$$

The constraints for *wait* and *signal*, in contrast, are a little more complex. Similarly to CLAP [Huang, Zhang, & Dolby 2013], we use a binary variable that indicates whether a given *signal* operation is mapped to a *wait* operation or not. This is necessary because each *signal* operation can signal exactly one *wait* operation, although in theory it can be mapped to all *wait* operations on the same object. Let $\mathcal{SG}$ be the set with the *signal* operations sg on a given object and let $\mathcal{WT}$ be the set of *wait* operations wt on the same object, but belonging to a thread different from that of sg. Also, let Sg and Wt denote the order of sg and wt, respectively, and b^{sg}_{wt} be the binary variable denoting whether sg is mapped to wt or not. The corresponding partial order constraints are the following:

$$\begin{aligned}&(\bigvee_{\forall sg \in \mathcal{SG}} Sg < Wt \ \wedge \ b^{sg}_{wt} = 1)\\ &\bigwedge \sum_{wt \in \mathcal{WT}} b^{sg}_{wt} \leq 1\end{aligned}$$

The constraints above state that, if a signal operation sg is mapped to a wait operation wt (*i.e.*, $b^{sg}_{wt} = 1$), then sg must occur before wt and sg cannot signal any other *wait* operation rather than wt (*i.e.*, $\sum_{wt \in \mathcal{WT}} b^{sg}_{wt} \leq 1$).

Locking constraints represent the mutual exclusion effects of *lock* (L) and *unlock* (U) operations. Let $\mathcal{P}$ denote the set of locking pairs on a given locking object and consider a particular pair L/U. The locking constraints entail the possible orderings between L/U and all the remaining pairs in $\mathcal{P}$ and are written as follows:

$$\begin{aligned}&\bigwedge_{\forall L'/U' \in \mathcal{P}} U < L' \ \vee\\ &\bigvee_{\forall L'/U' \in \mathcal{P}} (U' < L \ \wedge \bigwedge_{\forall L''/U'' \in \mathcal{P},\ L''/U'' \neq L'/U'} U < L'' \ \vee \ U'' < L')\end{aligned}$$

The constraint above is a disjunction of SMT expressions representing two possible cases. In the first case, L/U is the first pair acquiring the lock and, therefore, U happens before L'. In the second case, L/U acquires the lock released by another pair L'/U', hence U' happens before L. Moreover, for any other pair L''/U'', either L'' acquires the lock after U or L' acquires the lock released by U''. Figure 4.2d shows a subset of the locking constraints for our running example that involve the *lock/unlock* pair of thread *T0* $(L_{0.3}, U_{0.6})$.

Read-Write Constraints (ϕ_{rw}) Read-write SMT constraints encode the matching between read and write operations that leads to a particular read operation reading the value written by a particular write operation. Read-write constraints model the possible inter-thread interactions via shared memory. A read-write constraint encodes that, for any read operation r on a shared variable v, if r is matched to a write w of the same variable, then the order variable (and hence execution order) for all other writes on v is either smaller than that of w or greater than that of r. The constraint also implies that r's value variable takes on the symbolic value written by w. Note that read-write SMT constraints are special in that they affect order variables and value variables.

Let r_v be the value returned by a read r on v, and let W be a set of writes on v. Using R to denote the order of r and W_i the order of write w_i in $\mathcal{W}$, ϕ_{rw} can be written as follows:

$$\bigvee_{\forall w_i \in \mathcal{W}} (r_v = w_i \ \wedge \ W_i < R \bigwedge_{\forall w_j \in \mathcal{W}, w_j \neq w_i} W_j < W_i \ \vee \ W_j > R)$$

For instance, in our running example, thread *T0* reads *filled* at line 4. If *T0* reads 0 at that point, then the most recent write to *filled* must be the one at line 2. The matching between that read and write implies that the order of the write must precede the read operation (*i.e.*, $W_{0.2} < R_{0.4}$), and that all the other writes to *filled* (*e.g.*, $W_{1.20}$) either occur before $W_{0.2}$ or after $R_{0.4}$. The same reasoning is also applied for the remaining reads of the program on symbolic variables.

Memory Order Constraints (ϕ_{mo}) The memory-order constraints specify the order in which instructions are executed in a specific thread. Although is possible to express different memory consistency models [Huang, Zhang, & Dolby 2013], in this thesis we opted not to focus on relaxed memory ordering, instead focusing on sub-schedule generation and differential schedule projections. Therefore, we encode memory order constraints for sequential consistency (SC) only, meaning that statements in a thread execute in program order. For the running example in Figure 4.2b, the memory order constraint requires that operations in thread *T0* respect the constraint $W_{0.2} < L_{0.3} < R_{0.4} < W_{0.5} < U_{0.6}$.

4.1.4 Root Cause Generation

Each order variable referred to by an SMT constraint represents the ordering of two program events from the separate single-threaded symbolic traces. A binding of truth values to the SMT order variables corresponds directly to an ordering of operations in the otherwise unordered, separate, per-thread traces. Solving the constraint system binds truth values to variables, producing a multithreaded schedule. The constraint system includes a constraint representing the occurrence of the failure, so the produced multithreaded schedule manifests the failure (ϕ_{bug}). Solving the generated SMT formulae, Symbiosis produces a full, failing, multithreaded schedule ϕ_{fsch}. The entire multithreaded schedule may be long, complex, and may contain information that is irrelevant to the root cause of the failure. Symbiosis uses a special SMT formulation to produce a *root cause sub-schedule* that prunes some operations in the full schedule, but preserves event orderings that are *necessary* for the failure to occur. To compute the root cause sub-schedule, Symbiosis generates a new constraint system, denoted Φ_{root}, that is *designed to be unsatisfiable* in a way that reveals the necessary orderings. Symbiosis leverages the ability of the SMT solver to produce an explanation, of why a formula was unsatisfiable, to report only those necessary orderings.

To build the root cause sub-schedule SMT formula, Symbiosis logically inverts the *failure constraint*, effectively requiring the failure not to occur (*i.e.*, $\neg\phi_{bug}$). Symbiosis adds constraints to the formula that directly encode the event orders in ϕ_{fsch} (*i.e.*, the full, failing schedule that was previously computed). The complete root cause sub-schedule formula is then written as follows:

$$\Phi_{root} = \neg\phi_{bug} \land \phi_{path} \land \phi_{sync} \land \phi_{rw} \land \phi_{mo} \land \phi_{fsch} \tag{4.2}$$

The original SMT formula that Symbiosis used to find the full failing schedule considers *all* possible multithreaded schedules that are consistent with the symbolic, per-thread schedules. In contrast, the root cause sub-schedule SMT formula adds the failing schedule ϕ_{fsch} constraint, accommodating *only* the full failing schedule. Combining the inverted failure constraint and the ordering constraints for the full failing schedule yields an unsatisfiable constraint formula: the inverted failure constraint requires the failure not to occur and the failing schedule's ordering constraints require the failure to occur.

When an SMT solver, like Z3, attempts to solve the unsatisfiable formula, it produces an unsatisfiable (UNSAT) core, which is a subset of constraint clauses that conflict, leaving the formula unsatisfiable. The UNSAT core for Φ_{root} encodes the subset of clauses that conflict because the ϕ_{fsch} requires the failure to occur and $\neg\phi_{bug}$ requires the failure not to occur. The event orderings that correspond to those conflicting constraints are the ones in ϕ_{fsch} that imply ϕ_{bug}. Those orderings are a necessary condition for the failure; their corresponding constraints, together with $\neg\phi_{bug}$ are responsible for the unsatisfiability of Φ_{root}. Reporting the sub-schedule corresponding to the UNSAT core yields fewer total events than are in the full, failing schedule, yet includes event orderings necessary for the failure.

Figure 4.3a shows a possible failing schedule produced by the constraint system corresponding to the execution depicted in Figure 4.2d. The failure constraint ϕ_{bug} requires the corresponding execution to manifest the failure. The generated path and memory access constraints are compatible with the failure and the system is satisfiable, producing the failing execution trace shown (ϕ_{fsch}). Note that Symbiosis inserts a *synthetic unlock* event *(*$U_{2.20}$*)* in the model, in order to preserve the correct semantics of synchronization constraints (see Section 4.2).

In Figure 4.3b, the failure constraint is *negated*, requiring the corresponding execution not to manifest the failure (*i.e.*, $filled_{2.19} > 0$ and the assertion at line 19 does not fail). On the other hand, ϕ_{fsch} satisfies only the data-flows encoded in ϕ_{rw} that correspond to the failing schedule, which means that $R_{2.19}$ is forced to receive the value written by $W_{1.20}$. Consequently, $filled_{2.19}$ becomes 0 instead of greater than 0, as required by the negated failure constraint (note that $R_{2.19}$ defines the value of $filled_{2.19}$ read by the assertion). Since both sub-formulae ϕ_{fsch} and $\neg\phi_{bug}$ conflict with each other, the solver yields unsatisfiable for this model. Alongside, the solver outputs the UNSAT core containing the subset of constraints of ϕ_{fsch} that conflict with $\neg\phi_{bug}$ (see Figure 4.3b). Note that the UNSAT core shows why Φ_{root} is unsatisfiable: the negated failure constraint conflicts with the subset of ordering constraints from ϕ_{fsch} that cause *filled* to be less than 0 when thread 2 executes line 19.

In our experience, the UNSAT core produced by Z3 is typically not minimal (although it

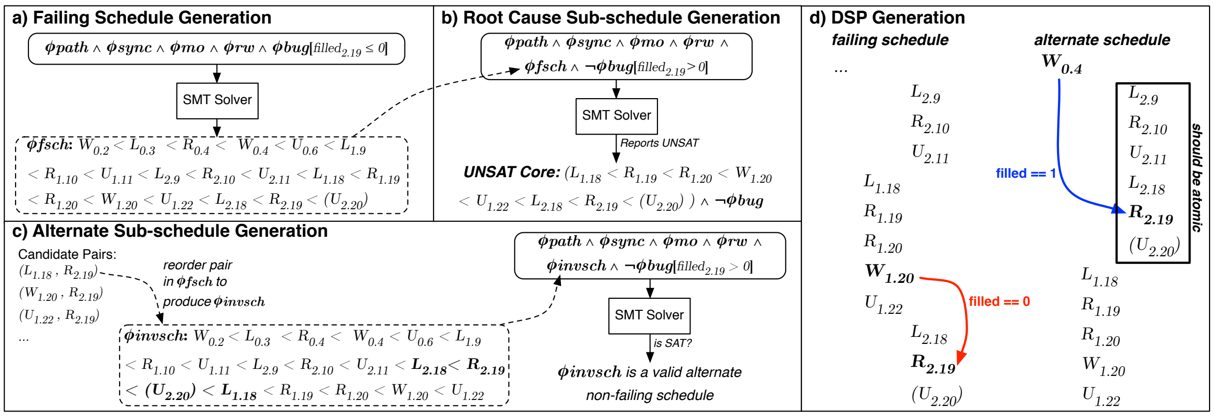

a) Failing Schedule Generation
φpath ∧ φsync ∧ φmo ∧ φrw ∧ φbug[filled2.19 ≤ 0]
SMT Solver
φfsch: W0.2 < L0.3 < R0.4 < W0.4 < U0.6 < L1.9 < R1.10 < U1.11 < L2.9 < R2.10 < U2.11 < L1.18 < R1.19 < R1.20 < W1.20 < U1.22 < L2.18 < R2.19 < (U2.20)
b) Root Cause Sub-schedule Generation
φpath ∧ φsync ∧ φmo ∧ φrw ∧ φfsch ∧ ¬φbug[filled2.19 > 0]
SMT Solver
Reports UNSAT
UNSAT Core: (L1.18 < R1.19 < R1.20 < W1.20 < U1.22 < L2.18 < R2.19 < (U2.20)) ∧ ¬φbug
c) Alternate Sub-schedule Generation
Candidate Pairs:
(L1.18 , R2.19)
(W1.20 , R2.19)
(U1.22 , R2.19)
...
reorder pair in φfsch to produce φinvsch
φinvsch: W0.2 < L0.3 < R0.4 < W0.4 < U0.6 < L1.9 < R1.10 < U1.11 < L2.9 < R2.10 < U2.11 < L2.18 < R2.19 < (U2.20) < L1.18 < R1.19 < R1.20 < W1.20 < U1.22
φpath ∧ φsync ∧ φmo ∧ φrw ∧ φinvsch ∧ ¬φbug[filled2.19 > 0]
SMT Solver
is SAT?
φinvsch is a valid alternate non-failing schedule
d) DSP Generation
failing schedule
alternate schedule
...
L2.9
R2.10
U2.11
L1.18
R1.19
R1.20
W1.20
U1.22
L2.18
R2.19
(U2.20)
W0.4
filled == 1
filled == 0
L2.9
R2.10
U2.11
L2.18
R2.19
(U2.20)
should be atomic
L1.18
R1.19
R1.20
W1.20
U1.22

Figure 4.3: Root cause and alternate schedule generation. a) Possible failing schedule produced by the SMT solver for the constraint system in Figure 4.2d ($(U_{2.20})$ represents a synthetic unlock event). b) Root cause sub-schedule, which corresponds to the UNSAT core produced by the solver. c) Candidate pair reordering and respective alternate schedule. d) Differential Schedule Projection generated by Symbiosis.

is always an overapproximation of the root cause). As a result, while helpful, an UNSAT core alone is not sufficient for debugging and necessitates a differential schedule projection to isolate a bug's root cause.

4.1.5 Alternate Schedule Generation

In addition to reporting the bug's root cause, Symbiosis also produces *alternate, non-failing schedules*. These alternate schedules are *non-failing* variants of the root cause sub-schedule, with the order of a single pair of events reversed. Alternate schedules are the key to building *differential schedule projections* (Section 4.1.6). Symbiosis generates alternate, non-failing schedules after it identifies the root cause. To generate an alternate schedule, Symbiosis selects a pair of events from different threads that were included in the bug's root cause. Symbiosis then generates a new constraint model, like the one used to identify the root cause. We call this model Φ_{alt}. The Φ_{alt} model includes the inverted failure constraint. The model also includes a set of constraints, denoted ϕ_{invsch}, that encode the original full, failing schedule, *except the constraint representing the order of the selected pair of events is inverted.* Inverting the order constraint for the pair of events yields the following new constraint model.

$$\Phi_{alt} = \neg\phi_{bug} \wedge \phi_{path} \wedge \phi_{sync} \wedge \phi_{rw} \wedge \phi_{mo} \wedge \phi_{invsch} \tag{4.3}$$

The new Φ_{alt} model corresponds to a different, full execution schedule in which the events in the pair occur in the order opposite to that in the full, failing schedule. If this new model is satisfiable, then reordering the pair of events in the full, failing schedule produces a new alternate schedule in which the failure does not manifest, as shown in Figure 4.3c.

If there are many event pairs in the root cause, then Symbiosis must generate and attempt to solve many constraint formulae. Symbiosis systematically evaluates a set of candidate pairs in a fixed order, choosing the pair separated by the fewest events in the original schedule first. The reasoning behind this design choice is that events in a pair that are further apart are less likely to be meaningfully related and, thus, less likely to change the failure behavior when their order is inverted. The experimental results in Section 4.3.3 show that this heuristic is effective for most cases.

By default, we configured Symbiosis to stop after finding a single alternate, non-failing schedule. However, the programmer can instruct Symbiosis to continue generating alternate schedules, given that studying sets of schedules may reveal useful invariants [Lucia, Wood, & Ceze 2011].

Arbitrary operation reorderings may yield infeasible schedules. Reordering may change inter-thread data flow, producing values that are inconsistent with a prior branch dependent on those values. The inconsistency between the data and the execution path makes the execution infeasible. Symbiosis produces only feasible schedules by including path constraints in its SMT model. If a reordering leads to inconsistency, the SMT path constraints become unsatisfiable and Symbiosis produces no schedule. We denote this property as *feasibility*.

Moreover, our event pair reordering technique has the property of guaranteeing that, if there exists a feasible alternate schedule that adheres to the original execution path and prevents the failure, Symbiosis finds it. We denote this property as *1-Recall*.[4]

The feasibility and 1-Recall properties are proved in the following paragraphs.

[4]Similarly to the definition in the field of information retrieval, we use *recall* to denote the fraction of relevant solutions (*i.e.*, solutions that prevent the failure and adhere to the same path constraints as the failing schedule) that Symbiosis successfully outputs. The *1-Recall* property, thus, indicates that all solutions output by Symbiosis are relevant.

4.1.5.1 Alternate Schedule Feasibility

Intuitively, what we want to show is that any alternate schedule, resulting from a reordering of events in a failing schedule, is feasible if deemed as satisfiable by Symbiosis's solver. To formally define the feasibility of a schedule, we rely on the concept of (sequential) consistency proposed in [Herlihy & Wing 1990] and adapt the notation used in [Huang, Meredith, & Rosu 2014]. Hence, a schedule is considered feasible if it is sequentially consistent.

Consider a *schedule* σ to be a sequence of *events* e. Events are operations performed by threads on *concurrent objects* (*e.g.*, shared variables, locks, etc) for data sharing and synchronization purposes. A concurrent object is behaviorally defined by a set of atomic operations and by a serial specification of its legal computations, when performed in isolation [Herlihy & Wing 1990]. For instance, a shared variable is a concurrent object containing read and write operations, whose serial specification states that each read returns the value of the most recent write.

Let σ_e denote the prefix of schedule σ up to e (inclusive): if $\sigma = \sigma_1 e \sigma_2$, then σ_e is $\sigma_1 e$. Moreover, let $\sigma_{[op,var,thread]}$ represent the restriction of σ to events involving operations of type op, on variable var, by thread $thread$. For instance, $\sigma_{[R,*,1]}$ represents the projection of schedule σ to read operations performed by thread 1 on all shared variables; $\sigma_{[W,v,*]}$ consists of the restriction of σ to write operations on shared variable v by any thread; etc.

An alternate schedule of σ is a schedule σ' that exhibits the same per-thread schedules as σ, but permits different orders of events from different threads: $\sigma'_{[*,*,t]} = \sigma_{[*,*,t]}$, for each thread t.

Schedule σ is *(sequentially) consistent* iff $\sigma_{[*,o,*]}$ satisfies o's serial specification for any concurrent object o [Herlihy & Wing 1990]. More formally, a schedule σ is sequentially consistent when it meets the following requirements:

Read Consistency. A read event returns the value written by the most recent write event on the same variable. Formally, if e is a read event in σ on variable v, then $data(e) = data(last_{write}(\sigma_{e[W,v,*]}))$, where $data(e)$ gives the value returned by the read event e, and $data(last_{write}(\sigma_{e[W,v,*]}))$ is the last value written to variable v in σ_e.

Lock Mutual Exclusion. Each *unlock* (U) event is preceded by a *lock* (L) event on the same lock object by the same thread, and a locking pair cannot be interleaved by any other L or U event on the same object. Formally, for any lock object l, if $\sigma_{[*,l,*]} = e_1e_2...e_n$ then $e_k = L$ for all odd indexes $k \leq n$, $e_k = U$ for all even indexes $k \leq n$, and $\mathit{thread}(e_k) = \mathit{thread}(e_{k+1})$ for all odd indexes k with $k < n$.

Must Happen-Before. Let $\mathit{start}(t)/\mathit{exit}(t)$ be the first/last event of thread t. Then, a *start* event in a thread t' can only occur in σ after t' is forked by thread t, *i.e.*, for any event $e = \mathit{start}(t')$ in σ, $\sigma_{[*,*,t']}$ begins with e and there exists precisely one $\mathit{fork}(t, t')$ event in σ_e. Similarly, a *join* event in a thread t can only occur in σ after t' has ended, *i.e.*, for any event $e = \mathit{exit}(t')$ in σ, $\sigma_{[*,*,t']}$ terminates with e and there exists precisely one $\mathit{join}(t, t')$ event in σ_e.

Note that branch conditions do not have serial specifications, hence they can affect the control flow of an execution, but not the consistency of its schedule. However, since we do not have information regarding operations in other execution paths rather than the one captured in the concrete trace, we conservatively assume that an alternate schedule must have the same control flow as the failing schedule (except for the assertion corresponding to the bug condition). Therefore, we can say that an alternate schedule is feasible iff it meets the aforementioned consistency requirements and adheres to the branch conditions of the failing schedule. We now prove the following theorem:

Theorem 1 (Feasibility)**.** *Given a feasible failing schedule σ, any alternate schedule σ' that is satisfiable by Symbiosis's solver is feasible.*

Proof. To prove the theorem above, we will first show that any alternate schedule that satisfies our SMT constraint model in Equation 4.3 is sequentially consistent, *i.e.*, it provides *read consistency*, *lock mutual exclusion*, and *must happen-before* properties. Then, we will show that any alternate schedule that is considered satisfiable by Symbiosis's solver, also satisfies the same path conditions as the failing schedule.

The *read consistency* property requires that a read event returns the value written by the most recent write event on the same variable. In our constraint model, this property holds from the read-write constraints. As shown in Section 4.1.3, the read-write constraints encode all possible linkages between reads and writes in the symbolic traces. This means that, for a given read event e on a shared variable v, the read-write formulae include a disjunction of constraints encoding the mapping between the value returned by e and all the existing writes on v. Hence, considering r_v to be the value returned by e in a schedule σ_e (*i.e.*, $r_v = data(e)$), there always exists a write w_l such that $w_l = data(last_{write}(\sigma_{e[W,v,*]}))$. In other words, w_l is the most recent write on v with respect to e and satisfies the following constraint: $r_v = w_l \wedge W_l < R_e \bigwedge_{\forall w_j \in \mathcal{W}, w_j \neq w_l} W_j < W_l \ \vee \ W_j > R_e$, where capital letters signify order variables, *i.e.*, W_l and R_e indicate the order of write w_l and read event e in schedule σ, respectively.

The *lock mutual exclusion* property holds from the locking constraints in our SMT model, as they encode the mutual exclusion effects of the acquisition and release of lock objects. To prove this, we show that any locking order that satisfies our locking constraint formulae is of form $L_1U_1L_2U_2L_kU_k...L_nU_n, \forall_{k \le n}$, with $thread(L_k) = thread(U_k)$, as required by the lock mutual exclusion property. Recall the locking constraints for a locking pair L/U on a lock object l in a thread t[5], as shown in Section 4.1.3:

$$
\begin{array}{ll}
i) & \displaystyle\bigwedge_{\forall L'/U' \in \mathcal{P}} U < L' \ \vee \\
ii) & \displaystyle\bigvee_{\forall L'/U' \in \mathcal{P}} (U' < L \ \wedge \bigwedge_{\forall L''/U'' \in \mathcal{P},\ L''/U'' \neq L'/U'} U < L'' \ \vee \ U'' < L')
\end{array}
$$

According to our SMT constraint system, it must be the case that either *i*) L/U is the first locking pair in the schedule or *ii*) it acquires the lock on l released by a previous locking pair L'/U' (and, here, all the other locking pairs either occur before L'/U' or after L/U). Let k denote the order in which the pair L/U holds the lock in a given schedule, *i.e.*, L_k/U_k is the k^{th} pair acquiring the lock in the schedule. If $k = 1$, then L_k/U_k is the first pair acquiring the lock on l (*i.e.*, L_1/U_1), which means that the portion of the locking constraint formula for L_1/U_1 that will be true is *i*): $U_1 < L_2 \wedge U_1 < L_3 \wedge ... \wedge U_1 < L_n$.

In turn, when $1 < k \le n$, the pair L_k/U_k will acquire the lock released by the pair L_{k-1}/U_{k-1}, thus satisfying the constraint sub-formula *ii*) instead: $U_{k-1} < L_k \bigwedge_{\forall 1 \le j \le n, j \neq k, j \neq k-1}, U_k < L_j \vee U_j < L_{k-1}$. However, note that, when $k = n$, the constraint will be of type: $U_1 < L_n \wedge U_2 < L_n \wedge ... \wedge U_{n-1} < L_n$, meaning the pair L_n/U_n is the last one acquiring the lock.

In sum, since the constraints enforce that a given pair L_k/U_k can only acquire the lock released by a single pair L_{k-1}/U_{k-1}, it follows that each locking pair will have a unique value of

[5]Note that our locking constraints operate over the locking pairs extracted from each thread's symbolic traces, therefore it is always the case that $thread(L) = thread(U)$ for every pair L/U.

k, *i.e.*, a unique position in the schedule. Therefore, any global locking order that satisfies our locking constraints is of the form $L_1U_1L_2U_2L_kU_k...L_nU_n, \forall_{k\leq n}$.

The *must happen-before* property follows directly from the partial order constraints described in Section 4.1.3: the operations S_t, E_t, $F_{tp,tc}$, $J_{tp,tc}$ correspond, respectively, to the events *start*(t), *exit*(t), *fork*(t,t'), and *join*(t,t') mentioned in the beginning of this section. Therefore, the partial order constraints in our SMT model directly encode the necessary happen-before guarantees required for a schedule to be sequentially consistent according to [Herlihy & Wing 1990]. □

4.1.5.2 Alternate Schedule 1-Recall

In addition to feasibility, another important property that we are interested in proving is the *1-Recall* property of Symbiosis with respect to finding alternate, non-failing schedules via event pair reordering. The 1-Recall property can be defined as the following theorem:

Theorem 2 (1-Recall)**.** *Given a failing schedule σ, let $\mathcal{A}$ be the set of feasible alternate schedules of σ that adhere to the same execution path and prevent the failure. Let $\mathcal{S}$ be the set of alternate schedules output by Symbiosis.*[6] *If $|\mathcal{A}| \geq 1$, then $|\mathcal{S}| \geq 1 \wedge \mathcal{S} \subseteq \mathcal{A}$.*

Proof. Informally, the above theorem states that, given a feasible failing schedule σ, if there exists a feasible (non-failing) alternate schedule σ', then Symbiosis will find it.

We divide the proof of the theorem in three steps. First, we define the condition necessary and sufficient that any alternate schedule $\sigma_a \in \mathcal{A}$ must verify in order not to trigger the failure. Second, we show that any alternate schedule σ' output by Symbiosis meets this condition (*i.e.*, $\mathcal{S} \subseteq \mathcal{A}$). Third, we show that, if there exist feasible alternate schedules that prevent the failure, Symbiosis finds at least one of them (*i.e.*, $|\mathcal{A}| \geq 1 \Rightarrow |\mathcal{S}| \geq 1$).

For a given failing execution, let $\mathcal{F}$ be the *minimal* set of events that are sufficient to trigger the concurrency failure, $\sigma_{[\mathcal{F}]}$ be the projection of the failing schedule σ to the events in $\mathcal{F}$ (*i.e.*, $\sigma_{[\mathcal{F}]}$ is the minimal ordered sequence of events that causes the concurrency failure), and $\sigma_{a[\mathcal{F}]}$ be the projection of the alternate schedule $\sigma_a \in \mathcal{A}$ to the events in $\mathcal{F}$.

[6]We say that an alternate schedule is output by Symbiosis iff it is deemed as satisfiable by the solver.

It has been shown that if $\sigma_{a[\mathcal{F}]} \neq \sigma_{[\mathcal{F}]}$, then σ_a does not trigger the failure [Lucia & Ceze 2013]. This result has been named *Avoidance-Testing Duality* and, informally, states that, given *any* ordered sequence of events that trigger a concurrency failure, it suffices to perturb just *one* pair of events to avoid the failure [Lucia & Ceze 2013].

Since, for any alternate schedule $\sigma_a \in \mathcal{A}$, $\sigma_{a[\mathcal{F}]} \neq \sigma_{[\mathcal{F}]}$ must hold, to prove that $\mathcal{S} \subseteq \mathcal{A}$, using the result above, we only need to show that $\forall_{\sigma' \in \mathcal{S}}, \sigma'_{[\mathcal{F}]} \neq \sigma_{[\mathcal{F}]}$.

Consider $\mathcal{R}$ to be the set of events belonging to the root cause produced by the constraint formula Φ_{root} (see Equation 4.2). Recall that Symbiosis generates alternate schedules by (exhaustively) selecting pairs of events from $\mathcal{R}$ to be inverted in the original failing schedule.

Let $\sigma' = invert(\sigma, e_j, e_k)$ be the alternate schedule σ' that Symbiosis produces by reordering the j^{th} and k^{th} events in σ, with $j < k$. Considering t_k as the thread of event e_k (*i.e.*, $t_k = thread(e_k)$) and rewriting $\sigma = e_1e_2...e_{j-1}e_je_{j+1}...e_{k-1}e_ke_{k+1}...e_n$ as $\sigma = \alpha e_j \beta e_k \gamma$, then $\sigma' = invert(\sigma, e_j, e_k) = \alpha \beta_{[*,*,t_k]} e_k e_j \beta \setminus \beta_{[*,*,t_k]} \gamma$. Here, $\beta_{[*,*,t_k]}$ corresponds to the events by thread t_k that occur between e_j and e_k in σ, and $\beta \setminus \beta_{[*,*,t_k]}$ corresponds to the set of events β excluding the events in $\beta_{[*,*,t_k]}$. In other words, the alternate schedule σ' is computed by placing e_k right before e_j, as well as all the events, belonging to the same thread of e_k, that occur between e_j and e_k in the failing schedule σ.

Let now $e_{f_j} and e_{f_k}$ be a pair of events selected by Symbiosis to be inverted, such that $e_{f_j}, e_{f_k} \in \mathcal{F}$. Note that we know that $\exists_{e_j,e_k \in \mathcal{R}} : e_j, e_k \in \mathcal{F}$ because the UNSAT core output by the solver (which corresponds to $\mathcal{R}$) always contains, at least, the events belonging to the minimal sequence of events that leads to the bug. Therefore, $\mathcal{F} \subseteq \mathcal{R}$ and $\exists_{e_j,e_k \in \mathcal{R}} : e_j, e_k \in \mathcal{F}$ holds by construction.

If $\sigma_{[\mathcal{F}]} = \alpha_f e_{f_j} \beta_f e_{f_k} \gamma_f$ is the minimal ordered sequence of events that causes σ to fail, and $\sigma' = invert(\sigma, e_{f_j}, e_{f_k})$ is the alternate schedule output by Symbiosis, then $\sigma'_{[\mathcal{F}]} = \alpha_f \beta_{f[*,*,thread(e_{f_k})]} e_{f_k} e_{f_j} \beta_f \setminus \beta_{f[*,*,thread(e_{f_k})]} \gamma_f$. Thus, $\sigma'_{[\mathcal{F}]} \neq \sigma_{[\mathcal{F}]}$ is true and $\sigma' \in \mathcal{A}$.

On the other hand, note that, for all event pairs $e_j, e_k \in \mathcal{R}$ and $\sigma'' = invert(\sigma, e_j, e_k)$, if $e_j, e_k \notin \mathcal{F}$, then $\sigma''_{[\mathcal{F}]} = \sigma_{[\mathcal{F}]}$ will hold and the solver will yield *unsatisfiable*. Thus, $\sigma'' \notin \mathcal{S}$ and it is not output by Symbiosis.

Finally, we prove that $|\mathcal{A}| \geq 1 \Rightarrow |\mathcal{S}| \geq 1$ by contradiction. Suppose $|\mathcal{A}| \geq 1$ and $\mathcal{S} = \emptyset$, then it must be the case that there is a feasible non-failing, alternate schedule σ_a that verifies the condition $\sigma_{a[\mathcal{F}]} \neq \sigma_{[\mathcal{F}]}$ and is not obtainable by reordering pairs of events in $\mathcal{R}$ (given that Symbiosis attempts to invert all pairs of events in $\mathcal{R}$). However, if σ_a is not the result of a reordering of events in $\mathcal{R}$, then it does not comprise events from $\mathcal{F}$ (since $\mathcal{F} \subseteq \mathcal{R}$). Consequently, σ_a cannot belong to $\mathcal{A}$, because it does not include the same set of events nor adheres to the same execution path as the original failing schedule σ, as required by the definition of $\mathcal{A}$. This contradicts our assumption. $\square$

As final remark, note that Symbiosis requires the alternate, non-failing schedule to adhere to the same control-flow as the original failing schedule. This means that, for concurrency bugs whose root cause is related to schedule-sensitive branches [Huang & Rauchwerger 2015] (*i.e.*, for path- and schedule-dependent bugs), Symbiosis is not able to produce an alternate schedule. In Chapter 5, we show how to generate alternate, non-failing schedules that follow an execution path different than that of the failing schedule.

4.1.6 Differential Schedule Projection Generation

Differential schedule projection (DSP) is a novel debugging methodology that uses root cause sub-schedules and non-failing alternate schedules. The key idea behind debugging with a DSP is to show the programmer the salient differences between failing, root cause schedules and non-failing, alternate schedules. Examining those differences helps the programmer understand how to *fix* the bug, rather than helping them understand the failure only, like techniques that solely report failing schedules.

Concretely, a DSP consists of a pair of sub-schedules decorated with several pieces of additional information. The first sub-schedule is the root cause sub-schedule, which is the *source* of the projection. The second sub-schedule is an excerpt from the alternate, non-failing schedule, which is the *target* of the projection.

The order of memory operations differs between the schedules and, as a result, the outcome of some memory operations may differ. A read may observe a different write's result in one schedule than it observed in another, or two writes may update memory in a different order in one schedule than in another, leaving memory in a different final state. These differences are precisely the changes in data-flow that contribute to the failure's occurrence. Symbiosis highlights the differences by reporting *data-flow variations*: data-flow between operations in the source sub-schedule that do not occur in the target sub-schedule and vice versa.

To simplify its output, Symbiosis reports only a subset of operations in the source and target sub-schedules. An operation is included if it makes up a data-flow variation or if it is one of a pair of operations that occur in a different order in one sub-schedule than in the other. Alternate, non-failing schedules vary in the order of a single pair of operations, so all operations that precede both operations in the pair occur in the same order in the source and target sub-schedules. Symbiosis does not report operations in the common prefix, unless it is involved in a data-flow variation. By selectively including only operations related to data-flow and ordering differences, a DSP focuses programmer attention on the changes to a failing execution that lead to a non-failing execution. Understanding those changes are the key to changing the program's code to fix the bug. For instance, the DSP in Figure 4.3d shows that the data-flow $W_{1.20} \rightarrow R_{2.19}$ (in ϕ_{fsch}) changes to $W_{0.4} \rightarrow R_{2.19}$ (in ϕ_{invsch}). This data-flow variation is the failure's root cause. In addition, note that, by reordering the events, the DSP also suggests that the block of operations $L_{2.9}$–$(U_{2.20})$ should execute atomically, which indeed fixes the bug.

4.1.7 DSP Optimization: Context Switch Reduction

The SMT solver does not take into account the number of context switches when solving the failing constraint system. As a consequence, the failing schedule produced may exhibit a fine-grained entanglement of thread operations, which hinders analysis. Although DSPs help obviate most of the interleaving complexity by pruning the common portions between the failing and the alternate schedules, they may still depict unnecessary data-dependencies. Figure 4.4a illustrates this scenario.

The program in the figure contains two threads (*T1* and *T2*), three shared variables (x, y, and z) and an assertion that checks whether $x \neq 0$. The DSP in Figure 4.4a shows a possible failing schedule and depicts the data-flow variations with respect to the corresponding alternate schedule. We can see that the DSP highlights differences in the data-flow for shared variables x, y, and z, although solely the one for x is indeed related to the bug's root cause. Note that, for this example, the alternate schedule is produced by inverting the event `x = 0` (in *T2*) with `assert(x!=0)` (in *T1*).

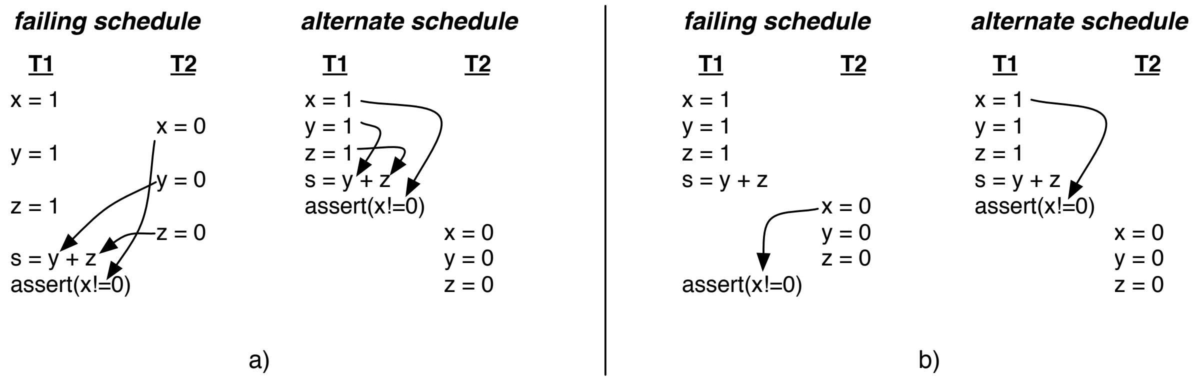


Figure 4.4: Context Switch Reduction. a) DSP *without* context switch reduction; b) DSP *with* context switch reduction. Arrows depict execution data-flows.

To mitigate the amount of irrelevant data-dependencies in DSPs, we apply a pre-processing stage to reduce the number of context switches in the failing schedule, prior to the generation of the alternate schedule. Since finding the minimal number of context switches that triggers a failure is NP-hard [Jalbert & Sen 2010], we developed an algorithm inspired in Tinertia [Jalbert & Sen 2010], which is a trace simplification heuristic that runs in polynomial time with respect to the size of the trace. Our context switch reduction (CSR for short) algorithm is described in Algorithm 6. We study the impact of CSR in DSP generation in Section 4.3.3.3.

The CSR algorithm receives a failing schedule σ and the constraint system Φ_{fail} (see Equation 4.1) as input. All the context switch reduction actions are applied to σ_{cur}, and whenever the schedule σ_{tmp}, resulting from the application of an action, satisfies the constraint system Φ_{fail}, that schedule is stored into σ_{cur}.

CSR starts by initializing σ_{cur} to the input schedule σ and then enters the main loop (lines 2-16). In lines 4-9, the algorithm does a forward pass over the schedule and applies the *moveUpSeg* action for each event e in the schedule (line 5). This action operates at the *thread segment*[7] level, therefore it has an effect only if e is the last operation of the segment (otherwise it just proceeds to the next event). When e is in the tail of the segment, *moveUpSeg* finds the next thread segment after e that is executed by the same thread and merges it with the thread segment that contains e. If this move produces a schedule that still satisfies the constraint model (line 6), then we have successfully found a schedule with one context switch less, and store it into σ_{cur} (line 7). As an example of this action, consider the event `x = 1` in the failing schedule of Figure 4.4a. If we apply *moveUpSeg* to this event, its thread segment will be augmented with

[7]We consider a *thread segment* to be a maximal sequence of consecutive events by the same thread.

Algorithm 6: Context Switch Reduction (CSR)

```
Input: failing schedule $\sigma$; constraint system $\Phi_{fail}$
Output: failing schedule $\sigma_{cur}$ with the number of context switches (potentially) reduced
$\sigma_{cur} \leftarrow \sigma$
repeat
    $\sigma_{old} \leftarrow \sigma_{cur}$
    for $i = 1$ to $|\sigma_{cur}|$ do
        $\sigma_{tmp} \leftarrow moveUpSeg(\sigma_{cur}, i)$
        if $solve(\Phi_{fail}, \sigma_{tmp})$ is satisfiable then
            $\sigma_{cur} \leftarrow \sigma_{tmp}$
        end
    end
    for $i = |\sigma_{cur}|$ to 1 do
        $\sigma_{tmp} \leftarrow moveDownSeg(\sigma_{cur}, i)$
        if $solve(\Phi_{fail}, \sigma_{tmp})$ is satisfiable then
            $\sigma_{cur} \leftarrow \sigma_{tmp}$
        end
    end
until $numCS(\sigma_{old}) \leq numCS(\sigma_{cur})$;
return $\sigma_{cur}$
```

the next thread segment by the same thread (*i.e.*, `y = 1`). The schedule resulting from this move will then be `x = 1; y = 1; x = 0;` (...).

The next step of the CSR algorithm does a backwards pass over the schedule and applies the *moveDownSeg* action for each event e in the schedule (lines 10-15). Symmetrically to *moveUpSeg*, *moveDownSeg* looks for the previous thread segment before e that is executed by the same thread and merges it with the thread segment that contains e. Hence, *moveDownSeg* only has an effect if e is the first action of the thread segment. This technique is particularly helpful to eliminate context switches in the presence of partial order invariants. For instance, consider two thread segments A and B of thread *T1*, interleaved by a segment C of thread *T2*. If B contains a `join` event and C contains an `exit` event, *moveUpSeg* will yield unsatisfiable when pulling B upwards to be merged with A, because B cannot occur before C. In contrast, *moveDownSeg* will be valid because A can be merged down with B and execute after C without breaking the partial order invariant.

At the end of each iteration of the main loop, CSR computes the number of context switches of σ_{cur} (given by $numCS(\sigma_{cur})$) and compares this value to that of σ_{old}. If σ_{cur} contains fewer context switches than σ_{old}, then the algorithm proceeds to further simplify σ_{cur}. Otherwise, CSR terminates and returns σ_{cur} as the simplified schedule.

Figure 4.4b shows the DSP obtained after applying the CSR algorithm to the failing schedule in Figure 4.4a. We can see that the single data-flow variation that is now depicted is the one that explains the failure.

It should be noted that Symbiosis could also leverage other trace simplification techniques, which do not rely on SMT solver invocations. For example, SimTrace [Huang & Zhang 2011] is a static technique that reduces the number of context switches in an execution schedule by computing a graph of dependences of the events in the schedule.

4.2 Implementation

In this section, we discuss the implementation details of Symbiosis.

4.2.1 Instrumenting Compiler and Runtime

Our Symbiosis prototype implements trace collection for both C/C++ and Java programs. C/C++ programs are instrumented via an LLVM function pass. Java programs are instrumented using Soot [Vallée-Rai, Co, Gagnon, Hendren, Lam, & Sundaresan 1999], which injects path logging calls into the program's bytecode. Like CLAP, we assign every basic block with a static identifier and, at the beginning of each block, we insert a call to a function that updates the executing thread's path. The function logs each block as the tuple *(thread Id, basic block Id)* whenever the block executes. The path logging function is implemented in a custom library that we link into the program. Although our prototype is fully functional, it has not been fully optimized yet. For instance, lightweight software approaches (*e.g.*, Ball-Larus [Ball & Larus 1994]) or a hardware accelerated approaches (*e.g.*, Vaswani et al [2005] and Intel PT [Intel Corporation 2013]) could also be used to improve the efficiency of path logging. The Symbiosis prototype is publicly available at `https://github.com/nunomachado/symbiosis`.

4.2.2 Symbolic Execution and Constraint Generation

Symbiosis's guided symbolic execution for C/C++ programs has been implemented on top of KLEE [Cadar, Dunbar, & Engler 2008]. Since KLEE does not support multithreaded executions, similarly to CLAP, we fork a new instance of KLEE's execution to handle each new thread created. We also disabled the part of KLEE that solves path conditions to produce test inputs because Symbiosis does not use them. For Java programs, we have used Java PathFinder (JPF) [Visser, Păsăreanu, & Khurshid 2004]. In this case, we have disabled the handlers for *join* and *wait* operations to allow threads to proceed their symbolic execution independently, regardless of the interleaving. Otherwise, we would have to explore different possible thread interleavings when accessing these operations, in order to find one conforming with the original execution.

Additionally, we made the following changes to both symbolic execution engines. First, we ignore states that do not conform with the threads' path profiles traced at runtime, which allows to guide the symbolic execution along the original paths alone. Second, we generate and output a per-thread symbolic trace containing read/write accesses to shared variables, synchronization operations, and path conditions observed across each execution path.

Consistent thread identification. Symbiosis must ensure that threads are consistently named across the original failing execution and the symbolic execution. Similarly to CoopREP, in Symbiosis we use a technique that relies on the observation that each thread spawns its children threads in the same order, regardless of the global order among all threads (see Section 3.1.1). Symbiosis instruments thread creation points, replacing the original PThreads/Java thread identifiers with new identifiers based on the parent-children order relationship. For instance, if a thread t_i forks its jth child thread, the child thread's identifier will be $t_{i:j}$.

Marking shared variables as symbolic. Precisely identifying accesses to shared data, in order to mark shared variables as symbolic, is a difficult program analysis problem, which is orthogonal to our work. Although it is possible to conservatively mark all variables as symbolic, varying the number of symbolic variables varies the size and complexity of the constraint system. For C/C++ programs we manually marked shared variables as symbolic. We also marked variables symbolic if their values were the result of calls to external libraries not supported by KLEE. For Java programs, we use Soot's *thread-local objects* (TLO) static escape analysis strategy [Halpert, Pickett, & Verbrugge 2007], which soundly over-approximates the set of shared variables in a program (*i.e.*, some non-shared variables might be marked shared). At instrumentation time, Symbiosis logs the code point of each shared variable access. During the symbolic execution, whenever JPF attempts to read or write a variable, it consults the log to check whether that variable is shared or not. If so, JPF treats the variable as symbolic.

Comparing the two approaches, manually identifying the shared variables is clearly more complex and tedious than employing a static analysis, as it requires a careful inspection of the code. Nevertheless, we opted for following the former approach in our prototype for C/C++ applications, because we were not familiar with a thread escape analysis, similar to that of Soot, for this kind of programs. Alternatively, one could employ static data race detectors to identify shared variables [Voung, Jhala, & Lerner 2007], although these often suffer from false positives.

Locks held at failure points. If a thread holds a lock when it fails, a reordering of operations in the critical region protected by the lock may lead to a deadlocking schedule. Other threads will wait indefinitely attempting to acquire the failing thread's held lock because the failing thread's execution trace includes no release. We skirt this problem by adding a *synthetic* lock release for each lock held by the failing thread at the failure point. The synthetic releases allow the failing thread's code to be reordered without deadlocks.

4.2.3 Schedule Generation and DSPs

We implemented failing and alternate schedule generation, as well as differential schedule projections, from scratch in around 4K lines of C++ code. After building the SMT constraint formula, Symbiosis solves it using Z3 [De Moura & Bjørner 2008]. Symbiosis then parses Z3's output to obtain the solution of the model, or the UNSAT core, when generating the root cause sub-schedule. Finally, to pretty-print its output, Symbiosis generates a graphical report (using Graphviz [8]) showing the differences between the failing and the alternate schedules.

4.3 Evaluation

Our evaluation of Symbiosis focuses on answering the following three questions:

- How efficient is Symbiosis in collecting path profiles and symbolic path traces? (Section 4.3.1)
- How efficient is Symbiosis in solving its SMT constraint formulae? (Section 4.3.2)

[8] http://www.graphviz.org

- How useful are DSPs to diagnose and fix concurrency bugs? (Section 4.3.3)

We substantiate our results with characterization data and several case studies, using buggy multithreaded C/C++ and Java applications, including both real-world and benchmark programs. We used four C/C++ test cases: *Crasher*, a toy program with an atomicity violation; *StringBuf*, a C++ implementation of a bug in the Java JDK1.4 `StringBuffer` library, developed in prior work [Flanagan & Qadeer 2003]; *BBuf*, a shared buffer implementation [Huang, Zhang, & Dolby 2013]; *Pfscan*, a real-world parallel file scanner adapted for research by Elmas et al. [Elmas, Burnim, Necula, & Sen 2013]; and *Pbzip2*, a real-world, parallel bzip2 compressor.[9] We used four Java programs: *Cache4j*, a real-world Java object cache, driven externally by concurrent update requests; and three tests from the IBM ConTest benchmarks [Farchi, Nir, & Ur 2003b]: *Airline*, *Bank*, and *TwoStage*. Columns 1-4 of Table 4.1 describe the test cases.

We evaluated the scalability of Symbiosis for *Pbzip2* and *Cache4j* by varying the size of their workload. For *Pbzip2*, we compressed input files of different sizes: 80KB (small), 2.6MB (medium), and 16MB (large). For *Cache4j*, we re-ran its test driver, for update loop iteration counts of 1 (small), 5 (medium), and 10 (large). In some cases, we inserted calls to the `sleep` function, changing event timing and increasing the failure rate. Our work is not targeting the orthogonal failure reproduction problem [Huang, Zhang, & Dolby 2013], so this change does not taint our results. We ran our C/C++ experiments on an 8-core, 3.5Ghz machine with 32GB of memory, running Ubuntu 10.04.4. For Java we used a dual-core i5, 2.8Ghz CPU with 8GB of memory, running OS X.

[9]In our experiments, we used a C version of *Pbzip2* from previous work [Kasikci, Zamfir, & Candea 2012].

Table 4.1: Benchmarks and performance. Column 2 shows lines of code; Column 3, the number of threads; Column 4, the number of shared program variables; Column 5, the number of accesses to shared variables; Column 6, the overhead of path profiling; Column 7, the size of the profile in bytes; Column 8, the symbolic execution time; Column 9, the number of SMT constraints; Column 10, the number of unknown SMT variables; Column 11, the time in seconds to solve the SMT system.

	Application	**LOC**	**#Threads**	**#Shared Variables**	**#Shared Accesses**	**Profiling Overhead**	**Log Size**	**Symbolic Time**	**#SMT Constraints**	**#SMT Variables**	**SMT Time**
C/C++	Crasher	70	6	4	266	25.4%	458B	0.02s	22295	400	1m2s
	StringBuf	151	2	5	69	16.7%	632B	0.05s	423	102	1s
	BBuf	377	5	11	143	34.4%	920B	13s	2710	239	8s
	Pfscan	830	5	9	74	6.6%	3.8K	1.87s	678	131	1s
	Pbzip2 (S)				176	2.5%	1.7K	11.16s	1361	289	1s
	Pbzip2 (M)	1942	9	14	367	1.3%	2.6K	36.17s	6771	564	26s
	Pbzip2 (L)				1156	2.5%	9.4K	7m11s	514548	2866	15h15m
Java	Airline	108	8	2	36	22%	262B	1.30s	2670	84	1s
	Bank	125	3	3	115	12.4%	788B	1.56s	8250	197	2s
	TwoStage	123	4	4	49	14.8%	196B	2.53s	264	88	1s
	Cache4j (S)				28	7.3%	366B	1.64s	122	51	1s
	Cache4j (M)	2344	4	7	1247	8.6%	17K	4.56s	303626	1810	51s
	Cache4j (L)				1411	9.3%	24K	4.76s	1142120	2051	1h25m

4.3.1 Trace Collection Efficiency

We measured the time and storage overhead of path profiling relative to native execution and the time cost of symbolic trace collection. Columns 6-10 of Table 4.1 report the results, averaged over five trials. Symbiosis imposes a tolerable path profiling overhead, ranging from 1.3% in *Pbzip2 (medium)* to 25.4% in *Crasher*. Curiously, the runtime slowdown is smaller for real-world applications (*Pfscan, Pbzip2, and Cache4j*) than for benchmarks. The reason is that the latter programs have more basic blocks with very few operations, making block instrumentation frequent. The space overhead of path profiling is also low, ranging from 196B (*TwoStage*) to 24K (*Cache4j*). Moreover, Symbiosis collects symbolic traces in just a few seconds for most test cases. The only exception is *Pbzip2 (large)*, which took KLEE around seven minutes to finish. JPF quickly produced the symbolic traces for all programs.

4.3.2 Constraint System Efficiency

The last three columns of Table 4.1 describe the SMT formulae Symbiosis built for each test case. The table also reports the amount of time Symbiosis takes to solve its SMT constraints with Z3, yielding a failing schedule. The data show that solver time is very low (*i.e.*, seconds) in most cases. Solver time often grows with constraint count, but not always. *Cache4j (large)* has more than double the constraints of *Pbzip2 (large)*, but was around 11 times faster. Figure 4.5 helps explain the discrepancy by showing the composition of the SMT formulations by constraint type. *Pbzip2* has many *locking* and *read-write* constraints, while *Cache4j* has many *read-write* constraints, but no *locking* constraints. The solution to locking constraints determines the execution's lock order, constraining the solution to read-write constraints. The formulation's complexity grows not with the count, but the interaction of these constraint types.

Symbiosis's SMT solving times are practical for debugging use. To produce a DSP, Symbiosis requires only a trace from a single, failing execution and does not require any changes to the code or input. We argue that our experiments are realistic because a programmer, when debugging, often has a bug report with a small test case that yields a short, simple execution. The data suggest that Symbiosis handles such executions very quickly (*e.g.*, *Pbzip2 (small)*, *Cache4j (medium)*). Moreover, debugging is a relatively rare development task, unlike compilation, which happens frequently. As such, giving Symbiosis minutes or hours to help solve hard bugs

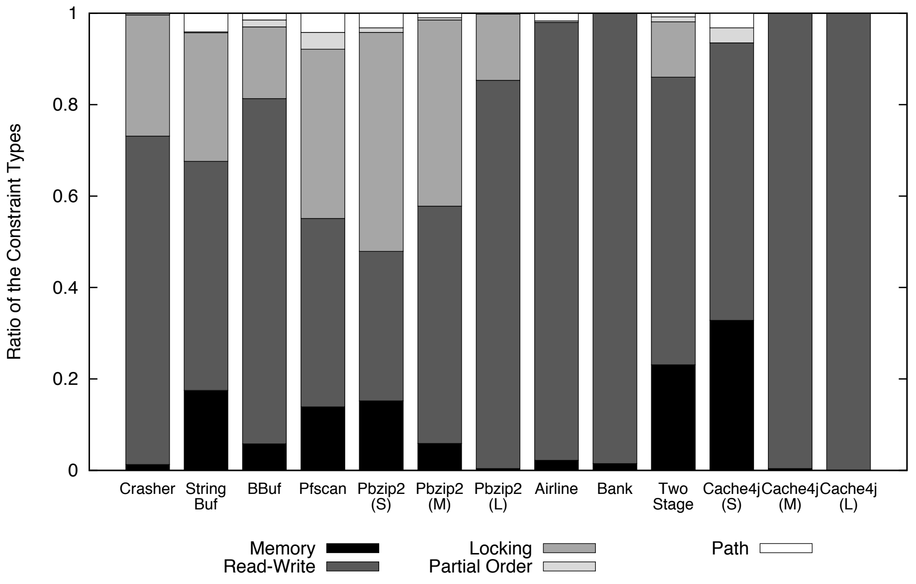


Figure 4.5: Breakdown of the SMT constraint types.

(like *Pbzip2 (large)*) is reasonable. Additionally, Symbiosis could use parallel SMT solving, like CLAP or incorporate lock ordering information, like [Bravo, Machado, Romano, & Rodrigues 2013], to decrease solver time.

4.3.3 Differential Schedule Projections Effectiveness

Symbiosis produces a graphical visualization of its differential schedule projections (DSPs) as a graph with specific identifying information on nodes and edges that reflects source code lines and variables. This information includes schedule variations, as well as data-flow variations.

In this section, we are interested in assessing the effectiveness of DSPs to diagnose concurrency failures. To this end, we first evaluate the efficacy of DSPs in isolating the root cause of the benchmark bugs. Second, we use case studies to further illustrate how DSPs can be used to understand and fix those bugs. Third, we investigate the impact of using context switch reduction when generating DSPs. Finally, we present a user study that assesses the benefits of DSPs over full failing schedules for concurrency bug diagnosis.

4.3.3.1 Root Cause Isolation Efficacy

To evaluate the efficacy of DSPs in isolating the bug's root cause, we compared the number of program events and data-flow edges in the differential schedule projection against those of

Table 4.2: Differential schedule projections. Columns 2 is the number of event pairs reordered to find a satisfiable alternate schedule (*#Alt. Pairs*). Column 3 shows the number of events in the failing schedule (*#Events Fail Sch.*) and Column 4 shows the number of events in the corresponding differential schedule projection (*#Events DSP*). Column 5 shows the number of data-flow edges in the failing schedule (*#D-F Fail Sch.*) and Column 6 shows the number of data-flow variations in the differential schedule projection (*#D-F DSP*). Columns 4 and 6 also show the percent change compared to the full schedule. Column 7 shows the number of operations involved in the data-flow variations (*#Ops to Grok*). Columns 8-9 show whether the differential schedule projection explains the failure, and whether it directly points to a fix of the underlying bug in the code.

Application	**#Alt. Pairs**	**#Events Fail Sch.**	**#Events DSP (Δ%)**	**#D-F Fail Sch.**	**#D-F DSP (Δ%)**	**Ops. to Grok**	**Expla-natory?**	**Finds Fix?**
Crasher	27	287	9 (↓**97**)	107	1 (↓**99**)	3	Y	Y
StringBuf	9	73	15 (↓**80**)	28	1 (↓**96**)	3	Y	Y
BBuf	3	157	19 (↓**88**)	79	1 (↓**99**)	3	Y	Y
Pfscan	5	93	16 (↓**83**)	32	1 (↓**97**)	3	Y	Y
Pbzip2 (S)	1	206	4 (↓**98**)	29	1 (↓**97**)	3		
Pbzip2 (M)	1	397	3 (↓**99**)	82	1 (↓**99**)	3	Y	N
Pbzip2 (L)	2	1223	168 (↓**86**)	264	2 (↓>**99**)	5		
Airline	1	58	6 (↓**90**)	25	2 (↓**92**)	6	Y	Y
Bank	181	124	31 (↓**75**)	72	2 (↓**97**)	5	Y	Y
TwoStage	14	60	3 (↓**95**)	27	1 (↓**96**)	3	Y	Y
Cache4j (S)	1	39	12 (↓**69**)	11	2 (↓**82**)	6		
Cache4j (M)	1	1257	10 (↓>**99**)	552	1 (↓>**99**)	3	Y	N
Cache4j (L)	1	1422	5 (↓>**99**)	628	1 (↓>**99**)	3		

full, failing executions computed by Symbiosis.

Table 4.2 summarizes our results. The most important result is that Symbiosis's differential schedule projections are simpler and clearer than looking at full, failing schedules. Symbiosis reports a small fraction of the full schedule's data-flows and program events in its output – on average, 90% fewer events and 96% fewer data-flows. By highlighting only the operations involved in the data-flow variations, Symbiosis focuses the programmer on just a few events (3 to 6 in our tests). Furthermore, all events Symbiosis reports are part of data-flow or event orders that dictate the presence or absence of the failure. DSPs depict those events only, simplifying debugging.

Symbiosis finds an alternate, non-failing schedule after reordering few event pairs – just 1 in many cases (*e.g.*, *Cache4j*, *Pbzip2*). Symbiosis reorders one pair at a time, starting from those closer in the schedule to failure, and the data show that this usually works well. *Bank* is an outlier – Symbiosis reordered 181 different pairs before finding an alternate non-failing schedule. The bug in this case is an atomicity violation that breaks a program invariant that is not checked until later in the execution. As a result, Symbiosis must search many pairs, starting from the failure point, to eventually reorder the operations that cause the atomicity violation.

Note that, even if a failure occurs only in the presence of a particular chain of event orderings, it suffices to reorder any pair in the chain to prevent that failure. As mentioned in Section 4.1.5.2, this phenomenon is called the *Avoidance-Testing Duality*, and is detailed in previous work [Lucia & Ceze 2013].

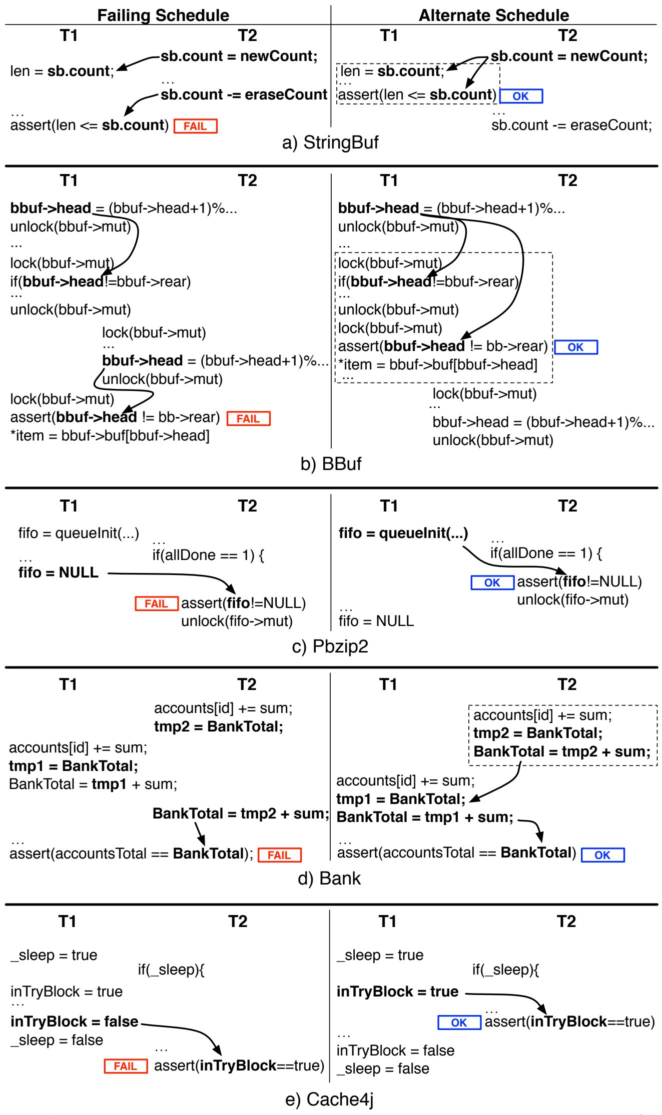
4.3.3.2 Differential Schedule Projections Case Studies

This section uses case studies to illustrate how differential schedule projections focus on relevant operations and help understand each failure.

StringBuf is an atomicity violation first studied in [Flanagan & Qadeer 2003] and its DSP is depicted in Figure 4.6a. *T1* reads the length of the string buffer, *sb*, while *T2* modifies it. When *T2* erases characters, the value *T1* read becomes stale and *T1*'s assertion fails . The DSP shows that the cause of the failure is *T2*'s second write, interleaving *T1*'s accesses to *sb.count*. Moreover, Symbiosis's alternate schedule suggests that, for *T1*, the write on value *len* and the verification of the assertion should execute atomically in order to avoid the failure. For this case, this is actually a valid bug fix.

BBuf contains producer/consumer threads that put/get items into/from a shared buffer for a given number of times. This program has an atomicity violation that allows consumer threads to get items from the buffer, even when it is empty. Figure 4.6b illustrates this failure: after getting an item from the buffer, consumer thread *T1* prepares to get another one, but first checks whether the buffer is empty (*i.e.*, `if(bbuf->head!=bbuf->rear)`). As the condition is true, *T1* proceeds to consume the item, but it is interleaved by *T2* in the meantime, which consumes the item first and updates the value of `bbuf->head`. This causes *T1* to later violate the assertion that enforces the buffer invariant. The alternate schedule in Figure 4.6b shows that executing atomically the two blocks that, respectively, check the conditional clause and the assertion prevents the failure.

Pbzip2 is an order violation studied in [Jalbert & Sen 2010]. Figure 4.6c shows Symbiosis's DSP that illustrates the failure's cause. *T1*, the producer thread, communicates with *T2* the consumer thread via the shared queue, *fifo*. If *T1* sets the *fifo* pointer to null while the consumer



```
a) StringBuf
Failing Schedule                         | Alternate Schedule
T1                   T2                  | T1                          T2
                     sb.count = newCount;|                             sb.count = newCount;
len = sb.count;                          | len = sb.count;
                     ...                 | ...
                     sb.count -= eraseCount | assert(len <= sb.count) OK
...                                      |                             ...
assert(len <= sb.count) FAIL             |                             sb.count -= eraseCount;

b) BBuf
T1                                       | T1
bbuf->head = (bbuf->head+1)%...          | bbuf->head = (bbuf->head+1)%...
unlock(bbuf->mut)                        | unlock(bbuf->mut)
...                                      | ...
lock(bbuf->mut)                          | lock(bbuf->mut)
if(bbuf->head!=bbuf->rear)               | if(bbuf->head!=bbuf->rear)
...                                      | ...
unlock(bbuf->mut)                        | unlock(bbuf->mut)
          T2                             | lock(bbuf->mut)
          lock(bbuf->mut)                | assert(bbuf->head != bb->rear) OK
          ...                            | *item = bbuf->buf[bbuf->head]
          bbuf->head = (bbuf->head+1)%...| ...
          unlock(bbuf->mut)              |          T2
lock(bbuf->mut)                          |          lock(bbuf->mut)
assert(bbuf->head != bb->rear) FAIL      |          ...
*item = bbuf->buf[bbuf->head]            |          bbuf->head = (bbuf->head+1)%...
                                         |          unlock(bbuf->mut)

c) Pbzip2
T1                   T2                  | T1                     T2
fifo = queueInit(...)                    | fifo = queueInit(...)
                     ...                 |                        ...
...                  if(allDone == 1) {  |                        if(allDone == 1) {
fifo = NULL                              |                     OK assert(fifo!=NULL)
                FAIL assert(fifo!=NULL)  |                        unlock(fifo->mut)
                     unlock(fifo->mut)   | ...
                                         | fifo = NULL

d) Bank
T1                   T2                  | T1                     T2
                     accounts[id] += sum;|                        accounts[id] += sum;
                     tmp2 = BankTotal;   |                        tmp2 = BankTotal;
accounts[id] += sum;                     |                        BankTotal = tmp2 + sum;
tmp1 = BankTotal;                        | accounts[id] += sum;
BankTotal = tmp1 + sum;                  | tmp1 = BankTotal;
                     BankTotal = tmp2 + sum; | BankTotal = tmp1 + sum;
...                                      | ...
assert(accountsTotal == BankTotal); FAIL | assert(accountsTotal == BankTotal) OK

e) Cache4j
T1                   T2                  | T1                     T2
_sleep = true                            | _sleep = true
                     if(_sleep){         |                        if(_sleep){
inTryBlock = true                        | inTryBlock = true
...                                      |                        ...
inTryBlock = false                       |                     OK assert(inTryBlock==true)
_sleep = false                           | ...
                     ...                 | inTryBlock = false
FAIL assert(inTryBlock==true)            | _sleep = false
```

Figure 4.6: Summary of Symbiosis's output for some of the test cases. Arrows depict data-flows and dashed boxes depict regions that Symbiosis suggests to be executed atomically.

thread is still using it, *T2*'s assertion fails. The alternate schedule in Figure 4.6b, explains the failure because reordering the assignment of *null* to *fifo* after the assertion prevents the failure. The DSP is, thus, useful for understanding the failure. However, to fix the code, the programmer must order the assertion with the null assignment using a `join` statement. The DSP does not provide this suggestion, so, despite helping explain the *failure*, it does not completely reveal how to fix the bug.

Bank is a benchmark in which multiple threads update a shared bank balance. It has an atomicity violation that leads to a lost update. Figure 4.6d shows the DSP for the failure: *T1* and *T2* read the same initial value of *BankTotal* and subsequently write the same updated value, rather than either seeing the result of the other's update. The final assertion fails, because *accountsTotal*, the sum of per-account balances, is not equal to *BankTotal*. The Figure shows that Symbiosis's DSP correctly explains the failure and shows that eliminating the interleaving of updates to *BankTotal* prevents the failure. It is noteworthy that in this example the atomicity violation is not *fail-stop* and happens in the middle of the trace. Scanning the trace to uncover the root cause would be difficult, but the DSP pinpoints the failure's cause precisely.

Cache4j has a data race that leads to an uncaught exception when one thread is in a `try` block and another interrupts it with the library `interrupt()` function [Sen 2008]. JPF doesn't support exception replay, so we slightly modified its code, preserving the original behavior, by replacing the exception with an assertion about a shared variable. Figure 4.6e shows that in our version of the code, *inTryBlock* indicates whether a thread is inside a `try-catch` block or not, the assertion `inTryBlock == true` replaces the `interrupt()` call. The program fails when *T1* is interrupted outside a `try` block as in the original code. The schedule variations reported in the DSP explain the cause of failure – if the entry to the `try` block (*i.e.*, `inTryBlock = true`) precedes the assertion, execution succeeds; if not, the assertion fails. The involvement of exceptions makes the fix for this bug somewhat more complicated than simply adding atomicity, but the understanding that the DSP provides points to the right part of the code and illustrates the correct behavior.

4.3.3.3 Impact of Context Switch Reduction

We evaluate the impact of the Context Switch Reduction (CSR) algorithm presented in Section 4.1.7. To this end, we ran Symbiosis with and without CSR and compared: *i)* the number of context switches of the failing schedule generated, *ii)* the number of event pairs reordered to find a satisfiable alternate schedule, and *iii)* the number events and data-flows in the DSPs produced.

Table 4.3 reports the results of our experiments. The most prominent observation is that our CSR algorithm is indeed effective in reducing the number of context switches (the failing schedules have 63% less context switches, on average). Table 4.3 also shows that, when using CSR, Symbiosis is able to find a satisfiable alternate schedule with less event pair reorderings in three of the test cases (*Crasher*, *Pfscan*, and *Bank*).

On the other hand, DSPs produced by Symbiosis with CSR tend to have slightly more

Table 4.3: Context Switch Reduction Efficacy. Column *#CS* indicates the number of schedule context switches; *#Alt Pairs* is the number of event pairs reordered to find a satisfiable alternate schedule; *#Events DSP* and *#D-F DSP* show, respectively, the number of events and number of data-flow in the corresponding differential schedule projection. *Time CSR* indicates the total amount of time required to perform the context switch reduction algorithm. Shaded cells indicate the cases where Symbiosis achieves better results with CSR than without.

Application	**Without CSR**				**With CSR**				
	#CS	**#Alt. Pairs**	**#Events DSP**	**#D-F DSP**	**#CS**	**#Alt. Pairs**	**#Events DSP**	**#D-F DSP**	**Time CSR *(Solver Calls)***
Crasher	104	27	9	1	34	21	9	1	19m33s *(166)*
StringBuf	7	9	15	1	4	9	15	1	2s *(14)*
BBuf	35	3	19	1	10	6	19	1	14s *(40)*
Pfscan	23	5	16	1	9	3	16	1	9s *(32)*
Pbzip2 (S)	74	1	4	1	14	1	7	1	14s *(59)*
Pbzip2 (M)	151	1	4	1	22	1	7	1	4m58s *(163)*
Pbzip2 (L)	292	1	4	1	36	1	7	1	5h11m *(353)*
Airline	28	1	6	2	8	1	8	2	5s *(33)*
Bank	6	181	31	2	4	154	46	2	6s *(9)*
TwoStage	16	14	3	1	4	14	6	1	2s *(15)*
Cache4j (S)	4	1	12	2	4	1	12	2	<1s *(2)*
Cache4j (M)	21	1	10	1	7	1	10	1	6m50s *(25)*
Cache4j (L)	29	1	5	1	7	1	5	1	16m19s *(29)*

events than those produced without CSR. The reason is because CSR produces schedules with more coarse-grained thread segments (*i.e.*, comprising more events) and our current DSP implementation does not eliminate events from the same thread segment that occur in-between two events involved in data-flow variations.

Another observation worth noting from Table 4.3 is that DSPs with CSR do not exhibit any reductions in terms of data-flow variations with respect to DSPs without CSR. The reason is because the feasible alternate schedules produced by Symbiosis are mainly the result of reordering a event from a thread (typically the one corresponding to the failure condition) with an event from another thread that is close in the schedule. Hence, most data-flows do not change after reordering the event pair, even if the failing schedule has unnecessary context switches.

Regarding the amount of time required to perform CSR (shown in the last column of Table 4.3), it is possible to see that Symbiosis took only a couple of seconds for most cases. However, for *Pbzip2 (L)* this time exceed 5 hours. The reason is because the failing schedule for this program contained a significant number of context switches, which required the CSR to invoke the solver several times (353 to be precise). Moreover, since the constraint model for *Pbzip2 (L)* is also the one with most constraints, each solver call becomes particularly costly.

In conclusion, despite CSR being effective in reducing the number of context switches in a failing schedule, our experiments show that this does not imply a reduction in the number of events and data-flow variations reported in the DSPs. Furthermore, since performing CSR can be costly for some programs, we argue that computing the DSP with the original failing schedule should probably be the most cost-effective approach for the majority of the cases.

4.3.3.4 User Study

To assess how useful for debugging a DSP is, compared to full failing schedule, we conducted a user study.

Participants. We recruited 48 participants, including 21 students (6 undergraduate, 9 masters, 6 doctoral) from Instituto Superior Técnico, 24 masters students from University of Pennsylvania, 1 doctoral student from Carnegie Mellon University, and 2 software engineers (with three years of experience in industry), to individually find the root cause of a given concurrency bug.

Study Design. The participants were randomly divided into two groups according to the type of debugging aid they were going to use in the experiment: the full schedule of a failing execution or the DSP for the same failing schedule.

Fifteen participants took the study in a proctored session. The remaining 33 participants received the study by email and returned it by email on completion.

Pior to initiating the experiment, each participant had access to a tutorial example that explained how the respective debugging aid could be used to find the root cause of a concurrency bug in a toy program. The goal of the tutorial was to guarantee that each participant knew how to read and understand its debugging aid (*i.e.*, the full failing schedule or the DSP) beforehand.

For the experiment, we provided each participant with the source code of a multithreaded program with a concurrency bug. This bug was a simplified version of the *StringBuf* error used in the previous sections and consisted of an atomicity violation that caused the failure of an assertion (the assertion included two conditions of which only one fails). In addition, each participant was given its corresponding debugging aid: the full schedule of a failing execution or the DSP for the same failing schedule.

The experiment consisted in analyzing the debugging aid and the program's source code, and answer a short survey with five questions. First, we asked which conjunct of the assertion condition was violated (allowing us to screen for wrong answers). Second, we asked the participant to write up to three sentences describing what caused the assertion to fail. Then, we asked the participant to rate, on a scale of 1 to 5, the difficulty of diagnosing the concurrency bug, as well as their experience in debugging multithreaded programs. Finally, we asked the participant to report the time they took to find the bug by choosing one of four intervals of time: *0-4min*, *5-9min*, *10-14min*, and *≥15min*.

Results. We analyze these results according to three different criteria, which are discussed below.

- *Correctness.* Do the participants correctly identified the root cause of the assertion violation? Does the type of debugging aid have an impact in the success rate?
- *Bug Difficulty.* Does the type of debugging aid have an impact in the self-reported bug difficulty?
- *Diagnosis Time.* Does the type of debugging aid have an impact in the time required to diagnose the bug? Are there other factors that significantly influence the diagnosis time (*e.g.*, the participant's debugging experience)?

To support the conclusions of the user study, we performed a statistical analysis over the obtained data. Concretely, for each criterion, we started by computing the correlation coefficient between the variables being analyzed (*e.g.*, identifying the correct root cause of the bug and using DSP as debugging aid). For the cases where the correlation coefficient indicated a statistically significant relationship between the variables, we also performed a *t-test* over the samples to further support that claim.

Correctness. We considered that participants had a correct answer when they correctly identified the failing conjunct in the assertion condition and provided a satisfactory explanation to the bug's root cause.

From the 48 participants, 38 answered correctly. In particular, the participants that received the DSP showed a slightly higher percentage of correct answers in comparison to those that received the full failing schedule (74% against 71%, respectively).

To statistically evaluate the relationship between the type of debugging aid and the cor-

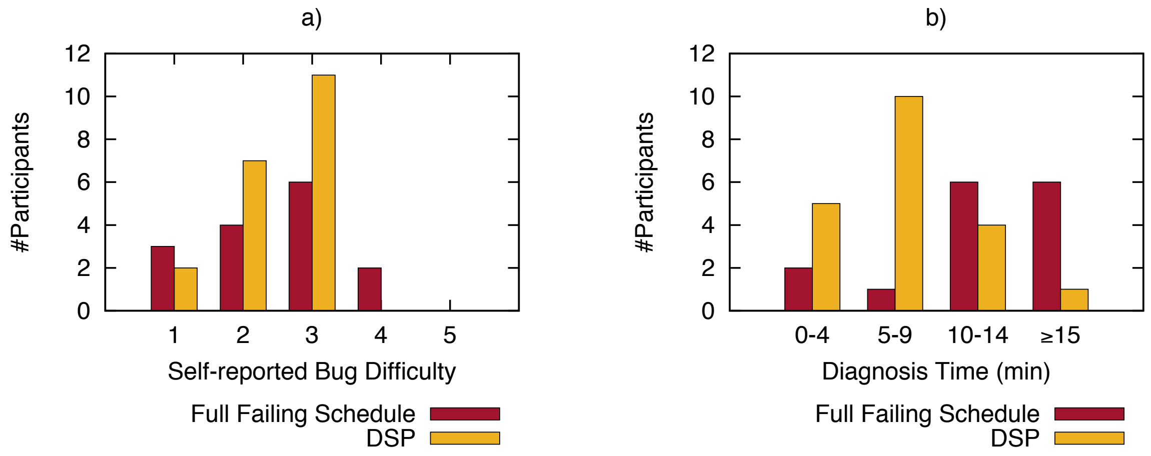


Figure 4.7: User Study Results (considering solely correct answers). a) Impact of the type of debugging aid in self-reported bug difficulty (1 means very easy, 5 means very hard); b) Impact of the type of debugging aid (full schedule/DSP) in diagnosis time.

rectness of the answer, we calculated the *point-biserial correlation coefficient*[10] between these two variables. We considered the variable debugging aid to have value 0 for the DSP and 1 for the full failing schedule. For the correctness of the answer, we considered 0 to indicate that the response is incorrect and 1 to indicate that it is correct.

The correlation value between the debugging aid variable and the correctness of the answer is -0.030, which means that there is no statistically significant correlation between the two variables. The negative value of the coefficient shows that there is a slight trend for people using the DSP to answer correctly more often than people with full failing schedules though.

Given that the majority of the participants successfully found the root cause of the bug, we believe that, for (somewhat) simple concurrency bugs, having any kind of debugging aid is indeed helpful for debugging.

Bug Difficulty. We are interested in understanding whether participants doing the experiment with DSPs would find the bug easier to debug or not. Figure 4.7a depicts the values for bug difficulty reported by the participants according to their type of debugging aid. From the figure, it is not possible to extract a clear trend, although we can see that no participant using DSPs rated the bug above 3, whereas two participants with the full schedule classified the bug

[10]We opted for using the point-biserial correlation coefficient to compute the correlation because the debugging aid variable is *naturally dichotomous*, *i.e.*, it corresponds to either using the DSP or the full failing schedule. The variable representing the correctness of the answer is also naturally dichotomous, as the answer can only be considered *correct* or *incorrect*. Note that the correlation coefficient varies between -1 and 1, where -1 (1) indicates a very strong negative (positive) correlation between the variables, and 0 indicates that there is no correlation at all between the variables.

as 4.

We computed the correlation coefficient for these two variables and obtained the value 0.010, which indicates there is indeed no statistically significant correlation between the type of debugging aid and the bug difficulty. However, the value of the coefficient points out a slim positive relationship between using the full schedule and finding the bug harder to diagnose.

Similarly, the correlation coefficient for the participants' experience and the bug difficulty (which has value -0.048) shows that there is no statistically significant correlation between these two variables, although the tendency for this case is that participants with more experience tend to find the bug easier than less experienced participants.

Diagnosis Time. Figure 4.7b) reports the participants' diagnosis time according to the type of debugging aid (full schedule or DSP) they received. The figure shows that participants that received the DSP tended to diagnose the bug in less time. To statistically evaluate this claim, we calculated the point-biserial correlation coefficient between type of debugging aid and the amount of time to diagnose the bug. Once more, we considered the variable debugging aid to have value 0 for the DSP and 1 for the full failing schedule.

The correlation value between the debugging aid variable and diagnosis time is 0.416, which means that, with a 99% confidence level, there is a significant positive relationship between these two variables. This indicates that using the full failing schedule is correlated to a greater amount of time to find the bug. In other words, the correlation supports the claim that using a DSP allows for faster bug diagnosis, as we expected.

We also performed a *Welch's t-test* to check whether the means of our two samples (the DSP sample vs. the full schedule sample) are statistically significantly different. A significant result for this test indicates that the sampled means correspond to different underlying populations, and, as our data suggests, that the DSP sample mean is smaller than the failing schedule sample mean. Let M_0 be the mean of the diagnosis times for the DSP tests, and M_1 the mean of the diagnosis times for the full failing schedule tests. We consider the null hypothesis to be "$M_0 = M_1$" and we test whether to reject the hypothesis.

The t-test's two-tailed reference value of t_p with 24 degrees of freedom[11] and a 95% confi-

[11]The number of degrees of freedom was calculated using the *Welch-Satterthwaite equation* for the Welch's t-test.

dence level ($p < 0.05$) is 2.064. Evaluating the Welsh's t-test for our data, $t = -2.281$. Since $2.064 < |-2.281|$ and, consequently, $t_p < t$, we can reject the null hypothesis and conclude that $M_0 \neq M_1$. This result shows that there is a statistically significant decrease in debugging time using a DSP, compared to using a failing schedule.

Finally, we computed the Pearson's correlation coefficient between participants' self-reported experience in debugging multithreaded programs and the diagnosis time, in order to assess whether the experience was also a significant factor in the diagnosis time or not. The correlation coefficient for this case showed that there was only an extremely weak, negative relationship between debugging experience and diagnosis time. This means that participants that reported higher experience tend to find the bug slightly faster than participants with less experience, although the difference is not statistically significant.

In summary, regarding the benefits of DSPs over full failing schedules, the main findings of our user study show that:

- *Correctness.* There is a slim correlation between using DSPs and a correct bug diagnosis, although it is not statistically relevant.
- *Bug Difficulty.* There is a slight correlation between using DSPs and finding the bug easier to debug, although it is not statistically relevant.
- *Diagnosis Time.* There is a *statistically significant* correlation between using DSPs and a faster diagnosis time.

Thus, in our study, all users had a similar perception of the difficulty of the bug at hand (which is not surprising, given that the intrinsic "hardness" of a bug is somehow independent of the tools used to find it). Furthermore, in both groups, approximately the same percentage of users were able to found the correct answer (which suggests that both groups had a similar ability/experience to recognize the right answer). Still, the group using DSPs was able to perform the diagnosis faster, which supports our claim that Symbiosis can reduce the debugging time.

Threats to validity. To simplify the study, we designed the whole experiment to be supported solely by textual material. As such, we crafted the experiment's concurrency bug in such a way that it could be solved in a practical time of 20 minutes, by a person not familiar with the program's source code. This fact might have diminished the debugging time difference between using a DSP and a full schedule because all participants, regardless of debugging aid, may have taken substantial time to understand the code. We expect that the observed difference in debugging time would be greater if the participant was already familiar with the code, eliminating the fixed time cost for understanding the code.

The diagnosis time for the participants in the offsite (email) setting was self-reported, which might be subject to some inaccuracies. Despite that, we observed that the results from the offsite setting are consistent with those onsite (the proctored session).

Since the self-reported debugging experience was not measured using exact metrics, it might be biased towards each participant's self-perception of what it means to be an expert in debugging multithreaded programs. In fact, the values obtained do not always match with education level of the participants (for instance, there was a doctoral student who reported an experience level of 1 and an undergraduate who reported an experience level of 3). To mitigate this threat, we computed the correlation coefficients for the diagnosis time and the bug difficulty using the education level instead of the self-reported experience. We observed similar results and, thus, concluded that self-reported experience was a valid factor to take into account.

Summary

This chapter described Symbiosis, a novel system that gets to the bottom of concurrency bugs. Symbiosis reports *focused root cause sub-schedules*, eliminating the need for a programmer to search through an entire execution for the bug's root cause. Symbiosis also reports novel *alternate, non-failing schedules*, which help illustrate *why* the root cause is the root cause and how to avoid failures. Our novel *differential schedule projection* (DSP) approach links the root cause and alternate schedules to data-flow information, giving the programmer deeper insight into the bug's cause than path information alone. An essential part of Symbiosis's mechanism is the use of an SMT solver and, in particular, its ability to report the part of a formula that makes it unsatisfiable. Symbiosis carefully constructs a deliberately unsatisfiable formula so that the conflicting part of that formula is the bug's root cause.

We built two Symbiosis prototypes, one for C/C++ and one for Java. We used them to show that for a variety of real-world and benchmark programs from the debugging literature that Symbiosis isolates bugs' root causes and providing differential schedule projections that show how to fix those root causes.

Symbiosis, however, needs to observe a failing execution and requires the alternate non-failing schedule to adhere to the original control-flow path in order to be able to generate the DSP. In the next chapter, we address these issues by presenting a technique denoted *production-guided schedule search*.

5 Cortex: Exposing Concurrency Bugs via Production-Guided Search

As mentioned in Section 2.4, the exponential nature of the space of possible schedules of a multithreaded program renders current explorative testing techniques impractical to uncover all failing schedules in a program. As a consequence, software is commonly shipped with latent concurrency bugs that may only manifest in deployment.

When a failure occurs, systems such as CoopREP and Symbiosis can be used to help debug its underlying bug. However, concurrency errors often stem from a very specific execution path and interleaving, which may happen rarely.

In this chapter, we aim at exposing and isolating concurrency bugs without ever needing to observe a failing execution. We leverage the observation that a failing schedule typically deviates in only a few critical ways from a non-failing schedule (as also shown in Chapter 4). Our main insight is to expose *new* failing schedules by perturbing the order of events and certain branch outcomes in a non-failing schedule. We further leverage abundant *production runs* of a program on deployed systems to guide our search of the enormous space of possible execution schedules [Candea 2011]: our *production-guided search* for a failing schedule targets schedules very similar to a non-failing schedule observed in production.

We present Cortex[1], a system that helps exposing and understanding concurrency bugs using traces from normal, non-failing production executions. Cortex starts by cooperatively collecting a set of per-thread path profiles from one or more production runs. Similarly to Symbiosis, each profile is used to guide a symbolic execution of the program producing a symbolic event trace for each thread that is compatible with the original execution's control-flow. Cortex combines an execution's per-thread symbolic traces to implement *production-guided* search for a new, failing execution – one that may depend on *both* schedule and path conditions.

[1]We have named our system Cortex after the cerebral cortex, which is a part of the human brain that receives and processes information from neurons to control several functions of the human body. Likewise, our system leverages information from multiple production runs to expose concurrency bugs.

Cortex's production-guided search is a novel approach to selecting a path and schedule. Starting from the computed symbolic execution, Cortex systematically reorders events in the schedule and inverts the outcome of certain branches, with a preference for executions that are most similar to the original. Cortex determines if a perturbed execution is feasible using a constraint system for a *Satisfiability Modulo Theories* (SMT) solver. The constraint system encodes synchronization, data-flow, event ordering, and the occurrence of a failure. If the SMT formulation is satisfiable, the execution that Cortex generated is feasible and the system reports the new failure. If the SMT formulation is infeasible, Cortex moves on to a different perturbation of the execution's schedule and branching behavior. Cortex favors executions that vary only slightly from the original, observed execution, putting its focus on failures that very nearly manifested in a previous execution.

Like in Chapter 4, we consider failures to be violations of assertions in the code. We argue that it is common for developers to ship code with assertions. For instance, Google pervasively uses tracing and assertions throughout live, production datacenter code via Dapper [Sigelman, Barroso, Burrows, Stephenson, Plakal, Beaver, Jaspan, & Shanbhag 2010]. Also, recent work showed that invariants can often be derived automatically [Kusano, Chattopadhyay, & Wang 2015], which broadens the applicability of Cortex.

In addition to exposing failing schedules, Cortex is also able to isolate the failure's root cause, even if it results from *schedule-dependent branches*. Schedule-dependent branches are branches whose decision may vary depending on the actual scheduling of concurrent threads [Huang & Rauchwerger 2015]. Failures resulting from schedule-dependent branches further exacerbate the challenge of diagnosing the bug, because the programmer must reason about different events in a failing execution and in a non-failing one.

Cortex leverages production-guided search to extend our work on *differential schedule projections* (see Chapter 4) and compute *differential path-schedule projections* (DPSPs). DPSPs zero in on the root cause of failures that stem from schedule-dependent branches by reporting the differences between a failing and a non-failing schedule, including variations in their event orderings, data-flow behavior, and control-flow decisions.

Our evaluation in Section 5.5 shows that Cortex is able to find failing schedules in concurrent programs by perturbing very few branch conditions. Moreover, we show that Cortex's production-guided search reduces the number of attempts to expose concurrency bugs by up to three orders of magnitude with respect to previous state-of-the-art concurrency testing techniques [Huang 2015; Coons, Musuvathi, & McKinley 2013].

The rest of the chapter is organized as follows. Section 5.1 introduces the problem of path- and schedule-dependent concurrency bugs. Section 5.2 describes the Cortex system, namely its architecture and the production-guided search technique used to find failing schedules. Section 5.3 provides a concrete example that illustrates how Cortex employs production-guided search to expose a concurrency bug that depends on schedule-sensitive branches. Section 5.4 discusses the implementation details. Section 5.5 presents the experimental evaluation results. Finally, we conclude the chapter by summarizing its main findings and contributions.

5.1 Path- and Schedule-Dependent Bugs

The variation in the schedule of a multithreaded program's execution may cause a variation in the execution's data flow, and subsequently, its control-flow. Such *schedule-sensitive branches* are often unintended by the developer and may even result in failures [Huang & Rauchwerger 2015].

We denote concurrency bugs that stem from schedule-sensitive branches *path- and schedule-dependent*. As an example of a path- and schedule-dependent bug, consider the toy multi-threaded program in Figure 5.1. In the example, *T1* and *T2* access four shared variables (x, y, w, and z). The program fails when it executes the schedule `8-1-2-3-4-9-10-5-6-7`, which causes the value of x at line 7 to be 0 and violate the assertion. All non-failing schedules for this program exhibit a different *control-flow path* than the failing schedule, because the code must not execute line 6 to guarantee that `x > 0` at line 7. Note that a failing execution requires a variation from the correct execution in both the order of events and the control-flow path executed.

As mentioned in Section 4.1.5, Symbiosis does not handle path- and schedule-dependent bugs, as its alternate schedules are required to adhere to the original control-flow of the original execution. Moreover, Symbiosis needs a trace from a failing execution to work, which might be hard to obtain in some cases. For instance, we ran the program in Figure 5.1 10,000 times and it did not fail a single time.

In the remaining of this chapter, we show how Cortex couples production-guided search with symbolic execution and SMT solving ideas from Symbiosis to expose and isolate path- and

```
         (initially x = y = w = z = 0)
        T1                      T2
 1: if(z > 0)               8: z = 1
 2:    w++                  9: if(w > 0)
 3: x = 1                  10:    y = 0
 4: y = 1
 5: if(y == 0)
 6:    x--
 7: assert(x > 0)
```

Figure 5.1: Example of a multithreaded program with a path- and schedule-dependent bug.

schedule-dependent bugs, namely the one in Figure 5.1, without the need to observe a failing execution.

5.2 Cortex

Cortex is an automated system for exposing and debugging path- and schedule-dependent failures in multithreaded programs. In contrast with other systems, Cortex does not need to observe a failed execution to isolate a failure. Instead, Cortex starts from a concrete, non-failing execution and explores alternative executions with only minor variations in their schedule and path from the non-failing schedule. Using an initial, non-failing execution from production turns Cortex's execution space exploration into a *production-guided search* for new failures. The exposed failures represent behavior that nearly happened in the observed execution, and is, thus, more likely to happen in some future execution. Like Symbiosis, Cortex summarizes only the differences between the exposed failing execution and the original non-failing execution to clearly isolate the root cause of the failure to the developer.

Cortex operates in four main steps: *static analysis*, *trace collection*, *production-guided search*, and *root cause isolation*. These steps are illustrated in Figure 5.2 and described in the following sections.

5.2.1 Static Analysis and Symbolic Trace Collection

Cortex follows the same approach as Symbiosis to capture thread symbolic traces. Concretely, Cortex starts by performing static program analysis and generating per-thread symbolic

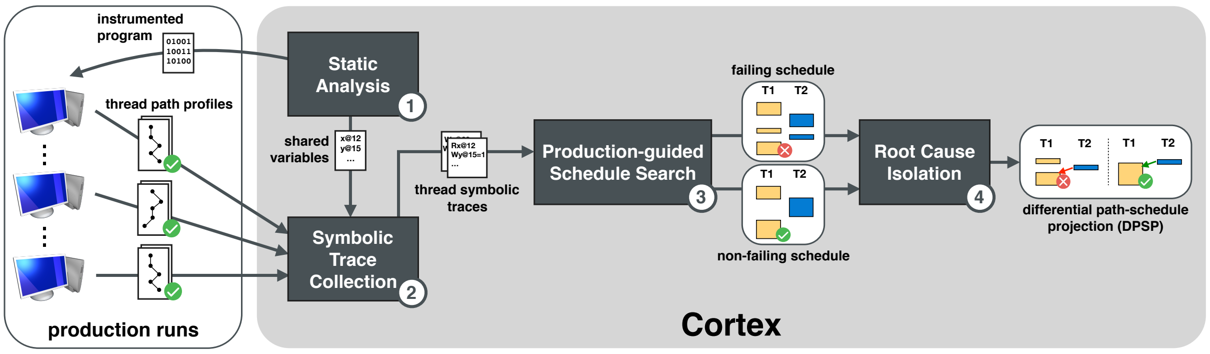


Figure 5.2: Overview of Cortex. 1) Cortex starts by instrumenting the program in order to allow capturing per-thread path profiles at runtime. 2) Cortex uses the path profiles, collected from multiple production runs, to guide a symbolic execution of the program and produce per-thread symbolic event traces that adhere to the original executions' control-flow. 3) Cortex leverages symbolic traces and SMT constraint solving to conduct a production-guided search for a new failing execution that may depend on both schedule and path conditions. 4) Whenever Cortex successfully uncovers a new failing schedule, it computes a differential path-schedule projection (using the uncovered failing schedule and a non-failing schedule) to isolate the underlying bug.

traces. During static analysis, Cortex instruments the program to capture the threads' execution path at runtime and identifies shared variables to later be marked as symbolic.

Given the infeasibility of exhaustively exploring all possible executions of a program, Cortex narrows the exploration to favor those paths that are most alike those observed in production. To this end, Cortex collects per-thread path profiles from the large number of production runs executed by the user instances of the program.

Production runs with the same control-flow output identical traces. However, paths with different control-flow may execute in production. Cortex can collect a variety of paths from multiple, different executions, potentially from distinct machines and execution environments. This form of cooperative path collection enables Cortex to leverage the execution diversity in multiple deployments to gather a representative collection of traces.

Cortex then uses symbolic execution to generate per-thread, symbolic traces from the collected, per-thread, concrete traces. As in Symbiosis, Cortex guides each thread's symbolic execution by its path trace. In other words, the symbolic execution proceeds solely across the execution path indicated by its corresponding path profile. Hence, instead of exploring all control-flow paths, as in a conventional symbolic execution, the symbolic execution in Cortex only follows branches taken in the original run. Cortex thus generates per-thread symbolic traces containing the same control-flow, synchronization, and shared memory accesses as the original, production run.

Cortex maintains a database of per-thread symbolic traces generated from any set of observed, per-thread concrete traces. Cortex organizes the traces in its database to facilitate its downstream search, using the executions' control-flow. Concretely, Cortex denotes a branch outcome as a binary value, with a 1 for branches taken and a 0 for branches not taken. An execution path is, thus, uniquely defined by a string of bits.

For example, consider a symbolic trace for thread *T1* in Figure 5.2 with path id 10. This path id indicates that the thread followed an execution path corresponding to the conditions [z>0] and [¬(y==0)]. Note that the 0 in the path id means that the corresponding path condition evaluates false during the execution, hence the negation symbol in the condition [¬(y==0)].

5.2.2 Production-Guided Search

After gathering the path profiles from production runs and generating their corresponding per-thread symbolic traces, Cortex starts its search for alternate, failing executions. Figure 5.3 illustrates Cortex's search procedure.

Cortex's search is guided by production in two ways. The first way is *schedule exploration*, during which Cortex searches multithreaded executions compatible with the set of per-thread symbolic traces from some observed execution. If schedule exploration fails to surface any new failures, for any traces in Cortex's database, Cortex uses *execution synthesis* as a second form of production-guided search. Cortex synthesizes new multithreaded executions by *modifying* the control-flow behavior in one or more of the threads' traces. Cortex then performs schedule exploration on the new synthesized execution.

5.2.2.1 Schedule Exploration

Given a combination of per-thread, symbolic path traces — from either a production run or a synthesized execution — Cortex checks if any interleaving of operations that make up those paths leads to a failure. First, Cortex selects an assertion from the trace at random. Next, Cortex builds a Φ_{fail} constraint model (from Section 4.1.3) with the symbolic information contained in the traces and the condition in the assertion (which corresponds to the bug constraint ϕ_{bug}).

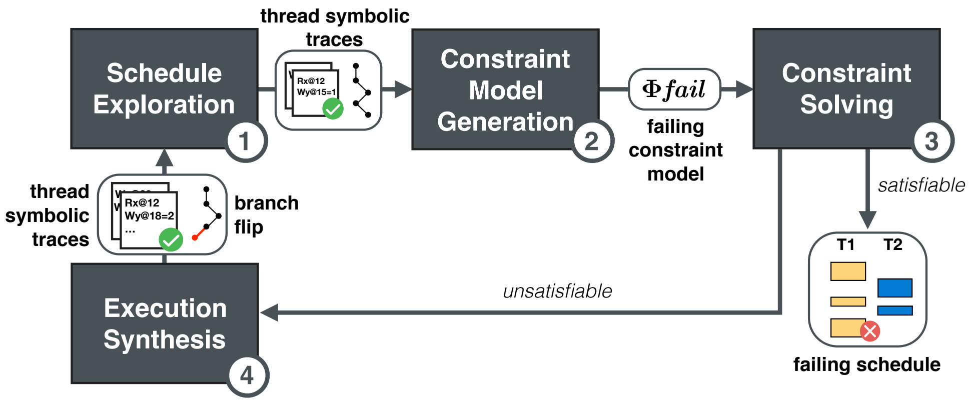


Figure 5.3: Detail of production-guided schedule search.

Cortex uses an SMT solver to check the satisfiability of the generated constraints. If the model is satisfiable, then the solver outputs the failing schedule. If the constraints are unsatisfiable, then there is no schedule that leads to a failure of the selected assertion for the given control-flow trace. Cortex applies this schedule exploration procedure to each execution in its database, reporting newly exposed failures as they manifest.

Note that *only* performing schedule exploration on executions observed in production will only expose strictly schedule-dependent failures. However, Cortex is not limited to these failures only, because it goes beyond schedule exploration with its execution synthesis technique.

5.2.2.2 Execution Synthesis

Execution synthesis generates new per-thread control-flow traces corresponding to *entirely novel executions* by making small perturbations in the control-flow observed in production runs. By synthesizing new executions with control-flow variations, and then applying schedule exploration to those synthesized executions, Cortex can expose new failures that are path- and schedule-dependent.

Synthesizing executions presents two main challenges: *i)* how to decide which alternate execution to synthesize, and *ii)* how to obtain per-thread symbolic traces for the alternate execution to be synthesized. Cortex addresses the first challenge with a novel heuristic denoted *branch condition flipping*. The heuristic perturbs the original control-flow observed during some production run, generating one or more new per-thread traces. Cortex addresses the second challenge using a combination of its trace database and symbolic execution. If Cortex has already observed some execution in which a thread followed the perturbed control-flow path, Cortex uses the per-thread symbolic trace for that path that is in its database. In contrast, if Cortex has not observed the perturbed path in some prior execution, then it synthesizes a new symbolic path trace by running a symbolic execution, guided by the perturbed control-flow trace.

Synthesizing a Control-flow Path. Cortex's synthesizes a new control-flow path by inverting path conditions on an existing path that are within a given distance from the selected assertion.

To identify the path conditions corresponding to the branches closest to the assertion, Cortex selects an execution from its database and generates a non-failing, multithreaded schedule using the following constraint model Φ_{ok}:

$$\Phi_{ok} = \neg\phi_{bug} \wedge \phi_{path} \wedge \phi_{sync} \wedge \phi_{rw} \wedge \phi_{mo} \tag{5.1}$$

This model is pretty much identical to the Φ_{fail} constraint system (see Equation 4.1), apart from the ϕ_{bug} constraint, which is negated in this case to allow generating a non-failing schedule. However, despite negating the failure constraint, note that Φ_{ok} differs from Φ_{root} (Equation 4.2) and Φ_{alt} (Equation 4.3) because it does not add constraints referring to a particular schedule to the constraint system (*i.e.*, Φ_{ok} does not strive to check the satisfiability of a single execution interleaving), as done by the latter two formulations.

After solving the constraints, Cortex examines the resulting schedule and selects the D branches closest to the assertion as candidates for inversion.

Cortex's path synthesis heuristic generates paths that are most similar to the original path trace first, generating new paths in order of deviation from the original. The first paths that the heuristic synthesizes are ones with a single branch outcome flipped, and Cortex gives priority to paths in which the flipped branch is closer to the assertion. Next, the heuristic generates new paths with two branch flips, again, prioritizing new paths with shorter total distance between branch flips and the assertion. Cortex's path synthesis heuristic continues considering complexes of increasingly many branch inversions up to the configurable threshold number D.

Figure 5.4a illustrates Cortex's path synthesis technique. There are two threads, *T1* and *T2*, and three branch conditions (Ⓐ and Ⓑ belong to *T1*, and Ⓒ belongs to *T2*). According to the non-failing schedule in Figure 5.4a, the closest branch to the assertion is Ⓐ, followed by Ⓑ

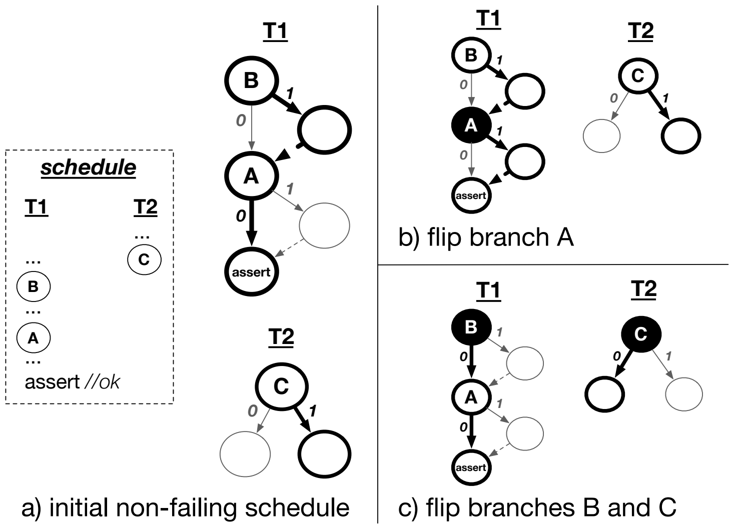


Figure 5.4: Branch condition flipping. Arrows and dashed arrows represent conditional and unconditional control-flow, respectively. Thicker arrows represent the execution path followed by the thread.

and Ⓒ. The figure also shows that the path conditions in the threads' symbolic traces are 10 (*i.e.*, taken, not taken) and 1 (*i.e.*, taken), respectively for *T1* and *T2*.

Figure 5.4b depicts the first branch condition that Cortex attempts to flip, namely Ⓐ. As a result, Cortex generates a new control-flow path for *T1* containing 11, while the trace for *T2* remains the same. Later, the path synthesis heuristic may generate another control-flow path by flipping the outcome of multiple branches. Figure 5.4c illustrates a case where Cortex simultaneously flips branches Ⓑ and Ⓒ, resulting in a path for *T1* containing 00 and a path for *T2* containing 0.

After synthesizing a new control-flow path, Cortex needs a new symbolic trace for the newly synthesized path. Cortex can either *find* an existing symbolic trace, or *synthesize* a new symbolic trace.

Finding a Symbolic Trace. The easiest way for Cortex to obtain a symbolic trace that is compatible with a newly synthesized control-flow path is to look for one in its database of traces collected from *any* prior, production execution. A compatible trace from the database must have an identical prefix of branch outcomes as the original, unperturbed trace, but must have the opposite outcome for the branch or branches flipped by the path synthesis heuristic.

When there is more than one compatible, symbolic trace in the database, Cortex considers each of them in turn, up to a maximum of N possible traces, and in ascending order of their path length. The tuple (D, N) allows tuning the search in terms of the number of different branches conditions flipped and the number of possible traces that are attempted for each branch flip. A high D means that Cortex flips path conditions far from the assertion, and a high N indicates that Cortex explores many paths with a common prefix.

Synthesizing a Symbolic Trace. When there is no trace in the database that matches a newly synthesized control-flow path, Cortex synthesizes a compatible trace using guided symbolic execution. Cortex uses the newly synthesized control-flow path to guide a symbolic execution of the thread up to, and including the flipped branch or branches. After reaching the flipped branch in the symbolic execution, Cortex has no information about which control-flow path to follow. Cortex allows the symbolic execution to run freely, exploring all paths, as in classical symbolic execution [King 1976; Cadar, Dunbar, & Engler 2008; Visser, Păsăreanu, & Khurshid 2004]. We heuristically stop the symbolic execution when it reaches the assertion or program exit along any path. We also stop symbolic execution after a configurable threshold timeout, to prevent the path explosion problem from hindering Cortex.

As an example, consider the scenario where Cortex has to synthesize the symbolic trace for *T1* required in Figure 5.4b. Cortex would run the program symbolically, forcing *T1* to take the branch 1 for the path condition Ⓑ, as well as for path condition Ⓐ. As *T1*'s execution ends with the assertion right after Ⓐ, Cortex would output a symbolic trace for *T1* that is compatible with the previously unobserved control-flow path 11.

5.2.3 Root Cause Isolation

Like prior systems on systematic concurrency testing [Huang 2015; Flanagan & Godefroid 2005], Cortex is able to report a newly exposed failing schedule, but unlike prior systems, Cortex also reports a concise summary of the failure's root cause. To summarize a failure's root cause, Cortex computes and reports a *differential path-schedule projection* (DPSP). DPSPs are an extension to *differential schedule projections*, presented in Chapter 4. Cortex computes DPSPs by analyzing an exposed failing execution and the original, non-failing execution that it was derived from. A DPSP reports the salient differences between the failing and non-failing schedule, including variation in their event orderings, data-flow behavior, and control-flow decisions. The key difference between DPSPs and the DSPs is that the latter do not incorporate differences in control-flow between a failing and a non-failing execution, while, critically, DPSPs include those differences.

Cortex produces the DPSP by computing a “diff” of the failing schedule against the non-failing schedule. To compute a DPSP, Cortex first compares the traces and prunes a prefix of operations common to both the failing and non-failing schedule. Cortex then examines data-flow in both traces and reports only data-flow edges that exist in one trace, but not the other. Control-flow variations are highlighted as data-flow variations involving operations that only executed in one trace, but not the other. DPSPs are helpful for debugging, because they allow developers to see only a very small number of relevant operations and data movement events, rather than forcing them to pore over a full execution schedule. Furthermore, DPSPs illustrate the failure alongside a very similar, but non-failing execution. The side-by-side comparison helps understand the failure and aids in debugging.

5.3 Running Example

This section synthesizes the entire Cortex debugging workflow using a detailed running example. Figure 5.5 shows how Cortex automatically computes the root cause of the failure in Figure 5.1.

Static analysis. Cortex's static analysis identifies and instruments basic blocks and shared variables. Figure 5.5a shows the program's control-flow graph. In the example code, z, w, y, and x are marked as symbolic.

Symbolic trace collection. The program executes in production, potentially many times, and a path profile for each thread is collected from each execution. From the path profiles, Cortex produces symbolic traces via symbolic execution. In particular, Cortex identifies each branch condition evaluated by each thread (enclosed in square brackets in Figure 5.5a) and symbolic execution follows those branches according to the path profile. Figure 5.5b shows the per-thread symbolic traces for three non-failing, production runs with different execution paths.[2] For instance, $\texttt{T1}_1$:`10` indicates that the trace for thread *T1* of production run 1 followed the control-flow path `10` (*i.e.*, taken, not taken). After producing the per-thread, symbolic traces, Cortex stores them in its trace database, depicted in Figure 5.5c as a prefix tree.

[2]In fact, the symbolic traces of Figure 5.5 are similar to those depicted in Figure 4.2c. However, here, we opted for not representing the actual variable symbolic symbols to improve readability.

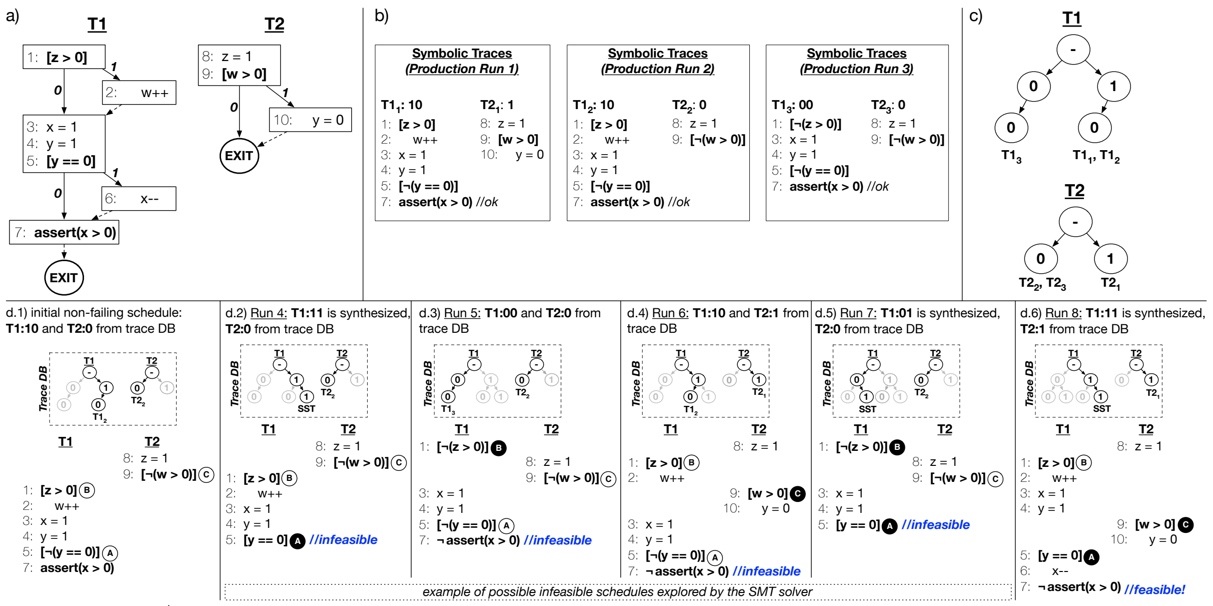

a)
T1
1: [z > 0]
2: w++
3: x = 1
4: y = 1
5: [y == 0]
6: x--
7: assert(x > 0)
EXIT
T2
8: z = 1
9: [w > 0]
10: y = 0
EXIT
b)
Symbolic Traces (Production Run 1)
T1₁: 10
T2₁: 1
Symbolic Traces (Production Run 2)
T1₂: 10
T2₂: 0
Symbolic Traces (Production Run 3)
T1₃: 00
T2₃: 0
//ok
c)
T1₃
T1₁, T1₂
T2₂, T2₃
T2₁
d.1) initial non-failing schedule: T1:10 and T2:0 from trace DB
d.2) Run 4: T1:11 is synthesized, T2:0 from trace DB
d.3) Run 5: T1:00 and T2:0 from trace DB
d.4) Run 6: T1:10 and T2:1 from trace DB
d.5) Run 7: T1:01 is synthesized, T2:0 from trace DB
d.6) Run 8: T1:11 is synthesized, T2:1 from trace DB
Trace DB
SST
//infeasible
//feasible!
example of possible infeasible schedules explored by the SMT solver

Figure 5.5: a) Schematic view of the program in Figure 5.1: boxes represent basic blocks, arrows depict conditional jumps (0 means *false* and 1 means *true*), dashed arrows depict unconditional jumps, round shapes represent the program's exit points, and [z > 0] represents a path condition; b) Per-thread symbolic traces path for three different correct production runs. $\mathsf{T1}_1$:10 indicates that the trace for thread *T1* from production run 1 has path id 10; c) Trace database, with per-thread path ids organized into prefix trees. The node label "-" indicates the root of the prefix tree. d) Production-guided schedule search employed by Cortex to find the failing schedule. SST stands for *synthesized symbolic trace.*

Production-guided search. Figures 5.5d.1-d.6 illustrate how Cortex uses production-guided search to expose a failing schedule from the non-failing, production schedules in its trace database.

First, Cortex tries to obtain a failing schedule by exploring the schedules that are compatible with the per-thread traces from production runs that are in the trace database. Cortex applies its schedule exploration algorithm (see Section 5.2.2.1) to production runs 1, 2 and 3. In this example, there is no failing schedule that simply interleaves the per-thread traces from any execution. Instead, Cortex needs to explore alternate executions that it derives from the observed executions via execution synthesis.

Cortex begins its search by arbitrarily selecting production run 2, which includes traces $\mathsf{T1}_2$ and $\mathsf{T2}_2$. Using the traces from this execution, Cortex generates a non-failing schedule by calling out to the SMT solver (Figure 5.5d.1). Cortex examines the non-failing schedule to identify the branches that are closest to the assertion. In the example, these branches are Ⓐ, Ⓑ (from trace $\mathsf{T1}_2$) and Ⓒ (from $\mathsf{T2}_2$).

In Figure 5.5d.2, Cortex explores a different execution path by flipping the branch condition Ⓐ. The resulting path prefix is thus obtained by inverting the second bit in the path of trace $\mathsf{T1}_2$, *i.e.*, by changing the path condition 10 to the path condition 11. Cortex checks its database for a symbolic trace for *T1* with the path prefix 11, but, in this example, there is no such path in the database. Consequently, Cortex needs to synthesize a new symbolic trace for *T1* with that prefix using symbolic trace synthesis.

Symbolic trace synthesis produces a symbolic trace T1:11. Cortex uses the generated trace, together with $\mathsf{T2}_2$, to synthesize a new execution that we refer to as “run 4”. Cortex performs schedule exploration on run 4, checking for interleavings of its threads’ operations that lead to a failure. The solver, however, yields *unsatisfiable* when evaluating the constraint system that encodes schedule exploration. The execution is infeasible because there is no feasible data-flow that allows the value of y at line 5 to be 0.

To continue its search for a feasible schedule, Cortex again applies its path synthesis heuristic to generate a new control-flow path to explore (Figure 5.5d.3). The next branch to flip is Ⓑ from trace $\mathsf{T1}_2$:10, which corresponds to the path 00. Cortex finds that its trace database for T1 already contains a trace with that path prefix (namely $\mathsf{T1}_3$). Cortex uses the trace that it found, alongside trace $\mathsf{T2}_2$, to synthesize a new execution (“run 5”). Cortex then performs schedule exploration on run 5, continuing its search for a feasible, failing alternate schedule.

Cortex proceeds according to this approach. When schedule exploration yields no failing schedule, Cortex synthesizes a new path, finds or synthesizes a new symbolic trace, creates a new execution, and re-applies schedule exploration. Figure 5.5d.4 and Figure 5.5d.5 show subsequent applications of the approach, which consist of runs 6 and 7, respectively. Note that, in Figure 5.5d.5, Cortex inverts the outcome of *two* branches, instead of a single branch because, at this point in its search, it has exhausted all options involving only a single branch inversion.

In Figure 5.5d.6, Cortex identifies a feasible, failing schedule for a newly synthesized execution that includes a synthesized symbolic trace for *T1* (previously generated in run 4), and $\mathsf{T2}_1$. The traces for T1 and T2 in the execution for which there is a failing schedule are the result of inverting the outcome of both branches Ⓐ and Ⓒ. Note that without Cortex’s unique ability to explore both schedule *and* path variations, this failure would not have been exposed.

Root cause isolation. With the failing and non-failing schedules that are the result of production-guided search, Cortex generates the DPSP depicted in Figure 5.6. On the left side

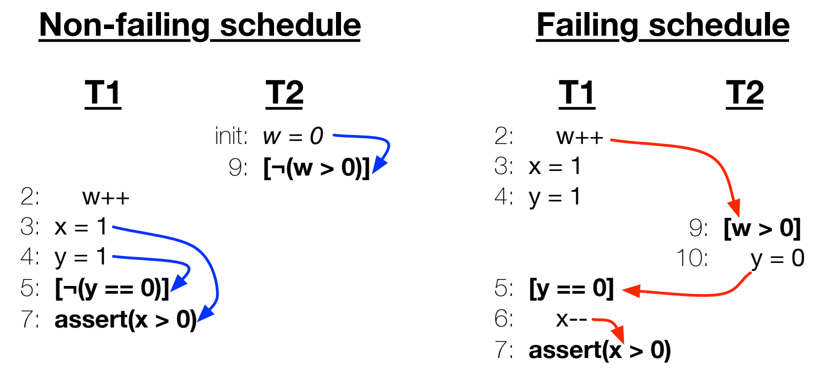


Figure 5.6: Differential path-schedule projection.

is the non-failing schedule and on the right is the failing schedule. The DPSP does not show operations that the two schedules have in common, highlighting instead only the parts of the execution trace that are different. The DPSP illustrates (in the **bold** lines) which control-flow outcomes differ between the schedules. The arrows in the figure indicate data-flow edges that exist in one schedule, but not in the other. Together these properties of the DPSP show the root cause of the failure: the failure is attributable to a change in the order of operations in the schedule, the data-flow changes resulting from those ordering changes, and the control-flow changes stemming from the changes in data-flow.

In particular, the example shows that the branch condition [w>0], which evaluates false in the non-failing schedule, becomes true in the failing schedule, because w at line 9 reads the value 1 (written by *T1* at line 2) rather than the initial value 0. Consequently, *T2* executes line 10 and sets y to 0, allowing branch condition [y==0] to be true at line 5. In contrast, in the non-failing schedule, *T1* takes the branch outcome corresponding to the condition ¬[y==0], because y at line 5 necessarily reads the value 1 written at line 4.

Finally, the DPSP shows that the assertion failure in the failing schedule is due to the value of x being decremented by *T1* at line 6. Conversely, the execution ends successfully in the non-failing schedule because the read of x at line 7 returns the value 1, written at line 3.

Note that Symbiosis simply reorders events in the failing schedule to obtain an alternate, non-failing schedule and produce a differential schedule projection. As such, Symbiosis would not be able to generate a DPSP like the one of Figure 5.6, because the failing schedule and the non-failing schedule for this case comprise not only sequences of different events, but also differing path conditions.

5.4 Implementation

We implemented a prototype of Cortex for Java programs. We use Soot [Vallée-Rai, Co, Gagnon, Hendren, Lam, & Sundaresan 1999] to perform the static analysis of Java bytecode, namely to inject probes at the beginning of each basic block that allow recording the path profile at runtime. Moreover, as in the previous chapters, we leverage Soot's *thread-local objects* (TLO) escape analysis to compute a sound over-approximation of the set of shared variables in Java programs. For each access on a shared variable we log an entry into a trace file containing the variable's reference and the source code line. Cortex consults the trace file during symbolic execution to identify which operations should to treat symbolically.

Cortex's production-guided schedule search and DPSP generation were implemented in around 1,200 lines of C and C++ code that extended our prototype of Symbiosis. In particular, we extended Symbiosis to *i)* efficiently store symbolic traces from multiple production runs, *ii)* expose strictly schedule-dependent, as well as path- and schedule-dependent concurrency bugs using traces from non-failing executions, and *iii)* perform symbolic trace synthesis during multiple path exploration.

Cortex organizes its database of per-thread path traces, represented as bit strings, into *tries* (*i.e.*, *prefix trees*). Tries are typically used for string retrieval and contain one node for every common prefix of stored strings. In Cortex's implementation, if strings representing two different traces share a prefix of n bits, the corresponding executions followed the same path until the n^{th} branch decision. Cortex's use of a trie to store traces minimizes the storage required for large numbers of traces collected from production.

Cortex uses Java PathFinder (JPF) [Visser, Păsăreanu, & Khurshid 2004] for symbolic execution and Z3 [De Moura & Bjørner 2008] to solve SMT constraints. We modified Java PathFinder to integrate it with the other parts of Cortex. First, when generating symbolic traces for the production run per-thread path profiles, we ignore states that do not conform with execution path traced at runtime. This allows guiding the symbolic execution along the original paths only. Second, when synthesizing new symbolic traces, we force JPF to follow original path solely up to the branch condition flip point. After that, JPF switches to the traditional mode, where it explores all branches for each condition on symbolic variables, using a breadth-first search heuristic. In this mode, we also set a timeout to the exploration, in order to cope with path explosion.

Our Cortex prototype assumes that bugs in programs are expressed in the code as assertion invariants. Assuming that production software contains assertions is reasonable, as many major industrial environments use production assertions and tracing [Sigelman, Barroso, Burrows, Stephenson, Plakal, Beaver, Jaspan, & Shanbhag 2010]. In our experiments, we added these assertions when they were not initially present. For cases where the error had the following form on the left, we inject the assertion as indicated on the right:

```
if(cond){               if(cond){
  //error                 assert(false)
}                        //error
else{                   }
  ...                   else{
}                         assert(true)
                         ...
                        }
```

Since JPF does not support arrays of symbolic length, we have also modified these cases in our experiments to have a constant size, without affecting the original buggy behavior of the program.

A prototype of Cortex is publicly available at `http://github.com/nunomachado/cortex-tool`.

5.5 Evaluation

Our evaluation of Cortex focuses on answering the following three questions:

- How efficient is Cortex in collecting symbolic traces from production runs? (Section 5.5.1)
- How effective is Cortex's production-guided search in finding concurrency bugs? (Section 5.5.2 and Section 5.5.3)
- How effective is Cortex in isolating the root cause of concurrency bugs? (Section 5.5.4)

We evaluated Cortex on the wide variety of multithreaded benchmarks shown in Table 5.1. These benchmarks have been used in prior work on concurrency debugging [Farchi, Nir, & Ur 2003b; Huang 2015; Flanagan & Qadeer 2003; Huang & Rauchwerger 2015]. We used 11 programs from the IBM ConTest benchmark suite [Farchi, Nir, & Ur 2003b]; *StringBuf*, a test driver of a bug in the Java JDK1.4 [Huang 2015]; *ExMCR*, a micro-benchmark used by J. Huang *et al.* [Huang 2015] to illustrate the benefits of MCR against other stateless model checking techniques. We have also tested with two real-world application bugs, namely *Pool* (which consists of a data race in Apache Commons Pool) and *Cache4j* (uncaught exception due to a data race). When presenting results, we sort test cases by "difficulty", *i.e*, benchmarks with fewer branches and smaller search spaces appear first in the tables.

We modeled the data collection of a production environment by executing each program 100 times and ensuring that none of the 100 executions triggered the bug. From these production runs, we generated non-failing, symbolic traces and applied production-guided search to expose a failing schedule for each benchmark. Once again, for *Cache4j*, we have experimented with different workloads to assess the scalability of the constraint solving phase. Concretely, we reran this test case by varying the worker thread's update loop to have 1 (small), 5 (medium), and 10 (large) iterations.

The experiments were conducted on an 8-core, 3.5Ghz machine with 32GB of memory, running Ubuntu 10.04.4.

5.5.1 Trace Collection Efficiency

The most important result is that for all of the benchmarks that we considered, Cortex exposed a new failing execution based on a small handful of observed, non-failing schedules, and did so in a practical amount of time. Table 5.1 reports the time and storage overhead imposed by Cortex on production runs to capture path profiles, as well as the time required to compute symbolic trace collection. We report average time values across all executions (concrete and symbolic) for each benchmark.

Cortex's path profiling overhead ranges from from 2.4% in *Loader* to 21.7% in *Cache4j (L)*. The overhead is tolerable, even for production, and similar to other prior work in this area [Machado, Lucia, & Rodrigues 2015; Huang, Zhang, & Dolby 2013]. Better software path

Table 5.1: Benchmarks and performance. *LOC* stands for lines of code, *#Threads* is the number of threads, *Profiling Overhead* indicates the runtime slowdown due to path profiling, *Log Size* shows the size of the path profiles, *Symbolic Execution* indicates the time required to perform symbolic execution, *#Branches* is number of branches, and *#Shared Events* is number of shared variable accesses, in each benchmark.

Program	**LOC**	**#Threads**	**Profiling Overhead**	**Log Size**	**Symbolic Execution**	**#Branches**	**#Shared Events**
Account	373	5	18.1%	1KB	0.6s	1	244
Critical	76	3	17.3%	260B	0.59s	4	36
ExMCR	95	3	17.1%	170B	0.42s	4	56
PingPong	388	6	18.9%	226B	0.47s	5	66
Piper	280	5	18.6%	470B	1.11s	12	182
Airline	136	8	9.1%	252B	2.53s	15	77
Garage	554	7	6.7%	105KB	56.50s	22	284
BubbleSort	376	6	13.4%	1KB	0.59s	24	161
Manager	219	5	16.4%	1.4KB	0.79s	56	331
Loader	146	11	2.4%	4KB	0.91s	56	386
StringBuf	1339	3	19.9%	1KB	1.13s	65	331
TicketOrder	246	4	9.5%	892B	0.85s	69	354
BufWriter	272	5	20.4%	4.8KB	4.84s	89	1245
Pool	10K	3	2.5%	960B	1.4s	21	198
Cache4j (S)	2.3K	4	18.4%	3KB	1.02s	51	541
Cache4j (M)	2.3K	4	20%	15KB	2.01s	233	2364
Cache4j (L)	2.3K	4	21.7%	21KB	3.47s	309	3105

profiling [Ball & Larus 1994] or hardware support [Vaswani, Thazhuthaveetil, & Srikant 2005] are orthogonal techniques that would reduce this overhead.

Regarding space overhead, Cortex produces traces with sizes ranging from 170B in *ExMCR* to 105KB in *Garage*. The symbolic execution time is typically low as well: JPF produced a symbolic trace in less than one minute for all programs. The programs with larger path profiles are also the ones with more shared symbolic events (*e.g.*, *BufWriter* and *Garage*). *Garage* has a long trace and symbolic execution time because it uses busy waiting.

5.5.2 Bug Exposing Efficacy

Table 5.2 reports experimental results that allow assessing Cortex's efficacy in finding failing schedules. Columns 3 and 4 of the table together show that Cortex was able to find a failing schedule for all programs, including ones with failures dependent on both the path and schedule ("path- and schedule-dependent"). We now characterize Cortex's ability to expose new failures.

Strictly schedule-dependent bugs. Column 3 of Table 5.2 shows that schedule exploration alone works for only 8 out of the 17 benchmarks, namely *Account*, *PingPong*, *Airline*, *StringBuf*, *BufWriter*, and the three *Cache4j* scenarios. The reason schedule search alone is adequate for these benchmarks is that these eight cases include assertions of the form `if(cond){assert(false)} else{assert(true)}`. Cortex finds the failing schedule via schedule exploration alone because the failures depend only on strictly schedule-dependent data flow to `cond`.

Efficiency of production-guided search. Column 4 in Table 5.2 shows that production-guided search finds a failing schedule for our path- and schedule-dependent bugs. Column 5 shows the number of branch outcome inversions Cortex performed to expose each failure. 3 out of 9 cases required inverting only the single closest branch to the assertion. This outcome supports our observation that failing executions are lurking in production, and that perturbing production executions is an effective search strategy for these otherwise elusive failures. The need for branch inversions, even in our production-guided search reinforces the fact that schedule exploration alone is insufficient.

Production run diversity. Collecting a diversity of production executions expedites Cortex's search for failures because it populates the trace database with traces, obviating the more costly execution synthesis step. Column 2 of Table 5.2 shows how many distinct non-failing executions Cortex observed during 100 runs of each benchmark. For all benchmarks except *BufWriter* and *Pool*, less than 50% of the collected executions are distinct. The data suggest that, even in small numbers of runs, executions are diverse and Cortex can leverage a large trace database in a large deployment.

Search parameters. The column labelled as (D, N) characterizes parameters used during Cortex's search for failures. Programs for which Cortex finds the failure with a single branch flip exhibit the pair (1,1) for (D, N). For those programs, Cortex exposed a bug by inverting the outcome of the single branch that was closest to the assertion.

In contrast, in *Critical*, *ExMCR*, *Garage*, *BubbleSort*, *Loader*, and *Pool* the optimal value for (D, N) varies significantly. For *Critical*, Cortex found the failing schedule after inverting two branch conditions (the *two* closest to the assertion) and performed schedule exploration using three different symbolic traces for each one of the two paths. For *ExMCR*, Cortex was able to find the failing execution in 6 attempts, but in this case it required 6 different combinations of branch inversions. Note that the number of attempts is actually greater than the product of the search parameters (D, N) for *ExMCR*, because Cortex needed to flip a combination of two

Table 5.2: Bug Exposing Results. Column 2 shows the number of different correct production runs observed; Column 3 marks bug found by schedule exploration only; Columns 4-8 provide details of production-guided search; Last column depicts the average time to solve the corresponding *satisfiable* SMT system.

Program	**#Different Production Runs**	**Schedule-Dependent Only**	**Path- and Schedule-Dependent**					**SMT Solving Time**
				Tries	**(D,N)**	**#Branches Flipped**	**#Synthesized Traces**	
Account	1	✓						29s
Critical	23		✓	6	(2,3)	2	4	<1s
ExMCR	1		✓	6	(4,1)	6	6	<1s
PingPong	39	✓						<1s
Piper	33		✓	1	(1,1)	1	1	1s
Airline	3	✓						<1s
Garage	2		✓	9	(3,4)	6	6	2s
BubbleSort	26		✓	4	(4,1)	4	4	<1s
Manager	39		✓	1	(1,1)	1	0	9s
Loader	1		✓	11	(11,1)	11	10	25s
StringBuf	12	✓						9s
TicketOrder	47		✓	1	(1,1)	1	0	1s
BufWriter	57	✓						2h56m
Pool	59		✓	17	(5,4)	15	8	1s
Cache4j (S)	11	✓						5s
Cache4j (M)	29	✓						1h30m
Cache4j (L)	37	✓						2h8m

branches simultaneously in order to trigger this failure.[3] *Garage* required fewer attempts than $D \times N$. The reason is that Cortex selected traces for some attempts that included *redundant* execution paths. Cortex discarded the redundant paths and found a failing schedule using the 4^{th} trace for the 6^{th} combination of branch flips.

For *BubbleSort* and *Loader*, Cortex experimented with only one trace per branch inversion, but it searched through 4 and 11 branch inversions, respectively, to compute the failing schedule. *Pool*, in turn, was the program for which Cortex required more tries and flips of branch conditions to expose the failure. This is because the only combination of branch inversions that allowed finding the failing schedule corresponded to flipping simultaneously the 4^{th} and 5^{th} branches closest to the assertion.

In conclusion, these results show that search parameters (D, N) affect significantly the number of attempts that production-guided schedule search requires to expose the concurrency bug.

Solving Time. The last column of Table 5.2 reports the average amount of time that the SMT

[3]We note that there are $2^D - 1$ different combinations of branch condition flips that can be attempted for a given search parameter D.

solver took to solve the constraint system (this value comprises only the case when the solver yielded *satisfiable*, because reporting *unsatisfiable* took at most 3 seconds for our test cases). The data shows that solving time is low for most cases, *i.e.*, a couple of seconds. The exception are benchmarks *BufWriter*, *Cache4j (M)*, and *Cache4j (L)*. Once more, this is due to the higher number of shared events in these programs. In particular, the solver took almost 3 hours for *BufWriter*, because the SMT constraint formulation for this program contains more than 920K read-write constraints and more than 6.5K locking constraints, which have a big impact in the solving time for this kind of constraint systems [Huang, Zhang, & Dolby 2013; Machado, Lucia, & Rodrigues 2015].

5.5.3 Cortex Compares Favorably to Systematic Testing

Unlike Cortex, systematic testing techniques search by fully exploring the space of possible executions. We directly compared Cortex to two state-of-the-art systematic testing techniques, namely MCR [Huang 2015] and iterative context bounding with dynamic partial order reduction (ICB-DPOR) [Coons, Musuvathi, & McKinley 2013]. Similarly to Cortex, MCR uses an SMT constraint-based approach to efficiently explore the space of possible schedules of a multithreaded execution in search for concurrency bugs. In particular, MCR starts from a concrete *seed interleaving* and builds a *maximal causal model* that allows checking correctness properties on all execution schedules equivalent to that seed interleaving. To further explore the state space, MCR iteratively generates new non-redundant schedules by enforcing read operations to return different values during a re-execution of the program. MCR then uses the newly generated schedules as seed interleavings for subsequent iterations.

On the other hand, ICB-DPOR simply bounds the number of thread preemptions that can occur when systematically exercising different execution schedules, thus not accounting for redundant interleavings (*i.e.*, interleavings that produce the same values for read operations).

Our goal is to show that Cortex examines *fewer executions* before exposing a failure than other approaches. We reproduce results for MCR and ICB-DPOR reported by J. Huang [Huang 2015] for the subset of benchmarks that have been used with all three systems. Table 5.3 reports the comparison.

The data show that, for most cases, Cortex searches orders of magnitude fewer executions

Table 5.3: Comparison between Cortex and other systematic concurrency testing techniques. Data for MCR and ICB-DPOR as reported by J. Huang [Huang 2015] ("*" indicates bugs that are schedule-dependent only). Shaded cells indicate the cases where Cortex outperforms the other systems.

Program	**#Attempts to find failing schedule**		
	Cortex	**MCR**	**ICB-DPOR**
Account*	1	2	20
ExMCR	6	46	3782
PingPong*	1	2	37
Airline*	1	9	19
BubbleSort	4	4	400
StringBuf*	1	2	10
Pool	17	3	6

than ICB-DPOR, and considerably fewer than MCR. The standout is *ExMCR*; *ExMCR* is a micro-benchmark that was designed to be *adversarial* to systematic concurrency testing systems. Cortex exposes the bug after searching just 6 executions, substantially outperforming both other systems.

The only benchmark where Cortex required more attempts to find the failing schedule than the other approaches was *Pool*. As mentioned before, for this program, Cortex was only able to trigger the failure after inverting the 4^{th} and 5^{th} branches at the same time. Hence, Cortex ended up spending time exploring combinations of branch flips that, despite being closer to the assertion, were ineffective to expose the failing schedule.

We believe the aforementioned scenario to be infrequent in practice, as shown by the outcomes of the other benchmarks. Therefore, we argue that the results in Table 5.3 further support our observation production-guided search is effective.

5.5.4 Differential Path-Schedule Projections Efficacy

We computed DPSPs for all of our benchmarks, using observed (and synthesized) non-failing schedules and corresponding failing schedules exposed by Cortex. We evaluated DPSPs by comparing the number of data-flows and events in the DPSPs to those in full schedules. Table 5.4 summarizes our results.

The data show that DPSPs are simpler than full, failing schedules. DPSPs include only the salient differences between the failing and the correct executions, directing the developer's attention towards the most relevant events and the data-flows involved in the root cause of the failure. On average, Cortex produced DPSPs with 83% fewer data-flows and 50% fewer events

Table 5.4: DPSP conciseness. Reduction achieved by Cortex in terms of number of data-flows and events with respect to full failing schedules ("*" indicates bugs that are schedule-dependent only).

Program	**#Data-flows Full**	**#Data-flows DPSP (%Reduction)**	**#Events Full**	**#Events DPSP (%Reduction)**
Account*	139	6 (↓**96%**)	244	58 (↓**76%**)
Critical	13	7 (↓**46%**)	36	13 (↓**64%**)
ExMCR	16	8 (↓**50%**)	56	39 (↓**30%**)
PingPong*	16	1 (↓**94%**)	66	5 (↓**92%**)
Piper	57	2 (↓**96%**)	182	51 (↓**72%**)
Airline*	29	3 (↓**90%**)	77	18 (↓**77%**)
Garage	139	26 (↓**81%**)	284	235 (↓**17%**)
BubbleSort	69	15 (↓**78%**)	161	144 (↓**11%**)
Manager	142	32 (↓**77%**)	331	220 (↓**34%**)
Loader	179	21 (↓**88%**)	386	281 (↓**27%**)
StringBuf*	115	1 (↓**99%**)	331	40 (↓**88%**)
TicketOrder	183	45 (↓**75%**)	354	290 (↓**18%**)
BufWriter*	745	95 (↓**87%**)	1245	1216 (↓**2%**)
Pool	73	34 (↓**53%**)	198	123 (↓**38%**)
Cache4j (S)*	211	1 (↓**99.5%**)	541	11 (↓**98%**)
Cache4j (M)*	862	3 (↓**99.7%**)	2364	1371 (↓**42%**)
Cache4j (L)*	1139	3 (↓**99.7%**)	3105	1094 (↓**65%**)

than full schedules. Considering benchmarks with path- and schedule-dependent bugs alone, the average reduction values are 72% and 35%, respectively for data-flows and events. These results provide evidence that DPSPs are a useful asset for root cause diagnosis, not only for schedule-dependent only failures, but also for path- and schedule-dependent failures.

Summary

This chapter presented Cortex, a system that is not only able to find concurrency bugs in programs, but also helps the programmer in identifying their root cause. For this, Cortex generates differential path-schedule projections (DPSPs) that capture the differences between non-failing and failing executions, even when threads follow different paths in each execution. These DPSPs have from 46% to 99% less data-flows than full failing executions, strongly simplifying the task of identifying the branches that are involved in the bug.

Contrary to most previous work, Cortex does not require a failure to be observed in production to avoid exploring the full space of possible executions. Instead, it is able to use non-failing executions from production runs as a starting point for exploration, generating synthetic executions that are likely to expose a bug (when it exists). Interestingly, with the benchmarks used in this work, Cortex was able to generate failing executions after a few (more precisely, from just 1 to 15, and 4 on average) cleverly guided branch flips.

6 Final Remarks

Concurrent programming is of paramount importance to take advantage of the pervasive parallel computer architectures. Unfortunately, the inherent complexity of concurrent programs opens the door for various types of concurrency bugs. Concurrency bugs are notoriously hard to debug and fix. First, the inherent non-deterministic nature of concurrency bugs makes their reproduction challenging. Second, it is hard to isolate the root cause of a concurrency error due to the large number of thread operations and interactions among them in failing schedules. Finally, failing schedules are difficult to expose because they usually stem from very specific thread interleavings, which manifest rarely. As a consequence, concurrent programs are often shipped with concurrency bugs that can originate failures in production and degrade the systems' reliability.

In this thesis, we develop techniques for the replay, root cause diagnosis and exposing of concurrency bugs in deployed programs. In particular, this dissertation introduces:

- A novel *cooperative record and replay approach* that leverages collaborative partial logging and in-house statistical techniques to effectively combine partial logs and produce a full log capable of reproducing the failure.

- A novel *differential schedule projection technique* that uses symbolic constraint solving to spot the important control-flow and data-flow changes between a failing and a non-failing schedule, which comprise the failure's root cause.

- A novel *production-guided schedule search* that uncovers failing schedules by exploring variations in the schedule and control-flow behavior of non-failing executions.

Chapters 3–5 describe in detail the aforementioned techniques. We have built prototypes for all the techniques described in this thesis and showed, by means of experimental evaluations, that our solutions are effective, efficient, and compare favorably to previous state-of-the-art systems. We believe that the techniques proposed in this thesis open a number of interesting questions that can be subject of further research. We outline future work as follows.

- *Informed Partial Logging.* The cooperative record and replay approach (described in Chapter 3) relies on random partial logging. Although often effective, randomly distributing the information to be recorded across multiple instances of the program disregards some issues (*e.g.*, load balancing, log overlapping, and shared variable dependencies) that may impact the efficiency and efficacy of our technique. Our early results on this line of research have shown that, for instance, by taking into account factors such as correlation of shared variables and log overlapping when devising the partial log strategy, one can achieve analogous replay capabilities with a significantly smaller number of user instances [Machado, Romano, & Rodrigues 2013]. Also, we believe that other types of static analysis (namely, program slicing and dependency analysis) could benefit cooperative record and replay.

- *Non-Assertion Bugs.* The constraint systems we use in Chapters 4 and 5 to uncover failing schedules and generate the DPSPs assume that failures manifest as assertion violations. Although assertions are commonly placed in code during development, concurrency bugs might not always be expressed as invariant failures. One could address this issue by extending these constraint models to check for other type of concurrency bugs (*e.g.*, data races [Huang 2015] or deadlocks) in addition to assertion violations.

- *Long Executions.* As shown in our experiments in Chapters 4 and 5, when the execution has a large number of shared events, the SMT solver can take a long time to solve the constraint system. For long executions, this problem is exacerbated and it can become hard to expose a failing schedule in a reasonable amount of time.

  To improve the scalability of the constraint solving phase, one could capture lightweight information, namely via cooperative partial logging (see Chapter 3), regarding the thread orderings observed at runtime. This data could then be used to prune the constraint model (by fixing some read-write linkages), without compromising the ability to expose failing schedules.

- *Privacy.* The work in this thesis relies on information captured from executions occurring in deployed programs. As such, for some cases, the data collected from the user instances may contain sensitive information. Although privacy concerns are outside the scope of this thesis, we believe that our techniques could benefit from anonymization mechanisms. For instance, in the context of root cause isolation, it would be interesting to investigate if a given failure could still be triggered by a thread interleaving in a slightly different path, thus allowing to anonymize the original control-flow executed by the end user [Matos, Garcia, & Romano 2015].

- *Automated Bug Fixing.* Although automated bug fixing is a complex and open research problem [Khoshnood, Kusano, & Wang 2015; Jin, Zhang, Deng, Liblit, & Lu 2012], we argue that the information provided by our differential path-schedule projections could be leveraged to automatically generate patches for concurrency bugs. For example, to fix atomicity violations, one could inject synchronizations operations to avoid the erroneous interleaving reported in the DPSP.

- *Distributed Systems Debugging.* Many of the concurrency problems addressed in this thesis also affect large-scale distributed systems, such as distributed databases, scalable computing frameworks, and social media platforms [Leesatapornwongsa, Lukman, Lu, & Gunawi 2016]. However, as distributed systems typically run complex distributed protocols on hundreds/thousands of independent machines, they are prone to concurrency bugs stemming from other non-deterministic factors not addressed in this thesis (*e.g.*, message arrivals, node crashes, reboots, and timeouts). Nevertheless, we believe that some of our techniques (namely, the differential path-schedule projections) could be also generalized to diagnose concurrency bugs in distributed environments, as long as the idiosyncratic factors of this kind of applications are possible to model as SMT constraints.

Bibliography

Altekar, G. & I. Stoica (2009). Odr: Output-deterministic replay for multicore debugging. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP '09, pp. 193–206. ACM.

Arulraj, J., P.-C. Chang, G. Jin, & S. Lu (2013). Production-run software failure diagnosis via hardware performance counters. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pp. 101–112. ACM.

Arulraj, J., G. Jin, & S. Lu (2014). Leveraging the short-term memory of hardware to diagnose production-run software failures. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pp. 207–222. ACM.

Associated Press, T. (2004). General electric acknowledges north-eastern blackout bug. http://www.securityfocus.com/news/8032. Accessed: 2016-1-18.

Bacon, D. F. & S. C. Goldstein (1991). Hardware-assisted replay of multiprocessor programs. In *Proceedings of the International Workshop on Parallel and Distributed Debugging*, PADD '91, pp. 194–206. ACM.

Ball, T. & J. R. Larus (1994, July). Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst. 16*(4), 1319–1360.

Beizer, B. (1990). *Software Testing Techniques (2nd Edition)*. New York, NY, USA: Van Nostrand Reinhold Co.

Boehm, H.-J. (2011). How to miscompile programs with "benign" data races. In *Proceedings of the International Conference on Hot Topic in Parallelism*, HotPar'11, pp. 3–9. USENIX Association.

Bravo, M., N. Machado, P. Romano, & L. Rodrigues (2013). Towards effective and efficient search-based deterministic replay. In *Proceedings of the Workshop on Hot Topics in De-*

pendable Systems, HotDep '13, pp. 10:1–10:6. ACM.

Bressoud, T. C. & F. B. Schneider (1995). Hypervisor-based fault tolerance. In *Proceedings of the International Symposium on Operating Systems Principles*, SOSP '95, pp. 1–11. ACM.

Bucur, S., V. Ureche, C. Zamfir, & G. Candea (2011). Parallel symbolic execution for automated real-world software testing. In *Proceedings of the European Conference on Computer Systems*, EuroSys '11, pp. 183–198. ACM.

Burckhardt, S., P. Kothari, M. Musuvathi, & S. Nagarakatte (2010). A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '10, pp. 167–178. ACM.

Burnim, J. & K. Sen (2008). Heuristics for scalable dynamic test generation. In *Proceedings of the International Conference on Automated Software Engineering*, ASE '08, pp. 443–446. IEEE Computer Society.

Cadar, C., D. Dunbar, & D. Engler (2008). Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the International Conference on Operating Systems Design and Implementation*, OSDI '08, pp. 209–224. USENIX Association.

Candea, G. (2011). Exterminating bugs via collective information recycling. In *Proceedings of the Workshop on Hot Topics in Dependable Systems*, HotDep '11. IEEE Computer Society.

Chen, Y., S. Zhang, Q. Guo, L. Li, R. Wu, & T. Chen (2015, September). Deterministic replay: A survey. *ACM Comput. Surv. 48*(2), 17:1–17:47.

Choi, J.-D. & H. Srinivasan (1998). Deterministic replay of java multithreaded applications. In *Proceedings of the International Symposium on Parallel and Distributed Tools*, SPDT '98, pp. 48–59. ACM.

Choi, J.-D. & A. Zeller (2002). Isolating failure-inducing thread schedules. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '02, pp. 210–220. ACM.

Chow, J., T. Garfinkel, & P. M. Chen (2008). Decoupling dynamic program analysis from execution in virtual environments. In *Proceedings of the USENIX Annual Technical Conference*, ATC'08, pp. 1–14. USENIX Association.

Committee, C. S. (2010). Programming Languages – C (draft). C standards committee paper JTC1 SC22 WG14 N1539. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1539.pdf. Accessed: 2016-2-2.

Committee, C. S. & P. Beker (2011). Programming Languages – C++ (draft). C++ standards committee paper JTC1 SC22 WG21 N3242. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf. Accessed: 2016-2-2.

Coons, K. E., M. Musuvathi, & K. S. McKinley (2013). Bounded partial-order reduction. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pp. 833–848. ACM.

CVEdetails (2016). CVE security vulnerabilities related to races. http://www.cvedetails.com/vulnerability-list/cweid-362/vulnerabilities.html. Accessed: 2016-1-18.

De Moura, L. & N. Bjørner (2008). Z3: An efficient smt solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pp. 337–340. Springer-Verlag.

Dunlap, G. W., D. G. Lucchetti, M. A. Fetterman, & P. M. Chen (2008). Execution replay of multiprocessor virtual machines. In *Proceedings of the International Conference on Virtual Execution Environments*, VEE '08, pp. 121–130. ACM.

Elmas, T., J. Burnim, G. Necula, & K. Sen (2013). Concurrit: A domain specific language for reproducing concurrency bugs. In *Proceedings of the International Conference on Programming Language Design and Implementation*, PLDI '13, pp. 153–164. ACM.

Emmi, M., S. Qadeer, & Z. Rakamarić (2011). Delay-bounded scheduling. In *Proceedings of the International Symposium on Principles of Programming Languages*, POPL '11, pp. 411–422. ACM.

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

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

Farzan, A., A. Holzer, N. Razavi, & H. Veith (2013). Con2colic testing. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ESEC/FSE '13, pp. 37–47. ACM.

Flanagan, C. & S. N. Freund (2004). Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proceedings of the International Symposium on Principles of Programming Languages*, POPL '04, pp. 256–267. ACM.

Flanagan, C., S. N. Freund, & J. Yi (2008). Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *Proceedings of the International Conference on Programming Language Design and Implementation*, PLDI '08, pp. 293–303. ACM.

Flanagan, C. & P. Godefroid (2005). Dynamic partial-order reduction for model checking software. In *Proceedings of the International Symposium on Principles of Programming Languages*, POPL '05, pp. 110–121. ACM.

Flanagan, C. & S. Qadeer (2003). A type and effect system for atomicity. In *Proceedings of the International Conference on Programming Language Design and Implementation*, PLDI '03, pp. 338–349. ACM.

Fonseca, P., R. Rodrigues, & B. B. Brandenburg (2014). Ski: Exposing kernel concurrency bugs through systematic schedule exploration. In *Proceedings of the International Conference on Operating Systems Design and Implementation*, OSDI'14, pp. 415–431. USENIX Association.

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

Gibbons, P. B. & E. Korach (1997, August). Testing shared memories. *SIAM J. Comput. 26*(4), 1208–1244.

Godefroid, P. (1991). Using partial orders to improve automatic verification methods. In *Proceedings of the International Workshop on Computer Aided Verification*, CAV '90, pp. 176–185. Springer-Verlag.

Godefroid, P. (1996). *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* Springer-Verlag New York, Inc.

Godefroid, P. (1997). Model checking for programming languages using verisoft. In *Proceedings* *of the International Symposium on Principles of Programming Languages*, POPL '97, pp. 174–186. ACM.

Godefroid, P. (2007). Compositional dynamic test generation. In *Proceedings of the International Symposium on Principles of Programming Languages*, POPL '07, pp. 47–54. ACM.

Godefroid, P., N. Klarlund, & K. Sen (2005). Dart: Directed automated random testing. In *Proceedings of the International Conference on Programming Language Design and Implementation*, PLDI '05, pp. 213–223. ACM.

Goodman, J. R. (1989, 03). Cache consistency and sequential consistency. Technical Report 61, SCI Committee.

Halpert, R. L., C. J. F. Pickett, & C. Verbrugge (2007). Component-based lock allocation. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pp. 353–364. IEEE Computer Society.

Hamming, R. (1950). Error Detecting and Error Correcting Codes. *Bell System Technical Journal 26*(2), 147–160.

Herlihy, M. P. & J. M. Wing (1990, July). Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*(3), 463–492.

Hoare, C. A. R. (1974, October). Monitors: An operating system structuring concept. *Commun. ACM 17*(10), 549–557.

Honarmand, N., N. Dautenhahn, J. Torrellas, S. T. King, G. Pokam, & C. Pereira (2013). Cyrus: Unintrusive application-level record-replay for replay parallelism. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pp. 193–206. ACM.

Huang, J. (2015). Stateless model checking concurrent programs with maximal causality reduction. In *Proceedings of the International Conference on Programming Language Design and Implementation*, PLDI '15, pp. 165–174. ACM.

Huang, J., P. Liu, & C. Zhang (2010). Leap: Lightweight deterministic multi-processor replay of concurrent java programs. In *Proceedings of the International Symposium on Foundations of Software Engineering*, FSE '10, pp. 207–216. ACM.

Huang, J., P. O. Meredith, & G. Rosu (2014). Maximal sound predictive race detection with control flow abstraction. In *Proceedings of the International Conference on Programming*

Language Design and Implementation, PLDI '14, pp. 337–348. ACM.

Huang, J. & L. Rauchwerger (2015). Finding schedule-sensitive branches. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ESEC/FSE '15, pp. 439–449. ACM.

Huang, J. & C. Zhang (2011). An efficient static trace simplification technique for debugging concurrent programs. In *Proceedings of the International Conference on Static Analysis*, SAS '11, pp. 163–179. Springer-Verlag.

Huang, J. & C. Zhang (2012). Lean: Simplifying concurrency bug reproduction via replay-supported execution reduction. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pp. 451–466. ACM.

Huang, J., C. Zhang, & J. Dolby (2013). Clap: Recording local executions to reproduce concurrency failures. In *Proceedings of the International Conference on Programming Language Design and Implementation*, PLDI '13, pp. 141–152. ACM.

Intel Corporation (2013). Intel Processor Trace. https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing. Accessed: 2016-2-25.

Jalbert, N. & K. Sen (2010). A trace simplification technique for effective debugging of concurrent programs. In *Proceedings of the International Symposium on Foundations of Software Engineering*, FSE '10, pp. 57–66. ACM.

Jiang, Y., T. Gu, C. Xu, X. Ma, & J. Lu (2014). Care: Cache guided deterministic replay for concurrent java programs. In *Proceedings of the International Conference on Software Engineering*, ICSE '14, pp. 457–467. ACM.

Jin, G., A. Thakur, B. Liblit, & S. Lu (2010). Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pp. 241–255. ACM.

Jin, G., W. Zhang, D. Deng, B. Liblit, & S. Lu (2012). Automated concurrency-bug fixing. In *Proceedings of the International Conference on Operating Systems Design and Implementation*, OSDI '12, pp. 221–236. USENIX Association.

Jose, M. & R. Majumdar (2011). Cause clue clauses: Error localization using maximum satisfiability. In *Proceedings of the International Conference on Programming Language Design and Implementation*, PLDI '11, pp. 437–446. ACM.

Joshi, A., S. T. King, G. W. Dunlap, & P. M. Chen (2005). Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the International Symposium on Operating Systems Principles*, SOSP '05, pp. 91–104. ACM.

Joshi, P., C.-S. Park, K. Sen, & M. Naik (2009). A randomized dynamic program analysis technique for detecting real deadlocks. In *Proceedings of the International Conference on Programming Language Design and Implementation*, PLDI '09, pp. 110–120. ACM.

Kasikci, B. (2013). Are "data races" and "race condition" actually the same thing in context of concurrent programming. http://stackoverflow.com/questions/11276259/are-data-races-and-race-condition-actually-the-same-thing-in-context-of-conc/. Accessed: 2016-2-1.

Kasikci, B., B. Schubert, C. Pereira, G. Pokam, & G. Candea (2015). Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Proceedings of the International Symposium on Operating Systems Principles*, SOSP '15, pp. 344–360. ACM.

Kasikci, B., C. Zamfir, & G. Candea (2012). Data races vs. data race bugs: Telling the difference with portend. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pp. 185–198. ACM.

Khoshnood, S., M. Kusano, & C. Wang (2015). Concbugassist: Constraint solving for diagnosis and repair of concurrency bugs. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '15, pp. 165–176. ACM.

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

Kramer, J. (2007, April). Is abstraction the key to computing? *Commun. ACM 50*(4), 36–42.

Kusano, M., A. Chattopadhyay, & C. Wang (2015). Dynamic generation of likely invariants for multithreaded programs. In *Proceedings of the International Conference on Software Engineering*, ICSE '15, pp. 835–846. IEEE Press.

Laadan, O., N. Viennot, & J. Nieh (2010). Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proceedings of the International*

Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '10, pp. 155–166. ACM.

Laadan, O., N. Viennot, C.-C. Tsai, C. Blinn, J. Yang, & J. Nieh (2011). Pervasive detection of process races in deployed systems. In *Proceedings of the International Symposium on Operating Systems Principles*, SOSP '11, pp. 353–367. ACM.

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

Lamport, L. (1979, September). How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput. 28*(9), 690–691.

Lampson, B. W. & D. D. Redell (1980, February). Experience with processes and monitors in mesa. *Commun. ACM 23*(2), 105–117.

LeBlanc, T. J. & J. M. Mellor-Crummey (1987, April). Debugging parallel programs with instant replay. *IEEE Trans. Comput. 36*, 471–482.

Lee, D., P. M. Chen, J. Flinn, & S. Narayanasamy (2012). Chimera: Hybrid program analysis for determinism. In *Proceedings of the International Conference on Programming Language Design and Implementation*, PLDI '12, pp. 463–474. ACM.

Leesatapornwongsa, T., J. F. Lukman, S. Lu, & H. S. Gunawi (2016). Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16. ACM.

Leveson, N. & C. Turner (1993). An investigation of the therac-25 accidents. *Computer 26*(7), 18–41.

Liblit, B., A. Aiken, A. X. Zheng, & M. I. Jordan (2003). Bug isolation via remote program sampling. In *Proceedings of the International Conference on Programming Language Design and Implementation*, PLDI '03, pp. 141–154. ACM.

Liu, P., X. Zhang, O. Tripp, & Y. Zheng (2015). Light: Replay via tightly bounded recording. In *Proceedings of the International Conference on Programming Language Design and Implementation*, PLDI 2015, pp. 55–64. ACM.

Lu, S., S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, & Y. Zhou (2007). Muvi: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of International Symposium on Operating Systems Principles*, SOSP '07, pp. 103–116. ACM.

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 International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '08, pp. 329–339. ACM.

Lu, S., J. Tucek, F. Qin, & Y. Zhou (2006). Avio: Detecting atomicity violations via access interleaving invariants. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '06, pp. 37–48. ACM.

Lucia, B. & L. Ceze (2009). Finding concurrency bugs with context-aware communication graphs. In *Proceedings of the International Symposium on Microarchitecture*, MICRO '09, pp. 553–563. ACM.

Lucia, B. & L. Ceze (2013). Cooperative empirical failure avoidance for multithreaded programs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pp. 39–50. ACM.

Lucia, B., L. Ceze, & K. Strauss (2010). Colorsafe: Architectural support for debugging and dynamically avoiding multi-variable atomicity violations. In *Proceedings of the International Symposium on Computer Architecture*, ISCA '10, pp. 222–233. ACM.

Lucia, B., J. Devietti, K. Strauss, & L. Ceze (2008). Atom-aid: Detecting and surviving atomicity violations. In *Proceedings of the International Symposium on Computer Architecture*, ISCA '08, pp. 277–288. IEEE Computer Society.

Lucia, B., B. P. Wood, & L. Ceze (2011). Isolating and understanding concurrency errors using reconstructed execution fragments. In *Proceedings of the International Conference on Programming Language Design and Implementation*, PLDI '11, pp. 378–388. ACM.

Machado, N., B. Lucia, & L. Rodrigues (2015). Concurrency debugging with differential schedule projections. In *Proceedings of the International Conference on Programming Language Design and Implementation*, PLDI '15, pp. 586–595. ACM.

Machado, N., P. Romano, & L. Rodrigues (2013). Property-driven cooperative logging for concurrency bugs replication. In *Proceedings of the International Workshop on Hot Topics* *in Parallelism*, HotPar '13, pp. 1–12. USENIX Association.

Matos, J., J. Garcia, & P. Romano (2015). Enhancing privacy protection in fault replication systems. In *Proceedings of the International Symposium on Software Reliability Engineering*, ISSRE '15, pp. 336–347. IEEE Computer Society.

Montesinos, P., L. Ceze, & J. Torrellas (2008). Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the International Symposium on Computer Architecture*, ISCA '08, pp. 289–300. IEEE Computer Society.

Montesinos, P., M. Hicks, S. T. King, & J. Torrellas (2009). Capo: A software-hardware interface for practical deterministic multiprocessor replay. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '09, pp. 73–84. ACM.

Musuvathi, M. & S. Qadeer (2007a). Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the International Conference on Programming Language Design and Implementation*, PLDI '07, pp. 446–455. ACM.

Musuvathi, M. & S. Qadeer (2007b, January). Partial-order reduction for context-bounded state exploration. Technical Report MSR-TR-2007-12, Microsoft Research.

Musuvathi, M., S. Qadeer, T. Ball, G. Basler, P. A. Nainar, & I. Neamtiu (2008). Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the International Conference on Operating Systems Design and Implementation*, OSDI '08, pp. 267–280. USENIX Association.

Myers, G. J. & C. Sandler (2004). *The Art of Software Testing.* John Wiley & Sons.

Naik, M., A. Aiken, & J. Whaley (2006). Effective static race detection for java. In *Proceedings of the International Conference on Programming Language Design and Implementation*, PLDI '06, pp. 308–319. ACM.

Narayanasamy, S., G. Pokam, & B. Calder (2005). Bugnet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the International Symposium on Computer Architecture*, ISCA '05, pp. 284–295. IEEE Computer Society.

Narayanasamy, S., Z. Wang, J. Tigani, A. Edwards, & B. Calder (2007). Automatically classifying benign and harmful data races using replay analysis. In *Proceedings of the Interna-*

tional Conference on Programming Language Design and Implementation, PLDI '07, pp. 22–31. ACM.

Netzer, R. H. B. (1993). Optimal tracing and replay for debugging shared-memory parallel programs. In *Proceedings of the International Workshop on Parallel and Distributed Debugging*, PADD '93, pp. 1–11. ACM.

NIST (2002, June). Software errors cost us economy $59.5 billion annually. *NIST News Release 2002-10*.

Park, C.-S. & K. Sen (2008). Randomized active atomicity violation detection in concurrent programs. In *Proceedings of the International Symposium on Foundations of Software Engineering*, FSE '08, pp. 135–145. ACM.

Park, S., S. Lu, & Y. Zhou (2009). Ctrigger: Exposing atomicity violation bugs from their hiding places. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '09, pp. 25–36. ACM.

Park, S., R. Vuduc, & M. J. Harrold (2012). A unified approach for localizing non-deadlock concurrency bugs. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, ICST '12, pp. 51–60. IEEE Computer Society.

Park, S., R. W. Vuduc, & M. J. Harrold (2010). Falcon: Fault localization in concurrent programs. In *Proceedings of the International Conference on Software Engineering*, ICSE '10, pp. 245–254. ACM.

Park, S., Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, & S. Lu (2009). Pres: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the International Symposium on Operating Systems Principles*, SOSP '09, pp. 177–192. ACM.

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

Price, C. (1995). MIPS IV Instruction set, revision 3.2. MIPS Technologies.

Qadeer, S. & J. Rehof (2005). Context-bounded model checking of concurrent software. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pp. 93–107. Springer-Verlag.

Regehr, J. (2011). Race condition vs. data race. http://blog.regehr.org/archives/490. Accessed: 2016-2-1.

Sakurity (2016). Hacking Starbucks for unlimited coffee. http://sakurity.com/blog/2015/05/21/starbucks.html. Accessed: 2016-1-18.

Samak, M. & M. K. Ramanathan (2014). Multithreaded test synthesis for deadlock detection. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, pp. 473–489. ACM.

Samak, M. & M. K. Ramanathan (2015). Synthesizing tests for detecting atomicity violations. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ESEC/FSE '15, pp. 131–142. ACM.

Samak, M., M. K. Ramanathan, & S. Jagannathan (2015). Synthesizing racy tests. In *Proceedings of the International Conference on Programming Language Design and Implementation*, PLDI '15, pp. 175–185. ACM.

Sen, K. (2008). Race directed random testing of concurrent programs. In *Proceedings of the International Conference on Programming Language Design and Implementation*, PLDI '08, pp. 11–21. ACM.

Sen, K. & G. Agha (2006a). Automated systematic testing of open distributed programs. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, FASE '06, pp. 339–356. Springer-Verlag.

Sen, K. & G. Agha (2006b). Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *Proceedings of the International Conference on Computer Aided Verification*, CAV '06, pp. 419–423. Springer-Verlag.

Sen, K., D. Marinov, & G. Agha (2005). Cute: A concolic unit testing engine for c. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ESEC/FSE '13, pp. 263–272. ACM.

Sewell, P., S. Sarkar, S. Owens, F. Z. Nardelli, & M. O. Myreen (2010, July). X86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM 53*(7), 89–97.

Sigelman, B. H., L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, & C. Shanbhag (2010). Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc.

Steven, J., P. Chandra, B. Fleck, & A. Podgurski (2000). jRapture: A capture/replay tool for observation-based testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '00, pp. 158–167. ACM.

Tillmann, N. & J. De Halleux (2008). Pex: White box test generation for .net. In *Proceedings of the International Conference on Tests and Proofs*, TAP '08, pp. 134–153. Springer-Verlag.

Tucek, J., S. Lu, C. Huang, S. Xanthos, & Y. Zhou (2007). Triage: Diagnosing production run failures at the user's site. In *Proceedings of the International Symposium on Operating Systems Principles*, SOSP '07, pp. 131–144. ACM.

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

Valmari, A. (1991). Stubborn sets for reduced state space generation. In *Proceedings of the International Conference on Applications and Theory of Petri Nets*, pp. 491–515. Springer-Verlag.

Vaswani, K., M. J. Thazhuthaveetil, & Y. N. Srikant (2005). A programmable hardware path profiler. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pp. 217–228. IEEE Computer Society.

Visser, W., C. S. Păsăreanu, & S. Khurshid (2004). Test input generation with java pathfinder. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '04, pp. 97–107. ACM.

Voung, J. W., R. Jhala, & S. Lerner (2007). Relay: Static race detection on millions of lines of code. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ESEC-FSE '07, pp. 205–214. ACM.

Wang, Y., H. Patil, C. Pereira, G. Lueck, R. Gupta, & I. Neamtiu (2014). Drdebug: Deterministic replay based cyclic debugging with dynamic slicing. In *Proceedings of the International*

Symposium on Code Generation and Optimization, CGO '14, pp. 98:98–98:108. ACM.

Wilson, P. F., L. D. Dell, & G. F. Anderson (1993). *Root Cause Analysis: A Tool for Total Quality Management.* American Society for Quality.

Xu, M., R. Bodik, & M. D. Hill (2003). A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the International Symposium on Computer Architecture*, ISCA '03, pp. 122–135. ACM.

Xu, M., R. Bodík, & M. D. Hill (2005). A serializability violation detector for shared-memory server programs. In *Proceedings of the International Conference on Programming Language Design and Implementation*, PLDI '05, pp. 1–14. ACM.

Yang, J., A. Cui, S. Stolfo, & S. Sethumadhavan (2012). Concurrency attacks. In *Proceedings of the International Conference on Hot Topics in Parallelism*, HotPar'12, pp. 15–15. USENIX Association.

Yang, Z., M. Yang, L. Xu, H. Chen, & B. Zang (2011). Order: Object centric deterministic replay for java. In *Proceedings of the USENIX Annual Technical Conference*, ATC '11, pp. 30–30. USENIX Association.

Yu, J., S. Narayanasamy, C. Pereira, & G. Pokam (2012). Maple: A coverage-driven testing tool for multithreaded programs. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pp. 485–502. ACM.

Yuan, D., H. Mai, W. Xiong, L. Tan, Y. Zhou, & S. Pasupathy (2010). Sherlog: Error diagnosis by connecting clues from run-time logs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pp. 143–154. ACM.

Zamfir, C. & G. Candea (2010). Execution synthesis: A technique for automated software debugging. In *Proceedings of the European Conference on Computer Systems*, EuroSys '10, pp. 321–334. ACM.

Zeller, A. & R. Hildebrandt (2002, February). Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng. 28*(2), 183–200.

Zhang, W., J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, & T. Reps (2011). Conseq: Detecting concurrency bugs through sequential errors. In *Proceedings of the International*

Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '11, pp. 251–264. ACM.

Zhou, J., X. Xiao, & C. Zhang (2012). Stride: Search-based deterministic replay in polynomial time via bounded linkage. In *Proceedings of the International Conference on Software Engineering*, ICSE '12, pp. 892–902. IEEE Press.