

© 2021 Umang Mathur

ALGORITHMIC ADVANCES IN DYNAMIC ANALYSIS FOR DETECTING
CONCURRENCY BUGS

BY

UMANG MATHUR

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Doctoral Committee:

Professor Mahesh Viswanathan, Chair
Professor Madhusudan Parthasarathy
Professor Grigore Rosu
Professor Vivek Sarkar, Georgia Institute of Technology

ABSTRACT

Knock Knock
Race Condition
Who's there?

This joke is a humble attempt to help the readers empathize with the agony of software developers trying to combat complex bugs arising due to concurrency issues regularly arising in their code. Concurrency is an indispensable programming paradigm and multi-threaded programs form the bedrock of most modern software applications. Multi-threaded programs, however, are also the most tricky to get right. Despite rigorous in-house testing, concurrency issues like data races, race conditions, deadlocks and atomicity violations incessantly find their way into production-level software. In the past, errors arising due to complex concurrency bugs in software have led to catastrophic loss of human lives and money. Tackling concurrency bugs, and in particular, efficiently *detecting* such bugs, has, therefore, been the center of attention in computer science research for several decades now.

Dynamic analysis techniques, in particular, have emerged as the de facto standard for detecting concurrency bugs. Such techniques, examine execution traces of programs, with an aim to detect concurrency bugs at runtime. This thesis advances the state-of-the-art in dynamic analysis for detecting concurrency bugs. We propose several algorithms for improving the precision, recall and efficiency of existing techniques for dynamically detecting concurrency bugs like data races and atomicity violations. We also investigate several complexity-theoretic questions establishing precise complexity bounds on several questions arising in dynamic concurrency bug detection.

We first consider the problem of detecting data races dynamically. Most popular techniques for dynamic race detection are either based on a principle of *lockset violations*, or on the happens-before partial order. While these techniques are usually employed at runtime, for detecting data races on-the-fly, there are many scenarios when executions can be, or need to be analyzed for concurrency bugs in an offline setting. Since executions can be extremely large, they are often stored in a compressed format to ease their warehousing. In this thesis, we study the problem of detecting data races when the analysis needs to be performed over an execution that has been compressed using a grammar-based compression scheme. We show how to detect data races efficiently in such a setting, without needing to decompress the (potentially) exponentially succinct compressed format.

We next study the problem of dynamic race *prediction*, which asks if one can infer the

presence of data races beyond those present in a single trace observed by monitoring a program while it is executing. Existing race detectors report false alarms, miss a lot of real races, or do not scale beyond small execution traces. We propose several algorithms that offer a good balance of *scalability* and *predictive power*, while being *sound* (no false positives). We also study the problem from a complexity-theoretic point of view and identify upper and lower bounds, both in the general setting and in settings when the observed execution trace satisfies special properties.

Next, we consider the problem of dynamically detecting atomicity violations. This thesis proposes a linear time vector-clock algorithm for a well-studied notion of atomicity, called *conflict serializability*, for which the only known algorithms ran in cubic time.

The algorithms proposed in this thesis have been implemented and evaluated against large benchmark suites to evaluate their effectiveness. The techniques developed in this thesis are backed by strong theoretical foundations that ensure that our algorithms are scalable, sound and have high predictive power, making them applicable for analyzing modern software systems.

To my parents, for their immense love, sacrifice and unconditional support.

ACKNOWLEDGMENTS

This thesis would not have been possible had it not been for the unconditional love and support of my family, advisor, collaborators and friends, and this piece of text cannot possibly come close to my gratitude for them.

My parents Anita Mathur and Sunil Chandra Mathur have been a constant source of encouragement, love and inspiration throughout my life. I would like to wholeheartedly devote this thesis to my parents, their love and the many sacrifices they have made for this thesis to become a reality. My brother Shreyansh, sister-in-law Aakanksha and my nephew Ashray have been there for me whenever I wanted them. They have made sure to protect me and to cheer me up whenever I felt alone away from home during the course of my Ph.D. My wife Rucha Kulkarni was there for me for the toughest part of my Ph.D. journey, has supported me in all possible ways, and has helped me become a better person.

I would like to thank my Ph.D. advisor Mahesh Viswanathan for believing in me, supporting me and transforming me into a researcher. I have always taken pride in the fact that I was advised by Mahesh, not just because he is an excellent Computer Scientist, but also because he is an excellent human being. Mahesh has given me an incredible amount of freedom — both academic and otherwise. My interactions with Mahesh have always been therapeutic and soothing. A friend of mine once told me — ‘*When I see Mahesh, I know that the rest of my day will be great!*’. The patience he showed during the initial days of my Ph.D., his encouragement and his belief in me have been unusual. His careful attention to detail, clarity of thought and excitement for research have been inspirational, not just for me but for most people around him. I sincerely hope to imbibe and pass on some of his qualities.

I would like to thank Madhusudan Parthasarathy, who has also mentored me during my Ph.D. journey. Madhu’s creativity and passion are energizing, and have also reflected with the research I did with him. His style of undertaking and presenting research has been complementary to Mahesh’s and I am indebted to have collaborated with him. Madhu worked with me for hours together on many interesting research problems. I fondly remember working with him in cafeterias in Urbana and in Lisbon (Portugal), and am amazed at the energy and hard work he puts into everything. I have always looked upto Madhu for advice on how to navigate my career, and am thankful for his invaluable support throughout my Ph.D.

The work presented in this thesis and the ideas they are built upon are a result of my col-

laborations and discussions with Dileep Kini, Andreas Pavlogiannis and Matthew S. Bauer. Dileep has been both a mentor and a friend to me during the initial days of my Ph.D. The work that I did with Dileep has been the starting point of this research, and I am indebted to him for his guidance and hand-holding. Matt has been a dear friend throughout my Ph.D and am grateful for the time we spent together in grad school. The summer of 2017 was when I started considering to continue the line of research presented in this thesis. At that time, Matt helped me enunciate research problems and their potential solutions. Much of the ideas presented here are inspired from my discussions with Matt. Matt has also helped me effectively navigate my Ph.D. journey, giving me advice and listening to me whenever I wanted to talk to someone. I met Andreas Pavlogiannis in January 2019, during the annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL) 2019 conference, and our discussions since then have resulted into some of the most foundational work in the area of concurrency bug detection, some of which are presented in this thesis. Andreas is one of the smartest people I have met, and his unending energy is awe-inspiring. I am extremely thankful to have collaborated with and learnt from him.

During the course of my Ph.D., I have also been fortunate to collaborate with many other people — Chuchu Fan, Sayan Mitra, Ashish Tiwari, Rohit Chadha, Prasad Sistla, Shambwaditya Saha, Adithya Murali, Paul Krogmeier, Vijayaraghavan Murali, Satish Chandra and Rucha Kulkarni. All of these people have positively shaped my career and the way I think. I am indebted to Ashish Tiwari, Jean Whitmore, Vijayaraghavan Murali and Satish Chandra for giving me the opportunity to pursue summer internships with them, thereby helping me take time off from grad school, and to work on exciting problems that arise in an industrial setting. I would also like to thank Grigore Rosu for helping me with many of the resources I required to test my ideas and for allowing me to implement some research techniques proposed in this thesis in his company.

I am fortunate to be friends with many incredible folks, who have been a source of support and guidance away from home. I would like to thank Owolabi Legunsen who gave me invaluable advice when I was looking for academic jobs, relentlessly reviewing my application materials and my job talk. Adithya Murali has been a dear friend and buddy in grad school, and has diligently listened to my ideas, and provided extremely helpful feedback. Adithya was always around when I wanted to chat about anything at all. I would like to thank Vipul Harsh and Vipul Venkataraman who were always there whenever I wanted to talk, especially during the initial years of my Ph.D. I would also like to thank Rucha and Pooja for being around during the later stages of my Ph.D. journey, and for co-parenting our lovely cats Devi and Montu who have undoubtedly made my life happier than ever before.

TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	1
1.1	Predictive Analysis	2
1.2	Linear Time Algorithms and Streaming Setting	3
1.3	Partial Orders	4
1.4	Data Structures	5
1.5	Complexity of Predictive Reasoning	6
1.6	Compression as an Algorithmic Paradigm	6
1.7	Publications	7
1.8	Organization	9
CHAPTER 2	PRELIMINARIES	11
2.1	Concurrent Programs and Semantics	11
2.2	Events and Execution Traces	12
2.3	Correct Reorderings	18
2.4	Predictive Analysis	28
CHAPTER 3	PRIMER ON DYNAMIC DATA RACE DETECTION	33
3.1	Lockset Violations	33
3.2	Happens-Before Based Race Detection	37
3.3	Vector Clock Algorithm for HB	41
3.4	Goldilocks Algorithm for HB	47
3.5	SAT/SMT Based Race Prediction	48
CHAPTER 4	SUMMARY OF CONTRIBUTIONS	51
CHAPTER 5	DATA RACE DETECTION ON COMPRESSED TRACES	54
5.1	Preliminaries	56
5.2	HB-Races in Compressed Traces	59
5.3	LockSet Algorithm for Compressed Traces	75
5.4	Evaluation	83
CHAPTER 6	ENHANCING THE PREDICTIVE POWER OF HAPPENS-BEFORE BASED RACE DETECTION	88
6.1	False Positives Reported by Existing Practical Dynamic Race Detection Tools	91
6.2	Precise Soundness Guarantee of HB	93
6.3	Characterizing Schedulable Races	96
6.4	Algorithm for Detecting HB-Schedulable Races	102
6.5	Experiments	109

CHAPTER 7	SOUND LINEAR TIME DATA RACE PREDICTION USING WEAK CAUSAL PRECEDENCE	117
7.1	Weak Causal Precedence	120
7.2	Vector-Clock Algorithm for WCP	125
7.3	Experimental Evaluation	134
CHAPTER 8	PREDICTING SYNCHRONIZATION PRESERVING DATA RACES IN LINEAR TIME	140
8.1	Synchronization-Preserving Data Races	142
8.2	Detecting Synchronization-Preserving Races	145
8.3	Experiments	162
CHAPTER 9	COMPLEXITY OF DYNAMIC RACE PREDICTION	167
9.1	Notations, Problem Statements and Summary of Results	169
9.2	Trace Ideals	175
9.3	The General Case	179
9.4	Tree Communication Topologies	186
9.5	Witnesses in Small Distance	203
CHAPTER 10	DETECTING CONFLICT SERIALIZABILITY VIOLATIONS USING VECTOR CLOCKS	207
10.1	Preliminaries	209
10.2	Challenges in Designing a Vector Clock Algorithm	212
10.3	Characterizing Conflict Serializability Violations Using Relations on Events.	214
10.4	Vector Clock Algorithm	218
10.5	Experimental Evaluation	230
CHAPTER 11	CONCLUSIONS AND FUTURE DIRECTIONS	235
11.1	State-of-the-Art Race Detector	235
11.2	Predictive Reasoning	236
REFERENCES		239

CHAPTER 1: INTRODUCTION

Software systems are getting increasingly complex and massive, and are affecting more aspects of our lives than ever before. Further, multi-threaded programming has become an indispensable part of modern software applications as shared-memory multi-core processors and affordable cloud resources keep making computations more efficient. However, the process of developing multi-threaded software is inherently challenging. To ensure that these applications work as intended, software developers have to reason about all possible thread interleavings in such programs. However, even for programs with moderate sizes and moderate number of processes, the number of possible thread interleavings gets unmanageably large, making manual reasoning impossible.

The challenge in reasoning about correctness of multi-threaded programs manifests directly in terms of concurrency bugs in these programs. Bugs such as data races and race conditions in programs result often result in unexpected program behaviors including memory crashes [1] due to buffer overflows, null-pointer dereferences, etc., and security flaws (such as use-after-free vulnerabilities), often hampering developer productivity [2], and at times, leading to more serious consequences including the Northeast blackout [3], mismatched NASDAQ share prices [4], and even loss of lives [5]. Other problems such as resource deadlocks lead to poor software performance, denial-of-service and software crashes, thereby, resulting into poor developer and user experience. Traditional software testing methods cannot expose such subtle *heisenbugs*, which can only be uncovered using specific thread schedules.

In this thesis, we propose algorithms and tools for detecting concurrency bugs. Existing techniques for detecting concurrency bugs can broadly be classified into static and dynamic analyses. Static analysis techniques analyze the source code of concurrent programs and infer the presence or absence of bugs [6–12]. On the other hand, dynamic analysis techniques operate at runtime, and infer the presence of concurrency bugs by observing execution traces generated. The algorithms presented here are dynamic analysis algorithms and primarily cater to two classes of concurrency bugs — data races and violations of atomicity specifications, but the underlying ideas go beyond just these classes of bugs.

The technique developed as part of this thesis focus primarily on three objectives — *soundness*, *scalability* and *high recall*. By soundness, we refer to the property that the technique does not report any false positives. Soundness is a crucial property and is fundamental in widespread adoption of a bug detection technique. When a bug detection technique is not sound, it can report false reports and the onus is on the user (developer) to investigate which of the reports indeed correspond to real bugs. This is especially challenging in the

context of concurrency bugs because in order to reason about the correctness of even a single bug report, developers need to reason about all possible interleavings of the threads of the program under inspection. Next, an algorithm can be trivially sound if it does not report any bugs at all! This is why *high recall* is a desirable property. The algorithms proposed in this thesis are developed so that they have high bug detection capability. This is again a challenge for dynamic analysis techniques for analyzing executions of concurrent programs, as observing the exact execution that witnesses a concurrency bug is like looking for a needle in a haystack. Finally, modern software applications such as web browsers, servers, android applications typically generate executions with a large number of events. This means that if the algorithms we develop need to be lightweight and highly efficient in order to be usable in any practical setting.

The algorithmic techniques developed in this thesis are centered around achieving high recall, scalability and soundness. Broadly, this thesis exploits several algorithmic and reasoning principles to achieve these properties. The first principle is *prediction* that allows reasoning about allowable reorderings of observed executions, thereby leading to high recall. However, naively enumerating all possible reorderings is inevitably slow and impractical. This thesis investigates the fundamental *complexity*-theoretic limits of predictive reasoning in the context of data race detection. Further, we propose algorithms for data race prediction that run in *linear time* and work in a *streaming setting*, thereby achieving the holy grail of dynamic analysis algorithms. The backbone of these algorithms is sound *partial orders* and efficient *data structures* like vector clocks. Finally, this thesis proposes *compression* as an algorithmic principle to speed-up dynamic detection of concurrency bugs, in an offline setting. We next discuss each of these aspects in detail.

1.1 PREDICTIVE ANALYSIS

One of the key paradigms this thesis advances is *predictive analysis*. When observing a single execution of a concurrent program, a concurrency bug may not be explicitly visible in such an observed execution — such a bug may appear only in very specific thread schedules. This means techniques that naively look for such visible patterns would likely need to perform analyses multiple times, each time considering a different execution, in the hope of encountering a lucky schedule. Predictive analysis techniques, on the other hand, reason about alternate executions that can be *inferred* from the observed execution, and report the presence of bugs in all these inferred executions as well (instead of only those in the observed one). Predictive analysis is thus key to the high recall of the techniques developed in this thesis.

These alternate executions that one infers in such a predictive style of reasoning, are essentially permutations of the observed execution trace. However, not all permutations make sense and we instead require these permutations additionally satisfy some consistency properties. These consistency properties essentially ensure that the inferred executions can be generated by any program that gave rise to the execution that was observed, thereby allowing program-agnostic reasoning. The key idea in doing so is to formalize properties of concurrent objects such as threads, locks and memory locations, that appear the execution touches. The set of allowable permutations that can be inferred are then those that would respect the semantics of the concurrent objects, and also allow the underlying program to maintain the same control flow as in the original execution. Once the set of allowable reorderings is formalized, one can look for a concurrency bug in any of these allowed reorderings of the observed execution.

1.2 LINEAR TIME ALGORITHMS AND STREAMING SETTING

Prior techniques that incorporate predictive reasoning for detecting concurrency bugs were known to be limited in scalability. Some of these techniques naively enumerate all possible permutations, checking each such permutation for consistency, and for the presence of a concurrency bug in a consistent permutation. Since the number of permutations of a trace with n events can be exponential in n , these techniques do not typically scale beyond traces with a few thousand events, even with optimizations that prune the search space. Other techniques for predictive, resort to symbolic reasoning. Here, one encodes the existence of a consistent permutation witnessing a concurrency bug as a constraint (typically SAT or SMT formula) and invoke a solver to determine the existence of a concurrency bug. While symbolic techniques perform better than techniques that resort to explicit enumeration, they still fall short to scale well for modern software applications which typically generate millions of events.

In this thesis, our focus is on developing techniques for dynamic analysis catered to modern software applications, which typically generate millions of events. We take inspiration from existing race detection techniques that are widely popular and therefore work with the premise that *algorithms whose running time scales super-linearly with trace sizes are unlikely to scale*. In fact, all the techniques for predictive race detection run in linear time. Prior dynamic analysis techniques for detecting concurrency bugs that run in linear time include race detection based on the happens-before (HB) partial order or Eraser’s lockset violation principle (both of which we discuss in detail in Chapter 3) are either unsound or miss a lot of races (low recall). The algorithms we propose here overcome both these problems.

Another useful premise we work with is that practical techniques for dynamic analysis must process traces in a *streaming setting* and perform analysis on-the-fly. There are two advantages of an algorithm that abides by this premise. First, this ensures that bugs are reported as soon as they occur in the trace observed, without waiting to see the entire execution trace. Second, an algorithm working in a streaming setting does not require random access to the entire execution trace. This means that one does not need to store the entire execution trace on disk (or in memory), which could otherwise take up massive amount of space, given that typical software applications can generate hundreds of millions of events.

1.3 PARTIAL ORDERS

Explicit/symbolic enumeration based predictive reasoning is not promising in terms of scalability. In this thesis, we instead resort to reasoning based on partial orders. The basic idea here is to define a partial order on the set of events of an observed execution. The intuition behind defining such a partial order on the events of a trace is that the orderings implied by the partial order capture causality between events. With this intuition, the space of allowable reorderings that one can reason about with such a partial order is essentially those reorderings which also respect the constraints of the partial order, or its linear extensions.

The widely popular approach for detecting data races is, in fact, based on the **HB** partial order. A race is reported whenever there are two conflicting events in the execution that are also unordered by **HB**. As noted before, however, **HB**-based race detection is prone to missing many bugs. This is because **HB** orders events in an execution very conservatively, giving rise to only a few linear extensions, and thus a limited space of allowable reorderings. The conservativeness arises primarily from the fact that **HB** forces an order between critical sections on the same lock even though it may be possible to reorder them and obtain an allowable reordering of the observed execution. Thus, key to overcoming the conservativeness of the **HB** partial order is to disallow such bogus orderings. However, coming up with a correct relaxation of such hard orderings is far from trivial. A key challenge in coming up with a *weakening* of **HB** arises because allowable reorderings implied by the partial order must be well-formed, and must not have overlapping critical sections. Naively disallowing orderings between all critical sections can thus result into an unsound partial order. On the other hand, carefully choosing how to order critical sections can be computationally expensive and thus undesirable. Indeed, prior partial order based approaches for better race prediction than **HB** have largely been either entirely unsuccessful or have resulted into

computationally expensive techniques.

In this thesis, we propose a partial order called *Weak Causal Precedence* (WCP) to address these challenges. WCP, similar to HB is defined as a uniform set of Datalog-like rules for ordering events in an execution. One of the salient aspects of WCP is in how it orders critical sections — critical sections (protected by the same lock) are ordered by WCP whenever they contain conflicting events, or contain events, inductively ordered by WCP. WCP is slightly more subtle than the English description in the previous sentence and is formally defined in Chapter 7. This subtlety arises primarily because the definition is designed to keep both soundness and computational efficiency in mind.

The HB partial order, beyond having limited prediction power, can also lead to false positives, if used incorrectly. More precisely, the soundness guarantee of HB is limited in that it only applies to the first race — pairs of conflicting events unordered by HB may not constitute a real data race. In this thesis, we propose another partial order SHB (*Schedulable Happens-Before*) that addresses this limitation of HB.

1.4 DATA STRUCTURES

A crucial aspect of the partial orders proposed in this thesis is that the ordering between two events is determined solely using events that have occurred in the past (or sometimes a fixed number of events in the future). This property, along with other crucial properties, allow us to design algorithms that enable effective use of *vector clocks* for determining the vector timestamps of events in an execution. A vector timestamp is simply a thread-indexed vector of integers and capture the reachability relation in the partial order exactly — two events are ordered by the partial order iff their vector timestamps are ordered. This is one of the key to obtaining streaming linear time algorithms.

In this thesis, we leverage vector clocks for coming up with linear time algorithms for data race prediction using WCP and SHB. We also propose the use of vector clocks for coming up with a linear time algorithm for the problem of atomicity checking using a popular criterion called *conflict serializability*. Here, one considers a transitive order on the events of a trace, and checks for the existence of cycles that use edges from this order. The only known algorithm for checking conflict serializability ran in time that is cubic in the size of the trace being observed, making it impractical for modern applications. We show that vector clocks can be used to characterize the existence of such cycles in linear time.

1.5 COMPLEXITY OF PREDICTIVE REASONING

In this thesis, we also undertake a formal investigation of fundamental questions about the computational aspects of predictive reasoning in the context of data race detection. For a trace with n events performed by k threads, explicit enumeration based techniques would enumerate all $O(k^n)$ linear extensions that respect the intra-thread ordering, which is an exponential time algorithm. On the other hand, one can non-deterministically guess the sequence of events in an alternate reordering and check if it is consistent and has a data race, showing that the problem of data race prediction is also in the complexity class **NP**. Beyond these two observations, not much was known about the complexity of data race prediction prior to the results in this thesis.

In this thesis, we first show that the problem can be solved in $O(k \cdot n^{2k-2})$ time, which is $O(\text{poly}(n))$ when the number of threads k is constant. We also show that the problem is **NP**-hard in general and in fact is unlikely to be fixed parameter tractable in the parameter k by showing that it is **W**[1]-hard in the parameter k . We also study special classes of the data race detection problem and characterize several tractable cases. While the results in this thesis pertain to data race detection, the techniques proposed can in fact generalize to characterize the complexity of predictive reasoning for many more classes of concurrency bugs.

1.6 COMPRESSION AS AN ALGORITHMIC PARADIGM

Another algorithmic paradigm proposed in this thesis is the use of compression to efficiently check for concurrency bugs from traces. Here, execution traces are compressed using a grammar based compression scheme, such as SLPs (straight-line programs), or LZ77 commonly used in command-line utilities like **gzip** and **tar**. Such compression schemes, exploit repetition in traces and can potentially result into exponentially succinct representations of the corresponding uncompressed traces. A key question then is — can one directly analyze a compressed execution in an efficient manner, without decompressing the trace? Extensive prior work on studying analogous problems on graphs other structures has shown that often tractable properties become intractable when posed on compressed representations.

Surprisingly, this thesis answers this question in the affirmative for data race detection using widely used analyses (race detection using Eraser-style locksets and using the **HB** partial order). We show that race detection can be performed in time that is linear in size of the compressed representation by coming up with algorithms that run in a compositional fashion. Our empirical evaluation of our algorithms show that compression can leverage

repetition in code, giving a speed up of up to $2500\times$, when compared to race detection on uncompressed traces, suggesting that compression can, in fact, be used as an algorithmic principle to speed-up race detection in an offline setting.

1.7 PUBLICATIONS

The material presented in this thesis is based on the following publications.

Dynamic Race Prediction in Linear Time [13]

Dileep Kini, Umang Mathur, Mahesh Viswanathan

In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017).

What Happens-After the First Race? Enhancing the Predictive Power of Happens-Before Based Dynamic Race Detection [14]

Umang Mathur, Dileep Kini, Mahesh Viswanathan

In Proceedings of the ACM Programming Languages 2, OOPSLA (November 2018).

Data Race Detection on Compressed Traces [15]

Dileep Kini, Umang Mathur and Mahesh Viswanathan

In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018).

Atomicity Checking in Linear Time using Vector Clocks [16]

Umang Mathur and Mahesh Viswanathan

In Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2020).

The Complexity of Dynamic Data Race Prediction [17]

Umang Mathur, Andreas Pavlogiannis and Mahesh Viswanathan

In Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2020).

Optimal Prediction of Synchronization-Preserving Races [18]

Umang Mathur, Andreas Pavlogiannis and Mahesh Viswanathan

In Proceedings of the ACM Programming Languages 3, POPL (January 2021).

During my Ph.D. research, I was also fortunate to contribute to the following publications.

Look for the Proof to Find the Program: Decorated-Component-Based Program Synthesis [19]

Adrià Gascón, Ashish Tiwari, Brent Carmer and Umang Mathur

In Proceedings of the 29th International Conference on Computer Aided Verification (CAV 2017).

Exact Quantitative Probabilistic Model Checking through Rational Search [20]

Matthew S. Bauer, Umang Mathur, Rohit Chadha, A. Prasad Sistla and Mahesh Viswanathan

In Proceedings of the 17th Conference on Formal Methods in Computer Aided Design (FMCAD 2017).

Controller Synthesis Made Real: Reach-avoid Specifications and Linear Dynamics [21]

Chuchu Fan, Umang Mathur, Sayan Mitra and Mahesh Viswanathan

In Proceedings of the 30th International Conference on Computer Aided Verification (CAV 2018).

Decidable Fragment of Second Order Logic With Applications to Program Synthesis [22]

P. Madhusudan, Umang Mathur, Shambwaditya Saha and Mahesh Viswanathan

In Proceedings of the 27th EACSL Annual Conference on Computer Science Logic (CSL 2018).

Decidable Verification of Uninterpreted Programs [23]

Umang Mathur, P. Madhusudan and Mahesh Viswanathan

In Proceedings of the ACM Programming Languages 3, POPL (January 2019).

Deciding Memory Safety for Single-Pass Heap-Manipulating Programs [24]

Umang Mathur, Adithya Murali, Paul Krogmeier, P. Madhusudan and Mahesh Viswanathan

In Proceedings of the ACM Programming Languages 3, POPL (January 2020).

Whats Decidable About Program Verification Modulo Axioms? [25]

Umang Mathur, P. Madhusudan and Mahesh Viswanathan

In Proceedings of the 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2020).

Decidable Synthesis of Programs with Uninterpreted Functions [26]

Paul Krogmeier, Umang Mathur, Adithya Murali, P. Madhusudan and Mahesh Viswanathan

In Proceedings of the 32nd International Conference on Computer Aided Verification (CAV 2020).

Exact Quantitative Probabilistic Model Checking through Rational Search [27]

Umang Mathur, Matthew S. Bauer, Rohit Chadha, A. Prasad Sistla and Mahesh Viswanathan

Formal Methods in System Design (FMSD 2020).

Scalable Statistical Root Cause Analysis on App Telemetry [28]

Vijayaraghavan Murali, Edward Yao, Umang Mathur and Satish Chandra

In Proceedings of the 43rd International Conference on Software Engineering, Software Engineering in Practice track (ICSE SEIP, 2021).

Dynamic Data-Race Detection through the Fine-Grained Lens [29]

Rucha Kulkarni, Umang Mathur and Andreas Pavlogiannis

In Proceedings of the 32nd International Conference on Concurrency Theory (CONCUR, 2021), to appear.

Checking LTL[F,G,X] on Compressed Traces in Polynomial Time [30]

Minjian Zhang, Umang Mathur and Mahesh Viswanathan

In Proceedings of the 2021 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021), to appear.

1.8 ORGANIZATION

The organization of this thesis is as follows. In Chapter 2, we discuss the preliminaries, notations and important related work that will be referred to throughout the document. In Chapter 3 we discuss important prior work on race detection. In Chapter 4, we summarize our technical contributions. In Chapter 5, we discuss the problem of data race detection on compressed traces and our algorithms for detecting **HB**-races and violations of lockset discipline. In Chapter 6, we resolve the limitations of the soundness guarantee of **HB** using a new partial order **SHB**. In Chapter 7, we describe data race prediction based on the partial

order WCP. In Chapter 8, we present another data race prediction technique that detects all *synchronization-preserving* data races. In Chapter 9, we present our complexity theoretic results in the context of data race prediction. Chapter 10 presents our linear time algorithm for dynamically detecting conflict serializability violations. We conclude with possible avenues for future work in Chapter 11.

CHAPTER 2: PRELIMINARIES

In this chapter, we will define basic definitions and fix notations that will be useful in the rest of the document.

2.1 CONCURRENT PROGRAMS AND SEMANTICS

We will consider shared-memory concurrent programs that run over a sequentially consistent memory model. In such a setting, a concurrent program can be modeled as having two components. First, there is a *global* or *shared* memory, addressed by shared memory locations. Second, each thread comprises of local memory and a local program. Each local program is deterministic and has the usual constructs of imperative programming like sequencing, assignments, conditionals and loops as well as synchronization constructs such as critical sections for enforcing mutual exclusion. The instructions in the local programs generate events that read from or write to local or shared memory, fork or join other threads or perform other synchronization events. Since the exact syntax of concurrent programs is not relevant for this exposition, we will omit it.

There are several important consequences of the assumptions about the chosen memory model and determinism in programs that will form the outline of our techniques. First, the assumption about sequential consistency implies that all the executions of the concurrent program running over such a memory model can be viewed as if they were sequences of events. In other words, each execution can be viewed as if it was produced by a centralized thread scheduler, which picks a single thread to execute at a time, before preempting it and switching to another thread. Another important aspect about sequential consistency is that an event that reads from a shared memory reads the same value that was written by the last write event on the same memory location. Finally, the assumption about determinism implies that the branches (corresponding to conditional statements) that each thread chooses in a given execution depend only upon the values of the local variables of that thread right before taking entering the conditional. In particular, if the thread encounters a conditional in two different executions but with the same value of the local variables, then the same branch will be taken in both these executions. More importantly, the next instruction that a thread performs is determined solely by the prior instructions performed in the thread.

2.2 EVENTS AND EXECUTION TRACES

The theme of this thesis is to present dynamic analysis algorithms for detecting concurrency bugs, which analyze executions of concurrent programs. With the above assumptions about concurrent programs and their semantics, we can now define execution traces of concurrent programs. We will restrict our attention only to finite executions. Let us fix a set **Threads** of thread identifiers (or simply threads), set **Vars** of variables (or memory locations) and set **Locks** of lock identifiers. An execution trace, or simply trace, over **Threads**, **Vars** and **Locks** is a sequence of events that satisfies some well-formedness properties.

Events. Every event is a tuple of the form $\langle id, t, op \rangle$, where $id \in \mathbb{N}$ is a unique identifier of the event, $t \in \mathbf{Threads}$ is a thread identifier, and op is an operation performed in the event, and can be one the following:

- $\mathbf{r}(x)$, denoting a *read* from some variable $x \in \mathbf{Vars}$
- $\mathbf{w}(x)$, denoting a *write* to some variable $x \in \mathbf{Vars}$
- $\mathbf{acq}(\ell)$, denoting *acquisition* of some lock $\ell \in \mathbf{Locks}$
- $\mathbf{rel}(\ell)$, denoting *release* of some lock $\ell \in \mathbf{Locks}$
- $\mathbf{fork}(u)$, denoting *forking* of a new thread $u \in \mathbf{Threads}$
- $\mathbf{join}(u)$, denoting the *join* of a child thread $u \in \mathbf{Threads}$
- $\mathbf{begin}$, denoting the beginning of a transaction
- $\mathbf{end}$, denoting the end of a transaction

Intuitively, an operation $\mathbf{r}$ (resp. $\mathbf{w}$) is observed when a thread reads the value of (resp. writes a value to) a (local/global) memory location. An event with a $\mathbf{fork}$ (resp. $\mathbf{join}$) operation is generated when a thread creates (resp. joins with) another thread. A critical section in the source concurrent program is observed as a lock acquire ($\mathbf{acq}$) and release ($\mathbf{rel}$) event. Finally, blocks of code of each thread can be annotated to be a single unit, making them transactions (often intended to be run atomically), and $\mathbf{begin}$ and $\mathbf{end}$ operations demarcate boundaries of these transactions in the executions.

We will use **Events** to denote the set of all events that use thread identifiers from the set **Threads**, locks from the set **Locks** and memory locations from **Vars**.

For an event $e = \langle t, id, op \rangle$, we use $\text{thr}(e)$ to denote the thread performing e ($\text{thr}(e) = t$) and $\text{op}(e)$ to denote the operation being performed in e (i.e., $\text{op}(e) = op$). We use $\text{type}(e)$ to

denote the projection of e to its thread and operation components. That is, for $e = \langle id, t, \text{op} \rangle$, we have $\text{type}(e) = \langle t, \text{op} \rangle$. We say e is *of the form* $\langle t, \text{op} \rangle$ if $\text{type}(e) = \langle t, \text{op} \rangle$. For an event e and for $f = \text{type}(e)$, we have $\text{thr}(f) = \text{thr}(e)$ and $\text{op}(f) = \text{op}(e)$. We will often abuse notation and use $\text{type}(e)$ in place of e when the identifier is either clear from the context or not important. For a set E of events, we denote by $\text{Types}(E)$ the set of types of the different events in E , i.e., $\text{Types}(E) = \{\text{type}(e) \mid e \in E\}$.

Execution Traces. An execution trace, or simply trace, is a sequence of events that satisfies certain well-formedness conditions that we outline shortly. We will use $\sigma, \sigma', \sigma_0, \sigma_1, \dots$ to denote traces.

We use Events_σ to denote the set of events in the trace σ . We use Threads_σ to denote the set of threads that appear in σ . That is, $t \in \text{Threads}_\sigma$ iff there is an event $e \in \text{Events}_\sigma$ such that $\text{thr}(e) = t$, $\text{op}(e) = \text{fork}(t)$ or $\text{op}(e) = \text{join}(t)$. Similarly, we use Vars_σ to denote the set of variables that are either written to or are read from in σ , and Locks_σ to denote the set of locks that are acquired or released in σ . For a variable $x \in \text{Vars}_\sigma$, we use $\text{Reads}_\sigma(x)$ to denote the set $\{e \in \text{Events}_\sigma \mid \text{op}(e) = \text{r}(x)\}$, $\text{Writes}_\sigma(x)$ to denote the set $\{e \in \text{Events}_\sigma \mid \text{op}(e) = \text{w}(x)\}$. and $\text{Accesses}_\sigma(x) = \text{Reads}_\sigma(x) \cup \text{Writes}_\sigma(x)$. Further, we let $\text{Reads}_\sigma = \bigcup_{x \in \text{Vars}_\sigma} \text{Reads}_\sigma(x)$, $\text{Writes}_\sigma = \bigcup_{x \in \text{Vars}_\sigma} \text{Writes}_\sigma(x)$ and $\text{Accesses}_\sigma = \bigcup_{x \in \text{Vars}_\sigma} \text{Accesses}_\sigma(x)$. For a lock $\ell \in \text{Locks}_\sigma$, we use $\text{Acquires}_\sigma(\ell)$ to denote the set $\{e \in \text{Events}_\sigma \mid \text{op}(\sigma)(e) = \text{acq}(\ell)\}$ and $\text{Releases}_\sigma(\ell)$ to denote $\{e \in \text{Events}_\sigma \mid \text{op}(\sigma)(e) = \text{rel}(\ell)\}$. Similarly, we use $\text{Acquires}_\sigma = \bigcup_{\ell \in \text{Locks}_\sigma} \text{Acquires}_\sigma(\ell)$ and $\text{Releases}_\sigma = \bigcup_{\ell \in \text{Locks}_\sigma} \text{Releases}_\sigma(\ell)$. Finally, we will denote $\text{Forks}_\sigma = \{e \in \text{Events}_\sigma \mid \exists u, \text{op}(e) = \text{fork}(u)\}$ and $\text{Joins}_\sigma = \{e \in \text{Events}_\sigma \mid \exists u, \text{op}(e) = \text{join}(u)\}$.

We extend the function $\text{type}(\sigma)$ to traces (or sequences of events) in the natural way:

$$\begin{aligned} \text{type}(\varepsilon) &= \varepsilon \\ \text{type}(\sigma \cdot e) &= \text{type}(\sigma) \cdot \text{type}(e) \end{aligned} \tag{2.1}$$

We note that all the above notations are analogously defined for $\text{type}(\sigma)$. For a trace type $\tau = \text{type}(\sigma)$, we have $\text{Threads}_\tau = \text{Threads}_\sigma$, $\text{Vars}_\tau = \text{Vars}_\sigma$ and $\text{Locks}_\tau = \text{Locks}_\sigma$. As with events, we will often abuse σ with $\text{type}(\sigma)$ when the event identifiers are not important.

Let us now enumerate the well-formedness conditions that we require traces to follow. These conditions stem from our assumptions about the semantics of various concurrent objects used in concurrent programs and operations performed using them. In the following, we fix a trace σ .

1. **Forks.** First, we require that for every thread u , there is at most one event $e \in \text{Events}_\sigma$ such that $\text{op}(e) = \text{fork}(u)$. Further, we require that for such an e , we have $\text{thr}(e) \neq u$.

Next, we require that if a thread is forked at an event e , then all events performed by that thread occur after e . That is, if $e = \langle \cdot, \cdot, \text{fork}(u) \rangle \in \text{Events}_\sigma$, then for any event $e' \in \text{Events}_\sigma$ such that $\text{thr}(e') = u$, we must have that e' occurs after e in the sequence σ .

2. **Joins.** We require that there is at most one event $e \in \text{Events}_\sigma$ such that $\text{op}(e) = \text{join}(u)$ and moreover, $\text{thr}(e) \neq u$. Further, if a join event for thread u exists, then we require that the thread has been forked once, by the same thread, prior to the fork event. That is, if there is an event $e \in \text{Events}_\sigma$ with $\text{op}(e) = \text{join}(u)$, then there is an event $e' \in \text{Events}_\sigma$ such that e' occurs before e in the sequence σ , $\text{op}(e) = \text{fork}(u)$ and $\text{thr}(e') = \text{thr}(e)$. Finally, we require that no events are performed by a thread after it has been joined. That is, if $e = \langle \cdot, \cdot, \text{join}(u) \rangle \in \text{Events}_\sigma$, then there is no event $e' \in \text{Events}_\sigma$ such that $\text{thr}(e') = u$ and e' occurs after e in the sequence σ .
3. **Lock Semantics.** We require that critical sections induced by the same lock do not overlap in σ . We formalize this as follows. Let $\{t_1, \dots, t_k\}$ be some enumeration of the threads in Threads_σ . For a lock $\ell \in \text{Locks}_\sigma$, let S_ℓ be the following context-free language (event identifiers are ignored).

$$S_\ell \rightarrow \varepsilon \mid S_{t_1, \ell} \mid \dots \mid S_{t_k, \ell} \mid S_\ell \cdot S_\ell \quad (2.2)$$

where for a thread $t \in \text{Threads}_\sigma$, the context-free language $S_{t, \ell}$ is defined as

$$S_{t, \ell} \rightarrow \varepsilon \mid \langle t, \text{acq}(\ell) \rangle \cdot S_{t, \ell} \cdot \langle t, \text{rel}(\ell) \rangle \quad (2.3)$$

For a lock $\ell \in \text{Locks}_\sigma$, let $\sigma|_\ell$ be the projection of the sequence σ onto the events from $\text{Acquires}_\sigma(\ell) \cup \text{Releases}_\sigma(\ell)$. We require that $\text{type}(\sigma|_\ell)$ is a prefix of some sequence in S_ℓ . For a release event $e = \langle \cdot, t, \text{rel}(\ell) \rangle$, we denote by $\text{match}_\sigma(e)$ as the matching acquire event on the same lock by the same thread. Likewise, for an acquire event e , we denote by $\text{match}_\sigma(e)$ the matching release event corresponding to e in σ if it exists (and $\text{match}_\sigma(e) = \text{undefined}$ otherwise).

4. **Transaction Events.** As with lock events, we require that transaction begin and end events occur as if they were matched, but do not require that transactions from different threads are non overlapping. Formally, for a thread t , let $\sigma|_{t, \text{txn}}$ be the projection of σ to the set of events e with $\text{type}(e) \in \{\langle t, \text{begin} \rangle, \langle t, \text{end} \rangle\}$. Then, we require that for each thread $t \in \text{Threads}_\sigma$, the projection $\sigma|_{t, \text{txn}}$ is a prefix of some word in the following

	t_1	t_2
1	fork (t_2)	
2	acq (ℓ)	
3	acq (m)	
4		begin
5	rel (m)	
6		acq (m)
7	w (x)	
8	rel (ℓ)	
9		r (x)
10		rel (m)
11		end
12	join (t_2)	

Figure 2.1: Example well-formed execution trace σ_1

context-free language:

$$S_{t,\text{txn}} \rightarrow \varepsilon \mid \langle t, \text{begin} \rangle \cdot S_{t,\text{txn}} \cdot \langle t, \text{end} \rangle \mid S_{t,\text{txn}} \cdot S_{t,\text{txn}} \quad (2.4)$$

We will use **Traces** to denote the set of all well-formed traces that use thread identifiers from **Threads**, memory location identifiers from **Vars** and lock identifiers from **Locks**.

Example 2.1. Let us illustrate our notations using an example. Consider Figure 2.1. We use table-like figures to draw execution traces. The trace σ_1 is a sequence of 12 events. The i^{th} row in the table corresponds to the i^{th} event in the trace, which we will denote by e_i . Each column in the figure corresponds to a thread. In trace σ_1 , there are two threads t_1, t_2 , two locks ℓ, m and one shared memory location x . Thus we have $\text{Events}_{\sigma_1} = \{e_1, \dots, e_{12}\}$, $\text{Threads}_{\sigma_1} = \{t_1, t_2\}$, $\text{Locks}_{\sigma_1} = \{\ell, m\}$ and $\text{Vars}_{\sigma_1} = \{x\}$. The event e_1 is performed by thread t_1 and forks thread t_2 , i.e., $\text{thr}(e_1) = t_1$ and $\text{op}(e_1) = \text{fork}(t_2)$. Likewise, $\text{thr}(e_{12}) = t_1$ and $\text{op}(e_{12}) = \text{join}(t_2)$. The second event $e_2 = \langle t_1, \text{acq}(\ell) \rangle$ is performed by thread t_1 and acquires lock ℓ ; since each event e_i is uniquely identified by the subscript i , we did not mention its identifier in the tuple $\langle t_1, \text{acq}(\ell) \rangle$. The matching release of e_2 is $\text{match}_{\sigma_1}(e_2) = e_8$ and thus $\text{match}_{\sigma_1}(e_8) = e_2$. Event $e_7 \in \text{Writes}_{\sigma_1}(x) \subseteq \text{Writes}_{\sigma_1}$ is a write to variable x and event e_9 reads from x . Events e_4 and e_{10} demarcate the begin and end of a transaction in thread t_2 . We remark that the fork event $e_1 = \langle t_1, \text{fork}(t_2) \rangle$ is performed before every event of t_2 and the join event $e_{10} = \langle t_1, \text{join}(t_2) \rangle$ is performed after every event that is performed by t_2 . Next, none of the locks ℓ or m have overlapping critical sections — lock ℓ is only acquired once and lock m is acquired by t_2 (event e_5) after it is released by t_1 (event e_4). Finally, the

	t_1	t_2
1	$\text{acq}(\ell)$	
2	$\text{acq}(m)$	
3	$\text{rel}(\ell)$	
4	$\text{rel}(m)$	
5		$\text{acq}(m)$
6		$\text{acq}(\ell)$
7		$\text{rel}(m)$
8		$\text{acq}(\ell)$
9		$\text{rel}(\ell)$
10		$\text{rel}(\ell)$

Figure 2.2: Well-formed trace σ_2

begin and end events in t_2 are well-matched. Thus, σ_1 is well-formed.

Re-entrant Locking. We remark that our well-formedness conditions on traces allows for re-entrant locking. That is, we allow a lock to be acquired multiple times, by the same thread, before being released. Traces where every lock is acquired exactly once before being released are said to be have non-re-entrant locking. For such a trace σ , $\text{type}(\sigma|_\ell)$ is a prefix of some sequence in the following language (where, as before $\text{Threads}_\sigma = \{t_1, \dots, t_k\}$) is some enumeration of the threads of σ :

$$S_\ell \rightarrow \varepsilon \mid \langle t_1, \text{acq}(\ell) \rangle \cdot \langle t_1, \text{rel}(\ell) \rangle \mid \dots \mid \langle t_k, \text{acq}(\ell) \rangle \cdot \langle t_k, \text{rel}(\ell) \rangle \mid S_\ell \cdot S_\ell \quad (2.5)$$

While none of the results in the thesis require non-re-entrant locking in traces, often we will impose this restriction for ease of presentation.

Well-nested locking. We say that a trace σ obeys well-nested locking or well-nesting for short, if additionally, each thread releases locks in the reverse order of their acquisition. Let us formalize this as follows. For a thread $t \in \text{Threads}_\sigma$, let $\sigma|_t$ be the projection of σ onto the events $\{e \in \text{Events}_\sigma \mid \text{thr}(e) = t\}$. Let $\text{Locks}_\sigma = \{\ell_1, \dots, \ell_m\}$ be some enumeration of the locks in σ . Then, the trace σ obeys well-nesting if $\text{type}(\sigma)$ is a prefix of some word in the following context-free language with starting non-terminal S_t .

$$S_t \rightarrow \varepsilon \mid \langle t, \text{acq}(\ell_1) \rangle \cdot S_t \cdot \langle t, \text{rel}(\ell_1) \rangle \mid \dots \mid \langle t, \text{acq}(\ell_m) \rangle \cdot S_t \cdot \langle t, \text{rel}(\ell_m) \rangle \mid S_t \cdot S_t \quad (2.6)$$

In general, we do not require traces to obey well-nesting and will make it explicit otherwise.

Example 2.2. Consider the trace σ_2 in Figure 2.2 comprising of threads t_1 and t_2 , acquiring

and releasing locks ℓ and m . Let us now observe that this trace observes re-entrant locking. This is because lock ℓ is acquired twice by the thread t_2 , once at event e_6 and then at event e_8 , without being released prior to the later acquire. The matching release events are $\text{match}_{\sigma_2}(e_6) = e_{10}$ and $\text{match}_{\sigma_2}(e_8) = e_9$. Next, we observe that this trace does not obey well-nesting. This is because, the acquire event e_2 is nested in the critical section of e_1 , but its matching release event $e_4 = \text{match}_{\sigma_2}(e_2)$ occurs after the matching release e_3 of e_1 . Similarly, the matching release e_{10} of the acquire event e_6 occurs after the release event e_7 , even though e_6 is sandwiched between the acquire e_5 and its matching release event e_7 . Finally, we remark that this is a well-formed trace — every lock is released as many times as it is acquired before being acquired by another thread.

Orders on traces. This presentation will often refer to several ordering relations imposed on the set of events of a trace. We define some fundamental ones here. Fix a trace σ . The *trace order* of σ is the total order on Events_σ , denoted $\leq_{\text{tr}}^\sigma$ such that for every two events $e_1, e_2 \in \text{Events}_\sigma$, $(e_1, e_2) \in \leq_{\text{tr}}^\sigma$ iff $e_1 = e_2$ or e_1 occurs before e_2 in the sequence σ . We also write $e_1 \leq_{\text{tr}}^\sigma e_2$ for $(e_1, e_2) \in \leq_{\text{tr}}^\sigma$. The order $<_{\text{tr}}^\sigma$ is the *strict total order* imposed by σ , i.e., $(e_1, e_2) \in <_{\text{tr}}^\sigma$ iff $e_1 \neq e_2$ and $e_1 \leq_{\text{tr}}^\sigma e_2$. The *thread order* (also called *intra-thread order*, or, *program order*), denoted $\leq_{\text{TO}}^\sigma$ is the smallest partial order on Events_σ such that $(e_1, e_2) \in \leq_{\text{TO}}^\sigma$ iff $e_1 \leq_{\text{tr}}^\sigma e_2$ and one of the following hold:

- $\text{thr}(e_1) = \text{thr}(e_2)$, or
- $\exists u \in \text{Threads}_\sigma$ such that $\text{op}(e_1) = \text{fork}(u)$ and $\text{thr}(e_2) = u$, or
- $\exists u \in \text{Threads}_\sigma$ such that $\text{thr}(e_1) = u$ and $\text{op}(e_2) = \text{join}(u)$.

Finally, the strict thread order $<_{\text{TO}}^\sigma$ is the order such that $(e_1, e_2) \in <_{\text{TO}}^\sigma$ iff $e_1 \neq e_2$ and $(e_1, e_2) \in \leq_{\text{TO}}^\sigma$. As with $\leq_{\text{tr}}^\sigma$, we write $e_1 <_{\text{tr}}^\sigma e_2$, $e_1 \leq_{\text{TO}}^\sigma e_2$ and $e_1 <_{\text{TO}}^\sigma e_2$ for respectively $(e_1, e_2) \in <_{\text{tr}}^\sigma$, $(e_1, e_2) \in \leq_{\text{TO}}^\sigma$ and $(e_1, e_2) \in <_{\text{TO}}^\sigma$. For an event $e \in \text{Events}_\sigma$, we denote by $\text{pred}_\sigma(e)$ the thread predecessor of e . That is $\text{pred}_\sigma(e)$ is the last event e' in the sequence σ such that $e' <_{\text{TO}}^\sigma e$ (and $\text{pred}_\sigma(e) = \text{undefined}$ if no such event exists). Notice that when $e' = \text{pred}_\sigma(e)$ exists, it is unique — either $\text{op}(e') = \text{fork}(\text{thr}(e))$ when e is the first event that $\text{thr}(e)$ performs, or otherwise e' is the unique event in σ that appears before e in σ and $\text{thr}(e') = \text{thr}(e)$.

Example 2.3. Consider again the trace σ_1 in Figure 2.1. Here, $e_i \leq_{\text{tr}}^\sigma e_j$ for every $1 \leq i \leq j \leq 12$ and $e_i <_{\text{tr}}^\sigma e_j$ for every $1 \leq i < j \leq 12$. Let us now describe the thread order imposed by σ_1 . We have $e_1 \leq_{\text{TO}}^\sigma e_2$, $e_1 \leq_{\text{TO}}^\sigma e_3$ and $e_3 \leq_{\text{TO}}^\sigma e_8$ because all these events belong to the

same thread t_1 . Likewise, we have $e_4 \leq_{\text{TO}}^\sigma e_9 \leq_{\text{TO}}^\sigma e_{11}$ as they belong to the same thread t_2 . Since e_1 forks t_2 , we have $e_1 \leq_{\text{TO}}^\sigma e_4$ and $e_1 \leq_{\text{TO}}^\sigma e_9$. Since e_{12} joins t_2 , we have $e_{10} \leq_{\text{TO}}^\sigma e_{12}$ and $e_4 \leq_{\text{TO}}^\sigma e_{12}$.

Critical Sections and Locks Held. For an acquire event $e \in \text{Acquires}_\sigma$, the critical section defined by e is the set $\text{CS}_\sigma(e) = \{e' \mid \text{thr}(e) = \text{thr}(e'), e \leq_{\text{TO}}^\sigma e' \leq_{\text{TO}}^\sigma \text{match}_\sigma(e)\}$ if $\text{match}_\sigma(e)$ exists; otherwise $\text{CS}_\sigma(e) = \{e' \mid e \leq_{\text{TO}}^\sigma e'\}$. For a release event $e \in \text{Releases}_\sigma$, $\text{CS}_\sigma(e) = \text{CS}_\sigma(\text{match}_\sigma)$. The set of locks held at an event e is the set $\text{locksHeld}_\sigma(e) = \{\ell \in \text{Locks}_\sigma \mid \exists a \in \text{Acquires}_\sigma(\ell) \text{ such that } e \in \text{CS}_\sigma(a)\}$.

Transactions. For a begin event e with type $\text{type}(e) = \langle t, \text{begin} \rangle$, the matching transaction end event is denoted by $\text{match}_\sigma(e)$ (and therefore $\text{type}(\text{match}_\sigma(e)) = \langle t, \text{end} \rangle$); and is undefined if it does not exist. For a begin event e , the transaction that starts at e is the set $\text{txn}_\sigma(e) = \{e' \mid \text{thr}(e) = \text{thr}(e'), e \leq_{\text{TO}}^\sigma e' \leq_{\text{TO}}^\sigma \text{match}_\sigma(e)\}$ if $\text{match}_\sigma(e)$ exists and $\text{txn}_\sigma(e) = \{e' \mid e \leq_{\text{TO}}^\sigma e'\}$ otherwise. Events are singleton transactions if they do not occur within a thread-local section marked by begin and end events. More precisely, an event e with $\text{op}(e) \notin \{\text{begin}, \text{end}\}$, is (part of) a *singleton* transaction if for every $f \in \text{Events}_\sigma$ such that $\text{op}(f) = \text{begin}$ and $\text{thr}(f) = \text{thr}(e)$, if $f \leq_{\text{TO}}^\sigma e$, then $\text{match}_\sigma(f)$ exists and $\text{match}_\sigma(f) \leq_{\text{TO}}^\sigma e$. For a singleton transaction induced by an event e , we say $\text{txn}_\sigma(e) = \{e\}$. Thus, $\text{txn}_\sigma(e)$ is well-defined for every event e in the trace σ . The set of all transactions (singleton or not) in σ is denoted $\text{Transactions}_\sigma$.

Conflicting Events. Given two event types $f_1 = \langle t_1, \text{op}_1 \rangle$ and $f_2 = \langle t_2, \text{op}_2 \rangle$, we say that f_1 and f_2 are conflicting, denoted $f_1 \asymp f_2$ if $t_1 \neq t_2$ there is a variable $x \in \text{Vars}$ such that $(\text{op}_1, \text{op}_2) \in \{(\text{r}(x), \text{w}(x)), (\text{w}(x), \text{w}(x)), (\text{w}(x), \text{r}(x))\}$. Two events $e_1, e_2 \in \text{Events}$ are said to be conflicting, denoted $e_1 \asymp e_2$ if $\text{type}(e_1) \asymp \text{type}(e_2)$.

2.3 CORRECT REORDERINGS

The central theme of this thesis is to design dynamic analysis techniques for detecting concurrency bugs. In this section, we will broadly define what we mean by a bug. Our notion of concurrency bugs, or bugs in general, is intrinsically tied to the setting of dynamic analyses, where one analyzes individual program executions, in isolation from the program source code. In this setting, the most natural choice is to define bugs as a predicate on executions, or equivalently a language of executions that satisfy the predicate. While this is a good first attempt, expressing even common concurrency bugs like deadlocks is not straightforward with the notion of a set of traces. Let us first discuss this notion and then

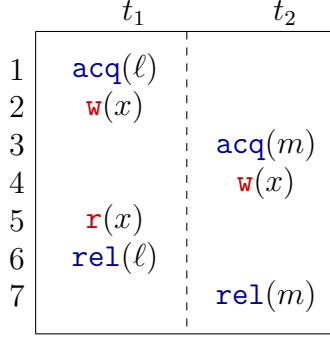


Figure 2.3: Trace σ_3 witnessing an intervening write.

motivate a generalization that captures our intent.

We formalize the notion of bugs as a set of trace types. Let $\text{TraceTypes} = \{\text{type}(\sigma) \mid \sigma \in \text{Traces}\}$ be the set of all types corresponding to well formed traces.

Definition 2.1 (Bug). A bug is a subset $L \subseteq \text{TraceTypes}$. We say that a trace $\sigma \in \text{Traces}$ witnesses a bug of type L if $\text{type}(\sigma) \in L$.

Let us illustrate the concept of a bug with the example below.

Example 2.4. Let $L_{\text{intervening-write}}$ be the following set of trace types:

$$\begin{aligned}
 L_{\text{intervening-write}} = \{ \tau \in \text{TraceTypes} \mid & \exists t_1 \neq t_2 \in \text{Threads}_\tau, \exists x \in \text{Vars}_\tau, \exists i < j < k \\
 & \tau_i = \langle t_1, \mathbf{w}(x) \rangle, \tau_j = \langle t_2, \mathbf{w}(x) \rangle, \tau_k = \langle t_1, \mathbf{r}(x) \rangle \quad (2.7) \\
 & \text{and } \forall i < j' < k, \text{thr}(\tau_{j'}) \neq t_1 \}
 \end{aligned}$$

The above set characterizes all traces (trace types to be precise) where a thread t_1 writes to and successively reads from the same memory location, but there is an intervening write to the same memory location, performed by another thread, that overwrites the value written by the write event performed by t_1 . Often, such a pattern is symptomatic of a problem in the code, as the developer of the code may have ignored the possibility of concurrent threads accessing the same memory location. Consider the trace σ_3 in Figure 2.3. The events e_2 and e_5 are successive writes and reads to the same memory location x in thread t_1 and are intervened by the write event e_4 in thread t_2 . Trace σ_3 thus witnesses a bug of type $L_{\text{intervening-write}}$, or simply an ‘intervening write bug’.

We remark that a witness to a bug is a well-formed trace (more specifically a trace whose type belongs to the language of the bug). While this style of defining bugs (as an arbitrary language) of executions is quite general, we need additional machinery to easily express the

class of concurrency bugs we are interested in. In the following, we outline the challenges that we will overcome using our additional machinery that we introduce shortly.

First, execution traces of concurrent programs are sensitive to thread scheduling. This means when looking for an execution generated by a program that has a specific pattern (such as an intervening write), one is unlikely to find a witness execution. Consider, for instance, the trace σ_3 in Example 2.4. This execution witnesses the ‘intervening write bug’ and was observed because of a specific scheduling of threads, namely obtained by first scheduling 2 operations in thread t_1 , followed by a context switch to thread t_2 , and then back to t_1 after performing 2 events in t_2 . Readers should observe that a program that generates σ_3 can, under a different thread scheduling observe different executions, albeit over the same set of the events of σ_3 . Two of these are shown in Figure 2.4. We remark that any program that generates σ_3 can also generate both of the alternate reorderings (we do not show the $\text{rel}(\ell)$ event in t_1 in these alternate traces, for reasons we will discuss shortly). But, alas, neither of the two executions witness the intervening write bug (i.e., $\{\text{type}(\sigma'_3), \text{type}(\sigma''_3)\} \cap L_{\text{intervening-write}} = \emptyset$). Depending upon the runtime environment, running the underlying program may only generate the traces from Figure 2.4 (instead of σ_3), even after running the program many times. This means that a dynamic analysis technique analyzing executions of the underlying program, without the knowledge of the source program may not be able to deduce the presence of an interleaving write bug, which might go unnoticed and uncover only on specific thread schedules. We will remedy this shortcoming by resorting to bug *prediction*, where the notion of a witness is more elaborate and encompasses traces that may not have been observed.

Second, we need a more expressive way of defining concurrency bugs, where the bug may be inferred only from executions that in fact do not witness the bug, but hint towards the existence a witness. A straightforward example of this is a deadlock. A deadlock happens

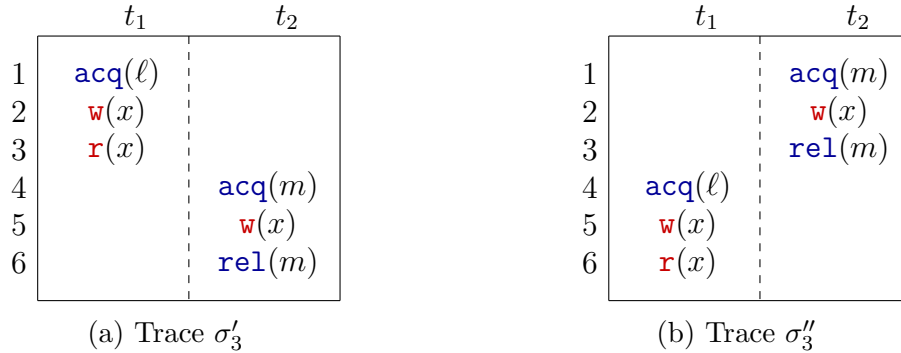


Figure 2.4: Alternate reorderings of trace σ_3 from Figure 2.3

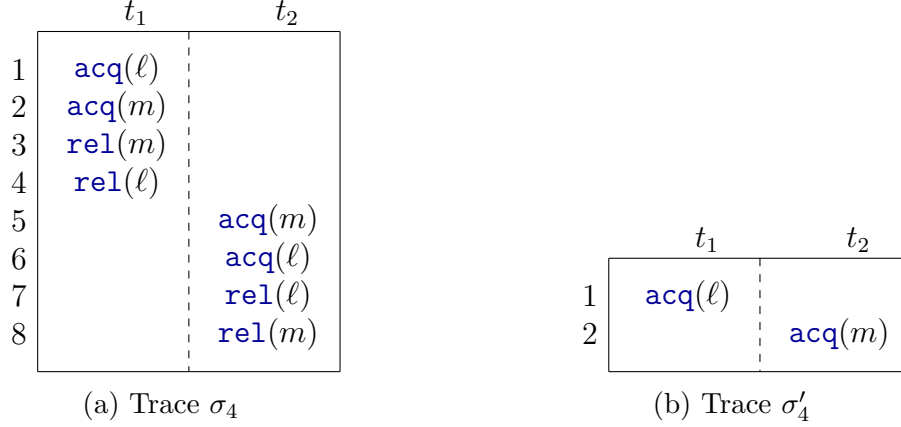


Figure 2.5: Traces witnessing a deadlock

when multiple threads, wait for a resource (locks, in our setting) in a cyclic fashion. Let us consider the trace σ_4 from Figure 2.5a. This trace has a *deadlock* pattern — two different threads (t_1, t_2) acquire two locks (ℓ, m) in a nested fashion, but in the inverse order. However, there is no real deadlock in this trace as at the time of acquisition of lock ℓ in t_2 , the thread t_1 had already released the lock, thereby causing no contention. Thus, this trace is not a witness to a deadlock bug. Now, consider an alternative thread scheduling of the same program that generated the trace σ_4 , where thread t_1 is preempted after it performs the first operation $\text{acq}(\ell)$. Then, the second thread performs $\text{acq}(m)$, as shown in trace σ'_4 from Figure 2.5b. Now when we know that this trace is being generated by the same program that also generated σ_4 , we also know that the next operations that each thread performs is blocked — t_1 cannot execute $\text{acq}(m)$ since it is already acquired by t_2 (and an acquisition of m by t_1 would result into violation of lock semantics), and t_2 cannot acquire ℓ because it is acquired by t_1 . Thus, trace σ'_4 can be deemed a witness of the deadlock bug, given that it is a reordering of trace σ_4 . Independent of the longer trace σ_4 , it is not a witness of a deadlock bug, as the original trace could very well be $\sigma'_{4,\text{extended}} = \sigma'_4 \cdot \langle t_1, \text{rel}(\ell) \rangle \cdot \langle t_2, \text{rel}(m) \rangle$. In the trace $\sigma'_{4,\text{extended}}$, the threads t_1 and t_2 are about to perform release events at the end of the prefix σ'_4 , or, as we say, the events *enabled* in the prefix σ'_4 of trace $\sigma'_{4,\text{extended}}$ are release events, instead of the acquire events when this is a prefix of σ_4 . We will augment the notion of enabled events with the notion of a bug so that we can incorporate the subtlety involved in characterizing such specific classes of bugs.

2.3.1 Defining Correct Reorderings

We noted previously that executions of concurrent programs are sensitive to thread scheduling and looking for a bug by executing a program multiple times is like looking for a needle

in a haystack. This is where *predictive analysis* can help. The basic idea here is that when an execution is observed, it can hint at many other possible executions that the underlying program can give rise to. As an example, consider an execution where the only operations that threads perform are accesses to thread local memory locations. In Figure 2.6, we illustrate such an example execution with 2 threads, with a horizontal alignment.

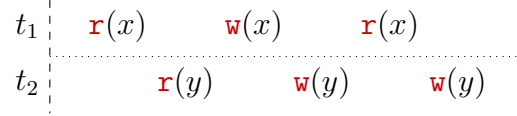


Figure 2.6: Execution with thread-local memory access events.

In this scenario, the underlying program can, under a different thread scheduling, also generate an executions where all the events from a given thread are clubbed together, as in Figure 2.7.

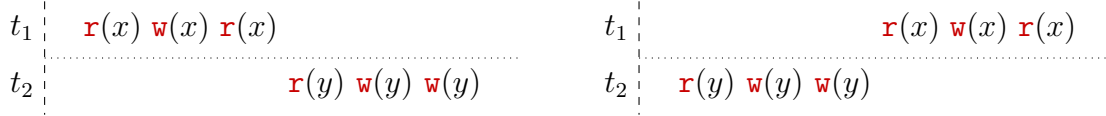


Figure 2.7: Executions obtained in alternate thread scheduling.

The same principle can be extended to infer more executions — arbitrarily permutations of events (that respect the order of events inside a given thread) are also valid executions of any program that generated the first execution. This principle of inferring similar looking executions from a single observed executions is the basis of predictive analysis.

The previous example, however, needs to be taken with a grain of salt. One needs to be careful about arbitrarily permuting the events of an observed executions as not all intra-thread-order-preserving permutations are valid executions. Some of these may not even be well formed. Consider the case when we observe the execution in Figure 2.8.

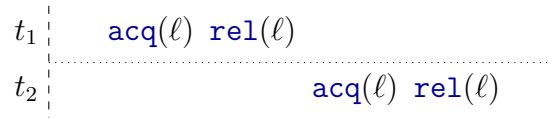


Figure 2.8: Trace with critical sections on lock ℓ , executed in different threads.

There are permutations (that preserve intra-thread-ordering) of this execution that are not even well-formed traces. See Figure 2.9

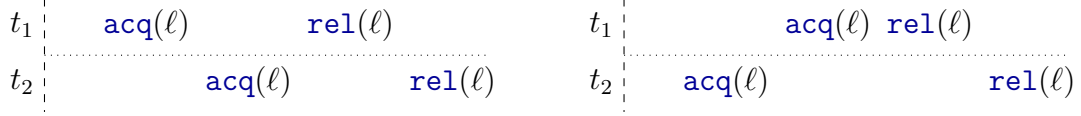


Figure 2.9: Permutations of trace in Figure 2.8 which are not well-formed.

Even worse, in the absence of enough knowledge about the underlying program, some of the permutations (even though well-formed) may not be real executions of the program that generated them. Let us illustrate this via an example.

Example 2.5. Consider the execution trace σ_5 from Figure 2.10a. This execution has two threads (t_1, t_2) accessing two variables (x, y). There are many concurrent programs that can generate this execution, and one such program is P_1 from Figure 2.11a, which generates σ_5 under a scheduler that executes the instructions of t_1 completely before switching context to the thread t_2 . Another program that also generates σ_5 is P_2 from Figure 2.11b. The second statement in P_2 is a conditional whose *then* branch is entered if the value of x read is 42, in which case the value 42 is written to y ; otherwise thread t_1 loops forever.

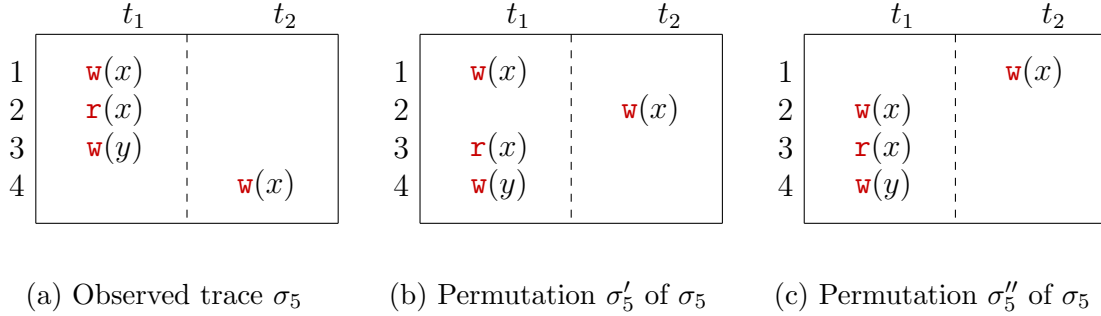


Figure 2.10: Observed trace and its well-formed permutations.

Now, consider the execution σ'_5 from Figure 2.10b. This execution is a permutation of σ_5 , that is both well-formed and also respects the order of events inside each thread. But is this necessarily a valid permutation that can be inferred when we observe the single trace σ_5 ? If the underlying program that generated σ_5 is known to be P_1 (from Figure 2.11a), then indeed inferring σ'_5 as a valid execution would be correct as P_1 can generate σ'_5 (under a scheduler that performs the first instruction ' $x := 42$ ' of t_1 , then switches context to t_2 , and then switches back to t_1 after executing ' $x := 0$ ' in t_2).

However, σ'_5 cannot generate the trace σ'_5 . When the thread scheduler switches context from t_1 to t_2 right after executing the assignment to x in P_2 , the value 0 will be written to x because of the only assignment in t_2 . Consequently, the conditional in t_1 will evaluate to

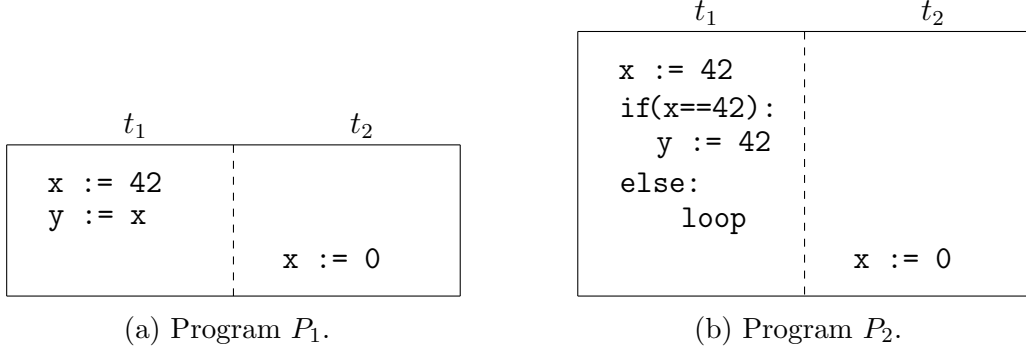


Figure 2.11: Programs P_1 and P_2 generate σ_5 and σ_5'' . σ_5' is generated by P_1 but not by P_2 .

false and the *else* branch will be chosen. Thus, when the underlying program is P_2 , it would be incorrect to infer σ_5' to be a valid permutation of the observed trace σ_5 . Now consider the third trace σ_5'' from Figure 2.10c. Observe that this execution can be generated by both P_1 and P_2 . A crucial property that this execution satisfies is that *every read event in σ_5'' observes the same write event as the corresponding read event in σ_5* . This property ensures that no matter what control flow that σ_5 undergoes, the permuted trace σ_5'' can also undergo. In fact, this ensures that *no matter what program generates σ_5 , the same program can also generate σ_5''* . Thus σ_5'' is a valid inferred permutation of σ_5 . We say that σ_5'' is a *correct reordering* of σ_5 and formalize this notion shortly.

In the previous example, we observed that when trying to infer alternate executions from a given observed execution, we would like to ensure that these alternate executions are feasible, or valid, in the sense that the underlying program that generated the observed execution, should also be able to generate the feasible alternate execution, albeit under a different thread scheduling. In the absence of any knowledge about the underlying program, we, in fact, require that valid alternate executions are those that are generated by any program that generated the observed execution.

How do we characterize the set of valid alternate execution that can be inferred from a given observed execution, in absence of the source program? The idea here is to leverage the semantics of our programming language, including the concurrency constructs. We will refer to the important assumptions that we made about our programs. We assume that programs obey local determinism — the next instruction a thread can execute solely depend upon the values of the local variables. Further, we assume that the programs run under a sequentially consistent memory model. These, together with the semantics of locks (critical sections in different threads on the same lock do not overlap) will be used to characterize a valid alternate execution, or *correct reordering*, that can be inferred from a given execution.

For a trace σ , we use $\text{lastWrite}_\sigma : \text{Reads}_\sigma \hookrightarrow \text{Writes}_\sigma$ to denote the function that maps the read events in σ to the write events they read from. That is, for a read event $e \in \text{Reads}_\sigma$ with $\text{op}(e) = \mathbf{r}(x)$, $\text{lastWrite}_\sigma(e) = e'$ where e' is the last (according to $<_{\text{tr}}^\sigma$) event e' such that $\text{op}(e') = \mathbf{w}(x)$ and $e' <_{\text{tr}}^\sigma e$ if such an e' exists, otherwise $\text{lastWrite}_\sigma(e)$ is undefined. Under sequential consistency, lastWrite_σ captures the write event from which each read gets its value. With this, we can formally define the notion of correct reorderings.

Definition 2.2 (Correct Reordering). Let σ be a trace. A trace ρ is said to be a correct reordering of σ if the following hold:

1. $\text{Events}_\rho \subseteq \text{Events}_\sigma$.
2. Events_ρ is downward closed with respect to $\leq_{\text{TO}}^\sigma$.
3. $\leq_{\text{TO}}^\rho$ is consistent with $\leq_{\text{TO}}^\sigma$. That is, for every $e_1, e_2 \in \text{Events}_\rho$, $e_1 \leq_{\text{TO}}^\rho e_2$ iff $e_1 \leq_{\text{TO}}^\sigma e_2$.
4. lastWrite_ρ respects lastWrite_σ . That is, for a read event $e \in \text{Reads}_\rho$, $\text{lastWrite}_\rho(e) = \text{lastWrite}_\sigma(e)$.

Let us first recall the most important principle behind the above definition:

If a program can generate an execution σ , then it can also generate any correct reordering of σ .

We omit a formal statement for the above principle as that requires formalizing the semantics of the underlying programming language and shared memory system. The above principle is essentially a guideline to define the set of alternate executions that one can conservatively estimate in the absence of any knowledge of the underlying program — in such a scenario, only consider an execution if it can be generated by *all* the programs that can generate the observed execution.

Let us now note some salient aspects of the above definition of correct reordering (Definition 2.2) and discuss their relationship with the above principle. First, the definition requires that the events of a correct reordering ρ are derived from those of the observed reordering σ . This captures our conservativeness about program-agnostic reasoning — since we do not know the underlying program, the best possible estimate of the set of events of a correct reordering is the set of events that we have observed. Second, we require that ρ is both downward-closed and consistent with respect to the thread order of the original trace σ . This means that for every thread t , the projection $\rho|_t$ is a prefix of $\sigma|_t$. This requirement stems from the local determinism property of programs — the ability to perform an instruction is completely determined by the values of the local variables at that time, which

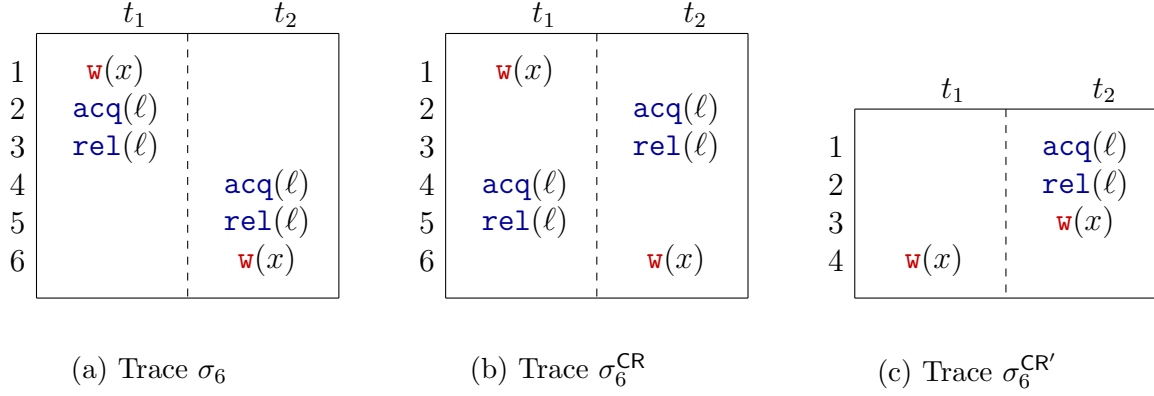


Figure 2.12: Permuting critical sections in correct reorderings.

in turn is completely determined by the sequence of instructions in the thread prior to that. Further, we also emphasize that correct reorderings are allowed to only partially execute a thread, because if a thread can execute a sequence of instructions, then it can also generate any prefix of that sequence. Finally, we require that every read event in ρ sees the same last write event as in σ . This ensures that, inductively, that every control flow that was executed in the original execution σ will be possible even in the correct reordering ρ , thereby ensuring that no matter what program generated σ , that program can also generate ρ .

This style of formalizing alternative executions based on semantics of concurrent objects was popularized by Herlihy and Wing [31, 32] where they used axiomatic semantics to characterize the behavior of concurrent objects. The work of Serbanuta et. al. [33] was amongst the first to propose the idea of feasible traces and maximal causal model for concurrent computations. The notion of correct reordering in Definition 2.2 is strictly subsumed by this notion. Our definition of correct reordering has been derived from [34], and is now widely used in recent literature [13, 14, 17, 35–37].

Let us now illustrate Definition 2.2 through some examples.

Example 2.6. First, let's consider the traces in Figure 2.10. First, observe that σ_5'' is a correct reordering of σ_5 . In fact, σ_5 is also a correct reordering of σ_5'' . This is because in both traces, the intra-thread ordering is the same, and the only read event, 2nd event of t_1 (e_2 in σ_5) has the same last write (first event in t_1) in both traces. However, σ_5' is not a correct reordering of σ_5 because $\text{lastWrite}_{\sigma_5}(e_2) = e_1$ (according to indices in σ_5) but $\text{lastWrite}_{\sigma_5'}(e_3) \neq e_1$ (according to indices in σ_5').

Example 2.7. Let us consider the trace σ_6 in Figure 2.12a. There are two critical sections on the same lock ℓ , one in each thread t_1 and t_2 . First, observe that all prefixes of the sequence σ_6 are correct reorderings of σ_6 (note that they are well-formed traces). Correct reorderings

of this trace that reverse the order of $\langle t_1, \text{rel}(\ell) \rangle$ and $\langle t_2, \text{acq}(\ell) \rangle$ must, in fact, reverse the order of entire critical sections (and thus the order of $\langle t_1, \text{acq}(\ell) \rangle$ and $\langle t_2, \text{rel}(\ell) \rangle$). One such correct reordering is σ_6^{CR} shown in Figure 2.12b. We again note that all prefixes of σ_6^{CR} are also correct reorderings of σ_6 . Next, consider the trace $\sigma_6^{\text{CR}'}$ from Figure 2.12c. This is also a correct reordering of σ_6 , where, the critical section in t_1 is in fact completely absent.

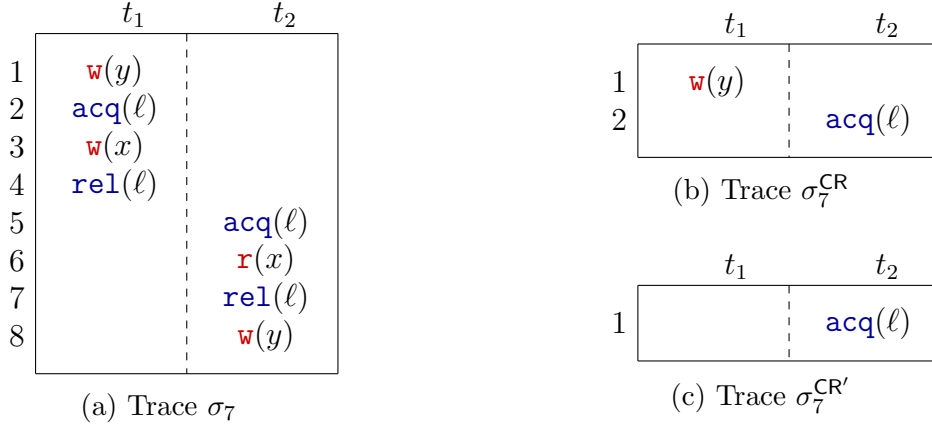


Figure 2.13: Permuting critical sections in correct reorderings.

Example 2.8. Let us now consider the trace σ_7 from Figure 2.13a. Here again, we have two critical sections on the same lock ℓ , one in each of t_1 and t_2 . This time, however, the critical sections are not empty and in fact protect access events on the shared memory location x . In fact, the read event e_6 in the second critical section observes the write e_3 in the first critical section as its last write (i.e., $\text{lastWrite}_{\sigma_7}(e_6) = e_3$). Now consider an arbitrary thread-order-preserving reordering of this trace. If this reordering inverts the order of the two critical sections, it must also invert the order of the events $\langle t_1, w(x) \rangle$ and $\langle t_2, r(x) \rangle$. If this happens, the latter read event will not observe the same last write as the original execution. In other words, any correct reordering that contains both the critical sections cannot invert the order of their appearance. In fact, any reordering that does not schedule the earlier critical section, but schedules the later critical section is also not a correct reordering, because of the same reason. However, it is possible to obtain a correct reordering of σ_7 that does not schedule the read event, but nevertheless schedules the acquire event in t_2 . Both σ_7^{CR} and $\sigma_7^{\text{CR}'}$ exhibit this behavior and are correct reorderings of σ_7 .

2.4 PREDICTIVE ANALYSIS

2.4.1 Predictable Concurrency Bugs

Let us now again consider the problem of detecting concurrency bugs. We had identified two main problems in the vanilla definition (Definition 2.1) of a bug in a dynamic setting. The first problem was the definition did not account for alternate reorderings that could be inferred from a given observed execution, and we defined the notion of correct reorderings (Definition 2.2) to address this. The second challenge was in the inability of Definition 2.1 to express common concurrency bugs like deadlocks where a reference trace is required to guess the enabled set of events in a trace. We address this challenge by defining a new notion of concurrency bug next.

Let us first define the notion of an enabled event. Let σ be a trace and let ρ be a correct reordering of σ (and thus $\mathbf{Events}_\rho \subseteq \mathbf{Events}_\sigma$). An event $e \in \mathbf{Events}_\sigma$ is said to be σ -enabled in ρ if $e \notin \mathbf{Events}_\rho$ and for every event $e' \in \mathbf{Events}_\sigma$, if $e' <_{\text{TO}}^\sigma e$, then $e' \in \mathbf{Events}_\rho$. We will denote by $\mathbf{enabledEvents}_\sigma(\rho)$ the set of all events that are σ -enabled in ρ .

Definition 2.3 (Predictable Concurrency Bug). A concurrency bug L is a subset $L \subseteq \text{TraceTypes} \times \mathcal{P}(\text{Types}(\mathbf{Events}))$. A trace σ is said to have a predictable concurrency bug L if there is a correct reordering ρ of σ and a set of events $E \subseteq \mathbf{enabledEvents}_\sigma(\rho)$ such that $(\text{type}(\rho), \text{type}(E)) \in L$. In this case the correct reordering ρ is said to witness the bug L .

2.4.2 Data Races, Deadlocks and Serializability Violations

Let us now describe three popular concurrency bugs using our formalism.

Data Race. A (predictable) data race captures two concurrent accesses (at least one of which is a write) to the same memory location. The notion of events being *concurrent* is captured by the notion of being simultaneously enabled. Let us formalize the concept of a data race. Now, the concurrency bug $L_{\text{data-race}}$ can be defined as follows.

$$L_{\text{data-race}} = \{(\tau, \{f_1, f_2\}) \mid \tau \in \text{TraceTypes}, f_1 \asymp f_2\} \quad (2.8)$$

In other words, a trace σ is said to have a predictable data race if there are two events $e_1, e_2 \in \mathbf{Events}_\sigma$ such that $e_1 \asymp e_2$ and there is a correct reordering ρ of σ such that both e_1 and e_2 are σ -enabled in ρ . Further, it is often useful to pinpoint the two conflicting events in race — a pair of conflicting events $(e_1, e_2) \in \mathbf{Events}_\sigma^2$ is said to be a predictable race of σ

if there is a correct reordering ρ of σ in which e_1 and e_2 are simultaneously σ -enabled.

Deadlock. A (predictable) deadlock occurs when multiple threads are waiting for resources in a cyclic fashion. In our setting, this manifests when multiple threads are attempting to acquire locks in a cyclic fashion. When talking about deadlocks, we assume that traces obey the well-nesting pattern of locking. Let us formalize the deadlock bug L_{deadlock} here. For a trace type $\tau \in \text{TraceTypes}$, we say that a lock ℓ is held by thread t in τ if there is an unmatched acquire of ℓ by thread t in τ , i.e., $|\{i \mid \tau_i = \langle t, \text{acq}(\ell) \rangle\}| > |\{i \mid \tau_i = \langle t, \text{rel}(\ell) \rangle\}|$.

$$\begin{aligned} L_{\text{deadlock}} = \{(\tau, S) \mid & \tau \in \text{TraceTypes}, \text{ and there are } k \text{ distinct locks } \ell_0, \dots, \ell_{k-1} \\ & \text{and distinct threads } t_0, \dots, t_{k-1} \text{ such that} \\ S = \{ & \langle t_{(i+1)\%k}, \text{acq}(\ell_i) \rangle \}_{0 \leq i \leq k-1} \\ & \text{and } \forall 0 \leq i \leq k-1, \ell_i \text{ is held by } t_i \text{ in } \tau\} \end{aligned} \quad (2.9)$$

Let us discuss the details in the description above. Let us fix a trace σ . For a $k \geq 2 \in \mathbb{N}$, a *deadlock pattern* of length k is a sequence $\langle (e_0, f_0), \dots, (e_{k-1}, f_{k-1}) \rangle$ such that there are k distinct threads $t_0, \dots, t_{k-1} \in \text{Threads}_\sigma$ and distinct locks $\ell_0, \dots, \ell_{k-1} \in \text{Locks}_\sigma$ such that for every $0 \leq i \leq k-1$, the following conditions hold:

- (a) $\text{op}(e_i) = \text{acq}(\ell_i)$,
- (b) $\text{op}(f_i) = \text{acq}(\ell_{(i+1)\%k})$,
- (c) $\text{thr}(e_i) = \text{thr}(f_i)$
- (d) $f_i \in \text{CS}_\sigma(e_i)$

We say that such a deadlock pattern of size k is a predictable deadlock of size k if there is a correct reordering ρ of σ in which each of $f_0, \dots, f_{k-1}$ is σ -enabled.

Atomicity. A trace is called serial if all transactions run uninterrupted in the trace. Let us formalize this. For a trace type $\tau \in \text{TraceTypes}$, let $\tau|_{\text{txn}}$ be the projection of τ to the set of event types $\{\langle t, \text{begin} \rangle, \langle t, \text{end} \rangle \mid t \in \text{Threads}\}$. A trace type τ is *serial* if $\tau|_{\text{txn}}$ is a prefix of some word in the language defined below (start symbol S_{txn}). Here, $\{t_1, \dots, t_k\}$ is some enumeration of Threads_τ .

$$\begin{aligned} S_{\text{txn}} &\rightarrow \varepsilon \mid S_{t_1, \text{txn}} \cdots \mid S_{t_k, \text{txn}} \mid S_{\text{txn}} \cdot S_{\text{txn}} \\ S_{t_i, \text{txn}} &\rightarrow \varepsilon \mid \langle t_i, \text{begin} \rangle \cdot S_{t_i, \text{txn}} \cdot \langle t_i, \text{end} \rangle \end{aligned} \quad (2.10)$$

A trace σ is said to be serial if $\text{type}(\tau)$ is serial. A trace σ is said to be serializable if there is a correct reordering ρ of σ such that $\text{Events}_\rho = \text{Events}_\sigma$ and ρ is serial. The notion of serializable traces generalizes the notion of serial traces — a trace is serializable, if intuitively, it behaves as if it was serial.

While the notion of serializability is pretty generic, a more popular notion is that of conflict serializability [38,39] which has been inspired from the theory of concurrency control in databases [40]. This is defined using the concept of Mazurkiewicz traces [41]. Let us formalize this in our setting. For a trace σ , we say that two events $e_1, e_2 \in \mathbf{Events}_\sigma$ are *dependent* if one of the following hold:

- (a) $\text{thr}(e_1) = \text{thr}(e_2)$, or,
- (b) $f_1 \asymp f_2$, or
- (c) there is a lock $\ell \in \mathbf{Locks}_\sigma$ such that $\{\text{op}(e_1), \text{op}(e_2)\} \subseteq \{\mathbf{acq}(\ell), \mathbf{rel}(\ell)\}$.

A trace ρ is said to be equivalent to σ , if $\mathbf{Events}_\rho = \mathbf{Events}_\sigma$ and for every two dependent events $e_1, e_2 \in \mathbf{Events}_\sigma$, we have $e_1 \leq_{\text{tr}}^\sigma e_2$ iff $e_1 \leq_{\text{tr}}^\rho e_2$. Observe that, in this case, ρ is in fact a correct reordering of σ ; the converse, is not true — not every correct reordering of a trace σ is equivalent to σ . A trace σ is said to be *conflict serializable* if there is an equivalent trace ρ such that ρ is a serial trace. We remark that conflict serializability is a stricter notion than serializability — a trace may not be conflict serializable, but may nevertheless be serializable.

2.4.3 Predictive Analysis: Algorithms, Complexity, Soundness and Predictive Power

Concurrent program executions are extremely sensitive to thread scheduling. Looking for specific patterns by observing executions is like looking for a needle in a haystack. For example, looking for executions where two conflicting events occur *consecutively* in an execution is an ineffective strategy for detecting data races. The notion of correct reorderings and predictable bugs provide a more robust mechanism of defining and detecting bugs. Predictive analysis aims to predict bugs by also analyzing correct reorderings of an observed execution, thereby increasing coverage and making the most out of a single execution. In this thesis, we will be proposing new algorithms for predicting concurrency bugs like data races. In this section, we will discuss how to quantify the effectiveness of these algorithms from the standpoint of their utility in a practical setting. There are several fronts on which one can compare different methods for predictive analysis. We will focus on algorithmic complexity, soundness and maximality and discuss each of these in detail.

Predictive Analysis Algorithms. An algorithm for predicting a specific bug L is a Turing Machine $\mathcal{A}$ that takes a trace σ as input and outputs YES or NO, depending upon its judgement of whether σ has a bug. In this sense, predictive analysis algorithms can be viewed

as acceptors of traces. Often however, we are interested in a more fine-grained output from a bug detector. For example, in the context of race detection, we are interested in pointing out which two events form a data race, and other information about the specifics of the source code (such as program counters) that potentially generates these events. Sometimes, we are interested in reporting more than one data races. This can be specifically useful when developers want to focus on more critical data races that should be fixed. In all such cases, predictive analysis algorithms can be viewed as functions that take a trace as input and return a list of bugs (such as data races).

When talking about complexity of these algorithms, we will often characterize the size of the input (trace) in various ways. Let us list some of the parameters that we will often use in analyzing the complexity of several algorithms. The number of events is the most prominent measure of the size of the trace and we will often use $\mathcal{N}$ to denote it. As we will show later, the complexity of prior work on race detection varies widely in terms of $\mathcal{N}$ — from linear in $\mathcal{N}$ to exponential in $\mathcal{N}$. Often, our algorithms will rely on specific properties of traces (for example, two events of the same thread cannot be enabled in the same trace) and as a result, their complexity will depend upon several parameters such as the number of threads ($\mathcal{T}$), number of memory locations ($\mathcal{V}$) and number of locks ($\mathcal{L}$) in the trace. It is common practice to describe the precise dependence of the complexity of a dynamic analysis algorithm in terms of these parameters.

Complexity. Concurrency bugs such as data races are often symptomatic of more serious problems like memory corruptions and security vulnerabilities. Automated concurrency bug detection is of utmost importance for developers, who would otherwise be chasing these bugs for prolonged periods of times, by manually consider all possible interleavings of the different threads or processes in their software. In modern systems, the number of events that get generated in a single execution can easily be of the order of billions. Thus, the asymptotic complexity of automated bug detection tools is an important factor in the widespread use of a technique.

When we compare different methods for detecting concurrency bugs, we will often compare them based on their asymptotic running time and space requirement, measured in terms of both the size of the input trace $\mathcal{N}$ as well as the parameters $\mathcal{T}$, $\mathcal{V}$ and $\mathcal{L}$. A naive way to predict many of the concurrency bugs we consider, is to enumerate all possible interleavings and check for bugs in each of them. In other words, many of the bug prediction problems are in the class NP, and naive deterministic algorithms for the same problems can have exponential time in the size of the input trace, which can be extremely large. A salient aspect of the algorithms proposed in this thesis run in time that increases only linear in $\mathcal{N}$

(when other parameters like $\mathcal{T}$, $\mathcal{V}$ and $\mathcal{L}$ are constant). Further, these algorithms take either constant or linear space. We also study complexity theoretic problems for race prediction in detail in this thesis.

Soundness. The algorithms proposed in this thesis are designed to be *sound*. An algorithm (that reports YES or NO on a trace) for detecting a class of bugs is sound if its output is NO for every trace σ that does not have a bug. In other words, a sound algorithm for detecting/predicting a bug never reports a false positive. Absence of false positives is important for the widespread adoption of a bug detection tool, as every bug that is reported needs to be carefully examined, thereby making the entire process of bug detection cumbersome. The same principle can be extended to the setting when the algorithm has a more fine grained output (such as a set of races). In this setting, an algorithm is called sound if every bug it reports is a real bug. Specifically, in the case of a data race prediction, an algorithm $\mathcal{A}$ is said to be sound if for every pair of conflicting events (e_1, e_2) in the input trace σ it reports, there is a correct reordering ρ of σ in which both e_1 and e_2 are σ -enabled. The same applies to the case of deadlock prediction — an algorithm is sound for deadlock prediction if for every deadlock pattern it reports, there is a correct reordering in which the deadlock pattern is enabled. One of the algorithms we propose for data race prediction in this thesis has a *weak soundness* guarantee — whenever the algorithm reports YES, then the input trace either has a predictable data race or a predictable deadlock.

Predictive Power and Maximality. Since soundness can be trivially achieved by an algorithm that never reports a bug, a desirable property we are interested in is high predictive power, or high recall. Algorithms with higher predictive power report a YES answer on more number of traces. More concretely, an algorithm $\mathcal{A}_1$ has a higher predictive power than an algorithm $\mathcal{A}_2$ if for every input trace σ , if $\mathcal{A}_2$ outputs YES, then $\mathcal{A}_1$ outputs YES. As with soundness, there is a natural extension of the notion of predictive power when algorithms output bug reports. In this case, we say that $\mathcal{A}_1$ has a higher predictive power than $\mathcal{A}_2$ if for every trace σ , the set of bugs reported by $\mathcal{A}_2$ is a subset of those reported by $\mathcal{A}_1$. Finally, an algorithm for concurrency bug prediction is said to be maximal (or complete) if it reports all the bugs in a trace (or it reports every trace σ that has a bug, when the output is in a YES/NO answer).

CHAPTER 3: PRIMER ON DYNAMIC DATA RACE DETECTION

Data race detection has been the subject of more than four decades of research. While both static and dynamic analysis techniques have been proposed in the literature, we will focus on dynamic ones here as they are more popular and more widely adopted than static techniques, at least at the time of writing this thesis. In this section, we discuss some of the most popular dynamic analyses for detecting data races. These techniques either solely focus on scalability or solely on high predictive power. Many of the techniques proposed in this thesis build upon these techniques but balance both scalability and predictive power.

3.1 LOCKSET VIOLATIONS

The lockset principle is based on the observation that two conflicting events cannot be in race if they are protected by a common lock. Formally,

Proposition 3.1. Let σ be a trace and let ρ be a correct reordering of σ . Let $e_1, e_2 \in \text{Events}_\sigma$ be two distinct events that are both σ -enabled in ρ . Then, $\text{locksHeld}_\sigma(e_1) \cap \text{locksHeld}_\sigma(e_2) = \emptyset$ and $\text{thr}(e_1) \neq \text{thr}(e_2)$.

Proof. Let $e_1, e_2 \in \text{Events}_\sigma$ be two distinct events that are both σ -enabled in ρ .

First observe that for a σ -enabled event e in ρ , we must have $e' \in \text{Events}_\rho$ for all $e' \in \text{Events}_\sigma$ with $e' <_{\text{TO}}^\sigma e$. Now assume on the contrary that $\text{thr}(e_1) = \text{thr}(e_2)$. This means either $e_1 <_{\text{TO}}^\sigma e_2$ or $e_2 <_{\text{TO}}^\sigma e_1$, implying that $\{e_1, e_2\} \cap \text{Events}_\rho \neq \emptyset$. This contradicts that both e_1 and e_2 are σ -enabled in ρ .

Let us now observe that for a σ -enabled event e in ρ , if $\ell \in \text{locksHeld}_\sigma(e)$, then there is an acquire a with $\text{thr}(a) = \text{thr}(e)$, $\text{op}(a) = \text{acq}(\ell)$ and $\text{match}_\sigma(a) \notin \text{Events}_\rho$. Now assume on the contrary that $\text{locksHeld}_\sigma(e_1) \cap \text{locksHeld}_\sigma(e_2) \neq \emptyset$. Let ℓ be one lock in $\text{locksHeld}_\sigma(e_1) \cap \text{locksHeld}_\sigma(e_2)$. We observed that there acquires a_1, a_2 with $\text{op}(a_1) = \text{op}(a_2) = \text{acq}(\ell)$ and $\text{thr}(a_1) \neq \text{thr}(a_2)$. This violates lock semantics in ρ , contradicting that ρ is a well formed trace. QED.

Given a trace σ , this principle can be used to check for the *absence* of data races — if all conflicting pairs of events are protected by a common lock, then there are no predictable races in σ . Consider, for example, trace σ_1 in Figure 3.1a. The only pair of conflicting events here is (e_2, e_5) , but they cannot be simultaneously enabled in any correct reordering ρ as they are protected by a common lock. Thus, there is no predictable data race in σ_1 .

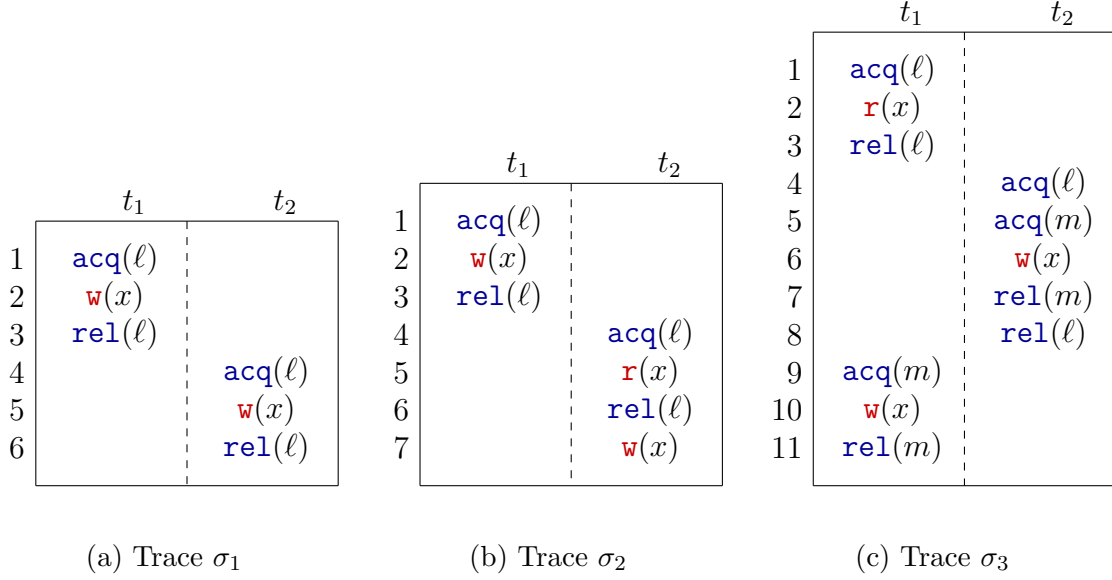


Figure 3.1: Illustrating the lockset principle.

Savage et. al. [42] proposed a technique for race detection that takes this principle a step further and is based on the following observation. In the following, we say that there is a predictable race on a variable x if there are conflicting events $e_1, e_2 \in \text{Accesses}_\sigma(x)$ such that (e_1, e_2) is a predictable race of σ .

Lemma 3.1. Let σ be a trace and let $x \in \text{Vars}_\sigma$ be a shared memory location. There is no predictable race on x in σ if any of the following hold:

1. All accesses on x are read accesses. That is, $\text{Accesses}_\sigma(x) = \text{Reads}_\sigma(x)$.
2. All accesses on x are performed by a single thread. That is, $\exists t \in \text{Threads}_\sigma, \forall e \in \text{Accesses}_\sigma, \text{thr}(e) = t$.
3. All accesses to x are protected by a common lock: $\bigcap_{e \in \text{Accesses}_\sigma(x)} \text{locksHeld}_\sigma(x) \neq \emptyset$.

Proof. We prove the contrapositive. Let (e_1, e_2) be a predictable data race on x (i.e., $e_1, e_2 \in \text{Accesses}_\sigma(x)$). Since $e_1 \asymp e_2$, we must have $w(x) \in \{\text{op}(e_1), \text{op}(e_2)\}$. Further, by Proposition 3.1, $\text{thr}(e_1) \neq \text{thr}(e_2)$ and thus there are at least two threads that access x . Finally, by Proposition 3.1, $\text{locksHeld}_\sigma(e_1) \cap \text{locksHeld}_\sigma(e_2) = \emptyset$, and thus $\bigcap_{e \in \text{Accesses}_\sigma(x)} \text{locksHeld}_\sigma(x) = \emptyset$. QED.

Let us now describe the workings of race detection algorithm employed by the ERASER tool [42] based on the lockset principle. Let $\mathcal{D} = \{\Lambda\} \cup \{\Lambda_t \mid t \text{ is a thread}\}$ is a ‘dummy’ set

Algorithm 3.1: ERASER's algorithm for detecting violations of lockset principle

```

1 function Initialization()
2   foreach  $t \in \text{Threads}$  do
3      $\text{AcquiredLocks}_t := \emptyset$ 
4   foreach  $x \in \text{Vars}$  do
5      $\text{LSet}_x := \top_{\text{All-Locks}}$ 
6 handler acquire( $t, \ell$ )
7    $\text{AcquiredLocks}_t := \text{AcquiredLocks}_t[\ell \mapsto \text{AcquiredLocks}_t(\ell) + 1]$ 
8 handler release( $t, \ell$ )
9    $\text{AcquiredLocks}_t := \text{AcquiredLocks}_t[\ell \mapsto \text{AcquiredLocks}_t(\ell) - 1]$ 
10 handler read( $t, x$ )
11    $\text{LSet}_x := \text{LSet}_x \cap (\{\Lambda, \Lambda_t\} \cup \{\ell \mid \text{AcquiredLocks}_t(\ell) > 0\})$ 
12   if  $\text{LSet}_x \cdot \text{isEmpty}()$  then
13     declare 'race on  $x$ '
14 handler write( $t, x$ )
15    $\text{LSet}_x := \text{LSet}_x \cap (\{\Lambda_t\} \cup \{\ell \mid \text{AcquiredLocks}_t(\ell) > 0\})$ 
16   if  $\text{LSet}_x \cdot \text{isEmpty}()$  then
17     declare 'race on  $x$ '

```

of locks such that $\mathcal{D} \cap \text{Locks} = \emptyset$. Let us define the lockset of an access event $e \in \text{Accesses}_\sigma$ as follows:

$$\text{lockSet}_\sigma(e) = \begin{cases} \{\Lambda, \Lambda_t\} \cup \text{locksHeld}_\sigma(e) & \text{if } \text{op}(a) = \mathbf{r}(x), \text{ for some } x \\ \{\Lambda_t\} \cup \text{locksHeld}_\sigma(e) & \text{if } \text{op}(a) = \mathbf{w}(x), \text{ for some } x \end{cases} \quad (3.1)$$

The lockset of a variable $x \in \text{Vars}_\sigma$ is then given as follows.

$$\text{lockSet}_\sigma(x) = \bigcap_{e \in \text{Accesses}_\sigma} \text{lockSet}_\sigma(e) \quad (3.2)$$

We note that $\Lambda \in \text{lockSet}_\sigma(x)$ if every access of x is a read access. Similarly, $\Lambda_t \in \text{lockSet}_\sigma(x)$ if every access of x is performed by a single thread t . Thus, $\text{lockSet}_\sigma(x)$ is non-empty only if there are atleast two write access to x , by different threads and at least one of them is a write access.

We present an outline of the ERASER's race detection in Algorithm 3.1. The algorithm runs in a streaming fashion, processing each event in the order it occurs in the trace, by updating data structures maintained in its state. The algorithm maintains the following

data structures:

- (a) **LSet_x**: For every variable x , **LSet_x** tracks the set **lockSet_σ(x)** for the trace $σ$ seen so far.
- (b) **AcquiredLocks**: For every thread t , **AcquiredLocks_t** is the set of locks that are held in thread t . More specifically, **LSet_x : Locks → ℕ** maps each lock to the number of times lock $ℓ$ has been acquired without being released in the trace seen so far.

The algorithm initializes **AcquiredLocks_t** to be the empty map, and **LSet_x** to the universal set $\top_{\text{All-Locks}}$ which is such that for any set of locks S , $S \cap \top_{\text{All-Locks}} = S$, and $S \cup \top_{\text{All-Locks}} = \top_{\text{All-Locks}}$. Also, $\top_{\text{All-Locks}} \cap \top_{\text{All-Locks}} = \top_{\text{All-Locks}}$ and $\top_{\text{All-Locks}} \cup \top_{\text{All-Locks}} = \top_{\text{All-Locks}}$.

Algorithm 3.1 outlines how the state is updated at an event of each type. The algorithm does not update its state at forks, joins, transaction begin or end events, and we therefore omit them from the description. At a read (resp. write) event of type $\langle t, \text{r}(x) \rangle$ (resp. $\langle t, \text{w}(x) \rangle$), the handler **read**(t, x) (resp. **write**(t, x)) is invoked. This handler updates **LSet_x** by intersecting it with the locks held at the event being processed and flags a race if this set becomes empty. Likewise the handler **acquire**(t, ℓ) (resp. **release**(t, ℓ)) is invoked when an event of type $\langle t, \text{acq}(\ell) \rangle$ (resp. $\langle t, \text{rel}(\ell) \rangle$) is processed. This handler updates the locks acquired by thread t appropriately.

Let us now see the correctness and the resulting guarantees of this algorithm.

Theorem 3.1 (Correctness of Algorithm 3.1). Given input trace $σ$, Algorithm 3.1 reports a race iff there is a variable $x \in \text{Vars}_σ$ such that **Writes_σ(x)** $\neq \emptyset$, at least two threads access x and $\bigcap_{e \in \text{Accesses}_σ(x)} \text{locksHeld}_σ(x) = \emptyset$.

Proof (Sketch). The proof relies on the invariant maintained by the algorithm which we state below. For a variable x and trace prefix π seen so far, let S_x^π be the value of **LSet_x** computed by Algorithm 3.1 after processing the events in π . Then we have

$$S_x^\pi = \begin{cases} \top_{\text{All-Locks}} & \text{if } x \notin \text{Vars}_\pi \\ \text{lockSet}_\sigma(x) & \text{otherwise.} \end{cases} \quad (3.3)$$

QED.

Corollary 3.1 (Unsoundness and Completeness). Algorithm 3.1 is complete — if a trace $σ$ has a predictable race on variables $x \in \text{Vars}_σ$, then Algorithm 3.1 reports a race on x . Further, Algorithm 3.1 is unsound — there is a trace $σ$ which has no predictable race but Algorithm 3.1 reports a race on $σ$.

The completeness guarantee follows from Theorem 3.1 and Lemma 3.1. Unsoundness is also pretty straightforward. Traces σ_2 and σ_3 from Figure 3.1 illustrate the unsoundness of the lockset principle. In trace σ_2 , we have $\text{locksHeld}_{\sigma_2}(e_2) \cap \text{locksHeld}_{\sigma_2}(e_7) = \emptyset$ and thus Algorithm 3.1 reports a race on x . However, there is no predictable race as the conflicting pair (e_2, e_5) is protected by a common lock and every correct rerordering ρ in which e_6 is enabled cannot have e_2 enabled because $e_2 = \text{lastWrite}_{\sigma_2}(e_5)$. In trace σ_3 , Algorithm 3.1 reports a race as LSet_x successively gets updated to $\{\Lambda_{t_1}, \ell\}$, $\{\ell\}$ and $\emptyset$. However, as we can see, every pair of conflicting events is protected by a common lock.

The unsoundness of ERASER’s lockset algorithm limits its applicability as a standalone race detection tool. However, this algorithm is a lightweight tool to report *potential* races and can be used to filter out race reports, which can then be subjected to a more heavyweight analysis, as in SAT/SMT based techniques [43, 44] and in ‘hybrid’ race detection approaches which combine lockset analysis with a happens-before analysis [45, 46]. The popularity of ERASER largely stems from its lightweight implementation. We state its time and space complexity in the following.

Theorem 3.2. Let σ be a trace having $\mathcal{N}$ events, $\mathcal{T}$ threads, $\mathcal{V}$ variables and $\mathcal{L}$ locks. Let r be the maximum number of times a lock is acquired in some thread in σ without being released. Then, Algorithm 3.1 runs in time $O(\mathcal{N} \cdot (\mathcal{L} + \log r))$ and utilizes $O(\mathcal{T} \cdot \mathcal{L} \cdot \log r + \mathcal{V} \cdot \mathcal{L})$ space.

Proof. The algorithm stores, for every thread t and lock ℓ , the number of times ℓ is acquired by t without being released, and for every variable x a set of locks ($\text{lockSet}(x)$). This gives a total space of $O(\mathcal{T} \cdot \mathcal{L} \cdot \log r + \mathcal{V} \cdot \mathcal{L})$.

Let us now analyze the time complexity. Time spent at an acquire and release event is $O(\log r)$ for arithmetic operations. Time spent at a read or write event is $O(\mathcal{L})$. This gives us the desired time complexity. QED.

3.2 HAPPENS-BEFORE BASED RACE DETECTION

Happens-Before based partial order is one of the most common approaches for dynamic data race detection, and is popular because of its soundness guarantee which we will state in this section. The happens-before (HB) partial order was proposed by Lamport [47] with applications to distributed computing. It was subsequently adapted for detecting data races [48, 49]. Let us first describe the partial order.

Definition 3.1. Let σ be a trace. The *happens-before* partial order on the events of σ , denoted $\leq_{\text{HB}}^\sigma$, is the smallest partial order such that

(a) $\leq_{\text{TO}}^\sigma \subseteq \leq_{\text{HB}}^\sigma$, and

(b) for every lock ℓ and events $e, e' \in \text{Events}_\sigma$ such that $e \leq_{\text{tr}}^\sigma e'$, $\text{op}(e) = \text{rel}(\ell)$ and $\text{op}(e') = \text{acq}(\ell)$, then we have $(e, e') \leq_{\text{HB}}^\sigma$.

The happens-before partial order, thus orders events in the same thread according to the thread order of the trace (Rule (a)) and orders critical sections (on the same lock) in the order in which they appear in the trace (Rule (b)). We remark that Definition 3.1 ensures that $\leq_{\text{HB}}^\sigma \subseteq \leq_{\text{tr}}^\sigma$, i.e., HB orders events in the same order as the order imposed by $\leq_{\text{tr}}^\sigma$. We will use $<_{\text{HB}}^\sigma$ to denote the order $\leq_{\text{HB}}^\sigma \setminus \text{Events}_\sigma^2$. As before, we will write $e \leq_{\text{HB}}^\sigma e'$ (resp. $e <_{\text{HB}}^\sigma e'$) to denote $(e, e') \in \leq_{\text{HB}}^\sigma$ (resp. $(e, e') \in <_{\text{HB}}^\sigma$). Further, we will write $e \parallel_{\text{HB}}^\sigma e'$ if neither $e \leq_{\text{HB}}^\sigma e'$, nor $e' \leq_{\text{HB}}^\sigma e$.

Let us first characterize the order defined in Definition 3.1 using an inductive formulation below.

Proposition 3.2. Let σ be a trace. The strict partial order $<_{\text{HB}}^\sigma$ is the transitive closure of the following relation

$$\begin{aligned} R_\sigma = & \{(e, f) \mid e <_{\text{tr}}^\sigma f, \text{thr}(e) = \text{thr}(f)\} \\ & \cup \{(e, f) \mid e <_{\text{tr}}^\sigma f, \exists u \in \text{Threads}_\sigma, \text{op}(e) = \text{fork}(u), \text{thr}(f) = u\} \\ & \cup \{(e, f) \mid e <_{\text{tr}}^\sigma f, \exists u \in \text{Threads}_\sigma, \text{thr}(e) = u, \text{thr}(f) = \text{join}(u)\} \\ & \cup \{(e, f) \mid e <_{\text{tr}}^\sigma f, \exists \ell \in \text{Locks}_\sigma, \text{op}(e) = \text{rel}(\ell) \text{ and } \text{op}(f) = \text{acq}(\ell)\} \end{aligned}$$

Proof. Observe that $R_\sigma = <_{\text{TO}}^\sigma \cup L_\sigma$, where $L_\sigma = \{(e, f) \mid e <_{\text{tr}}^\sigma f, \exists \ell \in \text{Locks}_\sigma, \text{op}(e) = \text{rel}(\ell) \text{ and } \text{op}(f) = \text{acq}(\ell)\}$. Based on Definition 3.1, we must have $R_\sigma \subseteq \leq_{\text{HB}}^\sigma$. Also note that R_σ is irreflexive, and since R_σ is a subset of the strict total order $<_{\text{tr}}^\sigma$, its transitive closure must also satisfy $R_\sigma^+ \subseteq <_{\text{tr}}^\sigma$, implying that R_σ^+ is irreflexive. With these observations, we can prove our desired result.

Let us first prove that $<_{\text{HB}}^\sigma \subseteq R_\sigma^+$. Assume on the contrary that $<_{\text{HB}}^\sigma \setminus R_\sigma^+ \neq \emptyset$. Consider $R' = \leq_{\text{HB}}^\sigma \cap R_\sigma^+$, where id_σ is the identity binary relation on Events_σ . Observe that $R_\sigma \subseteq R'$, R' is a partial order, and $R' \subseteq \leq_{\text{HB}}^\sigma$. This violates the property that $\leq_{\text{HB}}^\sigma$ is the smallest property that contains R_σ .

We now prove $R_\sigma^+ \subseteq <_{\text{HB}}^\sigma$. Consider a pair $(e, f) \in R_\sigma^+$. From the definition of transitive closure, there must be a sequence of events $e = g_1, g_2, \dots, g_k = f$ such that for each $1 \leq i < k$, $(g_i, g_{i+1}) \in R_\sigma$. This means for each $1 \leq i < k$, $(g_i, g_{i+1}) \in \leq_{\text{HB}}^\sigma$, and because $\leq_{\text{HB}}^\sigma$, we have $(e, f) \in <_{\text{HB}}^\sigma$. QED.

Let us now illustrate the HB partial order with the following example.

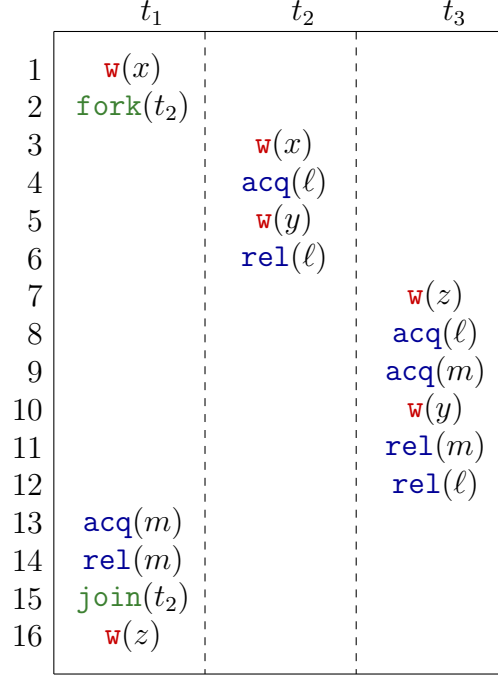


Figure 3.2: Trace σ_4 illustrating the happens-before partial order.

Example 3.1. Consider the trace σ_4 in Figure 3.2. Due to Rule (a), we have $e_1 \leq_{HB}^{\sigma_4} e_2 \leq_{HB}^{\sigma_4} e_6 \leq_{HB}^{\sigma_4} e_{15} \leq_{HB}^{\sigma_4} e_{16}$. Due to Rule (b), we have $e_6 \leq_{HB}^{\sigma_4} e_8$ as they are release and acquire events of critical sections on the same lock ℓ , and by transitivity, $e_1 \leq_{HB}^{\sigma_4} e_{12}$. Similarly, $e_{11} \leq_{HB}^{\sigma_4} e_{13}$ and thus $e_7 \leq_{HB}^{\sigma_4} e_{16}$. Further, e_3 and e_7 are not ordered by $\leq_{HB}^{\sigma_4}$, i.e., $e_3 \parallel_{HB}^{\sigma_4} e_7$. Similarly, we have $e_{12} \parallel_{HB}^{\sigma_4} e_{16}$.

One can define the HB partial order in an alternate manner, as follows.

Definition 3.2. Let σ be a trace and let $e, f \in \text{Events}_\sigma$ be two events. We say that there is an HB-path in σ from e to f if either $e = f$, or otherwise there is a sequence of events in Events_σ $g_1, g_2, \dots, g_k$ ($k > 1$) such that $e = g_1$, $f = g_k$, and for every $1 \leq i < k$, $g_i <_{tr}^\sigma g_{i+1}$, and one of the following holds:

- (a) $thr(g_i) = thr(g_{i+1})$, or
- (b) $op(g_i) = \text{rel}(\ell)$ and $op(g_{i+1}) = \text{acq}(\ell)$ for some lock $\ell \in \text{Locks}_\sigma$, or
- (c) $op(g_i) = \text{fork}(u)$ and $thr(g_{i+1}) = u$, for some thread $u \in \text{Threads}_\sigma$, or
- (d) $thr(g_i) = u$ and $op(g_{i+1}) = \text{join}(u)$, for some thread $u \in \text{Threads}_\sigma$.

The two definitions of happens-before partial order are equivalent:

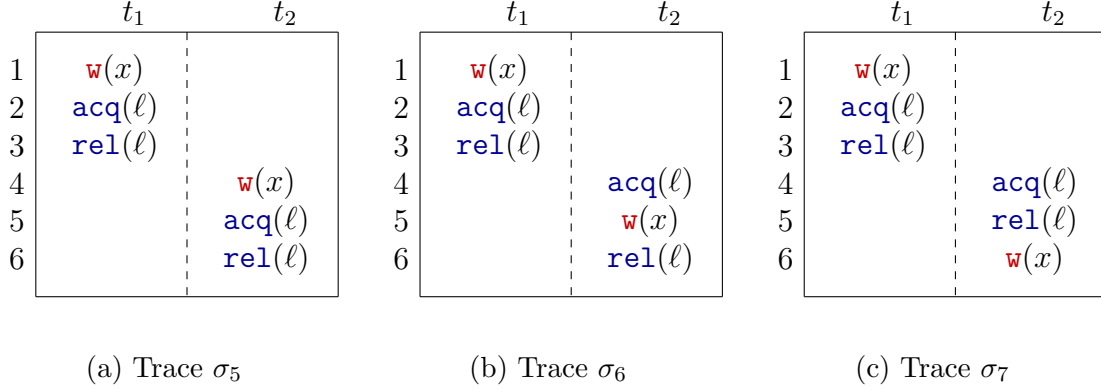


Figure 3.3: HB race detection: Soundness and Incompleteness

Lemma 3.2. Let σ be a trace and let $e, f \in \text{Events}_\sigma$ be two events. There is an HB-path in σ from e to f iff $e \leq_{\text{HB}}^\sigma f$.

Proof (Sketch). Follows directly from Proposition 3.2

QED.

Data race detection based on HB identifies pairs of conflicting events that are unordered by the HB partial order. For a trace σ and events $e, e' \in \text{Accesses}_\sigma$ such that $e \succ e'$, we say that (e, e') is an HB-race if $e \parallel_{\text{HB}}^\sigma e'$ and say that σ has an HB-race if there is such a pair of events in σ . We sometimes say that there is an HB-race between events e and e' to denote that (e, e') is an HB-race in the trace.

Theorem 3.3 (Soundness and predictive power of HB). Let σ be a trace. If σ has an HB-race, then σ has a predictable data race. Further, there is a trace which has a predictable data race but no HB-race.

Proof (Sketch). The incompleteness is illustrated in Example 3.2 and here we give a gist of the soundness guarantee. In the following proof we assume all pairs (e_1, e_2) are ordered in trace order as $e_1 \leq_{\text{tr}} e_2$.

Let σ be a trace with an HB-race. Consider the first such race (e_1, e_2) . That is, $e_1 \succ e_2$, $e_1 \parallel_{\text{HB}}^\sigma e_2$ and for every pair of conflicting events (e'_1, e'_2) in σ such that $e'_1 \parallel_{\text{HB}}^\sigma e'_2$, we have either $e_2 <_{\text{tr}}^\sigma e'_2$ or $e_2 = e'_2$ and $e'_1 <_{\text{tr}}^\sigma e_1$. Consider the set $S = \{e \mid e <_{\text{HB}}^\sigma e_1 \text{ or } e <_{\text{HB}}^\sigma e_2\}$. We first remark that $\{e_1, e_2\} \cap S = \emptyset$. Now consider the sequence ρ obtained by arranging the events in S as per $\leq_{\text{tr}}^\sigma$. Now, we note that ρ is a well formed trace. Further, it is a correct reordering of σ because (e_1, e_2) is the first HB race in σ . Finally, both e_1 and e_2 are σ -enabled in ρ .

QED.

Let us illustrate race-detection based on HB using examples.

Example 3.2. Consider the trace σ_5 in Figure 3.3a. Here, we have an HB-race (e_1, e_4) (i.e., $e_1 \parallel_{\text{HB}}^{\sigma_5} e_4$) and thus by Theorem 3.3, σ_5 has a predictable data race. Indeed, the empty trace ε is a correct reordering of σ_5 and both e_1 and e_4 are σ_5 -enabled in it. Next, consider the trace σ_6 in Figure 3.3b. Here, $e_1 \leq_{\text{HB}}^{\sigma_6} e_5$ and thus there is no HB-race. However, there is a predictable data race (e_1, e_5) because the trace $\sigma_6^{\text{CR}} = e_4$ is a correct reordering in which both e_1 and e_5 are enabled. Finally, in trace σ_7 from Figure 3.3c, there is no HB-race but the correct reordering σ_7^{CR} exposes the data race (e_1, e_6) .

The traces in the previous example shows that HB misses very simple races. This is primarily because HB conservatively orders critical sections on the same lock, intuitively implying that they must be causally related. This, however, is not true as a correct reordering of a trace may invert the order of critical sections. We next discuss two different algorithms for detecting data races based on HB (Section 3.3 and Section 3.4).

3.3 VECTOR CLOCK ALGORITHM FOR HB

The first algorithm for detecting data races based on the HB partial order is a *vector clock algorithm*, inspired from vector clock algorithms in the distributed systems literature [50, 51]. The basic idea here is to associate with every event, a timestamp such that the ordering on timestamps mimics the HB order. We will be describing race detection algorithm that were proposed by [49] with subsequent optimization proposed by [48]. We also be discussing a recently proposed optimization on the basic algorithm [52]. Some of the popular tools that employ the vector clock algorithm for race detection include THREADSANITIZER [53, 54] Helgrind [55] and Intel Inspector [56].

3.3.1 Vector Timestamps and Vector Clocks

Vector clock algorithms assign vector timestamps to events observed in a trace. Vector clocks find use not just in HB based race detection, but we also employ them in other algorithms we have proposed as part of this thesis.

Let us use **Threads** to denote the set of threads in the trace (we omit the subscript denoting the trace). A vector time (or timestamp) is a vector of non-negative integers, whose size/dimension is $\mathcal{T}$ (number of threads). For a thread $t \in \text{Threads}$, we denote the t^{th} component of a vector time V by $V(t)$. Alternatively, a vector timestamp can be viewed as a map $V : \text{Threads} \rightarrow \mathbb{N}$. We say a vector time V_1 is *less than* (or *ordered before* or simply *before*) another time V_2 (of the same dimension), denoted $V_1 \sqsubseteq V_2$ if $\forall t \in \text{Threads}. V_1(t) \leq V_2(t)$. In

this case, we say that V_2 is *greater than, ordered after* or *after* V_1 . The minimum vector time on threads **Threads** is $\perp_{\text{Threads}} = \lambda t.0$, and we will often use $\perp$ when **Threads** is clear from context. Next, the *join* of two vector times V_1 and V_2 is the time $V_1 \sqcup V_2 = \lambda t. \max(V_1(t), V_2(t))$. Finally, we use $V[t \mapsto c]$ to denote the timestamp $\lambda u.$ if $u = t$ then c else $V(u)$. Let us summarize these operations below.

$$\begin{array}{llll}
V_1 \sqsubseteq V_2 & \iff & \forall t. V_1(t) \leq V_2(t) & \textbf{(Point-wise Comparison)} \\
V_1 \sqcup V_2 & = & \lambda t. \max(V_1(t), V_2(t)) & \textbf{(Join)} \\
V[t \mapsto c] & = & \lambda u. \text{ if } (u = t) \text{ then } c \text{ else } V(u) & \textbf{(Update)} \\
\perp & = & \lambda t. 0 & \textbf{(Bottom)}
\end{array} \tag{3.4}$$

Vector clocks are variables (or place holders) for vector timestamps. That is, vector clocks are variables that take values from the space of vector times, and will be used in our algorithm to compute the timestamps associated with various events in a trace. All the operations on vector times can be naturally thought of as applying to vector clocks as well. It is customary practice to use bold faced letters such as $\mathbb{C}, \mathbb{L}, \dots$ for vector clocks and normal face letters $C, L, \dots$ for vector timestamps.

We will primarily use vector timestamps to mimic a given partial order. That is, the partial order imposed by the timestamps (ordered by $\sqsubseteq$) we compute will be isomorphic to the partial order we wish to mimic. For this, we will use the timestamps of earlier events to compute those of later events. Most of our algorithms are however streaming in nature and have a small memory footprint. We ensure this by not storing the timestamps of all events, but only of a fixed number of events at any given time, and store these in a fixed number of vector clocks. The algorithms typically keep changing the set of events which we track, by updating the vector clocks they maintain.

3.3.2 DJIT⁺ algorithm for HB race detection

Let us now discuss the details of the vector clock algorithm DJIT⁺ [48] for detecting data races using the happens-before order. A pseudocode for this algorithm is presented in Algorithm 3.2. We describe it next.

Vector Clocks. The algorithm uses several vector clocks, each having dimension **Threads**. It stores a vector clock $\mathbb{C}_t$ for every thread t in the trace, a vector clock $\mathbb{L}_\ell$ for every lock ℓ and two vector clocks ($\mathbb{R}_x$ and $\mathbb{W}_x$) for every memory location x in the trace. The clock $\mathbb{C}_t$ is initialized with $\perp[t \mapsto 1]$ for each thread t ; all other vector clocks are initialized to $\perp$.

Overview. The algorithm runs in a streaming fashion, processing the events of a trace in

Algorithm 3.2: DJIT⁺ vector clock algorithm for detecting an HB-race.

<pre> 1 function Initialization() 2 foreach $t \in \text{Threads}$ do 3 $\mathbb{C}_t := \perp[t \mapsto 1]$ 4 foreach $\ell \in \text{Locks}$ do 5 $\mathbb{L}_\ell := \perp$ 6 foreach $x \in \text{Vars}$ do 7 $\mathbb{R}_x := \perp$ 8 $\mathbb{W}_x := \perp$ 9 handler acquire(t, ℓ) 10 $\mathbb{C}_t := \mathbb{C}_t \sqcup \mathbb{L}_\ell$ 11 handler release(t, ℓ) 12 $\mathbb{L}_\ell := \mathbb{C}_t$ 13 $\mathbb{C}_t := \mathbb{C}_t[t \mapsto \mathbb{C}_t(t)+1]$ </pre>	<pre> 14 handler fork(t, u) 15 $\mathbb{C}_u := \mathbb{C}_t[u \mapsto 1]$ 16 $\mathbb{C}_t := \mathbb{C}_t[t \mapsto \mathbb{C}_t(t)+1]$ 17 handler join(t, u) 18 $\mathbb{C}_t := \mathbb{C}_t \sqcup \mathbb{C}_u$ 19 handler read(t, x) 20 if $\mathbb{W}_x \not\sqsubseteq \mathbb{C}_t$ then 21 declare ‘race’ and exit 22 $\mathbb{R}_x := \mathbb{R}_x \sqcup \mathbb{C}_t$ 23 handler write(t, x) 24 if $\mathbb{W}_x \not\sqsubseteq \mathbb{C}_t$ or $\mathbb{R}_x \not\sqsubseteq \mathbb{C}_t$ then 25 declare ‘race’ and exit 26 $\mathbb{W}_x := \mathbb{W}_x \sqcup \mathbb{C}_t$ </pre>
---	--

the order in which they appear in the trace. Depending upon the type of the event that is seen, the algorithm calls the appropriate handler and updates different vector clocks. The algorithm performs two objectives simultaneously. First, it computes the HB partial order, i.e., it computes the timestamps of each event so that these timestamps reflect the HB partial order on events. Of course, the timestamps of not all events are stored. Second, it uses the timestamps of some of these events (summarized in vector clocks) to check for HB-races.

Timestamps. The algorithm associates each event e in the trace, with a vector timestamp C_e^σ . We will often be writing C_e instead of C_e^σ when the trace σ is clear from the context. Let us next precisely state the value of C_e computed by Algorithm 3.2 for an event e of the trace σ .

$$C_e = \lambda t. \begin{cases} 1 + |S_e^t| & \text{if } \text{thr}(e) = t \\ |S_e^t| & \text{otherwise} \end{cases} \quad (3.5)$$

where, for an event e and thread u , the set S_e^u is defined as follows:

$$S_e^u = \{e' <_{\text{HB}}^\sigma e \mid \text{thr}(e') = u, e' \in \text{Releases}_\sigma \cup \text{Forks}_\sigma\} \cup \{e' \leq_{\text{HB}}^\sigma e \mid \text{op}(e') = \text{join}(u)\} \quad (3.6)$$

Lemma 3.3 states that the above timestamps accurately mimic the HB partial order.

Lemma 3.3. Let σ be a trace and let $e, e' \in \text{Events}_\sigma$ such that $\text{thr}(e) \neq \text{thr}(e')$. We have, $C_e \sqsubseteq C_{e'} \iff e \leq_{\text{HB}}^\sigma e'$

Proof. ($\Rightarrow$) Let e, e' be events such that $e \leq_{\text{HB}}^{\sigma} e'$ and $\text{thr}(e) \neq \text{thr}(e')$. Also let $t = \text{thr}(e)$. Now, let u be an arbitrary thread different from t and let $f \in \text{Events}_{\sigma}$ $\text{thr}(f) = u$. If $f <_{\text{HB}}^{\sigma} e$, then clearly $f <_{\text{HB}}^{\sigma} e'$ as $<_{\text{HB}}^{\sigma}$ is transitive. Thus, $S_e^u \subseteq S_{e'}^u$ and thus $C_e(u) = |S_e^u| \leq |S_{e'}^u| = C_{e'}(u)$. We now argue $C_e(t) \leq C_{e'}(t)$. Let $e = f_1, f_2, \dots, f_k = e'$ be the HB-path (Definition 3.2) from e to e' . Observe that for all $1 \leq i \leq k$, $e \leq_{\text{HB}}^{\sigma} f_i \leq_{\text{HB}}^{\sigma} e'$. Since $\text{thr}(e) \neq \text{thr}(e')$, there must be at least one $i < k$ such that $\text{thr}(f_i) = t$ and $\text{thr}(f_{i+1}) \neq t$. Let i^* be the earliest such i . If $\text{op}(f_{i^*+1}) = \text{join}(t)$, then $f_{i^*+1} \in S_{e'}^t$ and $f_{i^*+1} \notin S_e^t$ and thus $|S_e^t| + 1 \leq |S_{e'}^t|$ giving us $C_e(t) \leq C_{e'}(t)$. Otherwise $\text{thr}(f_{i^*}) = t$ and $\text{op}(f_{i^*}) \in \{\text{rel}(\ell), \text{fork}(u) \mid \ell \in \text{Locks}_{\sigma}, u \in \text{Threads}_{\sigma}\}$. Thus, $f_{i^*} \in S_{e'}^t$ and $f_{i^*} \notin S_e^t$ (as $e \leq_{\text{tr}}^{\sigma} f_{i^*}$). This gives us $C_e(t) = |S_e^t| + 1 \leq |S_{e'}^t| = C_{e'}(t)$.

($\Leftarrow$) Let e, e' be events such that $\text{thr}(e) \neq \text{thr}(e')$ and $C_e \sqsubseteq C_{e'}$. Also let $t = \text{thr}(e)$. We have $C_e(t) \leq C_{e'}(t)$ and thus $|S_e^t| + 1 \leq |S_{e'}^t|$. Thus, there is an event $f \in S_{e'}^t \setminus S_e^t$. There are two cases here. In the first case $\text{op}(f) = \text{join}(t)$. In this case, $e <_{\text{HB}}^{\sigma} f$, and $f \leq_{\text{HB}}^{\sigma} e'$ giving us our desired result. In the second case $\text{thr}(f) = t$ and since $f \notin S_e^t$, we must have $e <_{\text{TO}}^{\sigma} f$. This gives us $e <_{\text{HB}}^{\sigma} f <_{\text{HB}}^{\sigma} e'$ giving us our desired result. QED.

Computing Timetamps. The algorithm uses timestamps of earlier events to compute timestamps of later events. However, instead of storing the timestamps of all events, it stores the timestamps of only a few *frontier* events and uses these to compute later timestamps. These frontier timestamps are stored in vector clocks $\{\mathbb{C}_t\}_{t \in \text{Threads}}$ and $\{\mathbb{L}_{\ell}\}_{\ell \in \text{Locks}}$.

Let us describe what the values of these clocks are at a given point in time. Let us suppose that the prefix of the trace processed by the algorithm until some point is σ . For a thread $t \in \text{Threads}_{\sigma}$, let e_t^{σ} be the last event in σ with $\text{thr}(e_t^{\sigma}) = t$. Similarly, for a lock $\ell \in \text{Locks}_{\sigma}$, let e_{ℓ}^{σ} be the last event in σ with $\text{op}(e_{\ell}^{\sigma}) = \text{rel}(\ell)$. The following lemma then describes the invariant maintained by the algorithm.

Lemma 3.4. Let σ be a trace. For a thread t and lock ℓ , let T_t^{σ} and L_{ℓ}^{σ} be the values of the clock $\mathbb{C}_t$ and $\mathbb{L}_{\ell}$ after Algorithm 3.2 processes σ . Then,

$$T_t^{\sigma} = \begin{cases} \perp[t \mapsto 1] & \text{if } e_t^{\sigma} \text{ does not exist} \\ C_{e_t^{\sigma}}[t \mapsto C_{e_t^{\sigma}}(t) + 1] & \text{if } e_t^{\sigma} \in \text{Releases}_{\sigma} \cup \text{Forks}_{\sigma} \\ C_{e_t^{\sigma}} & \text{otherwise} \end{cases}$$

$$L_{\ell}^{\sigma} = \begin{cases} \perp & \text{if } e_{\ell}^{\sigma} \text{ does not exist} \\ C_{e_{\ell}^{\sigma}} & \text{otherwise} \end{cases}$$

Proof (Sketch). The statement can be proved by induction on the length of the trace. In

the base case, the length of the trace is 0 and the algorithm initializes $\mathbb{C}_t = \perp[t \mapsto 1]$ (and all other clocks to $\perp$).

In the inductive case we see an event e after reading the prefix trace ρ (and thus $\sigma = \rho \circ e$). If e is a read or write event, the algorithm correctly does not change the clock values. If $\text{op}(e) = \text{acq}(\ell)$, then for any thread u , we have $S_e^u = S_{e_t^\rho}^u \cup S_{e_\ell^\rho}^u$, which can be used to prove the new value of $\mathbb{C}_t$ using the old value of $\mathbb{L}_\ell$. Likewise, if $\text{op}(e) = \text{join}(t')$, then for any thread u , $S_e^u = S_{e_t^\rho}^u \cup S_{e_{t'}^\rho}^u$. The case of release and fork events are simpler and skipped. QED.

Thus, $\mathbb{C}_t$ stores the timestamp of the last event performed by thread σ as seen in the trace and $\mathbb{L}_\ell$ is the timestamp of the last release event of lock ℓ . Lemma 3.4 can be proved inductively.

Race Detection. Algorithm 3.2 performs race detection using the timestamps it computes. Since timestamps reflect the HB partial order (Lemma 3.3), a straightforward way to check for races would be to look for a pair of conflicting events (e, e') such that $C_e \not\sqsubseteq C_{e'}$. However, Algorithm 3.2 doesn't store the timestamps of all events and thus this approach is not possible. Instead, Algorithm 3.2 summarize the timestamps of a few number of conflicting events as follows. It uses vector clocks $\mathbb{R}_x$ and $\mathbb{W}_x$ for each variable x , often called *access history* clocks because of the following invariants.

Lemma 3.5. Let σ be a trace. For a variable x , let R_x^σ and W_x^σ be the values of the clocks $\mathbb{R}_x$ and $\mathbb{W}_x$ after Algorithm 3.2 processes σ . Then,

- $R_x^\sigma = \bigsqcup_{e \in \text{Reads}_\sigma(x)} C_e$,
- $W_x^\sigma = \bigsqcup_{e \in \text{Writes}_\sigma(x)} C_e$.

Proof (Sketch). Can be proved by induction on the length of the trace. QED.

Next, we discuss how the access history clocks can be used to detect data races.

Lemma 3.6. Let σ be a trace, e be the last event in σ and let ρ be the prefix of σ that ends before e (i.e., $\sigma = \rho \circ e$). Then, there is an event $e' \in \text{Reads}_\rho(x)$ (resp. $e' \in \text{Writes}_\rho(x)$) such that $e' \not\sqsubseteq_{\text{HB}}^\sigma e$ iff, $R_x^\rho \not\sqsubseteq C_e$ (resp. $W_x^\rho \not\sqsubseteq C_e$).

Proof (Sketch). We only sketch the proof for the case of reads. If $R_x^\rho \not\sqsubseteq C_e$, then there is a read event $e' \in \text{Reads}_\rho(x)$ such that $C_{e'} \not\sqsubseteq C_e$. Then, by Lemma 3.3, we must have $e' \not\sqsubseteq_{\text{HB}}^\sigma e$. Conversely, if there is an event $e' \not\sqsubseteq_{\text{HB}}^\sigma e$, then there is thread u such that $C_{e'}(u) > C_e(u)$. It is easy to see that $R_x^\rho(u) > C_e(u)$, giving us $R_x^\rho \not\sqsubseteq C_e$. QED.

Based on the above semantics of the clocks, we have the following:

Theorem 3.4. Let σ be a trace. Algorithm 3.2 reports a data race on σ iff there is an HB-race in σ .

Complexity. The algorithm uses $O(1)$ vector clock for each thread, lock and variable, each with dimension $\mathcal{T}$. Also every entry can be an integer with size $O(\log \mathcal{N})$. Every operation performs $O(1)$ operation on clocks, each taking $O(\mathcal{T} \cdot \log \mathcal{N})$ time. This gives us the following.

Theorem 3.5. Algorithm 3.2 runs in time $O(\mathcal{N} \cdot \mathcal{T} \cdot \log \mathcal{N})$ and uses space $O((\mathcal{T} + \mathcal{V} + \mathcal{L}) \cdot \mathcal{T} \cdot \log \mathcal{N})$.

When arithmetic operations are counted as constant time, then the above time complexity is linear in time and constant in space (assuming $\mathcal{T}, \mathcal{V}, \mathcal{L}$ to be $O(1)$).

Epoch Optimization. While the vector clock algorithm for HB race detection is the de facto approach for detecting data races because of its soundness guarantee and its linear runtime, it nevertheless can slow the runtime performance of a program, when race detection is performed at the same time. This is especially so when the number of threads is large, as the runtime increases with the number of threads. To this end, Flanagan et. al. [52] proposed *epoch optimization* and a modified algorithm based on this, called FASTTRACK, based on the insight that ‘*the full generality of vector clocks is unnecessary in most cases*’, and can result in significant performance enhancement, especially when the traces are predominated by read and write events. The first observation that they exploit here is that, in practice, the runtime of the DJIT⁺ algorithm is dominated by access events and thus an improvement in handling these events should be helpful in speeding up the overall algorithm. Second, if (e_1, e_2) is a first HB-race in σ , then all write events prior to e_2 must be ordered by $\leq_{\text{HB}}^\sigma$. When this happens, the full power of vector clocks is not needed — the $\mathbb{W}_x$ access history can be represented simply by keeping track of the last write event that was performed in the trace.

An epoch is a pair of an integer c and a thread t , denoted by $c@t$. Intuitively, epoch $c@t$ can be treated as the vector time $\perp[t \mapsto c]$. Thus, in order to compare an epoch $c@t$ with vector time V , it suffices to compare the t -th component of V with c . That is,

$$c@t \sqsubseteq V \quad \text{iff} \quad c \leq V(t). \quad (3.7)$$

Therefore, comparison between epochs is less expensive than that between vector times — $O(1)$ as opposed to $O(\mathcal{T})$ for full vector times. Until the first trace, it suffices to store an

epoch instead of the full blown clock $\mathbb{W}_x$. In practice, keeping an epoch instead of the clocks $\mathbb{W}_x$ was observed to give a drastic improvement in the runtime [52].

3.4 GOLDBLOCKS ALGORITHM FOR HB

Another algorithm for detecting **HB**-races is the Goldilocks algorithm proposed in [57]. While this algorithm is not known to be deployed in a practical race detector, it nevertheless is an important advancement in **HB** race detection, and one of the algorithms proposed in this these, is inspired from this algorithm. Here, instead of using vector clocks, one computes sets of locks and threads to track the **HB** partial order and detect **HB**-races. It nevertheless has the same gaurantee as the DJIT⁺ algorithm — it reports a race iff the input trace has an **HB** race. Further, it runs in time linear in the size of the trace and uses constant space (assuming $\mathcal{T}, \mathcal{V}$ and $\mathcal{L}$ to be constant).

Let σ be a trace. Consider the function $\text{After}_\sigma : \text{Events}_\sigma \rightarrow \text{Threads}_\sigma \cup \text{Locks}_\sigma$ defined as follows.

$$\begin{aligned} \text{After}_\sigma(e) = & \{t \in \text{Threads}_\sigma \mid \exists e' \in \text{Events}_\sigma, \text{thr}(e') = t, e \leq_{\text{HB}}^\sigma e'\} \\ & \cup \{t \in \text{Threads}_\sigma \mid \exists e' \in \text{Events}_\sigma, \text{op}(e') = \text{fork}(t), e \leq_{\text{HB}}^\sigma e'\} \\ & \cup \{\ell \in \text{Locks}_\sigma \mid \exists e' \in \text{Releases}_\sigma(\ell), e \leq_{\text{HB}}^\sigma e'\} \end{aligned} \quad (3.8)$$

The Goldilocks algorithm, similar to DJIT⁺, runs in a streaming fashion processing each event as it is observed in the trace. For every prefix σ of the trace, and for every thread t and variable x in σ , the Goldilocks algorithm maintains the sets $\text{GLS}_\sigma^{\mathbf{r}}(t, x)$ and $\text{GLS}_\sigma^{\mathbf{w}}(x)$ defined below. In the following we use, $e_{t, \mathbf{r}(x)}$ to denote the last event in σ with $\text{type}(e) = \langle t, \mathbf{r}(x) \rangle$ and $e_{\mathbf{w}(x)}$ to denote the last event in σ with $\text{op}(e) = \mathbf{w}(x)$. If such events do not exist, then we say they are **undefined**, and say that $\text{After}_\sigma(\text{undefined}) = \emptyset$.

$$\begin{aligned} \text{GLS}_\sigma^{\mathbf{r}}(t, x) &= \text{After}_{\sigma'}(e_{t, \mathbf{r}(x)}) \\ \text{GLS}_\sigma^{\mathbf{w}}(x) &= \text{After}_{\sigma'}(e_{\mathbf{w}(x)}) \end{aligned} \quad (3.9)$$

Finally, a race is declared at an event e if one of the following holds (here σ is the trace prefix that ends at e).

1. $\text{op}(e) = \mathbf{w}(x)$, $\text{GLS}_\sigma^{\mathbf{w}}(x) \neq \emptyset$ and $t \notin \text{GLS}_\sigma^{\mathbf{w}}(x)$.
2. $\text{op}(e) = \mathbf{w}(x)$ and there is a thread $t' \neq t$ such that $\text{GLS}_\sigma^{\mathbf{r}}(t', x) \neq \emptyset$ and $t \notin \text{GLS}_\sigma^{\mathbf{r}}(t', x)$.
3. $\text{type}(e) = \langle t, \mathbf{r}(x) \rangle$, $\text{GLS}_\sigma^{\mathbf{w}}(x) \neq \emptyset$ and $t \notin \text{GLS}_\sigma^{\mathbf{w}}(x)$.

The Goldilocks algorithm runs in time $O(\mathcal{N} \cdot (\mathcal{L} + \mathcal{T} \cdot \mathcal{V}))$ and uses space $O(\mathcal{T} \cdot \mathcal{V} \cdot (\mathcal{T} + \mathcal{L}))$.

3.5 SAT/SMT BASED RACE PREDICTION

In this section, we will describe race prediction based on constraint solving. As we have previously observed, data race detection based on **HB** suffers from the problem of poor predictive power. This is primarily because, **HB** conservatively orders critical sections on the same lock, even if they can be ordered in other correct reorderings. On the other hand, other approaches such as **ERASER**'s lockset violation detection report all predictable races, but are unsound. Constraint solving based race prediction approaches [43, 44] aimed to solve the problem of data race prediction in a sound and complete manner.

The basic idea behind these approaches is to cast the problem of data race prediction into a constraint satisfaction problem (typically, as a boolean satisfiability, SAT, problem or a satisfiability modulo theory, SMT, problem).

Let us, here, show how to cast the problem as a SAT formula. The formula we construct will essentially encode the property that ‘there exists a correct reordering ρ of the given trace σ in which some two conflicting events are σ -enabled’, denoted by $\varphi_{\text{predictable-race}}^\sigma$. Our formulae will use a total of $\mathcal{N} + \mathcal{N} \cdot (\mathcal{N} - 1)$ variables. First, for every event $e \in \mathbf{Events}_\sigma$, we have a boolean variable $\mathbf{x}_e$, which holds true iff e appears in the correct reordering ρ . Second, for every two events $e \neq e' \in \mathbf{Events}_\sigma$ we have a boolean variable $\mathbf{y}_{e,e'}$ which evaluates to true if both e and e' occur in ρ and $e \leq_{\text{tr}}^\rho e'$.

The formula is a conjunction of the following form:

$$\varphi_{\text{predictable-race}}^\sigma \equiv \varphi_{\text{total-order}}^\sigma \wedge \varphi_{\text{well-formed}}^\sigma \wedge \varphi_{\text{TO}}^\sigma \wedge \varphi_{\text{lastWrite}}^\sigma \wedge \varphi_{\text{enabled}}^\sigma \quad (3.10)$$

We define each of the conjuncts in the following.

Total-Order. The formula $\varphi_{\text{total-order}}^\sigma$ encodes that ρ is a sequence:

$$\begin{aligned} \varphi_{\text{total-order}}^\sigma \equiv & \bigwedge_{e \neq e' \in \mathbf{Events}_\sigma} (\mathbf{x}_e \wedge \mathbf{x}_{e'}) \implies (\mathbf{y}_{e,e'} \vee \mathbf{y}_{e',e}) \\ & \wedge \left(\bigwedge_{e \neq e' \in \mathbf{Events}_\sigma} \mathbf{y}_{e,e'} \implies \neg \mathbf{y}_{e',e} \right) \wedge \left(\bigwedge_{e \neq e' \in \mathbf{Events}_\sigma} \mathbf{y}_{e,e'} \implies (\mathbf{x}_e \wedge \mathbf{x}_{e'}) \right) \end{aligned} \quad (3.11)$$

Well-Formed. The formula $\varphi_{\text{well-formed}}^\sigma$ encodes that ρ is well-formed.

$$\varphi_{\text{well-formed}}^\sigma \equiv \bigwedge_{\ell \in \text{Locks}_\sigma} \left(\bigwedge_{\substack{a \neq a' \in \text{Acquires}_\sigma(\ell) \\ r = \text{match}_\sigma(a) \text{ exists}}} y_{a,a'} \implies y_{r,a'} \right) \wedge \left(\bigwedge_{\substack{a \neq a' \in \text{Acquires}_\sigma(\ell) \\ \text{match}_\sigma(a) = \text{undefined}}} x_a \wedge x_{a'} \implies y_{a',a} \right) \quad (3.12)$$

Thread-Order. The formula $\varphi_{\text{TO}}^\sigma$ encodes that ρ is consistent with $\leq_{\text{TO}}^\sigma$.

$$\varphi_{\text{TO}}^\sigma \equiv \bigwedge_{(e,e') \in \leq_{\text{TO}}^\sigma} x_{e'} \implies (x_e \wedge y_{e,e'}) \quad (3.13)$$

Last Write. The formula $\varphi_{\text{lastWrite}}^\sigma$ states that for every $r \in \text{Reads}_\sigma$, $\text{lastWrite}_\sigma(r) = \text{lastWrite}_\rho(r)$.

$$\begin{aligned} \varphi_{\text{lastWrite}}^\sigma \equiv & \bigwedge_{\substack{x \in \text{Vars}_\sigma \\ r \in \text{Reads}_\sigma(x) \\ w = \text{lastWrite}_\sigma(r) \text{ exists}}} x_r \implies \left(\bigwedge_{w' \in \text{Writes}_\sigma(x)} x_w \wedge y_{w,r} \right) \wedge \left(\bigwedge_{w' \in \text{Writes}_\sigma(x)} x_{w'} \implies (y_{w',w} \vee y_{r,w'}) \right) \\ & \bigwedge_{\substack{x \in \text{Vars}_\sigma \\ r \in \text{Reads}_\sigma(x) \\ \text{lastWrite}_\sigma(r) = \text{undefined}}} x_r \implies \left(\bigwedge_{w' \in \text{Writes}_\sigma(x)} x_{w'} \implies y_{r,w'} \right) \end{aligned} \quad (3.14)$$

Enabled. The formula $\varphi_{\text{enabled}}^\sigma$ states that there are some two conflicting events σ -enabled in ρ .

$$\varphi_{\text{enabled}}^\sigma \equiv \bigvee_{e \succ e' \in \text{Events}_\sigma} \left((\neg x_e \wedge \bigwedge_{f <_{\text{TO}}^\sigma e} x_f) \wedge (\neg x_{e'} \wedge \bigwedge_{f' <_{\text{TO}}^\sigma e'} x_{f'}) \right) \quad (3.15)$$

The correctness of the encoding is straightforward and is formalized below.

Lemma 3.7. For a trace σ , $\varphi_{\text{predictable-race}}^\sigma$ is satisfiable iff there is a predictable race in σ .

We note that the size of $\varphi_{\text{predictable-race}}^\sigma$ is $O(\text{poly}(\mathcal{N}))$. This gives us the following time complexity for dynamic race prediction.

Lemma 3.8. The problem of dynamic race prediction is solvable in $O(2^{\text{poly}(\mathcal{N})})$.

We remark that the work of Said et. al. [43] and Huang et. al. [44] construct a quantifier-free first order formula over the theory of integer linear arithmetic, but nevertheless, the size

of the formula thus constructed is $O(\text{poly}(\mathcal{N}))$ and the worst-case complexity is exponential in $\mathcal{N}$. SAT solving based approaches for race prediction often ship the formula constructed to off-the-shelf SAT/SMT solvers [58, 59]. The exponential complexity of SAT/SMT solving based approaches for race prediction makes these approaches expensive to use, especially when traces are long (millions of events). To make such approaches scale, existing work employ several optimizations and approximations. Most importantly, such approaches resort to a *windowing* strategy, where the trace is fragmented into smaller windows (say of about 1000 events), and a SAT encoding is constructed for each window separately. While these approaches are more scalable than the exact solving approach, they miss simple races than are between conflicting events across windows.

CHAPTER 4: SUMMARY OF CONTRIBUTIONS

In this chapter, we summarize the core technical results presented in this thesis.

1. We consider the problem of detecting data races on compressed traces in Chapter 5. We consider a grammar based compression scheme, called straight-line programs (SLPs), which forms the basis for compression algorithms like LZ77 [60]. An SLP representation of a trace σ is essentially a context free grammar G , such that the unique string contained in the language of G is σ , and can be exponentially succinct than the trace itself. We show that data races can be efficiently detected directly on the SLP representation of a trace without decompressing the grammar. In particular, we present algorithms for checking if the SLP corresponding to a trace (a) violates the lockset principle, and (b) has an HB-race that run in time that is linear in the size of the SLP and polynomial in $\mathcal{T}$, $\mathcal{V}$ and $\mathcal{L}$. Both the algorithms we present are compositional in nature — they compute summary information about smaller fragments of the trace (corresponding to lower-ranked non-terminals in the SLP grammar) and compose them to detect races and to compute summaries for larger fragments. Our evaluation suggests that compression can in fact be used as an algorithmic paradigm to speed up data race detection in an offline setting.
2. We next investigate the soundness guarantee of HB based race detection in detail. We observe that HB based race detection is sound only until the first HB race. In other words, for a trace σ , only one HB-race (namely the first one) is guaranteed to be a predictable data race in σ , and later HB-races (pairs of conflicting events unordered by $\leq_{\text{HB}}^\sigma$) need not be predictable data races. Often, however, it is desirable to identify multiple predictable data race in a single execution. Existing race detectors based on HB, on the other hand, end up reporting all HB-races, making them unsound. In Chapter 6, we present a technique to fix the unsoundness of HB beyond the first data race. In general, identifying which subset of HB races are predictable may be an intractable task, and we instead focus on those HB-races that are witnessed in correct reorderings that *respect* the $\leq_{\text{HB}}$ order of the original trace. We propose a partial order SHB (Schedulable Happens-Before) and use SHB to precisely characterize the class of HB-races that are witnessed by HB respecting correct reorderings. We then present a linear time vector clock algorithm for detecting all races identified by this characterization. Our evaluation shows that the performance of this algorithm is comparable to the DJIT⁺ algorithm (Algorithm 3.2), while this algorithm reports no false positives.

3. In Chapter 7, we consider the problem of data race prediction beyond the purview of the HB partial order. Race detection based on HB is known to be conservative — HB naively orders critical sections on the same lock in the order in which they appear in the trace, thereby ordering conflicting events even if they can be reordered in (non-HB respecting correct reorderings). We propose another partial order WCP (weak causal precedence) that weakens HB— if two events are ordered by WCP, then they are also ordered by HB, and thus every HB-race is also an WCP-race. We show that race detection based on WCP is weakly sound — a trace with a WCP race either has a predictable data race or a predictable deadlock. Further, WCP is strictly weaker than HB— there are traces which have a WCP race but no HB race, and thus has a strictly higher predictive power than HB. The proof of soundness of WCP is challenging, to say the least; in the words of Smaragdakis et. al [34]. who proposed CP, a prior weakening of HB:

“It is worth emphasizing that multiple researchers have fruitlessly pursued such a weakening of HB in the past. ... (Both the definition of CP and our proof of soundness are results of multi-year collaborative work, with several intermediate failed attempts.)”

The best part about WCP is that there is a linear time algorithm for detecting WCP races, which uses linear space. We also show that our algorithm is optimal in terms of time-space tradeoff, and further show that every single pass linear time algorithm for WCP based race detection must use linear space. Our evaluation of a prototype implementation of WCP based race detection shows that it scales well and can also report data races that HB miss.

4. We next present another method for predicting data races in Chapter 8. The starting point of the approach presented here is the principle behind HB’s predictive capability — HB reasons about correct reorderings in which order between critical sections on the same lock is the same as observed in the original execution. We show that this principle can in fact be extended further, while still remaining under the purview of tractability. We propose the notion of sync(hronization) preservation — a correct reordering of a trace σ is said to be sync-preserving if the order of all critical sections on the same lock *that are present in the correct reordering* is the same as that in the original trace. That is, some critical sections may be entirely absent from a sync-preserving correct reorderings, and we only require that the order between the *remaining* critical sections is identical to what was observed in the original execution.

We then define a sync-preserving data race to be a race that can be witnessed in a sync-preserving correct reordering. We then show that sync-preserving races can be detected in linear time (and linear space) and further show that its space usage is optimal. Theoretically, the predictive power of our algorithm is incomparable to WCP based race prediction. We also show that the problem of predicting data races that can only be witnessed in correct reorderings that flip the order between at least one pair of critical sections, is in fact NP-hard (W[1]-hard when parametrized by $\mathcal{T}$). Our evaluation results are promising and show that this approach can predict subtle data races that other comparable methods cannot predict within reasonable time.

5. In Chapter 9, we undertake a systematic study of complexity-theoretic questions pertaining data race prediction. Our first result is that (sound and complete) race prediction can be solvable in time $O(\mathcal{TN}^{2\mathcal{T}-2})$, which is *polynomial time* when the number of threads is constant. This is surprising, given that SAT/SMT solving based symbolic predictive analysis [43, 44] for race detection resorts to an exponential time algorithm even when the number of threads is constant. We then show that the problem is, in fact, W[1]-hard in the parameter $\mathcal{T}$, and thus there is no FPT algorithm for race prediction, parametrized by $\mathcal{T}$. We then investigate special cases of the race prediction problem and study the complexity in this setting. We obtain several algorithms that run in lower-order polynomial time and show that they are optimal by proving (almost) matching lower bounds.
6. We consider the problem of detecting violations of conflict serializability, which is a notion of atomicity. State-of-the-art algorithms for detecting conflict serializability violations construct a graph of transactions and detect cycles in this graph, each time a new event is seen. The total runtime of this algorithm is cubic in the size of the trace being analyzed. In Chapter 10, we propose a *linear time* algorithm for detecting conflict serializability violations. Our algorithm AERODROME is based on updating vector clocks. Our evaluation shows that this algorithm gives a huge speedup over prior approaches.

CHAPTER 5: DATA RACE DETECTION ON COMPRESSED TRACES

While dynamic analysis is sometimes performed online as the program is running, it is often performed offline, on a stored trace of the program. The reasons for performing offline dynamic analysis are many. The overhead of analyzing the trace as the program is running may be large, causing undesirable slowdowns. This is especially true for expensive dynamic analysis techniques which employ heavy-weight machinery such as the use of SMT solvers [43, 44], graph based analysis [38, 61] or even vector clocks [48, 52]. For example, in order to protect against *metadata races*, dynamic data race detectors must introduce additional synchronization, which can give rise to additional performance overhead. Experiments [62, 63] to evaluate the overhead due to such synchronization in implementations of the FASTTRACK [52] algorithm suggest that the resulting slow-down is significant. For concurrent programs, these latencies may also sometimes introduce subtle synchronization errors that may otherwise not be present in the original program when run alone without dynamic analysis. Often, it is desirable to perform multiple, different analyses on a single trace, and the kinds of analyses to be performed may even be unknown at the time the program is being observed. Storing the trace and later replaying it in a controlled environment, can help in debugging programs, in understanding performance overheads, in performance tuning and visualization [64]. Further, trace-driven simulations, used widely in computer architecture for quantitative evaluations of new ideas and designs [65, 66], heavily rely on random access to traces.

However program traces are often huge, recording millions and billions of events. When debugging a large software application, long traces are often necessary to ensure adequate code coverage. This is especially acute for multi-threaded programs where subtle concurrency bugs are often revealed only under specific thread schedules, in traces with millions of events [67]. In such circumstances, the only way to alleviate the warehousing needs of storing such traces is to compress them [65, 66].

In this thesis, we study the problem of detecting data races in programs by examining compressed traces and present our results in this chapter. The naïve approach to solving this problem would be to uncompress the trace and then process it using any one of the many algorithms that have been developed for dynamic data race detection. But is this necessary? Is this naïve algorithm, asymptotically, the best one can hope for? Studying the complexity of problems where the input is represented succinctly has a long history. Starting from the seminal paper by Galperin and Wigderson [68], where they studied the complexity of graph problems when the input graph is represented by a circuit, it has been observed

```

public class Test extends Thread{

    static final long ITERS = 1000000000L;
    static int y;

    public void inc() {
        y++;
    }

    @Override
    public void run() {
        for (long i = 0; i < ITERS; i++) {
            inc();
        }
    }

    public static void main(String args[]) throws Exception {
        final Test t1 = new Test();
        final Test t2 = new Test();
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("y (actual) = " + y);
        System.out.println("y (expected) = " + ITERS*2);
    }
}

```

Figure 5.1: A simple concurrent program in Java

that typically there is an exponential blowup in the complexity of problems when they are solved on compressed inputs [68–74]. Thus, often the naïve algorithm is the best algorithm asymptotically.

Our results are an exception to the above rule. We consider two classical race detection approaches — race detection based on the HB partial order and based on ERASER’s lockset violation principle — and extend them to work directly on the compressed trace without first uncompressing it. Our algorithms run in time that is linear in the size of the compressed trace. Thus, we show that compression can in fact be used as an algorithmic principle to speedup the analysis in this context. A prior version of our results has appeared in [15].

To understand why compression actually speeds up the analysis, consider the simple program shown in Figure 5.1. A single execution of this program generates about 680 million events, taking about 1.3 GB disk space. However, when this trace is compressed using the Sequitur algorithm [75–77], the compressed representation only occupies about 34 MB of disk space. The reason for this effective compression is that the program has a simple loop structure that is executed multiple times. Thus the program trace has a “regular” structure that the compression algorithm exploits. An algorithm processing the uncompressed trace is

agnostic to this regularity, and is forced to repeat the same analysis each time the sub-trace corresponding to the loop body is encountered. Compression makes this regular structure “visible”, and an algorithm working on the compressed representation can exploit it by only performing an analysis only once for each distinct sub-trace.

We consider compression schemes that compress traces as straight line programs (SLPs). SLPs are a special class of context-free grammars where the language of the grammar consists of a single string, namely, the trace being compressed. Several lossless compression schemes, like run-length encoding and the family of Lempel-Ziv encodings [60], can be converted efficiently to SLPs of similar size. Such a compact representation of a trace have also been considered previously [78]. Our algorithms on SLPs proceed inductively on the structure of the grammar that compute book-keeping information for each non-terminal in the grammar, compositionally. Our algorithms on SLPs proceed inductively on the structure of the grammar, and compute, in a compositional fashion, book-keeping information for each non-terminal in the grammar. Thus, a sub-trace generated by a non-terminal that may appear in many positions in the uncompressed trace, is analyzed only once.

We have implemented our algorithms in a tool called ZIPTRACK available publicly [79]. The traces are compressed using a popular SLP-based compression algorithm called Sequitur [77]. Our experiments on standard benchmark examples reveal that the algorithms on compressed traces perform well, and on large traces, often have an order of magnitude improvement over algorithms running in the uncompressed setting.

The rest of the chapter is organized as follows. We discuss preliminaries about SLPs and other notation in Section 5.1. In Section 5.2, we briefly present our happens-before based data race detection algorithm on compressed traces. Our algorithm for checking violations of the lockset discipline on compressed traces is presented briefly in Section 5.3. We present our experimental results in Section 5.4.

5.1 PRELIMINARIES

In the section we introduce basic notation, our assumptions about concurrent programs, the happens before ordering on events, and some classical algorithms for race detection.

Notations.. We fix a trace σ . For a thread $t \in \text{Threads}_\sigma$ and variable $x \in \text{Vars}_\sigma$, we denote $\text{Reads}_\sigma(t, x) = \{e \in \text{Reads}_\sigma(x) \mid \text{thr}(e) = t\}$. We will use $\text{Rd}(\sigma) \subseteq \text{Threads}_\sigma \times \text{Vars}_\sigma$ to denote the set of pairs (t, x) for which $\text{Reads}_\sigma(t, x) \neq \emptyset$. Similarly, $\text{Wr}(\sigma) = \{x \in \text{Vars}_\sigma \mid \text{Writes}_\sigma(x) \neq \emptyset\}$. For a thread $t \in \text{Threads}_\sigma$, we say $\text{ThEvents}_\sigma^{\text{join}}(t) = \{e \in \text{Events}_\sigma \mid \text{thr}(e) = t \text{ or } \text{op}(e) = \text{join}(t)\}$. Similarly, $\text{ThEvents}_\sigma^{\text{fork}}(t) = \{e \in \text{Events}_\sigma \mid \text{thr}(e) =$

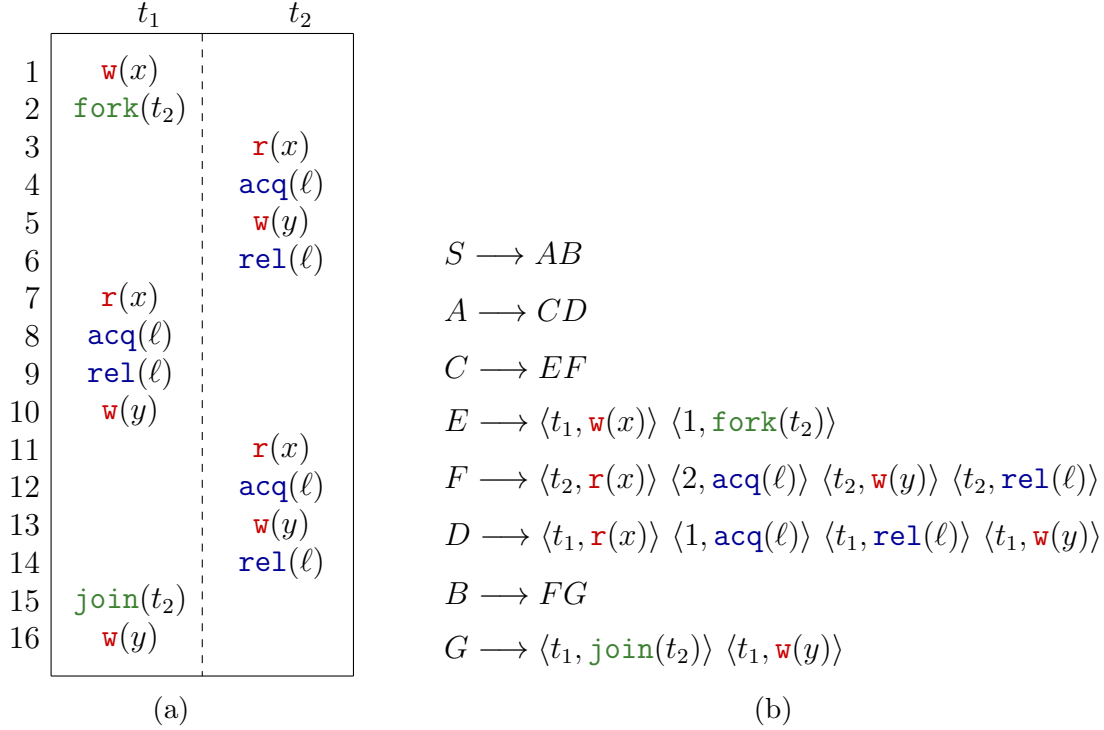


Figure 5.2: Example trace σ_1 and its SLP representation

t or $\text{op}(e) = \text{fork}(t)\}$. As before, when σ is clear from the context, we may drop it.

For a non-empty subset of events $S \subseteq \text{Events}_\sigma$, we will denote by $\text{Last}_\sigma(S)$ the (unique) event $e \in S$, that is latest in σ among the events in S . Similarly, $\text{First}_\sigma(S)$ is the event $e \in S$ that is earliest in σ amongst the events in S . When S is empty, we say both $\text{First}_\sigma(S)$ and $\text{Last}_\sigma(S)$ are undefined.

Example 5.1. We illustrate the above definitions on the example trace σ_1 shown in Figure 5.2. Let $S_1 = \text{Reads}_{\sigma_1}(t_2, x) = \{e_3, e_{11}\}$ and $S_2 = \text{Writes}_{\sigma_1}(y) = \{e_5, e_{10}, e_{13}, e_{16}\}$. The set $\text{Rd}(\sigma_1) = \{(1, x), (2, x)\}$ while $\text{Wr}(\sigma_1) = \{x, y\}$. Finally, $\text{Last}_{\sigma_1}(S_1) = e_{11}$, and $\text{First}_{\sigma_1}(S_2) = e_5$.

Straight Line Programs (SLP). We consider traces that are compressed using special context-free grammars called straight line programs (SLP). Recall that a context-free grammar (in Chomsky Normal Form) is $G = (T, N, S, R)$, where T is the set of terminals, N the set of non-terminals, $T \cup N$ is the set of symbols, $S \in N$ is the start symbol, and R is the set of rules in which each rule in R is either $A \rightarrow a$ or $A \rightarrow BC$, for $A, B, C \in N$ and $a \in T$. A *straight line program* is a context free grammar such that (a) for every non-terminal A , there is exactly one rule where A appears on the left, and (b) the non-terminals are ranked in such that way that in every rule, the non-terminals on the right are of larger rank than

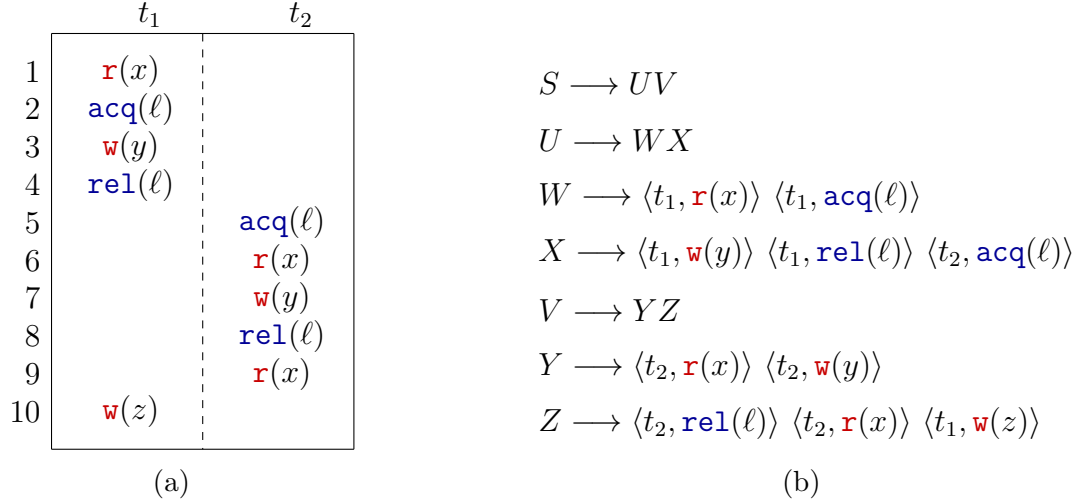


Figure 5.3: Example trace σ_2 and its SLP representation

the non-terminal on the left of the rule, i.e., for rules $A \rightarrow BC$, $A \prec B$ and $A \prec C$. It is easy to observe that the language of the grammar contains a single string, namely, the one that is being succinctly represented by the SLP. Without loss of generality, we will assume that every non-terminal in the SLP is *useful*, i.e., every non-terminal in the grammar appears in some sentential form in the unique derivation in the grammar. Thus, the language associated with any non-terminal A has a single string. We will call this (unique) string generated by non-terminal A a *chunk*, and denote it by $\llbracket A \rrbracket$. We will often abuse notation and refer to both $\llbracket A \rrbracket$ and A as “ A ”. For example, Locks_A will mean $\text{Locks}_{\llbracket A \rrbracket}$.

The *size* of an SLP $G = (T, N, S, R)$ will be taken to be $|T| + |N|$; note that this measure of size is linearly related to other measures of size one might consider like $|R|$ or sum of the sizes of all the rules in R . We make a couple of observations about the size of an SLP versus the size of the trace it represents. First, every trace $\sigma = e_1, e_2, \dots, e_n$ can be represented by a “trivial” SLP of size $O(n)$ as follows. The non-terminals are $\{A_{[i,i]} \mid 1 \leq i \leq n\} \cup \{A_{[1,i]} \mid 1 \leq i \leq n\}$ with start symbol $A_{[1,n]}$. Intuitively, $A_{[i,i]}$ represents the string e_i , while $A_{[1,i]}$ represents the prefix of length i . This is accomplished by the rules $A_{[i,i]} \rightarrow e_i$ and $A_{[1,i]} \rightarrow A_{[1,i-1]}A_{[i,i]}$ for each $1 \leq i \leq n$. Second, the SLP representation of a string σ maybe exponentially smaller than σ itself. For example, take $\sigma = a^{2^n}$. An $O(n)$ SLP representation for σ is as follows: $N = \{A_i \mid 0 \leq i \leq n\}$ with rules $A_0 \rightarrow a$, and $A_{i+1} \rightarrow A_i A_i$. One can inductively observe that $\llbracket A_i \rrbracket = a^{2^i}$, and so $\llbracket A_n \rrbracket = \sigma$.

Example 5.2. Figure 5.2 describes an SLP representation of trace σ_1 . The rules for E, F, D , and G are not strictly in the format of an SLP, but it can easily be converted into one; the representation in Figure 5.2 is sufficient for our illustrative purposes. We will again use

e_i to denote the i th event of σ_1 . Chunk E represents e_1, e_2 , F represents e_3, e_4, e_5, e_6 and $e_{11}, \dots, e_{14}$, D represents e_7, e_8, e_9, e_{10} , and G represents e_{15}, e_{16} . The sub-traces represented by the other non-terminals can be similarly discovered. As mentioned before, we will confuse the notation distinguishing between a non-terminal and the string it represents. Thus, for example, $\text{Threads}_E = \{t_1, t_2\}$.

Similarly, the SLP for σ_2 is shown in Figure 5.3. The sub-traces represented by non-terminals need not conform to thread and critical section boundaries. For example, the chunk $\llbracket X \rrbracket$ has partial critical sections of different threads.

Several well known algorithms for SLP based compression are known in the literature. The most basic and popular one is Sequitur [75, 76]. Sequitur takes a string as an input, and generates an SLP representing the trace. It runs in time and space linear in the size of the input string. The Sequitur algorithm works in an online incremental fashion; it reads the input string one character at a time, and updates the SLP generated so-far. It maintains a list of digrams (symbol pairs) that occur somewhere in the SLP so-far. On seeing a new character, the algorithm appends it at the end of the rule corresponding to the start symbol. The new digram formed (by appending the new character to the last symbol of the rule) is added to the list of digrams, if it is not already present. Otherwise, a new rule, with a fresh nonterminal generating the digram, is added to the SLP, and every occurrence of the digram is replaced by the freshly introduced non-terminal. At every step, non-terminals, that are not useful, are also removed. Other popular grammar based compression schemes include Sequential [80], LZ77 [60], LZW [81], Bisection [82], longest match [83] and Re-Pair [84].

5.2 HB-RACES IN COMPRESSED TRACES

In this section, we will present our algorithm for detecting HB-races in compressed traces represented by SLPs. The algorithm's running time will be linear in the size of the SLP (as opposed to algorithms analyzing uncompressed traces with running times at least linear in the size of the uncompressed trace). While it is very different from the classical vector clock algorithm, it is similar in flavor to the Goldilocks Algorithm.

5.2.1 Detecting Cross-Races

Our algorithm will proceed inductively. Starting from the non-terminals of largest rank, we will proceed to determine for each non-terminal A , whether there is an HB-race amongst the events in the chunk that A generates. In other words, for each non-terminal A , we

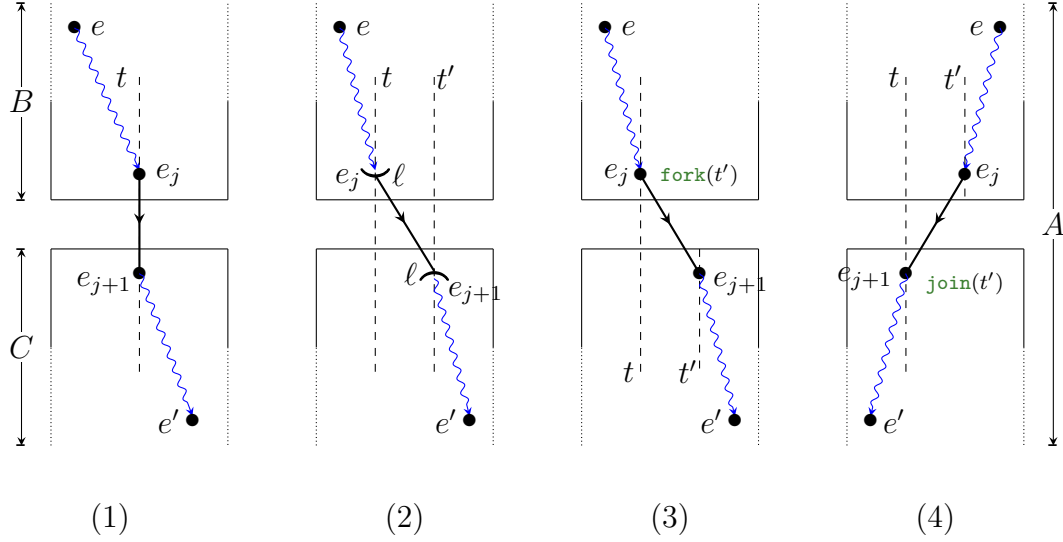


Figure 5.4: Illustrating the various scenarios that establish $e_j <_{\text{HB}}^{BC} e_{j+1}$. In (2), ‘ $\frown \ell$ ’ represents $\text{acq}\ell$, ‘ $\smile \ell$ ’ is $\text{rel}(\ell)$.

will determine the predicate $\text{HBRace?}(A)$ which is true if and only if there is an HB -race in $\llbracket A \rrbracket$. For a non-terminal A , whose (only) rule is of the form $A \rightarrow a$, where a is an event, $\text{HBRace?}(A)$ is clearly false, because $\llbracket A \rrbracket$, in this case, has only one event.

Let us now consider the case when the rule corresponding to A has the form $A \rightarrow BC$, where B and C are non-terminals of higher rank. If there is a race in chunk $\llbracket A \rrbracket$ between events (say) e and e' , then it is one of two kinds. The first case is when e and e' both belong to chunk $\llbracket B \rrbracket$ or both belong to chunk $\llbracket C \rrbracket$. The existence of such races can be determined by computing (inductively) the predicates $\text{HBRace?}(B)$ and $\text{HBRace?}(C)$. The other possibility is that $e \in \llbracket B \rrbracket$ while $e' \in \llbracket C \rrbracket$. Discovering the presence of such *cross-races*, is the main challenge we need to overcome.

Consider two events e, e' such that $e \in \llbracket B \rrbracket$ and $e' \in \llbracket C \rrbracket$. Suppose $e <_{\text{HB}}^{BC} e'$. Then, there is a sequence $e = e_1, e_2, \dots, e_n = e'$ that satisfies the conditions in Definition 3.2. Thus, for $1 \leq i \leq n - 1$, we have the trace order $e_i \leq_{\text{tr}}^{BC} e_{i+1}$. Also, $e = e_1 \in \llbracket B \rrbracket$, and $e_n = e' \in \llbracket C \rrbracket$. This means that there exists j such that for all $i \leq j$, $e_i \in \llbracket B \rrbracket$, and for all $i \geq j + 1$, $e_i \in \llbracket C \rrbracket$. In other words, (e_j, e_{j+1}) is how the sequence $e_1, \dots, e_n$ “crosses” the B - C boundary (see Figure 5.4). Observe that we have $e = e_1 \leq_{\text{HB}}^B e_j$ and $e_{j+1} \leq_{\text{HB}}^C e_n = e'$. It is important to note that the relationship between e and e_j (resp. e_{j+1} and e') only depends on the events in chunk $\llbracket B \rrbracket$ (resp. $\llbracket C \rrbracket$). Depending on which of the conditions (a), (b), (c), and (d) of Definition 3.2 hold for the pair (e_j, e_{j+1}) , we have one of the following: either e_j and e_{j+1} are events of the same thread, or e_j is a release event and e_{j+1} is an acquire event

on the same lock, or e_j is a fork event and e_{j+1} is an event of the child thread, or e_j is a join event and e_{j+1} is an event of the parent thread. These scenarios are illustrated in Figure 5.4. Thus, if an event $e \in B$ *happens-before* an event $e' \in C$ then there is a common thread or a common lock through which the ordering is “communicated” across the B - C boundary. The converse of this observation is also true. We now make this intuition precise.

For a trace σ , and event $e \in \text{Events}_\sigma$, recall the function After_σ from Section 3.4 (Equation (3.8)):

$$\begin{aligned} \text{After}_\sigma(e) = & \{t \in \text{Threads}_\sigma \mid \exists e' \in \text{Events}_\sigma, \text{thr}(e') = t, e \leq_{\text{HB}}^\sigma e'\} \\ & \cup \{t \in \text{Threads}_\sigma \mid \exists e' \in \text{Events}_\sigma, \text{op}(e') = \text{fork}(t), e \leq_{\text{HB}}^\sigma e'\} \\ & \cup \{\ell \in \text{Locks}_\sigma \mid \exists e' \in \text{Releases}_\sigma(\ell). e \leq_{\text{HB}}^\sigma e'\} \end{aligned} \quad (5.1)$$

We can, dually, define the set of locks/threads that have an event HB-before e in σ .

$$\begin{aligned} \text{Before}_\sigma(e) = & \{t \in \text{Threads}_\sigma \mid \exists e' \in \text{Events}_\sigma, \text{thr}(e') = t, e' \leq_{\text{HB}}^\sigma e\} \\ & \cup \{t \in \text{Threads}_\sigma \mid \exists e' \in \text{Events}_\sigma, \text{op}(e') = \text{join}(t), e' \leq_{\text{HB}}^\sigma e\} \\ & \cup \{\ell \in \text{Locks}_\sigma \mid \exists e' \in \text{Acquires}_\sigma(\ell). e' \leq_{\text{HB}}^\sigma e\} \end{aligned} \quad (5.2)$$

The main observation that underlies the algorithm is that **After** and **Before** sets can be used to discover HB ordering between events across chunks.

Lemma 5.1. Consider events $e \in \llbracket B \rrbracket$ and $e' \in \llbracket C \rrbracket$. $e \leq_{\text{HB}}^{BC} e'$ iff $\text{After}_B(e) \cap \text{Before}_C(e') \neq \emptyset$.

Proof. Recall that if $e \leq_{\text{HB}}^{BC} e'$ then there is a sequence $e = e_1, e_2, \dots, e_n = e'$ that satisfies the conditions in Definition 3.2. Let j be such that for all $i \leq j$, $e_i \in \llbracket B \rrbracket$ and for all $i \geq j+1$, $e_i \in \llbracket C \rrbracket$. Also, we have $e \leq_{\text{HB}}^B e_j$ and $e_{j+1} \leq_{\text{HB}}^C e'$.

If e_j, e_{j+1} satisfy condition (a) of Definition 3.2, then $t \in \text{After}_B(e) \cap \text{Before}_C(e')$, where t is the thread performing both e_j and e_{j+1} . If e_j, e_{j+1} satisfy condition (b), then $\ell \in \text{After}_B(e) \cap \text{Before}_C(e')$, where ℓ is the lock released/acquired by event e_j and e_{j+1} , respectively. Finally, if e_j, e_{j+1} satisfy condition (c) or (d), $t' \in \text{After}_B(e) \cap \text{Before}_C(e')$, where t' is the thread forked by e_j and the one performing e_{j+1} or the child thread joined in e_{j+1} .

On the other hand if $\text{After}_B(e) \cap \text{Before}_C(e') \neq \emptyset$, then we can demonstrate that $e \leq_{\text{HB}}^{BC} e'$. Let $u \in \text{After}_B(e) \cap \text{Before}_C(e')$. Let us first consider the case when u is a lock ℓ . Since $u = \ell \in \text{After}_B(e)$, then there is an event $e_1 \in \llbracket B \rrbracket$ such that e_1 is a **rel**(ℓ) event and $e \leq_{\text{HB}}^B e_1$. Similarly, since $u = \ell \in \text{Before}_C(e')$, there is an event $e_2 \in \llbracket C \rrbracket$ such that e_2 is an **acq**(ℓ) event and $e_2 \leq_{\text{HB}}^C e'$. Putting these together, we get that $e \leq_{\text{HB}}^{BC} e'$. Let us now consider the case when $u = t$. Again, $u = t \in \text{After}_B(e)$ means that there is an event $e_1 \in \llbracket B \rrbracket$ such that $e \leq_{\text{HB}}^B e_1$ and either e_1 is an event performed by t , or e_1 is a **fork**(t) event. Similarly,

$u = t \in \text{Before}_C(e')$, means that there is an event $e_2 \in \llbracket C \rrbracket$ such that $e_2 \leq_{\text{HB}}^C e'$ and either e_2 is an event performed by t , or e_2 is a **join**(t) event. In all these four cases, we can conclude that $e \leq_{\text{HB}}^{BC} e'$. QED.

Lemma 5.1 suggests that cross races in chunk BC can be discovered by maintaining the after and before sets of data access events. However, we don't need to maintain these sets for all access events; instead, we can do it only for the first and last events. This is the content of Lemma 5.3. In order to prove Lemma 5.3, we first note the following observation:

Lemma 5.2. Let σ be a trace σ , and let e_1, e_2 be events such that $e_1 \leq_{\text{HB}}^\sigma e_2$. Then, $\text{After}_\sigma(e_2) \subseteq \text{After}_\sigma(e_1)$ and $\text{Before}_\sigma(e_1) \subseteq \text{Before}_\sigma(e_2)$.

Proof (Sketch). Follows from the transitivity of $\leq_{\text{HB}}^\sigma$. QED.

We now formally state the observation we previously noted.

Lemma 5.3. If there is no HB-race in $\llbracket B \rrbracket$ or in $\llbracket C \rrbracket$, and if there is an HB-race between events $e \in \text{Events}_{\llbracket B \rrbracket}$ and $e' \in \text{Events}_{\llbracket C \rrbracket}$ then, there is an HB-race between last_B^e and $\text{first}_C^{e'}$, where

$$\text{last}_B^e = \begin{cases} \text{Last}_B(\text{Reads}_B(t, x)) & \text{if } e = \langle t, \mathbf{r}(x) \rangle \\ \text{Last}_B(\text{Writes}_B(x)) & \text{if } e = \langle t, \mathbf{w}(x) \rangle \end{cases}$$

and

$$\text{first}_C^{e'} = \begin{cases} \text{First}_C(\text{Reads}_C(t', x)) & \text{if } e' = \langle t', \mathbf{r}(x) \rangle \\ \text{First}_C(\text{Writes}_C(x)) & \text{if } e' = \langle t', \mathbf{w}(x) \rangle \end{cases}$$

Proof. First, notice that, for a given trace σ , and for $(t, x) \in \text{Rd}(\sigma)$, the elements of the set $\text{Reads}_\sigma(t, x)$ are totally ordered by the relation $\leq_{\text{HB}}^\sigma$. This follows from the rule (a) of Definition 3.2. Thus, for an event $e \in \text{Reads}_\sigma(t, x)$, we have $\text{First}_\sigma(\text{Reads}_\sigma(t, x)) \leq_{\text{HB}}^\sigma e \leq_{\text{HB}}^\sigma \text{Last}_\sigma(\text{Reads}_\sigma(t, x))$. Using Lemma 5.2, we must also have

$$\begin{aligned} \text{After}_\sigma(\text{Last}_\sigma(\text{Reads}_\sigma(t, x))) &\subseteq \text{After}_\sigma(e) \\ \text{After}_\sigma(e) &\subseteq \text{After}_\sigma(\text{First}_\sigma(\text{Reads}_\sigma(t, x))) \\ \text{Before}_\sigma(\text{First}_\sigma(\text{Reads}_\sigma(t, x))) &\subseteq \text{Before}_\sigma(e) \\ \text{Before}_\sigma(e) &\subseteq \text{Before}_\sigma(\text{Last}_\sigma(\text{Reads}_\sigma(t, x))) \end{aligned} \tag{5.3}$$

Next observe that for a trace σ , if there is no HB-race in σ , the elements of the set

$\text{Writes}_\sigma(x)$ for every $x \in \text{Wr}(\sigma)$ must be totally ordered by $\leq_{\text{HB}}^\sigma$, and thus for such a trace σ ,

$$\begin{aligned}
\text{After}_\sigma(\text{Last}_\sigma(\text{Writes}_\sigma(x))) &\subseteq \text{After}_\sigma(e) \\
\text{After}_\sigma(e) &\subseteq \text{After}_\sigma(\text{First}_\sigma(\text{Writes}_\sigma(x))) \\
\text{Before}_\sigma(\text{First}_\sigma(\text{Reads}_\sigma(x))) &\subseteq \text{Before}_\sigma(e) \\
\text{Before}_\sigma(e) &\subseteq \text{Before}_\sigma(\text{Last}_\sigma(\text{Writes}_\sigma(x)))
\end{aligned} \tag{5.4}$$

Now since there is a race between $e \in \llbracket B \rrbracket$ and $e' \in \llbracket C \rrbracket$, by Lemma 5.1, we have $\text{After}_B(e) \cap \text{Before}_C(e') = \emptyset$. Clearly, this means that $\text{After}_B(\text{last}_B^e) \cap \text{Before}_C(\text{first}_C^{e'}) = \emptyset$. Again, from Lemma 5.1, we have that $\text{last}_B^e \parallel_{\text{HB}}^{BC} \text{first}_C^{e'}$. Also, $\text{last}_B^e \asymp \text{first}_C^{e'}$ because $e \asymp e'$. Thus, there is an HB race between last_B^e and $\text{first}_C^{e'}$. QED.

Lemma 5.3 suggests that in order to check for *cross* races, it is enough to inductively maintain the after sets of the last read/write events and the before sets of the first read/write events of each variable and thread. We will denote these sets by **AfterLastRd**, **AfterLastWr**, **BeforeFirstRd** and **BeforeFirstWr**. Formally,

$$\begin{aligned}
\text{AfterLastRd}_D(t, x) &= \text{After}_D(\text{Last}_D(\text{Reads}_D(t, x))) \\
\text{AfterLastWr}_D(x) &= \text{After}_D(\text{Last}_D(\text{Writes}_D(x))) \\
\text{BeforeFirstRd}_D(t, x) &= \text{Before}_D(\text{First}_D(\text{Reads}_D(t, x))) \\
\text{BeforeFirstWr}_D(x) &= \text{Before}_D(\text{First}_D(\text{Writes}_D(x)))
\end{aligned} \tag{5.5}$$

where we set both $\text{After}_D(\text{undefined})$ and $\text{Before}_D(\text{undefined})$ to be $\emptyset$.

Based on all of these observations we can conclude that for a non-terminal A with rule $A \rightarrow BC$, we have,

$$\begin{aligned}
\text{HBRace?}(A) &\equiv \text{HBRace?}(B) \vee \text{HBRace?}(C) \\
&\vee \bigvee_{x \in \text{Wr}(B) \cap \text{Wr}(C)} \text{AfterLastWr}_B(x) \cap \text{BeforeFirstWr}_C(x) = \emptyset \\
&\vee \bigvee_{x \in \text{Wr}(B), (t, x) \in \text{Rd}(C)} \text{AfterLastWr}_B(x) \cap \text{BeforeFirstRd}_C(t, x) = \emptyset \\
&\vee \bigvee_{(t, x) \in \text{Rd}(B), x \in \text{Wr}(C)} \text{AfterLastRd}_B(t, x) \cap \text{BeforeFirstWr}_C(x) = \emptyset
\end{aligned} \tag{5.6}$$

Thus, our race detection algorithm will be complete if we can effectively compute the sets $\text{AfterLastRd}_B(t, x)$, $\text{AfterLastWr}_B(x)$, $\text{BeforeFirstRd}_C(t, x)$, and $\text{BeforeFirstWr}_C(x)$. We embark on this challenge in the next section.

Before we formally state the correctness of the **HBRace?** predicate, we first note the following simple observation.

Proposition 5.1. Let σ be a trace and $e_1, e_2 \in \sigma$. $e_1 \leq_{\text{HB}}^\sigma e_2$ iff for any trace $\sigma' = \sigma_1 \sigma \sigma_2$, we have $e_1 \leq_{\text{HB}}^{\sigma'} e_2$.

Informally, this means that the presence/absence of an HB-race between two events in a trace σ , does not get affected by the “context” in which σ is placed. The proof of the above proposition follows easily from Definition 3.2, and is skipped.

We can now state the correctness of the **HBRace?** predicate.

Theorem 5.1. For any non-terminal A , **HBRace?**(A) = **true** if and only if there are events $e_1, e_2 \in \llbracket A \rrbracket$ such that $e_1 \succ e_2$ and $e_1 \parallel_{\text{HB}}^A e_2$.

Proof. The proof is by induction on the ranking of the non-terminals. In the base case, the theorem clearly holds. In the inductive step, consider a non-terminal $A \rightarrow BC$. Suppose **HBRace?**(A) = **true**. Based on Equation (5.6), this means either **HBRace?**(B) or **HBRace?**(C) are **true** or some pair of after and before sets are disjoint. If **HBRace?**(B) (or **HBRace?**(C)) is **true**, then by induction hypothesis, there is an HB-race in $\llbracket B \rrbracket$ (or $\llbracket C \rrbracket$), and so there is a race in $\llbracket A \rrbracket$. This follows from Proposition 5.1. If not (that is, if both **HBRace?**(B) and **HBRace?**(C) are false), the condition in one of line 2, 3 or 4 in Equation (5.6) is true. The proof is the same no matter which of these conditions hold. Therefore, let us assume without loss of generality, that $\text{AfterLastRd}_B(t, x) \cap \text{BeforeFirstWr}_C(x) = \emptyset$ for some thread t and variable x . Since $(t, x) \in \text{Reads}_B$ and $x \in \text{Writes}_C$, both $\text{AfterLastRd}_B(t, x)$ and $\text{BeforeFirstWr}_C(x)$ are non-empty sets. Thus, both the events e, e' given by $e = \text{Last}_B(\text{Reads}_B(t, x))$ and $e' = \text{First}_C(\text{Writes}_C(x))$ are well defined. Also, by definition, $\text{After}_B(e) = \text{AfterLastRd}_B(t, x)$ and $\text{Before}_C(e') = \text{BeforeFirstWr}_C(x)$. Thus, we have $\text{After}_B(e) \cap \text{Before}_C(e') = \emptyset$. Then, by Lemma 5.1, $e \parallel_{\text{HB}}^e$ in the chunk $\llbracket BC \rrbracket = \llbracket A \rrbracket$. Also, e, e' are clearly conflicting (read and write of same variable). Thus, (e, e') is an HB race in the chunk A .

Conversely, suppose there are a pair of events $e_1, e_2 \in \llbracket A \rrbracket$ such that $e_1 \succ e_2$ and $e_1 \parallel_{\text{HB}}^e$. Without loss of generality, let us assume that e_1 is $\mathbf{r}(x)$ -event, and e_2 is a $\mathbf{w}(x)$ -event. If $e_1, e_2 \in \llbracket B \rrbracket$ or $e_1, e_2 \in \llbracket C \rrbracket$ then by induction hypothesis, **HBRace?**(B) or **HBRace?**(C) is **true**. By Equation (5.6) this would imply that **HBRace?**(A) is **true**. Now consider the case when there are no races in B or C , and there are $e_1 \in \llbracket B \rrbracket$ and $e_2 \in \llbracket C \rrbracket$ which are in race. Without loss of generality, let us assume that $e_1 = \langle t, \mathbf{r}(x) \rangle$ is a read event, and $e_2 = \langle t', \mathbf{w}(x) \rangle$ is a write event on the same variable x . Then, by Lemma 5.3, the events $e = \text{last}_B^{e_1} = \text{Last}_B(\text{Reads}_B(t, x))$ and $e' = \text{first}_C^{e_2} = \text{First}_C(\text{Writes}_C(x))$ are in race too. Then, again using Lemma 5.1, we must have that $\text{After}_B(e) \cap \text{Before}_C(e') = \emptyset$. Now, by definition of the sets **AfterLastRd** and **BeforeFirstWr**, we have $\text{AfterLastRd}_B(t, x) = \text{After}_B(e)$ and $\text{BeforeFirstWr}_C(x) = \text{Before}_C(e')$. Thus, clearly, **HBRace?**(A) = **true**. QED.

Example 5.3. Let us illustrate the ideas presented in this section through some examples. We will consider traces σ_1 and its SLP in Figure 5.2, and σ_2 with its SLP in Figure 5.3.

We begin by giving examples of **Before** and **After** sets.

$$\begin{aligned}
\text{After}_E(e_1) &= \{t_1, t_2\} & \text{After}_C(e_1) &= \{t_1, t_2, \ell\} \\
\text{Before}_G(e_{16}) &= \{t_1, t_2\} & \text{Before}_B(e_{16}) &= \{t_1, t_2, \ell\} \\
\text{After}_W(e_1) &= \{t_1\} & \text{After}_U(e_1) &= \{t_1, t_2, \ell\} \\
\text{Before}_X(e_3) &= \{t_1\} & \text{Before}_U(e_3) &= \{t_1, \ell\}
\end{aligned} \tag{5.7}$$

Let us highlight the significant aspects of these examples. $t_2 \in \text{After}_E(e_1)$ because of $e_2 = \langle t_1, \text{fork}(t_2) \rangle$ and $\ell \in \text{After}_C(e_1)$ because of event $e_6 = \langle t_2, \text{rel}(\ell) \rangle$. On the other hand, $\ell \notin \text{After}_W(e_1)$ because there is no $\text{rel}(\ell)$ event in chunk $\llbracket W \rrbracket$ (of σ_2). But when considering the chunk $\llbracket U \rrbracket$ (of σ_2), we have $\ell \in \text{After}_W(e_1)$ because of the event $e_4 = \langle t_1, \text{rel}(\ell) \rangle$. Next, $t_2 \in \text{Before}_G(e_{16})$ because of the join event e_{15} , and $\ell \in \text{Before}_B(e_{16})$ because of acquire event e_{12} . In trace σ_2 , $\ell \in \text{Before}_U(e_3)$ because of acquire event e_2 .

Now let us consider the computation of cross-races for the chunks in Figure 5.2. For $M \in \{D, E, F, G\}$, it is easy to see that $\text{HBRace?}(M) = \text{false}$, because each of these chunks only contain events of one thread. The absence of race between e_1 and e_3 can be seen because $\text{AfterLastWr}_E(x) = \{t_1, t_2\}$ and $\text{BeforeFirstRd}_F(t_2, x) = \{t_2\}$, both of which have the thread t_2 in common, and thus the intersection $\text{AfterLastWr}_E(x) \cap \text{BeforeFirstRd}_F(x)$ is non-empty. In fact, what this reasoning demonstrates is that there is no race between any $\mathbf{w}(x)$ -event in E and any $\mathbf{r}(x)$ -event in F . Similarly, the absence of a race between e_{13} and e_{16} can be seen because $\text{AfterLastWr}_F(y) \cap \text{BeforeFirstRd}_G(t_1, y) = \{t_2, \ell\} \cap \{t_1, t_2\} = \{t_2\} \neq \emptyset$. To reason about the events e_5 and e_{10} , observe that $\text{AfterLastWr}_F(y) = \{t_2, \ell\}$ and $\text{BeforeFirstWr}_D(y) = \{t_1, \ell\}$, both of which have the ℓ in common. Thus, we can conclude there is no race between any pair of $\mathbf{w}(y)$ -events crossing the chunk FD .

Our reasoning also reveals the existence of HB-concurrent events. For example, we have $\text{AfterLastRd}_F(t_2, x) = \{t_2, \ell\}$, and $\text{BeforeFirstRd}_D(t_1, x) = \{t_1\}$. Since these sets are disjoint, it reveals that there are a pair of $\mathbf{r}(x)$ -events (namely, e_3 and e_7) that are HB-concurrent; it is not a HB-race because these events are not conflicting (none of e_3 and e_7 is a write event). The race between e_{10} and e_{13} can be seen as follows. $\text{AfterLastWr}_A(y) = \{t_1\}$, and $\text{BeforeFirstWr}_B(y) = \{t_2, \ell\}$. We can see that there is a cross race in chunk AB , because these two sets are disjoint.

5.2.2 Computing Before and After Sets

Our discussion in Section 5.2.1 suggests that if we manage to inductively compute the sets **AfterLastRd**, **AfterLastWr**, **BeforeFirstRd**, and **BeforeFirstWr** (Equation (5.5)) for each chunk in the grammar, then we can use Equation (5.6) to determine if a chunk has a race. In this section we present such an inductive computation for these sets.

The base case for non-terminals with rule $A \rightarrow a$, where a is an event, is straightforward:

$$\mathbf{AfterLastRd}_A(t, x) = \begin{cases} \{t\}, & \text{if } a = \langle t, \mathbf{r}(x) \rangle \\ \emptyset, & \text{otherwise} \end{cases} \quad (5.8)$$

$$\mathbf{AfterLastWr}_A(x) = \begin{cases} \{t\} & \text{if } = \langle t, \mathbf{w}(x) \rangle \\ \emptyset & \text{otherwise} \end{cases} \quad (5.9)$$

$$\mathbf{BeforeFirstRd}_A(t, x) = \begin{cases} \{t\}, & \text{if } a = \langle t, \mathbf{r}(x) \rangle \\ \emptyset, & \text{otherwise} \end{cases} \quad (5.10)$$

$$\mathbf{BeforeFirstWr}_A(x) = \begin{cases} \{t\} & \text{if } = \langle t, \mathbf{w}(x) \rangle \\ \emptyset & \text{otherwise} \end{cases} \quad (5.11)$$

We now focus on the inductive step when we have a non-terminal with rule $A \rightarrow BC$.

First consider the case of $\mathbf{AfterLastRd}_A(t, x)$, which is equal to the set $\mathbf{After}_A(e)$, where e is the last event amongst the read events $\mathbf{Reads}_A(t, x)$. If variable x is never read by thread t in the chunk A (i.e., $\mathbf{Reads}_A(t, x) = \emptyset$), we will have $\mathbf{AfterLastRd}_A(t, x) = \emptyset$. Otherwise, depending upon where the last read event e occurs in the chunk $\llbracket A \rrbracket$, we have two cases to consider. In the first case, this last read event e belongs to the chunk $\llbracket C \rrbracket$. In this, clearly, $e = \mathbf{Last}_A(\mathbf{Reads}_A(t, x))$. Observe that since $\{e' \mid e \leq_{\mathbf{HB}}^C e'\} = \{e' \mid e \leq_{\mathbf{HB}}^A e'\}$, we have $\mathbf{After}_A(e) = \mathbf{After}_C(e)$. Thus, in this case, $\mathbf{AfterLastRd}_A(t, x) = \mathbf{AfterLastRd}_C(t, x)$. The interesting case is when $\mathbf{Reads}_C(t, x)$ is empty and $\mathbf{Reads}_B(t, x) \neq \emptyset$, i.e., the last read event e belongs to the chunk B . Since $\{e' \in \llbracket B \rrbracket \mid e \leq_{\mathbf{HB}}^B e'\} \subseteq \{e' \in \llbracket A \rrbracket \mid e \leq_{\mathbf{HB}}^A e'\}$, we have $\mathbf{After}_B(e) \subseteq \mathbf{After}_A(e)$. Consider $e' \in \llbracket C \rrbracket$ such that $e \leq_{\mathbf{HB}}^A e'$. As in the discussion on cross-races in Section 5.2.1, this means there is a pair of events $e_1 \in \llbracket B \rrbracket$ and $e_2 \in \llbracket C \rrbracket$ such that $e \leq_{\mathbf{HB}}^B e_1$, $e_2 \leq_{\mathbf{HB}}^C e'$, and either (1) e_1, e_2 are events of the same thread, or (2) e_1 is a fork event and e_2 is an event of the child thread, or (3) e_1 is an event of a child thread and e_2 is a join event, or (4) e_1 is a release event and e_2 is an acquire event on the same lock. In each of these cases, e_1 witnesses the membership of some thread/lock u in $\mathbf{After}_B(e)$, and e' is HB-after the “first” event (namely e_2) of u in chunk C . The definition of what it means for

an event to be “after” the “first” event of a thread/lock u is subtle, and is key in accurately capturing the intuitions just outlined.

For a non-terminal D and thread t , define

$$\text{AfterFirst}_D(t) = \text{After}_D(\text{First}_D(\text{ThEvents}_D^{\text{join}}(t))) \quad (5.12)$$

Similarly, for a lock ℓ , define

$$\text{AfterFirst}_D(\ell) = \text{After}_D(\text{First}_D(\text{Acquires}_D(\ell))) \quad (5.13)$$

We now formalize our intuitions in the following lemma.

Lemma 5.4. Let A be a non-terminal with rule $A \rightarrow BC$ and let $e \in \llbracket B \rrbracket$. Then

$$\text{After}_A(e) = \text{After}_B(e) \cup \bigcup_{u \in \text{After}_B(e)} \text{AfterFirst}_C(u)$$

We defer the proof of Lemma 5.4 until after we finish our informal discussion about the algorithm.

Lemma 5.4 gives us the following inductive definition for $\text{AfterLastRd}_A(t, x)$.

$$\text{AfterLastRd}_A(t, x) = \begin{cases} \text{AfterLastRd}_C(t, x) & \text{if } \text{AfterLastRd}_C(t, x) \neq \emptyset \\ \text{AfterLastRd}_B(t, x) \cup \bigcup_{u \in \text{AfterLastRd}_B(t, x)} \text{AfterFirst}_C(u) & \text{otherwise} \end{cases} \quad (5.14)$$

Notice that the second expression is $\emptyset$ if $\text{AfterLastRd}_B(t, x) = \emptyset$. Due to a very similar reasoning as in for AfterLastRd_A , the inductive definition of AfterLastWr_A when $A \rightarrow BC$ is as follows:

$$\text{AfterLastWr}_A(x) = \begin{cases} \text{AfterLastWr}_C(x) & \text{if } \text{AfterLastWr}_C(x) \neq \emptyset \\ \text{AfterLastWr}_B(x) \cup \bigcup_{u \in \text{AfterLastWr}_B(x)} \text{AfterFirst}_C(u) & \text{otherwise} \end{cases} \quad (5.15)$$

To complete the formal definition of $\text{AfterLastRd}_A(x)$, we need to give an inductive definition for the sets AfterFirst . In the base case, we have that non-terminal A has the singleton

rule $A \rightarrow a$, and in this case, AfterFirst_A is given as follows:

$$\begin{aligned} \text{AfterFirst}_A(t) &= \begin{cases} \{t, \ell\} & \text{if } a = \langle t, \text{rel}(\ell) \rangle \\ \{t, t'\} & \text{if } a = \langle t, \text{fork}(t') \rangle \\ \{t\} & \text{if } a = \langle t, o \rangle \text{ and } o \notin \{\text{rel}(\cdot), \text{fork}(\cdot)\} \\ \{t'\} & \text{if } a = \langle t', \text{join}(t) \rangle \\ \emptyset & \text{otherwise} \end{cases} \\ \text{AfterFirst}_A(\ell) &= \begin{cases} \{t\} & \text{if } a = \langle t, \text{acq}\ell \rangle \\ \emptyset & \text{otherwise} \end{cases} \end{aligned} \quad (5.16)$$

Consider the inductive step, of a non-terminal A with rule $A \rightarrow BC$ and let t be some thread. If $\text{First}_A(\text{ThEvents}_A^{\text{join}}(t)) \in \llbracket B \rrbracket$ then Lemma 5.4 forms the basis of our definition. However, it is possible that the set $\text{ThEvents}_B^{\text{join}}(t)$ is empty, while $\text{First}_A(\text{ThEvents}_A^{\text{join}}(t)) \in \llbracket C \rrbracket$. In this case, $\text{AfterFirst}_A(t) = \text{AfterFirst}_C(t)$. A similar reasoning applies for a lock ℓ as well. Putting all these observations together, we get

$$\text{AfterFirst}_A(u) = \text{AfterFirst}_B(u) \cup \bigcup_{u' \in \{u\} \cup \text{AfterFirst}_B(u)} \text{AfterFirst}_C(u') \quad (5.17)$$

We now discuss the definition of $\text{BeforeFirstRd}_A(t, x)$ and $\text{BeforeFirstWr}_A(x)$ for thread t and variable x . In the base case ($A \rightarrow s$), the definition is straightforward:

$$\text{BeforeFirstRd}_A(t, x) = \begin{cases} \{t\}, & \text{if } a = \langle t, \text{r}(x) \rangle \\ \emptyset, & \text{otherwise} \end{cases} \quad (5.18)$$

$$\text{BeforeFirstWr}_A(x) = \begin{cases} \{t\} & \text{if } = \langle t, \text{w}(x) \rangle \\ \emptyset & \text{otherwise} \end{cases} \quad (5.19)$$

Let us now consider the inductive case of $\text{BeforeFirstRd}_A(t, x)$, when the rule for A is $A \rightarrow BC$; the case of $\text{BeforeFirstWr}_A(x)$ is analogous. As before, the first event $e \in \llbracket A \rrbracket$ of the kind $\langle t, \text{r}(x) \rangle$ can either belong to $\llbracket B \rrbracket$ or to $\llbracket C \rrbracket$. In the former case, we have $\text{Before}_A(e) = \text{Before}_B(e)$. On the other hand, if $e \in \llbracket C \rrbracket$, in a manner similar to the case for $\text{After}_B(e)$, we need to “compose” $\text{Before}_C(e)$ with the “before” sets associated with the *last* events of threads/locks in chunk B .

For a non-terminal D , thread t and lock ℓ ,

$$\text{BeforeLast}_D(t) = \text{Before}_D(\text{Last}_D(\text{ThEvents}_D^{\text{fork}}(t))) \quad (5.20)$$

$$\text{BeforeLast}_D(\ell) = \text{Before}_D(\text{Last}_D(\text{Releases}_D(\ell))) \quad (5.21)$$

The dual of Lemma 5.4 is the following lemma; its proof is similar to the proof of Lemma 5.4 and will be skipped.

Lemma 5.5. Let A be a non-terminal with rule $A \rightarrow BC$ and let $e \in \llbracket C \rrbracket$. Then

$$\text{Before}_A(e) = \text{Before}_C(e) \cup \bigcup_{u \in \text{Before}_C(e)} \text{BeforeLast}_B(u)$$

Using Lemma 5.5, the inductive definition of $\text{BeforeFirstRd}_A(t, x)$ is

$$\text{BeforeFirstRd}_A(t, x) = \begin{cases} \text{BeforeFirstRd}_B(t, x) & \text{if } \text{BeforeFirstRd}_B(t, x) \neq \emptyset \\ \text{BeforeFirstRd}_C(t, x) \cup \bigcup_{u \in \text{BeforeFirstRd}_C(t, x)} \text{BeforeLast}_B(u) & \text{otherwise} \end{cases} \quad (5.22)$$

Similarly, the inductive definition of $\text{BeforeFirstWr}_A(x)$ is

$$\text{BeforeFirstWr}_A(x) = \begin{cases} \text{BeforeFirstWr}_B(x) & \text{if } \text{BeforeFirstWr}_B(x) \neq \emptyset \\ \text{BeforeFirstWr}_C(x) \cup \bigcup_{u \in \text{BeforeFirstWr}_C(x)} \text{BeforeLast}_B(u) & \text{otherwise} \end{cases} \quad (5.23)$$

To complete the algorithm, we need to give the inductive definition of $\text{BeforeLast}_A(u)$ for thread/lock u . The base case follows from the definition of BeforeLast :

$$\begin{aligned} \text{BeforeLast}_A(t) &= \begin{cases} \{t, \ell\} & \text{if } a = \langle t, \text{acq}(\ell) \rangle \\ \{t, t'\} & \text{if } a = \langle t, \text{join}(t') \rangle \\ \{t\} & \text{if } a = \langle t, o \rangle \text{ and } o \notin \{\text{acq}(\cdot), \text{join}(\cdot)\} \\ \{t'\} & \text{if } a = \langle t', \text{fork}(t) \rangle \\ \emptyset & \text{otherwise} \end{cases} \\ \text{BeforeLast}_A(\ell) &= \begin{cases} \{t\} & \text{if } a = \langle t, \text{rel}(\ell) \rangle \\ \emptyset & \text{otherwise} \end{cases} \end{aligned} \quad (5.24)$$

Again the interesting case is the inductive case of a non-terminal A with rule $A \rightarrow BC$. A similar reasoning as in the case of **AfterFirst** sets gives the following definition.

$$\text{BeforeLast}_A(u) = \text{BeforeLast}_C(u) \cup \bigcup_{u' \in \{u\} \cup \text{BeforeLast}_C(u)} \text{BeforeLast}_B(u') \quad (5.25)$$

This completes the description of the **HB**-algorithm on compressed traces. Its correctness is presented in Theorem 5.2, whose proof is deferred to later.

Theorem 5.2. The algorithm presented in this section correctly detects the existence of an **HB**-race.

Let us now state the complexity of the algorithm:

Theorem 5.3. Let G be an SLP of size g representing a string σ , having $\mathcal{T}$ threads, $\mathcal{L}$ locks, and $\mathcal{V}$ variables. The inductive algorithm described in this section to detect an **HB**-race runs in time $O(g((\mathcal{T} + \mathcal{L})^2(\mathcal{L} + \mathcal{TV})))$ and uses space $O(g((\mathcal{T} + \mathcal{L})(\mathcal{L} + \mathcal{TV})))$.

Proof. Let us begin with the space requirements of the algorithm. For each non-terminal A , the algorithm maintains a bit $\text{HBRace?}(A)$, the sets $\text{AfterFirst}_A(u)$ and $\text{BeforeLast}_A(u)$ for each thread u , and the collections $\text{AfterLastRd}_A(t, x)$, $\text{AfterLastWr}_A(x)$, $\text{BeforeFirstRd}_A(t, x)$, and $\text{BeforeFirstWr}_A(x)$ for each variable x and thread t . The set $\text{AfterFirst}_A(u)$ and the set $\text{BeforeLast}_A(u)$ have at most $\mathcal{T} + \mathcal{L}$ elements. Each of the sets $\text{AfterLastRd}_A(t, x)$, $\text{AfterLastWr}_A(x)$, $\text{BeforeFirstRd}_A(t, x)$ and $\text{BeforeFirstWr}_A(x)$ have size at most $\mathcal{T} + \mathcal{L}$. Putting all of this together, the total space used is $O(g(1 + (\mathcal{T} + \mathcal{L})(\mathcal{T} + \mathcal{L}) + \mathcal{V}(\mathcal{T} + \mathcal{L}) + (\mathcal{TV})(\mathcal{T} + \mathcal{L}))) = O(g(\mathcal{T} + \mathcal{L})(\mathcal{T} + \mathcal{L} + \mathcal{V} + \mathcal{TV})) = O(g((\mathcal{T} + \mathcal{L})(\mathcal{L} + \mathcal{TV})))$.

Let us now analyze the time requirements. The algorithm computes the various sets it maintains for each non-terminal. The time to compute $\text{AfterFirst}_A(u)$ (or $\text{BeforeLast}_A(u)$) is $O((\mathcal{T} + \mathcal{L})(\mathcal{T} + \mathcal{L}))$, because it involves taking the union of at most $(\mathcal{T} + \mathcal{L} + 1)$ **AfterFirst** (or **BeforeLast**) sets, each of which has size at most $\mathcal{L} + \mathcal{T}$. Next, let us analyze the time to compute $\text{AfterLastRd}_A(t, x)$ set; the time for $\text{AfterLastWr}_A(x)$, $\text{BeforeFirstRd}_A(t, x)$, and $\text{BeforeFirstWr}_A(x)$ is the same. Let us recall the inductive definition of $\text{AfterLastRd}_A(t, x)$ for the rule $A \rightarrow BC$.

$$\text{AfterLastRd}_A(t, x) = \begin{cases} \text{AfterLastRd}_C(t, x) & \text{if } \text{AfterLastRd}_C(t, x) \neq \emptyset \\ \text{AfterLastRd}_B(tx) \cup \bigcup_{u \in \text{AfterLastRd}_B(t, x)} \text{AfterFirst}_C(u) & \text{otherwise} \end{cases} \quad (5.26)$$

In the worst case, when $\text{AfterLastRd}_C(t, x)$ is $\emptyset$, one has to take the union of a maximum

of $\mathcal{T} + \mathcal{L} + 1$ sets, each of whose size is bounded by $(\mathcal{T} + \mathcal{L})$. So the time to compute $\text{AfterLastRd}_A(t, x)$ is $O((\mathcal{T} + \mathcal{L})(\mathcal{T} + \mathcal{L}))$.

Finally, let us analyze the time to compute $\text{HBRace?}(A)$. From Equation (5.6), this requires checking the disjointness of sets of size at most $\mathcal{T} + \mathcal{L}$. This takes time $O(\mathcal{T} + \mathcal{L})$.

All these observations together give a total running time of the entire algorithm to be $O(g[(\mathcal{T} + \mathcal{L})(\mathcal{T} + \mathcal{L})^2 + \mathcal{V}(\mathcal{T} + \mathcal{L})^2 + \mathcal{TV}(\mathcal{T} + \mathcal{L})^2 + \mathcal{V}(\mathcal{T} + \mathcal{L}) + \mathcal{TV}(\mathcal{T} + \mathcal{L})]) = O(g((\mathcal{T} + \mathcal{L})^2(\mathcal{L} + \mathcal{TV})))$. QED.

Let us now turn to proving Lemma 5.4 and its dual Lemma 5.5. These proofs depend on a simple observation that we establish first.

Lemma 5.6. Let A be a non-terminal with rule $A \rightarrow BC$. Let $e_1 \in \llbracket B \rrbracket$ and $e_2 \in \llbracket C \rrbracket$. The following three statements are equivalent.

1. $e_1 \leq_{\text{HB}}^A e_2$.
2. There is $u \in \text{After}_B(e_1)$ such that first_C^u is defined and $\text{first}_C^u \leq_{\text{HB}}^C e_2$.
3. There is $u \in \text{Before}_C(e_2)$ such that last_B^u is defined and $e_1 \leq_{\text{HB}}^B \text{last}_B^u$.

where

$$\text{first}_C^u = \begin{cases} \text{First}_C(\text{ThEvents}_C^{\text{join}}(u)) & \text{if } u \in \text{Threads}_C \\ \text{First}_C(\text{Acquires}_C(u)) & \text{if } u \in \text{Locks}_C \end{cases}$$

and

$$\text{last}_B^u = \begin{cases} \text{Last}_B(\text{ThEvents}_B^{\text{fork}}(u)) & \text{if } u \in \text{Threads}_B \\ \text{Last}_B(\text{Releases}_B(u)) & \text{if } u \in \text{Locks}_B \end{cases}$$

Proof. First observe that set of events $\text{ThEvents}_C^{\text{join}}(t)$, for some $t \in \text{Threads}_C$, is totally ordered with respect to $\leq_{\text{HB}}^C$. This is because all the events in this set are either of the form $\langle t, o \rangle$ or $\langle t', \text{join}(t) \rangle$, and (i) $\leq_{\text{HB}}^C$ orders all events in a single thread thread (Definition 3.2 (a)), and (ii) also orders all events of a thread before the join event of that thread (Definition 3.2 (d)). Similarly, the events of the sets $\text{ThEvents}_B^{\text{fork}}(t)$, $\text{Acquires}_C(\ell)$ and $\text{Releases}_B(\ell)$ are also ordered totally with respect to $\leq_{\text{HB}}^B$, $\leq_{\text{HB}}^C$ and $\leq_{\text{HB}}^B$ respectively.

Let us start by showing $1 \Rightarrow 2$. As in the discussion on cross-races in Section 5.2.1, this means there is a pair of events $e_3 \in \llbracket B \rrbracket$ and $e_4 \in \llbracket C \rrbracket$ such that $e_1 \leq_{\text{HB}}^B e_3$, $e_4 \leq_{\text{HB}}^C e_2$, and one of the following four conditions holds.

- e_3 and e_4 are events performed by the same thread (say) t . Thus, $\text{ThEvents}_C^{\text{join}}(t)$ is non-empty (contains e_4), and first_C^t is defined. Since $e_1 \leq_{\text{HB}}^B e_3$, we have $t \in \text{After}_B(e_1)$. Moreover, $\text{first}_C^t \leq_{\text{HB}}^C e_4 \leq_{\text{HB}}^C e_2$, establishing the claim.
- $e_3 = \langle t', \text{fork}(t) \rangle$ and $e_4 = \langle t, o \rangle$ for some t', t, o . Thus, $\text{ThEvents}_C^{\text{join}}(t)$ is non-empty (contains e_4), and first_C^t is defined. Then by definition, we have $t \in \text{After}_B(e_1)$, and $\text{first}_C^t \leq_{\text{HB}}^C e_4 \leq_{\text{HB}}^C e_2$, which proves the claim.
- $e_3 = \langle t, o \rangle$ and $e_4 = \langle t', \text{join}(t) \rangle$, for some t, t' , and o . Thus, $\text{ThEvents}_C^{\text{join}}(t)$ is non-empty (contains e_4), and first_C^t is defined. We have $t \in \text{After}_B(e_1)$. And, $\text{first}_C^t \leq_{\text{HB}}^C e_4 \leq_{\text{HB}}^C e_2$.
- $e_3 = \langle t, \text{rel}(\ell) \rangle$ and $e_4 = \langle t, \text{acq}(\ell) \rangle$ for some t, ℓ . Thus, $\text{Acquires}_C(\ell)$ is non-empty (contains e_4), and first_C^ℓ is defined. In this case we have $\ell \in \text{After}_B(e_1)$, and $\text{first}_C^\ell \leq_{\text{HB}}^C e_4 \leq_{\text{HB}}^C e_2$, which establishes the claim as well.

Now we show that $2 \Rightarrow 3$. Let us assume that there is $u \in \text{After}_B(e_1)$ such that first_C^u is defined and $\text{first}_C^u \leq_{\text{HB}}^C e_2$. Since $u \in \text{After}_B(e_1)$, let e_3 be the event that “witnesses” the membership of $u \in \text{After}_B(e_1)$. By definition, we have $e_1 \leq_{\text{HB}}^B e_3$. First observe that no matter what u (i.e., lock/thread) and first_C^u (i.e., acq ·, event, or join ·) are, we have $u \in \text{Before}_C(e_2)$ because of first_C^u . Next, no matter what u and e_3 are, $e_3 \leq_{\text{HB}}^B \text{last}_B^u$. Thus, $e_1 \leq_{\text{HB}}^B \text{last}_B^u$ which establishes the claim.

We complete the proof by showing that $3 \Rightarrow 1$. Suppose there is $u \in \text{Before}_C(e_2)$ such that last_B^u is defined and $e_1 \leq_{\text{HB}}^B \text{last}_B^u$. Let $e_3 = \text{last}_B^u$. We have $e_1 \leq_{\text{HB}}^B e_3$. Let $e_4 \in \llbracket C \rrbracket$ be the event that witnesses $u \in \text{Before}_C(e_2)$; thus, $e_4 \leq_{\text{HB}}^C e_2$. Since $\leq_{\text{HB}}^B \subseteq \leq_{\text{HB}}^A$ and $\leq_{\text{HB}}^C \subseteq \leq_{\text{HB}}^A$, we can conclude that $e_1 \leq_{\text{HB}}^A e_3$ and $e_4 \leq_{\text{HB}}^A e_2$. Finally, since $\leq_{\text{HB}}^A$ is a transitive relation, it would be enough to establish that $e_3 \leq_{\text{HB}}^A e_4$ in order to complete the proof. Considering the various possibilities for u, e_3 , and e_4 , we will have four sub-cases like in the proof of $1 \Rightarrow 2$, and in all cases we can conclude that $e_3 \leq_{\text{HB}}^A e_4$. QED.

Proof of Lemma 5.4. We need to prove that for a non-terminal A with rule $A \rightarrow BC$ and $e \in \llbracket B \rrbracket$,

$$\text{After}_A(e) = \text{After}_B(e) \cup \bigcup_{u \in \text{After}_B(e)} \text{AfterFirst}_C(u) \quad (5.27)$$

We prove containment in each direction. Consider $u' \in \text{After}_A(e)$. Let us assume that event e' “witnesses” the membership of u' in the set $\text{After}_A(e)$. If $e' \in \llbracket B \rrbracket$ then $u' \in \text{After}_B(e)$ because of e' . On the other hand, if $e' \in \llbracket C \rrbracket$, then $e \leq_{\text{HB}}^A e'$ and then by Lemma 5.6,

there is a $u \in \text{After}_B(e)$ such that $e' \in \{e \in \llbracket C \rrbracket \mid \text{first}_C^u \leq_{\text{HB}}^C e\}$, where first_C^u is defined as in Lemma 5.6. Thus, $u' \in \text{After}_C(\text{first}_C^u) = \text{AfterFirst}_C(u)$. This establishes the containment from left to right.

Let us now consider the other direction. Observe that $\text{After}_B(e) \subseteq \text{After}_A(e)$. Consider $u' \in \bigcup_{u \in \text{After}_B(e)} \text{AfterFirst}_C(u)$. Let u_1 be such that $u_1 \in \text{After}_B(e)$ and $u' \in \text{AfterFirst}_C(u_1)$. Further, let e' be the event witnessing the membership of u' in $\text{AfterFirst}_C(u_1)$. By Lemma 5.6, $e \leq_{\text{HB}}^A e'$ and therefore, $u' \in \text{After}_A(e)$ which establishes the containment from right to left. QED.

The proof of Lemma 5.5 is similar to the proof of Lemma 5.4 and is skipped.

Let us now state the correctness of our algorithm. The proof will rely on Lemmas 5.4 and 5.5, and Theorem 5.1:

Theorem 5.4. The inductive algorithm for HB described in this section is correct. That is, for an SLP with start symbol S , $\text{HBRace?}S$ is true iff there is an HB-race (e_1, e_2) in $\llbracket S \rrbracket$.

Proof. Theorem 5.1 already establishes the correctness of the definition of the predicate $\text{HBRace?}(\cdot)$. Thus, to complete the proof of the correctness of our inductive algorithm, all we need to prove is that the sets AfterFirst , BeforeLast , AfterLastRd , AfterLastWr , BeforeFirstRd , and BeforeFirstWr are all being computed correctly by their inductive definitions. We will argue the correctness of the sets AfterFirst and AfterLastRd ; the proof of correctness for the remaining cases is similar and skipped.

Let us begin the computation of $\text{AfterFirst}_A(u)$ for a non-terminal A and lock/thread u . When A has a rule $A \rightarrow a$, where a is an event, the computation of AfterFirst_A is consistent with the definition in Equation (5.12) and Equation (5.13). In the inductive step, consider A with rule $A \rightarrow BC$. We need to prove that

$$\text{AfterFirst}_A(u) = \text{AfterFirst}_B(u) \cup \bigcup_{u' \in \{u\} \cup \text{AfterFirst}_B(u)} \text{AfterFirst}_C(u') \quad (5.28)$$

Let us begin by considering the simple case when $\text{first}_A^u = \text{undefined}$. In this case, by definition, $\text{AfterFirst}_A(u) = \emptyset$. Also, since $\text{first}_A^u = \text{undefined}$, $\text{first}_B^u = \text{undefined}$ and $\text{first}_C^u = \text{undefined}$. Therefore, inductively, $\text{AfterFirst}_B(u) = \emptyset$ and $\text{AfterFirst}_C(u) = \emptyset$, giving us that the right hand side will also evaluate to $\emptyset$. So let us now consider the case where $\text{first}_A^u = e \in \llbracket A \rrbracket$ is defined. There are two possibilities to consider. If $e \in \llbracket C \rrbracket$ (and therefore $e \notin \llbracket B \rrbracket$) then $\text{AfterFirst}_B(u) = \emptyset$, and thus, we have the following.

$$\text{AfterFirst}_B(u) \cup \bigcup_{u' \in \{u\} \cup \text{AfterFirst}_B(u)} \text{AfterFirst}_C(u') \quad (5.29)$$

$$= \emptyset \cup \text{AfterFirst}_C(u) = \text{AfterFirst}_A(u) \quad (5.30)$$

which is indeed correct. The second sub-case to consider is when $e \in \llbracket B \rrbracket$. Then, by Lemma 5.4, we have

$$\text{AfterFirst}_A(u) = \text{After}_A(e) = \text{After}_B(e) \cup \bigcup_{u' \in \text{After}_B(e)} \text{AfterFirst}_C(u') \quad (5.31)$$

$$= \text{AfterFirst}_B(u) \cup \bigcup_{u' \in \{u\} \cup \text{AfterFirst}_B(u)} \text{AfterFirst}_C(u') \quad (5.32)$$

Since $u \in \text{AfterFirst}_B(u)$, we have $\text{AfterFirst}_B(u) = \{u\} \cup \text{AfterFirst}_B(u)$, and thus,

$$\text{AfterFirst}_A(u) = \text{AfterFirst}_B(u) \cup \bigcup_{u' \in \{u\} \cup \text{AfterFirst}_B(u)} \text{AfterFirst}_C(u') \quad (5.33)$$

Let us now establish the correctness of the inductive definition of $\text{AfterLastRd}_A(t, x)$ for a non-terminal A and variable x . Again the base case of the non-terminal having rule $A \rightarrow a$, where a is an event, is clearly correct as per the definition in Equation (5.5). In the inductive step of a rule $A \rightarrow BC$, we need to prove the correctness of the following equation:

$$\text{AfterLastRd}_A(t, x) = \begin{cases} \text{AfterLastRd}_C(t, x) & \text{if } \text{AfterLastRd}_C(t, x) \neq \emptyset \\ \text{AfterLastRd}_B(t, x) \cup \bigcup_{u \in \text{AfterLastRd}_B(t, x)} \text{AfterFirst}_C(u) & \text{otherwise} \end{cases} \quad (5.34)$$

Let $e = \text{Last}_A(\text{Reads}_A(t, x))$. If e is undefined, then both the sets $\text{AfterLastRd}_C(t, x)$ and $\text{AfterLastRd}_B(t, x)$ are $\emptyset$, and so is $\text{AfterLastRd}_A(t, x)$.

Next consider the case when e is defined and belongs to the chunk C (i.e., $e \in \llbracket C \rrbracket$). Then, clearly, $\text{After}_A(e) = \text{After}_C(e) = \text{AfterLastRd}_C(t, x)$. Inductively, $\text{AfterLastRd}_C(t, x)$ will indeed be non-empty and thus $\text{AfterLastRd}_A(t, x) = \text{AfterLastRd}_C(t, x)$.

The last case is when $e \in \llbracket B \rrbracket$. The correctness then follows from Lemma 5.4. QED.

Example 5.4. We conclude this section by showing that the before and after sets given in Example 5.3 are computed correctly using our inductive characterization. We will focus on trace σ_1 and its SLP grammar in Figure 5.2. Let us consider the computation of

$\text{AfterLastWr}_C(x)$. Observe that the last $\mathbf{w}(x)$ -event in C is e_1 . Further,

$$\text{AfterLastWr}_E(x) = \{t_1, t_2\} \quad \text{AfterFirst}_F(t_1) = \emptyset \quad \text{AfterFirst}_F(t_2) = \{t_2, \ell\} \quad (5.35)$$

Here $\text{AfterFirst}_F(t_1) = \emptyset$ because there is no event of thread t_1 in F . Using the inductive definition similar to Equation (5.14), we get $\text{AfterLastWr}_C(x) = \{t_1, t_2, \ell\}$ which is correct.

Next, consider the computation $\text{BeforeFirstWr}_B(y)$. Notice that the first $\mathbf{w}(y)$ -event in B is e_{13} , which is in the chunk F . This immediately gives $\text{BeforeFirstWr}_B(y) = \text{BeforeFirstWr}_F(y) = \{t_2, \ell\}$ using a characterization similar to Equation (5.22).

5.3 LOCKSET ALGORITHM FOR COMPRESSED TRACES

Similar to our algorithm for detecting HB-races on compressed traces, we will formulate an algorithm for detecting violations of the lockset discipline on SLPs in an inductive fashion. The challenge here again is similar — violations occurring inside a chunk $\llbracket B \rrbracket$ are also violations of any other chunk that contains $\llbracket B \rrbracket$, and detecting “cross” violations is, therefore, the key challenge. In this section, we will outline these ideas in detail.

Prior to describing the details of the algorithm, it would be helpful to restate some definitions from Chapter 3 in context of trace *chunks* corresponding to non-terminals in an SLP trace chunks are essentially suffixes of well-formed traces. First, we define the set of locks held at an event inside a chunk $\llbracket A \rrbracket$ that additionally also includes locks released but not acquired:

Definition 5.1 (Locks held at an event in a chunk). Let σ be a suffix of a well-formed trace and let e be an event that appears in σ . The set of locks held at e in σ is defined as

$$\begin{aligned} \text{locksHeld}_\sigma(e) = & \{ \ell \in \text{Locks}_\sigma \mid \exists e' = \langle t, \mathbf{acq}(\ell) \rangle \in \sigma \text{ such that } e' \leq_{\text{TO}}^\sigma e \text{ and, either} \\ & \text{match}(e') \text{ does not exist in } \sigma \text{ or, } e \leq_{\text{TO}}^\sigma \text{match}_\sigma(e') \} \\ & \cup \{ \ell \in \text{Locks}_\sigma \mid \exists e' = \langle t, \mathbf{rel}(\ell) \rangle \in \sigma \text{ such that } e \leq_{\text{TO}}^\sigma e' \text{ and} \\ & \text{match}_\sigma(e') \text{ does not exist in } \sigma \} \end{aligned}$$

The sets $\text{lockSet}_\sigma(e)$ and $\text{lockSet}_\sigma(x)$ are then defined analogously for trace chunks, as in Equation (3.1) and Equation (3.2), except that they now use Definition 5.1. In the following, σ is a suffix of a well-formed trace.

$$\text{lockSet}_\sigma(e) = \begin{cases} \{\Lambda, \Lambda_t\} \cup \text{locksHeld}_\sigma(e) & \text{if } \text{op}(a) = \mathbf{r}(x), \text{ for some } x \\ \{\Lambda_t\} \cup \text{locksHeld}_\sigma(e) & \text{if } \text{op}(a) = \mathbf{w}(x), \text{ for some } x \end{cases} \quad (5.36)$$

$$\text{lockSet}_\sigma(x) = \bigcap_{e \in \text{Accesses}_\sigma} \text{lockSet}_\sigma(e) \quad (5.37)$$

Additionally, we will need the following variant of locksets that map pairs $(t, x) \in \text{teads}\sigma \times \text{Vars}_\sigma$ to subsets of $\text{Locks}_\sigma \cup \mathcal{D}$, where σ is a suffix of a well-formed trace.

$$\text{lockSet}_\sigma(t, x) = \bigcap_{\substack{e \in \text{Accesses}_\sigma \\ \text{thr}(e) = t}} \text{lockSet}_\sigma(e) \quad (5.38)$$

5.3.1 Cross Violations

We will show how to compute $\text{lockSet}_A(t, x)$ for every non-terminal A and for every pair (t, x) of thread and variable, by inducting on the non-terminals in decreasing order of their rank. Checking if $\text{lockSet}_A(x) = \bigcap_{t \in \text{teads}A} \text{lockSet}_A(t, x) = \emptyset$ then follows easily.

The base case for non-terminals with rule $A \rightarrow a$ is straightforward:

$$\text{lockSet}_A(t, x) = \begin{cases} \{\Lambda, \Lambda_t\}, & \text{if } a = \langle t, \mathbf{r}(x) \rangle \\ \{\Lambda_t\}, & \text{if } a = \langle t, \mathbf{w}(x) \rangle \\ \top, & \text{otherwise} \end{cases} \quad (5.39)$$

Now consider the inductive step for non-terminals having rules of the form $A \rightarrow BC$. To understand what $\text{lockSet}_A(t, x)$ will be, it is useful to examine what $\text{locksHeld}_A(e)$ for an event e looks like. Consider a data access event $e \in \llbracket B \rrbracket$ performed by thread t . Clearly, $\text{locksHeld}_B(e) \subseteq \text{locksHeld}_A(e)$. But are they equal? The answer turns out to be no. Suppose a lock ℓ which is released in $\llbracket C \rrbracket$ by thread t but does not have a *matching* acquire in $\llbracket A \rrbracket$ (and hence, neither in $\llbracket B \rrbracket$). Such a lock ℓ will protect all the events performed before it in $\llbracket A \rrbracket$. Thus trivially, it will enclose all the events performed by t in chunk $\llbracket B \rrbracket$. As a consequence, ℓ must be included in the set $\text{locksHeld}_A(e)$ for every event $e \in \text{Events}_B$ with $\text{thr}(e) = t$. Lock ℓ_2 in Figure 5.5 (1) illustrates this. Similarly, for an event $e \in \llbracket C \rrbracket$ performed by thread t , the set $\text{locksHeld}_A(e)$ must additionally include locks which have been acquired by thread

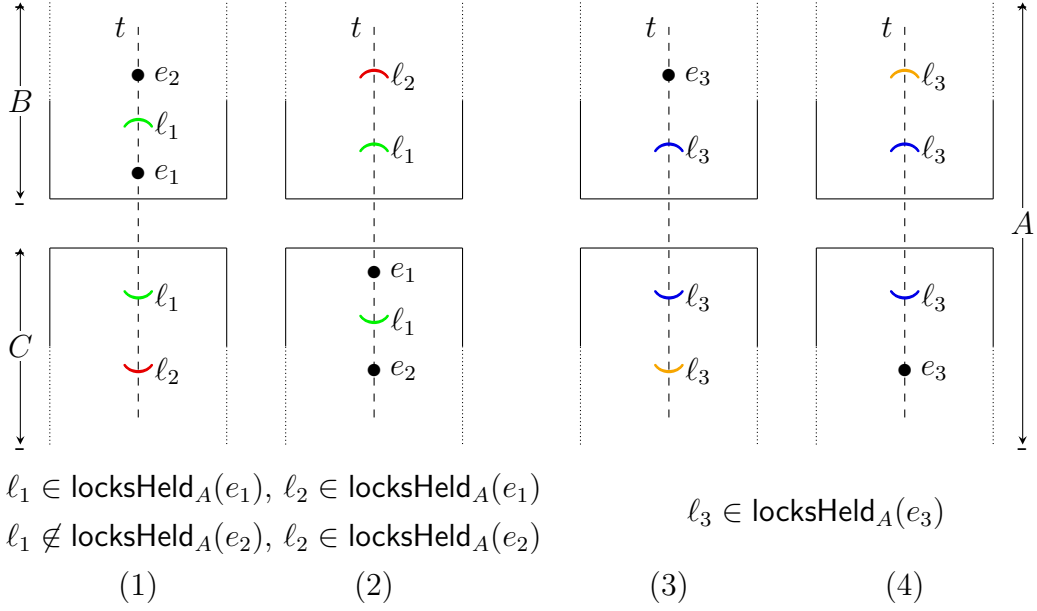


Figure 5.5: Unmatched acquire/release events protect all the events of the same thread in the neighboring chunk when not matched in the entire chunk ((1) and (2)). Re-entrant locks protect the neighboring chunk when the outermost unmatched acquire/release is unmatched ((3) and (4)). ‘ $\frown \ell_i$ ’ represents $\text{acq}(\ell_i)$, ‘ $\smile \ell_i$ ’ is $\text{rel}(\ell_i)$.

t in $\llbracket B \rrbracket$ but have not been matched in $\llbracket A \rrbracket$ (see lock ℓ_2 in Figure 5.5 (2)). However, one must be careful. A lock ℓ which was released by t in C (at event $e_{\text{rel}(\ell)}$) and whose matching acquire is in B (event $e_{\text{acq}(\ell)}$), does not affect the locks held by any event in B — for those events e in $B|_t$ which were after $e_{\text{acq}(\ell)}$, ℓ was already in $\text{locksHeld}_B(e)$, while for the events e before $e_{\text{acq}(\ell)}$, ℓ does not anyway protect e , and thus $\ell \notin \text{locksHeld}_B(e)$. This is illustrated through lock ℓ_1 in Figure 5.5 [(1) and (2)].

In the presence of re-entrant locks, we need to account for another fact. Since locks can be acquired and released multiple times, a lock that is released more times in C (by thread t) than it is acquired in B (by thread t) will protect all events of t in B , because the outermost release is still unmatched in A . The same holds for locks that have been acquired more times than they are released in C . Both these scenarios are shown in Figure 5.5 [(3) and (4)].

To formalize the above notions, we will now introduce some notation. For a non-terminal D , let us first define the number of *unmatched* acquire events of lock ℓ in thread t as

$$\text{OpenAcq}_D(t, \ell) = |\{e = \langle t, \text{acq}(\ell) \rangle \in D \mid \text{match}_D(e) \text{ is undefined}\}| \quad (5.40)$$

and the number of release events as

$$\text{OpenRel}_D(t, \ell) = |\{e = \langle t, \text{rel}(\ell) \rangle \in D \mid \text{match}_D(e) \text{ is undefined}\}| \quad (5.41)$$

Our intuitions, as discussed above, can then be captured for the more complex case of re-entrant locks as follows.

Lemma 5.7. Let A be a non-terminal with rule $A \rightarrow BC$. Let $e \in \text{Events}_B$ with $\text{thr}(e) = t$ and let $e' \in \text{Events}_C$ with $\text{thr}(e) = t'$. Then,

$$\begin{aligned} \text{locksHeld}_A(e) &= \text{locksHeld}_B(e) \cup \{\ell \mid \text{OpenRel}_C(t, \ell) > \text{OpenAcq}_B(t, \ell)\} \\ \text{locksHeld}_A(e') &= \text{locksHeld}_C(e') \cup \{\ell \mid \text{OpenAcq}_B(t', \ell) > \text{OpenRel}_C(t', \ell)\} \end{aligned}$$

Proof. We will prove the first equation. The proof is similar for the second equation.

We have event $e \in B|_t$. Let us first prove the containment

$$\text{locksHeld}_A(e) \subseteq \text{locksHeld}_B(e) \cup \{\ell \mid \text{OpenRel}_C(t, \ell) > \text{OpenAcq}_B(t, \ell)\} \quad (5.42)$$

Let $\ell \in \text{locksHeld}_A(e)$. Then, one of the following cases is possible.

1. there is an $e' = \langle t, \text{acq}(\ell) \rangle \in \llbracket A \rrbracket$ such that $e' \leq_{\text{TO}}^A e$ and e' is either not released at all in A , or $\text{match}_A(e') \in C|_t$. In either case, $\ell \in \text{locksHeld}_B(e)$.
2. there is an $e' = \langle t, \text{rel}(\ell) \rangle \in \llbracket A \rrbracket$ such that $e \leq_{\text{TO}}^A e'$, and either $\text{match}_A(e')$ is not in A or is acquired in $B|_t$. In the first case when e' is unmatched in A , we have $\text{OpenRel}_C(t, \ell) > \text{OpenAcq}_B(t, \ell)$. Otherwise, $\ell \in \text{locksHeld}_B(e)$.

In either case, ℓ is contained in the set on the right.

Now, let us prove the following containment:

$$\text{locksHeld}_A(e) \supseteq \text{locksHeld}_B(e) \cup \{\ell \mid \text{OpenRel}_C(t, \ell) > \text{OpenAcq}_B(t, \ell)\} \quad (5.43)$$

First, consider $\ell \in \text{locksHeld}_B(e)$. Clearly, $\ell \in \text{locksHeld}_A(e)$. Otherwise, consider ℓ such that $\text{OpenRel}_C(t, \ell) > \text{OpenAcq}_B(t, \ell)$. Thus, there is atleast one release event $e' = \langle t, \text{rel}(\ell) \rangle \in \llbracket C \rrbracket$ such that $\text{match}_A(e') \notin \llbracket A \rrbracket$. In this case again, $\ell \in \text{locksHeld}_A(e)$. QED.

Building on Lemma 5.7, we can now state the inductive definition of lockSet in terms of OpenAcq and OpenRel .

$$\text{lockSet}_A(t, x) = \cap \left(\begin{aligned} &(\text{lockSet}_B(t, x) \cup \{\ell \mid \text{OpenRel}_C(t, \ell) > \text{OpenAcq}_B(t, \ell)\}) \\ &(\text{lockSet}_C(t, x) \cup \{\ell \mid \text{OpenAcq}_B(t', \ell) > \text{OpenRel}_C(t', \ell)\}) \end{aligned} \right) \quad (5.44)$$

Example 5.5. Consider the SLP for σ_2 from Figure 5.3. **OpenAcq** and **OpenRel** for various non-terminals are given below.

$$\begin{aligned} \text{OpenAcq}_W(t_1, \ell) &= 1 & \text{OpenRel}_X(t_1, \ell) &= 1 \\ \text{OpenAcq}_X(t_2, \ell) &= 1 & \text{OpenRel}_Z(t_2, \ell) &= 1 \\ \text{OpenAcq}_U(t_2, \ell) &= 1 & \text{OpenRel}_V(t_2, \ell) &= 1. \end{aligned} \tag{5.45}$$

The values for all other combinations are 0. Note how the unmatched acquire in W and the unmatched release in X on thread 1 get matched in the concatenated chunk U , giving $\text{OpenAcq}_U(t_1, \ell) = \text{OpenRel}_U(t_1, \ell) = 0$. This is essentially the insight we will explore in Section 5.3.2 to inductively define **OpenAcq** and **OpenRel**.

Let us now see how **lockSet** computation takes place. First, $\text{lockSet}_W(t_1, x) = \{\Lambda, \Lambda_{t_1}\}$ since the only event of x in W is a read by thread t_1 . Also, $\text{lockSet}_X(t_1, y) = \{\ell, \Lambda_{t_1}\}$ as e_3 is protected by the (unmatched) release e_4 in X . In chunk Y , $\text{lockSet}_Y(t_2, x) = \{\Lambda, \Lambda_{t_2}\}$ and $\text{lockSet}_Y(t_2, y) = \{\Lambda_{t_2}\}$; interestingly, the locksets for Y does not reveal that both e_6 and e_7 are enclosed within the critical section of lock ℓ . On the other hand, using the inductive formulation discussed above, we can infer that $\text{lockSet}_V(t_2, y) = (\text{lockSet}_Y(t_2, y) \cup \{\ell\}) \cap \top_{\text{All-Locks}}$ which evaluates to $\{\Lambda_{t_2}, \ell\}$ as expected. The universal set $\top_{\text{All-Locks}}$ is described in Chapter 3. Again, the lock ℓ does not appear in $\text{lockSet}_U(t_1, x)$ eventhough it is unmatched in X , because it gets matched with e_2 in W . This also follows from the inductive definition of $\text{lockSet}_U(t_1, x)$.

Given Equation (5.44), our inductive formulation will be complete once we can inductively compute the functions **OpenAcq** and **OpenRel**. We describe this next.

5.3.2 Computing **OpenAcq** and **OpenRel**

The aim of this section is to give inductive definitions of the functions **OpenAcq** and **OpenRel**. The base case for non-terminals having rules of the form $A \rightarrow a$ is straightforward:

$$\text{OpenAcq}_A(t, \ell) = \begin{cases} 1, & \text{if } a = \langle t, \text{acq}(\ell) \rangle \\ 0, & \text{otherwise} \end{cases} \tag{5.46}$$

$$\text{OpenRel}_A(t, \ell) = \begin{cases} 1, & \text{if } a = \langle t, \text{rel}(\ell) \rangle \\ 0, & \text{otherwise} \end{cases} \tag{5.47}$$

In the inductive case we have a non-terminal A with production rule of the form $A \rightarrow BC$. For this case, let us first attempt to characterize the acquire events in $\llbracket A \rrbracket$ that have not been matched. Notice that if a lock is acquired (without a matching release) in the chunk $\llbracket C \rrbracket$, it would remain unmatched in the bigger chunk $\llbracket A \rrbracket$. In addition, the unmatched acquire events acquired in $\llbracket B \rrbracket$ whose matching release is not present in $\llbracket C \rrbracket$ will also contribute to the unmatched acquire events in $\llbracket A \rrbracket$. This is formalized below.

$$\text{OpenAcq}_A(t, \ell) = \text{OpenAcq}_C(t, \ell) + \max\{0, \text{OpenAcq}_B(t, \ell) - \text{OpenRel}_C(t, \ell)\} \quad (5.48)$$

Notice the use of the max operator in Equation (5.48). If the quantity $\text{OpenAcq}_B(t, \ell) - \text{OpenRel}_C(t, \ell)$ is negative, then there are more unmatched $\text{rel}(\ell)$ -events in $C|_t$, which should be accounted for in $\text{OpenRel}_A(t, \ell)$, instead of affecting the contribution of $\llbracket C \rrbracket$ towards the unmatched acquire events of $A|_t$.

Similar reasoning gives the inductive formulation for OpenRel

$$\text{OpenRel}_A(t, \ell) = \text{OpenRel}_B(t, \ell) + \max\{0, \text{OpenRel}_C(t, \ell) - \text{OpenAcq}_B(t, \ell)\} \quad (5.49)$$

This completes the description of our algorithm for computing locksets and checking violations of lockset discipline for compressed traces.

The algorithm presented here is correct, and we state this in Theorem 5.5. We first note the following simple but important observation, which is similar to Proposition 5.1, and informally states that, the presence/absence of lockset violations in a trace σ , does not get affected by the “context” in which σ is placed.

Proposition 5.2. Let σ be a trace and x be a variable accessed in σ . The following two statements are equivalent

1.

$$\bigcap_{t \in \text{eads}\sigma} \text{lockSet}_\sigma(t, x) = \emptyset$$

2. For any trace $\sigma' = \sigma_1\sigma\sigma_2$,

$$\bigcap_{t \in \text{eads}(\sigma')} \text{lockSet}_{\sigma'}(t, x) = \emptyset$$

Thus if a non-terminal different from the start symbol S of the grammar G reports a violation, there will be a violation in S too. Finally, the correctness of the inductive algorithm

will be complete if we prove that each of the inductive formulations in the previous section are correct:

Theorem 5.5. The algorithm presented in this section correctly detects violations of lockset discipline.

Proof. We will prove that all the inductive formulations are correct.

First, let us prove the correctness of the inductive formulation for $\text{OpenAcq}_A(t, \ell)$. We will skip the proof for $\text{OpenRel}_A(t, \ell)$ because it is similar.

Let us fix some notation before we begin. For non-terminal D , thread t and lock ℓ , let us define the following sets.

$$\text{unmatchedAcq}_D^{(t, \ell)} = \{e = \langle t \rangle \text{acq}(\ell) \in D \mid \text{match}(e) \notin \llbracket D \rrbracket\} \quad (5.50)$$

$$\text{unmatchedRel}_D^{(t, \ell)} = \{e = \langle t \rangle \text{rel}(\ell) \in D \mid \text{match}(e) \notin \llbracket D \rrbracket\}. \quad (5.51)$$

Then, $\text{OpenAcq}_D(t, \ell) = |\text{unmatchedAcq}_D^{(t, \ell)}|$ and $\text{OpenRel}_D(t, \ell) = |\text{unmatchedRel}_D^{(t, \ell)}|$.

In the base case, we have the rule $A \rightarrow a$. The set $\text{unmatchedAcq}_A^{(t, \ell)}$ is non-empty only if $a = \langle t \rangle \text{acq}(\ell)$ is an acquire event in which case $|\text{unmatchedAcq}_A^{(t, \ell)}| = 1$. In all other cases $|\text{unmatchedAcq}_A^{(t, \ell)}| = 0$. This justifies the base case.

In the inductive case, we have the rule $A \rightarrow BC$. Then, the set $\text{unmatchedAcq}_A^{(t, \ell)}$ can be expressed as a disjoint union $S_B \cup S_C$ ($S_B \cap S_C = \emptyset$), where $S_B = \{e \in \text{unmatchedAcq}_A^{(t, \ell)} \mid e \in B|_t\}$, and $S_C = \{e \in \text{unmatchedAcq}_A^{(t, \ell)} \mid e \in C|_t\}$. First, note that for every event $e \in S_C$, $\text{match}_e \notin A$ and thus $\text{match}_e \notin C$. Thus we have $S_C \subseteq \text{unmatchedAcq}_C^{(t, \ell)}$. Next, see that every event $e \in \text{unmatchedAcq}_C^{(t, \ell)}$ is clearly unmatched even in A , and thus belongs to $\text{unmatchedAcq}_A^{(t, \ell)}$, and thus to S_C . This gives $\text{unmatchedAcq}_C^{(t, \ell)} \subseteq S_C$. Hence, $S_C = \text{unmatchedAcq}_C^{(t, \ell)}$. The cardinality of $S_C = \text{unmatchedAcq}_C^{(t, \ell)} = \text{OpenAcq}_C(t, \ell)$. Next, consider the set $S_B^1 = \{e \in \text{unmatchedAcq}_B^{(t, \ell)} \mid \text{match}(e) \in \text{unmatchedRel}_C^{(t, \ell)}\}$. Also, let $S_B^2 = \text{unmatchedAcq}_B^{(t, \ell)} \setminus S_B^1$. Then, every event $e \in S_B^2$ is unmatched in A , and thus belongs to $\text{unmatchedAcq}_A^{(t, \ell)}$ and hence to $\text{unmatchedAcq}_B^{(t, \ell)}$. This means $S_B^2 \subseteq S_B$. Also, consider an event $e \in S_B$. Since it is unmatched in A , it must also be unmatched in B , and thus $e \in \text{unmatchedAcq}_B^{(t, \ell)}$. Also, it does not have a matching release in C and thus, $e \notin S_B^1$. Thus, $e \in S_B^2$. Clearly, $S_B = S_B^2$. Also, the cardinality of S_B^2 is just $\max\{0, \text{OpenAcq}_B(t, \ell) - \text{OpenRel}_C(t, \ell)\}$.

Since $S_B \cap S_C = \emptyset$, we have the desired result.

Let us now prove the correctness of the inductive formulation of lockSet . The base case $A \rightarrow a$ indeed matches Definition 5.1.

We consider the case $A \rightarrow BC$. If $\text{Accesses}_A(t, x) = \emptyset$, both $\text{Accesses}_B(t, x) = \emptyset$ and $\text{Accesses}_C(t, x) = \emptyset$. In this case, using the inductive hypothesis, $\text{lockSet}_B(t, x) = \top$ and $\text{lockSet}_C(t, x) = \top$. Thus, $\text{lockSet}_A(t, x)$ evaluates to $\top$, which is correct.

Now consider the case when $\text{Accesses}_A(t, x)$ is non-empty. If $\text{Accesses}_B(t, x)$ is empty, then inductively, $\text{lockSet}_B(t, x) = \top$ and $\text{lockSet}_A(t, x)$ evaluates to $\text{lockSet}_C(t, x)$. This is correct since $\text{Accesses}_A(t, x) = \text{Accesses}_C(t, x)$ and $\text{Wr}(A|_t) = \text{Wr}(C|_t)$ in this case. The case when $\text{Accesses}_C(t, x)$ is similar. Next, consider the case when $\text{Accesses}_B(t, x) \neq \emptyset$ and $\text{Accesses}_C(t, x) \neq \emptyset$. By the inductive hypothesis, $\Lambda_t \in \text{lockSet}_B(t, x)$ and $\Lambda_t \in \text{lockSet}_C(t, x)$. Clearly, $\Lambda_t \in \text{lockSet}_A(t, x)$, as expected. Now we have two cases. First, the case when $\text{Accesses}_A(t, x) = \text{Reads}_A(t, x)$, that is, there is no write event to x in $A|_t$. Then, $\text{Accesses}_B(t, x) = \text{Reads}_B(t, x)$ and $\text{Accesses}_C(t, x) = \text{Reads}_C(t, x)$. By induction, $\Lambda \in \text{lockSet}_B(t, x)$ and $\Lambda \in \text{lockSet}_C(t, x)$. Thus, $\Lambda \in \text{lockSet}_A(t, x)$, which is correct. Otherwise, at least one of $\text{lockSet}_B(t, x)$ and $\text{lockSet}_C(t, x)$ does not have Λ . As a result, $\Lambda \notin \text{lockSet}_A(t, x)$. Finally,

$$\text{RealLocks}_A^{(t,x)} = \bigcap_{e \in \text{Accesses}_A(t,x)} \text{locksHeld}_A(e) = \bigcap_{e \in \text{Accesses}_B(t,x)} \text{locksHeld}_A(e) \cap \bigcap_{e \in \text{Accesses}_C(t,x)} \text{locksHeld}_A(e) \quad (5.52)$$

By Lemma 5.7, for an event $e \in B$, we have

$$\text{locksHeld}_A(e) = \text{locksHeld}_B(e) \cup \{\ell \mid \text{OpenRel}_C(t, \ell) > \text{OpenAcq}_B(t, \ell)\} \quad (5.53)$$

Thus,

$$\begin{aligned} \bigcap_{e \in \text{Accesses}_B(t,x)} \text{locksHeld}_A(e) &= \bigcap_{e \in \text{Accesses}_B(t,x)} \left(\text{locksHeld}_B(e) \cup \{\ell \mid \text{OpenRel}_C(t, \ell) > \text{OpenAcq}_B(t, \ell)\} \right) \\ &= \left(\bigcap_{e \in \text{Accesses}_B(t,x)} \text{locksHeld}_B(e) \right) \cup \{\ell \mid \text{OpenRel}_C(t, \ell) > \text{OpenAcq}_B(t, \ell)\} \\ &= \text{RealLocks}_B^{(t,x)} \cup \{\ell \mid \text{OpenRel}_C(t, \ell) > \text{OpenAcq}_B(t, \ell)\} \end{aligned} \quad (5.54)$$

Similarly,

$$\bigcap_{e \in \text{Accesses}_C(t,x)} \text{locksHeld}_A(e) = \text{RealLocks}_C^{(t,x)} \cup \{\ell \mid \text{OpenAcq}_B(t, \ell) > \text{OpenRel}_B(t, \ell)\} \quad (5.55)$$

Thus, we have

$$\text{lockSet}_A(t, x) = \cap \begin{array}{l} (\text{lockSet}_B(t, x) \cup \{\ell \mid \text{OpenRel}_C(t, \ell) > \text{OpenAcq}_B(t, \ell)\}) \\ (\text{lockSet}_C(t, x) \cup \{\ell \mid \text{OpenAcq}_B(t', \ell) > \text{OpenRel}_C(t', \ell)\}) \end{array} \quad (5.56)$$

QED.

Let us now analyze the time and space complexity of the algorithm just described.

Theorem 5.6. Let G be an SLP of size g representing a string σ , having $\mathcal{T}$ threads, $\mathcal{L}$ locks, and $\mathcal{V}$ variables. Let r be the maximum number of times a thread acquires a lock without releasing it, in the trace generated by G . The inductive algorithm described in this section to detect violations of lockset discipline runs in time $O(g \cdot \mathcal{T} \cdot \mathcal{L} \cdot (\log r + \mathcal{V}))$ and uses space $O(g \cdot \mathcal{T} \cdot \mathcal{L} \cdot (\log r + \mathcal{V}))$.

Proof. Let us first analyze the space complexity. For every non-terminal, we need to store $\text{lockSet}(t, x)$ having size atmost $\mathcal{L}$, for every pair of thread t and variable x . For every non-terminal, we also need to store integers $\text{OpenAcq}(t, \ell)$ and $\text{OpenRel}(t, \ell)$, for every pair of lock ℓ and thread t . The size of every such integer is bounded by $O(\log r)$. The total space usage per non-terminal therefore is $O(\mathcal{T}\mathcal{L}\mathcal{V} + \mathcal{T}\mathcal{L}\log r)$, and the overall space requirement is $O(g \cdot \mathcal{T} \cdot \mathcal{L} \cdot (\log r + \mathcal{V}))$.

Let us now evaluate the time complexity. For every non-terminal, the algorithm detects lockset violation by checking if for some variable x , the intersection of atmost $\mathcal{T}$ sets, each of size $\mathcal{L}$ is empty. This takes time $O(\mathcal{T}\mathcal{L}\mathcal{V})$ per non-terminal. To compute lockSet , the algorithm, for every thread t and variable x performs set operations on constantly many sets of size $O(\mathcal{L})$. This takes time $O(\mathcal{T}\mathcal{L}\mathcal{V})$ for every non-terminal. Lastly, to compute OpenAcq and OpenRel , the algorithm performs arithmetic operations on constantly many integers (of size $O(\log r)$) for every thread and lock, taking time $O(\mathcal{T}\mathcal{L}\log r)$ per non-terminal. The overall time complexity therefore is $O(g \cdot \mathcal{T} \cdot \mathcal{L} \cdot (\log r + \mathcal{V}))$. QED.

5.4 EVALUATION

In order to gauge the effect of compression on the size of traces, and the subsequent effect on time taken to analyze these compressed traces for races, we conducted experiments on a large variety of benchmarks and evaluated our algorithms empirically. In this section, we describe the details of our implementation and experimental setup, and analyze the results of these experiments.

5.4.1 Implementation and Setup

Implementation. Our algorithms for detecting races on compressed traces, discussed in Section 5.2 and Section 5.3 have been implemented in our tool ZIPTRACK, which is publicly available at [79]. ZIPTRACK is written primarily in Java and analyzes traces generated by Java programs. ZIPTRACK firsts collects trace logs as sequence of events, which include read/write to memory locations, acquire/release of locks, and join/fork of threads. For this, we use the logging library provided by the commercial tool RVPredict [85]. After having generated the trace logs, ZIPTRACK calls the Sequitur algorithm (available at [77]) to compress these traces as straight line programs (see Section 5.1). ZIPTRACK then analyzes these SLPs to detect the presence of HB races and lockset discipline violations.

Optimizations. The SLPs generated using the Sequitur algorithm are not strictly CNF grammars; production rules in the grammar can have length > 2 as well. This is similar to the grammar shown in Figure 5.2, where both the non-terminals F and D have production rules of length 4. For detecting an HB-race on SLPs, ZIPTRACK employs the following optimizations that rely on existence of such long production rules. For a rule of the form $A \rightarrow a_1 a_2 \cdots a_k$, where each of $a_1, \dots, a_k$ are terminals, our tool ZIPTRACK uses a slight modification of the basic HB vector clock algorithm and uses the vector clock values to (i) determine if $\text{HBRace?}(A)$ holds, and (ii) compute the various sets associated with A (such as AfterLastRd_A , BeforeFirstRd_A , etc.). Next, for production rules where the right hand side has both terminals and non-terminals and has long contiguous sequences (or substrings) of terminals, we introduce new production rules in the grammar, with fresh non-terminals corresponding to these long sequences. For example, for a rule of the form $A \rightarrow b_1 \cdots b_k C d_1 \cdots d_m$, where b_i s and d_i s are terminals, we will introduce two new non-terminals B and D , with production rules $B \rightarrow b_1 \cdots b_k$ and $D \rightarrow d_1 \cdots d_m$, and replace the production rule of A by $A \rightarrow BCD$. This allows us to better exploit the vector-clock optimization.

Table 5.1: Columns 1-5 describe the benchmarks and traces. Columns 6 and 7 describe the size of the compressed traces and the compression ratios achieved. Columns 8-10 describe the performance of various HB-race detection algorithms on uncompressed traces. Column 11-12 reports the performance of ZipTRACK’s HB race detection on compressed traces and the resulting speedup achieved. Column 13, 14 and 15 report the performance of respectively Eraser’s lockset algorithm on uncompressed traces, ZipTRACK’s performance on compressed traces, and the resulting speedup achieved.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Program	Events	Threads	Locks	Memory	Grammar	Compr.	DJIT+	F.TRACK	Goldi.	Compr.	Speedup	Eraser	Compr.	Speedup
account	130	4	3	41	107	1.21	6	5	4	4	1	3	3	1
airline	137	4	0	44	132	1.04	8	4	5	6	0.67	2	1	2
array	47	3	2	30	47	1	5	4	4	3	1.33	3	1	3
boundedbuffer	337	2	2	63	194	1.74	8	8	2	18	0.11	3	2	1.5
bubblesort	4.2K	10	2	167	3.3K	1.29	14	13	12	92	0.13	2	4	0.5
bufwriter	11.8M	6	1	56	293	40238	20s	15.5s	36.3s	6	2600	1	4	0.25
critical	55	4	0	30	55	1	3	4	3	5	0.6	2	1	2
mergesort	3028	5	3	621	2795	1.08	5	7	5	13	0.38	3	6	0.5
pingpong	146	4	0	51	135	1.08	10	9	9	3	3	2	1	2
modlyn	164K	3	2	1197	88K	1.86	53	60	57	6	8.83	2	2	1
montecarlo	7.2M	3	3	876K	6.1M	1.18	317	271	302	87	3.11	300	1	300
raytracer	16.2K	3	8	3879	14.6K	1.11	58	32	32	315	0.1	25	133	0.19
derby	1.3M	4	1112	186K	735K	1.83	1006	1011	26s	592	1.7	848	1	21000
eclipse	90.6M	19	8300	11.2M	42.5M	2.13	34.6s	31.5s	3776s	17.4s	1.8	21737	1	21737
ftpsrver	49K	11	301	5461	30K	2.13	49	44	91	23	1.9	34	1	34
jigsaw	3M	13	280	103K	908K	3.37	2432	2309	1888	195	9.7	12	4	3
lusearch	216M	7	118	5.2M	66.6K	3.25	1392	968	700	7	100	814	2	407
xalan	122M	6	2491	4.4M	71M	1.7	5183	3008	3709	109	27.6	2779	1	2779

Setup and Benchmarks. Our experiments were conducted on an 8-core 2.6GHz 64-bit Intel Xeon(R) Linux machine, with 30GB heap space. To compare against Happens-Before and LockSet based analysis on uncompressed traces, we use RAPID [86], which implements the standard DJIT+ [48] vector clock algorithm, epoch optimizations like inFASTTRACK [52], the Goldilocks algorithm [57], and Eraser’s lockset algorithm [42], as described in [48]. Our evaluation benchmarks (Column 1 in Table 5.1) are carefully chosen with the goal of being comprehensive, and have been primarily derived from [44]. The first set of small-sized (LOC $\sim$ 50-300) benchmarks (**account** to **pingpong**) is derived from the IBM Contest benchmark suite [87]. The second set of medium sized (LOC $\sim$ 3K) benchmarks (**moldyn** to **raytracer**) is derived from the Java Grande Forum benchmark suite [88]. The third set (**derby** to **xalan**) of benchmarks (LOC $\sim$ 30K-500K) comes from the DaCaPo benchmark suite (version 9.12) [89] and large real world software including Apache FTPServer, W3C Jigsaw web server and Apache Derby. Columns 3, 4 and 5 in Table 5.1 report the number of threads, locks and variables in the traces generated from the corresponding programs in Column 1.

5.4.2 Results

Compression Ratio. To analyze the effect of compression on the size of traces, consider the compression ratios (ratio of the size of the original trace and the size of the grammar representation) shown in Column 7 in Table 5.1. The compression ratios are not significant for the small and medium sized benchmarks, barring **boundedbuffer** (compression ratio = 1.74), **moldyn** (compression ratio = 1.86) and the most notable **bufwriter** (compression ratio $>$ 40,000). The compression ratios for the large benchmarks are impressive; as large as 3.25. This can be attributed to the fact that in large executions, the large amount of redundancies make them amenable to larger compression. Despite smaller lines of code in the source code of **bufwriter**, the size of the execution observed is quite large, and thus the excellent compression ratio.

HB race detection. Columns 8, 9 and 10 in Table 5.1 represent the time taken to detect the presence of an HB race by respectively, DJIT+, FASTTRACK and Goldilocks. Column 11 denotes the time taken by our HB race detection algorithm for analyzing the traces compressed as SLPs and Column 12 reports the speedup achieved over the best of the three values in Columns 8, 9 and 10.

First, in the smaller examples (**account** - **pingpong**), the speed-up is not significant for most examples. This can be attributed to the low compression ratios, and significant initial set-up

times. In particular, the **bubblesort** example has a significant slow-down. One noteworthy small example that shows the power of compression is **bufwriter** where the compression ratio and the resulting speedup for race detection is very high ($> 2500x$).

For the medium sized examples, the compression ratios range in $1.1 - 1.86$. The speedup for **moldyn** and **montecarlo** is about $3 - 8x$, while for **raytracer**, we encounter a large slowdown. A possible explanation for the degraded performance in both **bubblesort** and **raytracer** is that, while the first race pair (e_1, e_2) occurs very early in the uncompressed trace, the SLP generated is such that, in order to discover any race, the entire grammar needs to be processed.

The performance improvements for the large benchmarks are noteworthy and the speed ups shoot to the order of $100x$. The **FastTrack** vector clock algorithm [52] is the gold standard for detecting HB races, and our evaluation indicates that analysis on compressed traces beats the advantages offered by vector-clocks and further epoch-like optimizations. In fact our algorithm is, in spirit, closer to the Goldilocks algorithm, for which the performance degradation deeply intensifies on larger benchmarks (also noted before in [52]). The speedups (over **FASTTRACK**) achieved by our approach, despite this similarity, must be attributed to the non-trivial compression ratios achieved. Overall, the average speed-up is about $2.9x$ over **FASTTRACK**, and around $200x$ over the Goldilocks algorithm.

Lockset violation detection. Columns 13 and 14 denote the time for detecting lockset violations on uncompressed and compressed traces respectively. Since, the compression on smaller examples is not large, we can observe that the speedup in such examples is not extraordinary. However, there is little or almost no slowdown. For the medium and large examples, **ZIPTRACK** detects violations of lockset discipline on compressed traces much faster than on uncompressed traces. In fact, the speed-ups shoot upto more than $20,000x$, and the time taken is almost always of the order of a few milliseconds. The average speed-up achieved over the Eraser algorithm is around $173x$.

Clearly, these large real-world examples illustrate the benefit of compression; compression can be leveraged not only for smaller storage spaces, but also for a more efficient race detection analysis.

CHAPTER 6: ENHANCING THE PREDICTIVE POWER OF HAPPENS-BEFORE BASED RACE DETECTION

The HB partial order is the basis of the most popular sound technique for race detection. There are two reasons for the popularity of HB based race detection. First, because it is sound, it does not report false positives. Low false positive rates are critical for the wide-spread use of debugging techniques [53, 90]. Second, these techniques have the lowest overhead among other sound race detection techniques [13, 34, 43, 44]. Many improvements [48, 52, 57] to the original vector clock algorithm [50] have helped reduce the overhead even further.

However, HB-based dynamic analysis tools suffer from some drawbacks. Even though HB is a sound technique, its soundness guarantee is only limited to the *first* pair of unordered conflicting events; a formal definition of “first” unordered pair is given later in the chapter. Thus, a trace may have many HB-unordered pairs of conflicting events (or HB-races) that are not predictable races. To see this, consider the example program and trace shown in Figure 6.1. The trace corresponds to first executing the statement of thread t_1 , before executing the statements of thread t_2 . The statement $y := x + 5$ requires first reading the value of x (which is 0) and then writing to y . Recall that HB orders (i) two events performed by the same thread, and (ii) synchronization events performed by different threads, in the order in which they appear in the trace. In this trace since there are no synchronization events, both (e_1, e_4) and (e_2, e_3) are in HB-race. Observe that while e_2 and e_3 can be simultaneously σ_1 -enabled (in the correct reordering $\rho = e_1$ of σ_1), there is no trace of the program (and thus no correct reordering) where e_1 and e_4 appear are enabled at the same time. Thus, even though the events e_1 and e_4 are unordered by HB, they do not constitute a data race. In other words, not all HB-races are guaranteed to be a predictable data race.

As a consequence, developers typically fix the first race discovered, re-run the program and the dynamic race detection algorithm, and repeat the process until no races are discovered. However, this iterative approach suffers from many disadvantages. First, running race detection algorithms can be expensive [90], and so running them many times is a significant overhead. Second, even though only the first HB-race is guaranteed to be a real race, it doesn’t mean that it is the *only* HB-race that is real. Consider the example shown in Figure 6.2. In the trace σ_2 (shown in Figure 6.2b), both pairs (e_1, e_4) and (e_2, e_3) are in HB-race. The prefix trace e_1 is a witness to the data race (e_2, e_3) . But (e_1, e_4) is also a valid data race. This can be seen by first executing $y := 1$; in thread t_2 , after which both the $\langle t_1, \mathbf{r}(x) \rangle$ and $\langle t_2, \mathbf{w}(x) \rangle$ are simultaneously enabled. The approach of fixing the first race, and then re-executing and performing race detection, not only unnecessarily ignores the race (e_1, e_4) ,

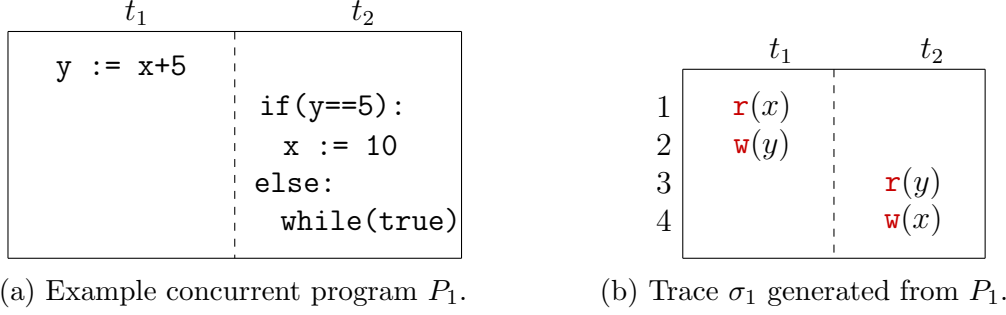


Figure 6.1: Concurrent program P_1 and its sample execution σ_1 . Initially $x = y = 0$.

but it might miss it completely because (e_1, e_4) might not show up as a **HB**-race in the next execution due to the inherent nondeterminism when executing multi-threaded programs. As a result, most practical race detection tools including THREADSANITIZER [53], Helgrind [55] and FASTTRACK [52] report more than one race, even if those races are likely to be false, to give software developers the opportunity to fix more than just the first race. In Section 6.1, we illustrate this observation on four practical dynamic race detection tools based on the happens-before partial order. Each of these tools resort to naïvely reporting races beyond the first race and produce false positives as a result.

In this chapter, we explore the question of detecting multiple data races using the **HB** partial order. to mimic the software developer’s strategy in using **HB**-race detectors — every time a race is discovered, force an order between the two events constituting the race and then analyze the subsequent events. This ensures that the **HB** soundness theorem then applies to the *next* race discovered, and so on. Such an algorithm can be proved to only discover valid data races. For example, in trace σ_1 (Figure 6.1), after discovering the race (e_2, e_3) assume that the events e_2 and e_3 are ordered when analyzing events after e_3 in the trace. By this algorithm, when we process event e_4 , we will conclude that (e_1, e_4) are not in

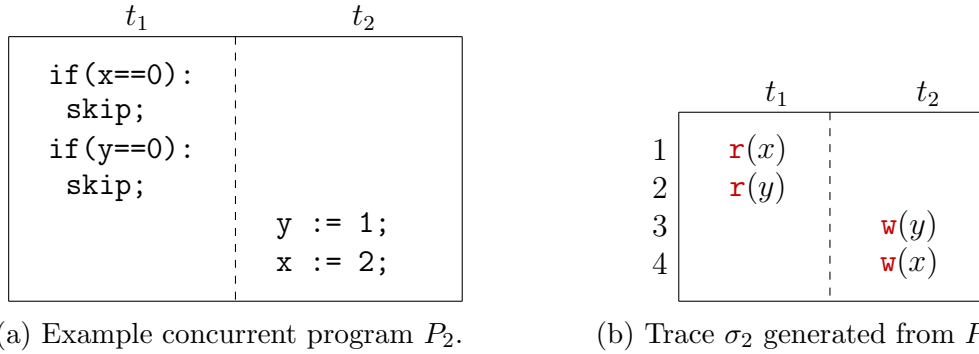


Figure 6.2: Concurrent program P_2 and its sample execution σ_2 . Initially $x = y = 0$.

race because e_1 comes before e_2 , e_2 has been force ordered before e_3 , and e_3 is before e_4 , and so e_1 is ordered before e_4 . However, force ordering will miss valid data races present in the trace. Consider the trace σ_2 from Figure 6.2. Here the force ordering algorithm will only discover the race (e_2, e_3) and will miss (e_1, e_4) which is a valid data race. Another approach is to search for a reordering of the events in the trace that respects the data dependencies amongst the read and write events, and the effect of synchronization events like lock acquires and releases as in SAT/SMT solving based approaches [44], which does not scale well as is, and can miss simple races when restricted to small ‘windows’ of the trace.

To tackle this problem, we present a new partial order on events in an execution that we call *schedulable happens-before* (SHB) to address these challenges. SHB is a *strengthening* of HB— some HB unordered events, will be ordered by SHB. However, the first HB race (which is guaranteed to be a real data race by the soundness theorem for HB) will also be SHB unordered. Further, every race detected using SHB is a predictable race. In addition, we prove that, not only does SHB discover every race found by the naïve force ordering algorithm and more (for example, SHB will discover both races in Figure 6.2), it will detect *all HB-schedulable* races. The fact that SHB detects precisely the set of HB-schedulable races, we hope, will make it popular among software developers because of its enhanced predictive power per trace and the absence of false positives.

We then present a simple vector clock based algorithm for detecting all SHB races. Because the algorithm is very close to the usual HB vector clock algorithm, it has a low overhead. We also show how to adapt existing improvements to the SHB algorithm, like the use of *epochs* [52], into the SHB algorithm to lower overhead. We believe that existing HB-based detectors can be easily modified to leverage the greater power of SHB-based analysis. We have implemented our SHB algorithm in RAPID [86] and analyzed its performance on standard benchmarks. Our experiments demonstrate that (a) many SHB unordered conflicting events may not be valid data races, (b) there are many valid races missed by the naïve force ordering algorithm, (c) SHB based analysis poses only a little overhead as compared to HB based vector clock algorithm, and (d) improvements like the use of epochs, are effective in enhancing the performance of SHB analysis. Our results in this chapter has previously been published in [14] and we closely follow this presentation here. The rest of the chapter is organized as follows. In Section 6.1 we first illustrate using simple programs how existing race detectors report false positives by incorrectly using the HB partial order to report races beyond the first one. Section 6.2 discusses the limitations of existing HB based race detection. In Section 6.3, we introduce the partial order SHB, present an exact characterization of schedulable races using this partial order, and illustrate the SHB using some examples. In Section 6.4, we describe a vector clock algorithm for detecting schedulable races based

```

public class Test extends Thread{
    static int x, y;
    public int id;
    @Override
    public void run() {
        if(id == 1){
            y = x + 5;
        }
        if(id == 2){
            if ( y==5 ){
                x = 10;
            }
            else{
                while(true);
            }
        }
    }
}

public static void main(String args[])
    throws Exception {
    final Test t1 = new Test();
    final Test t2 = new Test();
    t1.id = 1;
    t2.id = 2;
    t1.start();
    t2.start();
    t1.join();
    t2.join();
}

```

(a) Multi-threaded Java program

```

#include <pthread.h>
#include <stdio.h>

int x;
int y;

void *Thread1(void *a) {
    y = x + 5;
    return NULL;
}

void *Thread2(void *a) {
    if (y == 5){
        x = 10;
    }
    else{
        while(true){}
    }
    return NULL;
}

int main() {
    x = 0;
    y = 0;
    pthread_t t[2];
    pthread_create(
        &t[0], NULL, Thread1, NULL
    );
    pthread_create(
        &t[1], NULL, Thread2, NULL
    );
    pthread_join(t[0], NULL);
    pthread_join(t[1], NULL);
}

```

(b) Multi-threaded C program

Figure 6.3: Concurrent program from Figure 6.1a

on SHB. We then show how to incorporate epoch-based optimizations to this vector clock algorithm. Section 6.5 describes our experimental evaluation.

6.1 FALSE POSITIVES REPORTED BY EXISTING PRACTICAL DYNAMIC RACE DETECTION TOOLS

We evaluate existing tools that use happens-before based race detection to check if they report false (unschedulable) races.

We use the following program in Figure 6.1a to assess if these tools guarantee soundness. The Java source code for this program is in Figure 6.3a and the C source code is in Figure 6.3b

6.1.1 FASTTRACK

When run on FASTTRACK, the Java program in Figure 6.3a, FASTTRACK produces the output shown in Figure 6.4. That is, FASTTRACK flags both the fields `Test.y` and `Test.x`

```

##
## =====
## HappensBefore Error
##
##      Thread: 2
##      Blame: Test.y_I
##      Count: 1      (max: 100)
##      Guard State: [(0:0) (1:1)]
##      Class: Test
##      Field: null.Test.y_I
##      Locks: []
##      Prev Op: write-by-thread-1
##      Prev Op CV: [(0:0) (1:1)]
##      Cur Op: read
##      Cur Op CV: [(0:1) (1:0) (2:1)]
##      Stack: Use -stacks to show stacks...
## =====
##
##
## =====
## HappensBefore Error
##
##      Thread: 2
##      Blame: Test.x_I
##      Count: 1      (max: 100)
##      Guard State: [(0:0) (1:1)]
##      Class: Test
##      Field: null.Test.x_I
##      Locks: []
##      Prev Op: read-by-thread-1
##      Prev Op CV: [(0:0) (1:1)]
##      Cur Op: write
##      Cur Op CV: [(0:1) (1:0) (2:1)]
##      Stack: Use -stacks to show stacks...
## =====
##

```

Figure 6.4: Output of FASTTRACK on Java program in Figure 6.3a

to be in an HB-race. However, in any execution in `Test.x` both written and read by different threads, the read always occurs before the write (with a read on `Test.y` separating them).

6.1.2 ThreadSanitizer

We run the C program in Figure 6.3b on ThreadSanitizer (shipped with LLVM). ThreadSanitizer also reports a race on both global locations `x` and `y` (see output in Figure 6.5).

6.1.3 Helgrind

Helgrind is a data race detector integrated with Valgrind. We analyzed the C program in Figure 6.3b and show its output in Figure 6.6. Helgrind also incorrectly reports a race on both `x` and `y`.

```

=====
WARNING: ThreadSanitizer: data race (pid=76224)
  Read of size 4 at 0x0001030ba074 by thread T2:
    #0 Thread2(void*) simple_race.cc:13 (a.out:x86_64+0x100000d1e)

  Previous write of size 4 at 0x0001030ba074 by thread T1:
    #0 Thread1(void*) simple_race.cc:8 (a.out:x86_64+0x100000cc9)

  Location is global 'y' at 0x0001030ba074 (a.out+0x000100001074)

  Thread T2 (tid=747232, running) created by main thread at:
    #0 pthread_create <null>:1600736 (libclang_rt.tsan_osx_dynamic.dylib:x86_64h+0x283ed)
    #1 main simple_race.cc:27 (a.out:x86_64+0x100000df3)

  Thread T1 (tid=747231, finished) created by main thread at:
    #0 pthread_create <null>:1600736 (libclang_rt.tsan_osx_dynamic.dylib:x86_64h+0x283ed)
    #1 main simple_race.cc:26 (a.out:x86_64+0x100000dd4)

SUMMARY: ThreadSanitizer: data race simple_race.cc:13 in Thread2(void*)
=====
WARNING: ThreadSanitizer: data race (pid=76224)
  Write of size 4 at 0x0001030ba070 by thread T2:
    #0 Thread2(void*) simple_race.cc:14 (a.out:x86_64+0x100000d3a)

  Previous read of size 4 at 0x0001030ba070 by thread T1:
    #0 Thread1(void*) simple_race.cc:8 (a.out:x86_64+0x100000cae)

  Location is global 'x' at 0x0001030ba070 (a.out+0x000100001070)

  Thread T2 (tid=747232, running) created by main thread at:
    #0 pthread_create <null>:1600736 (libclang_rt.tsan_osx_dynamic.dylib:x86_64h+0x283ed)
    #1 main simple_race.cc:27 (a.out:x86_64+0x100000df3)

  Thread T1 (tid=747231, finished) created by main thread at:
    #0 pthread_create <null>:1600736 (libclang_rt.tsan_osx_dynamic.dylib:x86_64h+0x283ed)
    #1 main simple_race.cc:26 (a.out:x86_64+0x100000dd4)

SUMMARY: ThreadSanitizer: data race simple_race.cc:14 in Thread2(void*)
=====
ThreadSanitizer: reported 2 warnings

```

Figure 6.5: Output of ThreadSanitizer on C program in Figure 6.3b

6.1.4 DRD

Valgrind also provides another race detector DRD. We analyze program in Figure 6.3b in DRD, and show its output in Figure 6.7. Again, DRD reports two races, one of which is false.

6.2 PRECISE SOUNDNESS GUARANTEE OF HB

The notion of correct reorderings (discussed earlier in Chapter 2) captures a very generic notion of reorderings that can be inferred from an execution. Here, we consider a stricter notion of correct reordering that captures the essence of HB. Under the assumption that $\leq_{\text{HB}}^{\sigma}$ identifies certain causal dependencies between events of σ , we consider interleavings of

```

==2403== Helgrind, a thread error detector
==2403== Copyright (C) 2007-2015, and GNU GPL'd, by OpenWorks LLP et al.
==2403== Using Valgrind-3.12.0 and LibVEX; rerun with -h for copyright info
==2403== Command: ./a.out
==2403== -----
==2403== Possible data race during read of size 4 at 0x601044 by thread #3
==2403== Locks held: none
==2403==   at 0x4006A3: Thread2(void*) (simple_race.cc:13)
==2403==   by 0x4C3083E: mythread_wrapper (hg_intercepts.c:389)
==2403==   by 0x4E42E24: start_thread (in /usr/lib64/libpthread-2.17.so)
==2403==   by 0x596F34C: clone (in /usr/lib64/libc-2.17.so)
==2403== This conflicts with a previous write of size 4 by thread #2
==2403== Locks held: none
==2403==   at 0x40068E: Thread1(void*) (simple_race.cc:8)
==2403==   by 0x4C3083E: mythread_wrapper (hg_intercepts.c:389)
==2403==   by 0x4E42E24: start_thread (in /usr/lib64/libpthread-2.17.so)
==2403==   by 0x596F34C: clone (in /usr/lib64/libc-2.17.so)
==2403== Address 0x601044 is 0 bytes inside data symbol "y"
==2403== -----
==2403== Possible data race during write of size 4 at 0x601040 by thread #3
==2403== Locks held: none
==2403==   at 0x4006AE: Thread2(void*) (simple_race.cc:14)
==2403==   by 0x4C3083E: mythread_wrapper (hg_intercepts.c:389)
==2403==   by 0x4E42E24: start_thread (in /usr/lib64/libpthread-2.17.so)
==2403==   by 0x596F34C: clone (in /usr/lib64/libc-2.17.so)
==2403== This conflicts with a previous read of size 4 by thread #2
==2403== Locks held: none
==2403==   at 0x400685: Thread1(void*) (simple_race.cc:8)
==2403==   by 0x4C3083E: mythread_wrapper (hg_intercepts.c:389)
==2403==   by 0x4E42E24: start_thread (in /usr/lib64/libpthread-2.17.so)
==2403==   by 0x596F34C: clone (in /usr/lib64/libc-2.17.so)
==2403== Address 0x601040 is 0 bytes inside data symbol "x"
.....
==2403== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)

```

Figure 6.6: Output of Helgrind on the C program in Figure 6.3b

a trace σ that are consistent with $\leq_{\text{HB}}^{\sigma}$.

Definition 6.1 ($\leq_{\text{HB}}$ -respecting trace). Let σ and ρ be traces. We say trace ρ respects $\leq_{\text{HB}}^{\sigma}$ if $\text{Events}_{\rho} \subseteq \text{Events}_{\sigma}$, and for any $e, e' \in \text{Events}_{\sigma}$ such that $e \leq_{\text{HB}}^{\sigma} e'$ and $e' \in \text{Events}_{\rho}$, we have $e \in \text{Events}_{\rho}$ and $e \leq_{\text{tr}}^{\rho} e'$.

Thus, a $\leq_{\text{HB}}^{\sigma}$ -respecting trace is one whose events are downward closed with respect to $\leq_{\text{HB}}^{\sigma}$ and in which $\leq_{\text{HB}}^{\sigma}$ -ordered events are not flipped.

Example 6.1. We give examples of correct reorderings of σ_3 shown in Figure 6.8. The traces $\rho_1 = e_1e_2e_7$, $\rho_2 = e_4e_5e_6$, and $\rho_3 = e_1e_2e_3e_4e_5e_7$ are all examples of correct reorderings of σ_3 . Among these, the trace ρ_2 is not $\leq_{\text{HB}}^{\sigma_3}$ -respecting because it is not $\leq_{\text{HB}}^{\sigma_3}$ -downward closed — events e_1, e_2, e_3 are all ordered before e_4 according to $\leq_{\text{HB}}^{\sigma}$ but none of them are in ρ_2 .

```

==2624== drd, a thread error detector
==2624== Copyright (C) 2006-2015, and GNU GPL'd, by Bart Van Assche.
==2624== Using Valgrind-3.12.0 and LibVEX; rerun with -h for copyright info
==2624== Command: ./a.out
==2624==
==2624== Thread 3:
==2624== Conflicting load by thread 3 at 0x00601044 size 4
==2624==   at 0x4006A3: Thread2(void*) (simple_race.cc:13)
==2624==   by 0x4C30193: vgDrd_thread_wrapper (drd_pthread_intercepts.c:444)
==2624==   by 0x4E4FE24: start_thread (in /usr/lib64/libpthread-2.17.so)
==2624==   by 0x597C34C: clone (in /usr/lib64/libc-2.17.so)
==2624== Allocation context: BSS section of /home/umathur3/a.out
==2624== Other segment start (thread 2)
==2624==   (thread finished, call stack no longer available)
==2624== Other segment end (thread 2)
==2624==   (thread finished, call stack no longer available)
==2624==
==2624== Conflicting store by thread 3 at 0x00601040 size 4
==2624==   at 0x4006AE: Thread2(void*) (simple_race.cc:14)
==2624==   by 0x4C30193: vgDrd_thread_wrapper (drd_pthread_intercepts.c:444)
==2624==   by 0x4E4FE24: start_thread (in /usr/lib64/libpthread-2.17.so)
==2624==   by 0x597C34C: clone (in /usr/lib64/libc-2.17.so)
==2624== Allocation context: BSS section of /home/umathur3/a.out
==2624== Other segment start (thread 2)
==2624==   (thread finished, call stack no longer available)
==2624== Other segment end (thread 2)
==2624==   (thread finished, call stack no longer available)
==2624==
==2624== For counts of detected and suppressed errors, rerun with: -v
==2624== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 18 from 12)

```

Figure 6.7: Output of DRD on the C program in Figure 6.3b

Recall that a pair of conflicting events (e, e') in a trace σ is said to be an **HB-race** if $e \leq_{\text{tr}}^{\sigma} e'$ and further $e \parallel_{\text{HB}}^{\sigma} e'$. We say an **HB-race** (e, e') is the *first HB-race* if for any other **HB-race** $(f, f') \neq (e, e')$ in σ , either $e' <_{\text{tr}}^{\sigma} f'$, or $e' = f'$ and $f <_{\text{tr}}^{\sigma} e$. For example, the pair (e_2, e_3) in trace σ_1 from Figure 6.1 is the first **HB-race** of σ_1 . The soundness guarantee of **HB** says that if a trace σ has an **HB-race**, then the first **HB-race** is a predictable data race.

Theorem 6.1 (Soundness of **HB** for first race). Let σ be a trace with an **HB-race**, and let (e, e') be the first **HB-race**. Then, there is a correct reordering ρ of σ , such both e and e' are σ -enabled in ρ .

We remark that Theorem 6.1 is a stronger version of Theorem 3.3, and in fact the proof of Theorem 3.3 constructs a correct reordering in which the first **HB-race** is enabled. We will show that, in fact, both Theorem 6.1 and Theorem 3.3 follow from the main result we presented in this chapter, namely, Theorem 6.2.

Example 6.2. We conclude this section by giving examples of **HB-races**. Consider again σ_3 from Figure 6.8. Among the different pairs of conflicting events in σ_3 , the **HB-races** are (e_2, e_7) , (e_5, e_7) , (e_2, e_9) , (e_5, e_9) , (e_2, e_{10}) , (e_5, e_{10}) , (e_2, e_{12}) , and (e_5, e_{12}) . Of these, (e_5, e_7) is the first **HB-race**, and in fact, the **HB-race** (e_2, e_7) is not a predictable race.

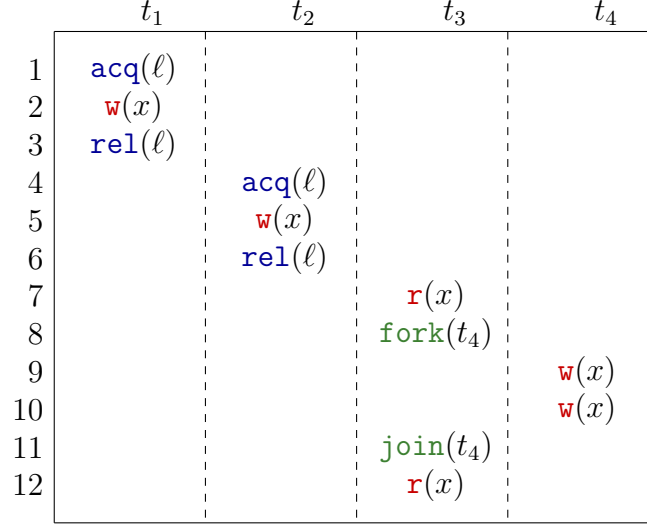


Figure 6.8: Trace σ_3 .

6.3 CHARACTERIZING SCHEDULABLE RACES

We previously observed that not all **HB**-races are predictable data races. The goal of this section is to characterize those **HB** races which correspond to actual predictable data races. We remark that the problem of checking if a given **HB**-race is a predictable race is likely to be intractable in general [17, 18]. In order to capture a more tractable notion, we will be considering **HB**-schedulable races, which we define next, and will introduce a new partial order, called *schedulable happens before*, to identify all **HB**-races which are **HB**-schedulable.

Definition 6.2 (**HB**-schedulable race). Let σ be a trace and let $e \leq_{\text{tr}}^{\sigma} e'$ be conflicting events in σ . We say that (e, e') is a $\leq_{\text{HB}}^{\sigma}$ -schedulable race if there is a correct reordering ρ of σ that respects $\leq_{\text{HB}}^{\sigma}$ in which both e and e' are σ -enabled.

Note that any $\leq_{\text{HB}}^{\sigma}$ -schedulable race is also a predictable data race of σ . Our aim is to characterize $\leq_{\text{HB}}^{\sigma}$ -schedulable races by means of a new partial order. The new partial order, given below, is a strengthening of $\leq_{\text{HB}}$.

Definition 6.3 (Schedulable Happens-Before). Let σ be a trace. Schedulable happens-before, denoted by $\leq_{\text{SHB}}^{\sigma}$, is the smallest partial order on Events_{σ} such that

- (a) $\leq_{\text{HB}}^{\sigma} \subseteq \leq_{\text{SHB}}^{\sigma}$
- (b) $\forall e, e' \in \text{Events}_{\sigma}, e' \in \text{Reads}_{\sigma} \wedge e = \text{lastWrite}_{\sigma}(e') \implies e \leq_{\text{SHB}}^{\sigma} e'$

The partial order $<_{\text{HB}}^{\sigma}$ is $\leq_{\text{SHB}}^{\sigma} \setminus \{(e, e) \mid e \in \text{Events}_{\sigma}\}$.

The partial order $\leq_{\text{SHB}}^\sigma$ can be used to characterize $\leq_{\text{HB}}^\sigma$ -schedulable races. We state this result, before giving examples illustrating the definition of $\leq_{\text{SHB}}^\sigma$.

Theorem 6.2. Let σ be a trace and $e_1 \leq_{\text{tr}}^\sigma e_2$ be conflicting events in σ . (e_1, e_2) is an $\leq_{\text{HB}}^\sigma$ -schedulable race iff either $\text{pred}_\sigma(e_2)$ is undefined, or $e_1 \not\leq_{\text{SHB}}^\sigma \text{pred}_\sigma(e_2)$.

The proof of Theorem 6.2 needs some auxiliary lemmas which we state and prove next.

Lemma 6.1. Let σ be a trace and σ' be a correct reordering of σ that respects $\leq_{\text{HB}}^\sigma$. For any e, e' such that $e \leq_{\text{SHB}}^\sigma e'$, if $e' \in \text{Events}_{\sigma'}$ and e' is not the last read event of its thread in σ' , then $e \in \text{Events}_{\sigma'}$ and $e \leq_{\text{tr}}^{\sigma'} e'$.

Proof. Consider any e, e' such that $e \leq_{\text{SHB}}^\sigma e'$, $e' \in \text{Events}_{\sigma'}$ and $e' = \langle t, \text{op} \rangle$ is not the last read event of the thread t in the trace σ' . Then it follows from Definition 6.3 that there is a sequence $e = e_0, e_1, \dots, e_n = e'$ such that for every $i \leq n-1$, $e_i \leq_{\text{tr}}^\sigma e_{i+1}$ and either (a) $e_i \leq_{\text{TO}}^\sigma e_{i+1}$ or (b) $e_i = \langle t_i, \text{rel}(\ell) \rangle$, $e_{i+1} = \langle t_{i+1}, \text{acq}(\ell) \rangle$, or (c) $e_{i+1} \in \text{Reads}_\sigma$ and $e_i = \text{lastWrite}_\sigma(e_{i+1})$.

We will prove by induction on i , starting from $i = n$, that $e_i \in \text{Events}_{\sigma'}$ and e_i is not the last read event of its thread in σ' . Observe that these properties hold for $e' = e_n$ — $e_n \in \text{Events}_{\sigma'}$ and e_n is not the last read event of its thread in σ' . Assume we have established the claim for e_{i+1} . Now there are three cases to consider for e_i . If $e_i \leq_{\text{TO}}^{\sigma'} e_{i+1}$ then clearly $e_i \in \text{Events}_{\sigma'}$ because $e_{i+1} \in \text{Events}_{\sigma'}$. Further, if e_i is a read event, then it is not the last event of its thread because e_{i+1} is after it. If $e_i = \langle t_i, \text{rel}(\ell) \rangle$ and $e_{i+1} = \langle t_{i+1}, \text{acq}(\ell) \rangle$ then $e_i \in \text{Events}_{\sigma'}$ because σ' respects $\leq_{\text{HB}}^\sigma$. Further e_i is not the last read event because it is not a read event! The last case to consider is where $e_i = \text{lastWrite}_\sigma(e_{i+1})$. In this case, by induction hypothesis, we know that e_{i+1} is not the last read event of its thread, and therefore by properties of a correct reordering, we have $e_i \in \text{Events}_{\sigma'}$. Notice that in this case e_i is not a read event, and so the claim holds. Thus, we have established that $e = e_0 \in \text{Events}_{\sigma'}$.

Next, we show that for every $i \leq n-1$, $e_i \leq_{\text{tr}}^{\sigma'} e_{i+1}$. If $e_i \leq_{\text{TO}}^\sigma e_{i+1}$ or $e_i = \langle t_i, \text{rel}(\ell) \rangle$ and $e_{i+1} = \langle t_{i+1}, \text{acq}(\ell) \rangle$ with $e_i \leq_{\text{tr}}^\sigma e_{i+1}$ then $e_i \leq_{\text{tr}}^{\sigma'} e_{i+1}$ because σ' respects $\leq_{\text{HB}}^\sigma$. On the other hand, if $e_i = \text{lastWrite}_\sigma(e_{i+1})$ then because σ' is a correct reordering of σ and e_{i+1} is not the last read event of its thread (established in the previous paragraph), we have $e_i = \text{lastWrite}_\sigma(e_{i+1}) = \text{lastWrite}_{\sigma'}(e_{i+1})$. This establishes the fact that $e = e_0 \leq_{\text{tr}}^{\sigma'} e_n = e'$, which completes the proof of the lemma. QED.

A slightly weaker form of the converse of Lemma 6.1 also holds.

Lemma 6.2. For a trace σ , let σ' be a trace with $\text{Events}_{\sigma'} \subseteq \text{Events}_\sigma$ such that (a) σ' is $\leq_{\text{SHB}}^\sigma$ downward closed, i.e., for any $e, e' \in \text{Events}_\sigma$ if $e \leq_{\text{SHB}}^\sigma e'$ and $e' \in \text{Events}_{\sigma'}$ then $e \in \text{Events}_{\sigma'}$,

and (b) $\leq_{\text{tr}}^{\sigma'} = \leq_{\text{tr}}^{\sigma} \cap (\text{Events}_{\sigma'} \times \text{Events}_{\sigma'})$. Then σ' is a correct reordering of σ that respects $\leq_{\text{HB}}^{\sigma}$. Further, for **every** read event $e \in \text{Reads}_{\sigma'}$, we have $\text{lastWrite}_{\sigma'}(e) \simeq \text{lastWrite}_{\sigma}(e)$, i.e., either both $\text{lastWrite}_{\sigma'}(e)$ and $\text{lastWrite}_{\sigma}(e)$ are undefined, or they are both defined and equal.

Proof. The trace σ' in the lemma is such that the events in σ' are downward closed with respect to $\leq_{\text{SHB}}^{\sigma}$ and in σ' they are ordered in exactly the same way as in σ . The fact that σ' respects $\leq_{\text{HB}}^{\sigma}$ simply follows from the fact that $\leq_{\text{HB}}^{\sigma} \subseteq \leq_{\text{SHB}}^{\sigma}$ and $\leq_{\text{HB}}^{\sigma} \subseteq \leq_{\text{tr}}^{\sigma}$. So the main goal is to establish that σ' is a correct reordering of σ that preserves the last writes of *all* read events.

First we show that σ' respects lock semantics. Suppose $e_1 = \langle t_1, \text{acq}(\ell) \rangle$ and $e_2 = \langle t_2, \text{acq}(\ell) \rangle$ are two lock acquire events for some lock ℓ such that $e_1 \leq_{\text{tr}}^{\sigma} e_2$ and $\{e_1, e_2\} \subseteq \text{Events}_{\sigma'}$. Let e'_1 be the matching release event for e_1 in σ ; such an e'_1 exists because σ is a valid trace. Then we have $e_1 \leq_{\text{HB}}^{\sigma} e'_1 \leq_{\text{HB}}^{\sigma} e_2$, and so $e'_1 \in \text{Events}_{\sigma'}$ and $e'_1 \leq_{\text{tr}}^{\sigma'} e_2$ because σ' respects $\leq_{\text{HB}}^{\sigma}$.

Next observe that since $\leq_{\text{TO}}^{\sigma} \subseteq \leq_{\text{HB}}^{\sigma}$ and σ' respects $\leq_{\text{HB}}^{\sigma}$, we can conclude that $\sigma'|_t$ is a prefix of $\sigma|_t$ for any thread t .

Finally, consider any $e' \in \text{Reads}_{\sigma'}$. Suppose $\text{lastWrite}_{\sigma}(e')$ is defined. Let $e = \text{lastWrite}_{\sigma}(e')$. Since $e \leq_{\text{SHB}}^{\sigma} e'$ and σ' is downward closed with respect to $\leq_{\text{SHB}}^{\sigma}$, we have $e \in \text{Events}_{\sigma'}$. Let $e_1 = \text{lastWrite}_{\sigma'}(e')$. We need to argue that $e_1 = e$. Suppose (for contradiction) it is not, i.e., $e \neq e_1$. Then either $e_1 \leq_{\text{tr}}^{\sigma} e$ or $e' \leq_{\text{tr}}^{\sigma} e_1$, because $e = \text{lastWrite}_{\sigma}(e')$. However, the fact that $e_1 = \text{lastWrite}_{\sigma'}(e')$ contradicts the fact that $\leq_{\text{tr}}^{\sigma'} = \leq_{\text{tr}}^{\sigma} \cap (\text{Events}_{\sigma'} \times \text{Events}_{\sigma'})$. Conversely, if $\text{lastWrite}_{\sigma'}(e')$ is defined then let $e = \text{lastWrite}_{\sigma'}(e')$. Since $\leq_{\text{tr}}^{\sigma'} = \leq_{\text{tr}}^{\sigma} \cap (\text{Events}_{\sigma'} \times \text{Events}_{\sigma'})$, we have $e \leq_{\text{tr}}^{\sigma} e'$. Thus, $\text{lastWrite}_{\sigma}(e')$ is defined. Let $e_1 = \text{lastWrite}_{\sigma}(e')$. Once again, since $e_1 \leq_{\text{SHB}}^{\sigma} e'$, and σ' is downward closed with respect to $\leq_{\text{SHB}}^{\sigma}$, we have $e_1 \in \sigma'$. Just like in the previous direction, we can conclude that $e = e_1$ because otherwise we violate the fact that $\leq_{\text{tr}}^{\sigma'}$ is identical to $\leq_{\text{tr}}^{\sigma}$ over $\text{Events}_{\sigma'}$. QED.

We now prove Theorem 6.2 below.

Proof of Theorem 6.2. Let us first prove the forward direction. That is, let (e_1, e_2) be an HB-race such that the event $e = \text{pred}_{\sigma}(e_2)$ is defined and $e_1 \leq_{\text{SHB}}^{\sigma} e$. Consider any correct reordering σ' that contains both e_1 and e_2 and respects $\leq_{\text{HB}}^{\sigma}$. First, since σ' is a correct reordering of σ , we must have $e \in \text{Events}_{\sigma'}$ and $e \leq_{\text{tr}}^{\sigma'} e_2$. Further, since $e_1 \leq_{\text{SHB}}^{\sigma} e$, from Lemma 6.1, $e_1 \leq_{\text{tr}}^{\sigma'} e$. Thus, we have that $e_1 \leq_{\text{tr}}^{\sigma'} e \leq_{\text{tr}}^{\sigma'} e_2$ for any correct reordering σ' of σ that respects $\leq_{\text{HB}}^{\sigma}$. This means, (e_1, e_2) cannot be a $\leq_{\text{HB}}^{\sigma}$ -schedulable race.

We now prove the backward direction. Consider an HB-race (e_1, e_2) such that either $\text{pred}_\sigma(e_2)$ is undefined, or if it exists, then it satisfies $e_1 \not\leq_{\text{SHB}}^\sigma \text{pred}_\sigma(e_2)$. Consider the set $S_{(e_1, e_2)}^\sigma$ defined as

$$S_{(e_1, e_2)}^\sigma = \{e \in \text{Events}_\sigma \setminus \{e_1, e_2\} \mid e \leq_{\text{SHB}}^\sigma e_1 \text{ or } e \leq_{\text{SHB}}^\sigma \text{pred}_\sigma(e_2)\} \quad (6.1)$$

where we assume that if $\text{pred}_\sigma(e_2)$ is undefined then no event e satisfies the condition $e \leq_{\text{SHB}}^\sigma \text{pred}_\sigma(e_2)$.

First we will show that $S_{(e_1, e_2)}^\sigma$ is downward closed with respect to $\leq_{\text{SHB}}^\sigma$. Consider e, e' such that $e \leq_{\text{SHB}}^\sigma e'$ and $e' \in S_{(e_1, e_2)}^\sigma$. By definition of $S_{(e_1, e_2)}^\sigma$, we have $e' \notin \{e_1, e_2\}$ and either $e' \leq_{\text{SHB}}^\sigma e_1$ or $e' \leq_{\text{SHB}}^\sigma \text{pred}_\sigma(e_2)$. Observe that if $e \notin \{e_1, e_2\}$, then it is clear that $e \in S_{(e_1, e_2)}^\sigma$ by definition since $\leq_{\text{SHB}}^\sigma$ is transitive. It is easy to see that $e \neq e_2$ — this is because since $e' \neq e_2$, and $\leq_{\text{SHB}}^\sigma \leq_{\text{tr}}^\sigma$, $e' <_{\text{tr}}^\sigma e_2$ and so $e <_{\text{tr}}^\sigma e_2$. So, all we have left to establish is that $e \neq e_1$. Suppose for contradiction $e = e_1$. Then it must be the case that $e' \leq_{\text{SHB}}^\sigma \text{pred}_\sigma(e_2)$. Since $e_1 = e \leq_{\text{SHB}}^\sigma e' \leq_{\text{SHB}}^\sigma \text{pred}_\sigma(e_2)$, we have $e_1 \leq_{\text{SHB}}^\sigma \text{pred}_\sigma(e_2)$, which contradicts our assumption about (e_1, e_2) .

Let us now consider a trace σ'' which consists of the events in $S_{(e_1, e_2)}^\sigma$ ordered according to $\leq_{\text{tr}}^\sigma$. That is, $\leq_{\text{tr}}^{\sigma''} = \leq_{\text{tr}}^\sigma \cap (S_{(e_1, e_2)}^\sigma \times S_{(e_1, e_2)}^\sigma)$. Since σ'' satisfies the conditions of Lemma 6.2, we can conclude that σ'' is a correct reordering of σ that respects $\leq_{\text{HB}}^\sigma$ and preserves the last-writes of *every* read event present.

Consider the trace $\sigma' = \sigma'' e_1 e_2$. First we prove that σ' respects $\leq_{\text{HB}}^\sigma$. To do that, we first show that for any event $e \in \text{Events}_\sigma$ such that $e \leq_{\text{HB}}^\sigma e_1$ and $e \neq e_1$, or $e \leq_{\text{HB}}^\sigma e_2$ and $e \neq e_2$, then $e \in S_{(e_1, e_2)}^\sigma$. If $e \leq_{\text{HB}}^\sigma e_1$ then $e \leq_{\text{SHB}}^\sigma e_1$ and so $e \in S_{(e_1, e_2)}^\sigma$. On the other hand, if $e \leq_{\text{HB}}^\sigma e_2$ (and $e \neq e_2$), since (e_1, e_2) is an HB-race, we must have $e \neq e_1$ and $e \leq_{\text{HB}}^\sigma \text{pred}_\sigma(e_2)$. So $e \in S_{(e_1, e_2)}^\sigma$. Now the fact σ' respects $\leq_{\text{HB}}^\sigma$ follows from the fact that σ'' respects $\leq_{\text{HB}}^\sigma$ and the claim just proved.

We now prove that σ' is a correct reordering. Observe that since σ' respects $\leq_{\text{HB}}^\sigma$, σ' is well formed (lock semantics is not violated) and preserves thread-wise prefixes ($\forall t, \sigma'|_t$ is a prefix of $\sigma|_t$). Further, σ'' is such that every read event in σ'' reads the same last write as in σ . Also, since e_1 and e_2 are the last events in their threads in σ' , we conclude that σ' is a correct reordering of σ that respects $\leq_{\text{HB}}^\sigma$. QED.

6.3.1 Examples Illustrating SHB Partial Order

We now illustrate the use of $\leq_{\text{SHB}}$ through some examples.

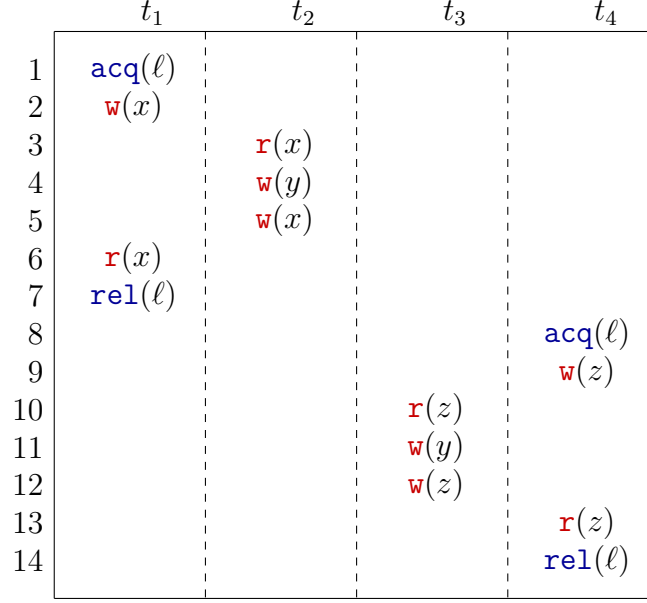


Figure 6.9: Trace σ_4 .

Example 6.3. In this example, we will look at different traces, and see how $\leq_{\text{SHB}}$ reasons. Let us begin by considering the example program and trace σ_1 from Figure 6.1. Notice that $\leq_{\text{HB}}^{\sigma_1} = \leq_{\text{TO}}^{\sigma_1}$, and so (e_1, e_4) and (e_2, e_3) are HB-races. Because $e_2 = \text{lastWrite}_{\sigma_1}(e_3)$, we have $e_1 \leq_{\text{SHB}}^{\sigma_1} e_2 \leq_{\text{SHB}}^{\sigma_1} e_3 \leq_{\text{SHB}}^{\sigma_1} e_4$. Using Theorem 6.2, we can conclude correctly that (a) (e_2, e_3) is $\leq_{\text{HB}}^{\sigma_1}$ -schedulable as $\text{pred}_{\sigma_1}(e_3)$ is undefined, but (b) (e_1, e_4) is not, as $e_1 \leq_{\text{SHB}}^{\sigma} \text{pred}_{\sigma}(e_4) = e_3$.

Let us now consider trace σ_2 from Figure 6.2. Observe that $\leq_{\text{HB}}^{\sigma_2} = \leq_{\text{SHB}}^{\sigma_2} = \leq_{\text{TO}}^{\sigma_2}$, and so both (e_1, e_4) and (e_2, e_3) are $\leq_{\text{HB}}^{\sigma_2}$ -schedulable races by Theorem 6.2. Note that, unlike force ordering, $\leq_{\text{SHB}}^{\sigma_2}$ correctly identifies all real data races.

Finally, let us consider two trace examples that highlight the kind of subtle reasoning $\leq_{\text{SHB}}$ is capable of. Let us begin with σ_3 from Figure 6.8. As observed in Example 6.2, the only HB-races in this trace are (e_2, e_7) , (e_5, e_7) , (e_2, e_9) , (e_5, e_9) , (e_2, e_{10}) , (e_5, e_{10}) , (e_2, e_{12}) , and (e_5, e_{12}) . Both (e_2, e_7) and (e_5, e_7) are $\leq_{\text{HB}}^{\sigma_3}$ -schedulable as demonstrated by the reorderings ρ_1 and ρ_3 from Example 6.1. However, the remaining are not real data races. Let us consider the pairs (e_2, e_9) and (e_5, e_9) for example. Theorem 6.2's justification for it is as follows: $e_2 \leq_{\text{HB}}^{\sigma_3} e_5 = \text{lastWrite}_{\sigma_3}(e_7) \leq_{\text{TO}}^{\sigma_3} e_8 = \text{pred}_{\sigma_3}(e_9)$. But, let us unravel the reasoning behind why neither (e_2, e_9) nor (e_5, e_9) are data races. Consider an arbitrary correct reordering σ' of σ_3 that respects $\leq_{\text{HB}}^{\sigma_3}$ and contains e_9 . Since e_8 is also an event of t_4 , $e_8 \in \text{Events}_{\sigma'}$. In addition, $e_7 \in \text{Events}_{\sigma'}$ as $e_7 \leq_{\text{TO}}^{\sigma_3} e_8$. Now, since $e_5 = \text{lastWrite}_{\sigma_3}(e_7)$, e_5 is before e_7 in σ' and since $e_2 \leq_{\text{HB}}^{\sigma_3} e_5$, e_2 must also be before e_7 . Therefore, e_7 and e_8 will be between e_2 and e_9 and between e_5 and e_9 . Similar reasoning can be used to conclude that the other pairs

are not $\leq_{\text{HB}}^{\sigma_3}$ -schedule as well.

Lastly, consider trace σ_4 shown in Figure 6.9. In this case, $\leq_{\text{SHB}}^{\sigma_4} = \leq_{\text{tr}}^{\sigma_4}$. All conflicting memory accesses are in **HB** race. While **HB** correctly identifies the first race (e_2, e_3) as valid, there are 3 **HB** races that are not real data races — (e_2, e_5) , (e_9, e_{12}) , and (e_4, e_{11}) . (e_2, e_5) is not valid because any correct reordering of σ_4 must have e_2 before e_3 and e_3 before e_5 . This is also captured by SHB reasoning because $e_2 \leq_{\text{SHB}}^{\sigma_4} e_3 \leq_{\text{TO}}^{\sigma_4} e_4 = \text{pred}_{\sigma_4}(e_5)$. A similar reasoning shows that (e_9, e_{12}) is not valid. The interesting case is that of (e_4, e_{11}) . Here, in any correct reordering σ' of σ_4 , the following must be true: (a) if $e_4 \in \text{Events}_{\sigma'}$ then $e_1 \in \text{Events}_{\sigma'}$; (b) if $e_{11} \in \text{Events}_{\sigma'}$ then $e_8 \in \text{Events}_{\sigma'}$; (c) if $\{e_1, e_4, e_7\} \subseteq \text{Events}_{\sigma'}$ then $e_1 \leq_{\text{tr}}^{\sigma'} e_4 \leq_{\text{tr}}^{\sigma'} e_7$; and (d) if $\{e_8, e_{11}, e_{14}\} \subseteq \text{Events}_{\sigma'}$ then $e_8 \leq_{\text{tr}}^{\sigma'} e_{11} \leq_{\text{tr}}^{\sigma'} e_{14}$. Therefore, any correct reordering σ' of σ_4 containing both e_4 and e_{11} contains e_1 and e_8 (because of (a) and (b)) and must contain at least one of e_7 or e_{14} to ensure that critical sections of ℓ don't overlap. Then in σ' , e_4 and e_{11} cannot be consecutive because either e_7 or e_{14} will appear between them (properties (c) and (d)). This is captured using SHB and Theorem 6.2 by the fact that $e_4 \leq_{\text{SHB}}^{\sigma_4} e_7 \leq_{\text{SHB}}^{\sigma_4} e_{10} = \text{pred}_{\sigma_4}(e_{11})$.

Let us now show how the soundness guarantees of **HB** (Theorem 6.1) follows from Theorem 6.2. Consider a trace σ whose first **HB**-race is (e_1, e_2) . We claim that (e_1, e_2) is a $\leq_{\text{HB}}^{\sigma}$ -schedulable race. Suppose (for contradiction) it is not. Then by Theorem 6.2, $e = \text{pred}_{\sigma}(e_2)$ is defined and $e_1 \leq_{\text{SHB}}^{\sigma} e$. Now observe that we $\neg(e_1 \leq_{\text{HB}}^{\sigma} e)$ (or otherwise $e_1 \leq_{\text{HB}}^{\sigma} e_2$, contradicting our assumption that (e_1, e_2) is an **HB**-race). Then, by the definition of $\leq_{\text{SHB}}^{\sigma}$ (Definition 6.3), there are two events e_3 and e_4 (possibly same as e_1 and e) such that $e_1 \leq_{\text{SHB}}^{\sigma} e_3$, $e_3 = \text{lastWrite}_{\sigma}(e_4)$, $e_4 \leq_{\text{SHB}}^{\sigma} e$, and $\neg(e_3 \leq_{\text{HB}}^{\sigma} e_4)$. Then (e_3, e_4) is an **HB** race, and it contradicts the assumption that (e_1, e_2) is the first **HB** race.

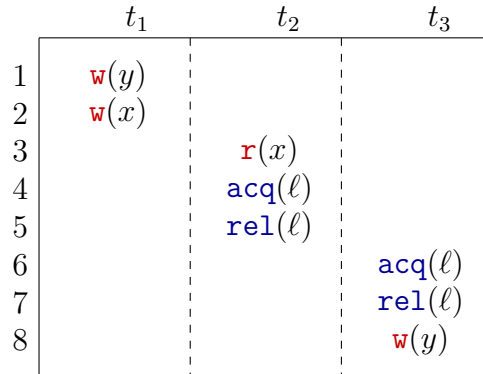


Figure 6.10: Trace σ_5 .

We now illustrate the limitations of **HB**-based reasoning.

Algorithm 6.1: *Vector Clock for Checking $\leq_{SHB}$ -schedulable races*

```

1 function Initialization()
2   foreach  $t \in \text{Threads}$  do
3      $\mathbb{C}_t := \perp[t \mapsto 1]$ 
4   foreach  $\ell \in \text{Locks}$  do
5      $\mathbb{L}_\ell := \perp$ 
6   for  $x \in \text{Vars}$  do
7      $\mathbb{LW}_x := \perp$ 
8      $\mathbb{R}_x := \perp$ 
9      $\mathbb{W}_x := \perp$ 
10 handler acquire( $t, \ell$ )
11    $\mathbb{C}_t := \mathbb{C}_t \sqcup \mathbb{L}_\ell$ 
12 handler release( $t, \ell$ )
13    $\mathbb{L}_\ell := \mathbb{C}_t$ 
14    $\mathbb{C}_t := \mathbb{C}_t[t \mapsto \mathbb{C}_t(t) + 1]$ 
15 handler fork( $t, u$ )
16    $\mathbb{C}_u := \mathbb{C}_t[u \mapsto 1]$ 
17    $\mathbb{C}_t := \mathbb{C}_t[t \mapsto \mathbb{C}_t(t) + 1]$ 
18 handler join( $t, u$ )
19    $\mathbb{C}_t := \mathbb{C}_t \sqcup \mathbb{C}_u$ 
20 handler read( $t, x$ )
21   if  $\mathbb{W}_x \not\sqsubseteq \mathbb{C}_t$  then
22     declare ‘race with write’
23      $\mathbb{C}_t := \mathbb{C}_t \sqcup \mathbb{LW}_x$ 
24      $\mathbb{R}_x := \mathbb{R}_x[t \mapsto \mathbb{C}_t(t)]$ 
25 handler write( $t, x$ )
26   if  $\mathbb{R}_x \not\sqsubseteq \mathbb{C}_t$  then
27     declare ‘race with read’
28   if  $\mathbb{W}_x \not\sqsubseteq \mathbb{C}_t$  then
29     declare ‘race with write’
30      $\mathbb{LW}_x := \mathbb{C}_t$ 
31      $\mathbb{W}_x := \mathbb{W}_x[t \mapsto \mathbb{C}_t(t)]$ 
32      $\mathbb{C}_t := \mathbb{C}_t[t \mapsto \mathbb{C}_t(t) + 1]$ 

```

Example 6.4. Consider the trace σ_5 in Figure 6.10. Here, (e_2, e_3) and $(e_1, 8)$ are both HB-races, with (e_2, e_3) also being the first HB-race. As a result, this is a HB-schedulable race flagged by the SHB partial order (because $\text{pred}_{\sigma_5}(e_3)$ is undefined). On the other hand, $e_1 \leq_{SHB}^{\sigma_5} e_8$ because $e_2 = \text{lastWrite}_{\sigma_5}(e_3)$ and $e_3 \leq_{HB}^{\sigma_5} e_8$. Indeed, (e_1, e_8) is not a $\leq_{HB}^{\sigma_5}$ -schedulable race. However, (e_1, e_8) is, in fact, a predictable race witnessed by the correct reordering $\rho = e_6 e_7$ in which both e_1 and e_8 are σ_5 -enabled.

The above example demonstrates that the notion of HB-schedulable races is not all powerful — SHB can miss predictable data races. However, SHB based reasoning provides a good balance of scalability and predictive power — there is a linear time algorithm for detecting HB-schedulable races (using the $\leq_{SHB}$ partial order), thus enhancing the predictive power of HB beyond the first data race, within the same time complexity. In Chapter 8, we discuss the notion of synchronization-preserving data races that further enhance the predictive power of SHB based reasoning.

6.4 ALGORITHM FOR DETECTING HB-SCHEDULABLE RACES

Our algorithm for detecting HB-schedulable races is based on efficient, vector clock based computation of the $\leq_{SHB}$ -partial order. It is similar to the standard DJIT⁺ algorithm [48]

to detect **HB**-races, also described in Section 3.2 (see Algorithm 3.2). Algorithm 6.1 depicts the vector clock algorithm for detecting $\leq_{\text{HB}}$ -schedulable races using the $\leq_{\text{SHB}}$ partial order. Similar to the vector clock algorithm for detecting **HB** races (Algorithm 3.2), Algorithm 6.1 maintains a state comprising of several vector clocks. As with Algorithm 3.2, the idea behind Algorithm 6.1 is to use these vector clocks to assign a vector timestamp to each event e (denoted by C_e) such that the ordering relation on the assigned timestamps ($\sqsubseteq$) enables determining the partial order $\leq_{\text{SHB}}$ on events. Let us precisely define the timestamp C_e of an event e in the trace σ that Algorithm 6.1 computes:

$$C_e = \lambda t. \begin{cases} 1 + |S_e^t| & \text{if } \text{thr}(e) = t \\ |S_e^t| & \text{otherwise} \end{cases} \quad (6.2)$$

where, for an event e and thread u , the set S_e^u is defined as follows:

$$S_e^u = \{e' <_{\text{SHB}}^\sigma e \mid \text{thr}(e') = u, e' \in \text{Releases}_\sigma \cup \text{Forks}_\sigma \cup \text{Writes}_\sigma\} \cup \{e' \leq_{\text{SHB}}^\sigma e \mid \text{op}(e') = \text{join}(u)\} \quad (6.3)$$

The above definition ensures the following.

Lemma 6.3. Let σ be a trace and let $e, e' \in \text{Events}_\sigma$ such that $\text{thr}(e) \neq \text{thr}(e')$. We have, $C_e \sqsubseteq C_{e'} \iff e \leq_{\text{SHB}}^\sigma e'$

The proof of Lemma 6.3 is similar to the proof of Lemma 3.3 and skipped.

The algorithm runs in a streaming fashion and processes each event in the order in which it occurs in the trace. Depending upon the type of the observed event, an appropriate handler is invoked. The formal parameter t in each of the handlers refers to the thread performing the event, and the parameters ℓ , x and u represent the lock being acquired or released, the memory location being accessed and the thread being forked or joined, respectively. The procedure **Initialization** assigns the initial values to the vector clocks in the state. We next present details of different parts of the algorithm.

6.4.1 Vector clocks in the State

The description of each of the vector clocks that are maintained in the state of Algorithm 6.1 is as follows:

1. **Clocks** $\mathbb{C}_t$: For every thread t in the trace being analyzed, the algorithm maintains a vector clock $\mathbb{C}_t$. The value stored in the clock $\mathbb{C}_t$ is the timestamp of the last event

of thread t seen in the trace so far. More precisely, let T_t^σ be the value of the clock $\mathbb{C}_t$ after observing trace σ and let e_t be the last event e in σ with $\text{thr}(e) = t$. Then, $T_t^\sigma = C_{e_t}$ if $e_t \notin \text{Releases}_\sigma \cup \text{Forks}_\sigma \cup \text{Writes}_\sigma$ and $T_t^\sigma = C_{e_t}[t \mapsto C_{e_t} + 1]$ otherwise.

2. **Clocks $\mathbb{L}_\ell$** : The algorithm maintains a vector clock $\mathbb{L}_\ell$ for every lock ℓ in the trace. At any point during the algorithm, the clock $\mathbb{L}_\ell$ stores the timestamp C_{e_ℓ} , where e_ℓ is the last event e in σ with $\text{op}(e_\ell) = \text{rel}(\ell)$.
3. **Clocks $\mathbb{LW}_x$** : For every memory location x accessed in the trace, the algorithm maintains a clock $\mathbb{LW}_x$ (Last Write to x) to store the timestamp C_{e_x} , of the last event e_x of the form $\langle \cdot, \text{w}(x) \rangle$.
4. **Clocks $\mathbb{R}_x$ and $\mathbb{W}_x$** : The clocks $\mathbb{R}_x$ and $\mathbb{W}_x$ store the read and write *access histories* of each memory location x . At any point in the algorithm, the vector time R_x stored in the the Read access history clock $\mathbb{R}_x$ is such that $\forall t, R_x(t) = C_{e_t^{\text{r}(x)}}(t)$ where $e_t^{\text{r}(x)}$ is the last event of thread t that reads x in the trace seen so far. Similarly, the vector time W_x stored in the Write access history clock $\mathbb{W}_x$ is such that $\forall t, W_x(t) = C_{e_t^{\text{w}(x)}}(t)$ where $e_t^{\text{w}(x)}$ is the last event of thread t that writes to x in the trace seen so far.

The clocks $\mathbb{C}_t$, $\mathbb{L}_\ell$, $\mathbb{LW}_x$ are used to correctly compute the timestamps of the events, while the access history clocks $\mathbb{R}_x$ and $\mathbb{W}_x$ are used to detect races.

6.4.2 Initialization and Clock Updates

For every thread t , the clock $\mathbb{C}_t$ is initialized to the vector time $\perp[t \mapsto 1]$. Each of the clocks $\mathbb{L}_\ell$, $\mathbb{LW}_x$, $\mathbb{R}_x$ and $\mathbb{W}_x$ are initialized to $\perp$. This is in accordance with the semantics of these clocks presented in Section 6.4.1.

When processing an acquire event e of type $\langle t, \text{acq}(\ell) \rangle$, the algorithm reads the clock $\mathbb{L}_\ell$ and updates the clock $\mathbb{C}_t$ with $\mathbb{C}_t \sqcup \mathbb{L}_\ell$ (see Line 9). This ensures that the timestamp C_e (which is the value of the clock $\mathbb{C}_t$ after executing Line 9) is such that $C_{e'} \sqsubseteq C_e$ for every event e' of type $\langle \cdot, \text{rel}(\ell) \rangle$ observed in the trace so far.

At a release event e of type $\langle t, \text{rel}(\ell) \rangle$, the algorithm writes the timestamp C_e of the current event e to the clock $\mathbb{L}_\ell$ (see Line 11). Notice that e is also the last release event of lock ℓ in the trace seen so far, and thus, this update correctly maintains the invariant stated in Section 6.4.1. This update ensures that any future events that acquire the lock ℓ can update their timestamps correctly. The algorithm then increments the local clock $\mathbb{C}_t(t)$ (Line 12). This ensures that if the next event e' in the thread t and the next acquire event

f of lock ℓ satisfy $e' \not\leq_{\text{SHB}} f$, then the timestamps of these events satisfy $C_{e'} \not\sqsubseteq C_f$. The updates performed by the algorithm at a fork (resp. join) event are similar to the updates performed when observing a release (resp. acquire) event.

At a read event $e = \langle t, \mathbf{r}(x) \rangle$, the clock $\mathbb{C}_t$ is updated with the join $\mathbb{C}_t \sqcup \mathbb{LW}_x$ (Line 21). Recall that $\mathbb{LW}_x$ stores the timestamp of the last event that writes to x (or in other words, the event $\text{lastWrite}(e)$) in the trace seen so far. This ensures that the timestamps C_e and $C_{\text{lastWrite}(e)}$ satisfy $C_{\text{lastWrite}(e)} \sqsubseteq C_e$. In addition, the algorithm also updates the component $\mathbb{R}_x(t)$ with the local component of the clock $\mathbb{C}_x$ (Line 22) in order to maintain the invariant described in Section 6.4.1.

At a write event $e = \langle t, \mathbf{w}(x) \rangle$, the algorithm updates the value of the last-write clock $\mathbb{LW}_x$ (Line 28) with the timestamp C_e stored in $\mathbb{C}_t$. The component $\mathbb{W}_x(t)$ is updated with the value of the local component $\mathbb{C}_t(t)$ to ensure the invariant described in Section 6.4.1 is maintained correctly. Finally, similar to the increment after a release event, the local clock is incremented in Line 30.

6.4.3 Checking for Races

At a read/write event e , the algorithm determines if there is a conflicting event e' in the trace seen so far such that (e', e) is an $\leq_{\text{HB}}$ -schedulable race. From Theorem 6.2 and Lemma 6.3, it follows that it is sufficient to check if $C_{e'} \not\sqsubseteq C_{\text{pred}(e)}$. However, since the algorithm does not explicitly store the timestamps of events, we use the access histories $\mathbb{R}_x$ and $\mathbb{W}_x$ to check for races. Below we briefly describe these checks. The formal statement of correctness is presented in Theorem 6.3. We briefly outline the ideas here.

Recall that, for an event $e = \langle t, \cdot \rangle$ if $\text{pred}(e)$ is undefined, the **Initialization** procedure ensures that $C_e = \perp[t \mapsto 1]$. In this case, we have $V \not\sqsubseteq C_e$, for any vector-timestamp V with non-negative entries such that $V(t) = 0$, $\perp \sqsubseteq V$ and $V \neq \perp$. Algorithm 6.1 correctly reports a race in this case (see Lines 19-20, 24-27).

On the other hand, if $\text{pred}(e)$ is defined, then the clock $\mathbb{C}_t$, at Line 19, 24 or 26, is either the timestamp $C_{\text{pred}(e)}$ (if $\text{pred}(e)$ was a read, join or an acquire event) or the timestamp $C_{\text{pred}(e)}[t \mapsto (c + 1)]$, where $c = C_{\text{pred}(e)}(t)$ (if $\text{pred}(e)$ was a write, fork or a release event). In either case, if the check $\mathbb{W}_x \sqsubseteq \mathbb{C}_t$ at Line 19 fails, then the read event e being processed is correctly declared to be in race with an earlier conflicting write event. Similarly, Algorithm 6.1 reports that a write event e is in race with an earlier read (resp. write) event based on whether the check on Line 24 (resp. Line 26) fails or not.

6.4.4 Correctness and Complexity

Here, we fix a trace σ . Recall that, for an event e , we say that C_e is the timestamp assigned by Algorithm 6.1 to event e . We say that Algorithm 6.1 reports a race at an event e , if it executes Line 22, Line 27 or Line 29 while processing the handler corresponding to e .

We are now ready to state the correctness of the algorithm we present here. For this we will first state some useful invariants. Let us suppose that the prefix of the trace processed by the algorithm until some point is σ . For a thread $t \in \mathbf{Threads}_\sigma$, let e_t^σ be the last event in σ with $\text{thr}(e_t^\sigma) = t$. Similarly, for a lock $\ell \in \mathbf{Locks}_\sigma$, let e_ℓ^σ be the last event in σ with $\text{op}(e_\ell^\sigma) = \mathbf{rel}(\ell)$. Finally, for a variable $x \in \mathbf{Vars}_\sigma$, let e_x^σ be the last event in σ with $\text{op}(e_x^\sigma) = \mathbf{w}(x)$. We remark that events e_t^σ , e_ℓ^σ or e_x^σ may not be defined. The following lemma then describes the invariant maintained by the algorithm.

Lemma 6.4. Let σ be a trace. For a thread t , lock ℓ and variable x , let T_t^σ , L_ℓ^σ and LW_x^σ be the values of the clock $\mathbb{C}_t$, $\mathbb{L}_\ell$ and $\mathbb{LW}_x$ after Algorithm 6.1 processes σ . Then,

$$\begin{aligned} T_t^\sigma &= \begin{cases} \perp[t \mapsto 1] & \text{if } e_t^\sigma \text{ does not exist} \\ C_{e_t^\sigma}[t \mapsto C_{e_t^\sigma}(t) + 1] & \text{if } e_t^\sigma \in \mathbf{Releases}_\sigma \cup \mathbf{Forks}_\sigma \\ C_{e_t^\sigma} & \text{otherwise} \end{cases} \\ L_\ell^\sigma &= \begin{cases} \perp & \text{if } e_\ell^\sigma \text{ does not exist} \\ C_{e_\ell^\sigma} & \text{otherwise} \end{cases} \\ LW_x^\sigma &= \begin{cases} \perp & \text{if } e_x^\sigma \text{ does not exist} \\ C_{e_x^\sigma} & \text{otherwise} \end{cases} \end{aligned}$$

The proof of is similar to that of Lemma 3.4 and thus skipped.

We now stat the correctness criteria for the race detection component of the algorithm.

Lemma 6.5. Let σ be a trace. For a variable x , let R_x^σ and W_x^σ be the values of the clocks $\mathbb{R}_x$ and $\mathbb{W}_x$ after Algorithm 3.2 processes σ . Then,

- for every thread t , $R_x^\sigma(t) = C_e(t)$, where e is the last event in σ with $\text{thr}(e) = t$ and $\text{op}(e) = \mathbf{r}(x)$
- for every thread t , $W_x^\sigma(t) = C_e(t)$, where e is the last event in σ with $\text{thr}(e) = t$ and $\text{op}(e) = \mathbf{w}(x)$

Proof (Sketch). Can be proved by induction on the length of the trace.

QED.

Lemma 6.6. Let σ be a trace, e be the last event in σ and let ρ be the prefix of σ that ends before e (i.e., $\sigma = \rho \circ e$). Also assume that $\text{pred}_\sigma(e) = f$ (and is thus defined). Then, there is an event $e' \in \text{Reads}_\rho(x)$ (resp. $e' \in \text{Writes}_\rho(x)$) such that $e' \not\leq_{\text{SHB}}^\sigma f$ iff, $R_x^\rho \not\sqsubseteq C_f$ (resp. $W_x^\rho \not\sqsubseteq C_f$).

The proof of Lemma 6.6 is similar to Lemma 3.6 and thus skipped.

Using all the lemmas above, we can show that the following holds.

Theorem 6.3. Let e be a read/write event $e \in \text{Events}_\sigma$. Algorithm 6.1 reports a race at e iff there is an event $e' \in \text{Events}_\sigma$ such that (e', e) is an $\leq_{\text{HB}}^\sigma$ -schedulable race.

The following theorem states that the asymptotic time and space requirements for Algorithm 6.1 are the same as that of the standard HB algorithm.

Theorem 6.4. Algorithm 6.1 runs in time $O(\mathcal{N}\mathcal{T} \log \mathcal{N})$ and uses $O((\mathcal{V} + \mathcal{L} + \mathcal{T})\mathcal{T} \log \mathcal{N})$ space.

Proof. Observe that for a trace of length $\mathcal{N}$, every component of each of the vector clocks is bounded by $\mathcal{N}$. Thus, each vector clock takes space $O(\mathcal{T} \log \mathcal{N})$, where $\mathcal{T}$ is the number of threads. We have a vector clock for each thread, lock, and variable, which gives us a space bound of $O((\mathcal{V} + \mathcal{T} + \mathcal{L})\mathcal{T} \log \mathcal{N})$. Notice that to process any event we need to update constantly many vector clocks. The time to update any single vector clock can be bounded by its size $O(\mathcal{T} \log \mathcal{N})$. Thus, the total running time is $O(\mathcal{N}\mathcal{T} \log \mathcal{N})$. QED.

6.4.5 Differences from the HB algorithm

While the spirit of Algorithm 6.1 is similar to standard HB vector clock algorithms (such as DJIT⁺ [48], Algorithm 3.2), it differs from them in the following ways. The most important difference is that our algorithm remains sound even beyond the first race, in contrast to DJIT⁺, which can lead to false positives if run beyond the first race. Next, we maintain an additional vector clock LW_x to track the timestamp of the last event that writes to memory location x (Line 30), and use this clock to correctly update $\mathbb{C}_t$ (Line 23). This difference is a direct consequence of the additional ordering edges in the $\leq_{\text{SHB}}$ partial order—every read event e is ordered after the event $\text{lastWrite}(e)$, unlike $\leq_{\text{HB}}$. For the same reason, ‘local’ component of the clock $\mathbb{C}_t$ is also incremented after every write event, in addition to after a release or a fork event (in contrast with DJIT⁺).

Algorithm 6.2: *Epoch Optimization for Algorithm 6.1*

<pre> 1 handler read(t, x) 2 if $\mathbb{W}_x \not\sqsubseteq \mathbb{C}_t$ then 3 declare 'race with write' 4 $\mathbb{C}_t := \mathbb{C}_t \sqcup \mathbb{LW}_x$ 5 if $\mathbb{R}_x$ is an epoch $c@u$ then 6 if $c \leq \mathbb{C}_t(u)$ then 7 $\mathbb{R}_x := \mathbb{C}_t(t)@t$ 8 else 9 $\mathbb{R}_x := \perp[t \mapsto \mathbb{C}_t(t)][u \mapsto c]$ 10 else 11 $\mathbb{R}_x := \mathbb{R}_x[t \mapsto \mathbb{C}_t(t)]$ </pre>	<pre> 12 handler write(t, x) 13 if $\mathbb{R}_x \not\sqsubseteq \mathbb{C}_t$ then 14 declare 'race with read' 15 if $(\mathbb{W}_x \sqsubseteq \mathbb{C}_t)$ then 16 $\mathbb{W}_x := \mathbb{C}_t(t)@t$ 17 else 18 declare 'race with write' 19 if $\mathbb{W}_x$ is an epoch $c@u$ then 20 $\mathbb{W}_x := \perp[t \mapsto \mathbb{C}_t(t)][u \mapsto c]$ 21 else 22 $\mathbb{W}_x(t) := \mathbb{C}_t(t)$ 23 $\mathbb{LW}_x := \mathbb{C}_t$ 24 $\mathbb{W}_x(t) := \mathbb{C}_t(t)$ 25 $\mathbb{C}_t(t) := \mathbb{C}_t(t) + 1$ </pre>
--	--

6.4.6 Epoch Optimization

Recall that an epoch is a pair of an integer c and a thread t , denoted by $c@t$ and is 'equivalent' to the vector time $\perp[t \mapsto c]$ and comparison between a vector clock and epoch can be performed as follows : $c@t \sqsubseteq V$ iff $c \leq V(t)$.

Algorithm 6.2 applies the epoch optimization to Algorithm 6.1. Here, similar to the FASTTRACK algorithm, we allow clocks $\mathbb{R}_x$ and $\mathbb{W}_x$ to be adaptive, while other clocks ($\mathbb{C}_t$, $\mathbb{L}_\ell$ and $\mathbb{LW}_x$) always store vector times. The optimization only applies to the **read** and **write** handlers and thus we omit the other handlers from the description as they are same as those described in Algorithm 6.1. We also omit the **Initialization()** procedure which only differs in that the $\mathbb{R}_x$ and $\mathbb{W}_x$ are initialized to the epoch $0@0$.

Depending upon how these clocks compare with the thread's clock $\mathbb{C}_t$, the clocks switch back and forth between epoch and vector time values:

- Initially, both $\mathbb{R}_x$ and $\mathbb{W}_x$ are assigned the epoch $0@0$. The element $0@0$ can be thought of as the analogue of $\perp$.
- The clock $\mathbb{W}_x$ is *fully adaptive* — it can switch back and forth between vector and epoch times depending upon how it compares with $\mathbb{C}_t$. Notice that, in the FASTTRACK algorithm proposed in [52], the clock $\mathbb{W}_x$ is always an epoch. The underlying assumption for such a simplification is that all the events that write to a given memory location are totally ordered with respect to $\leq_{\text{HB}}$. This assumption, however, need not

hold beyond the first HB race. After the first race is encountered, two $\mathbf{w}(x)$ events e and e' may be unordered by both $\leq_{\text{HB}}$ and $\leq_{\text{SHB}}$. In Algorithm 6.2, $\mathbb{W}_x$ has an epoch representation if and only if the last write event e on x is such that $e' \leq_{\text{SHB}} e$ for every event e' of the type $\langle \cdot, \mathbf{w}(x) \rangle$ in the trace seen so far. When performing a write event e of type $\langle t, \mathbf{w}(x) \rangle$, if $\mathbb{W}_x$ satisfies $\mathbb{W}_x \subseteq \mathbb{C}_t$ (Line 15), then the event e is ordered after all previous $\mathbf{w}(x)$ events, and thus, in this case, $\mathbb{W}_x$ is converted to an epoch representation independent of its original representation (see Line 16). Otherwise, there are at least two $\mathbf{w}(x)$ events that are not ordered by $\leq_{\text{SHB}}$ and thus $\mathbb{W}_x$ becomes a full-fledged vector clock (Line 20 and Line 22).

- The clock $\mathbb{R}_x$ is only *semi-adaptive* — we do not switch back to epoch representation once the clock $\mathbb{R}_x$ takes up a vector-time value. The clock $\mathbb{R}_x$ is initialized to be an epoch. When processing a read event $e = \langle t, \mathbf{r}(x) \rangle$, if the algorithm determines that there is a read event $e' = \langle t', \mathbf{r}(x) \rangle$ observed earlier such that $e' \not\leq_{\text{SHB}} e$, then the clock $\mathbb{R}_x$ takes a vector-time representation. After this point, $\mathbb{R}_x$ stays in the vector clock representation forever. The $\mathbb{R}_x$ clock is an epoch only if all the reads of x observed are ordered totally by $\leq_{\text{SHB}}$. Thus, in order to determine if $\mathbb{R}_x$ can be converted back to an epoch representation, one needs to check if $(\mathbb{R}_x \subseteq \mathbb{C}_t)$ every time a read event is processed. Since this is an expensive additional comparison and because most traces from real-world examples are dominated by read events, we avoid such a check and force $\mathbb{R}_x$ to be only semi-adaptive. This is similar to the FASTTRACK algorithm.

As with FASTTRACK, the epoch optimization for $\leq_{\text{SHB}}$ is sound and does not lead to any loss of precision — the optimized algorithm (Algorithm 6.2) declares a race at an event e iff the corresponding unoptimized algorithm (Algorithm 6.1) declares a race at e .

One must however note that the new clock $\mathbb{LW}_x$ does not have an adaptive representation, and is always required to be a vector clock. One can think of $\mathbb{LW}_x$ to be similar to, say, the clocks $\mathbb{L}_\ell$. These clocks are used to maintain the partial order, unlike the clocks $\mathbb{R}_x$ or $\mathbb{W}_x$ which are only used to check for races. Thus, one needs the full generality of vector times for $\mathbb{LW}_x$.

6.5 EXPERIMENTS

We first describe our implementation to detect $\leq_{\text{HB}}$ -schedulable races. We then present a brief description of the chosen benchmarks and finally the results of evaluating our implementation on these benchmarks.

6.5.1 Implementation

We have implemented our SHB-based race detection algorithms (Algorithm 6.1 and Algorithm 6.2) in our tool RAPID, which is publicly available at [86]. RAPID is written in Java and supports analysis on traces generated by the instrumentation and logging functionality provided by RVPredict [85] to generate traces from Java programs. The traces generated by RVPredict contain read, write, fork, join, acquire and release events. We assume that the traces are sequentially consistent, similar to the assumption made by [44]. We compare the performance of five dynamic race detection algorithms to demonstrate the effectiveness of SHB-based sound reasoning:

HB We implemented the DJIT⁺ algorithm for computing the $\leq_{\text{HB}}$ -partial order and detecting HB-races, in our tool RAPID. As with popular implementations of DJIT⁺, our implementation of DJIT⁺ discovers all $\leq_{\text{HB}}$ -unordered pairs of conflicting events. This serves as a base line to demonstrate how many false positives would result, if one considered all HB-races (instead of $\leq_{\text{HB}}$ -schedulable races). This algorithm is same as Algorithm 3.2 except that instead of doing an **exit** after a race is discovered, we continue discovering more races.

SHB This is the implementation of Algorithm 6.1 in our tool RAPID. The soundness guarantee of Algorithm 6.1 (Theorem 6.3) ensures that our implementation reports only (and all) $\leq_{\text{HB}}$ -schedulable races and thus reports no false alarms. As pointed out in Section 6.3, $\leq_{\text{SHB}}$ timestamps can be used for constructing a witness trace for a given $\leq_{\text{HB}}$ -schedulable race. We defer this functionality to future work.

FHB This is the algorithm that mimics a software developer’s strategy when using HB-race detectors. This algorithm is a slight variant of the DJIT⁺ algorithm and is implemented in our tool RAPID. Every time an HB-race is discovered, the algorithm force orders the events in race, before analyzing subsequent events in the trace. When processing a read event $e = \langle t, \mathbf{r}(x) \rangle$, if the algorithm discovers a race (that is, if the check $\neg(\mathbb{W}_x \sqsubseteq \mathbb{C}_t)$ passes), the algorithm reports a race and also updates the clock $\mathbb{C}_t$ as $\mathbb{C}_t := \mathbb{C}_t \sqcup \mathbb{W}_x$. Similarly, at a write event, the algorithm updates the clock $\mathbb{C}_t$ as $\mathbb{C}_t := \mathbb{C}_t \sqcup \mathbb{R}_x$ (resp. $\mathbb{C}_t := \mathbb{C}_t \sqcup \mathbb{W}_x$) if the check $\neg(\mathbb{R}_x \sqsubseteq \mathbb{C}_t)$ (resp. $\neg(\mathbb{W}_x \sqsubseteq \mathbb{C}_t)$) passes. This algorithm is sound — all races reported by this algorithm are schedulable, but it may fail to identify some races that are schedulable. The complete description of FHB is presented in Algorithm 6.3.

RVPredict RVPredict’s race detection technology relies on maximal causal models [44].

RVPredict is sound and does not report any false alarms. Besides, RVPredict, at least

Algorithm 6.3: *Vector Clock for FHB race detection*

```

1 function Initialization()
2   foreach  $t \in \text{Threads}$  do
3      $\mathbb{C}_t := \perp[t \mapsto 1]$ 
4   foreach  $\ell \in \text{Locks}$  do
5      $\mathbb{L}_\ell := \perp$ 
6   for  $x \in \text{Vars}$  do
7      $\mathbb{R}_x := \perp$ 
8      $\mathbb{W}_x := \perp$ 
9 handler acquire( $t, \ell$ )
10   $\mathbb{C}_t := \mathbb{C}_t \sqcup \mathbb{L}_\ell$ 
11 handler release( $t, \ell$ )
12   $\mathbb{L}_\ell := \mathbb{C}_t$ 
13   $\mathbb{C}_t := \mathbb{C}_t[t \mapsto \mathbb{C}_t(t) + 1]$ 
14 handler fork( $t, u$ )
15   $\mathbb{C}_u := \mathbb{C}_t[u \mapsto 1]$ 
16   $\mathbb{C}_t := \mathbb{C}_t[t \mapsto \mathbb{C}_t(t) + 1]$ 
17 handler join( $t, u$ )
18   $\mathbb{C}_t := \mathbb{C}_t \sqcup \mathbb{C}_u$ 
19 handler read( $t, x$ )
20  if  $\mathbb{W}_x \not\sqsubseteq \mathbb{C}_t$  then
21    declare ‘race with write’
22     $\mathbb{C}_t := \mathbb{C}_t \sqcup \mathbb{W}_x$  (* force ordering *)
23     $\mathbb{R}_x := \mathbb{R}_x[t \mapsto \mathbb{C}_t(t)]$ 
24 handler write( $t, x$ )
25  if  $\mathbb{R}_x \not\sqsubseteq \mathbb{C}_t$  then
26    declare ‘race with read’
27     $\mathbb{C}_t := \mathbb{C}_t \sqcup \mathbb{R}_x$  (* force ordering *)
28  if  $\mathbb{W}_x \not\sqsubseteq \mathbb{C}_t$  then
29    declare ‘race with write’
30     $\mathbb{C}_t := \mathbb{C}_t \sqcup \mathbb{W}_x$  (* force ordering *)
31   $\mathbb{W}_x := \mathbb{W}_x[t \mapsto \mathbb{C}_t(t)]$ 

```

in theory, guarantees to detect more races than any other sound race prediction tool, and thus more races than Algorithm 6.1 theoretically. **RVPredict** encodes the problem of race detection as a logical formula and uses an SMT solver to check for races. **RVPredict** can analyze the traces generated using its logging functionality, and thus is a natural choice for comparison.

Besides the vector clock algorithms (**HB**, **SHB**, **FHB**) described above, we also implemented the epoch optimizations for **HB** and **SHB** in **RAPID**.

6.5.2 Benchmarks

We measure the performance of our algorithms against traces drawn from a wide variety of benchmark programs that have previously been used to measure the performance of other race detection tools [13, 34]. and have been derived from the IBM Contest benchmark suite [87], the Java Grande Forum benchmark suite [88] and obtained from the DaCaPo benchmark suite (version 9.12) [89]. In Table 6.1, we also describe the characteristics of the generated traces that we use for analyzing our algorithms. The number of threads range from 3-12, the number of lock objects can be as high as 8K. The distinct memory locations accessed (Column 5) in the traces can go as high as 10M. The traces generated are dominated

Table 6.1: Benchmarks and metadata of the traces generated. Columns 1 and 2 describe the name and the lines of code in the source code of the chosen benchmarks. Column 3, 4 and 5 describe respectively the number of *active* threads, locks, and memory locations in the traces generated by the corresponding program in Column 1. Column 6 reports the total number of events in the trace. Columns 7, 8, 9, 10 and 11 respectively denote the number of read, write, acquire (or release), fork and join events.

1	2	3	4	5	6	7	8	9	10	11
Program	LOC	Thrds	Locks	Vars	Events					
					Total	Read	Write	Synch.	Fork	Join
airlinetickets	83	4	0	44	137	77	48	0	12	0
array	36	3	2	30	47	8	30	3	3	0
bufwriter	199	6	1	471	22.2K	15.8K	3.6K	1.4K	6	4
bubblesort	274	12	2	167	4.6K	4.0K	404	121	27	0
critical	63	4	0	30	55	18	31	0	4	2
mergesort	298	5	3	621	3.0K	2.0K	914	55	5	3
pingpong	124	6	0	51	147	57	71	0	19	0
moldyn	2.9K	3	2	1.2K	200.0K	182.8K	17.2K	31	3	1
raytracer	2.9K	3	8	3.9K	15.8K	10.4K	5.3K	60	3	1
derby	302K	4	1112	185.6K	1.3M	879.5K	404.5K	31.2K	4	2
ftpserver	32K	11	301	5.5K	49.0K	30.0K	7.8K	5.6K	11	4
jigsaw	101K	11	275	103.5K	3.1M	2.6M	413.5K	5.9K	13	4
xalan	180K	6	2491	4.4M	122.0M	101.7M	18.3M	1M	7	5
lusearch	410K	7	118	5.2M	216.4M	162.1M	53.9M	206.6K	7	0
eclipse	560K	14	8263	10.6M	87.1M	72.6M	12.9M	765.4K	16	3
Total					430.3M	340.2M	85.9M	2.1M	140	29

by access events, with the majority of events being read events (compare Columns 6, 7 and 8).

6.5.3 Setup

Our experiments were conducted on an 8-core 2.6GHz 46-bit Intel Xeon(R) Linux machine, with HotSpot 1.8.0 64-Bit Server as the JVM and 50 GB heap space. Using RVPredict’s logging functionality, we generated one trace per benchmark and analyzed it with the various race detection engines.

Our evaluation is broadly designed to evaluate our approach based on the following aspects:

1. **Reducing false positives:** HB based analysis can report false races beyond the first race discovered, while SHB based reasoning does not. We would like to gauge how many false positives are ruled out by SHB.
2. **Prediction power:** Both FHB and RVPredict are alternate extremes for alleviating HB’s unsoundness beyond the first data race. We would like to compare them with

the predictive power of SHB.

3. **Scalability:** While Algorithm 6.1 runs in linear time, it tracks additional clocks ($\mathbb{LW}_x$ for every memory location x accessed in the trace) over the standard HB vector clock algorithm. Since this can potentially slow down analysis, we would like to evaluate the performance overhead due to this additional book-keeping.
4. **Epoch optimization:** The standard epoch optimization popularized by FASTTRACK [52] is designed to work for the case when all the writes to a memory location are totally ordered. While this is true until the first race is discovered, this condition may not be guaranteed after the first race. We will evaluate the effectiveness of our adaptation of this optimization to work beyond the first race.

6.5.4 Evaluation

Our experimental results are summarized in Table 6.1, Table 6.2 and Table 6.3. Table 6.1 describes information about generated execution logs. Table 6.2 depicts the number of races and warnings raised by the different race detection algorithms. Columns 3-7 in Table 6.2 report the number of distinct pairs (pc_1, pc_2) of program locations corresponding to an identified data race. That is, for every *event* race pair (e_1, e_2) identified by the different race detection algorithms, we identify the pair of *program locations* that give rise to this event pair and report the total number of such program location pairs (counting the pairs (pc_1, pc_2) and (pc_2, pc_1) only once). Since each of the vector clock algorithms (HB, SHB and FHB) only report whether the event being processed is in race with some earlier event, we need to perform a separate analysis step using the vector timestamps, to determine the actual pair of events (and thus the corresponding pair of program locations) in race. In Columns 9, 10 and 11 in Table 6.2 we report the number of *warnings* raised by the three vector clock algorithms—HB, SHB and FHB respectively. A warning is raised when at a read/write event e , we determine if the event e is in race with an earlier event, counting multiple warnings for a single event only once.

In Table 6.3, Columns 2, 5, 8, 9 and 10 respectively report the time taken by different analyses engines — HB, SHB, FHB and RVPredict— on the trace generated. We also measure the time taken by the epoch optimizations for both HB and SHB vector clock algorithms (Columns 3 and 6 respectively) and report the speedup thus obtained over the naïve vector clock algorithms. (Columns 4 and 7 respectively). When analyzing the generated traces using WCP, we filter out events that are thread local; this does not affect any races. The memory requirement of a naïve vector clock algorithm for WCP, as described in [13] can be

Table 6.2: Number of races detected and warnings raised. Column 1 and 2 denote the benchmarks and the size of the traces generated. Columns 3, 4 and 5 respectively report the number of distinct program location pairs for which there are pair of events in a race, as identified by HB, SHB and FHB. Columns 6 and 7 denote the races reported by RVPredict when run with the parameters (window-size=1K, solver-timeout=60s) and (window-size=10K, solver-timeout=240s). Columns 9, 10 and 11 respectively denote the number of warnings generated when running the vector clock algorithms for detecting races using HB (unsound), SHB (sound and complete for HB-schedulable races) and FHB (naive algorithm that forces an order after every race discovered) analyses.

1	2	3	4	5	6	7	8	9	10
		Races					Warnings		
Program	#Events	HB	SHB	FHB	RVPredict		HB	SHB	FHB
						1K/60s	10K/240s		
airlinetickets	137	6	6	3	6	6	8	8	5
array	47	0	0	0	0	0	0	0	0
bufwriter	22.2K	2	2	2	2	0	8	8	8
bubblesort	4.6K	6	6	6	6	0	602	269	100
critical	55	8	8	1	8	8	3	3	1
mergesort	3.0K	3	1	1	1	2	52	1	1
pingpong	147	3	3	3	3	3	11	8	8
moldyn	200.0K	44	2	2	2	2	24657	103	103
raytracer	15.8K	3	3	3	2	3	118	8	8
derby	1.3M	26	13	11	12	-	89	29	28
ftpserver	49.0K	35	23	22	10	12	143	69	69
jigsaw	3.1M	8	4	4	4	2	14	4	4
xalan	122.0M	16	12	10	8	8	86	31	21
lusearch	216.4M	160	52	28	0	0	751002	232	119
eclipse	87.1M	64	61	31	5	0	173	164	103
Total	430.3M	384	196	127	69	46	776966	937	578

a bottleneck and removing thread local events allowed us to analyze the larger traces (xalan, lusearch and eclipse) without any memory blowup. We next discuss our results in detail.

Reducing False Positives. First, observe that both the number of races reported (Columns 3, 4 and 5 in Table 6.2) and the number of warnings raised (Columns 8, 9 and 10) by HB, SHB and FHB are monotonically decreasing, as expected — HB detects all $\leq_{\text{HB}}$ -schedulable races but additional false races, SHB detects exactly the set of $\leq_{\text{HB}}$ -schedulable races and FHB detects a subset of $\leq_{\text{HB}}$ -schedulable races. Next, the number of races reported by HB can be way higher than the actual number of $\leq_{\text{HB}}$ -schedulable races (see moldyn and lusearch). Similarly, the number of warnings raised can be an order of magnitude larger than those raised by either SHB or FHB. Clearly, many of these warnings are potentially spurious. Thus, an incorrect use of the popular HB algorithm can severely overwhelm developers and

hamper their productivity.

While Theorem 6.2 guarantees that each of the additional race pairs reported by HB (over those reported by SHB) cannot be scheduled in any correct reordering of the observed trace that respects the induced $\leq_{\text{HB}}$ partial order, it does not guarantee that these extra races are not predictable races nevertheless. In order to see if the extra races reported by HB (Column 3 in Table 6.2) can be scheduled in a correct reordering that does not respect the $\leq_{\text{HB}}$ order induced by the observed trace, we manually inspected the traces (annotated with their vector timestamps) of `mergesort`, `moldyn`, `derby`, `ftpserver`, and `jigsaw`. In each of these benchmarks, we found that all the extra race pairs reported by HB indeed *cannot* be scheduled in *any* correct reordering (even in correct reorderings not respecting $\leq_{\text{HB}}$).

Prediction Power. The naïve algorithm FHB, while sound, can miss a lot of real races (Column 5 in Table 6.2) and has a poor prediction power as compared to the sound SHB algorithm. See for example, `lusearch` and `eclipse` where FHB misses almost half the races reported by SHB. Next, observe that while RVPredict, in theory, is *maximally sound*, it can miss a lot of races, sometimes even more than the naïve FHB strategy (Columns 6 and 7 in Table 6.2). This is because RVPredict relies on SAT solving to determine data races. As a result, in order to scale to large traces obtained from real world software, RVPredict resorts to *windowing* — dividing the trace into smaller chunks and restricting its analysis to these smaller chunks. This strategy, while useful for scalability, can miss data races that are spread far across in the trace, yet can be captured using happens-before like analysis. Besides, since the underlying DPLL-based SAT solvers may not terminate within reasonable time, RVPredict sets a timeout for the solver — this means that even within a given window, RVPredict can miss races if the SAT solver does not return an answer within the set timeout. All these observations clearly indicate the power of $\leq_{\text{SHB}}$ -based reasoning.

Scalability. First, the size of the traces, that SHB and the other three linear time vector clock algorithms can handle, can be really large, of the order of hundreds of millions (`xalan`, `lusearch`, etc.). In contrast, RVPredict fails to scale for large traces, even after employing a windowing strategy. This is especially pronounced for the larger traces (`bufwriter`, `derby-xalan`). This suggests the power of using a linear time vector clock algorithm for dynamic race detection for real-world applications. The small and medium sized examples almost always finish with a few seconds for each of HB, SHB, FHB and WCP. The larger examples `xalan`, `lusearch` and `eclipse` can take as much as 4-10 minutes for the vector clock algorithms and 6-15 minutes for RVPredict’s analysis (window size of 10K, solver timeout of 4 minutes).

Epoch Optimization. The epoch optimization is indeed effective in improving the per-

Table 6.3: Time taken by different race detection algorithms on traces generated by the corresponding programs in Column 1. Column 2 denotes the taken for analyzing the entire trace with the DJIT⁺ vector-clock algorithm. Column 3 denotes the time taken by FASTTRACK-style optimization over the basic DJIT⁺ and Column 4 denotes the speedup thus obtained. Column 5 denotes the times for the vector clock implementation of Algorithm 6.1. Column 6 and 7 denote the time and speedup due to the epoch optimization for SHB (Algorithm 6.2). A ‘-’ in Columns 5 and 8 denote a downgraded performance due to epoch optimization. Column 8 denotes the time to analyze the traces using FHB (forcing an order in HB) analysis. Column 9 and 10 respectively denote the time taken by RVPredict using the parameters (window-size=1K, solver-timeout=60s) and (window-size=10K, solver-timeout=240s). A ‘-’ in Column 11 denotes that RVPredict did not finish within the set time limit of 4 hours.

1	2	3	4	5	6	7	8	9	10
	HB			SHB			FHB	RVPredict	
Program	VC (s)	Epoch (s)	Speed-up	VC (s)	Epoch (s)	Speed-up	(s)	1K/60s (s)	10K/240s (s)
airlinetickets	1.11	1.4	-	2.08	0.25	8.32x	2.33	1.34	1.33
array	1.13	1.66	-	0.55	0.25	2.2x	1.98	1.28	1.26
bufwriter	0.54	1.63	-	0.56	0.46	1.22x	0.96	3.61	1879
bubblesort	0.47	0.29	1.62x	0.78	0.34	2.3x	0.51	2.46	652
critical	0.68	0.24	2.8x	0.28	0.56	-	0.54	2.14	0.52
mergesort	0.4	1.58	-	0.43	0.35	1.23x	0.7	0.67	0.81
pingpong	0.31	0.31	1x	2.15	0.33	6.5x	0.41	2.3	0.57
moldyn	1.45	2.78	-	1.04	1.49	-	1.77	2.27	2.94
raytracer	0.95	2.25	-	1.82	0.41	4.44x	2.33	0.61	8.61
derby	4.27	2.91	1.46x	4.39	3.05	1.44x	5.56	72	-
ftpserver	0.84	0.84	1x	0.87	1.33	-	2.94	1.23	164
jigsaw	23	8.1	2.84x	10.04	7.35	1.37x	9.31	1.66	245
xalan	217	152	1.42x	222	184	1.21x	291	44	420
lusearch	362	383	-	444	337	1.32x	325	48	47
eclipse	525	200	2.63x	238	188	1.27x	168	25	951

formance of vector-clock algorithms even when all the write events to a memory location may not be totally ordered. The speedups vary from 1.2x to 8.3x on small and medium sized benchmarks and from 1.2x to 2.9x on larger traces. The speedup obtained for HB race detection is, in general, less than in the case of SHB algorithm. This is expected since SHB is strictly stronger than HB— every pair of events ordered by $\leq_{\text{HB}}$ is also ordered by $\leq_{\text{SHB}}$, and not every pair of events ordered by $\leq_{\text{SHB}}$ may be ordered by $\leq_{\text{HB}}$. As a result, a write event can get ordered after all previous write events more frequently when using $\leq_{\text{SHB}}$ than when using $\leq_{\text{HB}}$. This means that in the epoch optimization for SHB, the $\mathbb{W}_x$ clocks take up epoch representation more frequently than in the HB algorithm, and this difference is reflected in Columns 5 and 8 of Table 6.3.

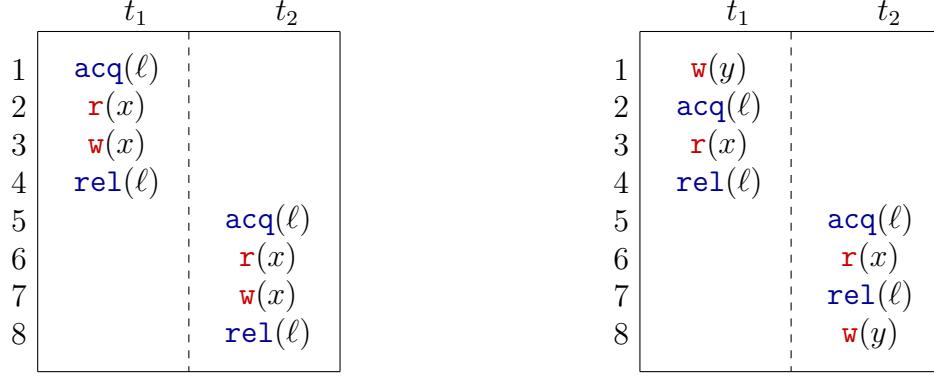
CHAPTER 7: SOUND LINEAR TIME DATA RACE PREDICTION USING WEAK CAUSAL PRECEDENCE

In Chapter 6, we showed that race prediction based on the **HB** partial order is not sound beyond the first **HB**-race. We proposed the partial order **SHB** to enhance the prediction power of **HB** beyond the first race. **SHB** based race detection, reports all **HB**-races which are witnessed by a correct reordering that respects the $\leq_{\text{HB}}$ partial order on the original observed trace. In this chapter again, we consider the problem of data race prediction, but instead consider the problem of predicting data races that **HB** can miss.

Constraint solving based techniques [43,44] can theoretically be employed for this purpose, but either do not scale or resort to windowing which can miss simple races that even **SHB** can catch. Instead, we are interested in a partial order based approach for race detection. As in $\leq_{\text{HB}}$, we want to order the events of an observed trace in a partial ordering $\leq_{\text{P}}$, that, intuitively captures *causality* — when two events are ordered by $\leq_{\text{P}}$ in the trace, then we order these two events in all reorderings. In this sense, the notion of *causality* that **HB** captures is such that it does not allow inverting the order of critical sections on the same lock — **HB** orders them in the order in which they appear in the trace. Doing so ensures that they don't overlap in any **HB**-respecting correct reordering. Nevertheless, this leads to a conservative notion of causality. On the other hand, partial order $\leq_{\text{P}}$ that naively disregards such orderings may lead to unsound reasoning. This is because a linearization ρ of partial order $\leq_{\text{P}}$ defined over the set of events of a trace σ (i) might not be a valid trace as it might violate lock semantics, or (ii) might not be a correct reordering of σ as there might be a read event whose corresponding last write event in ρ does not match with that in σ . Therefore, two conflicting events unordered by a partial order only indicates the possibility of, and does not necessarily guarantee the existence of a predictable race. The cleverness lies in designing partial orders for which this possibility is indeed a guarantee.

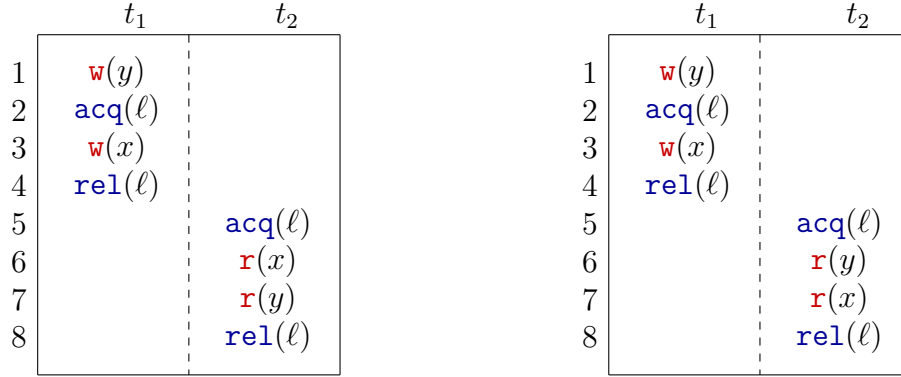
Consider, for example, the trace σ_1 shown in Figure 7.1a. The order of events e_4 and e_5 cannot be inverted in any correct reordering because the read event e_6 must occur after $e_3 = \text{lastWrite}_{\sigma_1}(e_6)$ in any correct reordering. Had the two $\text{w}(x)$ events e_3 and e_7 been absent, we could have actually swapped the two critical sections temporally to get a feasible alternate execution. For example, the trace σ_2 in Figure 7.1b can be reordered to expose a race on the access of y by performing the critical section of t_2 before the other. However, this race cannot be predicted using the **HB** partial order as $e_1 \leq_{\text{HB}}^{\sigma_2} e_8$. Naively deciding to not order critical sections in either of the two traces will make (e_3, e_6) unordered in both traces, even though they cannot be in race (they are protected by a common lock).

The partial order Causally-Precedes (**CP**) [34] was introduced to detect races missed by



(a) Trace σ_1 : Cannot swap critical sections. (b) Trace σ_2 : Can swap critical sections.

Figure 7.1: Traces showing when critical sections can/cannot be ordered.



(a) Trace σ_3 : No predictable race.

(b) Trace σ_4 : Predictable race.

Figure 7.2: Example traces showing how CP misses race due to small change.

HB such as those in Figure 7.1 while being sound. The CP relation is a subset of HB, which implies that CP can detect races above and beyond those detected by HB. Soundness of CP guarantees that if a trace has an CP race, then it either has a predictable race or a predictable deadlock. But there are two main drawbacks of CP. Firstly CP misses races that are predictable. Consider the traces shown in Figure 7.2; the only difference between the two traces is that events e_6 and e_7 have been swapped. There is no predictable race in trace σ_3 (Figure 7.2a) because the event e_6 in t_2 must be performed after $e_3 = \text{lastWrite}_{\sigma_3}(e_6)$ in t_1 , which prevents the critical sections from being reordered. On the other hand, trace σ_4 from Figure 7.2b has a predictable race on y witnessed by the correct reordering $\rho = e_5$ in which both e_1 and e_6 are enabled CP, however, does not detect a race in either trace, because it is agnostic to the ordering of events within a critical section. The second drawback of CP is that, while it can be detected in polynomial time [34], there is no known linear time algorithm. This severely hampers its use on real-world examples with traces several

gigabytes large. So any implementation of CP must resort to *windowing*, where the trace is partitioned into small fragments. This means, as with SMT solving based techniques, CP can only find races within bounded fragments of the trace, and so detects fewer races than what CP promises. Our experiments (Section 5.4) reveal windowing to be a serious impediment to race detection in large examples.

In this chapter we discuss the partial order Weak Causally Precedes (WCP) [13] (sometimes referred to as Weak Causal Precedence), which addresses the two drawbacks of CP in one shot. As the name suggests, WCP is a weakening of the CP partial order. Thus WCP detects all races that CP does and even more (like the race in Figure 7.2b as explained in Section 7.1.1). We prove that WCP enjoys the same soundness guarantees as CP. Additionally, like HB, WCP admits a linear time vector clock algorithm for race detection, thus solving the main open problem proposed in [34]. This is surprising because when HB was weakened to obtain CP it resulted in detecting more races but came at the cost of an expensive algorithm. But weakening CP to WCP not only allows for detecting more races but enables an efficient algorithm.

This material presented in this chapter has previously been published in [13] and we closely follow its presentation here. In this chapter, we discuss the following.

1. We define a new *sound* relation weak-causally-precedes (WCP), which is weaker than CP. The definition and its soundness proof, as claimed in [34], is challenging. Even though we had the benefit of following the work on CP, our experience concurs with the above observation. The subtlety in these ideas is highlighted by the fact that the soundness proof for CP, presented in [34], is *incorrect*; informed readers can find an explanation of the errors in the CP soundness proof in a detailed technical report [91].
2. We present *linear* time algorithm for dynamic data race prediction, based on the WCP partial order. It is based on searching for conflicting events that are unordered by WCP. We prove that our algorithm is *correct*. Our algorithm uses linear space in the worst case, as opposed to the logarithmic space requirement of happens-before vector clock algorithm. However, in our experiments, we did not encounter these worst case bounds. Our algorithm scales to traces with hundreds of millions of events and the memory usage stays below 3% for most benchmarks (see Table 7.1). We further show that our algorithm is *optimal* in terms of its asymptotic running time and memory usage.
3. Our experiments show the benefits of our algorithm, in terms of the number of races detected and its efficiency when compared to state of the art tools such as RVPredict.

They reveal the tremendous power of being able to analyze the entire trace as opposed to trace fragments that other sound race detection algorithms (other than those based on HB) are forced to be restricted to.

7.1 WEAK CAUSAL PRECEDENCE

In this section we illustrate the CP partial order [34] and contrast it with our contribution, the WCP partial order.

One of the main assumptions we will be adhering to is that traces obey **well-nested locking principle** as formally defined in Chapter 2. Informally, this means that locks are released in the inverse order of their acquisition.

We will now describe partial orders HB, CP (from literature) and WCP (our contribution).

The Causally-Precedes (CP) partial order was then introduced by Smaragdakis et al [34] as an improvement over HB. CP is a subset of HB i.e it has fewer orderings and hence more possible interleavings. It is thus able to detect more races than HB. CP based race detection is proved to be *weakly sound* — if CP reports a race, then the trace either has a predictable race or a predictable deadlock. However, it is not known if CP can be computed in linear time. The race detection algorithm for CP is polynomial time but not linear.

In the following, we say two acquire events $a_1 \neq a_2 \in \text{Acquires}_\sigma$ in a trace σ are said to be conflicting if there is a lock $\ell \in \text{Locks}_\sigma$ so that $\text{op}(a_1) = \text{op}(a_2) = \text{acq}(\ell)$. For two binary relations $R_1, R_2 \subseteq S \times S$ over a set S , the composition $R_1 \circ R_2$ is the binary relation $\{(a, c) \mid (a, b) \in R_1, (b, c) \in R_2\}$.

Definition 7.1 (Causally Precedes). For a trace σ , the CP partial order is $\leq_{\text{CP}}^\sigma = \prec_{\text{CP}}^\sigma \cup \leq_{\text{TO}}^\sigma$, where $\prec_{\text{CP}}^\sigma$ is the smallest transitive order such that the following hold.

- (a) Let $a_1, a_2 \in \text{Acquires}_\sigma$ be any two conflicting acquire events in σ such that $a_1 <_{\text{tr}}^\sigma a_2$. If there are events $e_1 \in \text{CS}_\sigma(a_1)$ and $e_2 \in \text{CS}_\sigma(a_2)$ such that $e_1 \asymp e_2$, then $(\text{match}_\sigma(a_1), a_2) \in \prec_{\text{CP}}^\sigma$.
- (b) Let $a_1, a_2 \in \text{Acquires}_\sigma$ be any two conflicting acquire events in σ such that $a_1 <_{\text{tr}}^\sigma a_2$. If there are events $e_1 \in \text{CS}_\sigma(a_1)$ and $e_2 \in \text{CS}_\sigma(a_2)$ such that $(e_1, e_2) \in \prec_{\text{CP}}^\sigma$, then $(\text{match}_\sigma(a_1), a_2) \in \prec_{\text{CP}}^\sigma$.
- (c) $\prec_{\text{CP}}^\sigma$ composes with $\leq_{\text{HB}}^\sigma$. That is, for any three events $e_1, e_2, e_3 \in \text{Events}_\sigma$, if either $(e_1, e_2) \in \prec_{\text{CP}}^\sigma$ and $e_2 \leq_{\text{HB}}^\sigma e_3$, or $e_1 \leq_{\text{HB}}^\sigma e_2$ and $(e_2, e_3) \in \prec_{\text{CP}}^\sigma$, then $(e_1, e_3) \in \prec_{\text{CP}}^\sigma$. In other words, $\prec_{\text{CP}}^\sigma \circ \leq_{\text{HB}}^\sigma \subseteq \prec_{\text{CP}}^\sigma$ and $\leq_{\text{HB}}^\sigma \circ \prec_{\text{CP}}^\sigma \subseteq \prec_{\text{CP}}^\sigma$.

We will write $e_1 \leq_{\text{CP}}^\sigma e_2$ (resp. $e_1 \prec_{\text{CP}}^\sigma e_2$) to denote $(e_1, e_2) \in \leq_{\text{CP}}^\sigma$ (resp. $(e_1, e_2) \in \prec_{\text{CP}}^\sigma$).

Similar to $\leq_{\text{HB}}$, race detection based on CP relies on identifying events unordered by $\leq_{\text{CP}}$. We say that a pair of conflicting events (e_1, e_2) in a trace σ is a CP-race of σ if $e_1 \parallel_{\text{CP}}^\sigma e_2$. A trace is said to have a CP-race if such a pair exists in the trace. The soundness guarantee of CP then states that if a trace σ does not have deadlock patterns, then the existence of a CP-race in σ implies that σ has a predictable data race. The proof of a slightly different statement was attempted in [34], and was shown to have a bug in [91]. Nevertheless this follows from Lemma 7.1 and Theorem 7.1 which we present later.

As seen in the examples in Figure 7.2, CP is agnostic to the relative order of the events inside the same critical sections. That is, if we were to consider two read/write events that are enclosed within the same set of critical sections in the trace, and we interchanged their positions, then the resulting trace would have exactly the same CP orderings across threads. This is because Rule (a) of CP, that depends upon the position of read/write events, is only concerned with whether or not it occurs inside some critical section and not how they are relatively order within the critical section. This constraint prevents CP it from detecting races, as we saw in Figure 7.2b.

Next we look at WCP, the partial order we proposed in [13]. It is obtained by weakening Rule (a) and Rule (b) of CP as follows, and is formally defined as follows.

Definition 7.2 (Weak Causally Precedes). For a trace σ , the WCP partial order is the union $\leq_{\text{WCP}}^\sigma = \prec_{\text{WCP}}^\sigma \cup \leq_{\text{TO}}^\sigma$, where $\prec_{\text{WCP}}^\sigma$ is the smallest transitive order such that the following hold.

- (a) Let $a_1, a_2 \in \text{Acquires}_\sigma$ be any two conflicting acquire events in σ such that $a_1 <_{\text{tr}}^\sigma a_2$. If there are events $e_1 \in \text{CS}_\sigma(a_1)$ and $e_2 \in \text{CS}_\sigma(a_2)$ such that $e_1 \succ e_2$, then $(\text{match}_\sigma(a_1), e_2) \in \prec_{\text{WCP}}^\sigma$.
- (b) Let $a_1, a_2 \in \text{Acquires}_\sigma$ be any two conflicting acquire events in σ such that $a_1 <_{\text{tr}}^\sigma a_2$ and $\text{match}_\sigma(a_2)$ exists. If there are events $e_1 \in \text{CS}_\sigma(a_1)$ and $e_2 \in \text{CS}_\sigma(a_2)$ such that $(e_1, e_2) \in \prec_{\text{WCP}}^\sigma$, then $(\text{match}_\sigma(a_1), \text{match}_\sigma(a_2)) \in \prec_{\text{WCP}}^\sigma$.
- (c) $\prec_{\text{CP}}^\sigma$ composes with $\leq_{\text{HB}}^\sigma$. That is, for any three events $e_1, e_2, e_3 \in \text{Events}_\sigma$, if either $(e_1, e_2) \in \prec_{\text{WCP}}^\sigma$ and $e_2 \leq_{\text{HB}}^\sigma e_3$, or $e_1 \leq_{\text{HB}}^\sigma e_2$ and $(e_2, e_3) \in \prec_{\text{WCP}}^\sigma$, then $(e_1, e_3) \in \prec_{\text{WCP}}^\sigma$. In other words, $\prec_{\text{WCP}}^\sigma \circ \leq_{\text{HB}}^\sigma \subseteq \prec_{\text{WCP}}^\sigma$ and $\leq_{\text{HB}}^\sigma \circ \prec_{\text{WCP}}^\sigma \subseteq \prec_{\text{WCP}}^\sigma$.

We will write $e_1 \leq_{\text{WCP}}^\sigma e_2$ (resp. $e_1 \prec_{\text{WCP}}^\sigma e_2$) to denote $(e_1, e_2) \in \leq_{\text{WCP}}^\sigma$ (resp. $(e_1, e_2) \in \prec_{\text{WCP}}^\sigma$).

We remark that the order $\prec_{\text{WCP}}^\sigma$ is different from $\leq_{\text{WCP}}^\sigma \setminus \{(e, e) \mid e \in \text{Events}_\sigma\}$. Similarly, $\prec_{\text{CP}}^\sigma$ is different from $\leq_{\text{CP}}^\sigma \setminus \{(e, e) \mid e \in \text{Events}_\sigma\}$.

Note that, Rule (a) of WCP orders the release event before the read/write event (and not the acquire as in CP). Thus, WCP makes a distinction between events of the same thread based on the relative order inside a critical section.

Rule (a) and Rule (b) in WCP are weaker versions of Rule (a) and Rule (b) in CP respectively. As a result, one can inductively prove that $\leq_{\text{WCP}}^\sigma \subseteq \leq_{\text{CP}}^\sigma$:

Lemma 7.1. For a trace σ , $\leq_{\text{WCP}}^\sigma \subseteq \leq_{\text{CP}}^\sigma$. Further, there is a trace σ for which $\leq_{\text{WCP}}^\sigma \neq \leq_{\text{CP}}^\sigma$.

An example of the trace mentioned in the above lemma that distinguishes $\leq_{\text{CP}}$ from $\leq_{\text{WCP}}$ is trace σ_4 from Figure 7.2b. Here, $e_4 \leq_{\text{CP}}^\sigma e_5$ but $e_4 \not\leq_{\text{WCP}}^\sigma e_5$.

Just like HB and CP, race detection based on WCP relies on identifying events unordered by $\leq_{\text{WCP}}$. A pair of conflicting events (e_1, e_2) in a trace σ is said to be a WCP-race if $e_1 \parallel_{\text{WCP}}^\sigma e_2$. From Lemma 7.1, it follows that every CP-race is also a WCP-race. Next we state the soundness theorem for WCP whose proof is provided in [91].

Theorem 7.1 (WCP Soundness). Let σ be a trace such that σ has no deadlock pattern. If σ has a WCP-race, then σ has a predictable data race.

A simple consequence of Theorem 7.1 is that a trace with a WCP-race either has a predictable data race or a deadlock pattern.

7.1.1 Illustrations

Going back to the example in Figure 7.2b let us see how WCP is able to detect a race that CP cannot. Note that the e_4 in t_1 is CP-ordered before e_5 in t_2 by Rule (a) of CP. Further using Rule (c) of CP we obtain that the access events on y (e_1 and e_6) CP-ordered ($e_1 \leq_{\text{CP}}^{\sigma_4} e_6$) and therefore CP does not detect the predictable race as uncovered by the witness correct reordering $\sigma_4^{\text{CR}} = e_5$ in which e_1 and e_6 are enabled simultaneously. WCP on the other hand does not order the $\text{rel}(\ell)$ event e_4 and $\text{acq}(\ell)$ event e_5 . Rule (a) of WCP only orders $e_4 \leq_{\text{WCP}}^{\sigma_4} e_7$. If we look at the example trace σ_3 to its left in Figure 7.2a the same reasoning can be used to obtain that WCP (and thus also CP) does not detect any race, and indeed there is no predictable race. In the case of WCP since the $\text{r}(x)$ event e_6 in t_2 appears above the $\text{r}(y)$ event e_7 , it ends up ordering e_1 and e_7 thus not declaring any race either.

The intuitive reason for formulating Rule (a) in CP is that “*two conflicting events have to occur in the same order in every correctly reordered execution if it is the case that they do not constitute a race. Consequently the critical sections containing them need to be ordered in their entirety*” [34]. This intuition fails correct when the conflicting events do not occur in the correct reordering. In the example in Figure 7.2b the two conflicting accesses on x do not

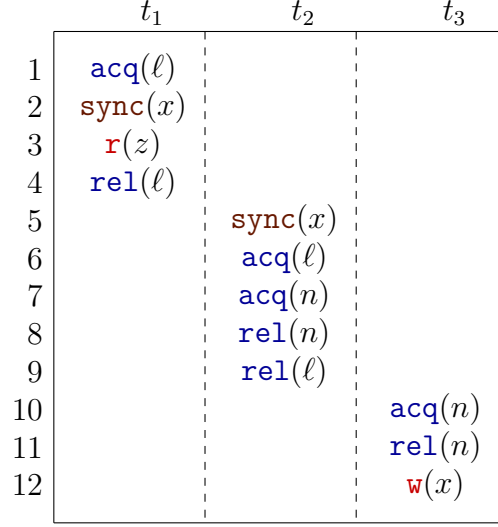


Figure 7.3: Example trace σ_5 to demonstrate how weakening Rule (b) is useful. No race predicted by CP. WCP predicts a race.

appear in the set of events that need to be scheduled in order to reveal a race, and hence any orderings derived from them need not be considered. But then if the read event $\langle t_2, \text{r}(x) \rangle$ is indeed included in some (other) reordering then the $\text{rel}(\ell)$ to $\text{acq}(\ell)$ ordering should be respected. Rule (a) of WCP does not enforce this ordering completely, it only makes sure that the later of the conflicting events is ordered after the earlier release. It would seem that this ordering is not sufficient to enforce lock semantics of the traces we would be interested in, but it turns out that this weaker version is sufficient to guarantee soundness.

We also weaken Rule (b) in the same spirit as above. When considering critical sections over the same lock containing ordered events, instead of ordering the critical sections entirely (as CP does by ordering $\text{rel}(\ell)$ of the earlier critical section and the $\text{acq}(\ell)$ of the latter) WCP only orders the earlier $\text{rel}(\ell)$ before the latter $\text{rel}(\ell)$.

The trace σ_5 in Figure 7.3 has a predictable race which is not detected by CP but is detected by WCP owing to weakening of Rule (b). Borrowing notation from [34], the event of type $\langle t, \text{sync}(x) \rangle$ is a shorthand for the type sequence $\langle t, \text{acq}(\ell_x) \rangle \langle t, \text{r}(x) \rangle \langle t, \text{w}(x) \rangle \langle t, \text{rel}(\ell_x) \rangle$, where ℓ_x is a unique lock associated with the variable x . For a given variable x , any two $\text{sync}(x)$ events are ordered by CP/WCP (more precisely $\prec_{\text{CP}}$ and $\prec_{\text{WCP}}$) using Rule (a) (for WCP the ordering is from the $\text{rel}(\ell_x)$ to the latter $\text{r}(x)$ but for our example this is not important). Now we describe the trace σ_5 in detail. First note that there is a predictable race between e_3 and e_{12} as witnessed by the correct reordering $\sigma_5^{\text{CR}} = e_1, e_2, e_{10}, e_{11}$. This race is not detected by CP as the conflicting events e_3 and e_{12} are CP related as follows: Firstly $e_2 \prec_{\text{CP}}^{\sigma_5} e_5$. We have $e_1 \leq_{\text{HB}}^{\sigma_5} e_2$ and $e_5 \leq_{\text{HB}}^{\sigma_5} e_6$ (thread order). Applying Rule (c)

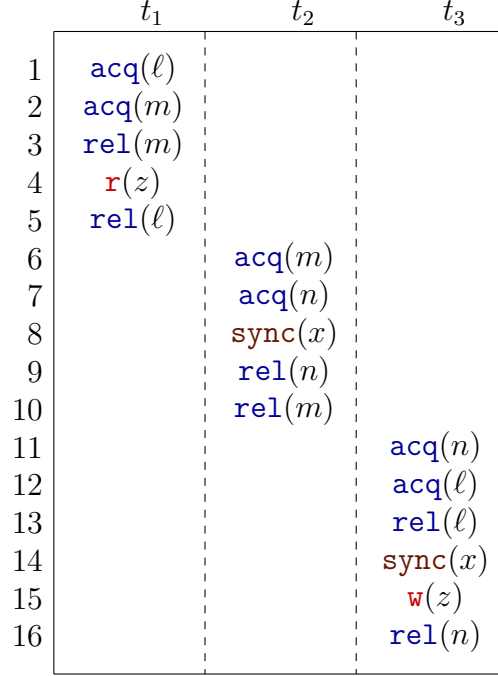


Figure 7.4: Example trace σ_6 exhibiting a predictable race detected by WCP but not by CP.

of composition with HB on $e_1 \leq_{\text{HB}}^{\sigma_5} e_2 \prec_{\text{CP}}^{\sigma_5} e_5 \leq_{\text{HB}}^{\sigma_5} e_6$ we get $e_1 \prec_{\text{CP}}^{\sigma_5} e_6$. Next we apply Rule (b) of CP on the critical sections over lock ℓ to get $e_4 \prec_{\text{CP}}^{\sigma_5} e_6$. We also have $e_8 \leq_{\text{HB}}^{\sigma_5} e_{10}$ (critical sections over same lock n). Applying Rule (c) of CP ($e_3 \leq_{\text{TO}}^{\sigma_5} e_4 \prec_{\text{CP}}^{\sigma_5} e_6 \leq_{\text{TO}}^{\sigma_5} e_8 \leq_{\text{HB}}^{\sigma_5} e_{10} \leq_{\text{TO}}^{\sigma_5} e_{12}$) we obtain $e_3 \prec_{\text{CP}}^{\sigma_5} e_{12}$. We see that this line of reasoning fails if you try to prove $e_3 \prec_{\text{WCP}}^{\sigma_5} e_{12}$ because e_4 and e_6 are not ordered by $\prec_{\text{WCP}}^{\sigma_5}$. Rule (b) of WCP would give us that the releases are ordered i.e., $e_4 \prec_{\text{WCP}}^{\sigma_5} e_9$ but this cannot be composed with HB edges to get a path to e_{12} . It turns out e_3 and e_{12} are not ordered by $\prec_{\text{WCP}}^{\sigma_5}$, and therefore the race that exists between them is correctly detected by WCP.

We present a more involved example in Figure 7.4 to once again see how a predictable race is revealed by WCP but not CP. Starting with the two sync(x) events that are CP related we get enclosing critical section on lock n to be CP related by Rule (b) i.e., $e_9 \prec_{\text{CP}}^{\sigma_6} e_{11}$. We then use $e_3 \leq_{\text{HB}}^{\sigma_6} e_6$ along with thread orderings and Rule (c) to get $e_1 \prec_{\text{CP}}^{\sigma_6} e_{12}$. Applying Rule (c) we obtain $e_5 \prec_{\text{CP}}^{\sigma_6} e_{12}$ which then gives $e_4 \prec_{\text{CP}}^{\sigma_6} e_{15}$ from rulerefcprule-c, hence CP declares that the two conflicting events e_4 and e_{15} are not in race. But they are indeed in race as revealed by the correct reorderin $\sigma_6^{\text{CR}} = e_6, e_7, e_8, e_9, e_{10}, e_{11}, e_{12}, e_{13}, e_{14}, e_1, e_2, e_3$ in which both e_4 and e_{15} are σ_6 -enabled. Unlike CP, WCP does not order e_4 and e_{15} . This is because WCP does not order e_9 and e_{11} since the weaker Rule (b) only gives $e_9 \prec_{\text{WCP}}^{\sigma_6} e_{14}$ which does not compose with the HB edge on the critical sections over lock ℓ .

In Figure 7.5 we show a trace that is only slightly different from the previous example

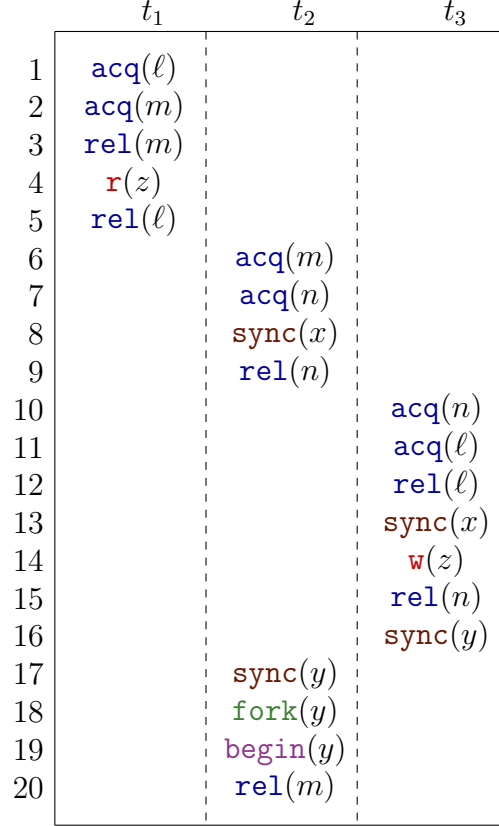


Figure 7.5: Example trace σ_7 with deadlock pattern but no predictable race.

but the subtlety involved results in the absence of a predictable race because of the presence of a deadlock pattern (namely $\langle (e_1, e_2), (e_6, e_7), (e_{10}, e_{11}) \rangle$). The correct reordering $\sigma_7^{\text{CR}} = e_1, e_6, e_{10}$ of σ_7 , in fact, witnesses the deadlock as the acquire events e_2 , e_7 and e_{11} are simultaneously σ_7 -enabled in it. Using identical reasoning as in the previous example we can derive that the two conflicting events e_4 and e_{14} are **CP** ordered and **WCP** unordered.

7.2 VECTOR-CLOCK ALGORITHM FOR WCP

In this section we describe a vector clock algorithm for detecting data races using **WCP**. As with vector clock algorithms for **HB** and **SHB**, this algorithm also computes timestamps of events and uses these timestamps to detect **WCP**-races.

7.2.1 Timestamps and Invariants

As before, the $\sqsubseteq$ ordering on timestamps will be isomorphic to $\leq_{\text{WCP}}$ and thus we will often refer to these timestamps as **WCP** timestamps/times. The formal definition of the

timestamp C_e^σ of an event in trace σ (often denoted C_e when σ is clear from context) is given below.

$$C_e = \lambda t. \begin{cases} 1 + |\{e' \leq_{\text{WCP}}^\sigma e \mid e' \neq e, \text{thr}(e') = t \text{ and } e' \in \text{Releases}_\sigma \cup \text{Forks}_\sigma\}| & \text{if } \text{thr}(e) = t \\ |\{e' \leq_{\text{WCP}}^\sigma e \mid \text{thr}(e') = t \text{ and } e' \in \text{Releases}_\sigma \cup \text{Forks}_\sigma\}| & \text{otherwise} \end{cases} \quad (7.1)$$

The above definition ensures the following.

Lemma 7.2. Let σ be a trace and let $e, e' \in \text{Events}_\sigma$ such that $\text{thr}(e) \neq \text{thr}(e')$. We have, $C_e \sqsubseteq C_{e'} \iff e \leq_{\text{WCP}}^\sigma e'$

Before we discuss the various data structures that the algorithm maintains, let us discuss some key invariants and relationships between the timestamps of different events of the trace; our algorithm will essentially leverage these to compute the timestamps of events. For an event e in trace σ , let us define some key set of events. The first set $\text{Pre}^\sigma(e)$ is the set of all events that are ordered before e according to $\leq_{\text{WCP}}^\sigma$:

$$\text{Pre}^\sigma(e) = \{e' \in \text{Events}_\sigma \mid e' \leq_{\text{WCP}}^\sigma e, e' \neq e\} \quad (7.2)$$

The following lemma states that the timestamp of e can be computed by the timestamps of events in $\text{Pre}^\sigma(e)$:

Lemma 7.3. Let σ be a trace and let $e \in \text{Events}_\sigma$ with $\text{thr}(e) = t$. Let f be the last event in σ before e with $\text{thr}(f) = t$; if no such event exists, we say $f = \text{undefined}$. Let $T_e = \bigsqcup_{e' \in \text{Pre}^\sigma(e)} C_{e'}$. Then,

$$C_e = \begin{cases} T_e & \text{if } f \neq \text{undefined}, f \notin \text{Releases}_\sigma \cup \text{Forks}_\sigma \\ T_e[t \mapsto T_e(t) + 1] & \text{otherwise} \end{cases}$$

The second set $\text{StrictPre}^\sigma(e)$ is analogous, but for $\prec_{\text{WCP}}^\sigma$:

$$\text{StrictPre}^\sigma(e) = \{e' \in \text{Events}_\sigma \mid e' \prec_{\text{WCP}}^\sigma e\} \quad (7.3)$$

Finally, $\text{HBPreEq}^\sigma(e)$ is the set of all events (including e) that are $\leq_{\text{HB}}$ -ordered before e :

$$\text{HBPreEq}^\sigma(e) = \{e' \in \text{Events}_\sigma \mid e' \leq_{\text{HB}}^\sigma e\} \quad (7.4)$$

The next lemma states how to compute $\text{Pre}()$ inductively. Here, we recall that $\text{pred}_\sigma(e)$ is the last event in σ that is $<_{\text{TO}}^\sigma$ before e , if it exists, and is **undefined** otherwise. We let $\text{Pre}^\sigma(\text{undefined}) = \emptyset$, $\text{StrictPre}^\sigma(\text{undefined}) = \emptyset$, $\text{HBPreEq}^\sigma(\text{undefined}) = \emptyset$ and assume $\{\text{undefined}\} = \emptyset$.

Lemma 7.4. Let σ be a trace and let $e \in \text{Events}_\sigma$ and let $t = \text{thr}(e)$. Let $\text{TOPred}_e = \text{Pre}^\sigma(\text{pred}_\sigma(e)) \cup \{\text{pred}_\sigma(e)\}$ and $\text{TOStrictPred}_e = \text{StrictPre}^\sigma(\text{pred}_\sigma(e))$. We have the following.

1. If $\text{op}(e) = \text{acq}(\ell)$ (for some $\ell \in \text{Locks}_\sigma$), then

$$\begin{aligned} \text{Pre}^\sigma(e) &= \text{TOPred}_e \cup \text{StrictPre}^\sigma(f) \\ \text{StrictPre}^\sigma(e) &= \text{TOStrictPred}_e \cup \text{StrictPre}^\sigma(f) \\ \text{HBPreEq}^\sigma(e) &= \text{HBPreEq}^\sigma(\text{pred}_\sigma(e)) \cup \text{HBPreEq}^\sigma(f) \cup \{e\} \end{aligned}$$

where f is the last event in σ before e such that $\text{op}(f) = \text{rel}(\ell)$ (**undefined** if there is no such event).

2. If $\text{op}(e) = \text{rel}(\ell)$ (for some $\ell \in \text{Locks}_\sigma$), then

$$\begin{aligned} \text{Pre}^\sigma(e) &= \text{TOPred}_e \cup \text{HBPreEq}^\sigma(\text{match}_\sigma(f)) \\ \text{StrictPre}^\sigma(e) &= \text{TOStrictPred}_e \cup \text{HBPreEq}^\sigma(\text{match}_\sigma(f)) \\ \text{HBPreEq}^\sigma(e) &= \text{HBPreEq}^\sigma(\text{pred}_\sigma(e)) \cup \{e\} \end{aligned}$$

where f is the last event in σ before e such that $\text{op}(f) = \text{acq}(\ell)$, $\text{thr}(f) \neq t$ and $f \leq_{\text{WCP}}^\sigma e$ ($f = \text{undefined}$ if there is no such event).

3. If $\text{op}(e) = \text{a}(x)$ (for some $x \in \text{Vars}_\sigma$, $\text{a} \in \{\text{r}, \text{w}\}$), then

$$\begin{aligned} \text{Pre}^\sigma(e) &= \text{TOPred}_e \cup \text{HBPreEq}^\sigma(f) \\ \text{StrictPre}^\sigma(e) &= \text{TOStrictPred}_e \cup \text{HBPreEq}^\sigma(f) \\ \text{HBPreEq}^\sigma(e) &= \text{HBPreEq}^\sigma(\text{pred}_\sigma(e)) \cup \{e\} \end{aligned}$$

where f is the last event in σ before e such that $\text{op}(f) = \text{rel}(\ell)$, and there as event $e' \in \text{CS}_\sigma(f)$ such that $e' \asymp e$ ($f = \text{undefined}$ if there is no such event).

4. If $\text{op}(e) = \text{fork}(u)$ (for some $u \in \text{Threads}_\sigma$), then

$$\begin{aligned} \text{Pre}^\sigma(e) &= \text{TOPred}_e \\ \text{StrictPre}^\sigma(e) &= \text{TOStrictPred}_e \\ \text{HBPreEq}^\sigma(e) &= \text{HBPreEq}^\sigma(\text{pred}_\sigma(e)) \cup \{e\} \end{aligned}$$

5. If $\text{op}(e) = \text{join}(u)$ (for some $u \in \text{Threads}_\sigma$, then

$$\begin{aligned}\text{Pre}^\sigma(e) &= \text{TOPred}_e \cup \text{HBPreEq}^\sigma(f) \\ \text{StrictPre}^\sigma(e) &= \text{TOStrictPred}_e \\ \text{HBPreEq}^\sigma(e) &= \text{HBPreEq}^\sigma(\text{pred}_\sigma(e)) \cup \{e\}\end{aligned}$$

where f is the last event of thread u (undefined if no such f exists).

Lemma 7.3 and Lemma 7.4 together give a recipe to compute the timestamp of an event using the timestamps of prior events in the trace. To formulate a vector clock algorithm, we will, in addition to the timestamp C_e , also compute the following vector times for various events in the trace:

$$\begin{aligned}P_e &= \bigsqcup_{e' \in \text{StrictPre}^\sigma(e)} C_{e'} \\ H_e &= \bigsqcup_{e' \in \text{HBPreEq}^\sigma(e)} C_{e'}\end{aligned}\tag{7.5}$$

Equipped with the different lemmas, notion of vector times and predecessor sets, we are now ready to present the algorithm, shown in Algorithm 7.1. Our algorithm works in a streaming fashion and processes the trace by looking at its events one-by-one from beginning to end. As it handles each event it updates its *state*. The state captures all the information required to assign a timestamp (a vector time) to the last event of the trace. At each step, an event is processed and the state is updated by calling the handler of appropriate type. The additional argument L in the **read** and **write** handlers specify the set of locks that are acquired by the thread performing the read or write event, at the time the event was generated. The additional arguments R and W in the **release** handler specify the set of variables that have respectively been read from and written to in the critical section of the release event being processed. All these information can be maintained on the fly.

7.2.2 Data structures maintained in the state

The algorithm maintains several vector clocks and FIFO queues in its state. We describe them next.

- **Clocks** $\mathbb{C}_t, \mathbb{P}_t, \mathbb{H}_t$. For every thread t , the clocks $\mathbb{C}_t$, $\mathbb{P}_t$ and $\mathbb{H}_t$ respectively maintain the timestamp C_{e_t} , P_{e_t} and H_{e_t} , where e_t is the last event e with $\text{thr}(e) = t$ in the trace seen so far.

Algorithm 7.1: *Vector Clock algorithm for detecting WCP-races*

```

1  function Initialization()
2    foreach  $t \in \text{Threads}$  do
3       $\mathbb{C}_t := \perp[t \mapsto 1]$ 
4       $\mathbb{P}_t := \perp;$ 
5       $\mathbb{H}_t := \perp[t \mapsto 1]$ 
6    foreach  $\ell \in \text{Locks}$  do
7       $\mathbb{H}_\ell := \perp; \mathbb{P}_\ell := \perp$ 
8      foreach  $x \in \text{Vars}$  do
9         $\mathbb{L}_{\ell,x}^{\mathbf{r}} := \perp; \mathbb{L}_{\ell,x}^{\mathbf{w}} := \perp$ 
10     foreach  $t \in \text{Threads}$  do
11        $\text{CSHist}_\ell^t := \emptyset$ 
12  function maxLB( $U, Lst$ )
13     $(C_{\max}, C'_{\max}) := (\perp, \perp)$ 
14    while not  $Lst \cdot \text{isEmpty}()$  do
15       $(C, C') := Lst \cdot \text{first}()$ 
16      if  $C \sqsubseteq U$  then
17         $(C_{\max}, C'_{\max}) := (C, C')$ 
18      else
19        break
20       $Lst \cdot \text{removeFirst}()$ 
21    return  $(C_{\max}, C'_{\max})$ 
22  handler fork( $t, u$ )
23     $\mathbb{H}_u := \mathbb{C}_t \sqcup \mathbb{H}_t$ 
24     $\mathbb{C}_u := \mathbb{C}_t \sqcup \mathbb{H}_t$ 
25     $\mathbb{P}_u := \mathbb{C}_t \sqcup \mathbb{H}_t$ 
26     $\mathbb{C}_t := \mathbb{C}_t[t \mapsto \mathbb{C}_t(t) + 1]$ 
27     $\mathbb{H}_t := \mathbb{H}_t[t \mapsto \mathbb{H}_t(t) + 1]$ 
28  handler join( $t, u$ )
29     $\mathbb{H}_t := \mathbb{H}_t \sqcup \mathbb{H}_u$ 
30     $\mathbb{C}_t := \mathbb{C}_t \sqcup \mathbb{H}_u$ 
31     $\mathbb{P}_t := \mathbb{P}_t \sqcup \mathbb{H}_u$ 
32  handler acquire( $t, \ell$ )
33     $\mathbb{H}_t := \mathbb{H}_t \sqcup \mathbb{H}_\ell$ 
34     $\mathbb{C}_t := \mathbb{C}_t \sqcup \mathbb{P}_\ell$ 
35     $\mathbb{P}_t := \mathbb{P}_t \sqcup \mathbb{P}_\ell$ 
36    foreach  $u \neq t$  do
37       $\text{CSHist}_\ell^u \cdot \text{addLast}((\mathbb{C}_t, \perp))$ 
38  handler release( $t, \ell, R, W$ )
39     $(C, C') := \text{maxLB}(\mathbb{C}_t, \text{CSHist}_\ell^t)$ 
40     $\mathbb{C}_t := \mathbb{C}_t \sqcup C'$ 
41     $\mathbb{P}_t := \mathbb{P}_t \sqcup C'$ 
42    foreach  $x \in R$  do  $\mathbb{L}_{\ell,x}^{\mathbf{r}} := \mathbb{L}_{\ell,x}^{\mathbf{r}} \sqcup \mathbb{H}_t$ 
43    foreach  $x \in W$  do  $\mathbb{L}_{\ell,x}^{\mathbf{w}} := \mathbb{L}_{\ell,x}^{\mathbf{w}} \sqcup \mathbb{H}_t$ 
44     $\mathbb{H}_\ell := \mathbb{H}_t$ 
45     $\mathbb{P}_\ell := \mathbb{P}_t$ 
46    foreach  $u \neq t$  do
47       $\text{CSHist}_\ell^u \cdot \text{last}() \cdot \text{updateRel}(\mathbb{H}_t)$ 
48     $\mathbb{C}_t := \mathbb{C}_t[t \mapsto \mathbb{C}_t(t) + 1]$ 
49     $\mathbb{H}_t := \mathbb{H}_t[t \mapsto \mathbb{H}_t(t) + 1]$ 
50  handler read( $t, x, L$ )
51     $\mathbb{C}_t := \mathbb{C}_t \sqcup_{\ell \in L} \mathbb{L}_{\ell,x}^{\mathbf{w}}$ 
52     $\mathbb{P}_t := \mathbb{P}_t \sqcup_{\ell \in L} \mathbb{L}_{\ell,x}^{\mathbf{w}}$ 
53    if  $\mathbb{W}_x \not\sqsubseteq \mathbb{C}_t$  then
54      declare ‘race’ and exit
55     $\mathbb{R}_x := \mathbb{R}_x \sqcup \mathbb{C}_t$ 
56  handler write( $t, x, L$ )
57     $\mathbb{C}_t := \mathbb{C}_t \sqcup_{\ell \in L} (\mathbb{L}_{\ell,x}^{\mathbf{r}} \sqcup \mathbb{L}_{\ell,x}^{\mathbf{w}})$ 
58     $\mathbb{P}_t := \mathbb{P}_t \sqcup_{\ell \in L} (\mathbb{L}_{\ell,x}^{\mathbf{r}} \sqcup \mathbb{L}_{\ell,x}^{\mathbf{w}})$ 
59    if  $\mathbb{R}_x \not\sqsubseteq \mathbb{C}_t$  or  $\mathbb{W}_x \not\sqsubseteq \mathbb{C}_t$  then
60      declare ‘race’ and exit
61     $\mathbb{W}_x := \mathbb{W}_x \sqcup \mathbb{C}_t$ 

```

- **Clocks $\mathbb{P}_\ell, \mathbb{H}_\ell$.** For every lock ℓ , the clocks $\mathbb{P}_\ell$ and $\mathbb{H}_\ell$ respectively maintain the timestamp P_{e_ℓ} and H_{e_ℓ} respectively, where e_ℓ is the last event e with $\text{op}(e) = \text{rel}(\ell)$ in the trace seen so far.
- **Clocks $\mathbb{L}_x^{\mathbf{r}}$ and $\mathbb{L}_x^{\mathbf{w}}$.** For every lock ℓ and variable x , the clocks $\mathbb{L}_x^{\mathbf{r}}$ and $\mathbb{L}_x^{\mathbf{w}}$ maintain

the timestamp $H_{e_{\ell, \mathbf{r}(x)}}$ and $H_{e_{\ell, \mathbf{w}(x)}}$ respectively, where $e_{\ell, \mathbf{r}(x)}$ (resp. $e_{\ell, \mathbf{w}(x)}$) is the last event e with $\text{op}(e) = \mathbf{rel}(\ell)$ such that there is an event $e' \text{CS}_\sigma(e)$ with $\text{op}(e') = \mathbf{r}(x)$ (resp. $\text{op}(e') = \mathbf{w}(x)$) in the trace seen so far.

- **FIFO queues CSHist.** For every lock ℓ and thread t , the algorithm maintains a FIFO queue CSHist_ℓ^t . The elements of these queues are pairs of timestamps (C, C') corresponding to the matching acquire and release events of lock ℓ (across all threads). The order of these entries is the same as the order of the critical sections of lock ℓ . The superscript t in CSHist_ℓ^t denotes a *reader* thread, which can remove elements from the FIFO queue when they are no longer needed. The objective of maintaining such queues is to mimic Rule (b) of WCP. For a release event e of type $\langle t, \mathbf{rel}(\ell) \rangle$ is performed, the set $\text{Pre}^\sigma(e)$ includes all the prior release events $e' \in \text{Releases}_\sigma(\ell)$ such that $\text{match}_\sigma(e') \prec_{\text{WCP}}^\sigma e$ (Rule (b) and $\leq_{\text{TO}}$ -composition Rule (c)). The FIFO queue helps maintain the correct invariant for $\mathbb{C}_t$ based on this rule.
- **Clocks $\mathbb{R}_x$ and $\mathbb{W}_x$.** These clocks, similar to the DJIT⁺ algorithm, maintain access history and are used for data race detection. The clock $\mathbb{R}_x$ stores the timestamp $\bigsqcup_{e \in \text{Reads}_\sigma(x)} C_e$. Similarly, the clock $\mathbb{W}_x$ stores the timestamp $\bigsqcup_{e \in \text{Writes}_\sigma(x)} C_e$.

Next we state the correctness of the algorithm which states the correspondence between the ordering of timestamps assigned to events (C_e for event e) and the WCP ordering.

Theorem 7.2 (Correctness of Algorithm 7.1). Algorithm 7.1 reports a race on an input trace σ iff σ has a WCP race.

Proof (Sketch). The proof follows from induction on the length of the trace, and requires us to show that each of the invariants described above hold. We omit the details here. QED.

7.2.3 Time and Space Complexity

We state the running time in Theorem 7.3. Here, we assume arithmetic operations take constant time.

Theorem 7.3. Algorithm 7.1 runs in time $O(\mathcal{N} \cdot (\mathcal{LT} + \mathcal{T}^2))$.

Proof. First note that the join operation on two vector clocks takes $O(\mathcal{T})$ time (assuming arithmetic can be done in constant time), as this amounts to taking the pointwise maximum across $\mathcal{T}$ different components of the vector clock. Each call to the **acquire** handler takes $O(\mathcal{T}^2)$ time as the loop over threads (Line 36) performs $O(\mathcal{T})$ vector clock join operations.

For the **release** handler, we analyze the total running time of all the invocation of **release** procedure. Note that the total running time of calls to $\text{maxLB} \cdot \cdot$ (Line 39) for all the **release** calls simply depends on the number of entries (pairs of acquire and release timestamps) removed from the queues, which can be bounded by the number of such entries ever added to the queues. Note that an entry corresponding to a critical section is added to $\mathcal{T} - 1$ queues (Line 36). Hence the total number of vector clock operations performed due to Line 39 is at most $O(\mathcal{N} \cdot \mathcal{T})$ (as there are at most $\mathcal{N}$ acquire/release events in the trace), and each step takes $O(\mathcal{T})$ time for vector time operations. Therefore, the total time spent in loop takes $O(\mathcal{N} \cdot \mathcal{T}^2)$ time in total across all **release** calls. A similar aggregate analysis can be done for Line 42 and Line 43 to conclude that they take $O(\mathcal{N} \cdot \mathcal{T})$ time in total. Other statements in **release** take $O(\mathcal{T})$ time in each invocation and thus $O(\mathcal{N} \cdot \mathcal{T})$ in total. The total time spent in calls to **acquire** and **release** handlers is thus $O(\mathcal{N} \cdot \mathcal{T}^2)$. Each of **read** and **write** handlers perform vector time operations that are proportional to the number of critical sections the correspond read and write events are contained in, which can be at most $\mathcal{L}$ and hence the total time spend in them amounts to $O(\mathcal{N} \cdot \mathcal{L} \cdot \mathcal{T})$. All other handlers perform $O(1)$ vector clock operations upon each invocation, and thus the contribution of these handlers is $O(\mathcal{N} \cdot \mathcal{T})$. This gives us the total running time of $O(\mathcal{N} \cdot (\mathcal{L}\mathcal{T} + \mathcal{T}^2))$. QED.

Note that for most applications the parameter $\mathcal{T}$ is usually small and parameter $\mathcal{L}$ is at most a few thousand. Typically, the bottleneck for any online race-detection technique is the length of the trace $\mathcal{N}$ which can be of the order of hundreds of millions or even billions ($10^8, 10^9$) especially for industrial scenarios. Our proposed algorithm is *linear* in the size of the trace $\mathcal{N}$ (unlike CP/RVPredict) and therefore truly scales to large traces without having to rely on any windowing strategy that would restrict the scope of the races detected. Our experimental evaluation (Section 7.3) supports this claim by showing that the algorithm scales well in practice.

We now state the space usage of our algorithm Theorem 7.4. Here, we assume arithmetic operations take constant time.

Theorem 7.4. Algorithm 7.1 uses space $O(\mathcal{T} \cdot (\mathcal{N} + \mathcal{T} + \mathcal{V} + \mathcal{L}))$.

Proof. The total number of vector clocks used is $O(\mathcal{T} + \mathcal{L} + \mathcal{V})$, each occupying $O(\mathcal{T})$ space. The total length of any of the FIFO data structures is bounded by the total number of release and acquire events. Besides, it is possible to maintain a common FIFO structure (with different input pointers) for all the FIFO queues in the set $\{\text{CSHist}_\ell^t\}_{t \in \text{Threads}}$ for each lock ℓ . Each entry uses $O(\mathcal{T})$ space (2 vector timestamps). This gives a total space usage of $O(\mathcal{T} \cdot (\mathcal{N} + \mathcal{T} + \mathcal{V} + \mathcal{L}))$. QED.

7.2.4 Lower Bounds

Our algorithm is clearly optimal in terms of running time (in terms of the length of the trace), since one has to spend linear time just looking at the entire trace. However, it can take up linear space in the worst case owing to the queues that hold the times of the previous acquires/releases. In this section, we state lower bounds result that states that any algorithm that implements WCP in linear time takes linear space. Hence our algorithm is also optimal in terms of the space requirements.

The goal is to show that one needs linear space to recognize WCP. But we want to do this when the number of threads $\mathcal{T}$ is constant, and the number of variables $\mathcal{V}$ and locks $\mathcal{L}$ is sublinear in the size of the trace ($O(\frac{\mathcal{N}}{\log \mathcal{N}})$). If the number of variables and locks is linear then it is easy to get a linear space lower bound, and even HB takes linear space in that case.

Theorem 7.5. Any algorithm that implements WCP by doing a single pass over the trace takes $\Omega(\mathcal{N})$ space.

Our lower bound proof of Theorem 7.5 relies on the membership problem of the following language.

The language $\mathcal{L}_n$. Given a natural number n , we define the equality language $\mathcal{L}_n$ as follows:

$$\mathcal{L}_n = \{u\#v \mid u, v \in \{0, 1\}^n \text{ and } u = v\} \quad (7.6)$$

That is, $\mathcal{L}_n$ is the language of two n -bit strings that are separated by $\#$ and are equal.

Lemma 7.5. Any streaming algorithm that recognizes $\mathcal{L}_n$ uses $\Omega(n)$ space.

Proof. Assume towards contradiction otherwise, i.e., there is a streaming algorithm that uses $o(n)$ space. Hence the state space of the algorithm is $o(2^n)$. Then, there exist two distinct n -bit strings $u_1 \neq u_2$, such that the algorithm is in the same state after parsing u_1 and u_2 . Hence, for any n -bit string v , the algorithm gives the same answer on inputs $u_1\#v$ and $u_2\#v$. Since the algorithm is correct, it reports that $u_1\#u_1$ belongs to $\mathcal{L}_n$. But then the algorithm reports that $u_2\#u_1$ also belongs to $\mathcal{L}_n$, a contradiction. The desired result follows. QED.

Proof Sketch for Theorem 7.5. We will essentially show how to “reduce” checking membership in $\mathcal{L}_n$ to checking WCP. The reduction uses constantly many threads, locks and variables. Let us consider the special case of $n = 3$. Suppose the input to $\mathcal{L}_3$ is $w = b_0b_1b_2c_0c_1c_2$.

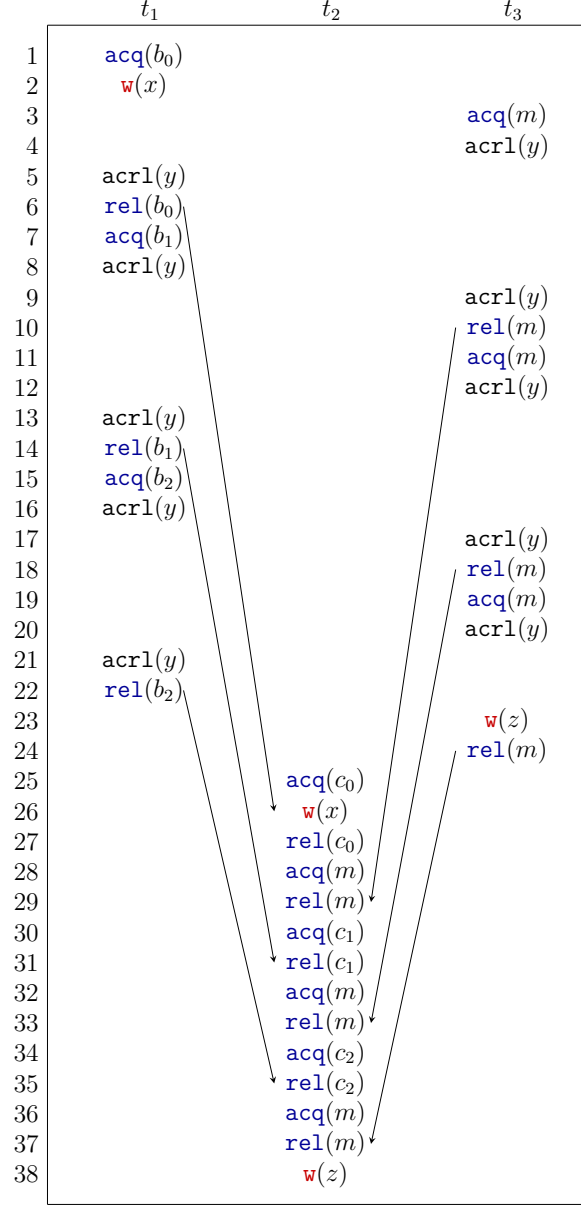


Figure 7.6: Example trace for showing linear space lower bound for WCP. $\text{acrl}(\ell)$ is shorthand for the sequence $\text{acq}(\ell)\text{rel}(\ell)$

For this input w , we will construct the trace shown in Figure 7.6, where in the trace, the lock $b_i, c_j \in \{\ell_0, \ell_1\}$, depending on what the corresponding bit in w is. The edges shown in the graph correspond to edges produced by Rule (a) or Rule (b) of WCP, but in this case the edges are contingent on the values of b_i s and c_i s. For example the edge from $\text{rel}(b_0)$ to $\text{rel}(c_0)$ and the edge between $\text{rel}(m)$ (call this edge m_0) on Lines 10 and 29 are dependent on $b_0 = c_0$. Going forward the the edge from $\text{rel}(b_1)$ to $\text{rel}(c_1)$ depends on the edge m_0 ($b_0 = c_0$) and $b_1 = c_1$. The argument continues till you reach the edge between $\text{rel}m$ between

Lines 24 and 37 which requires $b_0b_1b_2 = c_0c_1c_2$. This implies the two $\mathbf{w}(z)$ events are WCP ordered iff $b_i = c_i$ for all i . Therefore checking whether the word $w = b_0b_1b_2c_0c_1c_2$ boils down to checking whether there is a WCP-race. QED.

In Theorem 7.6, we prove a lower bound result that applies to any algorithm (not just single pass).

Theorem 7.6. For any algorithm that computes the WCP relation in time $T(\mathcal{N})$ and space $S(\mathcal{N})$ it is the case that $T(\mathcal{N})S(\mathcal{N}) \in \Omega(\mathcal{N}^2)$

Proof (Sketch). The argument presented in Theorem 7.5 can in fact be generalized to prove a combined time and space trade-off on any algorithm as follows. Note that the communication complexity of checking if two n -bit strings are equal is $\Omega(n)$. Consider the language $L_n = \{u\#^nv \mid u, v \in \{0, 1\}^n \text{ and } u = v\}$ and its membership problem. Now consider any Turing Machine M that is allowed to solve L_n by going back and forth. If M takes $T(n)$ time then we know it has to make at most $\frac{T(n)}{n}$ “rounds” of $\#^n$. If the space requirement of M is $S(n)$ then we know it can carry at most $S(n)$ bits in each rounds, implying it communicated $\frac{T(n)S(n)}{n}$ bits across the channel of $\#^n$. The total number of bits it needs to communicate in the end is n , which means $T(n)S(n) \in \Omega(n^2)$. The words $u\#^nv$ can once again be modeled in a trace as before with $\#^n$ being junk events in the trace and correspondence between membership in L_n and WCP-race detection can be shown exactly in the same way as before. QED.

Therefore, our WCP algorithm is optimal in terms of time/space trade-off as well.

7.3 EXPERIMENTAL EVALUATION

We implemented the vector clock algorithm (Algorithm 7.1) in our tool RAPID written in Java, available at [86]. Algorithm 7.1 runs in linear time. Also, since the sizes of data structures involved in the algorithm do not grow very fast, even for very large benchmarks, the memory requirement for the algorithm did not seem to be a huge bottleneck. Hence, we did not have to split our analysis into small windows, to handle large traces in RAPID. This is in stark contrast to most predictive dynamic race detection techniques [34, 44].

Table 7.1: Experimental results : Columns 1-2 describe the benchmarks (name and lines of source code respectively). Columns 3, 4 and 5 denote the number of events, threads and locks in the generated trace. Columns 6, 7 denote the number of distinct race pairs detected by RAPID by running WCP and HB analysis respectively. Columns 8 and 9 denote the number of races detected by RVPredict when the window sizes are respectively 1K and 10K and the solver timeouts are respectively 60 and 240 seconds. For programs for which WCP detects more races than HB, the corresponding entries in Column 6 are boldfaced. Column 10 represents the maximum number of races detected by RVPredict with all combinations of windows sizes (1K, 2K, 5K, 10K) and solver timeouts (60s, 120s, 240s). Column 11 reports the maximum value of the sum (over locks ℓ and threads t) of lengths of the queues CSHist_ℓ^t (Algorithm 7.1) attained at any point while performing WCP analysis on the generated trace, as a fraction of #events. Columns 12 and 13 respectively denote the time taken by RAPID for WCP and HB analysis. Columns 14 and 15 respectively denote the time taken by RVPredict for window sizes 1K and 10K, and solver timeouts 60s and 240s respectively. A ‘-’ in Column 9 and ‘TO’ in Column 15 represents timeout which we set to be 4 hours.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Program	LOC	#Events	#Thrd	#Locks	WCP	HB	#Races		Max	WCP Queue Length (%)	WCP	HB	Time	
							w=1K s=60s	w=10K s=240s					w=1K s=60s	w=10K s=240s
account	87	130	4	3	4	4	4	4	4	0.0	0.2s	0.3s	1s	1s
airline	83	128	2	0	4	4	4	4	4	0.0	0.2s	0.2s	0.8s	2s
array	36	47	3	2	0	0	0	0	0	4.3	0.2s	0.2s	1.1s	0.8s
boundedbuffer	334	333	2	2	2	2	2	2	2	0.0	0.3s	0.2s	1s	0.8s
bubblesort	274	4K	10	2	6	6	6	0	6	2.4	0.7s	0.5s	3.6s	7m3s
bufwriter	199	11.7M	6	1	2	2	2	2	2	10	47s	22.4s	4.1s	4.5s
critical	63	55	4	0	8	8	8	8	8	0.0	0.2s	0.2s	1.7s	0.9s
mergesort	298	3K	5	3	3	3	1	2	2	1.3	0.4s	0.4s	1.1s	1.4s
pingpong	124	146	4	0	7	7	7	7	7	0.0	0.5s	0.3s	1.2s	1.3s
modlyn	2.9K	164K	3	2	44	44	2	2	2	0.0	7.1s	2.4s	1.4s	17.4s
montecarlo	2.9K	7.2M	3	3	5	5	1	1	1	0.0	23.4s	16.2s	7.1s	5.7s
raytracer	2.9K	16K	3	8	3	3	2	3	3	0.0	2.4s	1s	1s	14.7s
derby	302K	1.3M	4	1112	23	23	11	-	14	0.6	16.2s	7s	31.2s	TO
eclipse	560K	87M	14	8263	66	64	5	0	8	0.4	6m51s	4m18s	26.2s	15m10s
ftpsrvr	32K	49K	11	304	36	36	10	12	12	2.2	5.7s	2.1s	3.8s	3m
jigsaw	101K	3M	13	280	14	11	6	6	6	0.0	18s	11.8s	2.8s	14.7s
lusearch	410K	216M	7	118	160	160	0	0	0	0.0	10m13s	6m48s	57.3s	46.7s
xalan	180K	122M	6	2494	18	15	7	8	8	0.1	7m22s	4m46s	43.1s	7m11s

Our experiments were conducted on an 8-core 2.6GHz 46-bit Intel Xeon(R) Linux machine, with HotSpot 1.8.0 64-Bit Server as the JVM and 50 GB heap space. We set a time limit of 4 hours for evaluating each of the techniques on each of the benchmarks. Our results are summarized in Table 7.1.

We compare the performance of RAPID only against sound techniques for race detection. For each of the techniques compared, as in [44], we attempt to analyze the following two characteristics of our algorithm:

1. *Race detection capability*, measured by the number of distinct race pairs detected. A WCP (HB) race pair is an unordered tuple of *program locations* corresponding to some pair of events in the trace that are unordered by the partial order WCP (HB). We compare the race detection capability of RAPID’s WCP vector clock algorithm with RVPredict (version 1.8.2), which, in theory, detects at least as many races as any sound dynamic race detection technique.
2. *Scalability*, measured by the time taken to analyze the entire trace. We compare our analysis time against HB vector clock algorithm for race detection (also implemented in RAPID), since HB is the simplest sound technique, and admits a fast linear time algorithm.

As stated earlier, WCP detects all the races detected using CP. Further, it is not clear if an algorithm based on CP relation can scale without windowing strategy. Therefore, we omit any comparison with CP [34].

We run all the three techniques (HB, WCP, RVPredict) on the same set of traces. This was possible because both RVPredict and RAPID can analyze a logged trace produced by RVPredict’s logging feature. We have tried to ensure that the comparison is fair; we implement the linear time vector clock algorithm for detecting HB [50], and do not restrict the HB analysis to small windows, unlike in [44] and [34].

RVPredict supports tuning of parameters like window sizes and timeout for its backend SMT solver. The tight interplay between window sizes and solver timeout in RVPredict makes it difficult to estimate the best combination of these parameters. Small windows result in a low number of reported races because every occurrence of most of the races occur across multiple windows. On the other hand, a large window implies that the logical formulae generated in RVPredict are too large to be solved for the SMT solver, within the timeout, as a result of which, most of the windows do not report any races. In Figure 7.7, we depict how the number of races vary for different values of solver timeouts and window sizes, for three benchmark examples, and as can be seen, there is no clear pattern. We run RVPredict on each of the benchmarks with several parameter combinations; we vary window

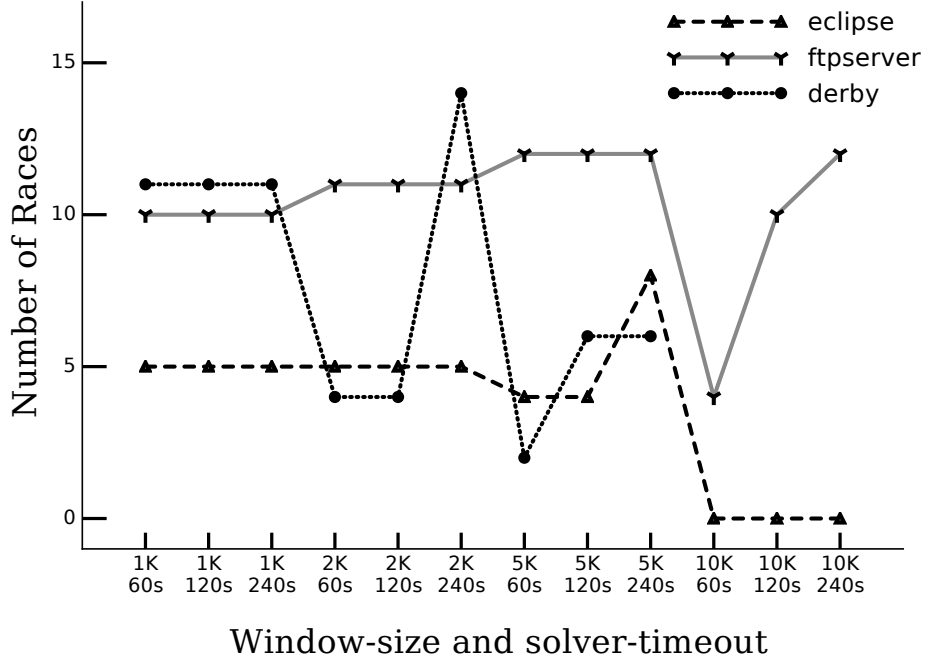


Figure 7.7: Number of races detected by RVPredict for different values of window size and solver timeout parameters

sizes as (1K, 2K, 5K, 10K) and solver timeout values as (60s, 120s, 240s). An attempt of testing RVPredict beyond these parameter values often led to very large running times or excessive memory requirements. In Table 7.1, we report the observations only for two of these combinations.

7.3.1 Benchmarks

Our evaluation benchmarks (Column 1) have primarily been derived from [44]. The benchmarks are designed for a comprehensive performance evaluation : the lines of code range from 60 to 0.5M, and the number of events vary from an order of 10 to 200M. The first set of small-sized benchmarks (account to pingpong) and are originally derived from IBM Contest benchmark suite [87]. The second set of medium sized benchmarks are derived from the Java Grande Forum benchmark suite [88]. The third set of benchmarks come from large real world software - Apache FTPServer, W3C Jigsaw web server, Apache Derby, and some applications from the DaCaPo benchmark suite (version 9.12) [89].

7.3.2 Scalability

Columns 12-15 report the times taken by WCP, HB and RVPredict. WCP analysis times are comparable to HB analysis for all the examples.

For the small set of examples, all the three techniques finish their analysis in a reasonably small time, roughly proportional to the size of the traces. However, for large examples, both WCP and HB outperform the running times of RVPredict. For ‘derby’, RVPredict exceeds the time limit of 4 hours for large window sizes. In general, it is difficult to gauge the running times of RVPredict from the length of the trace and window size, because the actual size of the internal logical formulation generated by RVPredict for every window, and the running times of its backend SMT solver crucially depend on how complex these windows are.

We highlight that the worst case linear space complexity of Algorithm 7.1 was not observed in our experiments. In Column 11, we report the maximum value of the total lengths of the FIFO queues (Section 7.2), attained at any point while performing WCP analysis on the traces, as a percentage of the number of events. As can be seen, this fraction stays below 3% for almost all examples, and is 10% for ‘bufwriter’ benchmark.

7.3.3 Bug Detection Capability

Columns 6 and 7 report the number of distinct HB and WCP race pairs. Columns 8-10 report the number of race pairs detected by RVPredict, with different parameters. The extra races discovered by WCP and not by HB (boldfaced entries in Column 6) were either found to be RVPredict races, or were manually inspected to be valid race pairs.

For the smaller benchmarks, when the number of events is relatively small, the number of race pairs detected by the three techniques is almost the same, with WCP and HB detecting the maximum number of races for each of these benchmark examples, despite the fact that RVPredict employs a theoretically more comprehensive technique than WCP.

For the larger benchmark examples, the number of races detected by RVPredict are much lower than those predicted by WCP or HB. For the ‘derby’ benchmark, RVPredict could not finish its analysis within 4 hours. In fact, for the benchmark ‘bubblesort’, RVPredict runs out of memory (50GB) for a window size of 5K. This is primarily because of the tight interplay between the window sizes and timeout parameter for the backend SMT solver. We conjecture that most of the races that are not reported by RVPredict either occur across windows, or are missed because the solver did not return with an answer within the specified time limit. On the other hand, the theoretical guarantee provided by the techniques used in RVPredict [44] would have ideally resulted in it detecting possibly more data races than both HB and WCP.

This indicates that a windowing strategy for analyzing large traces can potentially result in significant loss in the bug detection capability of any dynamic race detection technique. In fact, on careful analysis of the predicted races, we found that both HB and WCP detect races having a *distance* of millions of events — the *distance* of a race between program locations (pc_1, pc_2) is the minimum separation (in terms of the number of events in the trace) between any pair of events (e_1, e_2) exhibiting the (pc_1, pc_2) race. Specifically, both HB and WCP expose more than 25 races in ‘eclipse’ having a distance of at least 4.8 million events, with the maximum distance being 53 million. Clearly, any windowing based analysis will be incapable of catching these races.

In all the examples, the *set* of races detected by HB are a subset of the *set* of races detected by WCP. This is expected as WCP is a weakening of CP (and hence of HB). For the large benchmarks ‘eclipse’, ‘jigsaw’ and ‘xalan’, the number of races detected by WCP are more than those detected by HB. In all examples, the *set* of races detected by RVPredict are a subset of the races reported by WCP and HB, except in ‘mergesort’ and ‘ftpservers’, where RVPredict reports one extra race each.

Note that, our WCP based race detection algorithm does not report drastically more races than the simpler HB based algorithm. While this is surprising, given the optimistic comparison of CP and RVPredict versus HB, as reported in [34, 44], the apparent disparity can be explained by the fact that both [34] and [44] compare their techniques against a windowed implementation of HB based vector clock algorithm, which can potentially miss HB races between pairs of events (unordered by HB) that occur far apart in the trace, and thus, possibly missing out on program location pairs corresponding to these event pairs. As stated before, our implementation of HB vector clock algorithm is not *windowed* and catches these far-away event pairs unordered by HB. That being said, the few extra races missed by (our implementation of) HB, but predicted by WCP and/or RVPredict, are quite subtle.

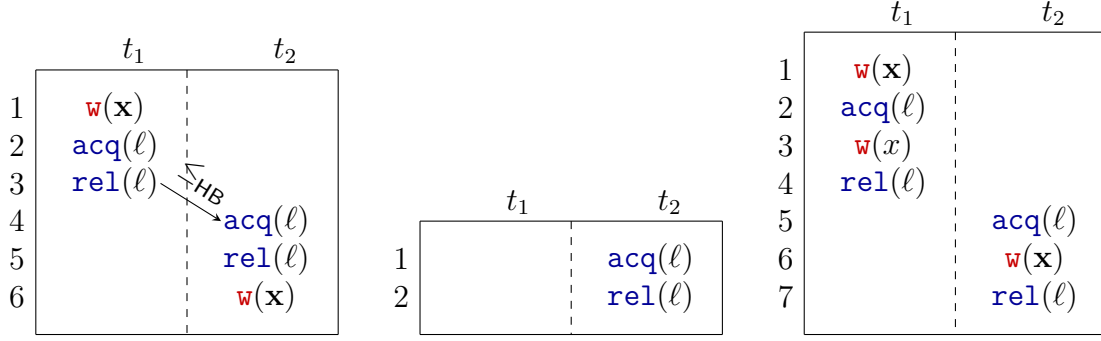
CHAPTER 8: PREDICTING SYNCHRONIZATION PRESERVING DATA RACES IN LINEAR TIME

In the previous chapters, we observed that **HB** misses many simple predictable data races because it conservatively orders critical sections on the same lock. The partial order **WCP** (Chapter 7) attempts to overcome this by weakening the **HB** partial order so that not all critical sections may be ordered. **WCP** detects provable more data races than both **HB** and its precursor **CP** [34]. Nevertheless, it misses races due to two main reasons. First, it orders critical sections when they contain conflicting events, even though they may be flipped in a correct reordering. The second (and also major) source of conservativeness arises from the composition of $\prec_{\text{WCP}}$ with $\leq_{\text{HB}}$. Finally, as with **HB**, race detection based on **WCP** is (weakly) sound only until the first **WCP** race, and thus all **WCP**-races may not be predictable data races. We illustrate these sources of conservativeness later in this chapter.

In this chapter, we take a different approach to resolving the conservativeness of **HB**. Instead of trying to weaken the $\leq_{\text{HB}}$ partial order (as in $\leq_{\text{WCP}}$), we instead resort to a more first principle approach by first characterizing a well defined class of data races and then formulating an algorithm to do so. As a starting point, we resort to the **HB** partial order, extract some foundational principles behind its scalability and generalize them to predict more data races. When reasoning about correct reorderings, happens-before analysis does not consider any execution in which the order of synchronization primitives is reversed from that in the observed execution (Chapter 6). We call such alternate executions *sync(hronization)-preserving executions*. Other, more powerful race prediction techniques [34–37, 44, 92] sacrifice this principle and consider alternate executions that are not sync-preserving. Naturally, this typically results in performance degradation, as the problem is in general **NP-hard** [17].

Data races exposed by **HB** are witnessed by correct reordering that preserve the ordering of synchronization primitives. Nevertheless, **HB** can still miss simple races that in fact adhere to this pattern. Let us consider the trace σ_1 shown in Figure 8.1a. Here, the partial order happens-before orders the first $\text{w}(x)$ (event e_1) and the last $\text{w}(x)$ (event e_6), and therefore, does not detect any race in this execution. However events e_1 and e_6 are in race. This can be exposed by the correct reordering shown in Figure 8.1b, which is obtained by *dropping* the critical section of lock ℓ performed by thread t_1 . Notice that the order of synchronization events (namely, $\text{acq}(\ell)$ and $\text{rel}(\ell)$ events) *that appear in the trace* of Figure 8.1b, are in the same order as in the trace of Figure 8.1a, and hence this is a sync-preserving execution. Thus, the notion of sync-preservation captures races beyond standard happens-before races.

Another important limitation of happens-before and virtually all partial-order methods [13, 14, 34, 36, 93] is highlighted in Figure 8.1c. The trace σ_2 has a race between e_1



(a) An observed trace σ_1 (b) Correct reordering σ_1^* witnessing a race in σ_1 . (c) An observed trace σ_2 .

Figure 8.1: (Figure 8.1a) shows a trace σ_1 with a sync-preserving race (e_1, e_6) missed by $\leq_{HB}$. (Figure 8.1b) shows a witness that exposes the race. (Figure 8.1c) shows a sync-preserving race (e_1, e_6) that is non-consecutive, due to the intermediate event e_3 .

and e_6 , both conflicting on variable x . Notice, however, that the intermediate event e_3 also accesses x , but is not in race with either e_1 or e_6 . Partial-order methods for race prediction are limited to capturing races *only between successive* conflicting accesses¹. Hence, distant races that are interjected with intermediate conflicting but non-racy events, are missed by such methods. On the other hand, sync-preservation is not bound to such limitations: (e_1, e_6) is characterized as a race under this criterion, regardless of the intermediate, non-racy e_3 , and is exposed by a witness that omits the critical section on lock ℓ in the thread t_1 .

Motivated by the above observations, we describe the following contributions in this chapter (also published previously [18]).

1. We introduce the novel notion of *sync(hronization)-preserving data races*. This is a *sound* notion of predictable races, and it *strictly subsumes* the standard notion of happens-before races. Moreover, it characterizes races between events that can be arbitrarily far apart in the input trace, as opposed to happens-before and other partial-order methods that only characterize races between *successive* conflicting accesses.
2. We develop an efficient, single-pass, nearly linear time algorithm SyncP that, given a trace σ , detects whether σ contains a sync-preserving race. In fact, our algorithm soundly reports *all* events e_2 which are in a sync-preserving race with an event e_1 that appears earlier in σ . This is contrast with WCP, which is not sound beyond the first race reported. Given $\mathcal{N}$ events in σ , our algorithm spends $\tilde{O}(\mathcal{N})$ time, where $\tilde{O}$ hides

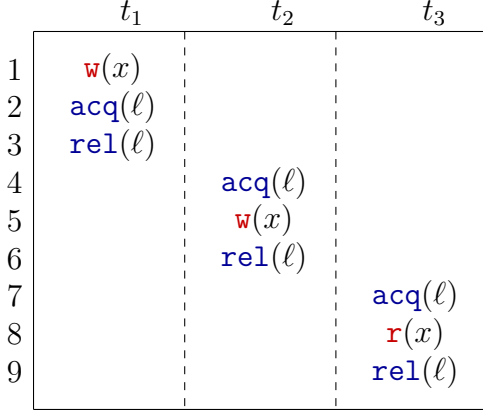
¹When the earlier access is a read instead of a write, this statement is true *per thread*.

factors poly-logarithmic in $\mathcal{N}$, when other parameters of the input (e.g., number of threads) are $\tilde{O}(1)$.

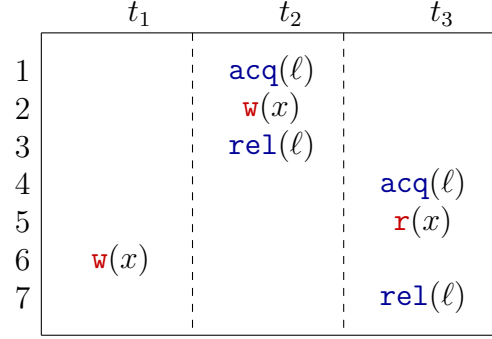
3. Although our algorithm performs a single pass of the trace, in the worst case, it might use space that is nearly linear in the length of the trace, i.e., $\tilde{O}(\mathcal{N})$ space. Hence follows a natural question: is there an efficient algorithm for sync-preserving race prediction that uses considerably less space? We answer this question in negative, by showing that *any* single-pass algorithm for detecting even a single sync-preserving race must use nearly linear space. Hence, our algorithm **SyncP** has nearly *optimal* performance in both time and space.
4. We next study the complexity of race prediction with respect to the number of synchronization reversals that might occur when constructing a witness that exposes the race. In the case of synchronization via locks, this number corresponds to the number of critical sections whose order is reversed in the witness trace. We prove that the problem of predicting races which can be witnessed by *a single* reversal (of two critical sections) is **NP**-complete and even **W[1]**-hard when parameterized by the number of threads. Thus, sync-preservation characterizes *exactly* the tractability boundary of race prediction, and our algorithm is nearly *optimal* for the tractable side. Moreover, our result shows that *any level* of synchronization suffices to make the problem of race prediction as hard as in the general case.
5. Finally, we have implemented our race prediction algorithm **SyncP** and evaluated its performance on standard benchmarks. Our results show that sync-preservation characterizes many races that are missed by state-of-the-art methods, and **SyncP** detects them efficiently.

8.1 SYNCHRONIZATION-PRESERVING DATA RACES

In general, the problem of data race prediction is intractable [17], and a sound and complete algorithm for data race prediction is unlikely to scale beyond programs of even moderate size. A recent trend in predictive analysis for race detection instead, aims to develop techniques that are sound but incomplete, with successively better prediction power (ability to report more data races) than previous techniques [13, 34–37]. Most of these techniques are either based on partial orders [13, 34, 48] or use graph-based algorithms [35, 36]. Here, we characterize a class of predictable data races, called *sync(hronization)-preserving* races, which we define shortly. We will later (Section 8.2) present an algorithm that reports a race



(a) Trace σ_6



(b) Sync-preserving correct reordering σ_6^{CR} of σ_6

Figure 8.2: Sync-preserving correct reordering and sync-preserving races

iff the input trace has a sync-preserving race. Since sync-preserving races are predictable races, our algorithm will be sound for race prediction.

Sync-preserving correct reordering. A correct reordering of a trace is said to be *sync(hronization)-preserving* if it does not reverse the order of synchronization constructs; in our formalism, traces use locks as synchronization primitives to enforce mutual exclusion. Formally, a correct reordering ρ of a given trace σ is *sync-preserving* with respect to σ if for every lock ℓ and for any two acquire events $e_1, e_2 \in \text{Acquires}_\rho(\ell)$, we have $e_1 \leq_{tr}^\rho e_2$ iff $e_1 \leq_{tr}^\sigma e_2$. In other words, the order of two critical sections on the same lock is the same in σ and ρ .

Let us illustrate this notion on an example.

Example 8.1. Consider trace σ_6 in Figure 8.2a. This trace has 3 critical sections on lock ℓ . Now consider the correct reordering σ_6^{CR} (Figure 8.2b) of σ_6 . Here, the critical section in thread t_1 is not present. But, nevertheless, the order amongst the remaining critical sections on ℓ (in threads t_2 and t_3) is the same as in σ_6 , making σ_6^{CR} a sync-preserving correct reordering of σ_6 . This example also demonstrates that the order of read and write events may be different in a trace and its sync-preserving correct reordering (as in Figure 8.2).

A pair of conflicting events (e_1, e_2) of a trace σ is said to be a *sync(hronization)-preserving race* of σ if there is a sync-preserving correct reordering ρ of σ in which e_1 and e_2 are σ -enabled.

Example 8.2. Let us again consider traces from Figure 8.2. Events e_1 and e_8 in σ_6 (Figure 8.2a) correspond respectively to events e_6 and e_7 in σ_6^{CR} (Figure 8.2b). These two events are σ_6 -enabled in the prefix $\rho = e_1 \cdot e_2 \cdot e_3 \cdot e_4 \cdot e_5$ of σ_6^{CR} . As a result, (e_1, e_8) is a sync-preserving

race of σ_6 . Likewise, (e_1, e_4) is also a sync-preserving race of σ_6 witnessed by the singleton sync-preserving correct reordering $\rho' = \langle t_2, \text{acq}(\ell) \rangle$, in which both e_1 and e_4 are enabled.

In this chapter, we will present a linear time algorithm for the following decision problem.

Problem 8.1 (Sync-Preserving Race Prediction). Given trace σ , determine if there is a pair of conflicting events (e_1, e_2) in σ such that (e_1, e_2) is a sync-preserving data race of σ .

Comparison with other approaches. Here we briefly compare sync-preserving races with other approaches in the literature for sound dynamic race prediction. Races reported using the famous *happens-before* HB partial order [48], and its extension to *schedulable-happens-before* SHB [14] are strictly subsumed by this notion. That is, these techniques only compute sync-preserving races, but can also miss simple cases of sync-preservation, as already illustrated in the examples of Figure 8.1. The *causally precedes* (CP) partial order [34], and its extension to the *weak causally precedes* (WCP) partial order [13] are capable of predicting races that reverse critical sections. However, they are closed under composition with HB, and as such can miss even simple sync-preserving races, even on two-threaded traces. We refer to [94] for comparison with other approaches like the DC [36] and SDP [37] partial orders.

8.1.1 Summary of Algorithms and Complexity

Before we proceed to technical details, let us first prove the main results. In later sections we present the details, i.e., algorithms, proofs and examples. Due to limited space, some technical proofs are relegated to [94]. Our first result is an algorithm for dynamic prediction of sync-preserving races. We show the following theorem.

Theorem 8.1. Sync-preserving race prediction is solvable in $O(\mathcal{N} \cdot \mathcal{T}^2 + \mathcal{A} \cdot \mathcal{V} \cdot \mathcal{T}^3)$ time and $O(\mathcal{N} + \mathcal{T}^3 \cdot \mathcal{V} \cdot \mathcal{L})$ space, for a trace σ with length $\mathcal{N}$, $\mathcal{T}$ threads, $\mathcal{A}$ acquires, and $\mathcal{V}$ variables.

In many settings the number of events $\mathcal{N}$ and number of acquires $\mathcal{A}$ are the dominating parameters, whereas the other parameters are much smaller, i.e., $\mathcal{T}, \mathcal{V} = \tilde{O}(1)$, where $\tilde{O}$ hides poly-logarithmic factors. Hence, the complexity of our algorithm becomes $\tilde{O}(\mathcal{N})$ for both time and space. Our next result shows that a linear space complexity is essentially unavoidable when predicting sync-preserving races with one-pass streaming algorithms.

Theorem 8.2. Any one-pass algorithm for sync-preserving race prediction on traces with ≥ 2 threads, $\mathcal{N}$ events and $\Omega(\log \mathcal{N})$ locks uses $\Omega(\mathcal{N} / \log^2 \mathcal{N})$ space.

Clearly, any algorithm must spend linear time, while Theorem 8.2 shows that the algorithm must also use (nearly) linear space. As our algorithm uses $\tilde{O}(\mathcal{N})$ time and space, it is optimal for both resources, modulo poly-logarithmic improvements. Our next theorem shows a combined time-space lower bound for the problem, which highlights that reducing the space usage must lead to an increased running time, given that the algorithm is executed on the Turing Machine model.

Theorem 8.3. Consider the problem of sync-preserving race prediction on traces with ≥ 2 threads, $\mathcal{N}$ events and $\Omega(\log \mathcal{N})$ locks. Consider any Turing Machine algorithm for the problem with time and space complexity $T(\mathcal{N})$ and $S(\mathcal{N})$, respectively. Then we have $T(\mathcal{N}) \cdot S(\mathcal{N}) = \Omega(\mathcal{N}^2 / \log^2 \mathcal{N})$.

Finally, we study the complexity of general race prediction as a function of the number of reversals of synchronization operations. Given our positive result in Theorem 8.1, can we relax our restriction of sync-preservation while retaining a tractable definition of predictable races? Our next theorem answers this question in negative.

Theorem 8.4. Dynamic race prediction on traces with a single lock and two critical sections is W[1]-hard parameterized by the number of threads.

Note that W[1]-hardness implies NP-hardness. The proof of the above statement is presented in [94] and we omit it from this presentation, as a similar proof is presented in Chapter 9.

Together, Theorem 8.1, Theorem 8.2 and Theorem 8.4 characterize *exactly* the tractability boundary of race prediction, and show that our algorithm is *time and space optimal* for the tractable side.

8.2 DETECTING SYNCHRONIZATION-PRESERVING RACES

In this section, we discuss our algorithm **SyncP** for detecting sync-preserving data races. The complete algorithm is presented in Section 8.2.4. The algorithm may appear complex at first glance and to make the exposition simple, we first present a high-level overview of the algorithm in Section 8.2.1. In the overview, we highlight important observations and algorithmic insights for solving smaller subproblems of the main problem of sync-preserving race prediction. Section 8.2.2 and Section 8.2.3 present details of the algorithms for the smaller subproblems, and pave the way for the final algorithm in Section 8.2.4. In Section 8.2.5, we present a matching space lowerbound result for detecting sync-preserving races, thereby showing the optimality of our algorithm.

8.2.1 Insights and Overview of the Algorithm

Our algorithm, **SyncP**, relies on several important observations that are crucial for detecting sync-preserving races in linear time. In order to present these observations, it is helpful to define intermediate subproblems.

Problem 8.2 (Sync-Preserving Race Prediction Given Pair). Given a trace σ and a pair of conflicting events (e_1, e_2) of σ , determine if (e_1, e_2) is a sync-preserving data race of σ .

Problem 8.3 (Sync-Preserving Race Prediction Given Event and Thread). Given a trace σ , an event e in σ and a thread $t \neq \text{thr}(e)$, check if there is an event $e' \leq_{\text{tr}}^{\sigma} e$ such that $\text{thr}(e') = t$ and (e', e) constitutes a sync-preserving race of σ .

Observe that a trace with $\mathcal{N}$ events can have $O(\mathcal{N}^2)$ conflicting pairs of events. Thus, an algorithm for Problem 8.2 that runs in time $O(T(\mathcal{N}))$ can be used to obtain an algorithm for Problem 8.3 (resp. Problem 8.1) that runs in time $O(\mathcal{N} \cdot T(\mathcal{N}))$ (resp. $O(\mathcal{N}^2 \cdot T(\mathcal{N}))$) by checking if every other event of the given thread t that conflicts with e is also in race with e (resp. every conflicting pair of events is a race). We will, however, present algorithms for all three problems that run in $O(\mathcal{N})$ time.

Efficiently solving Problem 8.2. Our first important observation towards a full fledged solution to Problem 8.2 is that when checking for the existence of a sync-preserving reordering (of a trace σ) that witnesses a race on a given pair (e_1, e_2) , it, suffices to only search for those reorderings ρ of σ which impose the same order as in σ , on *all of its events*, and not just on the critical sections. We formalize this in Lemma 8.1.

Lemma 8.1. If (e_1, e_2) is a sync-preserving race of σ , then there is a correct reordering ρ of σ such that both e_1, e_2 are σ -enabled in ρ and $\leq_{\text{tr}}^{\rho} \subseteq \leq_{\text{tr}}^{\sigma}$.

Proof. Let π be a sync-preserving correct reordering of σ . Let $E = \text{Events}_{\pi}$. Observe that E is $(\leq_{\text{TO}}^{\sigma}, \text{lastWrite}_{\sigma})$ -closed because π is a correct reordering of σ . We note a few observations about E . First, $E \subseteq \text{Events}_{\sigma}$. Second, E is downward closed with respect to $\leq_{\text{TO}}^{\sigma}$. Third, for every read event $e \in E$, we have $\text{lastWrite}_{\sigma}(e) \in E$. Fourth, for every lock ℓ , there is at most one acquire a of ℓ such that $a \in E$ but $\text{match}_{\sigma}(a) \notin E$.

Now, consider the sequence ρ obtained by linearizing E according to the total order of $\leq_{\text{tr}}^{\sigma}$ (i.e., ρ is the projection of σ onto the set E). We first argue that ρ is a well-formed trace. This follows because for every lock ℓ , there is at most one unmatched acquire event e in E of lock ℓ , and all the other acquires are earlier than e in σ (and hence in ρ). Next, ρ respects $\leq_{\text{TO}}^{\sigma}$ because E is downward closed with respect to $\leq_{\text{TO}}^{\sigma}$ and respects $\leq_{\text{tr}}^{\sigma}$. For the same reason, for every read event $e \in E$, we have $\text{lastWrite}_{\rho}(e) = \text{lastWrite}_{\sigma}(e)$. Thus,

ρ is a correct reordering of σ . Further, ρ is trivially a sync-preserving correct reordering of σ . QED.

The implication of Lemma 8.1 is the following. When we are searching for a correct reordering ρ of σ that witnesses a race (e_1, e_2) , and if we already have access to the set of events $S \subseteq \text{Events}_\sigma$ of a candidate reordering ρ , a simple check suffices — does S form a correct reordering of σ when linearized according to $\leq_{\text{tr}}^\sigma$? In other words, we do not need to enumerate and check all the (exponentially many) permutations of events in S . Thus, Problem 8.2 — ‘search for a sync-preserving correct reordering ρ ’ — reduces to a simpler problem — ‘search for an appropriate set of events’. Of course, not all sets $S \subseteq \text{Events}_\sigma$ of events can be linearized (according to $\leq_{\text{tr}}^\sigma$) to obtain a correct reordering of σ . At the very least, S should satisfy some sanity conditions which we outline next.

Definition 8.1 (Thread-Order and Last-Write Closure). Let σ be a trace. A set $S \subseteq \text{Events}_\sigma$ is said to be $(\leq_{\text{TO}}^\sigma, \text{lastWrite}_\sigma)$ -closed if (a) S is downward-closed with respect to $\leq_{\text{TO}}^\sigma$, and (b) for any read event $r \in \text{Events}_\sigma$, if $r \in S$ and if $\text{lastWrite}_\sigma(r)$ exists, then $\text{lastWrite}_\sigma(r) \in S$. The $(\leq_{\text{TO}}^\sigma, \text{lastWrite}_\sigma)$ -closure of a set $S \subseteq \text{Events}_\sigma$, denoted $\text{TLClosure}_\sigma(S)$ is the smallest set $S' \subseteq \text{Events}_\sigma$ such that $S \subseteq S'$ and S' is $(\leq_{\text{TO}}^\sigma, \text{lastWrite}_\sigma)$ -closed.

We remark that any correct reordering ρ of σ that contains events in the set S must also contain all the events in $\text{TLClosure}_\sigma(S)$.

Definition 8.2 (Sync-Preserving Closure). Let σ be a trace. A set $S \subseteq \text{Events}_\sigma$ is said to be sync-preserving closed if

- (a) S is $(\leq_{\text{TO}}^\sigma, \text{lastWrite}_\sigma)$ -closed, and
- (b) for any two acquire events $a_1, a_2 \in \text{Acquires}_\sigma(\ell)$ with $a_1 \leq_{\text{tr}}^\sigma a_2$, if both $a_1, a_2 \in S$, then $\text{match}_\sigma(a_1) \in S$

The sync-preserving closure of a set $S \subseteq \text{Events}_\sigma$, denoted $\text{SPClosure}_\sigma(S)$ is the smallest set $S' \subseteq \text{Events}_\sigma$ such that $S \subseteq S'$ and S' is sync-preserving closed.

Intuitively, the set $S' = \text{SPClosure}_\sigma(S)$ captures the additional set of events that must be present in any sync-preserving correct reordering ρ of σ given that ρ contains all events in S . First, any correct reordering of σ containing S will contain $\text{TLClosure}_\sigma(S)$ and thus $\text{TLClosure}_\sigma(S) \subseteq S'$ (Condition (a)). Second, if a correct reordering ρ is sync-preserving and contains two acquires $a_1 \leq_{\text{tr}}^\sigma a_2$ on the same lock ℓ , then we must also have $a_1 \leq_{\text{tr}}^\rho a_2$. Then, in order to ensure well-formedness of ρ , $\text{CS}_\sigma(a_1)$ must also finish entirely before a_2 in ρ , and thus ρ must contain $\text{match}_\sigma(a_1)$ (Condition (b)).

For two events $e_1, e_2 \in \text{Events}_\sigma$, we define the following.

$$\text{SPIdeal}_\sigma(e_1, e_2) = \text{SPClosure}_\sigma(\{\text{pred}_\sigma(e_1)\} \cup \{\text{pred}_\sigma(e_2)\}) \quad (8.1)$$

Here, we let $\{\text{pred}_\sigma(e)\} = \emptyset$ if $\text{pred}_\sigma(e) = \text{undefined}$). In essence, $\text{SPIdeal}_\sigma(e_1, e_2)$ contains the necessary set of events that must be present in any sync-preserving correct reordering that witnesses the race (e_1, e_2) . We next show that, in fact, it is also a sufficient set of events, given that it is disjoint from $\{e_1, e_2\}$.

Lemma 8.2. (e_1, e_2) is a sync-preserving race of σ iff $\{e_1, e_2\} \cap \text{SPIdeal}_\sigma(e_1, e_2) = \emptyset$.

Proof. Without loss of generality, we let $e_1 \leq_{\text{tr}}^\sigma e_2$.

($\Rightarrow$) Assume $\text{SPIdeal}_\sigma(e_1, e_2) \cap \{e_1, e_2\} \neq \emptyset$. Then we must have $e_1 \in \text{SPIdeal}_\sigma(e_1, e_2)$ and in particular $e_1 \in \text{SPClosure}_\sigma(\text{pred}_\sigma(e_2))$ (and thus $e = \text{pred}_\sigma(e_2) \neq \perp$). Then, either $e_1 \in \text{TLClosure}_\sigma(\{e\})$ or there is a release event e' such that $e_1 \leq_{\text{TO}}^\sigma e'$ and $e' \in \text{SPClosure}_\sigma(e)$. In either case, e_1 cannot be enabled in a sync-preserving correct reordering containing e .

($\Leftarrow$) Let $E = \text{SPIdeal}_\sigma(e_1, e_2)$ and let ρ be the sequence obtained by linearizing E as per $\leq_{\text{tr}}^\sigma$. Observe that ρ is well formed, respects $\leq_{\text{TO}}^\sigma$ and lastWrite_σ and thus is a correct reordering of σ . Further, the order of all critical sections is the same. Also, e_1 and e_2 are both enabled in ρ . QED.

Lemma 8.2 gives us a straightforward algorithm for Problem 8.2 — first compute $I = \text{SPIdeal}_\sigma(e_1, e_2)$ and check if neither $e_1, e_2 \notin I$. In Section 8.2.2 we show how to perform this computation in linear time.

Efficiently Solving Problem 8.3. As noted before, a linear time algorithm for Problem 8.2 guarantees a *quadratic* time algorithm for Problem 8.3. In order to design a more efficient *linear time* algorithm, we will exploit *monotonicity* of $\text{SPIdeal}_\sigma(\cdot, \cdot)$, which we formalize next.

Lemma 8.3. Let σ be a trace and let $e_1, e_2, e'_1, e'_2 \in \text{Events}_\sigma$ such that $e_1 \leq_{\text{TO}}^\sigma e'_1$ and $e_2 \leq_{\text{TO}}^\sigma e'_2$. Then, $\text{SPIdeal}_\sigma(e_1, e_2) \subseteq \text{SPIdeal}_\sigma(e'_1, e'_2)$.

Proof. The proof follows from the following observations. $e_1 \leq_{\text{TO}}^\sigma \text{pred}_\sigma(e'_1)$, $e_2 \leq_{\text{TO}}^\sigma \text{pred}_\sigma(e'_2)$ and $\text{SPClosure}_\sigma(S)$ is downward closed with respect to $(\leq_{\text{TO}}^\sigma)$ and sync-preserving-closed. QED.

Our linear time algorithm for Problem 8.3 exploits Lemma 8.3 as follows. Suppose we are checking if a given event e in the given trace σ is in sync-preserving race with some earlier conflicting event of thread t . To accomplish this, we can scan σ and enumerate the list L of events that belong to t and conflict with e . When checking for a race with the first such

event e'_{first} , we compute $I_{\text{first}} = \text{SPIdeal}_\sigma(e'_{\text{first}}, e)$. If a race is found, we are done. Otherwise, we analyze the next event e'_{next} in L and compute $I_{\text{next}} = \text{SPIdeal}_\sigma(e'_{\text{next}}, e)$. Here Lemma 8.3 ensures that $I_{\text{first}} \subseteq I_{\text{next}}$. Our algorithm exploits this observation by computing the latter set I_{next} incrementally, spending time that is proportional only to the number of *extra* events (i.e., events in $I_{\text{next}} \setminus I_{\text{first}}$). This principle is applied repeatedly to subsequent events of L , giving us an overall linear time algorithm (Section 8.2.3).

Efficiently solving Problem 8.1. A final ingredient in our incremental linear time algorithm for Problem 8.1 is the following observation which is a corollary of Lemma 8.3.

Corollary 8.1. Let σ be a trace and let $e_1, e_2, e'_2 \in \text{Events}_\sigma$ such that $e_1 \leq_{\text{tr}}^\sigma e_2 \leq_{\text{TO}}^\sigma e'_2$, $e_1 \asymp e_2$ and $e_1 \asymp e'_2$. If (e_1, e_2) is not a sync-preserving race, then (e_1, e'_2) is also not a sync-preserving race of σ .

Intuitively, this observation suggests the following. Suppose that, when looking for a sync-preserving race, the algorithm determines that e_2 is not in race with any earlier conflicting event. Then, for an event e'_2 (that appears later in the thread of e_2), we only need to investigate if e'_2 is in race with conflicting events e'_1 that appear *after* e_2 in the trace (i.e., $e_2 \leq_{\text{tr}}^\sigma e'_1 \leq_{\text{tr}}^\sigma e_2$), instead of additionally looking for races (e'_1, e'_2) where $e'_1 \leq_{\text{tr}} e'_2$.

High-level overview of SyncP. Equipped with Lemma 8.3 and Corollary 8.1, we now describe our incremental algorithm for Problem 8.1 that works in linear time. For ease of exposition, let us focus on the question — is there a write-write race on some fixed variable $x \in \text{Vars}$ when accessed in two fixed threads $t_1, t_2 \in \text{Vars}$. The algorithm scans the trace in a streaming forward pass and analyzes every event $e = \langle t_2, \mathbf{w}(x) \rangle$, checking if there is an earlier conflicting event $e' = \langle t_1, \mathbf{w}(x) \rangle \leq_{\text{tr}} e$ so that (e', e) is a sync-preserving race. In doing so, it computes $I = \text{SPIdeal}_\sigma(e', e)$ in linear time and checks if $e' \notin I$. If not, (e', e) is not a race and the algorithm checks if there is a different event $e'_{\text{next}} \leq_{\text{tr}} e$ so that (e'_{next}, e) is a race. This continues until there are no earlier events remaining that conflict with e . Each time, the ideal computation is performed incrementally, by using the previously computed ideals. After this, the algorithm moves to the next write event $e_{\text{next}} = \langle t_2, \mathbf{w}(x) \rangle$ in t_2 and checks if it is in race with some earlier event e'' of thread t_1 , where this time, e'' appears in the trace after the previously discarded event e of t_2 . Again, the closed set $\text{SPIdeal}_\sigma(e'', e_{\text{next}})$ is computed incrementally. We show that all this incremental computation can be performed efficiently and present an outline for our algorithm for Problem 8.1 in Section 8.2.4.

Next, we present high-level descriptions of the intermediate steps that we outlined above, and discuss important algorithmic insights and data-structures that help achieve efficiency.

Algorithm 8.1: *Checking if a given conflicting pair constitutes a sync-preserving race*

Input: Trace σ , Conflicting events e_1 and e_2 with $e_1 \leq_{tr}^\sigma e_2$

```

1 function ComputeSPClosure( $\sigma, e_1, e_2, I_0$ )
2    $I \leftarrow I_0 \cup \text{TLClosure}_\sigma(\{\text{pred}_\sigma(e_1)\}) \cup \text{TLClosure}_\sigma(\{\text{pred}_\sigma(e_2)\})$ 
3   repeat
4     if  $\exists \ell \in \text{Locks}_\sigma, \exists a_1, a_2 \in \text{Acquires}_\sigma(\ell)$  such that  $a_1 \leq_{tr}^\sigma a_2$  and  $a_1, a_2 \in I$  then
5        $I \leftarrow I \cup \text{TLClosure}_\sigma(\{\text{match}_\sigma(a_1)\})$ 
6   until  $I$  does not change
7   return  $I$ 
8  $I \leftarrow \text{ComputeSPClosure}(\sigma, e_1, e_2, \emptyset)$ 
9 if  $e_1 \notin I$  then
10  declare ‘race’

```

8.2.2 Checking if a given pair of conflicting events is a sync-preserving race

Algorithm 8.1 outlines our solution to Problem 8.2 (check if a given pair of events (e_1, e_2) is a sync-preserving race of σ). This algorithm computes the closure $\text{SPIdeal}_\sigma(e_1, e_2)$ in an iterative fashion and checks if it contains neither e_1 nor e_2 (see Lemma 8.2). We remark that when $e_1 \leq_{tr}^\sigma e_2$, Definition 8.2 ensures that $e_2 \notin \text{SPIdeal}_\sigma(e_1, e_2)$. Consequently, the check ‘ $e_1 \notin \text{SPIdeal}_\sigma(e_1, e_2)$ ’ (Line 9 in Algorithm 8.1) is equivalent to the condition ‘ $\{e_1, e_2\} \cap \text{SPIdeal}_\sigma(e_1, e_2) = \emptyset$ ’ (due to Lemma 8.2). The function **ComputeSPClosure** performs a fixpoint computation, starting from the set $I = \bigcup_{i \in \{1,2\}} \text{TLClosure}_\sigma(\{\text{pred}_\sigma(e_i)\})$ (when $I_0 = \emptyset$). The correctness of the algorithm follows from the correctness of the function **ComputeSPClosure**, which we formalize below.

Lemma 8.4. Let σ be a trace, $e_1, e_2 \in \text{Events}_\sigma$ and $I_0 \subseteq \text{Events}_\sigma$ be a $(\leq_{TO}^\sigma, \text{lastWrite}_\sigma)$ -closed set. Let I be the set returned by **ComputeSPClosure** (σ, e_1, e_2, I_0) in Algorithm 8.1. Then, $I = \text{SPClosure}_\sigma(I_0 \cup \{\text{pred}_\sigma(e_1)\} \cup \{\text{pred}_\sigma(e_2)\})$.

Proof (Sketch). The proof of correctness follows from Definition 8.2 and the observation that the smallest sync-preserving closed set that includes a set S can be determined in an iterative least fixed point loop as computed in Algorithm 8.1. QED.

Let us discuss the data-structures we use to ensure that Algorithm 8.1 runs in linear time and space.

Vector timestamps. Vector timestamps [50,51] are routinely used in distributed computing and also in prior work on race prediction [48,52] (also see Chapter 6 and Chapter 7). We use vector timestamps to represent sets of events that are $(\leq_{TO}^\sigma, \text{lastWrite}_\sigma)$ -closed; a formal

definition is deferred to Section 8.2.4. In Algorithm 8.1, the sets $\text{TLClosure}_\sigma(\{\text{pred}_\sigma(e_i)\})$ are $(\leq_{\text{TO}}, \text{lastWrite})$ -closed (Line 2). Further, the initial value $I_0 = \emptyset$ ensures that all subsequent values of I in `ComputeSPClosure` are $(\leq_{\text{TO}}, \text{lastWrite})$ -closed. All these sets can be represented as vector timestamps. The advantage of using vector timestamps is two-fold. First, these timestamps provide a succinct representation of sets — instead of representing a set explicitly as a collection of events, a vector timestamp only uses $\mathcal{T}$ integers. Second, the vector timestamps for $(\leq_{\text{TO}}, \text{lastWrite}_\sigma)$ -closed sets can be computed in a streaming fashion, incrementally, using vector timestamps of smaller subsets.

Projecting a trace to threads and locks. Let us consider the check in Line 4. Here, we look for two acquire events $a_1 \leq_{\text{tr}}^\sigma a_2$ in the current ideal I that acquire the same lock ℓ . How do we efficiently discover two such acquires? A straightforward but naive way is to enumerate all pairs of events in I and check if they are acquire events of the above kind. But this can take $O(\mathcal{N}^3)$ time, where $\mathcal{N} = |\text{Events}_\sigma|$ because the number of such pairs can be $O(\mathcal{N}^2)$ in the worst case and the number of times the ideal can change is $O(\mathcal{N})$. Instead, we rely on the following observation.

Proposition 8.1. Let $I \subseteq \text{Events}_\sigma$ be downward closed with respect to $\leq_{\text{TO}}^\sigma$. For every $t \in \text{Threads}_\sigma$ and every $\ell \in \text{Locks}_\sigma$, there is at most one acquire event a with $\text{thr}(a) = t$, $\text{op}(a) = \text{acq}(\ell)$ such that $a \in I$ and $\text{match}_\sigma(a) \notin I$. When such an event a exists, then $\text{match}_\sigma(a') \in I$ for every other acquire $a' <_{\text{TO}}^\sigma a$ of the form $\text{type}(a') = \langle t, \text{acq}(\ell) \rangle$.

Proof (Sketch). Assume on the contrary that there are two unmatched acquire events $a_1 \neq a_2$ both of type $\langle t, \text{acq}(\ell) \rangle$, in I . Without loss of generality, assume that $a_1 <_{\text{TO}}^\sigma a_2$. Then, by well-formedness, we have that $\text{match}_\sigma(a_1) <_{\text{TO}}^\sigma a_2$. Since I is closed under $\leq_{\text{TO}}^\sigma$, we must have $\text{match}_\sigma(a_1) \in I$, contradicting that a is unmatched in I . The second observation follows again from the observation that I is closed under $\leq_{\text{TO}}^\sigma$ and that $\text{match}_\sigma(a') \in I$. QED.

The above observation can be exploited as follows. Let I be the current ideal and let $e_{t,\ell}^I$ be the last acquire on lock ℓ performed by thread t such that $e_{t,\ell}^I \in I$. Let $\text{Acq}_\ell^I = \{e_{t,\ell}^I\}_{t \in \text{Threads}_\sigma}$ be the set of last acquire events in I of every thread. Let e_ℓ^I be the last event (according to trace order $\leq_{\text{tr}}^\sigma$) in Acq_ℓ^I . Then, for every other acquire $e' \in \text{Acq}_\ell^I \setminus \{e_\ell^I\}$, the matching release $\text{match}_\sigma(e')$ must be added in I . Hence, if we can efficiently determine the events $e_{t,\ell}^I$ each time, then we can also efficiently determine e_ℓ^I and thus efficiently perform the closure each time. So, how do we determine $e_{t,\ell}^I$ efficiently each time? We achieve this by maintaining a FIFO sequence $\text{CSHist}_{t,\ell}$, for every thread $t \in \text{Threads}_\sigma$ and lock $\ell \in \text{Locks}_\sigma$. For every critical section on lock ℓ in thread t , there is a corresponding entry in the list $\text{CSHist}_{t,\ell}$, and the order of these entries is the same as the order in which these critical sections were

Algorithm 8.2: *Checking if there is a sync-preserving race on a given event with a given thread*

Input: Trace σ , Event e of type $\langle t, a(x) \rangle$, Thread u

```

1 for each  $b \in \{\mathbf{r}, \mathbf{w}\}$  such that  $b \asymp a^2$ 
2   let  $L_b$  be the list of events  $e'$  of the form  $\langle u, b(x) \rangle$  such that  $e' \leq_{\text{tr}}^\sigma e$ 
3 for each  $b \in \{\mathbf{r}, \mathbf{w}\}$  such that  $b \asymp a$ 
4    $I_b \leftarrow \emptyset$ 
5   for each  $e' \in L_b$ 
6      $I_b \leftarrow \text{ComputeSPClosure}(\sigma, e', e, I_b)$ 
7     if  $e' \notin I_b$  then
8       declare ‘race’ and exit

```

performed in the trace σ . Every entry (corresponding to critical section with acquire a) is a pair $(\text{TLClosure}_\sigma(a), \text{TLClosure}_\sigma(\text{match}_\sigma(a)))$, represented as a pair of vector timestamps $(C_a, C_{\text{match}(a)})$.

Let us now see how we perform the check in Line 4 using these data structures $\{\text{CSHist}_{t,\ell}\}_{t,\ell}$. For the current ideal I , we essentially need to determine the last acquire $e_{t,\ell}^I$ of each thread t and lock ℓ that belongs to I . This can be done by traversing the list $\text{CSHist}_{t,\ell}$ starting from the earliest entry, until we encounter the entry corresponding to the last acquire $e_{t,\ell}^I$ that belongs to I (this corresponds to a simple timestamp comparison). All entries in $\text{CSHist}_{t,\ell}$ prior to the identified event $e_{t,\ell}^I$ can then be discarded from $\text{CSHist}_{t,\ell}$, because the ideal now contains their information and only grows monotonically through the course of the fixpoint computation and the discarded entries will not be of use from now on. Thus, every entry in the lists $\{\text{CSHist}_{t,\ell}\}_{t,\ell}$ is traversed only once and the overall fixpoint computation runs in linear time when the number of $\mathcal{T}$ is constant.

8.2.3 Checking for a Sync-Preserving Race on a Given Event with a Given Thread

Let us now consider Algorithm 8.2. This algorithm takes as input a trace σ , an event $e = \langle t, a(x) \rangle$ and a threads $u \neq t$, and checks if there is an event e' of thread u such that $e' \leq_{\text{tr}}^\sigma e$ and (e', e) is a sync-preserving race. Algorithm 8.2 works as follows. For the sake of simplicity, assume that e is a read event, i.e., $a = \mathbf{r}$. The algorithm assumes access to the list $L_{\mathbf{w}}$ of write events $e' = \langle u, \mathbf{w}(x) \rangle$ that appear prior to e (Line 2); these lists can be constructed in linear time, in a single pass traversal of the trace. The algorithm simply traverses $L_{\mathbf{w}}$ (according to trace order $\leq_{\text{tr}}^\sigma$) and checks for races with each event in $L_{\mathbf{w}}$, by

² $b \asymp a$ whenever not both b and a are read ($\mathbf{r}$) operations

Algorithm 8.3: *Detailed streaming algorithm for checking sync-preserving races*

<pre> 1 handler acquire(t, ℓ) 2 $\mathbb{C}_t := \mathbb{C}_t[t \mapsto \mathbb{C}_t(t) + 1]$ 3 $g_\ell := g_\ell + 1$ 4 foreach 5 $t_1, t_2 \in \text{Threads}, a_1, a_2 \in \{\mathbf{r}, \mathbf{w}\}, x \in \text{Vars}$ 6 do 7 $\text{CSHist}_{t,\ell}^{\langle t_1, t_2, a_1, a_2, x \rangle} \cdot$ 8 $\text{addLast}((g_\ell, \mathbb{C}_t, \perp))$ 9 10 handler release(t, ℓ) 11 $\mathbb{C}_t := \mathbb{C}_t[t \mapsto \mathbb{C}_t(t) + 1]$ 12 foreach $t_1, t_2 \in \text{Threads}, a_1, a_2 \in \{\mathbf{r}, \mathbf{w}\},$ 13 $x \in \text{Vars}$ do 14 $\text{CSHist}_{t,\ell}^{\langle t_1, t_2, a_1, a_2, x \rangle} \cdot \text{last}() \cdot$ 15 $\text{updateRel}(\mathbb{C}_t)$ 16 17 handler fork(t, u) 18 $\mathbb{C}_t := \mathbb{C}_t[t \mapsto \mathbb{C}_t(t) + 1]$ 19 $\mathbb{C}_u := \mathbb{C}_t$ 20 21 handler join(t, u) 22 $\mathbb{C}_t := \mathbb{C}_t[t \mapsto \mathbb{C}_t(t) + 1]$ 23 $\mathbb{C}_t := \mathbb{C}_t \sqcup \mathbb{C}_u$ </pre>	<pre> 24 handler read(t, x) 25 $C_{\text{pred}} := \mathbb{C}_t$ 26 $\mathbb{C}_t := \mathbb{C}_t[t \mapsto \mathbb{C}_t(t) + 1] \sqcup \text{LW}_x$ 27 foreach $u \in \text{Threads}$ do 28 $\text{AccessHist}_{t,\mathbf{r},x}^{\langle u \rangle} \cdot \text{addLast}(C_{\text{pred}}, \mathbb{C}_t)$ 29 foreach $u \neq t \in \text{Threads}$ do 30 $I := \mathbb{I}^{\langle u, t, \mathbf{w}, \mathbf{r}, x \rangle} \sqcup C_{\text{pred}}$ 31 $\mathbb{I}^{\langle u, t, \mathbf{w}, \mathbf{r}, x \rangle} :=$ 32 $\text{checkRace}(\text{AccessHist}_{u,\mathbf{w},x}^{\langle u \rangle}, I,$ 33 $\langle u, t, \mathbf{w}, \mathbf{r}, x \rangle)$ 34 35 handler write(t, x) 36 $C_{\text{pred}} := \mathbb{C}_t$ 37 $\mathbb{C}_t := \mathbb{C}_t[t \mapsto \mathbb{C}_t(t) + 1]$ 38 $\text{LW}_x := \mathbb{C}_t$ 39 foreach $u \in \text{Threads}$ do 40 $\text{AccessHist}_{t,\mathbf{w},x}^{\langle u \rangle} \cdot \text{addLast}(C_{\text{pred}}, \mathbb{C}_t)$ 41 foreach $u \neq t \in \text{Threads}, a \in \{\mathbf{r}, \mathbf{w}\}$ do 42 $I := \mathbb{I}^{\langle u, t, a, \mathbf{w}, x \rangle} \sqcup C_{\text{pred}}$ 43 $\mathbb{I}^{\langle u, t, a, \mathbf{w}, x \rangle} :=$ 44 $\text{checkRace}(\text{AccessHist}_{u,a,x}^{\langle u \rangle}, I,$ 45 $\langle u, t, a, \mathbf{w}, x \rangle)$ </pre>
--	---

computing the fixpoint closure sets as in Algorithm 8.1. Instead of computing the ideal from scratch, the algorithm exploits the monotonicity property outlined earlier in Lemma 8.3 by reusing the ideal $I_{\mathbf{w}}$ computed so far. As with Algorithm 8.1, this algorithm also uses vector timestamps and maintains lists $\{\text{CSHist}_{t,\ell}\}_{t \in \text{Threads}, \ell \in \text{Locks}}$ for computing successive ideals efficiently. Overall again, each entry in $\{\text{CSHist}_{t,\ell}\}_{t \in \text{Threads}, \ell \in \text{Locks}}$ is visited a constant number of times and thus Algorithm 8.2 runs in linear time and uses linear space.

8.2.4 Algorithm SyncP for Sync-Preserving Race Prediction

The pseudo-code for SyncP is presented in Algorithm 8.3 (with helper functions defined in Algorithm 8.4). This is a one pass streaming algorithm that processes events as they appear in the trace, modifying its state and detecting races on the fly. The algorithm maintains several data structures including vector clocks and FIFO queues in its state. We first describe these data structures, then discuss how the algorithm initializes and modifies

Algorithm 8.4: *Auxiliary functions for Algorithm 8.3*

```

1 function Initialization()
2   foreach  $t \in \text{Threads}$  do
3      $\mathbb{C}_t := \perp$ 
4   foreach  $x \in \text{Vars}$  do
5      $\mathbb{LW}_x := \perp$ 
6   foreach  $\ell \in \text{Locks}$  do
7      $g_\ell := 0$ 
8   foreach  $t_1 \neq t_2 \in \text{Threads}, a_1 \prec a_2 \in \{\mathbf{r}, \mathbf{w}\}, x \in \text{Vars}$  do
9      $\mathbb{I}^{(t_1, t_2, a_1, a_2, x)} := \perp$ 
10    foreach  $t \in \text{Threads}, \ell \in \text{Locks}$  do
11       $\text{CSHist}_{t, \ell}^{(t_1, t_2, a_1, a_2, x)} := \emptyset$ 
12  foreach  $u \in \text{Threads}$  do
13    foreach
14       $t \in \text{Threads}, a \in \{\mathbf{r}, \mathbf{w}\}, x \in \text{Vars}$  do
15         $\text{AccessHist}_{t, a, x}^{(u)} := \emptyset$ 
16  function maxLB( $U, Lst$ )
17     $(g_{\max}, C_{\max}, C'_{\max}) := (0, \perp, \perp)$ 
18    while not  $Lst \cdot \text{isEmpty}()$  do
19       $(g, C, C') := \text{First}_{Lst}$ 
20      if  $C \sqsubseteq U$  then
21         $(g_{\max}, C_{\max}, C'_{\max}) := (g, C, C')$ 
22      else
23        break
24       $Lst \cdot \text{removeFirst}()$ 
25  function ComputeSPClosure( $I, \langle t_1, t_2, a_1, a_2, x \rangle$ )
26    repeat
27      foreach  $\ell \in \text{Locks}$  do
28        foreach  $t \in \text{Threads}$  do
29           $(g_{\ell, t}, C_{\ell, t}, C'_{\ell, t}) := \text{maxLB}(I, \text{CSHist}_{t, \ell}^{(t_1, t_2, a_1, a_2, x)})$ 
30           $t_{\max} := \text{argmax}_{t \in \text{Threads}} \{g_{\ell, t}\}$ 
31           $I := I \sqcup \bigsqcup_{t \neq t_{\max} \in \text{Threads}} C'_{\ell, t}$ 
32    until  $I$  does not change
33    return  $I$ 
34  function checkRace( $Lst, I, \langle t_1, t_2, a_1, a_2, x \rangle$ )
35    while not  $Lst \cdot \text{isEmpty}()$  do
36       $(C_{\text{pred}}, C) := \text{First}_{Lst}$ 
37       $I := \text{ComputeSPClosure}(I \sqcup C_{\text{pred}}, \langle t_1, t_2, a_1, a_2, x \rangle)$ 
38      if  $C \not\sqsubseteq I$  then
39        declare  $(a_1, a_2)$  race on  $x$ 
40        break
41       $Lst \cdot \text{removeFirst}()$ 
42    return  $I$ 

```

them as it processes the trace, and finally discuss the time and space usage for this algorithm; many of these details have already been spelt out in Sections 8.2.1-8.2.3.

Data structures and initialization. The algorithm maintains the following data structures.

1. **Vector clocks.** The algorithm uses vector clocks primarily for two purposes. First, we assign timestamps to all events in the trace and use the following vector clocks for this purpose — for every thread, a dedicated clock $\mathbb{C}_t$, and for every variable x , a dedicated clock $\mathbb{LW}_x$. The timestamp of an event e is essentially a succinct representation of the set $\text{TLClosure}(e)$. Next, the algorithm computes $\text{SPIdeal}(\cdot, \cdot)$ sets and represents

them as timestamps. These are stored in vector clocks $\mathbb{I}^{\langle t_1, t_2, a_1, a_2, x \rangle}$, one for every tuple $(t_1, t_2, a_1, a_2, x) \in \text{Threads} \times \text{Threads} \times \{\mathbf{r}, \mathbf{w}\} \times \{\mathbf{r}, \mathbf{w}\} \times \text{Vars}$. All vector clocks are initialized with the timestamp $\perp = \lambda t.0$.

2. **Scalars.** For every lock ℓ , the algorithm maintains a scalar variable g_ℓ to record the index of the last acquire on ℓ seen in the trace seen so far. Each such scalar is initialized with 0.
3. **FIFO queues.** The algorithm maintains several FIFO queues, whose entries correspond to different events in the trace. The algorithm ensures that an event appears only once, and additionally ensures that the entries respect the order of appearance of the corresponding events in the trace. The first kind of FIFO queues are used in the fixpoint computation. For this, the algorithm maintains queues $\text{CSHist}_{t,\ell}^{\langle t_1, t_2, a_1, a_2, x \rangle}$, one for every thread t , lock ℓ and tuple (t_1, t_2, a_1, a_2, x) . Each entry of $\text{CSHist}_{t,\ell}^{\langle t_1, t_2, a_1, a_2, x \rangle}$ is a triplet (g_e, C_e, C'_e) and corresponds to an acquire event e of the type $\langle t, \text{acq}(\ell) \rangle$ — here, g_e is the index of e in the trace, C_e is the timestamp of e and C'_e is the timestamp of the matching release $\text{match}(e)$. The second kind of queues are of the form $\text{AccessHist}_{t,a,x}^{(u)}$, one for each $t, u \in \text{Threads}$, $a \in \{\mathbf{r}, \mathbf{w}\}$ and $x \in \text{Vars}$. Entry in such a FIFO queue corresponds to access events e of the form $\langle t, a(x) \rangle$. Each entry is of the form $(C_{\text{pred}(e)}, C_e)$ where $C_{\text{pred}(e)}$ is the timestamp of $\text{pred}(e)$ (and $\perp$ if $\text{pred}(e)$ does not exist) and C_e is the timestamp of event e . All FIFO queues are empty ($\emptyset$) in the beginning. We write First_L and $L \cdot \text{last}()$ to denote the first and last elements of the FIFO queue L . Further, $L \cdot \text{isEmpty}()$, $L \cdot \text{addLast}()$ and $L \cdot \text{removeFirst}()$ respectively represent functions that check for emptiness of L , add an element at the end of L and remove the earliest (first) element of L .

Let us now describe the working of **SyncP**. The algorithm works in a streaming fashion and processes each event e as soon as it appears, by calling the appropriate **handler** depending upon the operation performed in e (**read**, **write**, **acquire** or **release**). The argument for each handler is the thread performing the event and the object (variable or lock) accessed in the event. In each handler, the algorithm updates vector clocks to compute timestamps, and maintains the invariants of the FIFO queues. In addition, inside **read** and **write** handlers, the algorithm also checks for races using a fixpoint computation (function **ComputeSPClosure**). We explain some of these briefly.

Computing vector timestamps. The algorithm computes the timestamp of an event e , denoted C_e when processing the event e . The timestamps computed in the algorithm are

such that

$$\forall t \in \text{Threads}, C_e(t) = |\{f \in \text{Events} \mid \text{thr}(f) = t, f \in \text{TLCClosure}(e)\}| \quad (8.2)$$

Observe that, with this invariant, we have $C_e \sqsubseteq C_{e'}$ iff $e \in \text{TLCClosure}(e')$. The algorithm uses vector clocks $\{\mathbb{C}_t\}_{t \in \text{Threads}}$ and $\{\mathbb{LW}_x\}_{x \in \text{Vars}}$ and ensures that after processing an event e , 1. $\mathbb{C}_t$ stores the timestamp C_{e_t} , where e_t is the last event by thread t that occurs before e , and 2. $\mathbb{LW}_x$ stores the timestamp C_{e_x} , where e_x is the last event with $\text{op}(e_x) = \mathbf{w}(x)$ that occurs before e . The algorithm correctly maintains these values by appropriate vector clock operations on Lines 18, 27, 3, 7, 11 and 14.

Let us now describe the invariant for the vector clocks $\mathbb{I}^{(t_1, t_2, a_1, a_2, x)}$ (for a given tuple (t_1, t_2, a_1, a_2, x)). Let $e_{t_i, a_i, x}$ be the last event with $\text{thr}(e_{t_i, a_i, x}) = t_i$ and $\text{op}(e_{t_i, a_i, x}) = a_i(x)$ ($i \in \{1, 2\}$). Let $I \subseteq \text{Events}_\sigma$ be defined as follows. If $e_{t_1, a_1, x}$ does not exist, then $I = \text{SPClosure}_\sigma(\{\text{pred}_\sigma(e_2)\})$. Otherwise, let e be the first event in σ (according to trace order $\leq_{\text{tr}}^\sigma$) with $\text{thr}(e) = t_1$ and $\text{op}(e) = a_1(x)$ such that (e, e_2) is a sync-preserving race of σ . If no such event exists, then let $e = e_{t_1, a_1, x}$. Then, $I = \text{SPIdeal}_\sigma(e_1, e_2)$. Then, the timestamp stored in $\mathbb{I}^{(t_1, t_2, a_1, a_2, x)}$ is $\bigsqcup_{u \in \text{Threads}_\sigma} C_{e_u}^I$, where e_u^I is the last event of thread u which is in I .

Checking races. When processing an access event e of the type $\langle t, a(x) \rangle$, the algorithm checks for a race as follows. For every other thread u and for every other conflicting type $b \asymp a$, the algorithm calls `checkRace` with the list $Lst = \text{AccessHist}_{u, b, x}^{(t)}$ and the timestamp representation of the current ideal as argument. This function, similar to Algorithm 8.2, scans Lst and reports races by repeatedly performing fixpoint computations and checking membership in some set (using the timestamp comparison in Line 38). The closure computation is performed using the optimizations discussed in Section 8.2.2 (use of FIFO queues $\text{CSHist}_{t, \ell}^{(\dots)}$).

Space optimizations. Observe that, for a given thread t and lock ℓ , the algorithm, as presented, maintains $4\mathcal{T}^2 \cdot \mathcal{V}$ FIFO data structures (or queues)

$\{\text{CSHist}_{t, \ell}^{(t_1, t_2, a_1, a_2, x)}\}_{t_1, t_2 \in \text{Threads}, a_1, a_2 \in \{\mathbf{r}, \mathbf{w}\}, x \in \text{Vars}}$. The total number of entries across these queues will then be $O(\mathcal{A} \cdot 4\mathcal{T}^2 \cdot \mathcal{V})$. To this end, we observe that all the above data structures essentially have the same content, and are suffixes of a common queue corresponding to the critical sections on ℓ in thread t . Indeed, we exploit this redundancy and instead maintain a common underlying data-structure that stores all entries corresponding to acquires and releases on ℓ in t , and maintain a pointer for each $\langle t_1, t_2, a_1, a_2, x \rangle$. Such a pointer keeps track of the starting index of the FIFO queue $\text{CSHist}_{t, \ell}^{(t_1, t_2, a_1, a_2, x)}$. With this space optimization, we only store $O(\mathcal{A})$ entries along with additional $4 \cdot \mathcal{T}^3 \cdot \mathcal{V} \cdot \mathcal{L}$ pointers, one for every tuple $\langle t_1, t_2, a_1, a_2, x \rangle$ and every shared queue indexed by (t, ℓ) . The same observations also apply

to the FIFO queues $\{\text{AccessHist}_{t,a,x}^{(u)}\}_{u \in \text{Threads}}$

Lemma 8.5 (Correctness). For every access event e in the input trace σ , Algorithm 8.3 declares a race when processing e iff there is an event e' such that $e' \leq_{\text{tr}}^\sigma e$ and (e', e) is a sync-preserving race of σ .

The proof of Lemma 8.5 follows directly from the correctness of the semantics of clocks and other observations outlined in Section 8.2.1.

The time complexity of the algorithm can be determined as follows. The algorithm visits each entry in the FIFO queues $\text{AccessHist}_{t,a,x}^{(u)}$ once, performing constant number of vector clock operations, each running in $O(\mathcal{T})$ time. The total length of all these queues is $O(\mathcal{T} \cdot \mathcal{N})$ (more precisely, the number of access events in the trace). Similarly, the algorithm visits each entry in the FIFO queues $\text{CSHist}_{t,\ell}^{(t_1,t_2,a_1,a_2,x)}$ once, performing constantly many vector clock operations. The total number of entries in these queues is $O(\mathcal{T}^2 \cdot \mathcal{V} \cdot \mathcal{A})$. This gives us the following complexity for SyncP.

Lemma 8.6 (Complexity). Let σ be a trace with $\mathcal{T}$ threads, $\mathcal{L}$ locks, $\mathcal{V}$ variables and $\mathcal{N}$ events, of which $\mathcal{A}$ are acquire events. Then, Algorithm 8.3 runs in time $O(\mathcal{N} \cdot \mathcal{T}^2 + \mathcal{A} \cdot \mathcal{V} \cdot \mathcal{T}^3)$ and uses space $O(\mathcal{N} + \mathcal{T}^3 \cdot \mathcal{V} \cdot \mathcal{L})$ on input σ .

The proof of Theorem 8.1 follows from Lemma 8.5 and Lemma 8.6.

8.2.5 Linear Space and Quadratic Time-Space Lower Bound

In this section we prove the lower-bounds of Theorem 8.2 and Theorem 8.3, i.e., that any streaming algorithm for sync-preserving race prediction must essentially use linear space, while the time-space product of any algorithm for the problem must be quadratic in the length of the input trace. We will refer to the language $\mathcal{L}_n$ defined previously when proving lower bounds for WCP (see Section 7.2.4) (Equation (7.6)). We recall the language here again. Let n be a natural number. Then,

$$\mathcal{L}_n = \{u\#v \mid u, v \in \{0, 1\}^n \text{ and } u = v\} \quad (8.3)$$

Recall that there is a $\Omega(n)$ space lower bound for the problem of accepting words from this language (Lemma 7.5).

Reduction from $\mathcal{L}_n$ recognition to sync-preserving race prediction. Consider the language $\mathcal{L}_n$ for some n . We describe a transducer $\mathcal{A}_n$ such that, on input a string $s = u\#v$,

the output $\mathcal{A}_n(s)$ is a trace σ with 2 threads, $O(n \cdot \log n)$ events, $2 \cdot \log n + 2$ locks and two variables such that the following hold.

1. If $s \notin \mathcal{L}_n$, then σ has no predictable race.
2. If $s \in \mathcal{L}_n$, then σ has a sync-preserving race.

Moreover, $\mathcal{A}_n$ uses $O(\log n)$ working space. The transducer $\mathcal{A}_n$ uses two variables x and y , two sets of locks $A = \{a_1, \dots, a_{\log n}\}$ and $B = \{b_1, \dots, b_{\log n}\}$, plus two additional locks c and d . The trace σ consists of two local traces π_1, π_2 of threads t_1 and t_2 which encode the bits of u and v , respectively.

1. The local trace π_1 is constructed as follows. For every $i \in [n]$, π_1 contains two events. The first event e_i^1 accesses the variable x , and the second event f_i^1 accesses the variable y . e_i^1 is a write event $\mathbf{w}(x)$ if $u[i] = 1$, and a read event $\mathbf{r}(x)$ otherwise. f_i^1 is a read event $\mathbf{r}(y)$ if $u[i] = 1$, and a write event $\mathbf{w}(y)$ otherwise. The events e_i^1 and f_i^1 are each surrounded by the lock d . Additionally, they are enclosed in common critical sections of locks from A and B arbitrarily, as long as the following holds. For any $i < j$, we have

$$\begin{aligned} \text{locksHeld}_\sigma(e_j^1) \cap A &\not\subseteq \text{locksHeld}_\sigma(e_i^1) \cap A \\ \text{locksHeld}_\sigma(e_i^1) \cap B &\not\subseteq \text{locksHeld}_\sigma(e_j^1) \cap B. \end{aligned} \tag{8.4}$$

The same property also holds for the access events of y (f_i^1 and f_j^1) as e_i^1 and f_i^1 are enclosed in common critical sections of locks from A and B (i.e., $\text{locksHeld}_\sigma(e_i^1) \cap (A \cup B) = \text{locksHeld}_\sigma(f_i^1) \cap (A \cup B)$ for every i).

The above property can be easily met, for example, by making $\mathcal{A}_n$ perform a breadth-first traversal of the subset-lattice of A (resp., B) starting from the top (resp., bottom). Given the current i , the transducer surrounds e_i^1 and f_i^1 with the locks of the current element in the corresponding lattice. Finally, every write event e_i^1 (or f_i^1) is surrounded by the lock c .

2. The local trace π_2 is similar to π_1 , i.e., we have event e_i^2 followed by f_i^2 for each $1 \leq i \leq n$. e_i^2 (resp. f_i^2) is a write event $\mathbf{w}(x)$ (resp. a read event $\mathbf{r}(y)$) if $u[i] = 1$, otherwise it is a read event $\mathbf{r}(x)$ (resp. write event $\mathbf{w}(y)$). The locks that surround e_i^2 and f_i^2 are such that

$$\text{locksHeld}_\sigma(e_i^2) \cap (A \cup B) = A \cup B \setminus \text{locksHeld}_\sigma(e_j^1). \tag{8.5}$$

	t_1	t_2		t_1	t_2
1	acq(b_1)		33		nest ^d _{w₀(x)}
2	acq(b_2)		34		nest ^d _{w₀(y)}
3	acq(c)		35		acq(a_1)
4	nest ^d _{w₁(x)}		36		acq(a_2)
5	rel(c)		37		acq(c)
6	nest ^d _{r₁(y)}		38		w ₁ (x)
7	rel(b_2)		39		rel(c)
8	rel(b_1)		40		r ₁ (y)
9	acq(a_1)		41		rel(a_2)
10	acq(b_2)		42		rel(a_1)
11	nest ^d _{r₂(x)}		43		acq(b_1)
12	acq(c)		44		acq(a_2)
13	nest ^d _{w₂(y)}		45		r ₂ (x)
14	rel(c)		46		acq(c)
15	rel(b_2)		47		w ₂ (y)
16	rel(a_1)		48		rel(c)
17	acq(a_2)		49		rel(a_2)
18	acq(b_1)		50		rel(b_1)
19	acq(c)		51		acq(b_2)
20	nest ^d _{w₃(x)}		52		acq(a_1)
21	rel(c)		53		r ₃ (x)
22	nest ^d _{r₃(y)}		54		acq(c)
23	rel(b_1)		55		w ₃ (y)
24	rel(a_2)		56		rel(c)
25	acq(a_1)		57		rel(a_1)
26	acq(a_2)		58		rel(b_2)
27	acq(c)		59		acq(b_1)
28	nest ^d _{w₄(x)}		60		acq(b_2)
29	rel(c)		61		acq(c)
30	nest ^d _{r₄(y)}		62		w ₄ (x)
31	rel(a_2)		63		rel(c)
32	rel(a_1)		64		r ₄ (y)
			65		rel(b_2)
			66		rel(b_1)

Figure 8.3: Construction of the trace σ on input $s = u\#v$ where $u = 1011$ and $v = 1001$. nest_{op}^ℓ is a shorthand for the sequence $\text{acq}(\ell) \cdot \text{op} \cdot \text{rel}(\ell)$. Observe that the events $\langle t_1, \mathbf{r}_3(y) \rangle$ and $\langle t_2, \mathbf{w}_3(y) \rangle$ constitute a sync-preserving race, which encodes that $u[3] \neq v[3]$.

Finally, similarly to π_1 , every write event e_i^2 is surrounded by the lock c . Further, none of the events e_i^2 or f_i^2 are surrounded by the lock d (unlike in π_1). But, in the beginning of the local trace π_2 , we additionally perform $\mathbf{w}(x)$ and $\mathbf{w}(y)$ and surround

both of them by the lock d .

3. The trace is obtained by concatenating the two local traces π_1 followed by π_2 .

See Figure 8.3 for an illustration. Observe that $\mathcal{A}_n$ uses $O(\log n)$ bits of memory, for storing a bit-set of locks for each set A and B that must surround the current event e_i^1 (together with f_i^1) or e_i^2 (together with f_i^2).

The key idea of the construction is the following. There are no predictable races that involve the events $w_0(x)$ and $w_1(y)$ of thread t_2 as all conflicting events of thread t_1 are also surrounded by the common lock d . Thus, the only race can be between some event e_i^1 and e_j^2 or f_i^1 and f_j^2 . Next, the construction will ensure that any two events e_i^1, e_j^2 are surrounded by a common lock from the set $A \cup B$ iff $i \neq j$. Likewise, any two events f_i^1, f_j^2 are surrounded by a common lock from the set $A \cup B$ iff $i \neq j$. Hence, (e_i^1, e_j^2) or (f_i^1, f_j^2) may be a predictable race of σ only if $i = j$. In turn, if $u[i] = v[j]$, then either both e_i^1 and e_j^2 (and also both f_i^1 and f_j^2) are read events, or both are write events. In the former case the events are not conflicting, while in the latter case the two events are surrounded by lock c . In both cases no race occurs between e_i^1 and e_j^2 . On the other hand, if $u[i] \neq v[i]$, then one event is a read and the other is a write event. Hence, the two events conflicting, and one of them is not surrounded by lock c , thereby constituting a possible predictable race. We in fact show that in this case, one of (e_i^1, e_j^2) or (f_i^1, f_j^2) is in fact a sync-preserving race. The following lemma makes the above insight formal and establishes the correctness of our reduction.

Lemma 8.7. The following assertions hold.

1. If $s \in \mathcal{L}_n$, then σ has no predictable race.
2. If $s \notin \mathcal{L}_n$, then σ has a sync-preserving race.

Proof. We prove each item separately.

1. $s \in \mathcal{L}_n$. First, notice that by construction, for every $1 \leq i \leq n$, the pairs (e_i^1, e_i^2) and (f_i^1, f_i^2) are not predictable races of σ , as we have $u[i] = v[i]$ and thus either both events in these pairs are read events or both are write events and thus each is protected by the lock c . It remains to argue that (e_i^1, e_j^2) and (f_i^1, f_j^2) are not predictable races for any $1 \leq i, j \leq n$ with $i \neq j$. It suffices to show that $\text{locksHeld}_\sigma(e_i^1) \cap \text{locksHeld}_\sigma(e_j^2) \neq \emptyset$. We split cases based on the relation between i and j .

$i < j$. By construction, we have $\text{locksHeld}_\sigma(e_j^1) \cap B \not\subseteq \text{locksHeld}_\sigma(e_i^1) \cap B$. Moreover, we have $\text{locksHeld}_\sigma(e_j^2) \cap B = B \setminus \text{locksHeld}_\sigma(e_j^1)$. Thus, we have $\text{locksHeld}_\sigma(e_i^1) \cap \text{locksHeld}_\sigma(e_j^2) \cap B \neq \emptyset$, as desired.

$j < i$. By construction, we have $\text{locksHeld}_\sigma(e_i^1) \cap A \not\subseteq \text{locksHeld}_\sigma(e_j^1) \cap A$. Moreover, we have $\text{locksHeld}_\sigma(e_i^2) \cap A = A \setminus \text{locksHeld}_\sigma(e_i^1)$. Thus, we have $\text{locksHeld}_\sigma(e_i^1) \cap \text{locksHeld}_\sigma(e_j^2) \cap A \neq \emptyset$, as desired.

2. $s \notin \mathcal{L}_n$. Consider the first index i such that $u[i] \neq v[i]$. Observe that by construction either $(e_i^1$ is a read event and e_i^2 is a write event) or $(f_i^1$ is a read event and f_i^2 is a write event). In the first case, we will show that (e_i^1, e_i^2) is a sync-preserving race, and in the second case, we will show that (f_i^1, f_i^2) is a sync-preserving race. We use (g_1, g_2) to represent the pair (e_i^1, e_i^2) in the first case and (f_i^1, f_i^2) in the latter case. Observe that both these pairs have disjoint locks — lock c is only held by write events, lock d is only held by events of thread t_1 , and finally, by construction we have $\text{locksHeld}_\sigma(g_2) \cap (A \cup B) = A \cup B \setminus \text{locksHeld}_\sigma(g_1)$, and thus $\text{locksHeld}_\sigma(g_1) \cap \text{locksHeld}_\sigma(g_2) = \emptyset$. Now, consider the sequence ρ of events obtained by organizing the set of events that are either $\leq_{\text{TO}}$ before g_1 or g_2 as per the original trace order. We first remark that ρ is a well formed trace as the only critical sections that are open are on disjoint locks. Next, ρ has the same thread order as σ . Further, for every read event e in ρ , the last write event of e is ρ is the same as in σ , this is because, events in π_1 have last write in π_1 itself. In fact, even events in π_2 have last write events in π_2 (because of $\text{nest}_{w_0(x)}^d$ and $\text{nest}_{w_0(y)}^d$) in the beginning of π_2 . Thus, ρ is a correct reordering of σ . Further, it is a sync-preserving trace because no critical sections are flipped (notice that because of the choice of g_1 and g_2 , g_1 is always a read event and thus not surrounded by the lock c). Finally, both g_1 and g_2 are σ -enabled in ρ . Thus, σ has a sync-preserving race. QED.

We now prove Theorem 8.2.

Proof of Theorem 8.2. Consider any algorithm A_1 for sync-preserving race prediction, executed in the family of traces σ constructed in our above reduction. Let $m = n/\log n$, and assume towards contradiction that A_1 uses $o(m)$ space. Then we can pair A_1 with the transducer $\mathcal{A}_m$, and obtain a new algorithm A_2 for recognizing $\mathcal{L}_m$. Since $\mathcal{A}_m$ uses $O(\log m)$ space, the space complexity of A_2 is $o(m)$. However, this contradicts Lemma 7.5. The desired result follows. QED.

We now turn our attention to proving Theorem 8.3.

The set-equality problem. Our proof uses the set-equality problem, i.e., given two bit-sets $u, v \in \{0, 1\}^n$, the task is to decide whether $u = v$. The problem has a $\Omega(n)$ lower-bound for communication complexity [95], i.e., if u and v is given separately to Alice and Bob, respectively, the two parties need to exchange $\Omega(n)$ bits of information in order to decide whether $u = v$.

Proof of Theorem 8.3. Consider the language $\mathcal{L}'_n = \{u\#^nv : u, v \in \{0,1\}^n \text{ and } u = v\}$, and any Turing Machine M that decides $\mathcal{L}'_n$ using $T_M(n)$ time and $S_M(n)$ space. Let $K_M(n)$ be an upper-bound on the number of “passes” that M makes over the sequence $\#^n$ as it decides membership in $\mathcal{L}'_n$. In each pass, M “communicates” at most $S_M(n)$ bits of information. Since the set-equality problem has communication complexity $\Omega(n)$ [95], we have $K_M(n) \cdot S_M(n) = \Omega(n)$, i.e., M makes $\Omega(n/S_M(n))$ passes. Since each pass has to traverse n symbols $\#$, each pass costs time $\Omega(n)$. Hence, the total time is $T_M(n) = \Omega(n \cdot K_M(n))$, and thus $T_M(n) \cdot S_M(n) \geq n^2$.

We now turn our attention to sync-preserving race prediction. Let $m = n/\log n$. Using essentially the same reduction as above, we reduce the membership problem for $\mathcal{L}'_m$ to the sync-preserving race prediction problem on a trace σ with 2 threads, n events and $O(\log n)$ locks. Lemma 8.7 guarantees that σ has a predictable race iff $u \neq v$, and if so, then it has a sync-preserving race. Hence, any algorithm that solves sync-preserving race prediction on σ in time $T(n)$ and space $S(n)$ must satisfy that $T(n) \cdot S(n) = \Omega(m^2) = \Omega(n^2/\log^2 n)$. The desired result follows. QED.

8.3 EXPERIMENTS

In this section we report on an implementation and experimental evaluation of the techniques presented in this work. Our objective is two-fold. The first goal is to quantify the practical relevance of sync-preservation, i.e., whether in practice the definition captures races that are missed by the standard notion of happens-before and WCP [13] races. The second goal is to evaluate the performance of our algorithm **SyncP** for detecting sync-preserving races.

8.3.1 Experimental Setup

We have implemented **SyncP** (Algorithm 8.3) for predicting all sync-preserving races in our tool **RAPID** [86], written in Java, and evaluated it on a standard set of benchmarks.

Benchmarks. Our benchmark set consists of standard benchmarks found in the recent literature [13, 14, 35, 36, 44, 96]. It consists of 30 concurrent programs taken from standard benchmark suites: (i) the IBM Contest benchmark suite [87], (ii) the Java Grande forum benchmark suite [88], (iii) the DaCapo benchmark suite [89], (iv) the Software Infrastructure Repository [97], and (v) some standalone benchmarks. For each benchmark, we generated a single trace using RV-Predict [85] and evaluated all methods on the same trace.

Compared methods. We compare our algorithm with state-of-the-art sound race detectors, namely, SHB [14], WCP [13] and M2 [35]. Recall that SHB and WCP are linear time algorithms that perform a single pass of the input trace σ . SHB computes happens-before races and is sound even beyond the first race. On the other hand, WCP is only sound for the first race report. In order to allow WCP to soundly report more than one race, whenever a race is reported on an event pair (e_1, e_2) (i.e., we have $e_1 \parallel_{\text{WCP}}^\sigma e_2$), we force an order $e_1 \leq_{\text{WCP}}^\sigma e_2$ before proceeding with the next event of σ . This is a standard practice that has been followed in other works, e.g., [35, 36]. Finally, M2 is a more heavyweight algorithm that makes sound reports for all races by design, though its running time is a larger polynomial (of order n^4) [35].

Optimizations. In general, the benchmark traces can be huge and often scale to sizes of order as large as 10^8 . A closer inspection shows that many events, even though they perform accesses to shared memory, are non-racy and even totally ordered by fork-join mechanisms and data flows in the trace. We have implemented a lightweight, linear time, single-pass optimization of the input trace σ that filters out such events. The optimization simply identifies memory locations x whose conflicting accesses are totally ordered in σ by thread and data-flow orderings, and ignores all such accesses in σ . For a fair comparison, we employ the optimization in all compared methods. This is similar to FastTrack-like optimizations [52], which identify and ignore thread-local events. We note that, as each of the compared methods attempts to report as many races as possible, epoch-like optimizations were not applied.

Reported results. Our experiments were conducted on a 2.6GHz 64-bit Linux machine with Java 1.8 as the JVM and 30GB heap space. Each of the compared methods is evaluated on the same input trace σ . For every such input, the respective method reports the following race warnings.

1. *Racy events.* We report the number of events e_2 such that there is an event e_1 with $e_1 <_{\text{tr}} e_2$ for which a race (e_1, e_2) is detected.
2. *Racy source-code lines.* We report the number of distinct source-code lines which correspond to events e_2 that are found as racy in 1. This is a meaningful measure, as the same source-code line might be reported by many different events e_2 .
3. *Racy memory locations.* We report the number of different memory locations that are accessed by all the events e_2 that are found as racy in 1.

Table 8.1: Dynamic race reports in our benchmarks. $\mathcal{N}$ and $\mathcal{T}$ denote the number of events and number of threads in the respective trace. For races, an entry ‘ r (s)’ denotes the number r of events e_2 found to be in race with an earlier event e_1 , as well as the number s of unique source-code lines corresponding to such events e_2 . Bold-face entries highlight cases where there are sync-preserving races that are not happens-before races.

Benchmark	$\mathcal{N}$	$\mathcal{T}$	SHB		WCP		M2		SyncP	
			Races	Time	Races	Time	Races	Time	Races	Time
array	51	4	0 (0)	0.02s	0 (0)	0.03s	0 (0)	0.09s	0 (0)	0.04s
critical	59	5	3 (3)	0.19s	1 (1)	0.03s	3 (3)	0.11s	3 (3)	0.07s
account	134	5	3 (1)	0	3 (1)	0.06s	3 (1)	0.23s	3 (1)	0.09s
airtickets	140	5	8 (3)	0.02s	5 (2)	0.03s	8 (3)	0.13s	8 (3)	0.05s
pingpong	151	7	8 (3)	1.09s	8 (3)	0.04s	8 (3)	0.17s	8 (3)	0.06s
twostage	193	13	4 (1)	0.02s	4 (1)	0.09s	4 (1)	0.20s	4 (1)	0.10s
wronglock	246	23	12 (2)	0.02s	3 (2)	0.09s	25 (2)	0.43s	25 (2)	0.15s
bbuffer	332	3	3 (1)	0.01s	1 (1)	0.05s	3 (1)	0.11s	3 (1)	0.06s
prodcons	658	9	1 (1)	0.03s	1 (1)	0.09s	1 (1)	0.20s	1 (1)	0.10s
clean	1.0K	10	59 (4)	0.04s	33 (4)	0.14s	110 (4)	0.85s	60 (4)	0.17s
mergesort	3.0K	6	1 (1)	11m10s	1 (1)	0.12s	5 (2)	0.96s	3 (1)	0.13s
bubblesort	4.0K	13	269 (5)	0.03s	100 (5)	0.27s	374 (5)	8.05s	269 (5)	0.50s
lang	6.0K	8	400 (1)	0.10s	400 (1)	0.23s	400 (1)	1.31s	400 (1)	0.31s
readswrites	11K	6	92 (4)	0.12s	92 (4)	0.41s	228 (4)	12.74s	199 (4)	0.77s
raytracer	15K	4	8 (4)	0.02s	8 (4)	0.30s	8 (4)	0.40s	8 (4)	0.30s
bufwriter	22K	7	8 (4)	0.10s	8 (4)	0.70s	8 (4)	2.65s	8 (4)	0.84s
ftpserver	49K	12	69 (21)	6.91s	69 (21)	1.34s	85 (21)	4.11s	85 (21)	4.69s
moldyn	200K	4	103 (3)	0.05s	103 (3)	1.83s	103 (3)	1m25s	103 (3)	1.86s
linkedlist	1.0M	13	5.0K (4)	7.25s	5.0K (3)	27.07s	TO	TO	7.0K (4)	5m19s
derby	1.0M	5	29 (10)	0.01s	28 (10)	16.48s	30 (11)	22.49s	29 (10)	24.07s
jigsaw	3.0M	12	4 (4)	0.41s	4 (4)	19.53s	6 (6)	11.69s	6 (6)	17.30s
sunflow	11M	17	84 (6)	39.66s	58 (6)	47.14s	130 (7)	50.24s	119 (7)	55.30s
cryptorsa	58M	9	11 (5)	3m4s	11 (5)	6m35s	TO	TO	35 (7)	9m42s
xalan	122M	7	31 (10)	0.15s	21 (7)	15m30s	TO	TO	37 (12)	10m44s
lufact	134M	5	21K (3)	7m26s	21K (3)	14m57s	TO	TO	21K (3)	10m38s
batik	157M	7	10 (2)	9m49s	10 (2)	22m56s	TO	TO	10 (2)	11m59s
lusearch	217M	8	232 (44)	12.63s	119 (27)	13m40s	232 (44)	27m9s	232 (44)	14m5s
tsp	307M	10	143 (6)	15m2s	140 (6)	29m10s	TO	TO	143 (6)	20m19s
luindex	397M	3	1 (1)	24m40s	2 (2)	31m6s	TO	TO	15 (15)	31m46s
sor	606M	5	0 (0)	38m38s	0 (0)	TO	TO	TO	0 (0)	44m36s
Totals	2.0B	-	29520 (157)	1h51m	29133 (134)	≥ 3h15m	1846 (131)	≥ 8h30m	30862 (178)	2h40m

4. *Running time.* We measure the time of the algorithm required to process each benchmark, while imposing a 1-hour timeout (TO).

8.3.2 Experimental Results

We now turn our attention to the experimental results. Table 8.1 shows the races and running times reported by each method on each benchmark.

Coverage of sync-preserving races. We find that *every* race reported by SHB or WCP is a sync-preserving race, also reported by SyncP. On the other hand, bold-face entries

Table 8.3: Maximum race distances.

Table 8.2: Numbers of different memory locations that are detected as racy.

Benchmark	SHB	WCP	SyncP
ftpserver	49	49	50
jigsaw	4	4	5
xalan	7	6	9
cryptorsa	4	4	5
luindex	1	2	9
sunflow	14	10	17
linkedlist	927	927	932
Total	1006	1002	1027

Benchmark	SHB	WCP	SyncP
tsp	11K	11K	224M
batik	1.7M	1.7M	4.8M
cryptorsa	7.9M	7.9M	8.3M
jigsaw	428	428	121K
sunflow	10M	1.0M	10M
xalan	4K	4K	13K
ftpserver	11K	11K	11K
linkedlist	165K	165K	165K
luindex	783	783	6.9K
mergesort	57	57	1.4K
clean	355	47	1.2K
readswrites	13	13	696
wronglock	50	6	113

highlight benchmarks which have sync-preserving races that are not happens-before races. We see that such races are found in 11 out of 30 benchmarks. Interestingly, in the 5 most challenging out of these 11 benchmarks, the same pattern occurs if we focus on source-code lines (i.e., the entries in the parentheses). Hence, for these benchmarks, sync-preservation is *necessary* to capture many racy source-code lines, which happens-before would completely miss. We also remark that the more heavyweight analysis M2 misses several of these races due to frequent timeouts. In total, we have 18 unique source-code lines that are racy but only detected by SyncP. However, there are only 2 source-code lines that are caught by M2 but not by SyncP.

Running times. Our experimental times indicate that SyncP is quite efficient in practice. Among all algorithms, SyncP is the second fastest, being about 1.4 times slower than the fastest, lightweight SHB, while at the same time, being able to detect considerably more races than SHB (i.e., 1342 more racy events, and 21 more racy source-code lines). On the other hand, SyncP detects even more races than M2, due to timeouts, and even has almost equal detection capability with M2 on the cases that M2 does not time out. Due to the slow performance of M2 (i.e., over 8.5 hours and with several timeouts), we exclude it from the more refined analysis that follows.

Racy memory locations. We next proceed to evaluate the capability of SyncP in detecting racy memory locations. As all races detected by SHB or WCP are sync-preserving, the same follows for the racy memory locations, i.e., they are all detected as racy by SyncP. On the other hand, Table 8.2 shows a few cases in which SyncP has discovered racy variables that are missed by SHB and WCP. Hence, sync-preservation is more adequate to capture not only racy program locations, but also racy memory locations. We note that, in principle, many

different racy memory locations could correspond to the same static race (e.g., if memory is allocated dynamically). Note, however, that the additional reports of **SyncP** in Table 8.2 occur on benchmarks where it also makes more race reports in Table 8.1. Together, the two experimental tables give confidence that the new reported races are on entirely different variables.

Race distances. We examine the capability of **SyncP** to detect races that are far apart in the input trace. Table 8.3 shows maximum race distance of races (e_1, e_2) in various benchmarks, including the ones that contains sync-preserving races that are missed by happens-before. In each case, the distance is counted as the number of events in the input trace between e_1 and e_2 , for every event e_2 reported as racy. We see a sharp contrast between **SHB/WCP** and **SyncP**, with the latter being able to detect races that are far more distant in the input. This is in direct alignment with our theoretical observations already illustrated earlier in Figure 8.1c. Indeed, as partial orders, **SHB/WCP** can only detect races between conflicting accesses that are successive in the input trace. On the other hand, sync-preserving races may be interleaved with arbitrarily many conflicting, non-racy accesses, and our complete algorithm **SyncP** is guaranteed to detect them. Overall, all our experimental observations suggest that sync-preservation elegantly characterizes almost all races that are efficiently detectable, while having much higher predictive power than **HB**.

CHAPTER 9: COMPLEXITY OF DYNAMIC RACE PREDICTION

While the data race prediction problem is clearly in **NP** — guess a correct reordering and check if it witnesses a data race — its precise complexity was not identified in the literature prior to the work presented here (published in [17]). Evidence based on prior work, suggests a belief that the problem might be **NP**-complete. First, related problems, like data-race detection for programs with strong synchronization primitives [98–100], or verifying sequential consistency [101], are known to be **NP**-hard. Second, all known “complete” (or maximal) algorithms run in worst-case exponential time. These approaches either rely on an explicit enumeration of all correct reorderings [102, 103] or they are symbolic approaches that reduce the race prediction problem to a constraint satisfaction problem [43, 44, 104]. On the other hand, a slew of “partial order”-based methods have been proposed, whose goal is to predict data races in polynomial time, but at the cost of being incomplete and failing to detect data races in some traces. These include algorithms based on the classical *happens-before* partial order [14, 47, 50–53], and those based on newer partial orders that improve the prediction of data races over happens-before [13, 34–36].

In this chapter we study the problem of data-race prediction from a complexity-theoretic perspective. Our goal is to understand whether the problem is intractable, the causes for intractability, and conditions under which it can be solved efficiently. We provide partial answers to these questions, and in some cases characterize the tractability/intractability landscape precisely in the form of optimality results.

Contributions. Consider an input trace σ with $\mathcal{N}$ over $\mathcal{T}$ threads. Our main contributions are as follows. We refer to Section 9.1.2 for a formal summary.

Our first result shows that the data-race prediction problem is solvable in $O(\mathcal{T} \cdot \mathcal{N}^{2 \cdot (\mathcal{T}-1)})$ time, and can be improved to $O(\mathcal{N}^{\mathcal{T}})$ when certain additional parameters of σ are constant. We note that when $\mathcal{T}$ is a constant, as is the case with many software applications, our upper-bound is polynomial.

The observation that data race prediction is in polynomial time for constantly many threads naturally leads to two follow-up questions. Does the problem remain tractable for any $\mathcal{T}$? And if not, is it fixed parameter tractable (FPT) with respect to $\mathcal{T}$, i.e., is there an algorithm with running time of the form $O(f(\mathcal{T}) \cdot \mathcal{N}^{O(1)})$? Our second result answers both these questions in the negative, by showing that the problem is **W**[1]-hard. This formally establishes the **NP**-hardness of the problem, and excludes efficient algorithms when $\mathcal{T}$ is even moderately large (e.g., $\mathcal{T} = \Omega(\log \mathcal{N})$).

We then investigate whether there are practically relevant contexts where data-race pre-

diction is more efficiently solvable, i.e., the degree of the polynomial is fixed and independent of $\mathcal{T}$. We consider the case of traces over acyclic communication topologies, such as pipelines, server-clients hierarchies and divide-and-conquer parallelism. Our third result shows that, perhaps surprisingly, over such topologies data-race prediction can be solved in $O(\mathcal{T}^2 \cdot d \cdot \mathcal{N}^2 \cdot \log \mathcal{N})$ time, where d is the total number of synchronization variables (locks) and global memory locations.

In practice, the size $\mathcal{N}$ of the trace is by far the dominating parameter, while $\mathcal{T}$ and d are many orders of magnitude smaller. Hence, given the above upper-bound, the relevant question is whether the complexity on $\mathcal{N}$ can be improved further. Our fourth result shows that this is unlikely: we show that, under the Orthogonal Vectors conjecture, there is no $O(\mathcal{N}^{2-\epsilon})$ algorithm even for traces restricted to only 2 threads. As any trace with 2 threads induces an acyclic topology, our upper-bound is (conditionally) optimal up to polynomial improvements.

Finally, the majority of practical data-race prediction heuristics search for a data race witness among correct reorderings that are very similar to the observed trace σ , i.e., by only attempting a few event reorderings on σ . Motivated by these approaches, we investigate the complexity of a distance-bounded version of data-race predication, where the goal is to expose a data race by only looking at correct reorderings of σ that are a small distance away. Here, distance between traces is measured by the number of critical sections and write events whose order is reversed from σ . Our fifth result is a linear-time (and thus, optimal) algorithm for this problem, when certain parameters of the trace σ are constant. This result gives a solid basis for the principled development of fast heuristics for dynamic data-race prediction.

Technical contributions. Towards the main results in this chapter, we make several technical contributions that might be of independent interest. We summarize some of them below.

1. We improve the lower-bound of the well-known problem on verifying sequential consistency with read-mapping (VSC-rm) [101] from the long-lasting NP-hardness to W[1]-hard.
2. We show that VSC-rm can be solved efficiently on tree communication topologies of any number of threads, which improves a recent result of [35] for only 2 threads, as well as a result of [105] for more than 2 threads.
3. The first challenge in data-race prediction given a trace σ is to choose the set X of events of σ over which to attempt to construct a correct reordering. Identifying such

choices for X is a significant challenge [35, 36, 44]. We establish non-trivial upper-bounds on the number of choices for X , and show that they are constantly many when certain parameters of σ are constant.

4. Particularly for tree communication topologies, we show that a single choice for such X suffices.

Finally, we note that our notion of a predictable data race in a trace σ requires as a witness a reordering σ^* of σ in which every read event reads from the same write event as in σ . This guarantees that σ^* is valid in any program that produced σ . More permissive reorderings, e.g., requiring that every read event reads the same value, are also possible, and can capture potentially more races. Our notion of witness reflects the most common practice in race-detection literature, where trace logging typically does not track the values.

Related Work. Antoni Mazurkiewicz [41] [41, 106] used the notion of *traces* to mathematically model executions of concurrent programs. Bertoni et. al. [107] studied various language-theoretic questions about Mazurkiewicz traces. The folklore results about the NP-hardness of race detection are often attributed to Netzer and Miller [98–100]. However, the problem considered in their work differs in significant ways from the problem of data-race prediction. First, the notion of feasible executions in [98] (the counterpart of the notion of correct reorderings) requires that any two conflicting events be ordered in the same way as the observed execution, and hence, are less permissive. Next, the NP-hardness arises from the use of complex synchronization primitives like `wait` and `signal`, which are more powerful than the primitives we study here (release/acquire of locks and read/write of registers). The results due to Netzer and Miller, thus, do not apply to the problem of data-race prediction. Gibbons and Korach [101] establish NP-hardness for a closely related problem in distributed computing — verifying sequential consistency with a read mapping (VSC-rm). Yet again, the problem is different than the problem of race prediction. Complexity theoretic investigations have also been undertaken for other problems in distributed computing like linearizability [31, 101] and serializability [108]. Hence, although there have been many theoretical results on related problems in concurrency, none of them addresses dynamic data-race prediction. Our work fills this gap.

9.1 NOTATIONS, PROBLEM STATEMENTS AND SUMMARY OF RESULTS

Let us introduce some notations specific to the presentation in this chapter.

General notation. Given a natural number k , let $[k] = \{1, \dots, k\}$. Given a function $f : X \rightarrow Y$, we let $\text{dom}(f) = X$ and $\text{img}(f) = Y$. Given two functions f, g , we write $f \subseteq g$ to denote that $\text{dom}(f) \subseteq \text{dom}(g)$ and for every $x \in \text{dom}(f)$ we have $f(x) = g(x)$. Given a set $X' \subseteq \text{dom}(f)$, we denote by $f|_{X'}$ the function with $\text{dom}(f|_{X'}) = X'$ and $f|_{X'} \subseteq f$.

Notations for Events and Traces. In this chapter, we assume that traces do not contain fork and join events. This is not a limitation and all the results can be lifted to the case when threads are created dynamically. Further, for the sake of simplicity of presentation, we will assume locks are not re-entrants; this again does not affect any of our results.

For an event $e \in \text{Events}$, we use $\text{loc}(e)$ to denote the variable or lock that e accesses; recall that in this chapter we are assuming that **Events** only contains read, write, acquire or release events. We occasionally write $e(o)$ to denote an event e with $\text{loc}(e) = o$. For a thread $t \in \text{Threads}$, we denote by Writes^t (resp. Reads^t , Acquires^t , Releases^t) the set of all write (resp. read, acquire, release) events e for which $\text{thr}(e) = t$. Finally we say $\text{Events}^t = \text{Reads}^t \cup \text{Writes}^t \cup \text{Acquires}^t \cup \text{Releases}^t$. Finally, given a set of events $X \subseteq \text{Events}$, we say $\text{Reads}_X = \text{Reads} \cap X$, $\text{Writes}_X = \text{Writes} \cap X$, $\text{Accesses}_X = \text{Accesses} \cap X$, $\text{Acquires}_X = \text{Acquires} \cap X$ and $\text{Releases}_X = \text{Releases} \cap X$. We also write Threads_X , Vars_X and Locks_X to denote the set of threads, variables and locks that appear in the events in X . To maintain consistency with the notation used in the rest of the document, when $X = \text{Events}_\sigma$ for some trace σ , we will use the subscript σ instead of Events_σ . For example, we will use Writes_σ^t instead of $\text{Writes}_{\text{Events}_\sigma}^t$, etc.,.

We will often be referring to events with the symbols used in their operations. For example, we will often denote acquire events by **acq**, release events by **rel**, etc.,.

In this chapter we reason about reorderings of critical sections and access events. Much of the reasoning about locks is similar to those about variables and often the treatment of read events is similar to those of release events, and those of write events is similar to acquire events. Therefore, we will unify these events by defining $\text{RdRels}_X = \text{Reads}_X \cup \text{Releases}_X$ and $\text{WtAcqs}_X = \text{Writes}_X \cup \text{Acquires}_X$.

Conflicting events. We also extend the notion of conflicting events to lock events as follows. Given two distinct events $e_1, e_2 \in \text{Events}$, we say that e_1 and e_2 are *conflicting*, denoted by $e_1 \asymp e_2$, if (i) $\text{loc}(e_1) = \text{loc}(e_2)$ (i.e., both events access the same shared variable or the same lock) and (ii) $\{e_1, e_2\} \cap \text{Writes} \neq \emptyset$ or $\{e_1, e_2\} \cap \text{Acquires} \neq \emptyset$ i.e., at least one of them is either a write event (when $\text{loc}(e_1), \text{loc}(e_2) \in \text{Vars}$) or a lock-acquire event (when $\text{loc}(e_1), \text{loc}(e_2) \in \text{Locks}$). We extend the notion of conflict to sets of events in the natural way: two sets of events $X_1, X_2 \subseteq \text{Events}$ are called *conflicting*, denoted by $X_1 \asymp X_2$ if $\exists e_1 \in X_1, e_2 \in X_2$ such that $e_1 \asymp e_2$.

The *lock-nesting depth* of a trace σ is the quantity $\max_{e \in \text{Events}_\sigma} |\text{locksHeld}_\sigma(e)|$, i.e., the maximum number of locks acquired in a nesting fashion in the trace.

Reads-from functions. The unified treatment of lock events and access events warrants that we extend the last-write function on traces lastWrite to a more generic *reads-from function*. For a trace σ , the reads-from function is $\text{RF}_\sigma : \text{RdRels}_\sigma \rightarrow \text{WtAcqs}_\sigma$ as follows: $\text{RF}_\sigma(e_1) = e_2$ iff $e_2 <_{\text{tr}}^\sigma e_1$ and $\forall e'_2 \in \text{WtAcqs}_\sigma$ with $e_2 \succ e'_2$, we have $e'_2 <_{\text{tr}}^\sigma e_1 \Rightarrow e'_2 <_{\text{tr}}^\sigma e_2$. That is, RF_σ maps every read event e to $\text{lastWrite}_\sigma(e)$ and every release event e to $\text{match}_\sigma(e)$. For simplicity, we assume that σ starts with a write event to every location, hence RF_σ is well-defined. We can now define the relation $\leq_{\text{TRF}}$ as follows.

Definition 9.1 (Thread-Order Reads-From Relation). Let σ be a trace. The relation $\leq_{\text{TRF}}^\sigma$ is the smallest partial order such that

- $\leq_{\text{TO}}^\sigma \subseteq \leq_{\text{TRF}}^\sigma$, and
- for every $e_1, e_2 \in \text{Events}_\sigma$, if $e_1 = \text{RF}_\sigma(e_2)$, then $(e_1, e_2) \in \leq_{\text{TRF}}^\sigma$.

The strict version of $\leq_{\text{TRF}}^\sigma$ is the order $<_{\text{TRF}}^\sigma = \leq_{\text{TRF}}^\sigma \setminus \{(e, e) \mid e \in \text{Events}_\sigma\}$.

We write $e \leq_{\text{TRF}}^\sigma e'$ (resp. $e <_{\text{TRF}}^\sigma e'$) for $(e, e') \in \leq_{\text{TRF}}^\sigma$ (resp. $(e, e') \in <_{\text{TRF}}^\sigma$)

The lock-dependence factor. The *lock-dependence graph* of a trace σ is the graph $G_\sigma^{\text{LD}} = (V_\sigma^{\text{LD}}, E_\sigma^{\text{LD}})$ defined as follows.

- The set of vertices V_σ^{LD} is the set of acquire events Acquires_σ .
- The set of edges is such that $(e_1, e_2) \in E_\sigma^{\text{LD}}$ if (i) $e_1 \not\leq_{\text{TRF}}^\sigma e_2$, (ii) $e_1 <_{\text{TRF}}^\sigma \text{match}_\sigma(e_2)$, and (iii) $\text{match}_\sigma(e_1) \not\leq_{\text{TRF}}^\sigma \text{match}_\sigma(e_2)$.

Given an acquire event $\text{acq} \in V_\sigma^{\text{LD}}$, let A_σ be the set of lock-acquire events that can reach acq in G_σ^{LD} . We define the *lock dependence factor* of σ as $\max_{\text{acq} \in V_\sigma^{\text{LD}}} |A_{\text{acq}}|$.

Communication topology. The trace σ naturally induces a *communication topology* graph $G_\sigma = (V_\sigma, E_\sigma)$ where the vertices are $V_\sigma = \text{Threads}_\sigma$ and the edges are $E_\sigma = \{(t, t') \mid t \neq t' \text{ and } \text{Events}_\sigma^t \succ \text{Events}_\sigma^{t'}\}$. In words, we have one node in G_σ per thread, and there is an edge between two distinct nodes if the corresponding threads execute conflicting events (note that G_σ is undirected). For simplicity, we assume that G_σ is connected. In later sections, we will make a distinction between tree topologies (i.e., topologies that do not contain cycles) and general topologies (that might contain cycles). Common examples of tree topologies include stars (e.g., server-clients), pipelines, divide-and-conquer parallelism, and the special case of two threads.

9.1.1 Problem Statement

In the dynamic data-race prediction problem, we are given an observed trace σ , and the task is to identify whether σ has a predictable data race:

Problem 9.1 (Dynamic Data-Race Detection). Given a trace σ , decide whether σ has a predictable data race.

In this work we primarily focus on the following decision problem:

Problem 9.2 (Dynamic Data-Race Detection Given Pair). Given a trace σ and two access events $e_1, e_2 \in \text{Accesses}_\sigma$ such that $e_1 \succ e_2$, the task is to decide whether (e_1, e_2) is a predictable data race of σ .

Clearly, having established the complexity of Problem 9.2, the more general Problem 9.1 can be solved by answering the decision problem for all $O(\mathcal{N}^2)$ pairs of conflicting access events of σ , where $\mathcal{N}$ is the number of events in σ . In the other direction, as the following lemma observes, detecting whether σ has some predictable data race is no easier (upto a constant factor) than detecting whether a given event pair of σ constitutes a predictable data race.

Lemma 9.1. Given a trace σ of length $\mathcal{N}$ and two events $e_1, e_2 \in \text{Events}_\sigma$, we can construct a trace σ' in $O(\mathcal{N})$ time so that σ' has a predictable data race iff (e_1, e_2) is a predictable data race of σ .

Proof (Sketch). We outline the construction of σ' . We introduce two locks ℓ_1, ℓ_2 , and for each $i \in \{1, 2\}$, we surround e_i with the lock ℓ_i , i.e., we replace e_i with $\text{acq}(\ell_i), e_i, \text{rel}(\ell_i)$. For every other event $e \in \text{Events}_\sigma \cap \text{Accesses}_\sigma \setminus \{e_1, e_2\}$ with $\text{thr}(e) = t$, we replace e with the sequence $\langle t, \text{acq}(\ell_1) \rangle \cdot \langle t, \text{acq}(\ell_2) \rangle \cdot e \cdot \langle t, \text{rel}(\ell_2) \rangle \cdot \langle t, \text{rel}(\ell_1) \rangle$. Observe that the resulting sequence σ' is a valid trace. It is easy to see that (e_1, e_2) can be the only predictable data race of σ' , and any correct reordering of σ that witnesses the data race (e_1, e_2) can be transformed to a correct reordering of σ' that witnesses the same data race, and vice versa, following the same transformation as above. The desired result follows. QED.

To make the presentation simpler, we assume without loss of generality that there are no open critical sections in σ , i.e., every lock-acquire event acq is followed by a matching lock-release event $\text{match}_\sigma(\text{acq})$. Motivated by practical applications, we also study the complexity

of dynamic data-race prediction parameterized by a notion of distance between the input trace σ and the witness σ^* that reveals the data race.

Trace distances. Consider a trace σ and a correct reordering ρ of σ . The set of *reversals* between σ and ρ is defined as

$$\text{Reversals}(\sigma, \rho) = \{(e_1, e_2) \mid e_1, e_2 \in \text{WtAcqs}_\rho, e_1 \succ e_2, e_1 <_{\text{tr}}^\sigma e_2 \text{ and } e_2 <_{\text{tr}}^\rho e_1\} \quad (9.1)$$

In words, $\text{Reversals}(\sigma, \rho)$ contains the pairs of conflicting write events or lock-acquire events, the order of which has been *reversed* in ρ when compared to σ . We remark that this notion of reversals is in fact close to the notion of synchronization preserving correct reorderings from Chapter 8, albeit is not the same. A sync-preserving correct reordering cannot reverse the ordering of conflicting release events, but is allowed to change the order of write events arbitrarily.

The *distance* of ρ from σ is defined as $\delta(\sigma, \rho) = |\text{Reversals}(\sigma, \rho)|$. Our notion of distance, thus, only counts the number of reversals of conflicting write or lock-acquire events instead of counting reversals over all events (or even conflicting write-read events).

In this chapter, we study a *promise problem* relating to dynamic data race prediction. The promise problem ([109]) given languages L_{True} and L_{False} is to design an algorithm A such that $A(x) = \text{True}$ for every $x \in L_{\text{True}}$, $A(x) = \text{False}$ for every $x \in L_{\text{False}}$, and all for all other inputs $x \notin L_{\text{True}} \cup L_{\text{False}}$, the output $A(x)$ of the algorithm is allowed to be any of **True** or **False**.

Distance-bounded dynamic data race prediction. Consider a trace σ and two events e_1, e_2 of σ . Given an integer $b \geq 0$, the b -distance-bounded dynamic data-race prediction problem is the promise problem that allows for any answer (**True/False**) if (e_1, e_2) is a predictable data race of σ and every witness correct reordering σ^* is such that $\delta(\sigma, \sigma^*) > b$. That is, it is the promise problem with (a) L_{True} being the set of all inputs (σ, e_1, e_2) such that there is a correct reordering σ^* of σ that witnesses (e_1, e_2) as a predictable race, and $\delta(\sigma, \sigma^*) > b$, and (b) L_{False} being the set of all inputs (σ, e_1, e_2) being the set of all inputs such that (e_1, e_2) is not a predictable race of σ .

9.1.2 Summary of Main Results

Here we state the main results of this work, and present the technical details later in the chapter.

9.1.3 The General Case

First, we study the complexity of the problem with respect to various parameters of the input trace. These parameters are the number of threads, the number of variables, the lock-nesting depth, as well as the lock-dependence factor, which, intuitively, measures the amount of data flow between critical sections. We show the following theorem.

Theorem 9.1. Consider a trace σ with $\mathcal{N}$ events, $\mathcal{T}$ threads, lock-nesting depth γ , and lock-dependence factor ζ , and two events $e_1, e_2 \in \mathbf{Events}_\sigma$. The problem dynamic data-race prediction given pair for σ , e_1 and e_2 can be solved in $O(\alpha \cdot \beta)$ time, where $\alpha = \min(\mathcal{N}, \mathcal{T} \cdot \gamma \cdot \zeta)^{\mathcal{T}-2}$ and $\beta = \mathcal{T} \cdot \mathcal{N}^{\mathcal{T}}$.

In particular, the problem is polynomial-time solvable for a fixed number of threads $\mathcal{T}$. In practice, the parameters $\mathcal{T}$, γ and ζ behave as constants, and in such cases our upper-bound becomes $O(\mathcal{N}^{\mathcal{T}})$. Theorem 9.1 naturally leads to two questions, namely (i) whether there is a polynomial-time algorithm for any $\mathcal{T}$, and (ii) if not, whether the problem is FPT with respect to the parameter $\mathcal{T}$, i.e., can be solved in $O(f(\mathcal{T}) \cdot \mathcal{N}^{O(1)})$ time, for some function f . Question (ii) is very relevant, as typically $\mathcal{T}$ is several orders of magnitude smaller than $\mathcal{N}$. We complement Theorem 9.1 with the following lower-bound, which answers both questions in negative.

Theorem 9.2. The dynamic data-race prediction problem is W[1]-hard when parameterized by the number of threads.

9.1.4 Tree Communication Topologies

Next, we study the problem for tree communication topologies, such as pipelines and server-clients architectures. We show the following theorem.

Theorem 9.3. Let σ be a trace over a tree communication topology with $\mathcal{N}$ events, $\mathcal{T}$ threads, $\mathcal{V}$ variables and $\mathcal{L}$ locks, and let $e_1, e_2 \in \mathbf{Events}_\sigma$. The problem of dynamic data-race prediction given pair for σ , e_1 and e_2 can be solved in $O(\mathcal{T}^2 \cdot (\mathcal{V} + \mathcal{L}) \cdot \mathcal{N}^2 \cdot \log \mathcal{N})$ time.

Perhaps surprisingly, in sharp contrast to Theorem 9.2, for tree topologies there exists an efficient algorithm where the degree of the polynomial is fixed and does not depend on any input parameter (e.g., number of threads). Note that the dominating factor in this complexity is $\mathcal{N}^2$, while $\mathcal{T}$, $\mathcal{V}$ and $\mathcal{L}$ are typically much smaller. Hence, the relevant

theoretical question is whether the dependency on $\mathcal{N}$ can be improved further. We show that this is unlikely, by complementing Theorem 9.3 with the following conditional lower-bound, based on the Orthogonal Vectors conjecture [110].

Theorem 9.4. Let σ be a trace with n events, $\mathcal{T} \geq 2$ threads and $\mathcal{V} \geq 9$ shared variables with at least one lock ($\mathcal{L} \geq 1$). Also, let $e_1, e_2 \in \mathbf{Events}_\sigma$ be conflicting events in σ . There is no algorithm that solves the decision problem of dynamic data-race prediction given pair for σ, e_1 and e_2 in time $O(\mathcal{N}^{2-\epsilon})$, for any $\epsilon > 0$, unless the Orthogonal Vectors conjecture fails.

Since $\mathcal{T} = 2$ implies a tree communication topology, the result of Theorem 9.3 is conditionally optimal, up-to poly-logarithmic factors, for a reasonable number of threads and variables (e.g., when $\mathcal{T}, \mathcal{V} = \log^{O(1)}(\mathcal{N})$).

9.1.5 Witnesses in Small Distance

Finally, we study the problem with assumptions motivated from practical settings, namely, when (i) the number of threads, lock-nesting depth, lock-dependence factor of σ are bounded, and (ii) we are searching for a witness at a small distance from σ .

Theorem 9.5. Fix a reversal bound $\mathfrak{b} \geq 0$. Consider a trace σ with $\mathcal{N}$ and constant number of threads, lock-nesting depth and lock-dependence factor, i.e., $\mathcal{T} = O(1), \gamma = O(1)$ and $\zeta = O(1)$. The $\mathfrak{b}$ -distance-bounded dynamic data-race prediction problem for σ can be solved in $O(\mathcal{N})$ time.

9.2 TRACE IDEALS

In this section we introduce relevant notation on partial orders and trace ideals. These notions are a recurring theme throughout this chapter and offer a clean abstraction of the problem of data race prediction.

9.2.1 Partial Orders

Partially ordered sets. A *partially ordered set* (or *poset*) is a pair (X, P) where $X \subseteq \mathbf{Events}$ is a set of events and P is a partial order over X . As with other partial orders, we will often write $e_1 \leq_P e_2$ to denote $(e_1, e_2) \in P$. Given two events $e_1, e_2 \in X$ we write $e_1 <_P e_2$ to denote that $e_1 \leq_P e_2$ and $e_1 \neq e_2$. Further, we will write $e_1 \ll_P e_2$ to denote that $e_1 <_P e_2$ and there exists no event e such that $e_1 <_P e <_P e_2$. Likewise, given two distinct events

$e_1, e_2 \in X$, we say that e_1 and e_2 are *unordered* by P , denoted by $e_1 \parallel_P e_2$, if neither $e_1 \leq_P e_2$ nor $e_2 \leq_P e_1$.

We call an event $e \in X$ *maximal* if there exists no $e' \in X$ such that $e <_P e'$. Given a set $Y \subseteq X$, we denote by $P|Y$ the *projection* of P on Y , i.e., we have $P|Y \subseteq Y \times Y$, and for all $e_1, e_2 \in Y$, we have $e_1 \leq_{P|Y} e_2$ iff $e_1 \leq_P e_2$. Given two posets (X, P) and (X, Q) , we say that the partial order Q *refines* P , denoted by $Q \sqsubseteq P$, if for every two events $e_1, e_2 \in X$, if $e_1 \leq_P e_2$ then $e_1 \leq_Q e_2$. If Q refines P , we say that P is *weaker* than Q . We denote by $Q \sqsubset P$ the fact that $Q \sqsubseteq P$ and $P \not\sqsubseteq Q$. A *linearization* of (X, P) is a total order over X that refines P . An *order ideal* (or simply *ideal*) of a poset (X, P) is subset $Y \subseteq X$ such that for every two events $e_1 \in Y$ and $e_2 \in X$ with $e_2 \leq_P e_1$, we have $e_2 \in Y$. An event e is *executable* in ideal Y if $Y \cup \{e\}$ is also an ideal of (X, P) . The number of threads and variables of a poset (X, P) is the number of threads and variables of the events of X .

Partially ordered sets with reads-from functions. A *poset with a reads-from function* (or *rf-poset*) is a tuple $\mathcal{P} = (X, P, \text{RF})$ where (i) $\text{RF} : \text{RdRels}_X \rightarrow \text{WtAcqs}_X$ is a reads-from function such that for all $e \in \text{RdRels}_X$, we have $\text{RF}(e) \in \text{Writes}_X$ iff $e \in \text{Reads}_X$, and (ii) (X, P) is a poset where for all $e \in \text{RdRels}_X$ we have $\text{RF}(e) <_P e$. Notation from posets is naturally lifted to rf-posets, e.g., an ideal of $\mathcal{P}$ is an ideal of (X, P) .

Trace ideals. A *trace ideal* of σ is an ideal X of the poset $(\text{Events}_\sigma, \leq_{\text{TRF}}^\sigma)$. We say an event $e \in \text{Events}_\sigma \setminus X$ is *enabled* in X if for every $e' <_{\text{TO}}^\sigma e$, we have $e' \in X$. We call X *lock-feasible* if for every two lock-acquire events $\text{acq}_1, \text{acq}_2 \in \text{Acquires}_X$ with $\text{acq}_1 \asymp \text{acq}_2$, we have $\text{match}_\sigma(\text{acq}_i) \in X$ for some $i \in [2]$. We call X *feasible* if it is lock-feasible, and there exists a partial order P over X such that (i) $\leq_P \sqsubseteq \leq_{\text{TRF}}^{\sigma|X}$ and (ii) for every pair of lock-acquire events $\text{acq}_1, \text{acq}_2 \in \text{Acquires}_X$ with $\text{acq}_1 \asymp \text{acq}_2$, and $\text{match}_\sigma(\text{acq}_1) \notin X$, we have $\text{rel}_2 <_P \text{acq}_1$, where $\text{rel}_2 = \text{match}_\sigma(\text{acq}_2)$. If X is feasible, we define the *canonical rf-poset* of X as $(X, Q, \text{RF}_\sigma|X)$, where Q is the weakest among all such partial orders P . It is easy to see that Q is well-defined, i.e., there exists at most one weakest partial order among all such partial orders P .

Let us now define what it means for a trace ideal to be *realizable*.

Problem 9.3 (Realizability of feasible trace ideals). The realizability problem for an rf-poset $\mathcal{P} = (X, P, \text{RF})$ asks whether there exists a linearization σ^* of P such that $\text{RF}_{\sigma^*} = \text{RF}$. Given a trace σ and a feasible trace ideal X of σ , the realizability problem for X is the realizability problem of the canonical rf-poset (X, P, RF) of X . The following remark relates the decision problem of dynamic race prediction in σ with the realizability of trace ideals of σ .

Remark 9.1. If σ^* is a witness of the realizability of X , then σ^* is a correct reordering of σ . Two conflicting events $e_1, e_2 \in \text{Events}_\sigma$ are a predictable data race of σ iff there exists a realizable trace ideal X of σ such that e_1, e_2 are enabled in X .

Read pairs and triplets. For notational convenience, we introduce the notion of read pairs and read triplets. Given an rf-poset $\mathcal{P} = (X, P, \text{RF})$, a *read pair* (or *pair* for short) of $\mathcal{P}$ is a pair $(\mathbf{w}, \mathbf{r})$ such that $\mathbf{r} \in \text{RdRels}_X$ and $\mathbf{w} = \text{RF}(\mathbf{r})$ (note that $\mathbf{w} \in X$). A *read triplet* (or *triplet* for short) is a triplet $(\mathbf{w}, \mathbf{r}, \mathbf{w}')$ such that (i) $(\mathbf{w}, \mathbf{r})$ is a pair of $\mathcal{P}$, (ii) $\mathbf{w}' \in X$, and (iii) $\mathbf{w}' \neq \mathbf{w}$ and $\mathbf{r} \prec \mathbf{w}'$. We denote by $\text{Pairs}(\mathcal{P})$ and $\text{Triplets}(\mathcal{P})$ the set of pairs and triplets of $\mathcal{P}$, respectively.

Closed rf-posets. We call an rf-poset $\mathcal{P} = (X, P, \text{RF})$ *closed* if for every triplet $(\mathbf{w}, \mathbf{r}, \mathbf{w}') \in \text{Triplets}(\mathcal{P})$, we have (i) if $\mathbf{w}' <_{\mathcal{P}} \mathbf{r}$ then $\mathbf{w}' <_{\mathcal{P}} \mathbf{w}$, and (ii) if $\mathbf{w} <_{\mathcal{P}} \mathbf{w}'$ then $\mathbf{r} <_{\mathcal{P}} \mathbf{w}'$. Given, an rf-poset $\mathcal{P} = (X, P, \text{RF})$, the *closure* of $\mathcal{P}$ is an rf-poset $\mathcal{Q} = (X, Q, \text{RF})$ where Q is the weakest partial order over X such that $Q \sqsubseteq P$ and $\mathcal{Q}$ is closed. If no such Q exists, we let the closure of $\mathcal{P}$ be *undefined*. The closure is well-defined [35]. The associated Closure problem is, given an rf-poset $\mathcal{P}$, decide whether the closure of $\mathcal{P}$ is not *undefined*.

Remark 9.2. An rf-poset is realizable only if its closure exists and is realizable.

9.2.2 Bounds on the Number of Feasible Trace Ideals

Remark 9.1 suggests that the dynamic data-race prediction problem for a trace σ is reducible to deciding whether σ has some realizable trace ideal. In general, if σ has length $\mathcal{N}$ and $\mathcal{T}$ threads, there exist $\mathcal{N}^{\mathcal{T}}$ possible trace ideals to test for realizability. Here we derive another upper-bound on the number of such ideals that are sufficient to test, based on the number of threads of σ , its lock-nesting depth and its lock-dependence factor. These parameters typically behave as constants in practice, and thus understanding the complexity of dynamic data race prediction in terms of these parameters is crucial.

Causal cones. Given an event $e \in \text{Events}_\sigma$, the *causal cone* $\text{Cone}_\sigma(e)$ of $e \in \text{Events}_\sigma$ is the smallest trace ideal X of σ so that e is enabled in X . In words, we construct $\text{Cone}_\sigma(e)$ by taking the $\leq_{\text{TRF}}^\sigma$ -downwards closure of the thread-local predecessor e' of e (i.e., $e' \ll_{\text{TO}}^\sigma e$). Given a non-empty set of events $S \subseteq \text{Events}_\sigma$, we define the causal cone of S as $\text{Cone}_\sigma(S) = \bigcup_{e \in S} \text{Cone}_\sigma(e)$; notice that $\text{Cone}_\sigma(S)$ is a trace ideal.

Candidate ideal set. Given a set of events X , we denote by $\text{OpenAcqs}(X)$ the set of acquire events acq such that $\text{match}_\sigma(\text{acq}) \notin X$. Given two events $e_1, e_2 \in \text{Events}_\sigma$, the

candidate ideal set $\text{CIS}_\sigma(e_1, e_2)$ of e_1, e_2 is the smallest set of trace ideals of σ such that the following hold.

1. $\text{Cone}_\sigma(\{e_1, e_2\}) \in \text{CIS}_\sigma(e_1, e_2)$.
2. Let $Y \in \text{CIS}_\sigma(e_1, e_2)$, $\text{acq} \in \text{OpenAcqs}(Y)$, $\text{rel} = \text{match}_\sigma(\text{acq})$, and $Y' = \text{Cone}_\sigma(Y \cup \{\text{rel}\}) \cup \{\text{rel}\}$. If $e_1, e_2 \notin Y'$, then $Y' \in \text{CIS}_\sigma(e_1, e_2)$.

In light of Remark 9.1, we will decide whether (e_1, e_2) is a predictable data race by deciding the realizability of ideals in the candidate ideal set. Item 2 states that, as long as there is some ideal Y in the candidate ideal set such that Y leaves some critical section open, we construct another ideal $Y' \supset Y$ by choosing one such open critical section and closing it, and add Y' in the candidate set as well. Intuitively, the open critical section of Y might deem Y not realizable, while closing that critical section might make Y' realizable. Clearly, if $e_1 \in Y'$ or $e_2 \in Y'$, then the realizability of Y' does not imply a data race on e_1, e_2 as one of the two events is not enabled in Y' (Remark 9.1). As the following lemma shows, in order to decide whether (e_1, e_2) is a predictable data race of σ , it suffices to test for realizability all the ideals in $\text{CIS}_\sigma(e_1, e_2)$.

Lemma 9.2. (e_1, e_2) is a predictable data race of σ iff there exists a realizable ideal $X \in \text{CIS}_\sigma(e_1, e_2)$ such that $e_1, e_2 \notin X$.

Proof. The $(\Leftarrow)$ direction of the statement is straightforward, and here we focus on the $(\Rightarrow)$ direction. Let σ^* be a correct reordering that witnesses (e_1, e_2) , and $X^* = \text{Events}_{\sigma^*}$. We show that there exists an ideal $X \in \text{CIS}_\sigma(e_1, e_2)$ such that (i) $X \subseteq X^*$, and (ii) $\text{OpenAcqs}(X) \subseteq \text{OpenAcqs}(X^*)$. Observe that the two conditions imply the lemma: (i) since $X \subseteq X^*$, we have that $e_1, e_2 \notin X$, while both events are enabled in X , and (ii) since $X \subseteq X^*$ and $\text{OpenAcqs}(X) \subseteq \text{OpenAcqs}(X^*)$, we have that $\sigma^*|X$ is a correct reordering of σ , and hence X is realizable.

Consider any ideal $Y \in \text{CIS}_\sigma(e_1, e_2)$ such that $Y \subseteq X^*$. Clearly, at least one such Y exists, by taking $Y = \text{Cone}_\sigma(\{e_1, e_2\})$ and noticing that $Y \subseteq X^*$. If $\text{OpenAcqs}(Y) \not\subseteq \text{OpenAcqs}(X^*)$, there exists some lock-acquire event $\text{acq} \in \text{OpenAcqs}(Y)$ such that $\text{rel} \in X^*$, where $\text{rel} = \text{match}_\sigma(\text{acq})$. But then $Y' \in \text{CIS}_\sigma(e_1, e_2)$, where $Y' = Y \cup \text{Cone}_\sigma(\{\text{rel}\}) \cup \{\text{rel}\}$. Note that $Y' \subseteq X^*$, and repeat the process for $Y = Y'$. Since $Y \subset Y'$, this process can be repeated at most n times, thus at some point we have chosen an ideal $Y \in \text{CIS}_\sigma(e_1, e_2)$ with the desired properties of X .

The desired result follows.

QED.

The following lemma gives an upper-bound on $|\text{CIS}_\sigma(e_1, e_2)|$, i.e., on the number of ideals we need to test for realizability.

Lemma 9.3. We have $|\text{CIS}_\sigma(e_1, e_2)| \leq \min(\mathcal{N}, \alpha)^{\mathcal{T}-2}$, where $\alpha = \mathcal{T} \cdot \gamma \cdot \zeta$, and $\mathcal{T}$ is the number of threads, γ is the lock-nesting depth, and ζ is the lock-dependence factor of σ .

Proof. Let $Z = \text{CIS}_\sigma(e_1, e_2)$ and $X = \text{Cone}_\sigma(\{e_1, e_2\})$. For an ideal $Y \in Z \setminus \{X\}$, let rel_Y be the lock-release event that lead to $Y \in Z$ according to Item 2 of the definition of $\text{CIS}_\sigma(e_1, e_2)$, and $\text{acq}_Y = \text{match}_\sigma(\text{rel}_Y)$. In addition, we call the ideal $Y' \in Z$ with $\text{acq}_Y \in \text{OpenAcqs}(Y')$ that lead to adding $Y \in Z$ the parent of Y . We define inductively $A_X = \emptyset$, and $A_Y = A_{Y'} \cup \{\text{acq}_Y\}$, where Y' is the parent of Y . Note that every ideal $Y \in Z$ is uniquely characterized by A_Y .

Let G_σ^{LD} be the lock-dependence graph of σ . We show by induction that for every $Y \in Z \setminus \{X\}$ there exists a lock-acquire event $\text{acq} \in \text{OpenAcqs}(X)$ such that acq is reachable from acq_Y in G_σ^{LD} . Let Y_1 be the smallest (with respect to set inclusion) ancestor of Y such that $\text{acq}_Y \in \text{OpenAcqs}(Y_1)$. The statement holds if $Y_1 = X$, by taking $\text{acq} = \text{acq}_Y$. Otherwise, let Y_2 the parent of Y_1 , and we have $\text{acq}_{Y_1} \in \text{OpenAcqs}(Y_2)$. Note that (i) $\text{acq}_Y \not\prec_{\text{TRF}} \text{acq}_{Y_1}$ (since $\text{acq}_Y \notin Y_2$), (ii) $\text{acq}_Y <_{\text{TRF}} \text{rel}_{Y_1}$ (since $\text{acq}_Y \in \text{Cone}_\sigma(\text{rel}_{Y_1})$), and (iii) $\text{rel}_Y \not\prec_{\text{TRF}} \text{rel}_{Y_1}$ (since $\text{rel}_Y \notin Y_1$). It follows that $(\text{acq}_Y, \text{acq}_{Y_1})$ is an edge in G_σ^{LD} . By the induction hypothesis, we have that there exists some lock-acquire event $\text{acq} \in \text{OpenAcqs}(X)$ that is reachable from acq_{Y_1} in G_σ^{LD} . Hence acq is reachable by acq_Y in G_σ^{LD} , as desired.

Now, let $A = \bigcup_{Y \in Z} A_Y$, and by the previous paragraph, for every lock-acquire event $\text{acq}' \in A$, there exists a lock-acquire event $\text{acq} \in \text{OpenAcqs}(X)$ that is reachable from acq' in G_σ^{LD} . Hence, $|A| \leq |\text{OpenAcqs}(X)| \cdot \zeta$. In addition, since there are k threads and the lock-nesting depth is γ , we have $|\text{OpenAcqs}(X)| \leq k \cdot \gamma$, thus $|A| \leq k \cdot \gamma \cdot \zeta = \alpha$. Moreover, we trivially have $|A| \leq n$.

Finally, since (i) we have k threads, and (ii) in each ideal $Y \in Z$ the events e_1 and e_2 are enabled, we have $|Z| \leq |A|^{k-2} \leq \min(n, \alpha)^{k-2}$ such ideals.

The desired result follows. QED.

9.3 THE GENERAL CASE

In this section we address the general case of dynamic data-race prediction. The section is organized in two parts, which present the formal details of Theorem 9.1 and Theorem 9.2.

9.3.1 Upper Bound

In this section we establish Theorem 9.1. Recall that, by Lemma 9.2, the problem is reducible to detecting a realizable rf-poset in the candidate ideal set of the two events that are tested for a data-race. Rf-poset realizability is known to be NP-complete [101], and solvable in polynomial time when the number of threads is bounded [111]. Here we establish more precise upper-bounds, based on the number of threads. In particular, we show the following.

Lemma 9.4. Rf-poset realizability can be solved in $O(\mathcal{T} \cdot \mathcal{N}^{\mathcal{T}})$ time for an rf-poset of size $\mathcal{N}$ and $\mathcal{T}$ threads.

Frontiers and extensions. Let $\mathcal{P} = (X, P, \text{RF})$ be an rf-poset, and consider an ideal Y of $\mathcal{P}$. The *frontier* of Y , denoted $\text{Frontier}_{\mathcal{P}}(Y)$, is the set of pairs $(\mathbf{w}, \mathbf{r}) \in \text{Pairs}(\mathcal{P})$ such that $\mathbf{w} \in Y$ and $\mathbf{r} \notin Y$. An event e executable in Y is said to *extend* Y if for every triplet $(\mathbf{w}, \mathbf{r}, e) \in \text{Triplets}(\mathcal{P})$, we have $(\mathbf{w}, \mathbf{r}) \notin \text{Frontier}_{\mathcal{P}}(Y)$. In this case, we say that $Y \cup \{e\}$ is an *extension* of Y via e .

Ideal graphs and canonical traces. Let $\mathcal{P} = (X, P, \text{RF})$ be an rf-poset. The *ideal graph* of $\mathcal{P}$, denoted $G_{\mathcal{P}} = (V_{\mathcal{P}}, E_{\mathcal{P}})$ is a directed graph defined as follows.

1. $V_{\mathcal{P}}$ is the set of ideals of $\mathcal{P}$.
2. We have $(Y_1, Y_2) \in E_{\mathcal{P}}$ iff Y_2 is an extension of Y_1 .

The *ideal tree* of $\mathcal{P}$, denoted $T_{\mathcal{P}} = (\mathcal{I}_{\mathcal{P}}, \mathcal{R}_{\mathcal{P}})$ is a (arbitrary) spanning tree of $G_{\mathcal{P}}$ when restricted to nodes reachable from $\emptyset$. We let $\emptyset$ be the root of $T_{\mathcal{P}}$. Given an ideal $Y \in \mathcal{I}_{\mathcal{P}}$, we define the *canonical trace* σ_Y of Y inductively, as follows. If $Y = \emptyset$ then $\sigma_Y = \epsilon$. Otherwise, Y has a parent Y' in $T_{\mathcal{P}}$ such that $Y = Y' \cup \{e\}$ for some event $e \in X$. We define $\sigma_Y = \sigma_{Y'} \cdot e$. Lemma 9.4 relies on the following lemmas (proofs presented in [112])

Lemma 9.5. We have $X \in \mathcal{I}_{\mathcal{P}}$ iff $\mathcal{P}$ is realizable.

Proof. We prove each direction separately.

($\Rightarrow$). We prove by induction that every canonical trace σ_Y of $T_{\mathcal{P}}$ realizes Y . The statement clearly holds if $Y = \epsilon$. Otherwise, let $\sigma_Y = \sigma_{Y'} \circ e$, i.e., Y is the extension of Y' by the event e , and by the induction hypothesis we have that $\sigma_{Y'}$ realizes Y' . The statement clearly holds if $e \in \text{WtAcqs}_{\sigma}$, so we focus on the case where $e \in \text{RdRels}_{\sigma}$. Consider the pair $(\mathbf{w}, e) \in \text{Pairs}(X)$, and note that $\mathbf{w} <_{\text{TRF}} e$ and thus $\mathbf{w} \in Y'$. It remains to argue that for

every triplet $(\mathbf{w}, e, \mathbf{w}') \in \text{Triplets}(\mathcal{P})$, if $\mathbf{w}' \in Y'$ then $\mathbf{w}' <_{\sigma_{Y'}} \mathbf{w}$. Assume towards contradiction otherwise, thus there exist two ancestors $Y_1, Y_2 \in V_{\mathcal{P}}^T$ of Y' such that (i) $Y_2 = Y_1 \cup \{\mathbf{w}'\}$ and (ii) $\mathbf{w} \in Y_1$. In that case we have $(\mathbf{w}, \mathbf{r}) \in \text{Frontier}_{\mathcal{P}}(Y_1)$, hence $\mathbf{w}'$ could not have been executable in Y_1 , a contradiction.

($\Leftarrow$). Let σ^* be a witness trace that realizes $\mathcal{P}$. We argue that for every prefix σ' of σ^* , there exists a node $X' \in \mathcal{I}_{\mathcal{P}}$ such that $X' = \text{Events}_{\sigma'}$. The proof is by induction on the prefixes σ' . The statement clearly holds if $\sigma' = \epsilon$, by taking X' to be the root of $T_{\mathcal{P}}$. Otherwise, let $\sigma' = \sigma'', e$, for some event $e \in X$, and by the induction hypothesis, there exists $X'' \in \mathcal{I}_{\mathcal{P}}$ such that $X'' = \text{Events}_{\sigma''}$. It suffices to argue that e extends X'' . The statement is trivial if $e \in \text{RdRels}_X$, hence we focus on the case where $e \in \text{WtAcqs}_X$. It suffices to argue that for every pair $(\mathbf{w}, \mathbf{r}) \in \text{Frontier}_{\mathcal{P}}(X'')$ we have $(\mathbf{w}, \mathbf{r}, e) \notin \text{Triplets}(\mathcal{P})$. Consider any such triplet, and since σ^* is a witness of the realizability of X , we have $\mathbf{r} \in \text{Events}_{\sigma^*}$. By the induction hypothesis, we have $\mathbf{w} \in \text{Events}_{\sigma''}$ and $\mathbf{r} \notin \text{Events}_{\sigma''}$. But then $\text{RF}_{\sigma^*}(\mathbf{r}) \neq \mathbf{w}$, a contradiction.

The desired result follows. QED.

Lemma 9.6. The ideal graph $G_{\mathcal{P}}$ has $O(\mathcal{N}^{\mathcal{T}})$ nodes, where $\mathcal{N}$ and $\mathcal{T}$ are respectively the number of events and threads in $\mathcal{P}$.

Proof. The statement follows directly from the definition of poset ideals and the fact that the poset (X, P) has width at most $\mathcal{T}$. QED.

We are now ready to prove Lemma 9.4.

Proof of Lemma 9.4. Consider an rf-poset $\mathcal{P}$ of size $\mathcal{N}$ and $\mathcal{T}$ threads. By Lemma 9.5, to decide the realizability of $\mathcal{P}$ it suffices to construct the ideal graph $G_{\mathcal{P}}$ and test whether $X \in V_{\mathcal{P}}$. By Lemma 9.6, $G_{\mathcal{P}}$ has $O(\mathcal{N}^{\mathcal{T}})$ nodes, and since $\mathcal{P}$ has k threads, there each node of G_X has $\leq \mathcal{T}$ outgoing edges. Hence G_X can be constructed in $O(k \cdot \mathcal{N}^{\mathcal{T}}) = O(\beta)$ time.

The desired result follows. QED.

9.3.2 Hardness of Data Race Prediction

Here we establish that the problem of dynamic data-race prediction is $\text{W}[1]$ -hard when parameterized by the number of threads $\mathcal{T}$. Our proof is established in two steps. In the first step, we show the following lemma.

Lemma 9.7. Rf-poset realizability parameterized by the number of threads $\mathcal{T}$ is $\text{W}[1]$ -hard.

Rf-poset realizability is known to be **NP**-hard [101, Theorem 4.1], and Lemma 9.7 strengthens that result by showing that the problem is even unlikely to be **FPT**. In the second step,

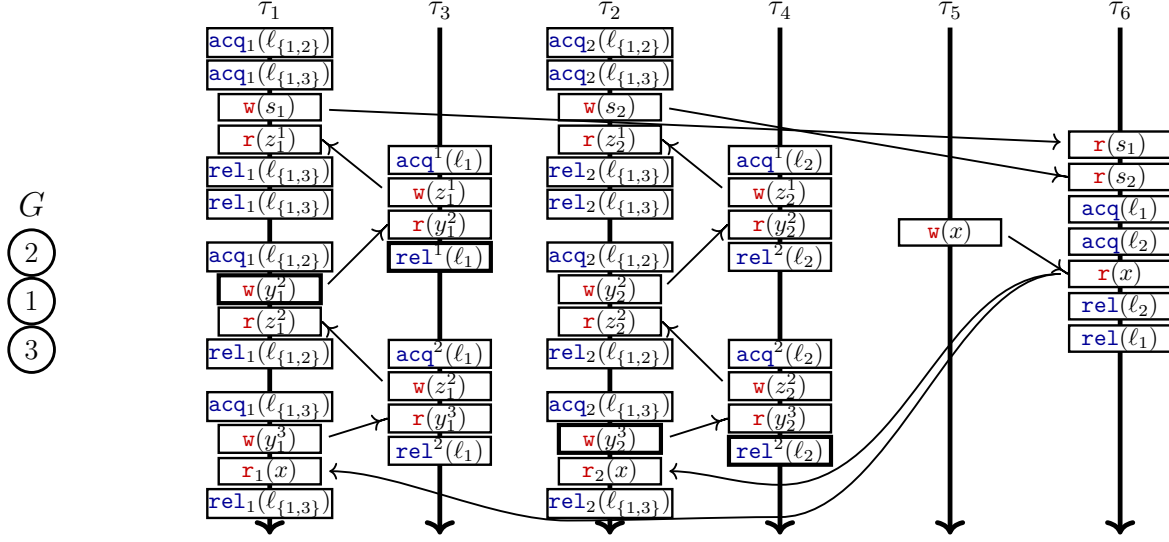


Figure 9.1: Illustration of the o-poset $\mathcal{P}_G$ given a graph G and independent-set size $c = 2$. Edges represent orderings in P .

we show how the class of $W[1]$ -hard instances of in Lemma 9.7 can be reduced to dynamic data-race prediction.

Hardness of rf-poset realizability. Our reduction is from the INDEPENDENT-SET(c), which takes as input an undirected graph $G = (V, E)$ and asks whether G has an independent set of size c . INDEPENDENT-SET(c) parameterized by c is one of the canonical $W[1]$ -hard problems [113].

Given an input $G = (V, E)$ of INDEPENDENT-SET(c) with $n = |V|$, we construct an rf-poset $\mathcal{P}_G = (X, P, \text{RF})$ of size $O(c \cdot n)$ and $O(c)$ threads such that $\mathcal{P}_G$ is realizable iff G has an independent set of size c . We assume wlog that every node in G has at least one neighbor, otherwise, we can remove all s such nodes and solve the problem for parameter $c' = c - s$. The rf-poset $\mathcal{P}_G$ consists of $k = 2 \cdot c + 2$ total orders (X_i, τ_i) . Figure 9.1 provides an illustration. In high level, for each $i \in [c]$, τ_i and τ_{c+i} are used to encode the i -th copy of G , whereas the last two total orders are auxiliary. Superscripts on the events and/or their variables refer to the node of G that is encoded by those events. Below we describe the events and certain orderings between them. The partial order P is the transitive closure of these orderings.

1. For $i = 2 \cdot c + 1$, τ_i consists of a single event $\tau_i = w(x)$.

2. For $i = 2 \cdot c + 2$, we have $\tau_i = \sigma \cdot \mathfrak{o}$, where

$$\sigma = \mathbf{r}(s_1), \dots, \mathbf{r}(s_c), \mathbf{acq}(\ell_1), \dots, \mathbf{acq}(\ell_c) \quad \text{and} \quad (9.2)$$

$$\mathfrak{o} = \mathbf{r}(x), \mathbf{rel}(\ell_c), \dots, \mathbf{rel}(\ell_1). \quad (9.3)$$

3. For each $i \in [c]$, we have $\tau_i = \tau_i^1 \cdot \tau_i^2 \cdot \dots \cdot \tau_i^n$, where each τ_i^j encodes node j of G and is defined as follows. Let $\bar{\tau}_i^j = \sigma_i^j \cdot \mathfrak{o}_i^j$, where

$$\sigma_i^j = \mathbf{acq}_i(\ell_{\{j, l_1\}}), \dots, \mathbf{acq}_i(\ell_{\{j, l_m\}}) \quad \text{and} \quad (9.4)$$

$$\mathfrak{o}_i^j = \mathbf{rel}_i(\ell_{\{j, l_m\}}), \dots, \mathbf{rel}_i(\ell_{\{j, l_1\}}) \quad (9.5)$$

where $l_1, \dots, l_m$ are the neighbors of j in G . For each $j \in [n] \setminus \{1, n\}$, the sequence τ_i^j is identical to $\bar{\tau}_i^j$, with the addition that the innermost critical section (i.e., between $\mathbf{acq}_i(\ell_{\{j, l_m\}})$ and $\mathbf{rel}_i(\ell_{\{j, l_m\}})$) contains the sequence $\mathbf{w}(y_i^j), \mathbf{r}(z_i^j)$. The sequence τ_i^1 is defined similarly, except that the innermost critical section contains the sequence $\mathbf{w}(s_i), \mathbf{r}(z_i^1)$. Finally, the sequence τ_i^n is defined similarly, except that the innermost critical section contains the sequence $\mathbf{w}(y_i^n), \mathbf{r}(x)$.

4. For each $i \in [c]$, we have $\tau_{c+i} = \tau_{c+i}^1 \cdot \tau_{c+i}^2 \cdot \dots \cdot \tau_{c+i}^{n-1}$, where

$$\tau_{c+i}^j = \mathbf{acq}^j(\ell_i), \mathbf{w}(z_i^j), \mathbf{r}(y_i^{j+1}), \mathbf{rel}^j(\ell_i).$$

Note that every memory location is written exactly once, hence the reads-from function RF is defined implicitly. In addition, for every read event $\mathbf{r}$, we have $\text{RF}(\mathbf{r}) <_{\mathbf{p}} \mathbf{r}$, as well as $\mathbf{r}(x) <_{\mathbf{p}} \mathbf{r}_i(x)$ for each $i \in [c]$.

Proof of Lemma 9.7. We show that $\mathcal{P}_G$ is realizable iff G has an independent set of size c . We prove each direction separately.

($\Rightarrow$) Let σ be a trace that realizes $\mathcal{P}_G$. For each $i \in [c]$, let m_i be the maximum integer j such that the event $\mathbf{r}(z_i^j)$ has a predecessor e_i in τ_i with $e_i <_{\text{tr}}^{\sigma} \mathbf{r}(x)$. Note that for each such i we have $\mathbf{w}(s_i) <_{\mathbf{r}}^{\sigma} (x)$, and since $\mathbf{w}(s_i)$ is the predecessor of $\mathbf{r}(z_i^1)$ in τ_i , the index m_i is well-defined. We argue that $A = \{l_i\}_{i \in [c]}$ is an independent set of G , where $l_i = m_i$ if $\mathbf{r}(x) <_{\text{tr}}^{\sigma} \mathbf{r}(z_i^{m_i})$ and $l_i = m_i + 1$ otherwise. We note that in the second case, $l_i = n$, as, otherwise, we would also have $\mathbf{acq}_{l_i}(\ell_i) <_{\text{tr}}^{\sigma} \mathbf{r}(x)$, and since $\mathbf{r}(x)$ is protected by lock ℓ_i , we would also have that $\mathbf{rel}_{l_i}(\ell_i) <_{\text{tr}}^{\sigma} \mathbf{r}(x)$. But then, $\mathbf{w}(y_1^{l_i+1}) <_{\text{tr}}^{\sigma} \mathbf{r}(x)$, which would contradict our choice of m_i .

Indeed, consider any distinct $i_1, i_2 \in [c]$ and any neighbor v_1 and v_2 of m_{i_1} and m_{i_2} , respectively. Note that our choice of $A = \{l_i\}_{i \in [c]}$ implies that both $\mathbf{acq}_{l_{i_1}}(\ell_{\{l_{i_1}, v_1\}}) <_{\text{tr}}^{\sigma} \mathbf{r}(x)$

and $\text{acq}_{i_2}(\ell_{\{l_{i_2}, v_2\}}) <_{\text{tr}}^{\sigma} \mathbf{r}(x)$. It suffices to argue that both $\mathbf{r}(x) <_{\text{tr}}^{\sigma} \text{rel}_{i_1}(\ell_{\{l_{i_1}, v_1\}})$ and $\mathbf{r}(x) <_{\text{tr}}^{\sigma} \text{rel}_{i_2}(\ell_{\{l_{i_2}, v_2\}})$, as then we have that $(l_{i_1}, l_{i_2}) \notin E$, which concludes that A is an independent set of G .

Assume towards contradiction that $\text{rel}_{i_1}(\ell_{\{l_{i_1}, v_1\}}) <_{\text{tr}}^{\sigma} \mathbf{r}(x)$. Clearly $l_{i_1} < n$, as $\mathbf{r}(x) <_P \text{rel}_{i_1}(\ell_{\{n, v_1\}})$. But then, $\text{acq}^{l_{i_1}}(\ell_{i_1}) <_{\text{tr}}^{\sigma} \text{rel}_{i_1}(\ell_{\{l_{i_1}, v_1\}})$ and thus $\text{acq}^{l_{i_1}}(\ell_{i_1}) <_{\text{tr}}^{\sigma} \mathbf{r}(x)$. Since $\mathbf{r}(x)$ appears in a critical section on lock ℓ_{i_1} , we have $\text{rel}^{l_{i_1}}(\ell_1) <_{\text{tr}}^{\sigma} \mathbf{r}(x)$. But then $e_{i_1} <_{\text{tr}}^{\sigma} \mathbf{r}(x)$, where e_{i_1} is the predecessor of $\mathbf{r}(z_i^{l_{i_1}+1})$ in τ_{i_1} , which contradicts our definition of l_1 . Hence, $\mathbf{r}(x) <_{\text{tr}}^{\sigma} \text{rel}_{i_1}(\ell_{\{l_{i_1}, v_1\}}), \text{rel}_{i_2}(\ell_{\{l_{i_2}, v_2\}})$, and thus A is an independent set of G .

($\Leftarrow$) Let A be an independent set of G of size c , and $l_1, \dots, l_c$ some arbitrary ordering of A . For each $i \in [c]$ let Y_i be the following events of threads i and $c+i$. 1. All strict predecessors of e_i in τ_i , where $e_i = \mathbf{r}(z_i^{l_i})$ if $l_i < n$ and $e_i = \mathbf{r}(x)$ otherwise. 2. If $l_i > 1$, all predecessors of $\text{rel}^{l_i-1}(\ell_i)$ in τ_{c+i} (including $\text{rel}^{l_i-1}(\ell_i)$). For each $i \in [c]$, let

$$Z_i = \{e \in Y_i \mid \exists \text{acq} \in \text{OpenAcqs}(Y_i) \text{ s.t. } \text{acq} \leq_P e\}, \quad (9.6)$$

i.e., Z_i contains all events of τ_i and τ_{c+i} that succeed some lock-acquire event that is open in Y_i . We argue that for any two distinct $i_1, i_2 \in [c]$, any two lock-acquire events $\text{acq}_1 \in Z_{i_1}$ and $\text{acq}_2 \in Z_{i_2}$ access a different lock. Note that the statement follows easily if one acq_1 or acq_2 belongs to τ_{c+i} , for some $i \in [c]$, as the locks accessed by each such total ordered are only also accessed by $\tau_{2 \cdot c+2}$. Hence, we focus on the case where each of acq_1 and acq_2 belong to τ_i , for some $i \in [c]$. Assume towards contradiction otherwise, hence there exist distinct $i_1, i_2 \in [c]$ such that (i) $\text{acq}_{i_1}(\ell_{\{l_{i_1}, l_{i_2}\}}) \in Z_{i_1}$ and $\text{acq}_{i_2}(\ell_{\{l_{i_1}, l_{i_2}\}}) \in Z_{i_2}$, while (ii) $\text{rel}_{i_1}(\ell_{\{l_{i_1}, l_{i_2}\}}) \notin Z_{i_1}$ and $\text{rel}_{i_2}(\ell_{\{l_{i_1}, l_{i_2}\}}) \notin Z_{i_2}$. It follows that $(l_{i_1}, l_{i_2}) \in E$, which contradicts the fact that A is an independent set of G .

We now construct a trace σ that realizes $\mathcal{P}_G$ in five phases, where initially, we have $\sigma = \epsilon$.

- (i) *Phase 1:* For each $i \in [c]$, we linearize the partial order $P|(Y_i \setminus Z_i)$ arbitrarily, and append it to σ .
- (ii) *Phase 2:* For each $i \in [c]$, we linearize the partial order $P|Z_i$ arbitrarily, and append it to σ .
- (iii) *Phase 3:* We append to σ the sequence $\sigma_1 \cdot \mathbf{w}(x), \mathbf{r}(x) \cdot \sigma_2$, where σ_1 and σ_2 are sequences over the events of $\tau_{2 \cdot c+2}$, as follows.

$$\sigma_1 = \mathbf{r}(s_1), \dots, \mathbf{r}(s_c), \text{acq}(\ell_1), \dots, \text{acq}(\ell_c) \quad \text{and} \quad (9.7)$$

$$\sigma_2 = \text{rel}(\ell_c), \dots, \text{rel}(\ell_1) \quad (9.8)$$

- (iv) *Phase 4*: For each $i \in [c]$, we linearize the partial order $P|(S_i \setminus Y_i)$, where S_i is the smallest ideal of (X, P) that contains the matching lock-release events of all lock-acquire events that are open in Y_i
- (v) *Phase 5*: For each $i \in [c]$ we linearize P over the remaining events of τ_i and τ_{c+i} arbitrarily, and append them to σ .

Finally, we argue that σ is a valid witness trace. It is straightforward to verify that σ is a linearization of P . Moreover, since every memory location is written exactly once in X , for every read event $\mathbf{r} \in \text{Reads}_X$ we have $\text{RF}_\sigma(\mathbf{r}) = \text{RF}(\mathbf{r})$. It remains to argue that σ respects the critical sections of $\mathcal{P}_G$. Observe that the only phase in which we interleave open critical sections between total orders τ_i that access the same lock is in Phase (ii). However, as we have shown, for every two distinct $i_1, i_2 \in [c]$, any two lock-acquire events $\text{acq}_1 \in Z_{i_1}$ and $\text{acq}_2 \in Z_{i_2}$ access a different lock. It follows that σ respects the critical sections of $\mathcal{P}_G$.

The desired result follows. QED.

Hardness of dynamic data-race prediction. Finally, we turn our attention to the hardness of dynamic data-race prediction. Consider the INDEPENDENT-SET(c) problem on a graph G and the associated rf-poset $\mathcal{P}_G = (X, P, \text{RF})$ defined above. We construct a trace σ with $\text{Events}_\sigma = X$ and $\text{RF}_\sigma = \text{RF}$ such that $(\mathbf{w}(x), \mathbf{r}(x))$ is a predictable data-race of σ iff $\mathcal{P}_G$ is realizable. In particular, σ consists of $2 \cdot c + 2$ threads t_i , one for each total order τ_i of $\mathcal{P}_G$. We obtain σ as

$$\sigma = \tau_{2 \cdot c + 1} \cdot \sigma_1 \cdot \dots \cdot \sigma_c \cdot \tau_{2 \cdot c + 2} \quad (9.9)$$

where each σ_i is an appropriate interleaving of the total orders τ_i and τ_{c+i} that respects the reads-from function RF . We conclude the proof of Theorem 9.2 by observing that G has an independent set of size c iff $(\mathbf{w}(x), \mathbf{r}(x))$ is a predictable data-race of σ .

Proof of Theorem 9.2. We show that $(\mathbf{w}(x), \mathbf{r}(x))$ is a predictable data race of σ iff $\mathcal{P}_G$ is realizable. If $\mathcal{P}_G$ is realizable, then $(\mathbf{w}(x), \mathbf{r}(x))$ is a predictable data-race of σ witnessed by the witness σ^* of the realizability of $\mathcal{P}_G$, as constructed in the proof of Lemma 9.7 (direction $\Leftarrow$), restricted to the events $\{e \in X : e <_{\sigma^*} \mathbf{w}(x)\}$.

For the inverse direction, let σ^* be a correct reordering of σ that witnesses the data race $(\mathbf{w}(x), \mathbf{r}(x))$. We construct a trace σ' that realizes $\mathcal{P}_G$ as

$$\sigma' = \sigma^* \cdot \mathbf{w}(x), \mathbf{r}(x), \text{rel}(\ell_c), \dots, \text{rel}(\ell_1) \cdot \sigma_4 \cdot \sigma_5, \quad (9.10)$$

where σ_4 and σ_5 are analogous to the linearizations in phase (iv) and Phase (v), respectively, of the construction in the proof of Lemma 9.7. In particular, let σ'' be the prefix of σ' until the event $\text{rel}(\ell_1)$. (a) We construct σ_4 as follows. For each $i \in [c]$, we linearize the set $S_i \setminus \text{Events}_{\sigma''}$ and append it to σ_4 , where S_i is the smallest ideal of X that contains the matching lock-release events of all lock-acquire events that are open in σ'' . It is easy to see that each S_i contains the event $\mathbf{r}_i(x)$, hence σ' respects the orderings $\mathbf{r}(x) <_P \mathbf{r}_i(x)$. Indeed, as $\text{RF}(\mathbf{r}_i(x)) = \mathbf{w}(x)$ and $\mathbf{w}(x) \notin \sigma^*$, we have that $\mathbf{r}_i(x) \notin \sigma^*$, and hence by definition, $\mathbf{r}_i(x) \in S_i$. (b) We construct σ_5 as follows. For each $i \in [c]$ we linearize the remaining events of τ_i and τ_{c+i} arbitrarily, and append them to σ_5 . The correctness is established similarly to the proof of Lemma 9.7.

The desired result follows. QED.

Remark 9.3. It is known that $\text{INDEPENDENT-SET}(c)$ cannot be solved in $f(c) \cdot n^{o(c)}$ time under ETH [114]. As our reduction to rf-poset realizability and dynamic data-race prediction uses $\mathcal{T} = O(c)$ threads, each of these problems does not have a $f(\mathcal{T}) \cdot \mathcal{N}^{o(\mathcal{T})}$ -time algorithm under ETH.

9.4 TREE COMMUNICATION TOPOLOGIES

In this section we focus on the case where the input trace σ constitutes a tree communication topology. The section is organized in two parts, which present the formal details of Theorem 9.3 and Theorem 9.4.

9.4.1 An Efficient Algorithm for Tree Topologies

In this section we present the formal details of Theorem 9.3. Recall that the Theorem 9.1 states an $O(\alpha \cdot \beta)$ upper-bound for dynamic data-race prediction, where β is the complexity of deciding rf-poset realizability, and α is an upper-bound on the number of candidate ideals whose realizability we need to check. For tree communication topologies, we obtain Theorem 9.3: (i) we show an improved upper-bound β on the complexity of the realizability of trace ideals over tree topologies, and (ii) we show that it suffices to check the realizability of a single trace ideal (i.e., $\alpha = 1$). We start with point (i), and then proceed with (ii).

Tree-inducible rf-posets. Let (X, P) be a poset where $X \subseteq \text{Events}$. We say (X, P) is *k-tree-inducible* if X can be partitioned into k sets $\{X_i\}_{1 \leq i \leq k}$, for some $k \in \mathbb{N}$, such that the following conditions hold.

1. The undirected graph $T = ([k], \{\{i, j\} \mid i \neq j, X_i \asymp X_j\})$ is a tree.
2. $P|_{X_i}$ is a total order for each $i \in [k]$.
3. For every node $v \in [k]$ such that v is an internal node in T and for every two connected components C_1, C_2 of T that are created after removing v from T , we have the following. Consider two nodes $i \in C_1$ and $j \in C_2$ and two events $e_1 \in X_i$ and $e_2 \in X_j$ such that $e_1 <_{\mathbf{P}} e_2$, there exists some event $e \in X_v$ such that $e_1 <_{\mathbf{P}} e <_{\mathbf{P}} e_2$.

We call an rf-poset $(X, P, \mathbf{RF})$ tree-inducible if (X, P) is tree-inducible. The insight is that traces from tree communication topologies yield tree-inducible trace ideals. Our motivation behind tree inducibility comes from the following lemma.

Lemma 9.8. Rf-poset realizability of tree-inducible rf-posets can be solved in $O(k^2 \cdot d \cdot n^2 \cdot \log n)$ time, where $n = |X|$, $d = |\mathbf{Locks}_X \cup \mathbf{Vars}_X|$ and $k = |\mathbf{Threads}_X|$.

The proof of Lemma 9.8 is in two steps. Recall the definition of closed rf-posets from Section 9.2.1. First, we show that a tree-inducible, closed rf-poset is realizable (Lemma 9.9). Second, we show that the closure of a tree-inducible rf-poset is also tree-inducible (Lemma 9.11).

Lemma 9.9. Every closed, tree-inducible rf-poset is realizable.

Indeed, consider a tree-inducible, closed rf-poset $\mathcal{P} = (X, P, \mathbf{RF})$. The witness σ realizing $\mathcal{P}$ is obtained in two steps.

1. We construct a poset (X, Q) with $Q \sqsubseteq P$ as follows. Initially, we let Q be identical to P . Let $\mathcal{P}$ be tree-inducible to a tree $T = ([k], \{\{i, j\} \mid X_i \asymp X_j\})$. We traverse T top-down, and for every node i and child j of i , for every two events $e_1 \in X_i$ and $e_2 \in X_j$ with $e_1 \asymp e_2$ and $e_2 \not\leq_{\mathbf{P}} e_1$, we order $e_1 <_Q e_2$. Finally, we transitively close Q .
2. We construct σ by linearizing (X, Q) arbitrarily.

Let us first establish the consistency of the above construction.

Lemma 9.10. (X, Q) is a poset.

Proof. Assume towards contradiction otherwise. Consider the process in which we insert the orderings in Q in sequence, and examine the first pair of conflicting events (e_1, e_2) such that we try to order $e_1 <_Q e_2$ whereas it already holds that $e_2 <_Q e_1$. For $j \in [2]$, let i_j be such that $e_j \in X_{i_j}$, and it follows that i_1 is the parent of i_2 in T . Observe that, by

the construction of Q , we have $e_1 \parallel_{e_2}^P$. Consider the following sequence of orderings that witnesses (transitively) that $e_2 <_Q e_1$.

$$e_2 = \bar{e}_1 \ll_Q \bar{e}_2 \ll_Q \dots \ll_Q \bar{e}_\ell = e_1 \quad (9.11)$$

Since $\mathcal{P}$ is tree-inducible, there exists some $j \in [\ell]$ such that $\bar{e}_j \in X_{i_1}$ and $e_2 <_P \bar{e}_j$. Since X_{i_1} is totally ordered in P , the events $\bar{e}_j$ and e_1 are ordered in P . Clearly $e_1 <_P \bar{e}_j$, otherwise we would have $e_2 <_P e_1$. But then we already have $\bar{e}_j <_Q e_1$, and hence a cycle already exists in Q , contradicting our assumption about (e_1, e_2) being the first pair where a cycle is encountered. The desired result follows. QED.

As the above lemma establishes that σ is well-defined, we can prove Lemma 9.9 by showing that σ is a witness of the realizability of $\mathcal{P}$.

Proof of Lemma 9.9. Consider any triplet $(\mathbf{w}, \mathbf{r}, \mathbf{w}') \in \text{Triplets}(\mathcal{P})$ and we argue that (i) $\mathbf{w} <_{\text{tr}}^\sigma \mathbf{r}$ and (ii) if $\mathbf{w} <_{\text{tr}}^\sigma \mathbf{w}'$ then $\mathbf{r} <_{\text{tr}}^\sigma \mathbf{w}'$. For (i), by the definition of rf-posets, we have $\mathbf{w} <_P \mathbf{r}$ and thus $\mathbf{w} <_{\text{tr}}^\sigma \mathbf{r}$. We now turn our attention to (ii). If $\mathbf{w}' \parallel_{\mathbf{r}}^P$ or $\mathbf{w}' \parallel_{\mathbf{w}}^P$, since $\mathcal{P}$ is closed, we have either $\mathbf{w}' <_P \mathbf{w}$ or $\mathbf{r} <_P \mathbf{w}'$, and hence $\mathbf{w}' <_Q \mathbf{w}$ or $\mathbf{r} <_Q \mathbf{w}'$. Since σ is a linearization of Q , the first case leads to a contradiction, whereas the second case leads to $\mathbf{r} <_{\text{tr}}^\sigma \mathbf{w}'$, as desired. Now assume that $\mathbf{w}' \parallel_{\mathbf{r}}^P$ and $\mathbf{w}' \parallel_{\mathbf{w}}^P$, and let $i \in [k]$ be such that $\mathbf{r} \in X_i$. Since $\mathcal{P}$ is tree-inducible, it follows that $\mathbf{w} \in X_i$ and $\mathbf{w}' \in X_j$ for some $i \neq j$, and such that (i, j) is an edge of T . Since $\mathbf{w} <_{\text{tr}}^\sigma \mathbf{w}'$, we have $\mathbf{w} <_Q \mathbf{w}'$, and thus, by construction, j is a child of i in T . Again, by construction, we have $\mathbf{r} <_Q \mathbf{w}$. Since σ is a linearization of Q , we have that $\text{RF}_\sigma(\mathbf{r}) \neq \mathbf{w}'$.

The desired result follows. QED.

The next lemma shows that tree-inducibility is preserved under taking closures (if the closure exists).

Lemma 9.11. Consider an rf-poset $\mathcal{P} = (X, P, \text{RF})$ and let $\mathcal{Q} = (X, Q, \text{RF})$ be the closure of $\mathcal{P}$. If $\mathcal{P}$ is tree-inducible then $\mathcal{Q}$ is also tree-inducible.

Proof. Let $T = ([k], \{(i, j) \mid X_i \asymp X_j\})$ be a rooted tree such that $\mathcal{P}$ is tree-inducible to T , and we argue that $\mathcal{Q}$ is also tree-inducible to T . Since $Q \subseteq P$, clearly the condition 1 and condition 2 of tree-inducibility are met. Condition 3 follows from the fact that (i) Q is the transitive closure of $P \cup Q'$, where Q' is a relation between conflicting events, and (ii) since T is a tree, for any two conflicting events e_1, e_2 we have either $e_1, e_2 \in X_i$ for some $i \in [k]$, or $e_1 \in X_i$ and $e_2 \in X_j$ for some $i, j \in [k]$ such that (i, j) is an edge of T .

The desired result follows. QED.

Now we can conclude the proof of Lemma 9.8, which is step (i) towards the proof of Theorem 9.3.

Proof of Lemma 9.8. Consider a tree-inducible rf-poset $\mathcal{P}$. In $O(k^2 \cdot d \cdot n^2 \cdot \log n)$ time, we can decide whether the closure of $\mathcal{P} = (X, P, \text{RF})$ exists [35]. By Remark 9.2, if the closure does not exist, $\mathcal{P}$ is not realizable. On the other hand, if the closure exists, denote it by $\mathcal{Q} = (X, Q, \text{RF})$. By Lemma 9.11, $\mathcal{Q}$ is tree-inducible, and by Lemma 9.9, it is realizable by a witness σ^* . Since $Q \sqsubseteq P$, we have that $\sigma^* \sqsubseteq P$, and thus σ^* also realizes $\mathcal{P}$.

The desired result follows.

QED.

Since, by Remark 9.2, an rf-poset is realizable only if its closure exists and is realizable, Lemma 9.9 and Lemma 9.11 allow to decide the realizability of an rf-poset by computing its closure. The complexity of the algorithm comes from the complexity of deciding whether the closure exists.

Recall that Lemma 9.3 provides an upper-bound on the number of trace ideals of σ that we need to examine for realizability in order to decide data-race prediction. We now proceed with point (ii) towards Theorem 9.3, i.e., we show that for tree communication topologies, a single ideal suffices. Our proof is based on the notion of lock causal cones below.

Lock causal cones. Consider a trace σ that defines a tree communication topology $G_\sigma = (V_\sigma, E_\sigma)$. Given an event $e \in \text{Events}_\sigma$ the *lock causal cone* $\text{LCone}_\sigma(e)$ of e is the set X defined by the following process. Consider that G_σ is rooted in $\text{thr}(e)$.

1. Initially X contains all predecessors of e in $(\text{Events}_\sigma, <_{\text{TO}}^\sigma)$. We perform a top-down traversal of G_σ , and consider a current thread t_1 visited by the traversal.
2. Let t_2 be the parent of t_1 in the traversal tree, and e_2 be the unique maximal event in $(X|t_2, <_{\text{TO}})$, i.e., e_2 is the last event of thread t_2 that appears in X . We insert in X all events $e_1 \in \text{Events}_\sigma|t_1$ such that $e_1 <_{\text{TRF}}^\sigma e_2$.
3. While there exists some acquire event $\text{acq}_1 \in X|t_1$ and there exists another acquire event $\text{acq}_2 \in \text{OpenAcqs}(X)$ with $\text{acq}_1 \preceq \text{acq}_2$ and $\text{thr}(\text{acq}_2) = t_2$, we insert in X all predecessors of rel_1 in $(\text{Events}_\sigma, <_{\text{TO}}^\sigma)$ (including rel_1), where $\text{rel}_1 = \text{match}_\sigma(\text{acq}_1)$.

Observe that, by construction, $\text{LCone}_\sigma(e)$ is a lock-feasible trace ideal of σ . In addition, for any two events $e_1, e_2 \in \text{Events}_\sigma$, the set $\text{LCone}_\sigma(e_1) \cup \text{LCone}_\sigma(e_2)$ is an ideal of σ , though not necessarily lock-feasible. Our motivation behind lock causal cones comes from the following lemma. Intuitively, we can decide a predictable data-race by deciding the realizability of the ideal that is the union of the two lock-causal cones.

Lemma 9.12. Let $X = \text{LCone}_\sigma(e_1) \cup \text{LCone}_\sigma(e_2)$. We have that (e_1, e_2) is a predictable data race of σ iff (i) $\{e_1, e_2\} \cap X = \emptyset$, and (ii) X is a realizable trace ideal of σ .

In order to prove Lemma 9.12, we will first establish the following. Intuitively, it states that if σ^* is a trace in which e is enabled, then $\sigma^*|_{\text{LCone}_\sigma(e)}$ also has this property.

Lemma 9.13. Let $X = \text{LCone}_\sigma(e)$, and σ^* be any correct reordering of σ in which e is enabled. The following assertions hold.

- (a) $X \subseteq \text{Events}_{\sigma^*}$.
- (b) Consider any thread $t_1 \neq \text{thr}(e)$, and thread t_2 such that (i) $X|_{t_1} \neq \emptyset$, and (ii) t_2 is a parent of t_1 in the tree topology G_σ rooted at $\text{thr}(e)$. Let e_2 be the unique maximal event of $X|_{t_2}$ in σ^* , and e_1 any event in $X|_{t_1}$. We have that $e_1 <_{\sigma^*} e_2$.

Proof. The proof is by induction on the steps of the process that constructs $\text{LCone}_\sigma(e)$. Clearly, both statements hold after step 1 of the process has been executed. Similarly, both statements hold easily after each time step 2 of the process has been executed. We now proceed with step 3 of the process. For each $i \in [2]$, consider the lock-acquire events $\text{acq}_i \in X$ as identified in this step, and $\text{rel}_i = \text{match}_\sigma(\text{acq}_i)$. Observe that e_2 appears in the critical section of acq_2 , while by the induction hypothesis, we have $\text{acq}_1, \text{acq}_2 \in \text{Events}_{\sigma^*}$. We distinguish the step that led to $\text{acq}_1 \in X$. Step 2. Then $\text{acq}_1 <_{\text{TRF}} e_2$, and thus (i) all predecessors of rel_1 (including rel_1) appear in σ^* , and (ii) $\text{rel}_1 <_{\text{tr}}^{\sigma^*} \text{acq}_2$, hence $\text{rel}_1 <_{\text{tr}}^{\sigma^*} e_2$, and thus $e_1 <_{\text{tr}}^{\sigma^*} e_2$ for all predecessors of rel_1 inserted to X . Step 3. For each $i \in [2]$, consider the lock-acquire events acq'_i as identified in that step, and $\text{rel}'_i = \text{match}_\sigma(\text{acq}'_i)$. Note that $\text{acq}_1 <_{\text{TO}} \text{rel}'_1$, and since, by the induction hypothesis, we have $\text{rel}'_1 <_{\text{tr}}^{\sigma^*} e_2$, we obtain $\text{acq}_1 <_{\text{tr}}^{\sigma^*} e_2$. Thus, again, (i) all predecessors of rel_1 (including rel_1) appear in σ^* , and (ii) $\text{rel}_1 <_{\text{tr}}^{\sigma^*} \text{acq}_2$, hence $\text{rel}_1 <_{\text{tr}}^{\sigma^*} e_2$, and thus $e_1 <_{\text{tr}}^{\sigma^*} e_2$ for all predecessors of rel_1 inserted to X .

The desired result follows. QED.

Next we have a technical lemma, which will allow us to conclude that if (e_1, e_2) is a predictable data race of σ , the ideal $\text{LCone}_\sigma(e_1) \cup \text{LCone}_\sigma(e_2)$ is lock-feasible.

Lemma 9.14. For any two conflicting events e_1, e_2 , consider the set $X = \text{LCone}_\sigma(e_1) \cup \text{LCone}_\sigma(e_2)$. For any two conflicting events $e'_1, e'_2 \in X$ such that $\text{thr}(e'_i) \neq \text{thr}(e_1)$ for each $i \in [2]$, we have that $e'_1, e'_2 \in \text{LCone}_\sigma(e_i)$, for some $i \in [2]$.

Proof. We assume that $\text{thr}(e_1) \neq \text{thr}(e_2)$, as the statement clearly holds otherwise. Assume w.l.o.g. that $e'_i \in \text{LCone}_\sigma(e_i)$ for each $i \in [2]$, and consider the tree $T = ([k], \{(i, j) \mid \text{Events}_\sigma|_{t_i} \prec$

$\text{Events}_\sigma[t_j\}$). Let $i_1, i_2, i'_1, i'_2 \in [k]$ be such that $\text{thr}(e_1) = t_{i_1}$, $\text{thr}(e_2) = t_{i_2}$, $\text{thr}(e'_1) = t_{i'_1}$ and $\text{thr}(e'_2) = t_{i'_2}$, and since $e_1 \succ e_2$, we have that (i_1, i_2) is an edge of T . Consider the two components C_1, C_2 that are created in T by removing the edge (i_1, i_2) , such that $i_1 \in C_1$ and $i_2 \in C_2$. Since $e'_1 \succ e'_2$ and $i'_1, i'_2 \neq i_1$, we have that $i'_1, i'_2 \in C_1$ or $i'_1, i'_2 \in C_2$. We only consider $i'_1, i'_2 \in C_2$, as the other case is similar.

Since $e'_1 \in \text{LCone}_\sigma(e_1)$, there exists event $e \in \text{LCone}_\sigma(e_1)$ such that $\text{thr}(e) = t_{i_2}$ and either $e'_1 = e$ or $e'_1 \in \text{LCone}_\sigma(e)$. If $e <_{\text{TO}} e_2$, then $\text{LCone}_\sigma(e) \subseteq \text{LCone}_\sigma(e_2)$ and thus $e'_1 \in \text{LCone}_\sigma(e_2)$. Otherwise, $e_2 \in \text{LCone}_\sigma(e_1)$, hence $\text{LCone}_\sigma(e_2) \subseteq \text{LCone}_\sigma(e_1)$ and thus $e'_2 \in \text{LCone}_\sigma(e_1)$.

The desired result follows. QED.

Proof of Lemma 9.12. For the $(\Leftarrow)$ direction, notice that e_1 and e_2 are enabled in X , and since X is realizable, we have that (e_1, e_2) is a predictable data race of σ . We now focus on the $(\Rightarrow)$ direction.

Let σ^* be a witness of the data race (e_1, e_2) . As e_1 and e_2 are enabled in σ^* , we have $\{e_1, e_2\} \cap \text{Events}_{\sigma^*} = \emptyset$. By Lemma 9.13, we have $X \subseteq \text{Events}_{\sigma^*}$, and thus $\{e_1, e_2\} \cap X = \emptyset$.

We now argue that $\sigma' = \sigma^*|X$ realizes X . Since $X \subseteq \text{Events}_{\sigma^*}$ and X is a trace ideal, it only remains to show that σ' respects the critical sections. Consider any two lock-acquire events $\text{acq}_1, \text{acq}_2 \in X$ such that $\text{acq}_1 \succ \text{acq}_2$ and $\text{acq}_1 <_{\sigma^*} \text{acq}_2$, and we argue that $\text{rel}_1 \in X$ where $\text{rel}_1 = \text{match}_\sigma(\text{acq}_1)$. Let $Y = \text{Events}_\sigma|\{\text{thr}(e_1), \text{thr}(e_2)\}$, and observe that $X|\{\text{thr}(e_1), \text{thr}(e_2)\} = Y$, thus the statement is true if $\{\text{thr}(\text{acq}_1), \text{thr}(\text{acq}_2)\} \subseteq \{\text{thr}(e_1), \text{thr}(e_2)\}$. Otherwise, we have $\text{thr}(\text{acq}_j) \neq \text{thr}(e_i)$ for some $i \in [2]$ and each $j \in [2]$. By Lemma 9.14, we have that $\text{acq}_1, \text{acq}_2 \in \text{LCone}_\sigma(e_l)$ for some $l \in [2]$. Assume towards contradiction that $\text{rel}_1 \notin X$. By the definition of $\text{LCone}_\sigma(e_l)$, we have that $\text{thr}(\text{acq}_1)$ is a parent of $\text{thr}(\text{acq}_2)$ in the tree G_σ rooted at $\text{thr}(e_l)$. By Lemma 9.13, we have $\text{acq}_2 <_{\sigma^*} e$, where e is the last event of $X|\text{thr}(\text{acq}_1)$ in σ^* . Note that e belongs to the critical section of acq_1 , hence we must have $\text{acq}_2 <_{\sigma^*} \text{acq}_1$, a contradiction.

The desired result follows. QED.

Finally, Theorem 9.3 follows immediately from Lemma 9.8 and Lemma 9.12.

Proof of Theorem 9.3. By Lemma 9.12, we have that (e_1, e_2) is a predictable data race of σ iff $\{e_1, e_2\} \cap X = \emptyset$ and X is realizable. By Lemma 9.8, deciding the realizability of X is done in $O(\mathcal{T}^2 \cdot (\mathcal{V} + \mathcal{L}) \cdot \mathcal{N}^2 \cdot \log \mathcal{N})$ time. The desired result follows. QED.

9.4.2 A Lower Bound for Two Threads

In this section we prove a conditional quadratic lower bound for dynamic data-race prediction for two threads. Our proof is via a reduction from the Orthogonal Vectors problem. To make it conceptually simpler, we present our reduction in two steps. First, we show a fine-grained reduction from the Orthogonal Vectors problem to the realizability of an rf-poset with 2 threads and 7 variables. Afterwards, we show how the realizability problem for the rf-posets of the first step can be reduced to the decision problem of dynamic data race prediction with 2 threads, 9 variables and 1 lock.

The Orthogonal Vectors problem (OV). An instance of Orthogonal Vectors consists of two sets A, B , where each set contains $n/2$ binary vectors in D dimensions. The task is to determine whether there exists a pair of vectors $(a, b) \in A \times B$ that is orthogonal, i.e., for all $i \in [D]$ we have $a[i] \cdot b[i] = 0$. There exist algorithms that solve the problem in $O(n^2 \cdot D)$ and $O(2^D \cdot n)$ time, simply by computing the inner product of each pair $(a, b) \in A \times B$ and following a classic Four-Russians technique, respectively. It is conjectured that there is no truly sub-quadratic algorithm for OV [110].

Conjecture 9.1 (Orthogonal Vectors). There is no algorithm for OV that operates in $O(n^{2-\epsilon} \cdot D^{O(1)})$ time, for any $\epsilon > 0$.

It is also known that SETH implies the OV conjecture [115].

We first relate OV with rf-poset realizability.

Lemma 9.15. Rf-poset realizability for an rf-poset with 2 threads and 7 variables has no $O(n^{2-\epsilon})$ -time algorithm for any $\epsilon > 0$, under the Orthogonal Vectors conjecture.

In the following, we provide a fine-grained reduction from OV to rf-poset realizability.

Reduction from OV to rf-poset realizability. For a fine-grained reduction from OV to rf-poset realizability, consider an OV instance (A, B) , where $A = (a_j)_{1 \leq j \leq n/2}$, $B = (b_l)_{1 \leq l \leq n/2}$, and each $a_j, b_l \in \{0, 1\}^D$. We will construct an rf-poset $\mathcal{P} = (X, P, \text{RF})$ with 2 threads and 7 variables such that the closure of $\mathcal{P}$ exists iff there exists a pair of orthogonal vectors $(a, b) \in A \times B$. Since 2 threads define a tree-inducible rf-poset, Remark 9.2 and Lemma 9.9 imply that $\mathcal{P}$ is realizable iff OV has a positive answer. The set X consists of two disjoint sets X_A, X_B , so that each is totally ordered in P . For ease of presentation, we denote by τ_A, τ_B the linear orders $(X_A, P|_{X_A})$ and $(X_B, P|_{X_B})$, respectively. To develop some insight, we start with a high-level view of the construction, and then proceed with the details.

Overview of the construction. The linear orders τ_A and τ_B encode the vectors of A and B , respectively. Each of τ_A and τ_B consists of $n/2$ segments, so that the i -th (resp. $(n/2 - i + 1)$ -th) segment of τ_A (resp. τ_B) encodes the contents of the i -th vector of A (resp., B). The two total orders are constructed with a closure computation in mind, which inserts event orderings in P one-by-one. In high-level, an ordering $e_1 < e_2$ encodes the test of whether the bits in a specific coordinate i of two vectors $a_j \in A$ and $b_l \in B$ have product 0. If yes, and moreover, $i < D$, then the closure conditions enforce a new ordering $e'_1 < e'_2$, which encodes the test of the bits in coordinate $i + 1$. Otherwise $i = D$, and the closure has been computed and an orthogonal pair has been found. On the other hand, if the bits in coordinate i have product 1, the two current vectors are not orthogonal, and the closure conditions enforce a new ordering $e''_1 < e''_2$, which encodes the test of the first coordinate of the next pair of vectors. The above is achieved using 7 variables $\{x_i\}_{i \in [7]}$.

Formal construction. We now present the formal details of the construction (illustrated in Figure 9.2). The construction creates various events which have the form e^{a_j} and e^{b_l} when they are used at the vector level, and have the form $e_i^{a_j}$ and $e_i^{b_l}$, where $i \in [D]$, when they are used at the coordinate level. As a general rule, for each $j, l \in [n/2 - 1]$, we have $e^{a_j} <_{\tau_A} e^{a_{j+1}}$ and $e^{b_{l+1}} <_{\tau_B} e^{b_l}$, both for events at the vector and at the coordinate level. At the coordinate level, we also have $e_{i+1}^{a_j} <_{\tau_A} e_i^{a_j}$ and $e_{i+1}^{b_l} <_{\tau_B} e_i^{b_l}$. For succinctness, we often write $e_1, e_2 < e_3$ to denote $e_1 < e_3$ and $e_2 < e_3$. We next describe the events and orderings between them. The partial order P is the transitive closure of these orderings.

Events on x_1 and x_2 . For every vector $a_j \in A$ and coordinate $i \in [D]$, we create three events $\mathbf{w}_i^{a_j}(x_1)$, $\mathbf{w}_i^{a_j}(x_2)$ and $\mathbf{r}_i^{a_j}(x_2)$. We make $\text{RF}(\mathbf{r}_i^{a_j}(x_2)) = \mathbf{w}_i^{a_j}(x_2)$, and order

$$\mathbf{w}_i^{a_j}(x_1), \mathbf{w}_i^{a_j}(x_2) <_{\tau_A} \mathbf{r}_i^{a_j}(x_2) . \quad (9.12)$$

For every vector $b_l \in B$ and coordinate $i \in [D]$, we create three events $\mathbf{w}_i^{b_l}(x_1)$, $\mathbf{r}_i^{b_l}(x_1)$ and $\mathbf{w}_i^{b_l}(x_2)$. We make $\text{RF}(\mathbf{r}_i^{b_l}(x_1)) = \mathbf{w}_i^{b_l}(x_1)$, and order

$$\mathbf{w}_i^{b_l}(x_1), \mathbf{w}_i^{b_l}(x_2) <_{\tau_B} \mathbf{r}_i^{b_l}(x_1) . \quad (9.13)$$

In addition, we order

$$\begin{aligned} \mathbf{w}_i^{a_j}(x_2) <_{\tau_A} \mathbf{w}_i^{a_j}(x_1) & \text{ iff } a_j[i] = 1 & \text{ and } \\ \mathbf{w}_i^{b_l}(x_1) <_{\tau_B} \mathbf{w}_i^{b_l}(x_2) & \text{ iff } b_l[i] = 1 . \end{aligned} \quad (9.14)$$

Observe that if $a_j[i] \cdot b_l[i] = 1$, and if we order $\mathbf{w}_i^{a_j}(x_1) < \mathbf{w}_i^{b_l}(x_1)$ then transitively $\mathbf{w}_i^{a_j}(x_2) < \mathbf{w}_i^{b_l}(x_2)$ and hence by closure $\mathbf{r}_i^{a_j}(x_2) < \mathbf{w}_i^{b_l}(x_2)$.

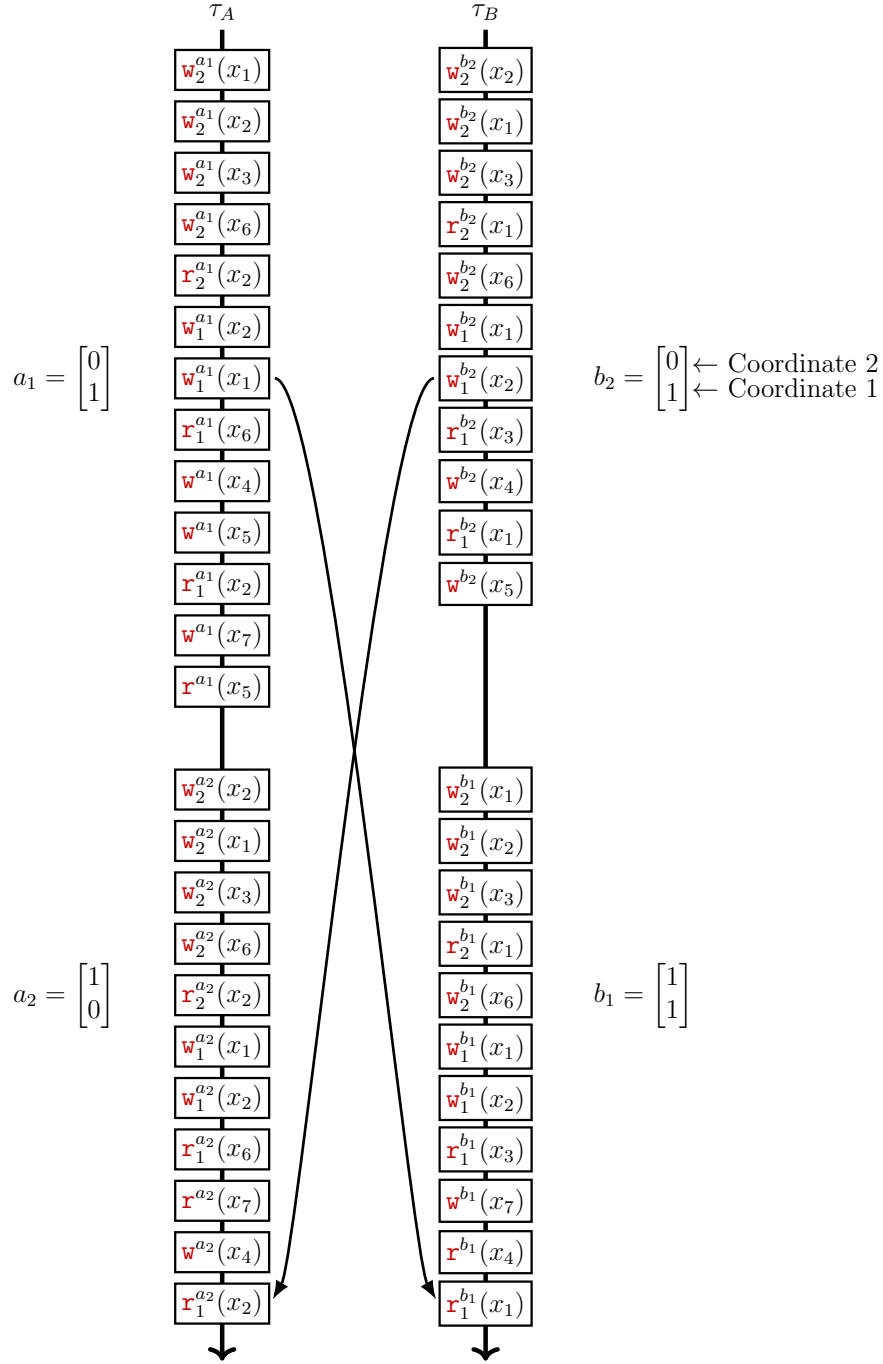


Figure 9.2: Illustration of the reduction of an OV instance $(A = \{a_1, a_2\}, B = \{b_1, b_2\})$ to the Closure problem of an $\mathcal{P}$.

Events on x_3 . Let $i \in [D - 1]$ be a coordinate. For every vector $a_j \in A$, we create an event $\mathbf{w}_{i+1}^{a_j}(x_3)$, and order

$$\mathbf{w}_{i+1}^{a_j}(x_1), \mathbf{w}_{i+1}^{a_j}(x_2) <_{\tau_A} \mathbf{w}_{i+1}^{a_j}(x_3) <_{\tau_A} \mathbf{w}_i^{a_j}(x_1), \mathbf{w}_i^{a_j}(x_2) . \quad (9.15)$$

For every vector $b_l \in B$, we create two events $\mathbf{w}_{i+1}^{b_l}(x_3)$ and $\mathbf{r}_i^{b_l}(x_3)$, and make $\text{RF}(\mathbf{r}_i^{b_l}(x_3)) = \mathbf{w}_{i+1}^{b_l}(x_3)$. We order

$$\begin{aligned} \mathbf{w}_{i+1}^{b_l}(x_1), \mathbf{w}_{i+1}^{b_l}(x_2) <_{\tau_B} \mathbf{w}_{i+1}^{b_l}(x_3) <_{\tau_B} \mathbf{r}_{i+1}^{b_l}(x_1) \quad \text{and} \\ \mathbf{w}_i^{b_l}(x_1), \mathbf{w}_i^{b_l}(x_2) <_{\tau_B} \mathbf{r}_i^{b_l}(x_3) <_{\tau_B} \mathbf{r}_i^{b_l}(x_1) . \end{aligned} \quad (9.16)$$

Observe that if we order $\mathbf{w}_i^{a_j}(x_1) < \mathbf{w}_i^{b_l}(x_1)$ then we also have $\mathbf{w}_{i+1}^{a_j}(x_3) < \mathbf{r}_i^{b_l}(x_3)$, hence by closure $\mathbf{w}_{i+1}^{a_j}(x_3) < \mathbf{w}_{i+1}^{b_l}(x_3)$ and thus $\mathbf{w}_{i+1}^{a_j}(x_1) < \mathbf{r}_i^{b_l}(x_1)$.

Events on x_6 . For every coordinate $i \in [D - 1]$, we do as follows. For every vector $a_j \in A$, we create two events $\mathbf{w}_{i+1}^{a_j}(x_6)$ and $\mathbf{r}_i^{a_j}(x_6)$, and make $\text{RF}(\mathbf{r}_i^{a_j}(x_6)) = \mathbf{w}_{i+1}^{a_j}(x_6)$. We order

$$\begin{aligned} \mathbf{w}_{i+1}^{a_j}(x_3) <_{\tau_A} \mathbf{w}_{i+1}^{a_j}(x_6) <_{\tau_A} \mathbf{r}_{i+1}^{a_j}(x_2) \quad \text{and} \\ \mathbf{w}_i^{a_j}(x_1), \mathbf{w}_i^{a_j}(x_2) <_{\tau_A} \mathbf{r}_i^{a_j}(x_6) <_{\tau_A} \mathbf{r}_i^{a_j}(x_2) . \end{aligned} \quad (9.17)$$

For every vector $b_l \in B$, we create one event $\mathbf{w}_{i+1}^{b_l}(x_6)$, and order

$$\mathbf{r}_{i+1}^{b_l}(x_1) <_{\tau_B} \mathbf{w}_{i+1}^{b_l}(x_6) <_{\tau_B} \mathbf{w}_i^{b_l}(x_1), \mathbf{w}_i^{b_l}(x_2) . \quad (9.18)$$

Observe that if we order $\mathbf{r}_{i+1}^{a_j}(x_2) < \mathbf{w}_{i+1}^{b_l}(x_2)$, since $\mathbf{w}_{i+1}^{b_l}(x_2) <_{\tau_B} \mathbf{r}_{i+1}^{b_l}(x_1)$, we also have $\mathbf{w}_{i+1}^{a_j}(x_6) < \mathbf{w}_{i+1}^{b_l}(x_6)$, hence by closure $\mathbf{r}_i^{a_j}(x_6) < \mathbf{w}_{i+1}^{b_l}(x_6)$ and thus $\mathbf{w}_i^{a_j}(x_2) < \mathbf{w}_i^{b_l}(x_2)$.

Events on x_4 . For every vector $a_j \in A$, we create one event $\mathbf{w}^{a_j}(x_4)$, and order

$$\mathbf{r}_1^{a_j}(x_6) <_{\tau_A} \mathbf{w}^{a_j}(x_4) <_{\tau_A} \mathbf{r}_1^{a_j}(x_2) . \quad (9.19)$$

For every vector $b_l \in B$, with $l \in [n/2 - 1]$, we create two events $\mathbf{w}^{b_{l+1}}(x_4)$ and $\mathbf{r}^{b_l}(x_4)$, and make $\text{RF}(\mathbf{r}^{b_l}(x_4)) = \mathbf{w}^{b_{l+1}}(x_4)$. We order

$$\begin{aligned} \mathbf{r}_1^{b_l}(x_3) <_{\tau_B} \mathbf{r}^{b_l}(x_4) <_{\tau_B} \mathbf{r}_1^{b_l}(x_1) \quad \text{and} \\ \mathbf{r}_1^{b_{l+1}}(x_3) <_{\tau_B} \mathbf{w}^{b_{l+1}}(x_4) <_{\tau_B} \mathbf{r}_1^{b_{l+1}}(x_1) . \end{aligned} \quad (9.20)$$

Observe that if we order $\mathbf{r}_1^{a_j}(x_2) < \mathbf{w}_1^{b_l}(x_2)$ then we also have $\mathbf{w}^{a_j}(x_4) < \mathbf{r}^{b_l}(x_4)$ (since $\mathbf{w}_1^{b_l}(x_2) <_{\tau_B} \mathbf{r}_1^{b_l}(x_3)$ by Equation (9.16)) and thus by closure $\mathbf{w}^{a_j}(x_4) < \mathbf{w}^{b_{l+1}}(x_4)$.

Events on x_5 and x_7 . For every vector $a_j \in A$, with $j \in [n/2 - 1]$, we create two events

$\mathbf{w}^{a_j}(x_5)$ and $\mathbf{r}^{a_j}(x_5)$, and make $\text{RF}(\mathbf{r}^{a_j}(x_5)) = \mathbf{w}^{a_j}(x_5)$. We also create two events $\mathbf{w}^{a_j}(x_7)$ and $\mathbf{r}^{a_{j+1}}(x_7)$, and make $\text{RF}(\mathbf{r}^{a_{j+1}}(x_7)) = \mathbf{w}^{a_j}(x_7)$. We order

$$\begin{aligned} \mathbf{w}^{a_j}(x_4) <_{\tau_A} \mathbf{w}^{a_j}(x_5) <_{\tau_A} \mathbf{r}_1^{a_j}(x_2) <_{\tau_A} \mathbf{w}^{a_j}(x_7) <_{\tau_A} \mathbf{r}^{a_j}(x_5) \text{ and} \\ \mathbf{r}_1^{a_{j+1}}(x_6) <_{\tau_A} \mathbf{r}^{a_{j+1}}(x_7) <_{\tau_A} \mathbf{w}^{a_{j+1}}(x_4) . \end{aligned} \quad (9.21)$$

We also create two events $\mathbf{w}^{b_{n/2}}(x_5)$ and $\mathbf{w}^{b_1}(x_7)$, and order

$$\begin{aligned} \mathbf{r}_1^{b_{n/2}}(x_1) <_{\tau_B} \mathbf{w}^{b_{n/2}}(x_5) <_{\tau_B} \mathbf{w}_D^{b_{n/2}-1}(x_1), \mathbf{w}_D^{b_{n/2}-1}(x_2) \text{ and} \\ \mathbf{r}_1^{b_1}(x_3) <_{\tau_B} \mathbf{w}^{b_1}(x_7) <_{\tau_B} \mathbf{r}^{b_1}(x_4) . \end{aligned} \quad (9.22)$$

Observe that if we order $\mathbf{r}_1^{a_j}(x_2) < \mathbf{w}_1^{b_{n/2}}(x_2)$ then we also have $\mathbf{w}^{a_j}(x_5) < \mathbf{w}^{b_{n/2}}(x_5)$ (since $\mathbf{w}_1^{b_{n/2}}(x_2) <_{\tau_B} \mathbf{r}_1^{b_{n/2}}(x_1)$ by a previous item) and thus by closure $\mathbf{r}^{a_j}(x_5) < \mathbf{w}^{b_{n/2}}(x_5)$. But then also $\mathbf{w}^{a_j}(x_7) < \mathbf{w}^{b_1}(x_7)$ (since $\mathbf{w}^{b_{n/2}}(x_5) <_{\tau_B} \mathbf{w}^{b_1}(x_7)$) and thus by closure $\mathbf{r}^{a_{j+1}}(x_7) < \mathbf{w}^{b_1}(x_7)$.

Final orderings. Finally, we order

$$\mathbf{w}^{b_{n/2}}(x_5) <_{\tau_B} \mathbf{w}_D^{b_{n/2}-1}(x_1), \mathbf{w}_D^{b_{n/2}-1}(x_2) . \quad (9.23)$$

For each $j \in [n/2 - 1]$, we order

$$\mathbf{r}^{a_j}(x_5) <_{\tau_A} \mathbf{w}_D^{a_{j+1}}(x_1), \mathbf{w}_D^{a_{j+1}}(x_2) . \quad (9.24)$$

For each $j \in [n/2 - 2]$, we order

$$\mathbf{r}^{b_{j+1}}(x_1) <_{\tau_B} \mathbf{w}_D^{b_j}(x_1), \mathbf{w}_D^{b_j}(x_2) \quad (9.25)$$

We make two orderings across τ_A and τ_B , namely

$$\mathbf{w}_1^{a_1}(x_1) <_P \mathbf{r}_1^{b_1}(x_1) \quad \text{and} \quad \mathbf{w}_1^{b_{n/2}}(x_2) <_P \mathbf{r}_1^{a_{n/2}}(x_2) . \quad (9.26)$$

Correctness. Observe that we have used 7 variables, while $|X_A| + |X_B| = O(n \cdot D)$, and the reduction can be easily computed in linear time. We now prove the correctness of the construction and complete the proof of Lemma 9.15. We first develop some helpful notation.

Given integers $j, l \in [n/2]$, we denote by X_j^l the events of X_A and X_B that correspond to vectors a_j and b_l , that is, events that have superscript a_j or b_l , with the following exception: if the observation $\text{RF}(\mathbf{r})$ of a read event $\mathbf{r}$ is not in X_j^l then $\mathbf{r}$ is also not in X_j^l . The notation carries over to partial orders Q_j^l and reads-from functions RF_j^l . We also use inequalities for the subscripts and superscripts of X (e.g., $X_j^{\leq n/2}$) to denote events that corresponds to

vectors $a_{j'}$ and $b_{l'}$ for j' and l' that satisfy the inequalities.

Consider a partial order Q' over a subset $Y \subseteq X$ such that $Q' \sqsubseteq \tau_A|Y$ and $Q' \sqsubseteq \tau_B|Y$. Given two events $e_1, e_2 \in Y$ such that $\{e_1, e_2\} \not\subseteq X_A$ and $\{e_1, e_2\} \not\subseteq X_B$ (i.e., the events belong to different sets among X_A and X_B) we say that a Q' has a *cross edge* $e_1 <_{Q'} e_2$ to mean that it has the ordering $e_1 <_{Q'} e_2$ and the orderings that are introduced transitively through it.

Before the final proof of Lemma 9.15, we present some technical, but conceptually simple, lemmas. Consider the two total orders τ_A and τ_B without any cross edges. The first lemma reasons at the coordinate level of two vectors a_j and b_l . It states that if we start with a cross edge $\mathbf{w}_1^{a_j}(x_1) < \mathbf{r}_1^{b_l}(x_1)$, the closure rules eventually lead to an ordering $\mathbf{r}_1^{a_j}(x_2) < \mathbf{w}_1^{b_l}(x_2)$ iff a_j and b_l are not orthogonal.

Lemma 9.16. For any $j, l \in [n/2]$, consider the rf-poset $\mathcal{Q}_j^l = (X_j^l, Q_j^l, \mathbf{RF}_j^l)$, where Q_j^l has a single cross edge $\mathbf{w}_1^{a_j}(x_1) <_{Q_j^l} \mathbf{r}_1^{b_l}(x_1)$. Let $\mathcal{S}_j^l = (X_j^l, S_j^l, \mathbf{RF}_j^l)$ be the closure of $\mathcal{Q}_j^l$. We have $\mathbf{r}_1^{a_j}(x_2) <_{S_j^l} \mathbf{w}_1^{b_l}(x_2)$ iff a_j and b_l are not orthogonal.

Proof (Sketch). First, observe that since $\mathbf{w}_1^{a_j}(x_1) <_{Q_j^l} \mathbf{r}_1^{b_l}(x_1)$, by closure, we will have $\mathbf{w}_1^{a_j}(x_1) <_{S_j^l} \mathbf{w}_1^{b_l}(x_1)$. By a simple induction on the events on variables x_1 and x_3 , we have that $\mathbf{w}_i^{a_j}(x_1) <_{S_j^l} \mathbf{w}_i^{b_l}(x_1)$ for each $i \in [D]$. In turn, one of these orderings leads to $\mathbf{w}_i^{a_j}(x_2) <_{S_j^l} \mathbf{w}_i^{b_l}(x_2)$ and thus $\mathbf{r}_i^{a_j}(x_2) <_{S_j^l} \mathbf{w}_i^{b_l}(x_2)$ iff $a_j[i] \cdot b_l[i] = 1$. By a simple induction on the events on variables x_2 and x_6 , we have that $\mathbf{r}_1^{a_j}(x_2) <_{S_j^l} \mathbf{w}_1^{b_l}(x_2)$ iff a_j and b_l are not orthogonal. QED.

The next lemma reasons at the vector B level. Consider any fixed $j \in [n/2]$. The lemma states that if we order $\mathbf{w}_1^{a_j}(x_1) < \mathbf{r}_1^{b_l}(x_1)$, if a_j is not orthogonal to $b_{l'}$ for any $l' \leq n/2 - 1$, the closure rules eventually lead to $\mathbf{w}_1^{a_j}(x_1) < \mathbf{r}_1^{b_{n/2}}(x_1)$. On the other hand, if there exists a smallest l_1 such that a_j is orthogonal to b_{l_1} , the closure rules will stop inserting cross edges between events of vectors a_j and all $b_{l'}$, for $l' \geq l_1 + 1$.

Lemma 9.17. For any $j \in [n/2]$ and $l \leq n/2 - 1$, consider the rf-poset $\mathcal{Q}_j^{\leq l+1} = (X_j^{\leq l+1}, Q_j^{\leq l+1}, \mathbf{RF}_j^{\leq l+1})$, where $Q_j^{\leq l+1}$ has a single cross edge $\mathbf{w}_1^{a_j}(x_1) <_{Q_j^{\leq l+1}} \mathbf{r}_1^{b_1}(x_1)$. Let $\mathcal{S}_j^{\leq l+1} = (X_j^{\leq l+1}, S_j^{\leq l+1}, \mathbf{RF}_j^{\leq l+1})$ be the closure of $\mathcal{Q}_j^{\leq l+1}$. The following assertions hold.

1. If a_j is not orthogonal to $b_{l'}$ for any $l' \in [l]$, then $\mathbf{w}_1^{a_j}(x_1) <_{S_j^{\leq l+1}} \mathbf{r}_1^{b_{l+1}}(x_1)$.
2. If a_j is orthogonal to $b_{l'}$, for some $l' \in [l]$, then there are no cross edges in $(X_j^{l_1+1 \leq l' \leq l+1}, S_j^{l_1+1 \leq l' \leq l+1})$, where l_1 is the smallest l' such that a_j is orthogonal to $b_{l'}$.

Proof (Sketch). The proof is by induction on l . For the base case ($l = 1$), consider the rf-poset $\mathcal{Q}_j^1$, and by Lemma 9.16, we have $\mathbf{r}_1^{a_j}(x_2) <_{S_j^1} \mathbf{w}_1^{b_1}(x_2)$ iff a_j and b_1 are not orthogonal.

Observe that if $\mathbf{r}_1^{a_j}(x_2) <_{S_j^1} \mathbf{w}_1^{b_1}(x_2)$, we also have $\mathbf{r}_1^{a_j}(x_2) <_{S_j^{\leq 2}} \mathbf{w}_1^{b_1}(x_2)$, and thus $\mathbf{w}^{a_j}(x_4) <_{S_j^{\leq 2}} \mathbf{r}^{b_1}(x_4)$. Then, by closure we have $\mathbf{w}^{a_j}(x_4) <_{S_j^{\leq 2}} \mathbf{w}^{b_2}(x_4)$ and thus transitively $\mathbf{w}_2^{a_j}(x_1) <_{S_j^{\leq 2}} \mathbf{r}_1^{b_2}(x_1)$. On the other hand, it can be easily seen that if $\mathbf{r}_1^{a_j}(x_2) \not<_{S_j^1} \mathbf{w}_1^{b_1}(x_2)$ then the cross edges in $(X_j^{\leq 2}, S_j^{\leq 2})$ are precisely the cross edges in $(X_j^{\leq 1}, S_j^{\leq 1})$, and thus there are no cross edges in (X_j^2, S_j^2) .

Now assume that the claim holds for l , and we argue that it holds for $l+1$. By the induction hypothesis, we have that if a_j is not orthogonal to $b_{l'}$ for any $l' \in [l]$, then $\mathbf{w}_1^{a_j}(x_1) <_{S_j^{\leq l+1}} \mathbf{r}_1^{b_{l+1}}(x_1)$ and thus $\mathbf{w}_1^{a_j}(x_1) <_{S_j^{\leq l+2}} \mathbf{r}_1^{b_{l+1}}(x_1)$. A similar analysis to the base case shows that if a_j is not orthogonal to b_{l+1} then $\mathbf{w}_1^{a_j}(x_1) <_{S_j^{\leq l+2}} \mathbf{r}_1^{b_{l+2}}(x_1)$. On the other hand, if a_j is orthogonal to b_{l+1} , it can be easily seen that the cross edges in $(X_j^{\leq l+2}, S_j^{\leq l+2})$ are precisely the cross edges in $(X_j^{\leq l+1}, S_j^{\leq l+1})$, and thus there are no cross edges in (X_j^{l+1}, S_j^{l+1}) , where $l_1 = l + 1$. Finally, if a_j is orthogonal to $b_{l'}$ for some $l' \in [l]$, the statement follows easily by the induction hypothesis.

The desired result follows. QED.

The next lemma reasons at the vector A level. The lemma states that if we order $\mathbf{w}_1^{a_1}(x_1) < \mathbf{r}_1^{b_1}(x_1)$, if $a_{j'}$ is not orthogonal to $b_{l'}$ for any $j' \leq n/2 - 1$ and $l' \leq n/2$, the closure rules eventually lead to $\mathbf{w}_1^{a_{n/2}}(x_1) < \mathbf{r}_1^{b_{n/2}}(x_1)$. On the other hand, if there exists a smallest j_1 such that a_{j_1} is orthogonal to $b_{l'}$, for some $l' \in [n/2]$, the closure rules will stop inserting cross edges between events of all vectors $a_{j'}$ and all vectors $b_{l'}$, for $j' \geq j_1 + 1$ and $l' \in [n/2]$.

Lemma 9.18. For any $j \in [n/2 - 1]$, consider the rf-poset $\mathcal{Q}_{\leq j+1}^{\leq n/2} = (X_{\leq j+1}^{\leq n/2}, Q_{\leq j+1}^{\leq n/2}, \mathbf{RF}_{\leq j+1}^{\leq n/2})$, where $Q_{\leq j+1}^{\leq n/2}$ has a single cross edge $\mathbf{w}_1^{a_1}(x_1) <_{Q_{\leq j+1}^{\leq n/2}} \mathbf{r}_1^{b_1}(x_1)$.

Let $\mathcal{S}_{\leq j+1}^{\leq n/2} = (X_{\leq j+1}^{\leq n/2}, S_{\leq j+1}^{\leq n/2}, \mathbf{RF}_{\leq j+1}^{\leq n/2})$ be the closure of $\mathcal{Q}_{\leq j+1}^{\leq n/2}$. The following assertions hold.

1. If $a_{j'}$ is not orthogonal to $b_{l'}$ for any $j' \in [j]$ and $l' \in [n/2]$, then $\mathbf{w}_1^{a_{j+1}}(x_1) <_{S_{\leq j+1}^{\leq n/2}} \mathbf{r}_1^{b_1}(x_1)$.
2. If $a_{j'}$ is orthogonal to $b_{l'}$ for some $j' \in [j]$ and $l' \in [n/2]$, then there are no cross edges in $(X_{j_1+1 \leq j' \leq j+1}^{\leq n/2}, S_{j_1+1 \leq j' \leq j+1}^{\leq n/2})$, where j_1 is the smallest j' such that $a_{j'}$ is orthogonal to $b_{l'}$, for some $l' \in [n/2]$.

Proof (Sketch). The proof is by induction on j . For the base case ($j = 1$), consider the rf-poset $\mathcal{Q}_1^{\leq n/2} = (X_1^{\leq n/2}, Q_1^{\leq n/2}, \mathbf{RF}_1^{\leq n/2})$, and by Lemma 9.17, we have that if a_1 is not orthogonal to $b_{l'}$ for any $l' \in [n/2 - 1]$, then $\mathbf{w}_1^{a_j}(x_1) <_{S_j^{\leq n/2}} \mathbf{r}_1^{b_{n/2}}(x_1)$. Consider the rf-poset $\mathcal{Q}_1^{n/2}$, and by Lemma 9.16, we have $\mathbf{r}_1^{a_1}(x_2) <_{S_1^{n/2}} \mathbf{w}_1^{b_{n/2}}(x_2)$ iff a_1 and $b_{n/2}$ are not orthogonal. Observe that if $\mathbf{r}_1^{a_1}(x_2) <_{S_1^{n/2}} \mathbf{w}_1^{b_{n/2}}(x_2)$, we also have $\mathbf{r}_1^{a_1}(x_2) <_{S_1^{\leq n/2}} \mathbf{w}_1^{b_{n/2}}(x_2)$, and thus $\mathbf{r}_1^{a_1}(x_2) <_{S_{\leq 2}^{\leq n/2}} \mathbf{w}_1^{b_{n/2}}(x_2)$, hence, transitively, $\mathbf{w}^{a_1}(x_5) <_{S_{\leq 2}^{\leq n/2}} \mathbf{w}^{b_{n/2}}(x_5)$. Hence, by closure,

we have $\mathbf{r}^{a_1}(x_5) <_{S_{\leq 2}^{\leq n/2}} \mathbf{w}^{b_{n/2}}(x_5)$ and thus transitively $\mathbf{w}^{a_1}(x_7) <_{S_{\leq 2}^{\leq n/2}} \mathbf{w}^{b_{n/2}}(x_7)$. Then, by closure we have $\mathbf{r}^{a_2}(x_7) <_{S_{\leq 2}^{\leq n/2}} \mathbf{w}^{b_{n/2}}(x_7)$ and thus transitively $\mathbf{w}_1^{a_2}(x_1) <_{S_{\leq 2}^{\leq n/2}} \mathbf{r}_1^{b_1}(x_1)$. On the other hand, it can be easily seen that if $\mathbf{r}_1^{a_1}(x_2) \not\prec_{S_1^{n/2}} \mathbf{r}_1^{b_{n/2}}(x_2)$ then the cross edges in $(X_{\leq 2}^{\leq n/2}, S_{\leq 2}^{\leq n/2})$ are precisely the cross edges in $(X_1^{\leq n/2}, S_1^{\leq n/2})$, and thus there are no cross edges in $(X_2^{\leq n/2}, S_2^{\leq n/2})$.

Now assume that the claim holds for j , and we argue that it holds for $j+1$. By the induction hypothesis, we have that if $a_{j'}$ is not orthogonal to $b_{l'}$ for any $j' \in [j]$ and $l' \in [n/2]$, then $\mathbf{r}_1^{a_{j+1}}(x_2) <_{S_{\leq j+1}^{\leq n/2}} \mathbf{w}_1^{b_1}(x_2)$ and thus $\mathbf{r}_1^{a_{j+1}}(x_2) <_{S_{\leq j+2}^{\leq n/2}} \mathbf{w}_1^{b_1}(x_2)$. A similar analysis to the base case shows that if a_{j+1} is not orthogonal to $b_{l'}$ for any $l' \in [n/2]$, then $\mathbf{r}_1^{a_{j+2}}(x_2) <_{S_{\leq j+2}^{\leq n/2}} \mathbf{w}_1^{b_1}(x_2)$. On the other hand, if a_{j+1} is orthogonal to some $b_{l'}$, it can be easily seen that the cross edges in $(X_{\leq j+2}^{\leq n/2}, S_{\leq j+2}^{\leq n/2})$ are precisely the cross edges in $(X_{\leq j+1}^{\leq n/2}, S_{\leq j+1}^{\leq n/2})$. Thus, there are no cross edges in $(X_{j_1+1}^{\leq n/2}, S_{j_1+1}^{\leq n/2})$, where $j_1 = j' + 1$. Finally, if $a_{j'}$ is orthogonal to $b_{l'}$ for some $j' \in [j]$ and $l' \in [n/2]$, the statement follows easily.

The desired result follows. QED.

We next have three lemmas that each is symmetric to Lemma 9.16, item 2 of Lemma 9.17, and item 2 of Lemma 9.18, respectively. The proof of these lemmas are analogous to previous lemmas, and are omitted here.

Lemma 9.19. For any $j, l \in [n/2]$, consider the rf-poset $\overline{\mathcal{Q}}_j^l = (X_j^l, \overline{\mathcal{Q}}_j^l, \mathbf{RF}_j^l)$, where $\overline{\mathcal{Q}}_j^l$ has a single cross edge $\mathbf{w}_1^{b_l}(x_2) <_{\overline{\mathcal{Q}}_j^l} \mathbf{r}_1^{a_j}(x_2)$. Let $\overline{\mathcal{S}}_j^l = (X_j^l, \overline{\mathcal{S}}_j^l, \mathbf{RF}_j^l)$ be the closure of $\overline{\mathcal{Q}}_j^l$. We have $\mathbf{r}_1^{b_l}(x_1) <_{\overline{\mathcal{S}}_j^l} \mathbf{w}_1^{a_j}(x_1)$ iff a_j and b_l are not orthogonal.

Lemma 9.20. For any $j \in [n/2]$ and $l \geq 2$, consider the rf-poset $\overline{\mathcal{Q}}_j^{\geq l-1} = (X_j^{\geq l-1}, \overline{\mathcal{Q}}_j^{\geq l-1}, \mathbf{RF}_j^{\geq l-1})$, where $\overline{\mathcal{Q}}_j^{\geq l-1}$ has a single cross edge $\mathbf{w}_l^{b_{n/2}}(x_2) <_{\overline{\mathcal{Q}}_j^{\geq l-1}} \mathbf{r}_1^{a_j}(x_2)$. Let $\overline{\mathcal{S}}_j^{\geq l-1} = (X_j^{\geq l-1}, \overline{\mathcal{S}}_j^{\geq l-1}, \mathbf{RF}_j^{\geq l-1})$ be the closure of $\overline{\mathcal{Q}}_j^{\geq l-1}$. If a_j is orthogonal to $b_{l'}$, for some $l' \in [l]$, then there are no cross edges in $(X_j^{l_2-1 \geq l' \geq l-1}, S_j^{l_2-1 \geq l' \geq l-1})$, where l_2 is the largest l' such that a_j is orthogonal to b_{l_2} .

Lemma 9.21. For any $j \geq 2$ with $j > 1$, consider the rf-poset $\overline{\mathcal{Q}}_{\geq j-1}^1 = (X_{\geq j-1}^1, \overline{\mathcal{Q}}_{\geq j-1}^1, \mathbf{RF}_{\geq j-1}^1)$, where $\overline{\mathcal{Q}}_{\geq j-1}^1$ has a single cross edge $\mathbf{w}_1^{b_{n/2}}(x_2) <_{\overline{\mathcal{Q}}_{\geq j-1}^1} \mathbf{r}_1^{a_{n/2}}(x_2)$. Let $\overline{\mathcal{S}}_{\geq j-1}^1 = (X_{\geq j-1}^1, \overline{\mathcal{S}}_{\geq j-1}^1, \mathbf{RF}_{\geq j-1}^1)$ be the closure of $\overline{\mathcal{Q}}_{\geq j-1}^1$. If $a_{j'}$ is orthogonal to $b_{l'}$ for some $j' \in [j]$ and $l' \in [n/2]$, then there are no cross edges in $(X_{j_2-1 \geq j' \geq j-1}^1, \overline{\mathcal{S}}_{j_2-1 \geq j' \geq j-1}^1)$, where j_2 is the largest j' such that $a_{j'}$ is orthogonal to $b_{l'}$, for some $l' \in [n/2]$.

Using Lemma 9.16, Lemma 9.17, Lemma 9.18 we can now prove Lemma 9.15.

Proof of Lemma 9.15. We show that $\mathcal{P}$ is realizable iff there exist $j, l \in [n/2]$ such that a_j is orthogonal to b_l . We prove each direction separately.

($\Rightarrow$). Since $\mathcal{P}$ is realizable, by Remark 9.2, we have that the closure of $\mathcal{P}$ exists. Let $\mathcal{K} = (X, K, \text{RF})$ be the closure of $\mathcal{P}$. Let $\mathcal{Q} = (X, Q, \text{RF})$ where Q has a single cross edge $\mathbf{w}_1^{a_1}(x_1) <_Q \mathbf{r}_1^{b_1}(x_1)$, and $\mathcal{S} = (X, S, \text{RF})$ be the closure of $\mathcal{Q}$. Note that $P \subseteq Q$ and thus $K \subseteq S$, and we have that $\mathbf{w}_1^{a_{n/2}}(x_2) \not<_S \mathbf{r}_1^{b_{n/2}}(x_2)$. If $\mathbf{w}_1^{a_{n/2}}(x_1) <_S \mathbf{w}_1^{b_{n/2}}(x_1)$, by Lemma 9.16, we have that $a_{n/2}$ is orthogonal to $b_{n/2}$. Otherwise, if $\mathbf{w}_1^{a_{n/2}}(x_1) <_S \mathbf{r}_1^{b_1}(x_1)$, by item 1 of Lemma 9.17, we have that $a_{n/2}$ is orthogonal to b_l , for some $l \in [n/2]$. Finally, if $\mathbf{w}_1^{a_{n/2}}(x_1) \not<_S \mathbf{r}_1^{b_1}(x_1)$, by item 1 of Lemma 9.18, we have that a_j is orthogonal to b_l , for some $j \leq n/2 - 1$ and $l \in [n/2]$.

($\Leftarrow$). Let $\mathcal{Q} = (X, Q, \text{RF})$ where Q has a single cross edge $\mathbf{w}_1^{a_1}(x_1) <_{Q_{\leq n/2}^{\leq j+1}} \mathbf{r}_1^{b_1}(x_1)$, and $\overline{\mathcal{Q}} = (X, \overline{Q}, \text{RF})$ where $\overline{Q}$ has a single cross edge $\mathbf{w}_1^{b_{n/2}}(x_2) <_{\overline{Q}} \mathbf{r}_1^{a_{n/2}}(x_2)$. Let $\mathcal{S} = (X, S, \text{RF})$ and $\overline{\mathcal{S}} = (X, \overline{S}, \text{RF})$ be the closures of $\mathcal{Q}$ and $\overline{\mathcal{Q}}$, respectively. We argue that $(X, S \cup \overline{S})$ is a poset. Note that this implies the lemma, as $(X, S \cup \overline{S}, \text{RF})$ is the closure of $\mathcal{P}$.

Let (j_1, l_1) (resp., (j_2, l_2)) be the lexicographically smallest (resp., largest) pair of integers in $[n/2] \times [n/2]$ such that a_{j_1} is orthogonal to b_{l_1} (resp., a_{j_2} is orthogonal to b_{l_2}). We assume wlog that $j_1 = j_2 = j$ and $l_1 = l_2 = l$, as in any other case, the relation $S \cup \overline{S}$ is strictly smaller. First, assume that $(X_j^l, S_j^l \cup \overline{S}_j^l)$ is a poset. By item 2 of Lemma 9.17, we have that there are no cross edges in $\mathcal{S}_j^{l+1 \leq l' \leq n/2}$, while by Lemma 9.20, we have that there are no cross edges in $\overline{\mathcal{S}}_j^{l-1 \geq l' \geq 1}$. Similarly, by item 2 of Lemma 9.18, we have that there are no cross edges in $\mathcal{S}_{j+1 \leq j'}^{\leq n/2}$, while by Lemma 9.21, we have that there are no cross edges in $\overline{\mathcal{S}}_{j-1 \geq j'}^{\geq 1}$. It follows that $(X, S \cup \overline{S})$ is a poset.

It remains to argue that $(X_j^l, S_j^l \cup \overline{S}_j^l)$ is a poset. The relation $S_j^l \cup \overline{S}_j^l$ is the transitive closure of a relation that consists of the total orders $\sigma_A|X_j^l$ and $\sigma_B|X_j^l$, together with the following relations, for each $i \in [D]$.

$$\begin{aligned} \mathbf{w}_i^{a_j}(x_1) <_{S_j^l} \mathbf{w}_i^{b_l}(x_1) \quad \text{and} \quad \mathbf{w}_i^{a_j}(x_3) <_{S_j^l} \mathbf{w}_i^{b_l}(x_3) \\ \mathbf{w}_i^{b_l}(x_2) <_{\overline{S}_j^l} \mathbf{w}_i^{a_j}(x_2) \quad \text{and} \quad \mathbf{w}_i^{b_l}(x_6) <_{\overline{S}_j^l} \mathbf{w}_i^{a_j}(x_6) \end{aligned} \tag{9.27}$$

It is straightforward to verify that if $S_j^l \cup \overline{S}_j^l$ has a cycle then, for some $i \in [D]$, we have

$$\mathbf{w}_i^{a_j}(x_2) <_{\tau_A} \mathbf{w}_i^{a_j}(x_1) \quad \text{and} \quad \mathbf{w}_i^{b_l}(x_1) <_{\tau_B} \mathbf{w}_i^{b_l}(x_2) . \tag{9.28}$$

Thus by construction, $a_j[i] = 1$ and $b_l[i] = 1$, which contradicts the fact that a_j is orthogonal to b_l .

The desired result follows. QED.

This concludes Lemma 9.15, as any algorithm for rf-poset realizability on the above instances that runs in $O((n \cdot D)^{2-\epsilon})$ time also solves OV in $O(n^{2-\epsilon} \cdot D^{O(1)})$ time. Although the full proof is rather technical, the correctness is conceptually straightforward. We illustrate the key idea on the example of Figure 9.2, where we perform closure operations by inserting new orderings in a partial order $<$.

By construction, we have $\mathbf{w}_1^{a_1}(x_1) < \mathbf{r}_1^{b_1}(x_1)$, which signifies testing the first coordinate of vectors a_1 and b_1 . Note that $a_1[1] \cdot b_1[1] = 1$, for which our encoding guarantees that eventually $\mathbf{r}_1^{a_1}(x_2) < \mathbf{w}_1^{b_1}(x_2)$. Indeed, since $\mathbf{w}_1^{a_1}(x_1) < \mathbf{r}_1^{b_1}(x_1)$, by closure we also have $\mathbf{w}_1^{a_1}(x_1) < \mathbf{w}_1^{b_1}(x_1)$. In turn, this leads to $\mathbf{w}_1^{a_1}(x_2) < \mathbf{w}_1^{b_1}(x_2)$, and by closure, we also have $\mathbf{r}_1^{a_1}(x_2) < \mathbf{w}_1^{b_1}(x_2)$. This leads to $\mathbf{w}^{a_1}(x_4) < \mathbf{r}^{b_1}(x_4)$, and by closure, we have $\mathbf{w}^{a_1}(x_4) < \mathbf{w}^{b_2}(x_4)$. This last ordering leads to $\mathbf{w}_1^{a_1}(x_1) < \mathbf{r}_1^{b_2}(x_1)$, which signifies testing the first coordinate of vectors a_1 and b_2 , i.e., moving with the next vector of B .

The process for a_1 and b_2 is similar to a_1 and b_1 , as the two vectors are found not orthogonal already in the first coordinate. As previously, we eventually arrive at $\mathbf{r}_1^{a_1}(x_2) < \mathbf{w}_1^{b_2}(x_2)$. Note that this leads to $\mathbf{w}^{a_1}(x_5) < \mathbf{w}^{b_2}(x_5)$, and by closure, we have $\mathbf{r}^{a_1}(x_5) < \mathbf{w}^{b_2}(x_5)$. In turn, this leads to $\mathbf{w}^{a_1}(x_7) < \mathbf{w}^{b_1}(x_7)$, and by closure, we have $\mathbf{r}^{a_2}(x_7) < \mathbf{w}^{b_1}(x_7)$. This last ordering leads to $\mathbf{w}_1^{a_2}(x_1) < \mathbf{r}_1^{b_1}(x_1)$, which signifies testing the first coordinate of vectors a_2 and b_1 , i.e., moving with the next vector of A and the first vector of B .

The process for a_2 and b_1 is initially different than before, as $a_2[1] \cdot b_1[1] = 0$, i.e., the test on the first coordinate does not deem a_2 and b_1 not orthogonal. By closure, the ordering $\mathbf{w}_1^{a_2}(x_1) < \mathbf{r}_1^{b_1}(x_1)$ leads to $\mathbf{w}_1^{a_2}(x_1) < \mathbf{w}_1^{b_1}(x_1)$. This leads to $\mathbf{w}_2^{a_2}(x_3) < \mathbf{r}_1^{b_1}(x_3)$, and by closure, we have $\mathbf{w}_2^{a_2}(x_3) < \mathbf{w}_2^{b_1}(x_3)$. This leads to $\mathbf{w}_2^{a_2}(x_1) < \mathbf{r}_2^{b_1}(x_1)$, which signifies testing the second coordinate of vectors a_2 and b_1 . As $a_2[2] \cdot b_2[2] = 1$, the two vectors are discovered as non-orthogonal, which is captured by an eventual ordering $\mathbf{r}_2^{a_2}(x_2) < \mathbf{w}_2^{b_1}(x_2)$. This ordering which witnesses non-orthogonality is propagated downwards to the first coordinate, i.e., $\mathbf{r}_1^{a_2}(x_2) < \mathbf{w}_1^{b_1}(x_2)$. This propagation is made by events on variable x_6 . Indeed, first note that, as $\mathbf{r}_2^{a_2}(x_2) < \mathbf{w}_2^{b_1}(x_2)$, we also have $\mathbf{w}_2^{a_2}(x_6) < \mathbf{w}_2^{b_1}(x_6)$, and by closure, we have $\mathbf{r}_1^{a_2}(x_6) < \mathbf{w}_2^{b_1}(x_6)$. This leads to $\mathbf{w}_1^{a_2}(x_2) < \mathbf{w}_1^{b_1}(x_2)$, and by closure, $\mathbf{r}_1^{a_2}(x_2) < \mathbf{w}_1^{b_1}(x_2)$, which marks the two vectors as non-orthogonal. This leads to $\mathbf{w}^{a_2}(x_4) < \mathbf{r}^{b_1}(x_4)$, and by closure, we have $\mathbf{w}^{a_2}(x_4) < \mathbf{w}^{b_2}(x_4)$. This last ordering leads to $\mathbf{w}_1^{a_2}(x_1) < \mathbf{r}_1^{b_2}(x_1)$, which signifies testing the first coordinate of vectors a_2 and b_2 , i.e., moving with the next vector of B .

The process for a_2 and b_2 is initially similar to the previous case, as $a_2[1] \cdot b_2[1] = 0$. However, because we also have $a_2[2] \cdot b_2[2] = 0$, we will *not* order $\mathbf{r}_2^{a_2}(x_2) < \mathbf{w}_2^{b_2}(x_2)$, and the closure will terminate after ordering $\mathbf{w}_2^{a_2}(x_1) < \mathbf{w}_2^{b_2}(x_1)$. Since no cyclical orderings were introduced, the closure of $\mathcal{P}$ exists, and by Lemma 9.9, $\mathcal{P}$ is realizable. Finally, observe that *if* we eventually had $\mathbf{r}_1^{a_2}(x_2) < \mathbf{w}_1^{b_2}(x_2)$ (signifying that a_2 and b_2 are not orthogonal,

hence there is no orthogonal pair in $A \times B$), this would create a cycle with the ordering $\mathbf{w}_1^{b_2}(x_2) <_P \mathbf{r}_1^{a_2}(x_2)$, and by Remark 9.2, $\mathcal{P}$ would *not* be realizable.

Reduction to dynamic data-race prediction. Consider an instance of the rf-poset $\mathcal{P} = (X, P, \text{RF})$ constructed in the above reduction, and we construct a trace σ and two events $e_1, e_2 \in \text{Events}_\sigma$ such that $\mathcal{P}$ is realizable iff (e_1, e_2) is a predictable data race of σ . The trace σ consists of two threads t_A, t_B and two local traces τ'_A and τ'_B such that τ'_A and τ'_B contain the events of t_A and t_B , respectively. Each of τ'_A and τ'_B is identical to τ_A and τ_B of $\mathcal{P}$, respectively, with some additional events inserted in it. In particular, besides the variables $x_i, i \in [7]$ that appear in the events of X , we introduce one variable y and one lock ℓ . For the event set, we have

$$\begin{aligned} \text{Events}_\sigma = & X \cup \{\mathbf{w}(y), \mathbf{r}(y)\} \cup \{\mathbf{acq}_A(\ell), \mathbf{rel}_A(\ell)\} \cup \\ & \{\mathbf{acq}_B(\ell), \mathbf{rel}_B(\ell)\} \cup \{\mathbf{w}(z), \mathbf{r}(z)\} . \end{aligned} \quad (9.29)$$

The local traces τ'_A and τ'_B are constructed as follows.

1. For τ'_A , we insert an empty critical section $\mathbf{acq}_A(\ell), \mathbf{rel}_A(\ell)$ right after $\mathbf{w}_1^{a_1}(x_1)$. Additionally, we insert the read event $\mathbf{r}(y)$ right before $\mathbf{r}_1^{a_{n/2}}(x_2)$, and the event $\mathbf{r}(z)$ as the last event of τ'_A .
2. For τ'_B , we insert the write event $\mathbf{w}(y)$ right after $\mathbf{w}_1^{b_{n/2}}(x_2)$. Additionally, we insert $\mathbf{w}(z)$ right after $\mathbf{r}_1^{b_1}(x_1)$, and surround these two events with $\mathbf{acq}_B(\ell), \mathbf{rel}_B(\ell)$.

Finally, we obtain σ as $\sigma = \tau'_B \circ \tau'_A$, i.e., the two local traces are executed sequentially and there is no context switching. The task is to decide whether $(\mathbf{w}(z), \mathbf{r}(z))$ is a predictable data race of σ .

Proof of Theorem 9.4. We complete the proof of Theorem 9.4 by showing that, in the above construction, $\mathcal{P}$ has a closure iff $(\mathbf{w}(z), \mathbf{r}(z))$ is a data race of σ , witnessed by a correct reordering σ^* . Observe that $\mathcal{Q}$ has 2 threads, and is thus tree-inducible. Due to Lemma 9.9, it suffices to show that $\mathcal{P}$ is realizable iff $(\mathbf{w}(z), \mathbf{r}(z))$ is a data race of σ .

($\Rightarrow$). Assume that $\mathcal{P}$ is realizable and σ' is a witness trace. We construct the correct reordering σ^* as follows. We have $\text{Events}_{\sigma^*} = \text{Events}_\sigma \setminus \{\mathbf{rel}_2(\ell), \mathbf{w}(z), \mathbf{r}(z)\}$. We make σ^* identical to σ' , i.e., $\sigma^*|X = \sigma'$. For the events $\text{Events}_{\sigma^*} \setminus X$, we make (i) $\mathbf{w}(y)$ appear right after $\mathbf{w}_1^{b_{n/2}}(x_2)$, (ii) $\mathbf{r}(y)$ appear right before $\mathbf{r}_1^{a_{n/2}}(x_2)$, (iii) $\mathbf{acq}_A(\ell), \mathbf{rel}_A(\ell)$ appear right after $\mathbf{w}_1^{a_1}(x_1)$, and (iv) $\mathbf{acq}_B(\ell)$ appear last in σ^* . Observe that σ^* is a correct reordering in which $\mathbf{w}(z)$ and $\mathbf{r}(z)$ are enabled, hence σ^* is a witness of the data race $(\mathbf{w}(z), \mathbf{r}(z))$.

($\Leftarrow$). Assume that $(\mathbf{w}(z), \mathbf{r}(z))$ is a predictable data race of σ , and let σ^* be a correct reordering of σ witnessing the data race. We construct a trace σ' as $\sigma' = \sigma^*|X$, and argue that σ' realizes $\mathcal{P}$. Clearly $\text{RF}_{\sigma'} = \text{RF}$. To see that σ' is a linearization of (X, P) , it suffices to argue that

$$\mathbf{w}_1^{b_{n/2}}(x_2) <_{\sigma'} \mathbf{r}_1^{a_{n/2}}(x_2) \quad \text{and} \quad \mathbf{w}_1^{a_1}(x_1) <_{\sigma'} \mathbf{r}_1^{b_1}(x_1) \quad (9.30)$$

The first ordering follows from the fact that

$$\mathbf{w}_1^{b_{n/2}}(x_2) <_{\sigma^*} \mathbf{w}(y) <_{\sigma^*} \mathbf{r}_y <_{\sigma^*} \mathbf{r}_1^{a_{n/2}}(x_2) . \quad (9.31)$$

For the second ordering, observe that $\text{Events}_{\sigma^*} = \text{Events}_{\sigma} \setminus \{\text{rel}_2(\ell), \mathbf{w}(z), \mathbf{r}(z)\}$. Hence,

$$\mathbf{w}^{a_1}(x_1) <_{\sigma^*} \text{rel}_1(\ell) <_{\sigma^*} \text{acq}_2(\ell) <_{\sigma^*} \mathbf{r}_a^{b_1}(x_1) . \quad (9.32)$$

The desired result follows. QED.

9.5 WITNESSES IN SMALL DISTANCE

The results in the previous sections neglect information provided by the input trace σ about constructing a correct reordering that witnesses the data race. Indeed, our hardness results show that, in the worst case, the orderings in σ provide no help. However, in practice when a data race exists, a witness trace σ^* can be constructed that is similar to σ .

Problem 9.4. The distance-bounded realizability problem of feasible trace ideals. Given a natural number $\mathbf{b}$, a trace σ and a feasible trace ideal X of σ , the solution to the $\mathbf{b}$ -distance-bounded realizability problem is **False** if X is not realizable, **True** if there is a witness σ^* that realizes X such that $\delta(\sigma, \sigma^*) \leq \mathbf{b}$, and can be any answer (**True/False**) if X is realizable but any witness σ^* that realizes X is such that $\delta(\sigma, \sigma^*) > \mathbf{b}$.

We remark that this formulation is that of a *promise problem* [109]. We are interested in the case where $\mathbf{b} = O(1)$. There exists a straightforward algorithm that operates in $O(|X|^{2^{\mathbf{b}}})$ time. The algorithm iterates over all possible subsets of pairs of conflicting write and lock-acquire events that have size at most $\mathbf{b}$, and tries all possible combinations of conflicting-write reversals in that set. Theorem 9.5 is based on the following lemma, which states that the problem can be solved much faster when k is also constant.

Lemma 9.22. Consider a natural number $\mathfrak{b}$, a trace σ over $\mathcal{N}$ events and $\mathcal{T}$ threads, and a feasible trace ideal X of σ . The $\mathfrak{b}$ -distance-bounded realizability problem for X can be solved in $O(\mathcal{T}^{\mathfrak{b}+O(1)} \cdot \mathcal{N})$ time.

Proof of Theorem 9.5. By Lemma 9.22, given a trace ideal X of σ , we can solve the $\mathfrak{b}$ -distance-bounded realizability problem for X in $O(\mathcal{N})$ time. The proof then follows by Lemma 9.2 and Lemma 9.3, as to decide whether (e_1, e_2) is a predictable data race of σ , it suffices to examine $O(1)$ trace ideals of σ . QED.

In the remaining of this section we prove Lemma 9.22. We first define the notion of read extensions of graphs. Afterwards, we present the algorithm for the lemma, and show its correctness and complexity.

Read extensions. Consider a digraph $G = (X, E)$ where X is a set of events. Given two events $e_1, e_2 \in G$, we write $e_1 \rightsquigarrow e_2$ to denote that e_2 is reachable from e_1 . We call G *write-ordered* if for every two distinct conflicting write or lock-acquire events $\mathbf{w}_1, \mathbf{w}_2 \in \text{WtAcqs}_X$, we have $\mathbf{w}_1 \rightsquigarrow \mathbf{w}_2$ or $\mathbf{w}_2 \rightsquigarrow \mathbf{w}_1$ in G . Given an acyclic write-ordered graph $G_1 = (X, E_1)$, the *read extension* of G_1 is the digraph $G_2 = (X, E_2)$ where $E_2 = E_1 \cup A \cup B$, where the sets A and B are defined as follows.

$$A = \{(\mathbf{r}, \mathbf{w}) \in \text{RdRels}_X \times \text{WtAcqs}_X \mid \mathbf{r} \asymp \mathbf{w} \text{ and } (\text{RF}_\sigma(\mathbf{r}), \mathbf{w}) \in E_1\} , \quad (9.33)$$

$$B = \{(\mathbf{w}, \mathbf{r}) \in \text{WtAcqs}_X \times \text{RdRels}_X \mid \mathbf{r} \asymp \mathbf{w} \text{ and } (\mathbf{w}, \text{RF}_\sigma(\mathbf{r})) \in E_1\} . \quad (9.34)$$

A fast algorithm for distance-bounded rf-poset realizability. Let $\mathcal{P} = (X, P, \text{RF})$ be the canonical rf-poset of X , and the task is to decide the realizability of $\mathcal{P}$ with $\mathfrak{b}$ reversals. We describe a recursive algorithm for solving the problem for some rf-poset $\mathcal{Q} = (X, Q, \text{RF})$ with $\mathfrak{b}'$ reversals, for some $\mathfrak{b}' \leq \mathfrak{b}$, where initially $Q = P$ and $\mathfrak{b}' = \mathfrak{b}$.

Algorithm and correctness. Consider the set

$$C = \{(\mathbf{w}_1, \mathbf{w}_2) \in \text{WtAcqs}_X \times \text{WtAcqs}_X \mid \mathbf{w}_1 \asymp \mathbf{w}_2 \text{ and } \mathbf{w}_1 \parallel_Q \mathbf{w}_2 \text{ and } \mathbf{w}_1 <_{\text{tr}}^\sigma \mathbf{w}_2\} . \quad (9.35)$$

We construct a graph $G_1 = (X, E_1)$, where $E_1 = (\leq_{\text{TRF}}^\sigma \mid X) \cup C$. Note that G_1 is write-ordered. If it is acyclic, we construct the read extension G_2 of G_1 . Observe that if G_2 is acyclic then any linearization σ^* of G realizes $\mathcal{Q}$, hence we are done. Now consider that either G_1 or G_2 is not acyclic, and let $G = G_1$ if G_1 is not acyclic, otherwise $G = G_2$. Given a cycle Cyc of G , represented as a collection of edges, define the set of *cross-edges* of Cyc as

$\text{Cyc} \setminus Q$. Note that, since there are k threads, G has a cycle with $\leq k$ cross edges. In addition, any trace σ^* that realizes $\mathcal{Q}$ must linearize an rf-poset (X, Q_a, RF) where $a = (e_1, e_2)$ ranges over the cross-edges of Cyc . In particular, we take $Q_a = Q \cup \{b\}$, where

$$b = \begin{cases} (e_2, e_1), & \text{if } a \in \text{WtAcqs}_X \times \text{WtAcqs}_X \\ (\text{RF}(e_2), e_1), & \text{if } a \in \text{WtAcqs}_X \times \text{RdRels}_X \\ (e_2, \text{RF}(e_1)), & \text{if } a \in \text{RdRels}_X \times \text{WtAcqs}_X. \end{cases} \quad (9.36)$$

Observe that any such choice of b reverses the order of two conflicting write events or lock-acquire events in σ . Since there are $\leq \mathcal{T}$ cross edges in Cyc , there are $\leq \mathcal{T}$ such choices for Q_a . Repeating the same process recursively for the rf-poset (X, Q_a, RF) for $\mathcal{b}' - 1$ levels solves the $\mathcal{b}'$ -distance-bounded realizability problem for $\mathcal{Q}$. Since initially $\mathcal{b}' = \mathcal{b}$ and $Q = P$, this process solves the same problem for $\mathcal{P}$ and thus for X .

Complexity. The recursion tree above has branching $\leq \mathcal{T}$ and depth $\leq \mathcal{b}$, hence there will be at most $\mathcal{T}^{\mathcal{b}}$ recursive instances. We now provide some lower-level algorithmic details which show that each instance can be solved in $O(\mathcal{T}^{O(1)} \cdot \mathcal{N})$ time. The main idea is that each of the graphs G_1 and G_2 have a sparse transitive reduction [116] of size $O(\mathcal{T} \cdot \mathcal{N})$, and thus each graph can be analyzed in $O(\mathcal{T} \cdot \mathcal{N})$ time.

For the graph G_1 , we construct a sparse graph G'_1 that preserves the reachability relationships of G_1 as follows. We traverse the trace σ top-down. For the current event $\mathbf{w}$ such that $\mathbf{w} \in \text{WtAcqs}_X$, for every $i \in [k]$, let $\mathbf{w}_1$ be the last event of thread i which precedes $\mathbf{w}$ in σ , and such that $\mathbf{w}_1 \in \text{WtAcqs}_X$ and $\mathbf{w} \succ \mathbf{w}_1$. Let $\mathbf{w}'_1$ be the last event of σ such that $\mathbf{w}'_1 \leq_{\text{TO}} \mathbf{w}_1$ and $\mathbf{w} \not\leq_Q \mathbf{w}'_1$ (note that possibly $\mathbf{w}'_1 = \mathbf{w}_1$). If such $\mathbf{w}'_1$ exists, since X is an ideal, we have $\mathbf{w}'_1 \in X$. We introduce the edge $\mathbf{w}'_1 \rightarrow \mathbf{w}$ in G'_1 .

Similarly, for the graph G_2 , we construct a sparse graph G'_2 that preserves the reachability relationships of G_2 as follows. We iterate over all read and lock-release events $\mathbf{r} \in X$. For each such $\mathbf{r}$ and thread $i \in [k]$, we insert two edges $(\mathbf{w}_2, \mathbf{r})$ and $(\mathbf{r}, \mathbf{w}_3)$ in G'_2 , where $\mathbf{w}_2$ (resp., $\mathbf{w}_3$) is the latest predecessor (resp., earliest successor) of $\text{RF}(\mathbf{r})$ in thread i that conflicts with $\mathbf{r}$.

We now argue that all events $\mathbf{w}_1$, $\mathbf{w}'_1$, $\mathbf{w}_2$ and $\mathbf{w}_3$ above can be identified in $O(\mathcal{N})$ total time. For every $i \in [\mathcal{T}]$ and memory location x , we construct a total order τ_i^x of all write events or lock-acquire events on location x of thread t_i . Clearly all such total orders can be constructed in $O(\mathcal{N})$ time.

1. Events $\mathbf{w}_1$: as we traverse σ top-down, we simply remember for each memory location x , the last write event or lock-acquire event on location x for each thread. Hence, the

total time for identifying all such events is $O(\mathcal{N})$.

2. Events $\mathbf{w}'_1$: Given the event $\mathbf{w}_1$ on memory location x , we simply traverse the total order τ_i^x from $\mathbf{w}'_1$ backwards until we find either (i) an event $\mathbf{w}'_1$ with the desired properties, or (ii) an event $\mathbf{w}''_1$ that has been examined before when inserting edges to thread t_i . In the first case, we simply add the edge $(\mathbf{w}'_1 \rightarrow \mathbf{w})$ as described above. In the second case we do nothing, as the desired ordering is already present due to transitivity through a write event $\mathbf{w}'$ of thread t_i that has been examined before (thus $\mathbf{w}' <_{\text{TO}} \mathbf{w}$). Hence, every event in every total order τ_i^x is examined $O(1)$ times, thus the total time for identifying all $\mathbf{w}'_1$ events is $O(\mathcal{N})$.
3. Events $\mathbf{w}_2$ and $\mathbf{w}_3$: after we have constructed G'_1 , each pair of such events can be retrieved from G'_1 in $O(1)$ time. Hence the total time for identifying all such events is $O(\mathcal{N})$.

Finally, the above process creates graphs G'_1 and G'_2 that have $O(\mathcal{T}^{O(1)} \cdot \mathcal{N})$ edges. Detecting a cycle Cyc' in either G'_1 or G'_2 is done by a simple DFS, which takes linear time in the number of edges. Converting Cyc' to a cycle Cyc such that Cyc has at most $\mathcal{T}$ cross edges can be done in $O(|\text{Cyc}'|) = O(\mathcal{T}^{O(1)} \cdot \mathcal{N})$ time, by removing multiple edges whose endpoints are TO-ordered.

CHAPTER 10: DETECTING CONFLICT SERIALIZABILITY VIOLATIONS USING VECTOR CLOCKS

Writing correct multi-threaded programs is extremely difficult. It is the class of software that is most prone to errors. Reasoning about multi-threaded programs is notoriously challenging due to the inherent nondeterminism that arises from thread scheduling in such systems. If the program satisfies certain fundamental properties then reasoning about them becomes easier, and if such properties are violated then it is often symptomatic of more serious bugs in the software. *Atomicity* is one such classical concurrency property, which guarantees that a programmer reasoning about a concurrent program can assume that atomic blocks of code can be executed sequentially without any context switches in between. Atomicity allows programmers to reason about atomic blocks without worrying about the effects of other threads. Unfortunately, violation of atomicity specifications is quite common and is the root cause in a majority of real-world bugs [117–123].

Various approaches to identifying atomicity violations have been explored. Static analysis based approaches for atomicity checking are usually conservative, computationally expensive, and often rely on user annotations, like type annotations [124–129]. The advantage of static analysis approaches is that they may successfully *prove* that a program satisfies all its atomicity requirements. Dynamic analysis for atomicity violations, on the other hand, have the advantage that they are fully automated and are computationally less expensive [38,61,118,130–132]. Though they cannot prove that a program satisfies its atomicity specification, dynamic analysis can be used to check if an observed trace is witness to the violation of atomicity. Given their scalability, dynamic analysis techniques for detecting atomicity violations have proved to be very useful in practice.

In this chapter, we will focus on sound and precise dynamic analyses. Most sound and precise dynamic analyses [38,61,132] for atomicity violation are based on checking the *conflict serializability* of an observed program execution as defined in Section 2.4.2. Determining if an execution is conflict serializable can be reduced to checking for the existence of a cycle in a graph called the transaction graph. The transaction graph has atomic blocks (a.k.a. transactions) as vertices, and edges between blocks that contain non-commutable events. A path from atomic block A to B indicates that A must be executed before B in a serial execution, and so a cycle in such a graph indicates that the execution is not equivalent to a serial one. All prior sound and precise dynamic analyses for conflict serializability [38,61] rely on this idea and thus have an asymptotic complexity of cubic time — each new event of the trace requires updating the transaction graph, and checking for cycles; the number of edges can be quadratic in the number of events, giving a quadratic processing time *per*

event.

The central question motivating this chapter is the following: Is a cubic running time necessary for checking conflict serializability? Or are there sub-cubic algorithms for this problem? The main result of this chapter is a new, *linear time* algorithm for checking conflict serializability.

For other concurrency specifications, like data race detection, that admit sound and precise linear time algorithms, the key to achieving an efficient algorithm is the use of vector clocks [13, 14, 37, 50]. Such algorithms rely on computing vector timestamps for events in a streaming fashion as the trace is generated, and using these timestamps to recover the causal order between a pair of events. However, generalizing such an algorithmic principle to conflict serializability checking is far from straightforward. This is because checking conflict serializability requires identifying causal orders between transactions (or atomic blocks) and not individual events. For this reason, Flanagan-Freund-Yi [38], in fact, dismiss the possibility of a vector clock based algorithm for conflict serializability checking:

“The traditional representation of clock vectors [50] is not applicable because our happens-before relation is over compound transactions and not individual operations.”

The challenge is to discover a way to associate a single timestamp with a transaction, even though new causal dependencies are discovered as each individual event in the trace is processed. This is further complicated by the following observation. Vector timestamps implicitly summarize the set of all events that must be ordered before. However, the set of transactions that must be executed before a transaction T might be known only well after all the events of T have been seen (see Example 10.2). These observations suggest that a scheme of assigning vector timestamps to transactions may only be computed if the algorithm makes multiple streaming passes over the trace, which may result in an algorithm that is not linear time.

We address these challenges by assigning vector timestamps to individual events in a trace. The induced order on events is then used to discover the ordering relationship between transactions, and thereby determining if a trace is conflict serializable. For a trace containing a bounded number of variables, threads, and locks, our algorithm, AERODROME, is a single pass, streaming algorithm that runs in linear time. As with standard vector clock algorithms, such as those used in data race detection [48, 52], our algorithm summarizes information in *vector clocks* and thus does not need to store the timestamp of all the events in the trace to detect serializability violations.

We have implemented AERODROME in our tool RAPID [86] and have compared its performance against Velodrome [38] on various benchmark programs. Atomicity specifications (i.e., which blocks of code should be regarded as atomic) are hard to come by. One naïve specification is to consider each method call to be atomic. Since often there is a `main` method for each thread, this means that the entire computation of each thread should be atomic. Programs are unlikely to satisfy such strong atomicity specifications, but running detection algorithms against these, gives us a baseline. We use such naïve specifications for some programs in our benchmark. For such benchmarks, conflict serializability is trivially violated in a small prefix of the observed trace. The resulting transaction graph is thus small, the overhead of maintaining vector clocks outweighs the benefits of a linear time algorithm, and Velodrome slightly outperforms AERODROME. For other programs in our benchmark, we use the more realistic atomicity specifications given in [61]. Here transactions consist of smaller blocks of code, and the resulting transaction graph has many transactions. For such examples, our algorithm significantly outperforms Velodrome. This suggests that on realistic atomicity specifications, the benefits of having a linear time algorithm can be significant.

The rest of the chapter is organized as follows. In Section 10.1, we discuss preliminary notations such as that of concurrent program traces and the definition of conflict serializability. In Section 10.2, we use motivating examples to illustrate the challenges involved in developing a linear time vector clock algorithm for dynamically checking conflict serializability. We then discuss an alternate characterization of conflict serializability in Section 10.3 that is amenable to an efficient vector clock algorithm. In Section 10.4, we discuss AERODROME, a single pass linear time vector clock algorithm for checking conflict serializability based on this characterization, which is also the main contribution of the chapter. Section 10.4 also discusses the correctness and complexity guarantees of the algorithm and optimizations for improving the performance of AERODROME. Our implementation of AERODROME in our tool RAPID and its performance evaluation on a suite of benchmark programs is discussed in Section 10.5. Some proofs and additional discussion can be found in the full version [133].

10.1 PRELIMINARIES

Fix a trace σ . We will say that a transaction $T \in \text{Transactions}_\sigma$ is *completed* in σ if either T is a singleton transaction, or the corresponding end transaction event $\langle \cdot, \text{end} \rangle \in \text{Events}_\sigma$. If T is not completed in σ , it is said to be *active*.

Our notion of atomicity allows for serial traces which respect the relative order of some events. We define this order next.

Definition 10.1 (Conflict-Happens-Before). Let σ be a trace. The *conflict-happens-before* order on the events of σ , denoted $\leq_{\text{CHB}}^\sigma$ is the smallest partial order on the set Events_σ such that

- $\leq_{\text{HB}}^\sigma \subseteq \leq_{\text{CHB}}^\sigma$, and
- for every two events $e, e' \in \text{Accesses}_\sigma$, if $e \succ e'$, then $(e, e') \in \leq_{\text{CHB}}^\sigma$

We often use $e \leq_{\text{CHB}}^\sigma e'$ to denote $(e, e') \in \leq_{\text{CHB}}^\sigma$.

Atomicity is closely related to the property of conflict serializability. Informally, this property requires that an execution be equivalent to a *serial* execution by commuting adjacent events that are not conflicting accesses or lock events (acquires and releases) accessing the same lock; an execution is serial if for every thread t in the trace and for every transaction T of thread t , there are no events of any other thread between the begin and end events of T . In this context, if two events e and e' are ordered by $\leq_{\text{CHB}}^\sigma$, then their order is the same in all equivalent executions. To capture conflict serializability, such a causal relationship needs to be lifted to transactions. Consider two transactions T and T' with events $e \in T$ and $e' \in T'$ such that $e \leq_{\text{CHB}}^\sigma e'$. If the goal in a serial execution is to schedule all events of T consecutively, given that e is before e' in all equivalent executions, it must be the case that *every* event of T should happen before each event of T' .

Let us now define the induced ordering between transactions.

Definition 10.2 (Transactional Happens Before). For a trace σ , the *transactional happens before* of σ is the following binary relation on $\text{Transactions}_\sigma$:

$$<_{\text{Txn}}^\sigma = \{(T, T') \in \text{Transactions}_\sigma \mid \exists e \in T, e' \in T', e \leq_{\text{CHB}}^\sigma e'\}.$$

In words, we say that transaction T must *happen before* transaction T' if $(T, T') \in <_{\text{Txn}}^\sigma$ (also denoted by $T <_{\text{Txn}}^\sigma T'$). Observe that $<_{\text{Txn}}^\sigma$ may not be a partial order, and may in fact contain cycles. We now present the definition of conflict serializability (which implies atomicity) from [38]; an equivalent definition was presented in Section 2.4.2.

Definition 10.3 (Conflict Serializability [38]). A trace σ is *conflict serializable* if there is no sequence of $k > 1$ distinct transactions $T_0, T_1 \dots T_{k-1}$ such that for every $0 \leq i \leq k-1$, we have $T_i <_{\text{Txn}}^\sigma T_{(i+1) \bmod k}$. If σ is not conflict serializable, then such a sequence $T_0, \dots, T_{k-1}$ is said to be a witness to the violation.

Observe that, for a trace σ that is not conflict serializable, it must have at least one non-singleton transaction, as otherwise $<_{\text{Txn}}^\sigma = \leq_{\text{CHB}}^\sigma$ which is acyclic.

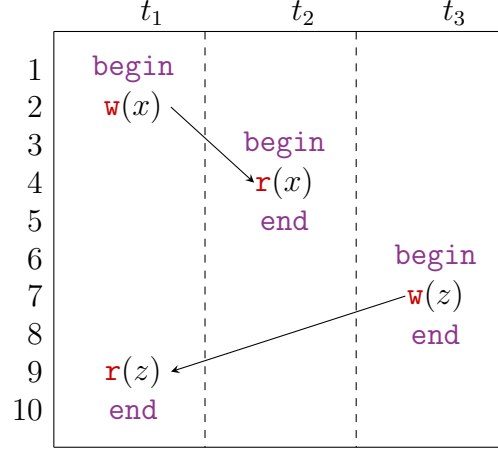


Figure 10.1: Trace ρ_1 . Taking T_i to be the transaction of thread t_i , we have $T_3 \prec_{\text{Txn}}^{\rho_1} T_1 \prec_{\text{Txn}}^{\rho_1} T_2$.

Example 10.1. Consider the trace ρ_1 in Figure 10.1. This trace is a sequence of 10 events, performed by three different threads t_1, t_2 and t_3 . In all our examples, we will use e_i to denote the i^{th} event in the trace. This trace has three transactions — transaction $T_1 = e_1 e_2 e_9 e_{10}$ is performed in t_1 , transaction $T_2 = e_3 e_4 e_5$ is performed in t_2 and transaction $T_3 = e_6 e_7 e_8$ is performed in t_3 . All pairs of events, both of which are performed by the same thread (such as (e_1, e_2) or (e_2, e_{10}) in ρ_1) are conflicting. In addition, (e_2, e_4) and (e_7, e_9) are conflicting pairs of events in ρ_1 and we use an explicit arrow ($\longrightarrow$) to depict such inter-thread conflicting pairs. We have $T_1 \prec_{\text{Txn}}^{\rho_1} T_2$ because $e_2 \leq_{\text{CHB}}^{\rho_1} e_4$ and $T_3 \prec_{\text{Txn}}^{\rho_1} T_1$ because $e_7 \leq_{\text{CHB}}^{\rho_1} e_9$. Also note that $\leq_{\text{CHB}}$ is a transitive order and thus $e_1 \leq_{\text{CHB}}^{\rho_1} e_5$ because $e_1 \leq_{\text{CHB}}^{\rho_1} e_2$, $e_2 \leq_{\text{CHB}}^{\rho_1} e_4$ and $e_4 \leq_{\text{CHB}}^{\rho_1} e_5$. Finally, the trace ρ_1 is conflict serializable and the equivalent serial execution is the sequence $\rho_1^{\text{serial}} = e_6 e_7 e_8 e_1 e_2 e_9 e_{10} e_3 e_4 e_5$, in which the order of transaction is $T_3 T_1 T_2$. Observe that the relative order of conflicting events in ρ_1^{serial} is the same as in the original trace ρ_1 .

Based on Definition 10.3, a cyclic dependency on transactions using $\prec_{\text{Txn}}^\sigma$ suggests that σ does not have an equivalent serial execution and hence the program does not satisfy its atomicity specification. Previous techniques [38, 61] for checking conflict serializability dynamically, rely on constructing a directed graph. The vertices in such a graph are the different transactions in the observed trace, the edges correspond to the order imposed by $\prec_{\text{Txn}}$ and checking violations of conflict serializability reduces to searching for a cycle in this graph. These algorithms run in time that is cubic in the length of the observed trace as they check for cycles each time a new edge is added in the graph, whose size is quadratic in the size of the trace.

10.2 CHALLENGES IN DESIGNING A VECTOR CLOCK ALGORITHM

Vector clocks have been very useful in designing linear time algorithms for dynamic analysis of multi-threaded systems [13, 14, 36, 37, 48, 48, 52]. The broad principle behind these algorithms, is to assign vector *timestamps* to events as the trace is generated/observed so that the ordering between these assigned timestamps captures causal ordering. Notice that, conflict serializability is defined in terms of the relation $\leq_{\text{Txn}}$ on transactions (Definition 10.3), and thus, the most straightforward vector clock algorithm would rely on assigning timestamps to *transactions* in such a way that the timestamp of transaction T_1 is less than or equal to timestamp of transaction T_2 if and only if $T_1 \leq_{\text{Txn}} T_2$. However, since a transaction is a *sequence of events* (and not a single event), the first challenge is figuring out how to assign and update timestamps of transactions when individual events are being continuously generated by the execution; this is one of the reasons why such algorithms were deemed impossible for atomicity in [38]. However, there is a deeper and more fundamental challenge with assigning timestamps to transactions, as illustrated in the following example.

Example 10.2. Consider again the trace ρ_1 in Figure 10.1. Notice that there is a “path” from T_3 to T_2 (via T_1) using $\leq_{\text{Txn}}^{\rho_1}$, even though T_3 starts after T_2 is completed in the trace ρ_1 . Further the discovery that T_3 has a path to T_2 can be made only after the event e_9 is generated in the trace, and at that point, *both* T_2 and T_3 have completed. This poses serious challenges when designing a vector clock algorithm. A vector clock algorithm assigning a timestamp to transaction T that is consistent with $\leq_{\text{Txn}}$, needs to know (explicitly or implicitly) the set of transactions that have a path to T ; this is because the algorithm needs to ensure that the timestamp assigned to T is ordered after the timestamps assigned to all these “predecessor” transactions. However, as transaction T_2 in trace ρ_1 illustrates, this may require knowing future events and transactions.

Example 10.2 illustrates that transactions T' that have a $\leq_{\text{Txn}}$ -path to a transaction T may only be determined by events that appear after T itself. This suggests that one is unlikely to get a linear time streaming algorithm that assigns timestamps to transactions for detecting atomicity violations.

Therefore, we explore the possibility of an algorithm that assigns timestamps to *events* (not transactions), but which can nonetheless enable checking conflict serializability. The first key question to address is *which relation among events should the timestamps try to capture implicitly?* Recall that, the relation $\leq_{\text{Txn}}$ (on transactions) is defined in terms of the relation $\leq_{\text{CHB}}$ (on events), and therefore, a natural first step to explore, is to see if computing $\leq_{\text{CHB}}$ is sufficient to detect atomicity violations.

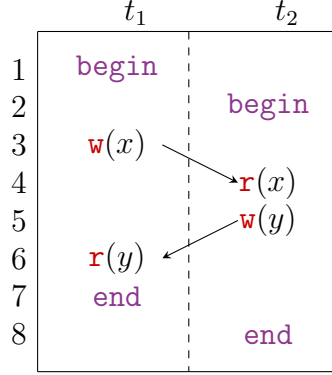


Figure 10.2: Trace ρ_2 . There is a cycle in the transaction graph that can be realized by a path using $\leq_{\text{CHB}}$ -edges that begins and ends in the same transaction.

Example 10.3. Consider the trace ρ_2 in Figure 10.2 with two transactions T_1 and T_2 in threads t_1 and t_2 respectively. Here, we have, $T_1 \leq_{\text{Txn}}^{\rho_2} T_2$ and $T_2 \leq_{\text{Txn}}^{\rho_2} T_1$, thus giving us a violation of conflict serializability with the sequence T_1, T_2 witnessing the violation. Now consider the following $\leq_{\text{CHB}}$ path in the trace — $e_1 \leq_{\text{CHB}}^{\rho_2} e_4 \leq_{\text{CHB}}^{\rho_2} e_5 \leq_{\text{CHB}}^{\rho_2} e_7$. This path, in fact, is symptomatic of the atomicity violation because it starts and ends in the same transaction (transaction T_1) and passes through another transaction (transaction T_2).

The atomicity violation in trace ρ_2 in Example 10.3 can be deduced based on the observation that there are 3 events e, f, g (e_1, e_5, e_7 in ρ_2 , specifically) such that $\text{txn}_{\rho_2}(e) = \text{txn}_{\rho_2}(g)$, $\text{txn}_{\rho_2}(e) \neq \text{txn}_{\rho_2}(f)$, and $e \leq_{\text{CHB}}^{\rho_2} f \leq_{\text{CHB}}^{\rho_2} g$. If we can prove that this is equivalent to Definition 10.3, then all we need to do is to compute (implicitly using vector clocks) the $\leq_{\text{CHB}}$ ordering. Unfortunately, this is not true, i.e., violations of conflict serializability cannot be detected by simply using $\leq_{\text{CHB}}$ ordering and searching for the above kind of $\leq_{\text{CHB}}$ paths. We illustrate this in the next example.

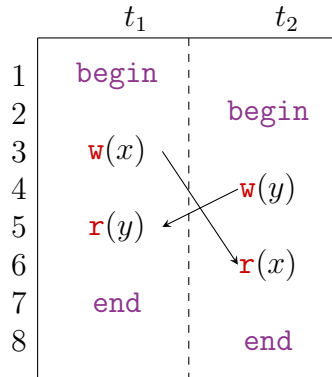


Figure 10.3: Trace ρ_3 . There is no $\leq_{\text{CHB}}$ path that starts and ends in the same transaction.

Example 10.4. Consider trace ρ_3 in Figure 10.3. As before, let T_1, T_2 be the two transactions by threads t_1 and t_2 respectively. Here, both $T_1 \prec_{\text{Txn}}^{\rho_3} T_2$ (because $e_3 \leq_{\text{CHB}}^{\rho_3} e_6$) and $T_2 \prec_{\text{Txn}}^{\rho_3} T_1$ (because $e_4 \leq_{\text{CHB}}^{\rho_3} e_5$), thus giving us a conflict serializability violation. However, there is no $\leq_{\text{CHB}}$ -path that starts and ends in the same transaction. If vector timestamps are used to compute $\leq_{\text{CHB}}$, then violations of conflict serializability cannot be detected by checking ordering of vector timestamps of events.

10.3 CHARACTERIZING CONFLICT SERIALIZABILITY VIOLATIONS USING RELATIONS ON EVENTS.

Example 10.4 demonstrates that $\leq_{\text{CHB}}$ is not the right relation on events to detect violations of conflict serializability. Then, what is the right relation to track? In order to identify that, we will first recast Definition 10.3 in terms of events.

Definition 10.4 (Path Through Transactions). Let σ be a trace and $e, f \in \text{Events}_\sigma$. We will say that there is a *path* from e to f *through transactions* in trace σ (denoted $e \xrightarrow{*}_\sigma f$), if there is a sequence of pairs of events $(e_1, f_1), (e_2, f_2), \dots, (e_k, f_k)$ ($k > 1$) such that (a) $e = e_1$ and $f = f_k$, (b) $\text{txn}_\sigma(e_i) = \text{txn}_\sigma(f_i)$, while $\text{txn}_\sigma(f_i) \neq \text{txn}_\sigma(e_{i+1})$, for every i , and (c) $f_i \leq_{\text{CHB}}^\sigma e_{i+1}$ for every $i < k$.

Using the notion of path between events through transactions, we can recast the notion of conflict serializability as follows.

Proposition 10.1. A trace σ is not conflict serializable if and only if there is a pair of events e, f such that $e \xrightarrow{*}_\sigma f$ and $f \leq_{\text{CHB}}^\sigma e$.

Proof. Follows from Definition 10.3 and the definition of $\xrightarrow{*}_\sigma$. QED.

We observe some properties about the relation $\xrightarrow{*}$ defined above, as they will be useful in establishing further results. Here, for two binary relations $R_1, R_2 \subseteq S \times S$ over a set S , we use $R_1 \circ R_2$ to denote the composition $\{(e, e') \mid \exists e'' \in S, (e, e'') \in R_1, (e'', e') \in R_2\}$.

Proposition 10.2. Let σ be a trace. $\xrightarrow{*}_\sigma$ composes with $\leq_{\text{CHB}}^\sigma$. That is, $\xrightarrow{*}_\sigma \circ \leq_{\text{CHB}}^\sigma \subseteq \xrightarrow{*}_\sigma$ and $\leq_{\text{CHB}}^\sigma \circ \xrightarrow{*}_\sigma \subseteq \xrightarrow{*}_\sigma$.

Proof (Sketch). Let $(e, f) \in \xrightarrow{*}_\sigma$ and $(f, g) \in \leq_{\text{CHB}}^\sigma$. Then, there is a sequence (of length $k > 1$) $(e_0, f_0) \dots (e_{k-1}, f_{k-1})$ such that for every $0 \leq i < k$, $\text{txn}_\sigma(e_i) = \text{txn}_\sigma(f_i)$ and $f_i \leq_{\text{CHB}}^\sigma e_{i+1 \bmod k}$. If $\text{txn}_\sigma(f) = \text{txn}_\sigma(g)$, then the sequence $(e_0, f_0) \dots (e_{k-1}, g)$ shows that $e \xrightarrow{*}_\sigma g$. Otherwise, the $k + 1$ length sequence $(e_0, f_0) \dots (e_{k-1}, f_{k-1}), (g, g)$ shows that $e \xrightarrow{*}_\sigma g$. The proof for left composition is similar. QED.

Proposition 10.3. Let σ be a trace. $\xrightarrow{*}_\sigma$ composes with $\xrightarrow{*}_\sigma$. That is, $\xrightarrow{*}_\sigma \circ \xrightarrow{*}_\sigma \subseteq \xrightarrow{*}_\sigma$. And thus, $\xrightarrow{*}_\sigma$ is transitive.

Proof (Sketch). Let $e \xrightarrow{*}_\sigma f$ and $f \xrightarrow{*}_\sigma g$ witnessed by the sequences $(e_1, f_1) \dots (e_k, f_k)$ and $(e'_1, f'_1) \dots (e'_m, f'_m)$ with $k > 1$, $m > 1$, $e = e_1$, $f = f_k = e'_1$ and $g = f'_m$. Then the $k + m$ sequence $(e_1, f_1) \dots (e_k, f'_1) \dots (e'_m, f'_m)$ witnesses $e \xrightarrow{*}_\sigma g$. QED.

Though $\xrightarrow{*}_\sigma$ gives us a characterization of conflict serializability, it is not clear how to compute it algorithmically in a single pass over the trace. Instead, what we will compute is a slight restriction of the relation $\xrightarrow{*}_\sigma$, defined as follows.

Definition 10.5. For events e, f in trace σ , we say $e \leq^\sigma_E f$, if there is an event g in σ such that $e \leq^\sigma_{\text{CHB}} g$ and either (a) $g = f$, or (b) $g \xrightarrow{*}_\sigma f$ and $\text{txn}_\sigma(g)$ is completed in σ .

Before we state the alternate characterization of conflict serializability in terms of the new relation $\leq^\sigma_E$, let us first observe the following property of $\leq^\sigma_E$.

Proposition 10.4. For a trace σ , the relation $\leq^\sigma_E$ is transitive. That is, $\leq^\sigma_E \circ \leq^\sigma_E \subseteq \leq^\sigma_E$.

Proof (Sketch). Let $(e, f) \in \leq^\sigma_E$ and $(f, g) \in \leq^\sigma_E$. Then, there is an event g' such that $f \leq^\sigma_{\text{CHB}} g'$ and either $g' = g$ or $g' \xrightarrow{*}_\sigma g$ and $\text{txn}_\sigma(g')$ is completed in σ . There are two cases to consider:

- $e \leq^\sigma_{\text{CHB}} f$. In this case we get $e \leq^\sigma_{\text{CHB}} g'$ (due to transitivity of $\leq^\sigma_{\text{CHB}}$) and either $g' = g$, or, $g' \xrightarrow{*}_\sigma g$ with $\text{txn}_\sigma(g')$ being complete in σ . Either way, $e \leq^\sigma_E g$.
- $e \not\leq^\sigma_{\text{CHB}} f$. In this case, there is an f' such that $e \leq^\sigma_{\text{CHB}} f'$, $\text{txn}_\sigma(f')$ is completed in σ and $f' \xrightarrow{*}_\sigma f$. Now observe that the composition $f' \xrightarrow{*}_\sigma f \leq^\sigma_{\text{CHB}} g'$ gives $f' \xrightarrow{*}_\sigma g'$ (due to *Proposition 10.2*). If $g' = g$, then we have $f' \xrightarrow{*}_\sigma g$. Otherwise, we get $f' \xrightarrow{*}_\sigma g$ due to *Proposition 10.3*. Either way we have $e \leq^\sigma_E g$.

QED.

In [16], we presented the following characterization of conflict serializability violations using the new relation $\leq^\sigma_E$. For a non-singleton transaction T in trace σ , we use T_{begin} to denote the begin transaction event $\langle \cdot, \text{begin} \rangle$ of T and T_{end} to denote the end transaction event $\langle \cdot, \text{end} \rangle$ of T .

Theorem 10.1. The following observations hold.

1. Any trace σ with a non-singleton transaction T , events e and f such that $f \in T$, $e \notin T$, $T_{\text{begin}} \leq^\sigma_E e$ and $e \leq^\sigma_E f$, is not conflict serializable.

2. Let σ be a trace that is not conflict serializable with a witness $T_0, \dots, T_{k-1}$ such that each T_i , except possibly one, is complete in σ . Then there is a non-singleton transaction T and events e, f in σ such that $f \in T$, $e \notin T$, $T_{\text{begin}} \prec_{\text{E}}^{\sigma} e$ and $e \leq_{\text{E}}^{\sigma} f$.

Proof (Sketch). ($\Leftarrow$) Observe that for any pair of events e_1, e_2 if $\text{txn}_{\sigma}(e_1) \neq \text{txn}_{\sigma}(e_2)$ and $e_1 \prec_{\text{E}}^{\sigma} e_2$ then $e_1 \xrightarrow{*}_{\sigma} e_2$. This means $T_{\text{begin}} \xrightarrow{*}_{\sigma} e \xrightarrow{*}_{\sigma} f$. Since $\text{txn}_{\sigma}(f) = \text{txn}_{\sigma}(T_{\text{begin}})$, we can rewrite this as $T_{\text{begin}} \xrightarrow{*}_{\sigma} e \xrightarrow{*}_{\sigma} T_{\text{begin}}$ or simply $T_{\text{begin}} \xrightarrow{*}_{\sigma} T_{\text{begin}}$. The rest of the proof follows from Proposition 10.1, the observation that $T_{\text{begin}} \leq_{\text{CHB}}^{\sigma} T_{\text{begin}}$.

($\Rightarrow$) Let $T_0, \dots, T_{k-1}$ be a witness sequence for the conflict serializability violation of σ ($k > 1$ and $T_i \neq T_j$ for every $i \neq j$). Then, we must have a sequence of pairs of events $(e_0, f_0), \dots, (e_{k-1}, f_{k-1})$ such that $\text{txn}_{\sigma}(e_i) = \text{txn}_{\sigma}(f_i) = T_i$, and $f_i \leq_{\text{CHB}}^{\sigma} e_{(i+1) \bmod k}$. Observe that for every $i \neq j$, we have $e_i \xrightarrow{*}_{\sigma} f_j$. Let m be the index of the only active transaction in σ amongst $\{T_i\}_{i=0}^{k-1}$; if all transactions are completed, pick m to be any index such that T_m is non-singleton (one such index will exist). Now let $T = T_m$, $e = e_{(m+1) \bmod k} \notin T$ and $f = e_m \in T$. Now, $T_{\text{begin}} \leq_{\text{CHB}}^{\sigma} f_m \leq_{\text{CHB}}^{\sigma} e_{(m+1) \bmod k} = e'$ and thus $T_{\text{begin}} \prec_{\text{E}}^{\sigma} e'$. Also, because of the choice of m , the transaction $\text{txn}_{\sigma}(e)$ is completed in σ and $e \xrightarrow{*}_{\sigma} f$ and thus $e \leq_{\text{E}}^{\sigma} f$. QED.

Here, we present an alternate characterization that more closely reflects how we detect conflict serializability violations using the relation $\prec_{\text{E}}$. Let us first start with the following definition, which characterizes a set of traces that can be detected using the streaming vector-clock algorithm we present later in this chapter.

Definition 10.6. A trace σ is said to be streaming unserializable if either of the following hold:

1. There is a non-singleton transaction T , and events e and f in σ such that $f \in T$, $e \notin T$, $T_{\text{begin}} \prec_{\text{E}}^{\sigma} e$ and $e \leq_{\text{CHB}}^{\sigma} f$.
2. There are two non-singleton transaction $T \neq T'$ in σ such that $T_{\text{begin}} \prec_{\text{E}}^{\sigma} T'_{\text{begin}}$ and $T'_{\text{begin}} \prec_{\text{E}}^{\sigma} T_{\text{end}}$.

We now prove that *streaming unserializable* (Definition 10.6) traces are essentially the same as traces that violate conflict serializability.

Theorem 10.2. The following observations hold.

1. A streaming unserializable trace is not conflict serializable.
2. Let σ be a trace that is not conflict serializable with a witness $T_0, \dots, T_{k-1}$ such that each T_i , except possibly one, is complete in σ . Then, σ is streaming unserializable.

Proof (Sketch). ($\Rightarrow$). First, consider the case when there is a non-singleton transaction T and events e, f with $f \in T$, $T_{\text{begin}} \prec_{\text{E}}^{\sigma} e$ and $e \leq_{\text{CHB}}^{\sigma} f$. Observe that $e \prec_{\text{E}}^{\sigma} f$ and the rest follows from Theorem 10.1. Next consider the case when there are two non-singleton transactions $T \neq T'$ with $T_{\text{begin}} \prec_{\text{E}}^{\sigma} T'_{\text{begin}} \prec_{\text{E}}^{\sigma} T_{\text{end}}$. Taken $e = T'_{\text{begin}}$ and $f = T_{\text{end}}$ and use Theorem 10.1 to obtain the result.

($\Leftarrow$). Let $T_0, \dots, T_{k-1}$ be a witness sequence for the conflict serializability violation of σ ($k > 1$ and $T_i \neq T_j$ for every $i \neq j$). Then, we must have a sequence of pairs of events $(e_0, f_0), \dots, (e_{k-1}, f_{k-1})$ such that $\text{txn}_{\sigma}(e_i) = \text{txn}_{\sigma}(f_i) = T_i$, and $f_i \leq_{\text{CHB}}^{\sigma} e_{(i+1) \bmod k}$. Observe that for every $i \neq j$, we have $e_i \xrightarrow{*}_{\sigma} f_j$. Let m be the index of the only active transaction in σ amongst $\{T_i\}_{i=0}^{k-1}$; if all transactions are completed, pick m to be the index of the transaction T_m that is non-singleton and ended last amongst the non-singleton ones (one such index will exist). In the first case, the transaction $T_{(m+1) \bmod k} \neq T_m$ is singleton, then we have $f_m \leq_{\text{CHB}}^{\sigma} e_{(m+1) \bmod k}$, $e_{(m+1) \bmod k} = f_{(m+1) \bmod k}$ and $e_{(m+1) \bmod k} \xrightarrow{*}_{\sigma} f_m$. Taking $T = T_m$, $e = e_{(m+1) \bmod k}$ and $f = f_m$, we get that σ is streaming unserializable. In the second case, there is another transaction (amongst $T_0, \dots, T_{k-1}$) that is not singleton but is completed. Let $n \neq m$ be the next index (after m , modulo k) such that T_n is not singleton but completed. In this case, observe that $f_m \leq_{\text{CHB}}^{\sigma} e_n$ and further, $f_n \xrightarrow{*}_{\sigma} e_m$. Taking $T = T_n$ and $T' = T_m$, we observe that $T_{\text{begin}} \prec_{\text{E}}^{\sigma} T'_{\text{begin}}$ as $T_{\text{begin}} \xrightarrow{*}_{\sigma} T'_{\text{begin}}$ and T_n is completed. Also, $T'_{\text{begin}} \leq_{\text{CHB}}^{\sigma} f_m \leq_{\text{CHB}}^{\sigma} e_n \leq_{\text{CHB}}^{\sigma} T_{\text{end}}$, and thus $T'_{\text{begin}} \prec_{\text{E}}^{\sigma} T_{\text{end}}$. This gives us the desired $T_{\text{begin}} \prec_{\text{E}}^{\sigma} T'_{\text{begin}} \prec_{\text{E}}^{\sigma} T_{\text{end}}$. QED.

We conclude this section with examples illustrating both the definition $\prec_{\text{E}}$ and the use of Theorem 10.2.

Example 10.5. Let us begin by looking at trace ρ_3 in Figure 10.3. Let σ_i denote the prefix of ρ_3 upto (and including) event e_i . In trace σ_6 , we have $e_3 \prec_{\text{E}}^{\sigma_6} e_6$, $e_4 \prec_{\text{E}}^{\sigma_6} e_5$, and $e_1 \prec_{\text{E}}^{\sigma_6} e_6$ because they are related by $\leq_{\text{CHB}}$. Here, $e_1 \xrightarrow{*}_{\sigma_6} e_4$ because $\text{txn}_{\sigma_6}(e_1) = \text{txn}_{\sigma_6}(e_3)$, $e_3 \leq_{\text{CHB}}^{\sigma_6} e_6$ and $\text{txn}_{\sigma_6}(e_6) = \text{txn}_{\sigma_6}(e_4)$. However, it is not the case that $e_1 \prec_{\text{E}}^{\sigma_6} e_4$. On the other hand, if we consider σ_7 , then $e_1 \prec_{\text{E}}^{\sigma_7} e_4$ as the transaction in t_1 is complete in σ_7 . In σ_7 (and therefore also in the full trace ρ_3), conditions of Theorem 10.1 are satisfied — $e_1 \prec_{\text{E}}^{\sigma_7} e_4$ and $e_4 \prec_{\text{E}}^{\sigma_7} e_7$.

Example 10.6. Consider trace ρ_4 in Figure 10.4; this is a slight modification of trace ρ_1 from Figure 10.1 that now has an atomicity violation. Observe that in prefix σ_{11} , $e_1 \prec_{\text{E}}^{\sigma_{11}} e_8$. This is because $e_1 \leq_{\text{CHB}}^{\sigma_{11}} e_5$, transaction $\text{txn}_{\sigma_{11}}(e_5)$ is complete in σ_{11} and $e_5 \xrightarrow{*}_{\sigma_{11}} e_8$. Further, $e_8 \leq_{\text{CHB}}^{\sigma_{11}} e_{11}$. Thus by Theorem 10.2, there is a violation of conflict serializability.

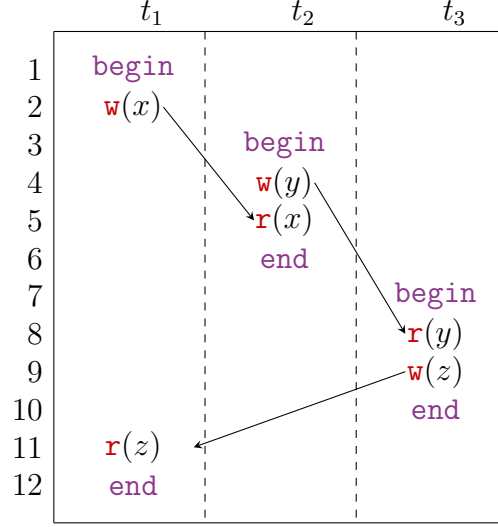


Figure 10.4: Trace ρ_4 . Each transaction is a $\leq_{\text{Txn}}$ predecessor of the other.

10.4 VECTOR CLOCK ALGORITHM

Based on the intuitions developed in Section 10.2, we will now describe our vector clock based algorithm called AERODROME, for checking violations of conflict serializability.

Our algorithm AERODROME is a single pass linear time algorithm. It processes events in the trace as they are generated and (implicitly) assigns vector timestamps to each of these events. Broadly, the goal of the algorithm will be to assign vector timestamps that capture the relation $\leq_E$ (Definition 10.5) and use Theorem 10.2 to discover conflict serializability violations. Similar to vector clock algorithms for HB, SHB, WCP and other prior works on race detection [13, 48, 52], AERODROME does not explicitly store the timestamps of each event in the trace; it instead maintains the timestamps of constantly many events using constantly many vector clocks. This small set of vector clocks is adequate for detecting conflict serializability violations. In the following we formally state the notion of timestamps that the trace, as well the invariants maintained in the different data structures used in the algorithm. The precise pseudocode for the AERODROME algorithm is presented in Algorithm 10.1.

10.4.1 Timestamps capturing $\leq_E$

Before we proceed to discuss the vector clock algorithm, we first precisely describe the notion of timestamps that the algorithm will compute for each event. First, let us note a crucial difference between timestamps assigned by traditional vector clock algorithms, like those assigned when computing the $\leq_{\text{HB}}$ partial order (Algorithm 3.2), and the notion of timestamps we use to design AERODROME. The timestamp associated with an event

Algorithm 10.1: AERODROME: *Vector Clock Algorithm for Checking Violation of Conflict Serializability*

```

1  function Initialization()
2    foreach  $t \in \text{Threads}$  do
3       $\mathbb{C}_t := \perp[t \mapsto 1]$ 
4       $\mathbb{C}_t^{\text{begin}} := \perp$ 
5    foreach  $\ell \in \text{Locks}$  do
6       $\mathbb{L}_\ell := \perp$ 
7       $\text{lastRelThr}_\ell := \text{NIL}$ 
8    foreach  $x \in \text{Vars}$  do
9       $\mathbb{W}_x := \perp$ 
10      $\text{lastWThr}_x := \text{NIL}$ 
11     foreach  $t \in \text{Threads}$  do
12        $\mathbb{R}_{t,x} := \perp$ ;
13  function checkAndGet( $clk, t$ )
14    if  $t$  has an active transaction and
15        $\mathbb{C}_t^{\text{begin}} \sqsubseteq clk$  then
16      declare ‘violation’ and exit
17  handler acquire( $t, \ell$ )
18    if  $\text{lastRelThr}_\ell \neq t$  then
19      checkAndGet( $\mathbb{L}_\ell, t$ )
20  handler release( $t, \ell$ )
21     $\mathbb{L}_\ell := \mathbb{C}_t$ 
22     $\text{lastRelThr}_\ell := t$ 
23  handler fork( $t, u$ )
24     $\mathbb{C}_u := \mathbb{C}_u \sqcup \mathbb{C}_t$ 
25  handler join( $t, u$ )
26    checkAndGet( $\mathbb{C}_u, t$ )
27  handler read( $t, x$ )
28    if  $\text{lastWThr}_x \neq t$  then
29      checkAndGet( $\mathbb{W}_x, t$ )
30     $\mathbb{R}_{t,x} := \mathbb{C}_t$ 
31  handler write( $t, x$ )
32    if  $\text{lastWThr}_x \neq t$  then
33      checkAndGet( $\mathbb{W}_x, t$ )
34    foreach  $u \in \text{Threads} \setminus \{t\}$  do
35      checkAndGet( $\mathbb{R}_{u,x}, t$ )
36     $\mathbb{W}_x := \mathbb{C}_t$ 
37     $\text{lastWThr}_x = t$ 
38  handler begin( $t$ )
39     $\mathbb{C}_t := \mathbb{C}_t[t \mapsto \mathbb{C}_t(t) + 1]$ 
40     $\mathbb{C}_t^{\text{begin}} := \mathbb{C}_t$ 
41  handler end( $t$ )
42    foreach  $u \in \text{Threads} \setminus \{t\}$  do
43      if  $\mathbb{C}_t^{\text{begin}} \sqsubseteq \mathbb{C}_u$  then
44        checkAndGet( $\mathbb{C}_t, u$ )
45    foreach  $\ell \in \text{Locks}$  do
46      if  $\mathbb{C}_t^{\text{begin}} \sqsubseteq \mathbb{L}_\ell$  then
47         $\mathbb{L}_\ell := \mathbb{C}_t \sqcup \mathbb{L}_\ell$ 
48    foreach  $x \in \text{Vars}$  do
49      if  $\mathbb{C}_t^{\text{begin}} \sqsubseteq \mathbb{W}_x$  then
50         $\mathbb{W}_x := \mathbb{C}_t \sqcup \mathbb{W}_x$ 
51      foreach  $u \in \text{Threads}$  do
52        if  $\mathbb{C}_t^{\text{begin}} \sqsubseteq \mathbb{R}_{u,x}$  then
53           $\mathbb{R}_{u,x} := \mathbb{C}_t \sqcup \mathbb{R}_{u,x}$ 

```

does not depend upon future events, in the case of $\leq_{\text{HB}}$. That is, for a trace $\sigma = \sigma_1 \circ e \circ \sigma_2$ containing event e , the timestamp associated with e satisfies $C_e^{\sigma_1 \circ e \circ \sigma_2} = C_e^{\sigma_1 \circ e}$. This property will, however, not be true of the notion of timestamps we will use for our algorithm. Let us now precisely state the definition of timestamps assigned by AERODROME.

Consider a complete observed trace σ . For an event $e \in \sigma$, let $\text{prefix}^\sigma(e)$ be the shortest prefix of σ that contains e . For an event e and thread u in trace σ , let us define $B_e^{\sigma,u}$ to be

the set:

$$B_e^{\sigma,u} = \{f \in \text{Events}_\sigma \mid f \prec_{\mathbb{E}}^\sigma e, \text{thr}(f) = u, \text{op}(f) = \text{begin}\} \quad (10.1)$$

That is, $B_e^{\sigma,u}$ is the set of begin transaction events performed by thread u that are ordered (according to $\prec_{\mathbb{E}}^\sigma$) before e . We remark that $B_e^{\sigma,u}$ may not be equal to $B_e^{\sigma',u}$, where σ' is a prefix of σ . As an example, consider the trace ρ_4 from Figure 10.4. For the prefix $\pi = \text{prefix}^{\rho_4}(e_5)$ of ρ_4 ending at the event e_5 , we have $e_1 \not\prec_{\mathbb{E}}^\pi e_4$ (even though $e_1 \prec_{\mathbb{E}}^\pi e_5$). This is because the transaction $\text{txn}_\pi(e_1)$ is not complete in π . However, for the full trace ρ_4 , we have $e_1 \prec_{\mathbb{E}}^{\rho_4} e_4$ because $\text{txn}_{\rho_4}(e_1)$ is complete in ρ_4 and $e_1 \xrightarrow{*}_{\rho_4} e_5$.

We remark an important property that will be useful to prove later results:

Proposition 10.5. Let σ be a trace, let u be a thread in σ and let e be an event in σ . The set $B_e^{\sigma,u}$ is downward closed with respect to thread order. That is, for two events $e_1 \leq_{\text{tr}}^\sigma e_2$ such that $\text{thr}(e_1) = \text{thr}(e_2)$, and $\text{op}(e_1) = \text{op}(e_2) = \text{begin}$, if $e_2 \in B_e^{\sigma,u}$, then $e_1 \in B_e^{\sigma,u}$.

Proof (Sketch). If $e_2 \prec_{\mathbb{E}}^\sigma e$, then there is an event f such that $e_2 \leq_{\text{CHB}}^\sigma f$ and either $f = e$ or txn_f is completed in σ and $f \xrightarrow{*}_\sigma e$. Notice that $e_1 \leq_{\text{CHB}}^\sigma e_2 \leq_{\text{CHB}}^\sigma f$, giving us $e_1 \prec_{\mathbb{E}}^\sigma e$ and thus $e_1 \in B_e^{\sigma,u}$. QED.

We can now state the timestamp of an event in a prefix of a trace. The vector timestamp C_e^π of event e in prefix π is given by

$$C_e^\pi = \lambda t. \begin{cases} 1 + |B_e^{\pi,t}| & \text{if } t = \text{thr}(e) \text{ or } B_e^{\pi,t} \neq \emptyset \\ 0 & \text{otherwise} \end{cases} \quad (10.2)$$

The above definition can now be used to prove the following intermediate results

Lemma 10.1. Let σ be a trace. Let $e, f \in \text{Events}_\sigma$ with $\text{op}(e) = \text{begin}$. Then,

$$e \prec_{\mathbb{E}}^\sigma f \iff C_e^\sigma \sqsubseteq C_f^\sigma.$$

Proof (Sketch). ($\Rightarrow$.) Let $e \prec_{\mathbb{E}}^\sigma f$. Let us fix a thread u . The only interesting case to consider is when $C_e^\sigma(u) > 0$. In this case, $B_e^{\sigma,u} \neq \emptyset$ because $\text{op}(e) = \text{begin}$. Now, we remark that for every $e' \in B_e^{\sigma,u}$, we have $e' \prec_{\mathbb{E}}^\sigma e$ and further due to Proposition 10.4, $e' \prec_{\mathbb{E}}^\sigma f$. Thus, $B_e^\sigma \subseteq B_f^\sigma$, giving us $C_e^\sigma(u) \leq C_f^\sigma(u)$.

($\Leftarrow$.) Assume $C_e^\sigma \sqsubseteq C_f^\sigma$. Let $u = \text{thr}(e)$. Notice that $e \in B_e^{\sigma,u}$ as $e \prec_{\mathbb{E}}^\sigma e$ (since $e \leq_{\text{CHB}}^\sigma e$) and $\text{op}(e) = \text{begin}$. Thus, $1 < C_e^\sigma(u) \leq C_f^\sigma(u)$. This means $B_f^{\sigma,u} \neq \emptyset$. We remark that e is the last event (according to $\leq_{\text{tr}}^\sigma$) in $B_e^{\sigma,u}$.

Let e' be the last event (according to $\leq_{\text{tr}}^\sigma$) in $B_f^{\sigma,u}$. If $e' \leq_{\text{tr}}^\sigma e$, then $B_f^{\sigma,u} \subsetneq B_e^{\sigma,u}$, contradicting $C_e^\sigma(u) \leq C_f^\sigma(u)$. Thus, $e \leq_{\text{tr}}^\sigma e'$, and thus $e \in B_f^{\sigma,u}$ (due to Proposition 10.5), giving us $e \leq_{\text{E}}^\sigma f$. QED.

10.4.2 Data structures maintained in the State

The most crucial set of clocks maintained by the algorithm are those of the form $\mathbb{C}_t$, for each thread $t \in \text{Threads}$. The clock $\mathbb{C}_t$, intuitively, stores the timestamp of the last event performed by the thread t so far. That is, when performing an event e with $\text{thr}(e) = t$, the timestamp assigned to e by AERODROME is, in fact, determined by the value of the clock $\mathbb{C}_t$ right after e was processed by the algorithm. This is similar in spirit to vector clock algorithms for data race detection [13, 14, 48, 52].

The algorithm also checks for violations of conflict serializability using the characterization in Theorem 10.2, which relies on the timestamp of the begin event of a transaction. The algorithm, therefore, also maintains another clock $\mathbb{C}_t^{\text{begin}}$ which intuitively stores the timestamp of the last begin event performed by thread t .

The goal of these vector timestamps is to capture the relation $\leq_{\text{E}}$. Since $\leq_{\text{E}}$ is defined using $\leq_{\text{CHB}}$, we need to ensure that the vector timestamps reflect the orderings induced by $\leq_{\text{CHB}}$. In order to capture the intra-thread dependencies imposed by $\leq_{\text{CHB}}$ and $\leq_{\text{E}}$, we need auxiliary clocks. Consider an event e of the form $\langle t, \text{acq}(\ell) \rangle$. All previously encountered events with operations on lock ℓ are $\leq_{\text{CHB}}$ -before e . Hence the timestamp of e must be after those assigned to such events. To do this, AERODROME will maintain a vector clock $\mathbb{L}_\ell$ for each lock ℓ , that stores the timestamp of the last $\text{rel}(\ell)$ seen so far; this will be used to ensure that the timestamp of e is appropriately larger. Similarly, we need to ensure that the timestamp of every write event is after the timestamp of all previous writes and reads to the same variable, and that of a read event is after the timestamp of previous writes. Therefore, for every variable x , AERODROME has a clock $\mathbb{W}_x$ that stores the timestamp of the last write $\text{w}(x)$ -event and a clock $\mathbb{R}_{t,x}$ that stores the time of the last event of type $\langle t, \text{r}(x) \rangle$.

Recall that, when considering paths between events through transactions ($\xrightarrow{*}$), we need to make sure that consecutive transactions along the path are distinct. AERODROME tracks this constraint by maintaining scalar variables lastRelThr_ℓ and lastWThr_x , which store the identifier of the thread that performed the last release on ℓ and write on x , respectively.

10.4.3 Initialization and Updates to State

Each of the clocks $\mathbb{C}_t$ are initialized with the time $\perp[t \mapsto 1]$, all other clocks are initialized to $\perp$, and all the scalar variables are initialized to a default value of **NIL**. As new events are observed in the trace, the algorithm updates these vector clocks in a manner that is consistent with tracking the $\leq_E$ -relation.

Recall that, for an event e the timestamp C_e^σ is defined in terms of the size of the sets $\{B_e^{\sigma,t}\}_{t \in \text{Threads}_\sigma}$. A crucial component in maintaining all the timestamps is to keep track of the number of $\langle t, \text{begin} \rangle$ event performed by each thread t . In light of this, when processing a begin event e with $\text{type}(e) = \langle t, \text{begin} \rangle$, the algorithm first increments the *local* component of $\mathbb{C}_t$ (Line 39 - ' $\mathbb{C}_t := \mathbb{C}_t[t \mapsto \mathbb{C}_t(t) + 1]$ ').

When processing an acquire event e of type $\langle t, \text{acq}(\ell) \rangle$, the algorithm makes sure that the timestamp of e is ordered after the timestamp of the last **rel**(ℓ)-event e_ℓ in the trace so far. This is achieved by updating ' $\mathbb{C}_t := \mathbb{C}_t \sqcup \mathbb{L}_\ell$ ' in the procedure **CHECKANDGET** (invoked at Line 19); the procedure **CHECKANDGET** also checks for conflict serializability violation before updating $\mathbb{C}_t$, but more on that later. Of course, if e_ℓ is performed by the same thread t (Line 18), then, this is already ensured and no explicit update is required.

At a write event e of type $\langle t, \text{w}(x) \rangle$, **AERODROME** ensures that the timestamp of e is ordered after all the prior reads and writes on x by calling **checkAndGet**() in Line 33 and Line 35. The algorithm then updates $\mathbb{W}_x$ to be the timestamp of e (see Line 36) and **lastWThr_x** to t , thus preserving the semantics of the clock $\mathbb{W}_x$ and the scalar variable **lastWThr_x**. The updates performed at a read event are similar.

At a fork event e of type $\langle t, \text{fork}(u) \rangle$, the algorithm updates the clock of the child thread u (' $\mathbb{C}_u := \mathbb{C}_u \sqcup \mathbb{C}_t$ ' in Line 24) so that all events of u are ordered after e . At a join event e with $\text{type}(e) = \langle t, \text{join}(u) \rangle$, the algorithm updates $\mathbb{C}_t$ to $\mathbb{C}_t \sqcup \mathbb{C}_u$ so that all events of thread u are ordered before e .

Let us now consider the updates performed at an end-transaction event e ($\text{type}(e) = \langle t, \text{end} \rangle$). Let e^{begin} denote the matching begin transaction event. Observe that for an event f , if $e^{\text{begin}} \leq_E f$, then $e \leq_E f$ because **txn**(e) is completed in σ . That is, all future events that are $\leq_E$ -after e^{begin} must be assigned a timestamp after that of e . This is ensured by updating clocks $\mathbb{C}_u$ for all threads u that satisfy $\mathbb{C}_t^{\text{begin}} \sqsubseteq \mathbb{C}_u$ (Line 42, Line 43, Line 44), and clocks $\mathbb{L}_\ell$, $\mathbb{W}_x$, and $\mathbb{R}_{u,x}$ (Line 45, Line 46, Line 48, Line 49, Line 51, Line 53).

10.4.4 Checking Violations of Atomicity

The basic idea in checking for serializability violations is to check the *streaming unserializable* characterization of Definition 10.6 (see Theorem 10.2). The algorithm essentially checks for either of the two cases 1 or 2 in Definition 10.6. The algorithm detects violations of atomicity at various points by a call to the procedure `CHECKANDGET`. The checks can be broadly classified into two categories, one for each of the two cases in Definition 10.6. To handle the first case (item 1 in Definition 10.6), the algorithm reports a violation at an event e of type $\langle t, op \rangle$ such that there is an earlier event e' (performed by a thread $t' \neq t$) that conflicts with e (and thus $e' \leq_{\text{CHB}} e$). In this case, the violation is reported when $e^{\text{begin}} \prec_E e'$ (where e^{begin} is the begin event of $\text{txn}(e)$). This check is performed at acquire events (Line 19), at read events (Line 29) and at write events (Line 35, Line 33). To handle the second case (item 2 in Definition 10.6), the algorithm reports atomicity violations when processing an end event e with $\text{type}(e) = \langle t, \text{end} \rangle$ (with a matching begin event e^{begin}). In this case, the algorithm detects a violation when there is another thread $u \neq t$ having an active transaction, with begin event e_u^{begin} and last event is e_u , such that $e^{\text{begin}} \prec_E e_u$ (Line 43) and $e_u^{\text{begin}} \prec_E e$ (Line 44). These checks for violations of conflict serializability are performed in `checkAndGet()` (Line 13), which takes two arguments: `clk` (vector timestamp) and `t` (thread identifier), and declares a violation if (a) thread `t` has an active transaction (and is thus a non-singleton transaction), and (b) `clk` is ordered after $\mathbb{C}_t^{\text{begin}}$, which is the timestamp of the begin event of the (active) transaction of t (Line 14). Whenever a violation is found, the algorithm exits. Otherwise, the algorithm continues after updating the value of the clock $\mathbb{C}_t$ to $\mathbb{C}_t \sqcup \text{clk}$ (Line 16).

10.4.5 Nested and Unary Transactions

In the case of nested transactions, it is enough to only consider the outermost transactions and ignore the inner transactions. This is because if there is a cycle involving a transaction T that is nested inside another transaction T' , then there is clearly also a cycle involving T' . As a result, we simply ignore the begin and end events that have a non-zero nesting depth.

Events that are not enclosed by begin and end transaction events constitute a trivial atomic block, namely, one consisting of only that single event. These were called *unary* transactions in [38]. Our algorithm does not report a violation at unary transactions (in the procedure `checkAndGet()`; see Line 14 and Line 15) as these are not active transactions. The algorithm, nevertheless, is still correct as a unary transaction (corresponding to a read, write, acquire or join event) can only correspond to a cycle that involves another non-unary transactions.

	t_1	t_2	$\mathbb{C}_{t_1}$	$\mathbb{C}_{t_2}$	$\mathbb{W}_x$	$\mathbb{W}_y$
1	begin		$\langle 2, 0 \rangle$			
2		begin		$\langle 0, 2 \rangle$		
3	$\mathbf{w}(x)$				$\langle 2, 0 \rangle$	
4		$\mathbf{r}(x)$		$\langle 2, 2 \rangle$		
5		$\mathbf{w}(y)$				$\langle 2, 2 \rangle$
6	$\mathbf{r}(y)$		Conf. serializ. violation			$(\mathbb{C}_{t_1}^{\text{begin}} \sqsubseteq \mathbb{W}_y)$
7	end					
8		end				

Figure 10.5: AERODROME on Trace ρ_2 .

10.4.6 AERODROME on Example Traces

Let us illustrate AERODROME's workings on the traces from Section 10.2. Even though these examples do not use any synchronization primitives like locking, they contain all the features needed to highlight the subtle aspects of AERODROME.

Let us begin with the simplest trace ρ_2 from Figure 10.2. We show the values of the relevant vector clocks in Figure 10.5. In this figure, we only depict the value of a vector clock in row i if its value has changed after processing the i^{th} event e_i in the trace. We do not show the values of the clocks $\mathbb{R}_{t_1,x}$, $\mathbb{R}_{t_2,x}$, $\mathbb{R}_{t_1,y}$ or $\mathbb{R}_{t_2,y}$ as they are not important here. There are two threads and thus the size of each vector clock is 2. The clocks $\mathbb{C}_{t_1}$ and $\mathbb{C}_{t_2}$ are initialized to the timestamps $\langle 1, 0 \rangle$ and $\langle 0, 1 \rangle$ respectively, and all other clocks are initialized to $\perp = \langle 0, 0 \rangle$. The local clocks increment after a begin event (Line 39 in Algorithm 10.1) and thus the clocks $\mathbb{C}_{t_1}$ and $\mathbb{C}_{t_2}$ become $\langle 2, 0 \rangle$ and $\langle 0, 2 \rangle$ after e_2 . Further, these are also the values of the clocks $\mathbb{C}_{t_1}^{\text{begin}}$ and $\mathbb{C}_{t_2}^{\text{begin}}$ from this point onwards until the end of the execution. After processing $e_3 = \langle t_1, \mathbf{w}(x) \rangle$, the value of the clock $\mathbb{W}_x$ becomes $\langle 2, 0 \rangle$ (Line 36). At event e_4 , the call to CHECKANDGET (see Line 29) with arguments $(\langle 2, 0 \rangle, t_2)$ updates the clock $\mathbb{C}_{t_2}$ to $\langle 2, 2 \rangle$ (Line 16). The clock $\mathbb{W}_y$ gets the value of $\mathbb{C}_{t_2} = \langle 2, 2 \rangle$ after processing e_5 . Finally, at event e_6 , the algorithm calls CHECKANDGET with arguments $(\langle 2, 2 \rangle, t_1)$. In this procedure, the algorithm asserts that $\mathbb{C}_{t_1}^{\text{begin}} \sqsubseteq \mathbb{W}_y$ and declares an atomicity violation.

Let us next consider the trace ρ_3 from Figure 10.3. AERODROME's run on this trace is shown in Figure 10.6. Updates corresponding to the first four events are straightforward. In event e_5 , $\mathbb{C}_{t_1}$ gets updated to $\langle 2, 2 \rangle$ because of the call to CHECKANDGET in Line 29. Notice that this call does not raise any violation of atomicity because at this point, $\mathbb{C}_{t_1}^{\text{begin}} = \langle 2, 0 \rangle$ and the clock $\mathbb{W}_y$ is $\langle 0, 2 \rangle$ thus failing the check $\mathbb{C}_{t_1}^{\text{begin}} \sqsubseteq \mathbb{W}_y$ in Line 14. The same explanation applies to the $\mathbf{r}(x)$ event e_6 in t_2 and thus no atomicity violation is reported

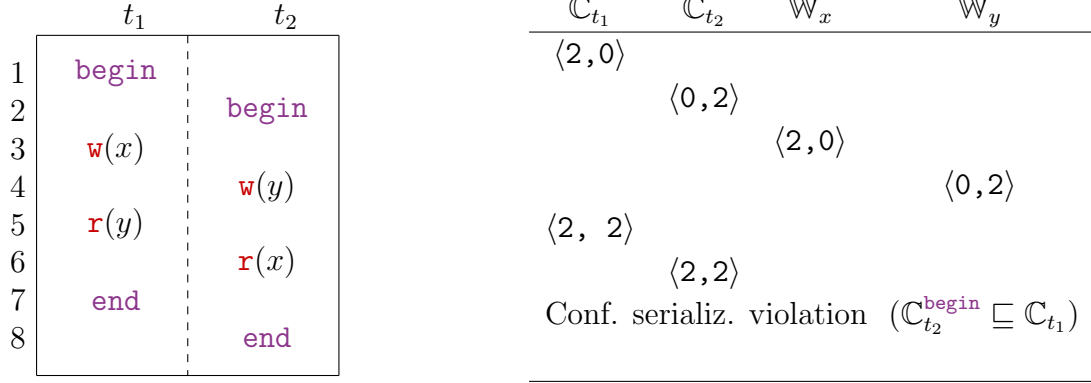


Figure 10.6: AERODROME on Trace ρ_3 .

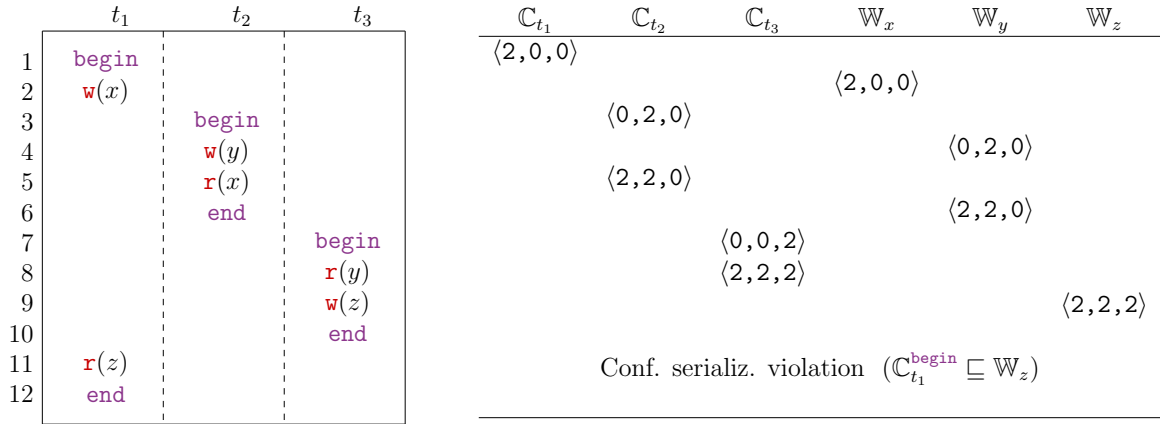


Figure 10.7: AERODROME on Trace ρ_4 .

here as well. Next, the algorithm processes the end event $e_7 = \langle t_1, \text{end} \rangle$. At this point, the algorithm checks if any event in the currently active transaction of t_2 is ordered after e_1 (condition $\mathbb{C}_{t_1}^{\text{begin}} \sqsubseteq \mathbb{C}_{t_2}$ in Line 43 of Algorithm 10.1). This check succeeds since $\mathbb{C}_{t_1}^{\text{begin}} = \langle 2, 0 \rangle$ and $\mathbb{C}_{t_2} = \langle 2, 2 \rangle$ at this point. The algorithm then checks if $\mathbb{C}_{t_2}^{\text{begin}} \sqsubseteq \mathbb{C}_{t_1}$ in the procedure CHECKANDGET and thus declares an atomicity violation. This illustrates the subtlety in how the algorithm reports atomicity violations at an end event.

We will now illustrate how Algorithm 10.1 detects the atomicity violation in the more involved trace ρ_4 from Figure 10.4. This example illustrates how AERODROME handles dependencies between transactions introduced by future events. The run of AERODROME on ρ_4 is shown in Figure 10.7. We omit the updates to the clocks $\mathbb{R}_{t_i, u}$ ($i \in \{1, 2, 3\}$, $u \in \{x, y, z\}$) as they do not play a significant role in this example. All vector clocks have dimension 3 because there are three threads in ρ_4 . As before, the clocks are initialized as follows: $\mathbb{C}_{t_1} = \langle 1, 0, 0 \rangle$, $\mathbb{C}_{t_2} = \langle 0, 1, 0 \rangle$ and $\mathbb{C}_{t_3} = \langle 0, 0, 1 \rangle$; all other clocks are initialized to $\langle 0, 0, 0 \rangle$. The begin events result in incrementing of local clocks and thus $\mathbb{C}_{t_1} = \langle 2, 0, 0 \rangle$

after e_1 . Further, the clock $\mathbb{W}_x$ gets updated to the value of $\mathbb{C}_{t_1}$ at the end of e_2 . The next two events e_3 and e_4 are processed in a similar fashion. At event $e_5 = \langle t_2, \mathbf{r}(x) \rangle$, the clock $\mathbb{C}_{t_2}$ gets updated to $\langle 2, 2, 0 \rangle$ (Line 16 in Algorithm 10.1). After this, the transaction in t_2 ends. The clocks of none of the threads is updated because of e_6 as neither thread t_1 nor t_3 have clock values larger than $\mathbb{C}_{t_2}^{\text{begin}}$ (Line 43). However the write and read clocks are updated. Specifically, the clock $\mathbb{W}_y$ maintaining the timestamp to the last write to y is such that $\mathbb{C}_{t_2}^{\text{begin}} \sqsubseteq \mathbb{W}_y$ and thus, the algorithm updates $\mathbb{W}_y$ to $\mathbb{W}_y \sqcup \mathbb{C}_{t_2} = \langle 2, 2, 0 \rangle$ (Line 49 in Algorithm 10.1). Event e_7 is a begin event and updates $\mathbb{C}_{t_3}$ to $\langle 0, 0, 2 \rangle$. Now at the $\mathbf{r}(y)$ event e_8 , the clock $\mathbb{C}_{t_3}$ gets updated with $\mathbb{W}_y$ which at this point evaluates to $\langle 2, 2, 0 \rangle$, thus giving $\mathbb{C}_{t_3} = \langle 2, 2, 2 \rangle$. The write clock $\mathbb{W}_z$ then gets updated to $\langle 2, 2, 2 \rangle$ after e_9 . More clock updates happen at e_{10} (though not shown in Figure 10.7) Finally, an atomicity violation is detected at event $e_{11} = \langle t_1, \mathbf{r}(z) \rangle$; the algorithm checks if the clock $\mathbb{W}_z$ *knows* some event in t_1 ($\mathbb{C}_{t_1}^{\text{begin}} \sqsubseteq \mathbb{W}_z$) and declares a violation of conflict serializability as this check passes.

10.4.7 Reducing the number of Read Clocks

Recall that Algorithm 10.1 maintains, a vector clock $\mathbb{R}_{t,x}$ for every pair of thread t and memory location x . Therefore, the number of such vector clocks that need to be tracked in the basic algorithm is $O(\mathcal{T} \cdot \mathcal{V})$. Storing and updating these many clocks can be expensive, when the number of memory locations that need to be tracked is prohibitively large, as is the case for most real world software. We tackle this using our optimization to reduce the number of clocks from $O(\mathcal{T} \cdot \mathcal{V})$ to $O(\mathcal{V})$. To understand the optimization, we need to first understand the role served by clocks $\mathbb{R}_{t,x}$. First, these clocks help detect atomicity violation — at a write event e with type $\langle t, \mathbf{w}(x) \rangle$, a violation is reported if there is a thread $u \neq t$ such that $\mathbb{C}_t^{\text{begin}} \sqsubseteq \mathbb{R}_{u,x}$ (Line 14 invoked from Line 35 in Algorithm 10.1). Second, these clocks are used to update $\mathbb{C}_t$ — at a write event $e = \langle t, \mathbf{w}(x) \rangle$, we set $\mathbb{C}_t := \bigsqcup_{u \neq t} \mathbb{C}_t \sqcup \mathbb{R}_{u,x}$ (Line 16 invoked iteratively at Line 35).

The reduction in the number of clocks is achieved by instead maintaining one clock (per memory location) for each of the above two purposes instead of maintaining $O(\mathcal{T})$ many clocks (per memory location). First, for updating clocks correctly at write events, we will maintain a single clock $\mathbb{R}_x$ for each location x . This clock stores the value $\bigsqcup_u \mathbb{R}_{u,x}$ at each point while processing the trace. Next, to perform checks for violations of conflict serializability, we will have another clock $\text{ch}\mathbb{R}_x$ (*check read*). This clock will store the value $\bigsqcup_u \mathbb{R}_{u,x}[0/u]$ at each point in the analysis. Based on the invariants maintained by the algorithm, one can show that checking $\mathbb{C}_t^{\text{begin}} \sqsubseteq \bigsqcup_{u \neq t} \mathbb{R}_{u,x}$ is equivalent to checking $\mathbb{C}_t^{\text{begin}} \sqsubseteq \text{ch}\mathbb{R}_x$. This optimization and other useful optimizations that improve the performance

of AERODROME, are outlined in greater detail in [133].

10.4.8 Correctness and Complexity of AERODROME

Let us first prove the correctness of Algorithm 10.1. We will focus on the basic algorithm without the optimizations; the optimizations do not fundamentally change how our correctness argument is structured.

Clock Invariants. Our correctness argument is largely centered around computing the timestamps of each event. We prove this computation correct by defining the invariants that are maintained in each of the clocks. We state these invariants next. For an arbitrary prefix π of σ , we will find it useful to introduce notation for identifying some specific events in π . For a pair $p = \langle t, op \rangle$, ev_p^π denotes the last event of the form p in π ; note that for some pairs p , this maybe undefined as there might be no event of this form in π . Thus, for example, $\text{ev}_{\langle t, \text{begin} \rangle}^\pi$ denotes the last transaction begin event performed by thread t in π . Sometimes, it will be convenient to leave one of the two arguments in the pair $p = \langle t, op \rangle$ unspecified, and in this case ev_p^π will denote the last event of type identified by the specified argument. Thus, for example, $\text{ev}_{\langle \cdot, \mathbf{w}(x) \rangle}^\pi$ is the last $\mathbf{w}(x)$ -event in π (regardless of the thread performing it), and $\text{ev}_{\langle t, \cdot \rangle}^\pi$ is the last event of thread t in π (regardless of the operation).

Lemma 10.2. Let σ be a trace and let π of σ . For a thread t , lock ℓ and variable x , let T_t^π , $T_t^{\pi, \text{begin}}$, L_ℓ^π , R_x^π and W_x^π be the values of the clocks $\mathbb{C}_t$, $\mathbb{C}_t^{\text{begin}}$, $\mathbb{L}_\ell$, $\mathbb{R}_{t,x}$ and $\mathbb{W}_x$ after Algorithm 10.1 processes π . Then,

$$\begin{aligned} T_t^\pi &= \begin{cases} \perp[t \mapsto 1] & \text{if } \text{ev}_{\langle t, \cdot \rangle}^\pi \text{ does not exist} \\ C_{\text{ev}_{\langle t, \cdot \rangle}^\pi}^\pi & \text{otherwise} \end{cases} \\ T_t^{\pi, \text{begin}} &= \begin{cases} \perp & \text{if } \text{ev}_{\langle t, \text{begin} \rangle}^\pi \text{ does not exist} \\ C_{\text{ev}_{\langle t, \text{begin} \rangle}^\pi}^\rho & \text{otherwise, where } \rho = \text{prefix}^\pi(\text{ev}_{\langle t, \text{begin} \rangle}^\pi) \end{cases} \\ L_\ell^\pi &= \begin{cases} \perp & \text{if } \text{ev}_{\langle \cdot, \text{rel}(\ell) \rangle}^\pi \text{ does not exist} \\ C_{\text{ev}_{\langle \cdot, \text{rel}(\ell) \rangle}^\pi}^\pi & \text{otherwise} \end{cases} \\ R_x^\pi &= \begin{cases} \perp & \text{if } \text{ev}_{\langle t, \mathbf{r}(x) \rangle}^\pi \text{ does not exist} \\ C_{\text{ev}_{\langle t, \mathbf{r}(x) \rangle}^\pi}^\pi & \text{otherwise} \end{cases} \\ W_x^\pi &= \begin{cases} \perp & \text{if } \text{ev}_{\langle t, \mathbf{w}(x) \rangle}^\pi \text{ does not exist} \\ C_{\text{ev}_{\langle t, \mathbf{w}(x) \rangle}^\pi}^\pi & \text{otherwise} \end{cases} \end{aligned}$$

The lemma is proved by an induction on the length of the trace processed by the algorithm. The proof is straightforward, and skipped.

Together with Lemma 10.2 and Theorem 10.2, we have the following.

Theorem 10.3. On any trace σ , Algorithm 10.1 reports a violation of conflict serializability iff σ is not conflict serializable with a witness $T_0, \dots, T_{k-1}$ such that each T_i , except possibly one, is complete in σ .

Correctness of Read Clock optimization. The algorithm maintains the invariant that for two events e_1 and e_2 with $\text{thr}(e_1) = t_1$, we have $C_{e_1}^\pi \sqsubseteq C_{e_2}^\pi$ iff $C_{e_1}^\pi(t_1) \leq C_{e_2}^\pi(t_1)$, where π is some prefix of the trace. In other words, in order to compare the timestamps of two events, it is enough to compare the *local time* corresponding to the thread of the smaller timestamp. Now, observe that, at a write event $e = \langle t, \mathbf{w}(x) \rangle$, the algorithm detects an atomicity violation by either comparing with the clock of the last write event ($\mathbb{W}_x$), or by comparing with the clocks of the last read events of each thread, except the thread t . Let us consider the second check. Observe that in this case, a violation is raised if there is a thread $u \neq t$ such that $\mathbb{C}_t^{\text{begin}} \sqsubseteq \mathbb{R}_{u,x}$. Based on our earlier observation about local times, this check is equivalent to the check $\exists u \neq t \cdot \mathbb{C}_t^{\text{begin}}(t) \sqsubseteq \mathbb{R}_{u,x}(t)$. Now observe that

$$\begin{aligned} \exists u \neq t \cdot \mathbb{C}_t^{\text{begin}}(t) \sqsubseteq \mathbb{R}_{u,x}(t) &\text{ iff } \mathbb{C}_t^{\text{begin}}(t) \sqsubseteq \bigsqcup_{u \neq t} \mathbb{R}_{u,x}(t) \\ &\text{ iff} \\ \mathbb{C}_t^{\text{begin}}(t) \sqsubseteq \bigsqcup_u \mathbb{R}_{u,x}[0/u](t) &\text{ iff } \mathbb{C}_t^{\text{begin}} \sqsubseteq \bigsqcup_u \mathbb{R}_{u,x}[0/u] \end{aligned} \tag{10.3}$$

Based on the above observation, we can perform the check for atomicity if we have a single clock that maintains the timestamp $\bigsqcup_{e_{t,\mathbf{r}(x)}} C_{e_{t,\mathbf{r}(x)}}[0/t]$, where $e_{u,\mathbf{r}(x)}$ is the last event of the form $\langle u, \mathbf{r}(x) \rangle$ seen in the trace so far. For this, will use a new single clock $\text{ch}\mathbb{R}_x$ to store this value and inductively maintain this in the algorithm.

Next, observe that the algorithm ensures that the timestamp of the last event in a given thread is larger than the timestamp of any earlier event in the same thread. This means that, at any point, $\mathbb{R}_{x,t} \sqsubseteq \mathbb{C}_t$ and thus we have $\mathbb{R}_x \sqcup \mathbb{C}_t = \mathbb{C}_t$ at any point in the algorithm. Now, let us consider how the algorithm updates $\mathbb{C}_t$ with the various $\mathbb{R}_{u,x}$ clocks at a write event. Precisely, if an atomicity violation is not detected when comparing with the read clocks, the value of $\mathbb{C}_t$ becomes $C_t^{\text{old}} \sqcup \bigsqcup_{u \neq t} \mathbb{R}_{u,x}$, where C_t^{old} is the value of $\mathbb{C}_t$ before the updates. Coupled with our previous observation, this new value is the same as the value $C_t^{\text{old}} \sqcup \mathbb{R}_{t,x} \sqcup \bigsqcup_{u \neq t} \mathbb{R}_{u,x}$ which, in turn, can be re-written as $C_t^{\text{old}} \sqcup \bigsqcup_u \mathbb{R}_{u,x}$. Thus, we can maintain the timestamp

Algorithm 10.2: *Reducing the number of read clocks in Algorithm 10.1*

```

1 function Initialization()
2   foreach  $t \in \text{Threads}$  do
3      $\mathbb{C}_t := \perp[t \mapsto 1]$ 
4      $\mathbb{C}_t^{\text{begin}} := \perp$ 
5   foreach  $\ell \in \text{Locks}$  do
6      $\mathbb{L}_\ell := \perp$ 
7      $\text{lastRelThr}_\ell := \text{NIL}$ 
8   foreach  $x \in \text{Vars}$  do
9      $\mathbb{W}_x := \perp$ 
10     $\text{lastWThr}_x := \text{NIL}$ 
11     $\mathbb{R}_x := \perp$ 
12     $\text{chR}_x := \perp$ 
13 function checkAndGet( $clk_1, clk_2, t$ )
14   if  $t$  has an active transaction and
15      $\mathbb{C}_t^{\text{begin}} \sqsubseteq clk_1$  then
16     declare ‘violation’ and exit
17    $\mathbb{C}_t := \mathbb{C}_t \sqcup clk_2$ 
18 handler acquire( $t, \ell$ )
19   if  $\text{lastRelThr}_\ell \neq t$  then
20      $\text{checkAndGet}(\mathbb{L}_\ell, \mathbb{L}_\ell, t)$ 
21 handler release( $t, \ell$ )
22    $\mathbb{L}_\ell := \mathbb{C}_t$ 
23    $\text{lastRelThr}_\ell := t$ 
24 handler fork( $t, u$ )
25    $\mathbb{C}_u := \mathbb{C}_u \sqcup \mathbb{C}_t$ 
26 handler join( $t, u$ )
27    $\text{checkAndGet}(\mathbb{C}_u, \mathbb{C}_u, t)$ 
28 handler read( $t, x$ )
29   if  $\text{lastWThr}_x \neq t$  then
30      $\text{checkAndGet}(\mathbb{W}_x, \mathbb{W}_x, t)$ 
31    $\mathbb{R}_x := \mathbb{R}_x \sqcup \mathbb{C}_t$ 
32    $\text{chR}_x := \text{chR}_x \sqcup \mathbb{C}_t[t \mapsto 0]$ 
33 handler write( $t, x$ )
34   if  $\text{lastWThr}_x \neq t$  then
35      $\text{checkAndGet}(\mathbb{W}_x, \mathbb{W}_x, t)$ 
36    $\text{checkAndGet}(\text{chR}_x, \mathbb{R}_x, t)$ 
37    $\mathbb{W}_x := \mathbb{C}_t$ 
38    $\text{lastWThr}_x = t$ 
39 handler begin( $t$ )
40    $\mathbb{C}_t := \mathbb{C}_t[t \mapsto \mathbb{C}_t(t) + 1]$ 
41    $\mathbb{C}_t^{\text{begin}} := \mathbb{C}_t$ 
42 handler end( $t$ )
43   foreach  $u \in \text{Threads} \setminus \{t\}$  do
44     if  $\mathbb{C}_t^{\text{begin}} \sqsubseteq \mathbb{C}_u$  then
45        $\text{checkAndGet}(\mathbb{C}_t, \mathbb{C}_t, u)$ 
46   foreach  $\ell \in \text{Locks}$  do
47     if  $\mathbb{C}_t^{\text{begin}} \sqsubseteq \mathbb{L}_\ell$  then
48        $\mathbb{L}_\ell := \mathbb{C}_t \sqcup \mathbb{L}_\ell$ 
49   foreach  $x \in \text{Vars}$  do
50     if  $\mathbb{C}_t^{\text{begin}} \sqsubseteq \mathbb{W}_x$  then
51        $\mathbb{W}_x := \mathbb{C}_t \sqcup \mathbb{W}_x$ 
52      $\mathbb{R}_x := \mathbb{C}_t \sqcup \mathbb{R}_x$ 
53      $\text{chR}_x := \mathbb{C}_t[t \mapsto 0] \sqcup \text{chR}_x$ 

```

$\sqcup_{e_{t,\mathbf{r}(x)}} C_{e_{t,\mathbf{r}(x)}}$ in a single clock ($e_{u,\mathbf{r}(x)}$ as before, is the last event of the form $\langle u, \mathbf{r}(x) \rangle$ seen in the trace so far). We use a new clock $\mathbb{R}_x$ to maintain this value.

We present the read clock optimization in Algorithm 10.2.

We now state the time and space complexity for the optimized version discussed in Section 10.4.7. We will use $\mathcal{N}_{\text{non-end}}$ and $\mathcal{N}_{\text{end}}$ for the number of non-end events and end events in the trace (and thus $\mathcal{N} = \mathcal{N}_{\text{non-end}} + \mathcal{N}_{\text{end}}$).

Theorem 10.4. The algorithm takes $O(\mathcal{T} \log \mathcal{T} \cdot (\mathcal{N}_{\text{non-end}} + (\mathcal{T} + \mathcal{L} + \mathcal{V})\mathcal{N}_{\text{end}}))$ time and $O(\mathcal{T} \log \mathcal{T} \cdot (\mathcal{T} + \mathcal{V} + \mathcal{L}))$ space.

The complexity observations easily follow from the description of the algorithm and the optimization discussed in Section 10.4.7.

10.5 EXPERIMENTAL EVALUATION

In this section, we describe our implementation of AERODROME and the results of evaluating it on benchmark programs.

10.5.1 Implementation

We have implemented AERODROME in RAPID [86]. The primary goal of the evaluation is to assess if the theoretical bound (linear time) of the algorithm also translates to effective performance in practice, or in other words, *does our vector clock algorithm perform better than existing approaches such as the classical graph based algorithm (Velodrome) proposed in [38]?* We emphasize that the primary purpose of the evaluation is to compare different *algorithms* for checking atomicity instead of comparing different *tools* that implement these algorithms.

Logging. In order to evaluate our algorithm against the above objective and to ensure a fair comparison with other approaches, we must ensure that all competing candidate algorithms analyze the same trace. However, the dynamic behavior of a concurrent program can vary significantly across different runs, even when starting with the same input. In order to ensure fairness, we compare the performance of the different algorithms on the *same* dynamic execution. Our tool RAPID therefore first extracts an execution trace from a concurrent programs and then analyzes the same trace against all candidate algorithms. We use RoadRunner [134] to log traces from our set of benchmark programs. RoadRunner uses load time program instrumentation and can be extended to log various events — read and write accesses to memory locations, acquire and release of synchronization objects (locks), forks and joins of threads, and events generated at the entry and exit of each method, which we respectively mark as transaction begin and end events.

Velodrome. The Velodrome algorithm [38] runs in (worst case) cubic time and analyzes traces by building a directed graph, with transactions as nodes in the graph and where the edges correspond to $\prec_{Txn}$ relation between transactions. There was no publicly available implementation of Velodrome that analyzes logged executions. Thus, we also implement this algorithm in RAPID. We use the Java graph library JGraphT [135] to implement various graph operations (adding nodes and edges, cycle detection, etc.) in Velodrome algorithm. In

our implementation of Velodrome, we also incorporate garbage collection as an optimization suggested in [38] — transactions with no incoming edges do not participate in cycles and can be deleted from the graph. In line with the objective of our evaluation, we analyze AERODROME and Velodrome on the same trace (generated by RoadRunner) to ensure a fair comparison.

Other techniques. The tool DoubleChecker [61] is a state-of-the-art tool for checking conflict serializability in a sound and complete manner. DoubleChecker implements a two-phase analysis — the first phase performs a fast but imprecise analysis and reports an over-approximation of the actual set of cycles in the transaction graph. The second phase then filters out the false positives from this set with a more fine grained analysis. DoubleChecker’s performance crucially relies on the first phase being carried out while the program executes. Therefore, one cannot get performance data for DoubleChecker on a logged trace. As a result, there can be no fair comparison between our algorithm and DoubleChecker as one cannot guarantee that the two analyses run on the same trace. In order to gauge if DoubleChecker will significantly outperform our implementation of AERODROME, we ran DoubleChecker’s publicly available implementation [136] on a subset of our benchmarks. On these benchmarks, DoubleChecker’s performance was slower by an order of magnitude. While these experiments do not indicate that DoubleChecker performs worse than our algorithm, they do suggest that our algorithm will be competitive against DoubleChecker. We choose not to present these numbers in this paper, because they are not an apples-to-apples comparison.

10.5.2 Atomicity Specifications and Benchmarks

Atomicity Specifications. In general, the logging mechanism in RoadRunner instruments and tracks all events corresponding to entering and exiting methods. A naïve atomicity specification would be to mark all method boundaries as atomic. However, as expected, not all methods are intended to be atomic. Thus, atomicity specifications need to be specially identified by developers, by supplying manual annotations [124]. In the absence of such static annotations, we use atomicity specifications from prior work [61] whenever possible (Table 10.1). DoubleChecker infers these specifications in an iterative fashion. First, all methods are marked atomic. Next, on detecting atomicity violations, one removes the boundaries of one of the transactions participating in a cycle. This continues until the last violation. In our work, we directly use the specifications inferred by Biswas et al [61] as they were available publicly. For the benchmarks (Table 10.2) for which no specifications were available, we declare all methods except the `main` and `run` methods to be atomic.

Table 10.1: Trace characteristics and running times for benchmarks with atomicity specifications from DoubleChecker.

1	2	3	4	5	6	7	8	9	10
Program ,	$\mathcal{N}$	$\mathcal{T}$	$\mathcal{L}$	$\mathcal{V}$	$\mathcal{N}_{\text{end}}$	Atomic?	Velodrome (s)	Aerodrome (s)	Speed-up
avroa	2.4B	7	7	1079K	498M	✗	TO	1.5	> 24000
elevator	280K	5	50	725	22.6K	✓	162	1.7	97
hedc	9.8K	7	13	1694	84	✗	0.07	0.06	1.16
luindex	570M	3	65	2.5M	86M	✗	581	674	0.86
lusearch	2.0B	14	772	38M	306M	✗	TO	5.5	> 6545
molodyn	1.7B	4	1	121K	1.4M	✗	TO	54.9	> 650
montecarlo	494M	4	1	30.5M	812K	✗	TO	0.75	> 48000
philo	613	6	1	24	0	✓	0.02	0.02	1
pmd	367M	13	223	12.9M	81M	✗	3.1	3.8	0.82
raytracer	2.8B	4	1	12.6M	277M	✓	TO	55m40s	> 10.7
sor	608M	4	2	1M	637K	✗	6.9	9.6	0.72
sunflow	16.8M	16	9	1.2M	2.5M	✗	67.9	0.65	104.5
tsp	312M	9	2	181M	9	✗	4.2	5.7	0.73
xalan	1.0B	13	8624	31M	214M	✗	1.6	2.0	0.8

Table 10.2: Trace characteristics and running times for benchmarks with naive atomicity specifications.

1	2	3	4	5	6	7	8	9	10
Program	$\mathcal{N}$	$\mathcal{T}$	$\mathcal{L}$	$\mathcal{V}$	$\mathcal{N}_{\text{end}}$	Atomic?	Velodrome (s)	Aerodrome (s)	Speed-up
batik	186M	7	1916	4.9M	15M	✗	52.7	65.5	0.81
crypt	126M	7	1	9M	50	✗	92.1	104	0.88
fop	96M	1	115	5M	25M	✓	88.3	92.5	0.95
lufact	135M	4	1	252K	642M	✗	2.4	2.9	0.82
series	40M	4	1	20K	20M	✗	61.0	15.3	3.98
sparsematmult	726M	4	1	1.6M	25	✗	1210	1197	1.01
tomcat	726M	4	1	1.6M	25	✗	3.4	4.5	0.75

Benchmarks and Setup. Our benchmark programs (Table 10.1 and Table 10.2) are derived from the DaCaPo benchmark suite [89] adapted to run with RoadRunner [134], Java Grande Forum [88] and microbenchmarks from [137] and have been used in prior work [61]. Our experiments were conducted on a 2.6GHz 64-bit Linux machine with Java 1.8 as the JVM and 30GB heap space. In each table, Column 1 depicts the name of the benchmark. Column 2 reports the number of events in the trace generated from the corresponding benchmark program in Column 1. Observe that the number of events in the execution traces can vary from a few hundred to billions of events and our algorithm can scale to such large traces. Column 3, 4 and 5 report the number of distinct threads, locks and variables accessed in the trace generated. Column 6 reports the number of transactions in the trace. Column 7 reports ‘✗’ if an atomicity violation was detected and reports ‘✓’ otherwise. Columns 8 and 9 report the time (in seconds) taken by respectively the Velodrome algorithm and AERODROME introduced in this article to analyze the trace generated; a ‘TO’ represents timeout after 10 hours. Column 10 reports the speed-up of AERODROME over Velodrome.

10.5.3 Evaluation Results

For the first set of benchmarks (Table 10.1), we use the atomicity specification obtained from prior work [61]. For the second set of benchmarks (Table 10.2), we use default atomicity specifications (all methods except `main` and `run` are assumed to be atomic). The specifications from [61] are carefully crafted to ensure that spurious atomicity violations are not reported. In the absence of careful specifications, we can expect that the violations will be reported early on in executions.

Let us first consider the first set of benchmarks from Table 10.1. On most of these benchmarks, the violations of atomicity are discovered late in the trace. This is expected as the specifications are realistic and do not declare all methods to be atomic. The performance of AERODROME is significantly better than that of Velodrome. Velodrome times out on most of these benchmarks (time limit was set to be 10 hours). This is because of the prohibitively large number of transactions that get accumulated in these traces. Consider, for example, the case of `sunflow` for which AERODROME takes less than a second, while Velodrome spends about 68 seconds. In this benchmark, the number of nodes in the graph analyzed by Velodrome is about 9000, at the point where the violation is reported. This coupled with the cubic runtime complexity, results in the notable slowdown. Notice that, the slowdown is despite the garbage collection optimization implemented in Velodrome.

Our algorithm, on the other hand, has a linear running time. Similarly, in the benchmark **avro**, the number of transactions is more than 393K in the prefix of the trace in which AERODROME reports an atomicity violation. Any super linear time analysis is unlikely to scale for so many transactions, and Velodrome, in fact, does not return an answer within 10 hours. AERODROME, on the other hand, scales to traces with more than a billion events (**avro**, **lusearch**, **modyn**, **raytracer**, **xalan**) and demonstrates the effectiveness of a linear time vector clock algorithm. For the examples on which AERODROME does not give a huge speedup over Velodrome, we discovered that the number of nodes in Velodrome’s graph analysis is fairly small owing to garbage collection; for example, there were 13 nodes in the graph for **pmd**, 4 nodes in **sor** and 13 nodes in **xalan**.

In the second set of benchmarks, we notice that the performance of Velodrome is comparable to that of our algorithm AERODROME. This is expected because the atomicity specifications are inadequate and do not reflect realistic ones — typically most methods are non-atomic and developers have to identify a smaller set of candidate code blocks that they think are atomic. As a result, on these benchmarks, violations are detected early on in the trace and thus, the size of the transaction graph in Velodrome’s analysis is small. A detailed analysis of the traces suggests that in all these benchmarks, the number of nodes in the transaction graph constructed by Velodrome did not grow more than 4, except for **tomcat**, for which the size of the graph grows to 21. In this case, the cost of maintaining vector clocks and updating them at every event overrides their potential benefits, and as a result, the graph based algorithm runs faster.

CHAPTER 11: CONCLUSIONS AND FUTURE DIRECTIONS

Reasoning about multi-threaded programs is notoriously challenging due to the inherent nondeterminism that arises from thread scheduling in such systems. In this thesis, we explored and proposed foundational ideas and algorithms based on them for dynamically detecting concurrency bugs to aid software developers in writing correct concurrent software. The three guiding principles behind the algorithms we propose here are soundness, scalability and high recall for concurrency bug detection. We presented techniques for detecting data races and atomicity violations that utilize several paradigms like compression and predictive reasoning, constructs such as partial orders and data structures like vector clocks. The ideas presented here can be extended to many different settings and we explore some of these avenues for future work here.

11.1 STATE-OF-THE-ART RACE DETECTOR

In this thesis, we propose several predictive algorithms for detecting data races that improve the existing state-of-the-art in terms the number of data races they expose. However, in order for our techniques to be widely popular amongst developers of concurrent software, they need to be incorporated in a blazing fast practically usable tool. While the algorithms we proposed have asymptotic complexity that is similar to the algorithms employed in existing dynamic race detectors [53, 55] (primarily based on the happens-before partial order [47, 48]), race detectors like THREADSANITIZER are orders of magnitude faster than naive implementations of the algorithms we proposed, thanks to the myriad of heuristics employed in these tools that make them extremely efficient and useful for practitioners [2]. There are several technical challenges that need to be overcome to design an extremely fast predictive race detector. These challenges provide a rich avenue for future work.

The first challenge is the linear dependence of the runtime of these algorithms on the number of threads in the execution being monitored. Prior work on HB race detection for programs with structured parallelism [138–141] and the likes [142, 143] has been successful in improving this dependence by exploiting specific structures in trace topologies.

The second challenge comes from selective ordering of critical sections. HB naively orders all critical sections on the same lock, missing even simple races that can be uncovered by predictive algorithms proposed in this thesis [13, 18] and the later works derived from them [36, 37]. Each of the sound predictive techniques do so by maintaining additional book-keeping information that grows linearly with number of locks, memory locations tracked

or sometimes the number of critical sections tracked. This can be prohibitive especially when these parameters are large, for example in software that dynamically create objects. Further, in order to ensure atomicity of the metadata tracked by the dynamic analysis algorithms, additional synchronization needs to be enforced, slowing down the program when analyzed during runtime. FASTTRACK [52] style epoch optimizations can be promising in this scenario. Recently, Roemer et. al. [93] proposed some heuristics to improve the runtime overhead of partial order based predictive race detection techniques. However, there are still miles to go in designing a more practical race detector that is as usable as HB-based race detectors.

We also expect other enhancements for HB-based online race detection to be readily useful for predictive algorithms proposed here. These include the use of static analysis for placement of race checks during instrumentation time [144, 145], array shadow state compression [146], sophisticated protocols for reducing metadata synchronization overhead [62, 147].

11.2 PREDICTIVE REASONING

One of the key ideas explored in this thesis is predictive style of reasoning where one detects bugs in alternate executions beyond the execution observed while performing the analysis. There are several directions that are interesting for future work.

Control/data flow and synchronization abstraction. Our notion of correct reorderings preserve the last write corresponding to every read event. This is a conservative assumption that ensures that the control flow opted by an execution will nevertheless be allowed in every correct reordering. However, predictive reasoning can be enhanced in presence of the knowledge of richer information about the execution path. This includes information about control and data flow [44].

Let us illustrate using some examples. Consider the scenario where we also log the values

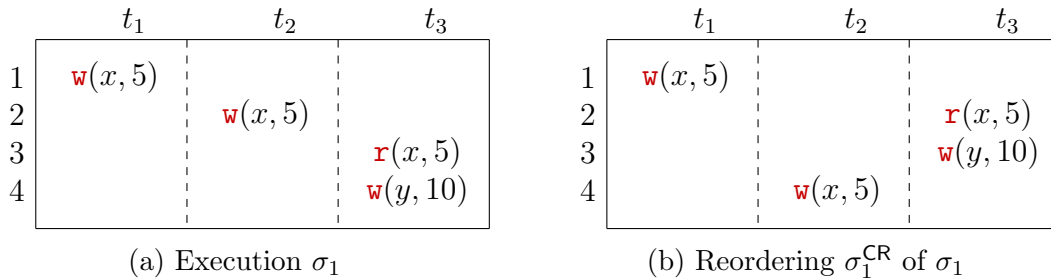


Figure 11.1: Predictive reasoning with data flow

read or written at memory access events. An example trace is σ_1 in Figure 11.1a. Events

e_1, e_2 and e_4 are write events that write the values 5, 5 and 10 to the variables x , x and y respectively and event e_3 reads the value 5 in the variable x . Here, $\text{lastWrite}_{\sigma_1}(e_3) = e_2$. However, the reordering σ_1^{CR} shown in Figure 11.1b is also a valid reordering — any program that generated σ_1 will also generate σ_1^{CR} because all the reads in σ_1^{CR} observe the same value as in σ_1 and thus observe the same control flow.

Next, consider the scenario where we also log a **branch** event whenever an execution takes a control flow branch. In this scenario, we can infer additional reorderings when a branch is not observed in some thread t following a read event performed by t . In particular, correct reorderings no longer need to preserve the last write (or even the value read) by such a read event.

Finally, a key observation we made in our prediction model is to abstract the implementations of mutual exclusion primitives as abstract objects called locks, instead of treating them as registers. Thus, instead of reasoning about low level implementations about sequences of read and write operations generated when acquiring and releasing locks, we could reason about the abstract semantics of lock objects, thereby allowing us to reason about more reorderings where the critical sections can be reordered. The same principle can be extended to other synchronization primitives such as those used in lock-free/wait-free algorithms [148, 149].

Beyond sequential consistency. A central assumption in our predictive algorithms is that executions are generated by a program running over a sequentially consistent memory [150]. Under this assumption, we could model execution traces as sequences of events. Further, under this assumption, the value returned by a read event is the same as that written to by the last write event accessing the same memory location. Modern hardware architectures and programming languages, however, do not offer such a strong guarantee — they run under weak memory semantics [151–153] In most cases, our model of prediction under this assumption is sound — any reordering inferred under the sequential consistency assumption is also a valid execution under a relaxed memory model. However, limiting predictive reasoning to this assumption can lead to missed bugs, which can otherwise only be observed under the assumption of weak/relaxed memory consistency. Developing predictive reasoning techniques under weaker memory semantics is an interesting future direction.

Predictive Monitoring. Often, software bugs may only be described using complex temporal patterns that describe intricate sequence of events that lead to a crash — ‘memory location x is freed twice without an intervening reallocation’ — or are symptomatic of security vulnerabilities — ‘resource R is accessed without prior authentication’ — or witness violations of desired invariants — ‘no consensus was reached in a leader election protocol

within first N rounds'. Heavyweight techniques for finding such bugs, such as static analyses or model checking, cannot be deployed in a large development environment — they either produce too many false alarms or do not scale. Lightweight formal methods like runtime monitoring, on the other hand, detect such temporal bugs at runtime and are ideal in an industry setting. As with traditional concurrency bugs like data races, one can naturally ask the predictive analogue to this question — can a single observed execution be used to infer violations of a temporal property in any of its reorderings? Predicting such temporal patterns from executions of concurrent programs or predictive monitoring can help directly expose deeper bugs such as security vulnerabilities in modern concurrent software, instead of inferring these vulnerabilities from the presence of classical concurrency bugs like data races. An interesting future direction is to develop the theoretical foundations and efficient algorithms for online predictive monitoring to employ them to dynamically predict errors in practical real world software.

REFERENCES

- [1] “Crash in Chromium Browser due to Data Race.” <https://bugs.chromium.org/p/chromium/issues/detail?id=15577>.
- [2] D. Vyukov, “ThreadSanitizer: Slaughtering Data Races, Google Testing Blog.”
- [3] S. 2004, “Software Bug Contributed to Blackout,” <https://www.securityfocus.com/news/8016>.
- [4] “PCWorld 2012 : NASDAQ’s facebook glitch came from race conditions.”
- [5] N. G. Leveson and C. S. Turner, “An investigation of the therac-25 accidents,” *Computer*, vol. 26, no. 7, pp. 18–41, July 1993.
- [6] M. Naik, A. Aiken, and J. Whaley, “Effective static race detection for java,” pp. 308–319, 2006. [Online]. Available: <http://doi.acm.org/10.1145/1133981.1134018>
- [7] D. Engler and K. Ashcraft, “RacerX: Effective, Static Detection of Race Conditions and Deadlocks,” *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 237–252, Oct. 2003.
- [8] P. Pratikakis, J. S. Foster, and M. Hicks, “LOCKSMITH: Practical Static Race Detection for C,” *ACM Trans. Program. Lang. Syst.*, vol. 33, no. 1, pp. 3:1–3:55, Jan. 2011.
- [9] J. W. Voun, R. Jhala, and S. Lerner, “RELAY: Static Race Detection on Millions of Lines of Code,” in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE ’07. New York, NY, USA: ACM, 2007, pp. 205–214.
- [10] S. Zhan and J. Huang, “ECHO: Instantaneous in Situ Race Detection in the IDE,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 775–786.
- [11] C. Radoi and D. Dig, “Practical Static Race Detection for Java Parallel Loops,” in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA 2013. New York, NY, USA: ACM, 2013, pp. 178–190.
- [12] S. Blackshear, N. Gorogiannis, P. W. O’Hearn, and I. Sergey, “Racerd: compositional static race detection,” *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–28, 2018.

- [13] D. Kini, U. Mathur, and M. Viswanathan, “Dynamic race prediction in linear time,” in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: ACM, 2017. [Online]. Available: <http://doi.acm.org/10.1145/3062341.3062374> pp. 157–170.
- [14] U. Mathur, D. Kini, and M. Viswanathan, “What happens-after the first race? enhancing the predictive power of happens-before based dynamic race detection,” *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 145:1–145:29, Oct. 2018. [Online]. Available: <http://doi.acm.org/10.1145/3276515>
- [15] D. Kini, U. Mathur, and M. Viswanathan, “Data race detection on compressed traces,” in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: ACM, 2018. [Online]. Available: <http://doi.acm.org/10.1145/3236024.3236025> pp. 26–37.
- [16] U. Mathur and M. Viswanathan, “Atomicity checking in linear time using vector clocks,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: <https://doi.org/10.1145/3373376.3378475> p. 183199.
- [17] U. Mathur, A. Pavlogiannis, and M. Viswanathan, “The complexity of dynamic data race prediction,” in *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, ser. LICS 20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: <https://doi.org/10.1145/3373718.3394783> p. 713727.
- [18] U. Mathur, A. Pavlogiannis, and M. Viswanathan, “Optimal prediction of synchronization-preserving races,” *Proc. ACM Program. Lang.*, vol. 5, no. POPL, Jan. 2021. [Online]. Available: <https://doi.org/10.1145/3434317>
- [19] A. Gascón, A. Tiwari, B. Carmer, and U. Mathur, “Look for the proof to find the program: Decorated-component-based program synthesis,” in *CAV 2017*.
- [20] M. S. Bauer, U. Mathur, R. Chadha, A. P. Sistla, and M. Viswanathan, “Exact quantitative probabilistic model checking through rational search,” ser. FMCAD ’17.
- [21] C. Fan, U. Mathur, S. Mitra, and M. Viswanathan, “Controller Synthesis Made Real: Reach-avoid Specifications and Linear Dynamics,” Tech. Rep.
- [22] P. Madhusudan, U. Mathur, S. Saha, and M. Viswanathan, “A Decidable Fragment of Second Order Logic With Applications to Synthesis,” ser. CSL 2018.
- [23] U. Mathur, P. Madhusudan, and M. Viswanathan, “Decidable verification of uninterpreted programs,” vol. 3, no. POPL, Jan. 2019. [Online]. Available: <https://doi.org/10.1145/3290359>

- [24] U. Mathur, A. Murali, P. Krogmeier, P. Madhusudan, and M. Viswanathan, “Deciding memory safety for single-pass heap-manipulating programs,” *Proc. ACM Program. Lang.*, vol. 4, no. POPL, Dec. 2019. [Online]. Available: <https://doi.org/10.1145/3371103>
- [25] U. Mathur, P. Madhusudan, and M. Viswanathan, “What’s decidable about program verification modulo axioms?” in *Tools and Algorithms for the Construction and Analysis of Systems*, A. Biere and D. Parker, Eds. Cham: Springer International Publishing, 2020, pp. 158–177.
- [26] P. Krogmeier, U. Mathur, A. Murali, P. Madhusudan, and M. Viswanathan, “Decidable synthesis of programs with uninterpreted functions,” in *Computer Aided Verification*, S. K. Lahiri and C. Wang, Eds. Cham: Springer International Publishing, 2020, pp. 634–657.
- [27] U. Mathur, M. S. Bauer, R. Chadha, A. P. Sistla, and M. Viswanathan, “Exact quantitative probabilistic model checking through rational search,” *Formal Methods in System Design*, Jul 2020. [Online]. Available: <https://doi.org/10.1007/s10703-020-00348-y>
- [28] V. Murali, E. Yao, U. Mathur, and S. Chandra, “Scalable statistical root cause analysis on app telemetry,” 2020.
- [29] R. Kulkarni, U. Mathur, and A. Pavlogiannis, “Dynamic Data-Race Detection through the Fine-Grained Lens.”
- [30] M. Zhang, U. Mathur, and M. Viswanathan, “Checking LTL[F,G,X] on Compressed Traces in Polynomial Time.”
- [31] M. P. Herlihy and J. M. Wing, “Linearizability: A correctness condition for concurrent objects,” *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, p. 463492, July 1990. [Online]. Available: <https://doi.org/10.1145/78969.78972>
- [32] M. P. Herlihy and J. M. Wing, “Axioms for concurrent objects,” in *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL ’87. New York, NY, USA: Association for Computing Machinery, 1987. [Online]. Available: <https://doi.org/10.1145/41625.41627> p. 1326.
- [33] T. F. Serbanuta, F. Chen, and G. Rosu, “Maximal causal models for sequentially consistent systems,” in *Proceedings of the third International Conference on Runtime Verification (RV’12)*, ser. LNCS, vol. 7687. Springer, September 2012, pp. 136–150.
- [34] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan, “Sound predictive race detection in polynomial time,” pp. 387–400, 2012. [Online]. Available: <http://doi.acm.org/10.1145/2103656.2103702>
- [35] A. Pavlogiannis, “Fast, sound, and effectively complete dynamic race prediction,” *Proc. ACM Program. Lang.*, vol. 4, no. POPL, Dec. 2019. [Online]. Available: <https://doi.org/10.1145/3371085>

- [36] J. Roemer, K. Genç, and M. D. Bond, “High-coverage, unbounded sound predictive race detection,” in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: ACM, 2018. [Online]. Available: <http://doi.acm.org/10.1145/3192366.3192385> pp. 374–389.
- [37] K. Genç, J. Roemer, Y. Xu, and M. D. Bond, “Dependence-aware, unbounded sound predictive race detection,” *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: <https://doi.org/10.1145/3360605>
- [38] C. Flanagan, S. N. Freund, and J. Yi, “Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs,” in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’08. ACM, 2008. [Online]. Available: <http://doi.acm.org/10.1145/1375581.1375618> pp. 293–303.
- [39] A. Farzan and P. Madhusudan, “Monitoring atomicity in concurrent programs,” in *Computer Aided Verification*, A. Gupta and S. Malik, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 52–65.
- [40] C. Papadimitriou, *The Theory of Database Concurrency Control*. USA: Computer Science Press, Inc., 1986.
- [41] A. Mazurkiewicz, “Trace theory,” in *Petri Nets: Applications and Relationships to Other Models of Concurrency*, W. Brauer, W. Reisig, and G. Rozenberg, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 278–324.
- [42] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, “Eraser: A Dynamic Data Race Detector for Multi-threaded Programs,” *SIGOPS Oper. Syst. Rev.*, vol. 31, no. 5, pp. 27–37, Oct. 1997.
- [43] M. Said, C. Wang, Z. Yang, and K. Sakallah, “Generating Data Race Witnesses by an SMT-based Analysis,” ser. NFM’11.
- [44] J. Huang, P. O. Meredith, and G. Rosu, “Maximal sound predictive race detection with control flow abstraction,” ser. PLDI ’14.
- [45] R. O’Callahan and J.-D. Choi, “Hybrid dynamic data race detection,” pp. 167–178, 2003. [Online]. Available: <http://doi.acm.org/10.1145/781498.781528>
- [46] A. Sen and O. Kalaci, *Hybrid Dynamic Data Race Detection in SystemC*, 2016.
- [47] L. Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System,” *Commun. ACM*, vol. 21, no. 7, pp. 558–565, July 1978.
- [48] E. Pozniarsky and A. Schuster, “Efficient on-the-fly data race detection in multithreaded c++ programs,” pp. 179–190, 2003. [Online]. Available: <http://doi.acm.org/10.1145/781498.781529>

- [49] A. Itzkovitz, A. Schuster, and O. Zeev-Ben-Mordehai, “Toward integration of data race detection in dsm systems,” *J. Parallel Distrib. Comput.*, vol. 59, no. 2, p. 180203, Nov. 1999. [Online]. Available: <https://doi.org/10.1006/jpdc.1999.1574>
- [50] F. Mattern, “Virtual Time and Global States of Distributed Systems,” in *Parallel and Distributed Algorithms*. North-Holland, 1988, pp. 215–226.
- [51] J. FIDGE, “Timestamps in message-passing systems that preserve the partial ordering,” in *Proc. 11th Australian Comput. Science Conf.*, 1988, pp. 56–66.
- [52] C. Flanagan and S. N. Freund, “Fasttrack: Efficient and precise dynamic race detection,” pp. 121–133, 2009. [Online]. Available: <http://doi.acm.org/10.1145/1542476.1542490>
- [53] K. Serebryany and T. Iskhodzhanov, “ThreadSanitizer: Data Race Detection in Practice,” ser. WBIA ’09, 2009.
- [54] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov, “Dynamic race detection with llvm compiler,” in *Runtime Verification*, S. Khurshid and K. Sen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 110–114.
- [55] A. Muehlenfeld and F. Wotawa, “Fault detection in multi-threaded c++ server applications,” in *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP ’07. New York, NY, USA: ACM, 2007. [Online]. Available: <http://doi.acm.org/10.1145/1229428.1229457> pp. 142–143.
- [56] <https://software.intel.com/content/www/us/en/develop/tools/inspector.html>.
- [57] T. Elmas, S. Qadeer, and S. Tasiran, “Goldilocks: A race and transaction-aware java runtime,” in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’07. New York, NY, USA: ACM, 2007. [Online]. Available: <http://doi.acm.org/10.1145/1250734.1250762> pp. 245–255.
- [58] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, “Chaff: Engineering an efficient sat solver,” in *Proceedings of the 38th Annual Design Automation Conference*, ser. DAC ’01. New York, NY, USA: Association for Computing Machinery, 2001. [Online]. Available: <https://doi.org/10.1145/378239.379017> p. 530535.
- [59] L. de Moura and N. Bjørner, “Z3: An efficient smt solver,” in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.
- [60] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression,” *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.
- [61] S. Biswas, J. Huang, A. Sengupta, and M. D. Bond, “Doublechecker: Efficient sound and precise atomicity checking,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 28–39.

- [62] B. P. Wood, M. Cao, M. D. Bond, and D. Grossman, “Instrumentation bias for dynamic data race detection,” *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017. [Online]. Available: <https://doi.org/10.1145/3133893>
- [63] J. Devietti, B. Wood, K. Strauss, L. Ceze, D. Grossman, and S. Qadeer, “RADISH: Always-on Sound and Complete Race Detection in Software and Hardware,” in *Proceedings of the Annual International Symposium on Computer Architecture*, 2012, pp. 201–212.
- [64] S. P. Reiss and M. Renieris, “Encoding program executions,” in *Proceedings of the 23rd International Conference on Software Engineering*, ser. ICSE ’01. USA: IEEE Computer Society, 2001, p. 221230.
- [65] S. Kaplan, Y. Smaragdakis, and P. Wilson, “Flexible reference trace reduction for VM simulations,” *ACM Transactions on Modeling and Computer Simulation*, vol. 13, no. 1, pp. 1–38, 2003.
- [66] A. Milenković and M. Milenković, “An efficient single-pass trace compression technique utilizing instruction streams,” *ACM Transactions on Modeling and Computer Simulation*, vol. 17, no. 1, 2007.
- [67] M. Eslamimehr and J. Palsberg, “Sherlock: Scalable deadlock detection for concurrent programs,” ser. FSE 2014.
- [68] H. Galperin and A. Wigderson, “Succinct Representations of Graphs,” *Information and Control*, vol. 56, no. 3, pp. 183–198, 1983.
- [69] C. Papadimitriou and M. Yannakakis, “A note on succinct representations of graphs,” *Information and Control*, vol. 71, no. 3, pp. 181–185, 1986.
- [70] A. Lozano and J. Balcázar, “The complexity of graph problems for succinctly represented graphs,” in *Proceedings of the International Workshop on Graph-Theoretic Concepts in Computer Science*, 1986, pp. 277–286.
- [71] J. Balcázar, “The complexity of searching implicit graphs,” *Artificial Intelligence*, vol. 86, no. 1, pp. 171–188, 1996.
- [72] H. Veith, “Succinct Representation, Leaf Languages, and Projection Reductions,” in *Proceedings of the IEEE Conference on Computational Complexity*, 1996, pp. 118–126.
- [73] J. Feigenbaum, S. Kannan, M. Vardi, and M. Viswanathan, “Complexity of Problems on Graphs Represented as OBDDs,” in *Proceedings of the Annual Symposium on Theoretical Aspects of Computer Science*, 1998, pp. 216–226.
- [74] B. Das, P. Scharpfenecker, and J. Torán, “Succinct encodings of graph isomorphism,” in *Proceedings of the International Conference on Languages and Automata Theory and Applications*, 2014, pp. 285–296.

- [75] C. Nevill-Manning and I. Witten, “Identifying hierarchical structure in sequences: A linear time algorithm,” *Journal of Artificial Intelligence*, vol. 7, pp. 67–82, 1997.
- [76] C. Nevill-Manning, “Inferring sequential structure,” Ph.D. dissertation, University of Waikato, 1996.
- [77] “Sequitur: Inferring Hierarchies From Sequences,” 2017, <http://www.sequitur.info/>, accessed: 2017-08-01.
- [78] J. R. Larus, “Whole program paths,” in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, ser. PLDI '99. New York, NY, USA: Association for Computing Machinery, 1999. [Online]. Available: <https://doi.org/10.1145/301618.301678> p. 259269.
- [79] U. Mathur, “ZipTrack,” 2020, accessed: 2020-10-25. [Online]. Available: <https://github.com/umangm/ziptrack>
- [80] E.-H. Yang and J. C. Kieffer, “Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform. i. without context models,” *IEEE Transactions on Information Theory*, vol. 46, no. 3, pp. 755–777, 2000.
- [81] T. Welch, “A technique for high-performance data compression,” *Computer*, vol. 17, no. 6, pp. 8–19, 1984.
- [82] J. Kieffer, E.-H. Yang, G. Nelson, and P. Cosman, “Universal lossless compression via multilevel pattern matching,” *IEEE Transactions on Information Theory*, vol. 46, no. 4, pp. 1227–1245, 2000.
- [83] J. Kieffer and E.-H. Yang, “Grammar-based codes: a new class of universal lossless source codes,” *IEEE Transactions on Information Theory*, vol. 46, no. 3, pp. 737–754, 2000.
- [84] N. Larsson and A. Moffat, “Off-line dictionary-based compression,” *Proceedings of the IEEE*, vol. 88, no. 11, pp. 1722–1732, 2000.
- [85] G. Rosu, “RV-Predict, Runtime Verification.”
- [86] U. Mathur, “RAPID,” 2020, accessed: 2020-11-01. [Online]. Available: <https://github.com/umangm/rapid>
- [87] E. Farchi, Y. Nir, and S. Ur, “Concurrent Bug Patterns and How to Test Them,” in *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 286.2–.
- [88] L. A. Smith and J. M. Bull, “A multithreaded java grande benchmark suite,” in *Proceedings of the third workshop on Java for high performance computing*, 2001.

- [89] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, “The dacapo benchmarks: Java benchmarking development and analysis,” pp. 169–190, 2006. [Online]. Available: <http://doi.acm.org/10.1145/1167473.1167488>
- [90] C. Sadowski and J. Yi, “How developers use data race detection tools,” in *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*, ser. PLATEAU ’14. New York, NY, USA: ACM, 2014. [Online]. Available: <http://doi.acm.org/10.1145/2688204.2688205> pp. 43–51.
- [91] D. Kini, U. Mathur, and M. Viswanathan, “Dynamic Race Prediction in Linear Time,” *CoRR*, vol. abs/1704.02432, 2017. [Online]. Available: <http://arxiv.org/abs/1704.02432>
- [92] J. Huang and A. K. Rajagopalan, “Precise and maximal race detection from incomplete traces,” pp. 462–476, 2016. [Online]. Available: <http://doi.acm.org/10.1145/2983990.2984024>
- [93] J. Roemer, K. Genç, and M. D. Bond, “Smarttrack: Efficient predictive race detection,” in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: <https://doi.org/10.1145/3385412.3385993> p. 747762.
- [94] U. Mathur, A. Pavlogiannis, and M. Viswanathan, “Optimal prediction of synchronization-preserving races,” *CoRR*, vol. abs/2010.16385, 2020. [Online]. Available: <https://arxiv.org/abs/2010.16385>
- [95] A. C.-C. Yao, “Some complexity questions related to distributive computing(preliminary report),” in *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing*, ser. STOC 79. New York, NY, USA: Association for Computing Machinery, 1979. [Online]. Available: <https://doi.org/10.1145/800135.804414> p. 209213.
- [96] M. Yu, J.-S. Lee, and D.-H. Bae, “Adaptivelock: Efficient hybrid data race detection based on real-world locking patterns,” *International Journal of Parallel Programming*, Jun 2018. [Online]. Available: <https://doi.org/10.1007/s10766-018-0579-5>
- [97] H. Do, S. G. Elbaum, and G. Rothermel, “Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact.” *Empirical Software Engineering: An International Journal*, vol. 10, no. 4, pp. 405–435, 2005.
- [98] R. H. Netzer and B. P. Miller, “On the complexity of event ordering for shared-memory parallel program executions,” in *In Proceedings of the 1990 International Conference on Parallel Processing*, 1990, pp. 93–97.

- [99] R. H. B. Netzer and B. P. Miller, “What are race conditions? some issues and formalizations,” *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 1, p. 7488, Mar. 1992. [Online]. Available: <https://doi.org/10.1145/130616.130623>
- [100] R. N. Netzer and B. P. Miller, “Detecting data races in parallel program executions,” in *In Advances in Languages and Compilers for Parallel Computing, 1990 Workshop*. MIT Press, 1989, pp. 109–129.
- [101] P. B. Gibbons and E. Korach, “Testing shared memories,” *SIAM J. Comput.*, vol. 26, no. 4, pp. 1208–1244, Aug. 1997. [Online]. Available: <http://dx.doi.org/10.1137/S0097539794279614>
- [102] K. Sen, G. Roşu, and G. Agha, “Detecting Errors in Multithreaded Programs by Generalized Predictive Analysis of Executions,” in *Proceedings of the 7th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems*, ser. FMOODS’05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 211–226.
- [103] F. Chen and G. Roşu, “Parametric and sliced causality,” in *Proceedings of the 19th International Conference on Computer Aided Verification*, ser. CAV’07. Berlin, Heidelberg: Springer-Verlag, 2007. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1770351.1770387> pp. 240–253.
- [104] C. Wang, S. Kundu, M. Ganai, and A. Gupta, “Symbolic predictive analysis for concurrent programs,” in *Proceedings of the 2Nd World Congress on Formal Methods*, ser. FM ’09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 256–272.
- [105] M. Chalupa, K. Chatterjee, A. Pavlogiannis, K. Vaidya, and N. Sinha, “Data-centric dynamic partial order reduction,” in *Proceedings of the 45rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’18, 2018.
- [106] I. J. Aalbersberg, “Theory of traces,” *Theor. Comput. Sci.*, vol. 60, no. 1, p. 182, Sep. 1988. [Online]. Available: [https://doi.org/10.1016/0304-3975\(88\)90051-5](https://doi.org/10.1016/0304-3975(88)90051-5)
- [107] A. Bertoni, G. Mauri, and N. Sabadini, “Membership problems for regular and context-free trace languages,” *Inf. Comput.*, vol. 82, no. 2, p. 135150, Aug. 1989. [Online]. Available: [https://doi.org/10.1016/0890-5401\(89\)90051-5](https://doi.org/10.1016/0890-5401(89)90051-5)
- [108] C. H. Papadimitriou, “The serializability of concurrent database updates,” *J. ACM*, vol. 26, no. 4, p. 631653, Oct. 1979. [Online]. Available: <https://doi.org/10.1145/322154.322158>
- [109] S. Even, A. L. Selman, and Y. Yacobi, “The complexity of promise problems with applications to public-key cryptography,” *Inf. Control*, vol. 61, no. 2, p. 159173, May 1984. [Online]. Available: [https://doi.org/10.1016/S0019-9958\(84\)80056-X](https://doi.org/10.1016/S0019-9958(84)80056-X)

- [110] K. Bringmann, “Fine-Grained Complexity Theory (Tutorial),” in *36th International Symposium on Theoretical Aspects of Computer Science (STACS 2019)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), R. Niedermeier and C. Paul, Eds., vol. 126. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. [Online]. Available: <http://drops.dagstuhl.de/opus/volltexte/2019/10243> pp. 4:1–4:7.
- [111] P. A. Abdulla, M. F. Atig, B. Jonsson, M. Lång, T. P. Ngo, and K. Sagonas, “Optimal stateless model checking for reads-from equivalence under sequential consistency,” *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: <https://doi.org/10.1145/3360576>
- [112] U. Mathur, A. Pavlogiannis, and M. Viswanathan, “The complexity of dynamic data race prediction,” 2020.
- [113] R. G. Downey and M. R. Fellows, *Parameterized Complexity*, ser. Monographs in Computer Science. Springer, 1999. [Online]. Available: <https://doi.org/10.1007/978-1-4612-0515-9>
- [114] J. Chen, X. Huang, I. A. Kanj, and G. Xia, “Strong computational lower bounds via parameterized complexity,” *Journal of Computer and System Sciences*, vol. 72, no. 8, pp. 1346 – 1367, 2006. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0022000006000675>
- [115] R. Williams, “A new algorithm for optimal 2-constraint satisfaction and its implications,” *Theor. Comput. Sci.*, vol. 348, no. 2, pp. 357–365, Dec. 2005. [Online]. Available: <http://dx.doi.org/10.1016/j.tcs.2005.09.023>
- [116] A. V. Aho, M. R. Garey, and J. D. Ullman, “The transitive reduction of a directed graph,” *SIAM Journal on Computing*, vol. 1, no. 2, pp. 131–137, 1972.
- [117] S. Lu, S. Park, E. Seo, and Y. Zhou, “Learning from mistakes: A comprehensive study on real world concurrency bug characteristics,” in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII. New York, NY, USA: ACM, 2008. [Online]. Available: <http://doi.acm.org/10.1145/1346281.1346323> pp. 329–339.
- [118] C. Flanagan and S. N. Freund, “Atomizer: A dynamic atomicity checker for multithreaded programs,” in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’04. New York, NY, USA: Association for Computing Machinery, 2004. [Online]. Available: <https://doi.org/10.1145/964001.964023> pp. 256–267.

- [119] T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi, “Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems,” in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: <https://doi.org/10.1145/2872362.2872374> pp. 517–530.
- [120] J. Wang, W. Dou, Y. Gao, C. Gao, F. Qin, K. Yin, and J. Wei, “A comprehensive study on real world concurrency bugs in node.js,” in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. IEEE Press, 2017, pp. 520–531.
- [121] H. Liu, G. Li, J. F. Lukman, J. Li, S. Lu, H. S. Gunawi, and C. Tian, “Dcatch: Automatically detecting distributed concurrency bugs in cloud systems,” in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: <https://doi.org/10.1145/3037697.3037735> pp. 677–691.
- [122] P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy, “An empirical study on the correctness of formally verified distributed systems,” in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys ’17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: <https://doi.org/10.1145/3064176.3064183> pp. 328–343.
- [123] L. Chew and D. Lie, “Kivati: Fast detection and prevention of atomicity violations,” in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys ’10. New York, NY, USA: Association for Computing Machinery, 2010. [Online]. Available: <https://doi.org/10.1145/1755913.1755945> pp. 307–320.
- [124] C. Flanagan and S. Qadeer, “A type and effect system for atomicity,” in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, ser. PLDI ’03. New York, NY, USA: ACM, 2003. [Online]. Available: <http://doi.acm.org/10.1145/781131.781169> pp. 338–349.
- [125] C. Flanagan and S. Qadeer, “Types for atomicity,” in *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, ser. TLDI ’03. New York, NY, USA: Association for Computing Machinery, 2003. [Online]. Available: <https://doi.org/10.1145/604174.604176> pp. 1–12.
- [126] R. Agarwal and S. D. Stoller, “Type inference for parameterized race-free java,” in *Verification, Model Checking, and Abstract Interpretation*, B. Steffen and G. Levi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 149–160.

- [127] C. Flanagan, S. N. Freund, and S. Qadeer, “Exploiting purity for atomicity,” in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSSTA ’04. New York, NY, USA: Association for Computing Machinery, 2004. [Online]. Available: <https://doi.org/10.1145/1007512.1007543> pp. 221–231.
- [128] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller, “Automated type-based analysis of data races and atomicity,” in *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP ’05. New York, NY, USA: Association for Computing Machinery, 2005. [Online]. Available: <https://doi.org/10.1145/1065944.1065956> pp. 83–94.
- [129] L. Wang and S. D. Stoller, “Static analysis of atomicity for programs with non-blocking synchronization,” in *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP ’05. New York, NY, USA: ACM, 2005. [Online]. Available: <http://doi.acm.org/10.1145/1065944.1065953> pp. 61–71.
- [130] M. Xu, R. Bodík, and M. D. Hill, “A serializability violation detector for shared-memory server programs,” in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’05. New York, NY, USA: Association for Computing Machinery, 2005. [Online]. Available: <https://doi.org/10.1145/1065010.1065013> pp. 1–14.
- [131] S. Lu, J. Tucek, F. Qin, and Y. Zhou, “AVIO: detecting atomicity violations via access interleaving invariants,” in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, 2006. [Online]. Available: <https://doi.org/10.1145/1168857.1168864> pp. 37–48.
- [132] A. Farzan and P. Madhusudan, “Monitoring atomicity in concurrent programs,” in *Proceedings of the 20th International Conference on Computer Aided Verification*, ser. CAV ’08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 52–65.
- [133] U. Mathur and M. Viswanathan, “Atomicity checking in linear time using vector clocks,” *CoRR*, vol. abs/2001.04961, 2020. [Online]. Available: <https://arxiv.org/abs/2001.04961>
- [134] C. Flanagan and S. N. Freund, “The RoadRunner Dynamic Analysis Framework for Concurrent Programs,” in *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE ’10. New York, NY, USA: ACM, 2010, pp. 1–8.
- [135] D. Michail, J. Kinable, B. Naveh, and J. V. Sichi, “Jgrapht—a java library for graph data structures and algorithms,” *arXiv preprint arXiv:1904.08355*, 2019.
- [136] S. Biswas, “DoubleChecker,” 2014, accessed: 2020-01-15. [Online]. Available: <https://sourceforge.net/p/jikesrvm/research-archive/45/>

- [137] C. von Praun and T. R. Gross, “Static conflict analysis for multi-threaded object-oriented programs,” in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, ser. PLDI ’03. New York, NY, USA: ACM, 2003. [Online]. Available: <http://doi.acm.org/10.1145/781131.781145> pp. 115–128.
- [138] M. A. Bender, J. T. Fineman, S. Gilbert, and C. E. Leiserson, “On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs,” in *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, 2004, pp. 133–144.
- [139] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav, “Scalable and precise dynamic datarace detection for structured parallelism,” in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: <https://doi.org/10.1145/2254064.2254127> p. 531542.
- [140] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav, “Efficient data race detection for async-finish parallelism,” in *Proceedings of the First International Conference on Runtime Verification*, ser. RV’10. Berlin, Heidelberg: Springer-Verlag, 2010, p. 368383.
- [141] R. D. Blumofe and C. E. Leiserson, “Scheduling multithreaded computations by work stealing,” *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 720–748, 1999.
- [142] D. Dimitrov, M. Vechev, and V. Sarkar, “Race detection in two dimensions,” in *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA ’15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: <https://doi.org/10.1145/2755573.2755601> p. 101110.
- [143] K. Agrawal, J. Devietti, J. T. Fineman, I.-T. A. Lee, R. Utterback, and C. Xu, “Race detection and reachability in nearly series-parallel dags,” in *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2018, pp. 156–171.
- [144] C. Flanagan and S. N. Freund, “Redcard: Redundant check elimination for dynamic race detectors,” in *Proceedings of the 27th European Conference on Object-Oriented Programming*, ser. ECOOP’13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 255–280.
- [145] D. Rhodes, C. Flanagan, and S. N. Freund, “Bigfoot: Static check placement for dynamic race detection,” in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: ACM, 2017. [Online]. Available: <http://doi.acm.org/10.1145/3062341.3062350> pp. 141–156.

- [146] J. R. Wilcox, P. Finch, C. Flanagan, and S. N. Freund, “Array shadow state compression for precise dynamic race detection (t),” in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 155–165.
- [147] M. D. Bond, M. Kulkarni, M. Cao, M. Zhang, M. Fathi Salmi, S. Biswas, A. Sengupta, and J. Huang, “Octet: Capturing and controlling cross-thread dependences efficiently,” in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA ’13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: <https://doi.org/10.1145/2509136.2509519> p. 693712.
- [148] A. Kogan and E. Petrank, “A methodology for creating fast wait-free data structures,” in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP ’12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: <https://doi.org/10.1145/2145816.2145835> p. 141150.
- [149] M. Herlihy, V. Luchangco, and M. Moir, “Obstruction-free synchronization: Double-ended queues as an example,” in *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.* IEEE, 2003, pp. 522–529.
- [150] L. Lamport, “How to make a multiprocessor computer that correctly executes multiprocess programs,” *IEEE Trans. Comput.*, vol. 28, no. 9, p. 690691, Sep. 1979. [Online]. Available: <https://doi.org/10.1109/TC.1979.1675439>
- [151] S. V. Adve and K. Gharachorloo, “Shared memory consistency models: A tutorial,” *Computer*, vol. 29, no. 12, p. 6676, Dec. 1996. [Online]. Available: <https://doi.org/10.1109/2.546611>
- [152] J. Manson, W. Pugh, and S. V. Adve, “The java memory model,” in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’05. New York, NY, USA: Association for Computing Machinery, 2005. [Online]. Available: <https://doi.org/10.1145/1040305.1040336> p. 378391.
- [153] M. J. Batty, “The c11 and c++ 11 concurrency model,” Ph.D. dissertation, 2015.